

С. А. Немюгин

TURBO  
**PASCAL**

Санкт-Петербург  
Москва • Харьков • Минск  
2000



**ББК32.973.2-018.1я7**

**УДК 681.3.06(075)**

**H50**

**H50 Turbo Pascal / С. А. Немногин. — СПб: Издательство «Питер», 2000. — 496 с.: ил.  
ISBN 5-8046-0137-7**

**В учебнике дается систематическое изложение основ программирования в системе Турбо Паскаль. Рассматриваются вопросы системного программирования, программирование графики и звука. Даются введение в объектно-ориентированное программирование и методы вычислений. Книга содержит большое число исходных текстов программ с подробными комментариями.**

**Учебник адресован прежде всего студентам и школьникам старших классов.**

**ББК32.973.018.1я7**

**УДК 681.3.06(075)**

**ISBN 5-8046-0137-7**

**© с. А. Немногин, 2000**

**© Серия, оформление, Издательство «Питер», 2000**

# Краткое содержание

<b>Урок 1.</b>	Введение в Паскаль .....	12
<b>Урок 2.</b>	Программирование на языке Паскаль.....	52
<b>Урок 3.</b>	Типы данных в языке Паскаль.....	96
<b>Урок 4.</b>	Элементы системного программирования на языке Паскаль.....	148
<b>Урок 5.</b>	Основы программирования графики .....	196
<b>Урок 6.</b>	Графика VGA. программирование трехмерных и динамических изображений.....	254
<b>Урок 7.</b>	Рекурсия и рекурсивные алгоритмы.....	304
<b>Урок 8.</b>	Программирование звука .....	332
<b>Урок 9.</b>	Введение в объектно-ориентированное программирование .....	358
<b>Урок 10.</b>	Турбо Паскаль для вычислений.....	378
<b>Приложение А.</b>	Работа в интегрированной среде Турбо Паскаля .....	437
<b>Приложение Б.</b>	Список некоторых прерываний DOS и BIOS.....	443
<b>Приложение В.</b>	Ответы и решения .....	454
	Алфавитный указатель.....	475

# Содержание

Предисловие .....	9
<b>Урок 1. Введение в Паскаль.....</b>	<b>12</b>
Основные элементы языка Паскаль.....	13
Программа.....	16
Разделы описаний .....	18
Описание переменных.....	18
Типы Word, Real и Extended.....	19
Описание констант .....	19
Исполняемые операторы .....	20
Вызовы процедур .....	20
Составной оператор .....	20
Структурные операторы.....	21
Первые программы .....	28
Подпрограммы-функции .....	36
Ввод и вывод.....	42
Модули .....	46
Упражнения .....	48
Что нового мы узнали? .....	51
<b>Урок 2. Программирование на языке Паскаль.....</b>	<b>52</b>
Процедуры .....	53
Область видимости идентификаторов.....	55
Формальные и фактические параметры .....	57
Параметры-значения, параметры-переменные и нетипизированные параметры.....	57
Пример программы .....	58
Упражнения .....	61

Предопределенные типы переменных.....	62
Символьный тип .....	64
Управляющие символы .....	66
Ввод символов с клавиатуры.....	67
Строковый тип .....	71
Операции над строками .....	73
Массивы.....	76
Множества.....	80
Целый и логический (булев) типы.....	82
Целые типы .....	82
Логический (булев) тип.....	87
Скалярные типы .....	89
Вещественные типы.....	90
Старшинство операций .....	94
Что нового мы узнали? .....	95
<b>Урок 3. Типы данных в языке Паскаль .....</b>	<b>96</b>
Работа с типами данных.....	97
Типы, определяемые пользователем .....	97
Совместимость типов .....	98
Работа с файлами.....	100
Текстовые файлы.....	100
Типизированные файлы .....	112
Нетипизированные файлы .....	113
Записи .....	116
Записи с фиксированными частями .....	116
Записи с вариантами .....	121
Указатели.....	123
Связные списки .....	130
Модуль для вычислений с полиномами .....	135
Работа с памятью .....	144
Что нового мы узнали? .....	146
<b>Урок 4. Элементы системного программирования на языке Паскаль.....</b>	<b>148</b>
Операционная система MS-DOS — краткий курс .....	149
Память.....	152
Среда.....	154
Прерывания .....	155

Модуль DOS .....	156
Работа с файловой системой .....	156
Программирование для MS-DOS и BIOS.....	162
Прерывания .....	166
Прерывания BIOS .....	169
Мышь.....	183
Пример использования модуля mouse.....	192
Другие устройства.....	194
Что нового мы узнали? .....	195

## **Урок 5. Основы программирования графики 196**

Графика.....	197
Текстовый и графический режимы .....	197
Графические координаты.....	198
Переключение между текстовым и графическим режимами ....	200
Примеры программ .....	202
Модуль Graph .....	208
Программа «Игла Бюффона» (бросаем иголки по-научному).....	222
Программа «Жизнь.....	226
Принципы программирования графики.....	231
Инициализация графического режима. Пиксели .....	232
Отрезки прямых .....	234
Отсечение линий .....	242
Окружность.....	247
Использование модуля mouse для программирования мыши в графическом режиме.....	250
Что нового мы узнали? .....	253

## **Урок 6. Графика VGA, программирование трехмерных и динамических изображений ..... 254**

Технические подробности.....	255
Графика VGA.....	258
Трехмерная графика .....	265
Векторы и операции над векторами .....	266
Векторные преобразования .....	268
Перспективные изображения и проекции .....	273
Модуль graphs3d.....	276
Построение непрозрачных объектов .....	285

Программирование динамических изображений .....	290
Спрайты.....	298
Что нового мы узнали? .....	303
<b>Урок 7. Рекурсия и рекурсивные алгоритмы.....</b>	<b>304</b>
Рекурсия .....	305
Примеры программ с использованием рекурсии .....	313
Разностные уравнения.....	314
Перебор с возвратами .....	316
Рекурсивные графические алгоритмы .....	320
Комбинаторные вычисления.....	326
Что нового мы узнали? .....	330
<b>Урок 8. Программирование звука.....</b>	<b>332</b>
Использование встроенного динамика.....	333
Программа «Виртуальное пианино».....	337
Программирование SoundBlaster.....	341
Звук и его свойства.....	341
Звуковая карта .....	344
Программирование звуковой карты .....	345
Другие возможности.....	356
Что нового мы узнали? .....	357
<b>Урок 9. Введение в объектно-ориентированное программирование. .....</b>	<b>358</b>
Что такое объектно-ориентированное программирование .....	359
Объекты.....	360
Инкапсуляция .....	362
Наследование.....	362
Виртуальные методы .....	363
Динамическое создание объектов.....	365
Полиморфизм .....	366
Модуль matrices.....	366
Что нового мы узнали? .....	377
<b>Урок 10. Турбо Паскаль для вычислений.....</b>	<b>378</b>
Вычисления, связанные с теорией чисел .....	379
Простые числа .....	381

Вычисления с полиномами.....	385
Линейная алгебра .....	391
Симметричные матрицы и проблема собственных значений .....	397
Решение нелинейных уравнений .....	405
Вычисление интегралов.....	409
Решение дифференциальных уравнений.....	422
Методы Рунге—Кутта.....	422
Что нового мы узнали? .....	436
<b>Приложение А. Работа в интегрированной среде Турбо Паскаля .....</b>	<b>437</b>
<b>Приложение Б. Список некоторых прерываний DOS И BIOS .....</b>	<b>443</b>
<b>Приложение В. Ответы и решения .....</b>	<b>454</b>
<b>Алфавитный указатель.....</b>	<b>475</b>

# Предисловие

Цель данного учебного курса — обучить программированию на Паскале, точнее, на том его расширении, которое было создано фирмой Borland (ныне Inprise) и называется Турбо Паскаль. Предполагается, что читатель этой книги уже имеет определенную подготовку в области информатики и программирования. Это требование, правда, не является совсем уж обязательным, и я, по возможности, старался дать объяснение основным, а также более специальным терминам и понятиям из области программирования. Такие отступления, по необходимости максимально короткие, встречаются в основном в первых главах (уроках), содержащих вводный материал по программированию на Турбо Паскале.

Для успешной работы с книгой желателен, кроме того, определенный уровень математической подготовки (а в уроке, посвященном программированию звука, даже музыкальной!). Это связано с тем, что в книге используются примеры, для понимания которых надо владеть математикой, во всяком случае, на уровне старших классов средней школы. Последний урок в этом смысле самый сложный, так как он посвящен решению сложных вычислительных задач. Здесь уже пригодится знание высшей математики. Уровень — младшие курсы вуза. Суммируя, обозначу потенциального читателя — это школьники старших классов средних и специализированных школ, а также студенты. Ну и, конечно, все остальные!

Особенность данной книги заключается в том, что в ней содержится много конкретных примеров, иллюстрирующих теоретический материал. Если у вас, читатель, не хватает терпения читать стандартные учебники по программированию с их обстоятельным, педантичным подходом и большим количеством абстрактных примеров, возможно, эта книга для вас.

Вторичная цель книги — дать примеры работающих программ, предназначенных для решения конкретных задач, достаточно часто встречающихся в повседнев-

ной жизни программиста, вычислителя, научного работника, инженера, школьника и студента.

Паскаль — замечательный язык программирования, который относительно прост в изучении, довольно ясен и логичен и, будучи первым изучаемым языком программирования, приучает к хорошему стилю. Паскаль воспитывает дисциплину структурного программирования и программирования вообще лучше, чем другие языки программирования, такие, как, например, БЕЙСИК.

Паскаль — гибкий и развитый в отношении типов данных язык. Привлекательны его рекурсивные возможности, а также поддержка технологии объектно-ориентированного программирования.

Несколько слов об истории языка. Алгоритмический язык АЛГОЛ был разработан в 1950-х-60-х годах. Паскаль стал «наследником» АЛГОЛА. Его разработчиком был швейцарский ученый Никлаус Вирт, собиравшийся использовать этот язык для обучения своих студентов методам разработки компиляторов. Время рождения языка Паскаль - начало 70-х годов. По сравнению с АЛГОЛОМ, Паскаль проще и яснее. У него намного лучшие возможности обработки данных и имеются встроенные процедуры ввода/вывода, которых не было в АЛГОЛЕ. Турбо Паскаль фирмы Borland является расширением стандарта языка и содержит, кроме того, интегрированную среду, намного ускоряющую и облегчающую процесс разработки программ. Этот программный продукт прошел через 6 версий, прежде чем появился Турбо Паскаль 7.0. Этой версии я и придерживаюсь в данной книге.

Турбо Паскаль, в свою очередь, положил начало новой линии продуктов фирмы Borland — Delphi, системе быстрой разработки приложений для Windows. Используемый в Delphi язык программирования Object Pascal сохранил основные черты Турбо Паскаля, обогатившись новыми возможностями. Имеются и другие реализации языка Паскаль, в том числе предназначенные для работы не на персональных компьютерах (как Турбо Паскаль), а на других компьютерных платформах.

Обучение языку программирования проходит намного более эффективно с изучением примеров и выполнением упражнений. Ученик должен видеть примеры готовых программ, а также иметь возможность решать задачи. Все это есть в данной книге. Примеры и упражнения составляют, вероятно, большую ее часть.

Изучение программирования на языке Паскаль может дать хороший старт в огромный и увлекательный мир программирования. Автор надеется, что предлагаемая читателю книга поможет ему в этом.

И, наконец, самое приятное. Автор хотел бы поблагодарить заведующую компьютерной редакцией издательского дома «Питер» Екатерину Строганову за поддержку данного издания и терпение, а также редактора, критика которого помогла существенно улучшить первоначальный вариант книги.

Автор также будет рад получить отзывы и замечания по адресу

[nemnugin@mph.phys.spbu.ru](mailto:nemnugin@mph.phys.spbu.ru)

## От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты [compr@piter-press.ru](mailto:compr@piter-press.ru) (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение.

Подробную информацию о наших книгах вы найдете на Web-сайте нашего изда-  
тельства <http://www.piter-press.ru>.



## **1** УРОК

# Введение в Паскаль

- 
- Основные элементы языка Паскаль
  - Первые программы
  - Подпрограммы-функции
  - Ввод и вывод
  - Модули
  - Упражнения
-

**В** этом уроке мы познакомимся с основными элементами языка Паскаль. Вначале дается перечень символов и зарезервированных слов языка, рассматривается структура программы. Даётся описание основных операторов языка. Вводятся подпрограммы-функции и библиотечные модули. Даются начальные сведения по программированию ввода/вывода данных и работе с внешними файлами. Завершается урок упражнениями. В данном уроке и далее в книге мы будем сопровождать описание конструкций языка и приемов программирования примерами программ — как относительно простых, так и более сложных.

## Основные элементы языка Паскаль

Любой естественный язык строится из элементарных составляющих — букв, образующих алфавит языка. Буквы используются для построения слов, слова складываются в предложения, а предложения... Из предложений состоит любой текст — письмо, роман, секретное донесение. Всякий язык программирования организован примерно так же. Имеется алфавит языка, то есть набор символов, которые можно использовать в программе. Существуют *зарезервированные слова*, имеющие вполне определенный смысл и определенное назначение. Их нельзя изменять: любая неточность в написании таких слов является серьезной ошибкой. В отличие от естественных языков человеческого общения, в языках программирования можно вводить свои собственные слова и придавать этим словам свой собственный смысл. Небольшую программу можно уподобить письму или маленькому рассказу. Большой проект — это роман. Как и обычное письмо, программа может быть написана хорошим или плохим «слогом» (стилем), и чем лучше стиль, тем понятнее программа, тем меньше вероятность появления в ней ошибок.

Напомним вначале некоторые основные понятия программирования. Опытный программист уже знает, а новичку полезно узнать, что главными элементами любой программы являются переменные, константы и операторы. *Переменная* — это ячейка (или несколько ячеек) оперативной памяти компьютера. Такой ячейке присвоено определенное имя, ее содержимое может изменяться в ходе выполнения программы. Вид информации, содержащейся в

ячейке, набор преобразований, которые можно выполнять над этой информацией, и множество допустимых значений определяются типом переменной. *Константа* отличается от переменной тем, что ее значение фиксировано и не может быть изменено в ходе выполнения программы. *Операторы* задают те или иные действия, которые должна выполнять программа.

Язык Турбо Паскаль состоит приблизительно из 80 зарезервированных слов и специальных символов. Алфавит языка составляют буквы латинского алфавита, цифры, а также специальные символы, такие, например, как +, -, \_. Специальными символами языка являются и некоторые пары символов. Как уже отмечалось, зарезервированные слова в языке Паскаль могут применяться только по своему прямому назначению, то есть в качестве имен операторов, названий операций и т. д. В табл. 1.1 приведен алфавитный список зарезервированных слов. В большинстве случаев овладение даже небольшой частью этого «словаря» достаточно для начала успешной работы по программированию на Паскале.

**Таблица 1.1.** Зарезервированные слова языка Паскаль

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

В дальнейшем мы разберем применение приведенных в данной таблице зарезервированных слов, а сейчас лишь кратко поясним смысл некоторых из них, наиболее часто используемых в программах на Паскале. Заголовки, то есть первые операторы программ и библиотечных модулей — это *program* и *unit*. Для описания переменных, констант и составных частей программы — подпрограмм-процедур и подпрограмм-функций — используются зарезервированные слова *var*, *const*, *procedure*, *function*. Операторы описания типов переменных, задаваемых пользователем, — это *type*, *array*, *string*, *record...end*, *file of...*. Слова, используемые для программирования составных операторов, а также

начинающие и оканчивающие последовательность исполняемых операторов программы, — `begin` и `end`. Операторами, управляющими ходом выполнения программы (они так и называются — *управляющие операторы*), являются `if...then...else`, `for...to...do`, `repeat...until`, `case...of...end`, `for...downto...do`, `while...do`. В библиотечных модулях используются зарезервированные слова `implementation`, `interface`. Зарезервированные слова для обозначения арифметических и логических операций — `div`, `mod`, `shl`, `shr`, `and`, `or`, `not` и некоторые другие. В программах, написанных с использованием методов объектно-ориентированного программирования, применяются зарезервированные слова `object`, `constructor`, `destructor`, `public` и `virtual`.

Как уже было отмечено, кроме зарезервированных слов в программах на языке Паскаль используются как отдельные специальные символы, так и пары символов, которые имеют специальное значение. Перечень таких символов приведен в табл. 1.2.

**Таблица 1.2.** Одиночные и двойные специальные символы языка Паскаль

<code>:=</code>	Присваивание переменной (слева от символа) значения выражения (справа от символа)
<code>:</code>	Разделитель операторов в программе
<code>( )</code>	Скобки для арифметических и логических выражений
<code>:</code>	Разделитель в описаниях переменных и формате операторов вывода
<code>..</code>	Многоточие для списков
<code>+</code>	Бинарные операции (не только арифметические!)
<code>-</code>	
<code>*</code>	
<code>/</code>	
<code>=</code>	Логическое равенство, элемент описания констант и типов
<code>&lt;&gt;</code>	Логическое неравенство
<code>&lt; &gt;</code>	Отношения «меньше чем» и «больше чем»
<code>&lt;=</code>	Отношения «меньше или равно» и «больше или равно»
<code>&gt;=</code>	
<code>.</code>	Конец программы или модуля, а также десятичная точка в константах вещественного типа
<code>,</code>	Ограничители константы строкового типа
<code>{}</code>	Пары скобок для комментариев
<code>(* *)</code>	
<code>,</code>	Разделитель элементов списка
<code>[ ]</code>	Скобки для ссылки на элемент массива или указания диапазона значений индекса

Одних только зарезервированных слов и специальных символов недостаточно для написания полноценной программы, ведь в нее надо вводить данные, а результат ее работы должен быть доступен пользователю. Все это обеспечивают специальные операторы ввода/вывода. Важным элементом современных программ является графическое отображение результатов работы. Эти и мно-

жество других возможностей поддерживаются *библиотечными модулями*. Подробнее о программировании операций ввода/вывода речь пойдет позже.

## Программа

Программой могут называть разные вещи. Это может быть исходный текст программы — обычный текстовый файл, содержащий запись операторов программы на языке программирования. Такая запись понятна человеку, но непонятна компьютеру. Исходный текст должен быть *откомпилирован* (отрансляирован), то есть переведен на язык машинных команд, понятный компьютеру. В этом случае создается *исполняемый файл* (его отличительная черта — имя, оканчивающееся на .exe). Именно исполняемый файл иногда называют программой. И, наконец, программой могут называть набор всевозможных файлов, как исполняемых, так и текстовых (а возможно, и в других форматах).

Мы будем считать, что программа представляет собой последовательность операторов и других элементов языка, построенную в соответствии с определенными правилами и предназначенную для решения определенной задачи. Первым в программе идет зарезервированное слово `program`. За ним, после одного или нескольких пробелов, следует *идентификатор* — имя программы. Идентификаторы могут содержать любое количество символов, но Турбо Паскаль распознает только первые 63 из них, что, разумеется, намного превосходит реальные потребности. Идентификатор должен начинаться буквой или символом подчеркивания. Затем могут идти буквы, цифры и символы подчеркивания. Взятая в целом, фраза `program s_kate;` является заголовком программы с именем `s_kate`. Каждое описание должно завершаться точкой с запятой. Таким образом, первая строка любой программы имеет вид

`program name;`

В Турбо Паскале оператор заголовка программы может быть опущен. Имя программы никогда в ней фактически не используется, и оно совершенно не связано с именем внешнего файла, содержащего текст программы.

После заголовка программы обычно идут описания переменных, констант, методов, подпрограмм и других объектов, используемых в программе. Эта часть программы называется *разделом описаний*.

Каждая программа обязательно должна иметь часть, которая выполняет какие-либо действия и называется *разделом операторов* (иногда — *телом программы*).

Минимально допустимой выполняемой частью программы является составной оператор

```
begin
  S1; S2; ... ; Sn;
end
```

где `S1, ..., Sn` — операторы, а зарезервированные слова `begin` и `end` играют роль скобок, но только для операторов, а не для математических выражений. Они так и называются *операторными скобками*. Каждому `begin` в программе должен

соответствовать `end`. Обратное, вообще говоря, неверно, так как `end` может заканчивать разделы, начинающиеся зарезервированными словами `case` и `record`. За телом программы должна следовать точка — признак того, что здесь находится конечная точка останова программы. Структура программы изображена на рис. 1.1.

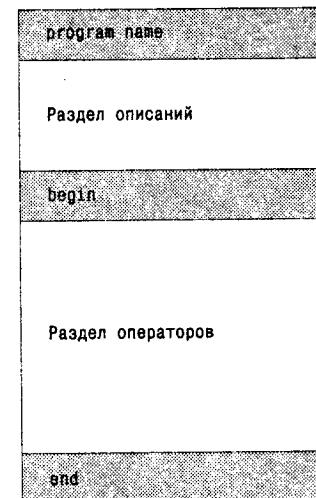


Рис.1.1. Структура программы на языке Паскаль

Приведу еще два примера. Операторы в программе могут размещаться как на отдельных строках, так и по несколько в строке. Операторы разделяются точкой с запятой. В первом из приведенных ниже примеров между каждыми двумя последовательными разделителями находится *пустой оператор*. Обратите внимание на то, что допускаются дополнительные пары `begin..end` и дополнительные точки с запятой.

```
program DE;
begin
  ; ; ; ;
end.
```

```
program FG;
begin
  begin
    ;
  end;
  ;
end.
```

Сделаю замечание по поводу формата записи текста программы, так как это вопрос не только эстетический, но и вопрос эффективности труда программиста. При наборе текста программы ее операторы следует располагать таким образом, чтобы понятной была логика работы программы. Для этого используются пробелы между операторами или элементами операторов, а также отсту-

пы. Каждый программист имеет свой стиль использования пробелов и отступов. Пробелы и отступы помогают читателю программы определить уровень подчиненности каждой строки. Компьютерные программы содержат большое количество информации в концентрированном виде, и хорошее форматирование может оказаться очень полезным. Надо иметь в виду, что вряд ли вновь написанная программа сразу будет работать правильно. На отладку программы может уйти больше половины времени ее разработки. Должное внимание к хорошему форматированию, тщательному выбору имен переменных и других объектов программы, разумное документирование программ может значительно увеличить производительность труда. Не следует размещать в одной строке более одного-двух операторов. При записи программы на Паскале допускаются пустые строки, которые можно использовать для выделения логически связанных групп операторов.

Важной частью исходного текста программы являются *комментарии*. Комментарий представляет собой текст, который находится между фигурными скобками или между парами символов, состоящими из круглой скобки и звездочки (см. табл. 1.2). Текст комментария не обрабатывается компилятором и не включается в исполняемый файл. Комментарии позволяют включить подробное описание программы и пояснения к ней прямо в исходный текст. Грамотное и уместное применение комментариев упрощает понимание программы, облегчает жизнь ее автору и программистам, работающим с уже готовым текстом. У фигурных скобок есть и нестандартное применение — во время отладки часто возникает необходимость временно убрать из программы какие-то операторы, сохранив, тем не менее, их запись. Простейший способ — заключить соответствующий фрагмент программы в фигурные скобки.

## Разделы описаний

### Описание переменных

Все переменные, используемые в программе, должны быть перечислены в *разделе описания переменных*. Этот раздел состоит из *предложений описания переменных*. Таких предложений может быть несколько, размещаются они между заголовком программы, подпрограммы или модуля и зарезервированным словом `begin`, открывающим раздел операторов программы, подпрограммы или модуля. Располагаться предложения описания переменных могут вместе (и это одна из составных частей хорошего стиля программирования), но могут и чередоваться с описаниями других объектов: констант, процедур, функций и т. д. Предложение описания переменных имеет вид

```
var v1, v2,... : type_id;
```

Здесь `v1, v2...` — список переменных, в котором имена переменных разделяются запятыми, а `type_id` задает тип переменных из данного списка. Если в данной программе используются переменные разных типов, то в предложении `var` приводятся списки имен переменных каждого типа:

```
var v_1_1, v_1_2... : type_id_1;
v_2_1, v_2_2... : type_id_2;
...
v_n_1, v_n_2... : type_id_n;
```

Пример описания переменных:

```
var
  cows, sheeps : Word;
  overman      : Real;
  milkmaid     : Extended;
```

Здесь **Word**, **Real** и **Extended** — названия типов.

## Типы **Word**, **Real** и **Extended**

Паскаль — это гибкий язык, в котором имеется большое число различных типов. Сейчас мы познакомимся только с некоторыми из них. **Word** — зарезервированное слово, обозначающее целочисленный тип с диапазоном значений [0, 65 535]. Переменные типа **Word** могут принимать целые значения только из указанного интервала.

Числовой тип **Real** — это вещественные значения из диапазона плюс-минус  $[2,9 \times 10^{-39}, 1,7 \times 10^{38}]$ . Переменные типа **Real** не могут принимать значения, сколь угодно близкие к нулю.

Тип **Extended** имеет более широкие возможности, чем **Real**, так как его диапазон составляет плюс-минус  $[3,4 \times 10^{-4932}, 1,1 \times 10^{4932}]$ , и поэтому в программах рекомендуется использовать именно этот тип.

## Описание констант

В разделе описаний программы должны быть описаны не только переменные, но и константы. В простейшем случае предложение описания констант имеет вид

```
const
  v_1 = val_1;
  v_2 = val_2;
  ...
  v_n = val_n;
```

Здесь **v\_1**, **v\_2**, ..., **v\_n** — имена констант, а **val\_i** — значения этих констант. Позже мы узнаем, что константы в Паскале бывают двух видов — *нетипизированные* (как в данном случае) и *типизированные*.

Пример описания констант:

```
const
  my_birth_year    = 1905;
  mass_of_electron = 9.1095e-28;
  my_salary        = 'invisible';
```

## Исполняемые операторы

Тело каждой программы или подпрограммы состоит из последовательности операторов, каждый из которых выполняет определенное действие. Рассмотрим некоторые операторы языка Паскаль. Начнем с *оператора присваивания*:

`variable := expression;`

*Выражение* справа от символа присваивания состоит из констант, переменных, обращений к функциям и знаков операций. Вначале вычисляется значение выражения. Затем полученное значение заносится в ячейку памяти компьютера, зарезервированную под переменную, имя которой указано в левой части оператора присваивания. Значение выражения должно быть совместимо по типу с указанной переменной. Например, любые значения целого типа могут быть присвоены другим переменным целого типа, значение любого выражения целого типа может быть присвоено любой переменной вещественного типа и т. д.

Примеры операторов присваивания:

```
a1 := 0.5;
y := x / (1.0 + x);
```

## Вызовы процедур

Процедуры можно считать нестандартными операторами языка. Процедура программируется пользователем или содержится в готовом наборе процедур (библиотеке, библиотечном модуле) и обычно выполняет достаточно сложное действие. Процедура вызывается путем указания ее имени, если она не имеет параметров, или ее имени с параметрами, заключенными в круглые скобки. Тем самым инициируются действия, объем которых может быть довольно большим. Примерами вызовов процедур являются

```
WriteLn;
Read(infile, x);
```

Программирование процедур обсуждается во втором уроке.

## Составной оператор

Составной оператор — это оператор вида

```
begin
  S_1;
  S_2;
  ...
  S_n;
end;
```

где операторы  $S_i$ , в свою очередь, могут быть простыми или составными операторами. Составной оператор трактуется как один оператор. Такая конструкция используется в ситуациях, когда, согласно формальным правилам язы-

ка, разрешается использование лишь одного оператора, а в действительности требуется несколько операторов. С этим приходится сталкиваться, например, при программировании циклов или условных операторов.

Два слова о знаках пунктуации. Строго говоря, символ «точка с запятой» в Паскале является не *ограничителем*, а *разделителем* операторов. Поэтому точка с запятой после `S := S + 1;` необязательна. Тем не менее использование точки с запятой облегчает модификацию программы, например, при включении добавочных операторов.

Пример составного оператора:

```
begin
  S := S + 1;
  a := S - sqr(S);
end;
```

## Структурные операторы

Структурные операторы строятся из специальных зарезервированных слов, логических выражений и других операторов. Каждый такой оператор явно или неявно содержит одну или несколько логических проверок.

### Оператор if...then...

Оператор `if...then...` называется *условным оператором* и имеет вид

```
if expression1 then statement1;
```

где выражение `expression1` является *логическим*. Логическое выражение принимает одно из двух возможных значений — `True` (истина) или `False` (ложь). Часто в роли логического выражения выступает какое-то условие, которое может выполняться либо нет. В первом случае его значение — «истина», а во втором — «ложь». Программирование логических выражений мы будем разбирать позже. Если логическое выражение `expression1` принимает значение «истина», то выполняется оператор `statement1`. В противном случае выполняться будет оператор, следующий за данным логическим оператором.

Следует отметить, что, согласно формальным правилам языка, в условном операторе после `then` допускается применение *только одного* оператора. Но в практике программирования чаще возникают ситуации, когда при выполнении условия в логическом выражении `expression1` следует выполнить *несколько* операторов языка. Решается эта проблема, как уже было сказано, применением составного оператора.

Операторы `if...then...` можно вкладывать друг в друга, так как конструкция

```
if expression2 then statement2;
```

также является оператором и может заменить оператор `statement1`:

```
if expression1 then if expression2 then statement2;
```

Пример условного оператора:

```
if Centigrade = 0 then Write('Температура замерзания воды');
```

## Оператор if...then...else...

Этот оператор является полной версией условного оператора и имеет вид

```
if expression then statement1 else statement2;
```

Выполняется данный оператор следующим образом: если выражение `expression` принимает значение «истина», то управление передается на оператор `statement1`, если же нет, то на оператор `statement2`. Приведу ошибочный вариант данного оператора:

```
if expression then statement1 else; statement2;
```

Здесь первая точка с запятой завершает оператор `if...then...else`, не выполняя никаких действий в случае `else`, а затем (в любом случае) выполняется оператор `statement2`.

### Оператор

```
if expression1 then
  if expression2 then
    statement2
  else
    statement1;
```

допускает двоякую интерпретацию. Первый вариант соответствует последовательности операторов

```
if expression1 then
begin
  if expression2 then
    statement2
  else
    statement1;
end;
```

Второй вариант:

```
if expression1 then
begin
  if expression2 then
    statement2
end
else
  statement1;
```

Компилятор Паскаля всегда выбирает первый из приведенных вариантов — каждому `else` соответствует ближайший предшествующий `if`. Если требуется реализация второго варианта, можно использовать операторные скобки `begin..end`. В общем случае, чтобы четко определить, что чему подчинено, используйте `begin..end` аналогично круглым скобкам в арифметических выражениях.

Пример условного оператора:

```
if Two = 2 then
  Writeln('Два равно 2');
```

```

else
  Writeln('Это не 2. В чем дело? ');

```

### Вложенные операторы if...then...else...

Как уже отмечалось, условные операторы можно вкладывать друг в друга, программируя таким образом сложные ветвления. Рассмотрим следующий оператор:

```

if expression_1 then
  statement_1
else
  if expression_2 then
    statement_2
  else if expression_3 then
    statement_3
  ...
  else if expression_n then
    statement_n;

```

Вначале вычисляется значение логического выражения `expression_1`. Если оно истинно, выполняется оператор `statement_1`, если же это значение ложно, вычисляется значение выражения `expression_2`. В том случае, когда полученное значение истинно, будет выполняться оператор `statement_2`, при значении «ложь» будет вычисляться выражение `expression_3` и т. д.

Если выражения `expression_i` независимы, то есть вычисление их значений в любом порядке дает один и тот же результат для каждого из них, имеет смысл располагать их в таком порядке, чтобы выражение, с наибольшей вероятностью принимающее значение «истина», стояло на первом месте, выражение, принимающее значение «истина» с меньшей вероятностью, — на втором и т. д. Это уменьшит время выполнения данного фрагмента программы, особенно если вложенный оператор появляется в цикле, который выполняется много-кратно.

Пример вложенных условных операторов:

```

if Two = 2 then
  if One = 1 then
    Writeln('Единица равна 1')
  else
    Writeln('Единица не равна 1')
else
  if Three = 3 then
    Writeln('Три равно 3')
  else
    Writeln('Три не равно 3');

```

### Оператор case...of...end

Для ситуаций, где имеется несколько (три и более) альтернатив, больше подходит оператор `case`. Этот оператор называется *оператором выбора* и имеет следующий вид:

```
case expression of
```

```

values_1: statement_1;
values_2: statement_2;
...
values_n: statement_n;
else statement;
end;

```

Рассмотрим элементы этой конструкции. Во-первых, это три зарезервированных слова: `case`, `of` и `end`. Между `case` и `of` находится выражение `expression`, принимающее значение, которое, возможно, имеется в одном из списков значений, находящихся слева от двоеточий. Данное выражение называется *селектором* оператора `case`. Каждый оператор, идущий за двоеточием, отделяется от следующего списка значений точкой с запятой. Ветвь `else`, отвечающая всем не перечисленным значениям выражения `expression`, необязательна. При выполнении данного оператора вначале вычисляется значение селектора. Затем выбирается тот список значений, которому принадлежит полученное значение, и выполняется соответствующий оператор.

В списках значений оператора `case` допустимыми являются типы переменных, называемые *скалярными* (они будут обсуждаться позже), включая целые и исключая вещественные типы. Любое заданное значение селектора может входить в список значений неоднократно, но выполняться будет лишь первая подходящая ветвь. Замечу, что «стилистически» такая конструкция выглядит не очень изящно. Если значение селектора отсутствует в списках значений, ни одна из альтернатив выполняться не будет. В этом случае выполняется ветвь `else` оператора `case` или (если эта ветвь отсутствует) следующий за `case` оператор.

Поясню применение данного оператора следующим примером. Пусть необходимо преобразовать целое число  $N$  в зависимости от величины остатка от его деления на 17 следующим образом:

```

если  $N \bmod 17 = 0$ , то  $N := 0$ ;
если  $N \bmod 17 = 1$  или  $6$ , то  $N := -N$ ;
если  $N \bmod 17 = 2, 3$  или  $5$ , то  $N := 2 \times N$ ;
если  $N \bmod 17 = 4$ , то  $N := 3 \times N$ ;
во всех прочих случаях  $N := 5 \times N$ .

```

Решение этой задачи на Паскале выглядит следующим образом:

```

case  $N \bmod 17$  of
  0 :  $N := 0$ ;
  1, 6 :  $N := -N$ ;
  2, 3, 5:  $N := 2 \times N$ ;
  4 :  $N := 3 \times N$ ;
  else :  $N := 5 \times N$ ;
end;

```

В данном примере селектором является выражение  $N \bmod 17$ . Кроме того, имеются 4 списка значений и ветвь `else`.

**Оператор цикла while...do...**

Оператор цикла является важнейшим оператором и имеется в большинстве современных языков программирования (а сама идея цикла возникла еще в XIX веке!). Цикл позволяет многократно выполнить некоторое множество действий, задаваемых операторами, составляющими его тело. В Паскале имеется несколько разновидностей оператора цикла. Начнем с оператора *цикла с предусловием*. Данный оператор имеет вид

```
while expression do statement;
```

При выполнении этого оператора вначале вычисляется значение логического выражения *expression*. Если это значение истинно, выполняется оператор *statement*, затем значение выражения проверяется вновь и т. д., до тех пор, пока выражение не примет значение «ложь». Если выражение принимает значение «ложь» при первой же проверке, то оператор *statement* не выполняется вообще. Особо отмечу частный случай:

```
while True do statement;
```

Здесь оператор *statement* будет выполняться бесконечно.

Пример оператора цикла с предусловием:

```
while Counter < 10 do begin
  Write('Значение счетчика равно ',Counter);
  Writeln;
  Counter := Counter + 2;
end;
```

**Оператор цикла repeat...until...**

Оператор *цикла с постусловием* имеет вид

```
repeat statement until expression;
```

Здесь вначале выполняется оператор *statement*, а затем вычисляется значение логического выражения *expression*. Процесс повторяется, пока выражение *expression* принимает значение «ложь». Как только это значение станет истинным, выполнение цикла прекращается. Оператор *statement* может быть любым, в том числе и составным оператором:

```
begin
  statement_1;
  statement_2;
  ...
  statement_n;
end;
```

В цикле *repeat...until...* операторные скобки *begin..end* могут быть опущены. Таким образом, в общем случае оператор *repeat...until...* имеет следующий вид:

```
repeat
  statement_1;
  statement_2;
  ...;
```

```
statement_n;
until expression;
```

Точка с запятой перед зарезервированным словом `until` необязательна. В приведенном ниже частном случае

```
repeat
  statement_1;
  statement_2;
  ...
  statement_n;
until False;
```

цикл выполняется бесконечно. Еще раз обращаю ваше внимание на то, что если в операторе `while...do...` проверка выполняется в начале цикла; то в цикле `repeat...until...` проверка выполняется в последнюю очередь, и тело цикла *в любом случае* выполняется хотя бы один раз.

Вот пример цикла с постусловием:

```
repeat
  Write('Значение счетчика равно ',Count);
  Writeln;
  Count := Count + 2;
until Count = 10;
```

### Операторы цикла `for...to...do...` и `for...downto...do...`

Третий вариант оператора цикла — *цикл со счетчиком*. Можно считать, что есть две очень похожих друг на друга разновидности цикла со счетчиком. Первый из этих операторов имеет вид

```
for j := expression1 to expression2 do statement;
```

Здесь переменная `j`, называемая управляющей переменной цикла `for`, является произвольным идентификатором, который объявляется как переменная любого скалярного типа (к скалярным относятся целый, символьный, булев и перечислимые типы).

При выполнении оператора `for` сначала вычисляется значение выражения `expression1`, затем вычисляется значение выражения `expression2`, далее управляющая переменная цикла последовательно пробегает все значения от `expression1` до `expression2`. В том случае, когда значение `expression1` оказывается больше значения `expression2`, тело цикла не будет выполняться вовсе. Эти значения остаются неизменными в ходе выполнения всего цикла `for`. Рассматриваемый вариант цикла `for` эквивалентен следующей последовательности операторов:

```
j := expression1;
k := expression2;
while j <= k do
begin
  statement;
  inc(j);
end;
```

в предположении, что при каждом выполнении оператора **statement** не изменяются значения **j** и **k**.

Оператор **for** вида **for j := expression1 to expression2 do statement;** неэквивалент последовательности операторов

```
begin
  j := expression1;
  while j <= expression2 do
    begin
      statement;
      j := j + 1;
    end;
  end;
```

потому что выражение **expression2** может изменяться при каждом выполнении оператора **statement** в цикле **while**.

В теле цикла **for** следует избегать операторов, изменяющих значение управляющей переменной **j**. Несмотря на то что использование подобных конструкций не приводит к ошибкам компиляции, они потенциально опасны и могут приводить к неприятным последствиям. Рассмотрим пример:

```
sum := 0;
for k := 1 to 100 do
begin
  sum := sum + Sqr(k); k := k + 2;
end;
```

Этот фрагмент программы является попыткой просуммировать  $n^2$  по всем целым значениям вида  $n = (3*k+1)$ , лежащим в диапазоне от 1 до 100. Здесь допущена ошибка реализации алгоритма, так как управляющая переменная **k** изменяется в составном операторе, управляемом той же переменной **k**. Правильной будет конструкция следующего вида:

```
sum := 0;
for k := 0 to 33 do sum := sum + Sqr(3*k + 1);
```

или

```
sum := 0; k := 1;
repeat
  sum := sum + Sqr(k); k := k + 3;
until k > 100;
```

После выполнения цикла **for** значение управляющей переменной становится неопределенным.

Вариант **for...downto...do...** цикла **for** аналогичен циклу **for...to...do...** за исключением того, что в нем управляющая переменная на каждом шаге выполнения не **увеличивается**, а **уменьшается** на единицу:

```
for j := expression1 downto expression2 do statement;
```

Подводя итоги, для применения циклов можно сформулировать следующие рекомендации:

- Используйте цикл `for` в том случае, когда *точно знаете*, сколько раз должно быть выполнено тело цикла. В противном случае обратитесь к циклам `repeat` или `while`.
- Используйте `repeat`, если необходимо, чтобы тело цикла выполнялось *по крайней мере один раз*.
- Используйте `while`, если хотите, чтобы проверка была произведена *прежде*, чем будет выполняться тело цикла.

Иногда бывает удобно проводить проверку на возможный выход из цикла где-нибудь в его середине, а не в начале или конце. Такой выход из цикла обеспечивается процедурой `Break` модуля `System`, которая прерывает выполнение самого внутреннего вложенного цикла, будь то `for`, `while` или `repeat`. Указанный модуль подключается к программе автоматически, если в этом есть необходимость. Пример:

```
while true do
begin
  statement1;
  if expression then Break;
  statement2;
end;
```

Следует также упомянуть процедуру `Continue`, которая прерывает выполнение тела самого внутреннего цикла `for`, `while` или `repeat` и передает управление на его заголовок, так что начинается выполнение очередной итерации цикла.

## Первые программы

Получив первоначальный запас теоретических знаний, попробуем применить его на практике. Сначала разберем примеры простых программ. Первая из них приведена в листинге 1.1.

**Листинг 1.1.** Самая первая программа

```
program first_program;

begin
end.
```

Это простейшая программа на языке Паскаль. Она предназначена для того, чтобы... просто быть, то есть никаких реальных действий она не выполняет. В ней не предусмотрен ни ввод, ни вывод данных. Нет арифметических выражений. Эта программа лишь демонстрирует структуру, то есть внутреннее устройство программы. Напомню, что любая программа или библиотечный модуль начинаются с заголовка. У программы это ключевое слово `program`, за которым следует имя программы. Имя может состоять из букв, цифр и символов подчеркивания. Длину имени можно считать произвольной, хотя вряд ли

имеет смысл придумывать очень длинные имена. Кроме того, неважно, в каком регистре клавиатуры имя набрано. Между заголовком и оператором `begin` размещаются описания переменных и других объектов программы. Между зарезервированными словами `begin` и `end` находятся исполняемые операторы программы. Завершает программу точка. В нашем первом примере нет ни описаний, ни исполняемых операторов.

Второй пример сложнее. Мы вычислим сумму натуральных чисел от 1 до 20. Читатель, знакомый с математикой, знает, что сумму первых  $n$  натуральных чисел можно найти по формуле  $S_n = n(n + 1)/2$ . Используя эту формулу, мы сможем проверить, правильно ли работает программа и не допущены ли во время ее набора ошибки. При разработке программ их отладка, а также проверка правильности работы являются обязательной составной частью работы программиста. Ведь даже если все операторы программы написаны правильно с точки зрения формальных правил языка, ошибка может быть допущена в самом алгоритме или в его записи на языке программирования. Программу можно считать готовым продуктом, только если программист убедился сам и убедил заказчика в том, что программа работает правильно, дает правильный результат. Для тестирования программы используются такие наборы значений входных параметров или такие предельные случаи ее работы, для которых известен точный результат. Результат работы программы в этом случае сравнивается с точными значениями.

#### Листинг 1.2. Вычисление суммы натуральных чисел

```
program summation;
var
    i, summa: Word;
begin
    {переменной summa присвоим начальное значение}
    summa := 0;
    for i := 1 to 20 do summa := summa + i;
    WriteLn('1 + 2 + ... + 20 = ', summa);
    Write('Нажмите клавишу <Enter>:');
    ReadLn;
end.
```

В этой программе по сравнению с первым примером появились новые элементы. Это описание двух переменных, используемых в программе:

```
var
    i, summa: Word.
```

Исполняемая часть программы, начинающаяся строкой `begin` и завершающаяся строкой `end`, уже не пустая, она содержит исполняемые операторы. Первая строка в разделе операторов — комментарий, заключенный в фигурные скобки (напомню, что в качестве ограничителей комментария допустимы и пары символов, состоящие из круглой скобки и звездочки). Оператор `summa := 0;` инициализирует переменную `summa`, используемую для хранения частичной суммы, присваивая ей нулевое значение. Замечу, что до первого оператора присваивания, содержащего имя переменной в левой части, ее значение не определено.

Затем идет цикл со счетчиком. В нашем примере тело цикла выполняется 20 раз, и каждый раз к значению переменной `s` прибавляется значение переменной — счетчика `i`.

Следующие две строки реализуют вывод результата на экран. Для этого в программах на языке Паскаль используются операторы вывода `Write` и `WriteLn`. Они содержатся в библиотечном модуле `System` и во время компиляции автоматически включаются в исполняемый код. Вначале выводится символьная строка. Текст, выводимый на экран, заключается в одиночные кавычки '`...`'. Затем выводится численное значение — `summa`.

Последние две буквы в имени процедуры `WriteLn` означают, что после того, как вывод закончен, курсор переходит на начало следующей строки. Следующий оператор предлагает пользователю нажать клавишу `Enter`, а оператор `ReadLn` без параметров ожидает нажатие этой клавиши. После нажатия клавиши `Enter` выполнение программы завершается.

### 8 ВНИМАНИЕ

В Паскале не различаются большие и маленькие буквы, то есть `Sum`, `sum`, `SUM` и т. д. представляют собой один и тот же идентификатор.

### ПРИМЕЧАНИЕ

Обратим внимание на знаки пунктуации в этой программе. Точка с запятой (`:`) не только завершает описания, но и разделяет операторы в исполняемой части программы. Запятая (`,`) всегда используется для того, чтобы разделить элементы списка, — в данном случае в разделе `var` и при вызове процедуры `WriteLn`. Двоеточие (`:`) отделяет список описываемых переменных от названия их типа `Word`. Программы, как и обычные предложения, заканчиваются точкой.

### ПРИМЕЧАНИЕ

В рассматриваемой программе имеются два оператора присваивания `:=`. В Паскале одиночный знак равенства для присваивания никогда не используется, он имеет другое назначение.

В арифметических выражениях используются символы арифметических операций. Эти символы приведены в табл. 1.2. Так, например, в произведениях между сомножителями должен находиться символ операции умножения `*`. Математическое выражение  $ax^2$  в программе на Паскаль записывается как `a*x*x`. Следует заметить, что особенностью языка Паскаль является отсутствие стандартной встроенной функции вычисления произвольной степени числа, кроме второй. Это, конечно же, неудобно при программировании сложных вычислений. При программировании арифметических выражений следует помнить о приоритетах операций, то есть о порядке их выполнения. Первыми выполняются арифметические операции умножения и деления, они считаются равноприоритетными операциями. Затем выполняются операции сложения и вычитания, тоже равноприоритетные. Если подряд идут несколько равноприоритетных операций, они выполняются слева направо. Порядок выполнения

операций может быть изменен с помощью круглых скобок. При наличии в арифметическом выражении круглых скобок первыми будут выполняться операции в круглых скобках, начиная с самых внутренних.

Следующая программа предназначена для вывода таблицы соответствия между температурными шкалами Цельсия и Фаренгейта в интервале температур от точки замерзания воды до точки ее кипения. Температурная шкала Фаренгейта была предложена немецким физиком Габриэлем Фаренгейтом и используется в настоящее время в ряде англоязычных стран. В этой шкале при стандартном атмосферном давлении температура замерзания воды равна 32 °F, а температура кипения составляет 212 °F. В более привычной для нас шкале Цельсия аналогичными опорными точками являются, соответственно, 0 °C и 100 °C. Эти значения и используются для пересчета одних температур в другие. Нетрудно проверить, что формула для пересчета имеет вид:  $t_f = 9/5*t_c + 32$ , где  $t_f$  — температура по Фаренгейту, а  $t_c$  — температура по Цельсию.

### Листинг 1.3. Вывод таблицы соответствия температур по Цельсию и Фаренгейту

```
program Celsius_to_Fahrenheit;
var
  i, Celsius, Fahrenheit: Word;
begin
  Writeln('Таблица соответствия между температурными шкалами');
  Writeln('Цельсия и Фаренгейта');
  Writeln;
  for i := 0 to 20 do begin
    Celsius := 5 * i;
    Fahrenheit := 32 + Celsius * 9 div 5;
    Write('  C =', Celsius);
    Write('  F =', Fahrenheit);
    Writeln;
  end;
  Writeln('Нажмите <Enter>');
  ReadLn;
end.
```

Здесь есть новый элемент — операция `div`. В Паскале имеются две разновидности операции деления. Это обычная операция деления `/` и операция целочисленного деления `div`. В первом случае делимое и делитель могут быть любого числового типа, а во втором они должны быть целыми числами. Результат целочисленного деления — тоже целое число, которое получается отбрасыванием дробной части частного. Есть еще операция вычисления остатка от деления одного целого числа на другое — `mod`. В программе переменная `Fahrenheit` имеет целый тип `Word`, поэтому применение операции `/` привело бы к вещественному результату и, как следствие, к ошибке компиляции «Type mismatch» («Несоответствие типов»). Отмечу, что Паскаль является языком со строгим контролем за соблюдением типов. Это несомненное достоинство языка, так как, принуждая программиста быть аккуратным при работе с переменными, система программирования на Паскале избавляет его от необходимости искать ошибки на этапе выполнения программы. Это значительно более трудная проблема.

Оператор вывода `Write` отличается от оператора `WriteLn` тем, что при выводе значений не происходит переход на следующую строку. В нашем случае оба числа будут выведены в одной строке.

В следующей программе вычисляется сумма:

$$S_n = \sum_{i=1}^n \frac{1}{i^2}.$$

#### Листинг 1.4. Вычисление суммы

```
program summation_2;
var
  i, n: Word;
  t, add, summa: Real;
begin
  Write('Введите количество слагаемых n = ');
  ReadLn(n);
  summa := 0;
  for i := n downto 1 do
  begin
    t := 1.0/i;
    add := Sqr(t);
    summa := summa + add;
  end;
  WriteLn('Сумма 1/i^2 от i = 1 до ', n);
  WriteLn('S = ', summa);
  Write('Нажмите клавишу <Enter>:');
  ReadLn;
end.
```

В этой программе есть переменные типов `Real` и `Word`. Во втором операторе присваивания внутри цикла используется функция вычисления квадрата числа — `Sqr`. Кроме того, здесь мы встречаемся с разновидностью цикла `for`, в которой благодаря использованию ключевого слова `downto` вместо `to` значение управляющей переменной `i` уменьшается каждый раз на единицу. В рассматриваемом случае лучше всего проводить суммирование в обратном порядке, то есть от слагаемых с наибольшими номерами, принимающими *наименьшие* значения, к слагаемым с наименьшими номерами, принимающими *наибольшие* значения. При суммировании в прямом порядке, начиная с некоторого номера, может возникнуть ситуация, когда при каждом проходе цикла к относительно большому значению суммы будет прибавляться сравнительно малое значение очередного слагаемого. Погрешность выполнения такой операции увеличивается — это особенность машинной арифметики. При суммировании в обратном порядке слагаемые не так сильно различаются между собой по величине, и, следовательно, точность вычисления полной суммы будет выше.

Следующая программа предназначена для решения диофантовых уравнений. Таким образом мы попадаем в один из сложных разделов математики. Не стоит этого пугаться, ведь мы вооружены компьютером и знанием, правда, еще далеко не полным, языка программирования. Маленькая историческая справка. Открытие диофантовых уравнений связано с именем греческого математика

Диофанта, личности полулегендарной. О его жизни практически ничего не известно, но сегодня диофантов анализ — это обширная и важная область математики. Полная математическая теория разработана только для линейных уравнений, а общий метод решения нелинейных диофантовых уравнений все еще неизвестен. Иногда анализ простого на вид нелинейного диофантова уравнения может представлять огромные трудности даже для математика высокой квалификации. Используя компьютер, оснащенный системой программирования на Турбо Паскале, мы сможем применить его возможности для решения как линейных, так и нелинейных диофантовых уравнений.

Самое простое линейное диофантово уравнение имеет вид  $ax + by = c$ , где  $a$ ,  $b$  и  $c$  — заданные числа, а  $x$  и  $y$  — неизвестные. Особенность этих уравнений заключается в том, что для них ищутся *целочисленные* решения. Это можно сделать методом перебора.

Для того чтобы немного оживить наше повествование, решим следующую старинную задачу из области экономики сельского хозяйства. Зажиточный крестьянин потратил 100 рублей на покупку 100 различных домашних животных. Каждая корова обошлась ему в 10 рублей, свинья в 3 рубля, а овца в 50 копеек. Предполагая, что крестьянин приобрел по крайней мере по одному животному каждого вида, найдем, сколько голов скота каждого вида он купил. Условие задачи записывается в виде двух уравнений:

$$\begin{aligned}10x + 3y + z/2 &= 100; \\x + y + z &= 100,\end{aligned}$$

где  $x$ ,  $y$ ,  $z$  — количество коров, свиней и овец соответственно. Избавимся от знаменателя в первом уравнении, умножив его на 2. Из полученного таким образом уравнения вычтем второе. Это позволяет исключить переменную  $z$ . Получаем уравнение  $19x + 5y = 100$ . Решениями данного уравнения должны быть целые положительные числа (видел ли кто-нибудь отрицательное число коров?), меньшие 100. Следующая программа предназначена для решения данного уравнения.

#### Листинг 1.5. Решение линейного диофантова уравнения

```
program diophantine_equation_1;
var
  x, y: Integer;
begin
  WriteLn('Целые решения уравнения 19x + 5y = 100 из диапазона');
  WriteLn('1 <= x <= 100, 1 <= y <= 100:');
  for x := 1 to 100 do
    for y := 1 to 100 do
      if 19*x + 5*y = 100 then
        WriteLn('(x, y) = ('. x, ', ', y, ')');
  Write('Нажмите <Enter>');
  ReadLn;
end.
```

Эта программа знакомит нас с новыми элементами. Здесь имеется двойной вложенный цикл `for`. Внутренний цикл содержит условный оператор `if... then...`. Опера-

тор `WriteLn` выполняется только в том случае, когда истинно условие в операторе `if`. В данном случае это условие  $19*x+5*y=100$ . Обратим внимание на то, что знак равенства обозначает здесь не оператор присваивания, а логическое отношение равенства двух значений. Результатом такого сравнения может быть или `True` (истина), если условие выполнено, или `False` (ложь), если условие не выполнено.

Оператор вывода

```
WriteLn('(x, y) = (' , x, ', ', y, ')');
```

используется для вывода на экран четырех элементов, которые разделяются запятыми. Последовательности символов, начинающиеся и заканчивающиеся одиночными кавычками ('), являются строками текста (строковыми константами). В нашем примере это '(x, y) = (' , ' и ')'. На экран будет выведен тот набор символов, который находится между кавычками. Нетекстовыми элементами списка вывода являются идентификаторы переменных `x` и `y`. На экран будут выведены значения этих переменных.

Как я уже упоминал, решение нелинейных диофантовых уравнений — это более сложная проблема. Но компьютер и умелое применение методов вычислительной математики часто позволяют быстро получить решение даже самых сложных задач. Вот пример кубического диофанта уравнения:

$$x^3 = y^2 + 2.$$

Известно его решение:  $x = 3$ ,  $y = 5$ . Усложним задачу и решим уравнение

$$x^3 = y^2 + 63.$$

Понятно, что по сравнению с первым диапазоном поиска решений придется увеличить.

#### Листинг 1.6. Решение нелинейного диофанта уравнения

```
program diophantine_equation_2;
var
  x, y, z, w, n: LongInt;
begin
  {Вначале найдем наибольшее n, для которого n^2 + 63 <= MaxLongint.
  MaxLongint = 2 147 483 647 – встроенная константа модуля System,
  задающая максимальное значение переменной типа LongInt}
  n := MaxLongint - 63;
  n := Trunc(Sqrt(n)); {Trunc(t) – целая часть величины t}
  n := n-8;
  x := 0;
  WriteLn('Все целые решения уравнения x^3 = y^2 + 63.');
  WriteLn('для 1 <= y <= ', n, ':');
  for y := 1 to n do
  begin
    repeat
      z := y*y + 63;
      Inc(x); {Функция Inc(x) увеличивает значение x на единицу}
      w := x*x*x;
    until w >= z;
    if w = z then
      WriteLn('(x, y) = (' , x, ', ', y, ')')
```

```

    else Dec(x); {Функция Dec(x) уменьшает значение x на единицу}
end;
Write('Работа закончена, нажмите <Enter>:');
ReadLn;
end.

```

Для того чтобы увеличить диапазон поиска решений уравнения, целые переменные программы x, y, z, w, n описаны как переменные типа `LongInt` («длинное целое»). Диапазон значений для типа «длинное целое» составляет  $[-2\ 147\ 483\ 648, +2\ 147\ 483\ 647]$ . Здесь же используется и оператор цикла `repeat...until...`. И, наконец, в программе содержится обращение к встроенным функциям округления к нулю (`Trunc`), увеличения и уменьшения значения аргумента на единицу (`Inc` и `Dec`).

В следующей программе, предназначеннной для вычисления произведения заданного числа сомножителей вида  $1 + (-1)^n/n^2$ , отметим функцию `Odd` из модуля `System`. Это логическая функция, которая принимает значение «истина» только в том случае, когда аргумент — нечетное число.

#### Листинг 1.7. Вычисление произведения

```

program product;
const
  m = 400;
var
  p, x : Extended;
  n : Word;
begin
  product := 1.0;
  for n := 2 to m do
  begin
    x := Sqr(1.0/n);
    if Odd(n) then
      {Для нечетных n}
      product := product * (1.0 - x)
    else
      {Для четных n}
      product := product * (1.0 + x);
  end;
  writeln('Произведение для 2 <= n <= ', m, ':');
  writeln;
  writeln('P (1 + (-1)^n /n^2) =', product);
  writeln;
  write('Нажмите <Enter>:');
  ReadLn;
end.

```

Следующая задача — вычисление суммы значений  $1/n^5$  в прямом и обратном порядке. Программа `forward_back_sum` знакомит нас с зарезервированным словом `uses`, которое дает возможность подключать к программе библиотечные модули. В данном примере используются процедуры для работы с экраном в текстовом режиме. Для очистки экрана используется процедура `ClrScr`, которая находится в библиотечном модуле `Crt`. Перед вычислением четвертой сте-

пени значения целой (типа Word) переменной k ее значение присваивается вещественной переменной x. Это делается для того, чтобы избежать переполнения. Ведь диапазон значений вещественных переменных значительно больше, чем диапазон значений целых переменных типа Word.

#### Листинг 1.8. Суммирование в прямом и обратном порядке

```
program forward_back_sum;
uses
  Crt; {Crt - библиотечный модуль, который содержит процедуры
        для работы с экраном в текстовом режиме}
var
  x, summa, ammus : Real;
  k                 : Word;
begin
  ClrScr;
  WriteLn(' 1/n^5, 1 to 1000');
  {Суммирование в прямом порядке}
  summa := 0.0;
  for k := 1 to 1000 do
  begin
    x := k;
    summa := summa + 1.0/(x*Sqr(Sqr(x)));
  end;
  {Суммирование в обратном порядке}
  ammus := 0.0;
  for k := 1000 downto 1 do
  begin
    x := k;
    ammus := ammus + 1.0/(x*Sqr(Sqr(x)));
  end;
  WriteLn('Прямая сумма      = ', summa);
  WriteLn('Обратная сумма     = ', ammus);
  WriteLn('Разность            = ', summa - ammus);
  WriteLn;
  Write('Нажмите <Enter>');
  ReadLn;
end.
```

Перед тем как продолжить чтение книги, еще раз просмотрите разобранные нами программы. Убедитесь, что вам в них все понятно. Поэкспериментируйте с ними, попробуйте поменять параметры и посмотрите, как это повлияет на результаты вычислений.

## Подпрограммы-функции

Важным принципом современного программирования является принцип модульности. В модульной программе отдельные ее части, предназначенные для решения каких-то частных задач, организованы в подпрограммы. В такой организации есть два больших преимущества. Во-первых, один и тот же фрагмент

можно использовать многократно как в одной, так и в разных программах, не набирая его текст заново. Во-вторых, программы лучше писать небольшими частями. Такие программы легче читать, тестировать и отлаживать. У них, как правило, более четкая логическая структура.

В языке Паскаль модульность обеспечивается использованием подпрограмм-функций, подпрограмм-процедур и модулей. Более подробное знакомство с процедурами мы отложим до второго урока, а первоначальные сведения о модулях даются в конце данного урока. Сейчас мы познакомимся с правилами программирования и применением функций.

Описание подпрограммы-функции должно располагаться в разделе описаний, то есть между заголовком программы (или другой подпрограммы) и зарезервированным словом `begin`. Описание подпрограммы-функции начинается с заголовка, который имеет вид

```
function name_of_function (arguments_list): type_of_result;
```

Здесь идентификатор типа результата `type_of_result` описывает тип значения, носителем которого является идентификатор (имя) функции `name_of_function`. Список параметров `arguments_list` содержит перечисление идентификаторов переменных — параметров функции. Эти параметры используются для передачи данных в подпрограмму-функцию. После имени переменной или группы имен следуют двоеточие и идентификатор типа переменных из этой группы. Список параметров может отсутствовать. Имя функции задается в соответствии с обычными правилами Паскаля.

Внутренняя структура подпрограммы-функции аналогична структуре программы, то есть сначала в ней идут описания, а затем, после зарезервированного слова `begin` — исполняемые операторы. Завершается подпрограмма-функция зарезервированным словом `end`, но за ним следует не точка, а символ «точка с запятой». В разделе описаний подпрограммы-функции могут содержаться описания других подпрограмм. Таких описаний может быть несколько. В теле подпрограммы-функции должен присутствовать оператор присваивания, в левой части которого указано имя функции, а в правой находится выражение.

Обращение к подпрограмме-функции производится просто путем указания ее имени в составе какого-либо выражения. Это может быть арифметическое выражение, если функция арифметическая. Имя функции в вызывающей программе может появиться только в правой части оператора присваивания.

Сделаю одно замечание. В дальнейшем наши программы будут использовать математический сопроцессор. *Математический сопроцессор* — это устройство, предназначенное для ускорения математических вычислений с вещественными числами. Были времена, когда математический сопроцессор был необязательной частью компьютера. Внутреннему представлению данных в сопроцессоре соответствует вещественный тип `Extended`. Указание транслятору генерировать код для математического сопроцессора задается директивой компилятора `{$N+}`, размещаемой в начале программы. Обратите внимание на то, что это не комментарий, хотя здесь используются фигурные скобки. Конструкция `{...}` со знаком доллара сразу после открывающей фигурной скобки всегда обозначает

директиву компилятора. Директива компилятора позволяет включить требуемый режим его работы.

### ПРИМЕЧАНИЕ

Директиву {\$N+} можно опустить, если в интегрированной среде Турбо Паскаля выбраны опции Options | Compiler | 8087/80287 (в интегрированной среде разработки компиляция для 80x87 выполняется по умолчанию) и Options | Compiler | Emulation (одновременно). Если же вы не располагаете математическим сопроцессором, следует выбрать в диалоговом окне пункт Emulation, но в этом случае не стоит рассчитывать на высокую скорость выполнения программы.

В программе `geron_sqrt`, предназначенней для вычисления квадратного корня из числа (в данном примере из двойки) методом Герона, используется подпрограмма-функция `geron`. Метод Герона – это метод последовательных приближений. Если задано число  $a$  и из него требуется приближенно вычислить квадратный корень, то вначале выбирается произвольное начальное приближение  $x_0$ . Затем задается точность вычислений  $\varepsilon > 0$  и строится последовательность  $x_{n+1} = (x_n + a / x_n) / 2$ . Вычисления прекращаются при выполнении условия  $|x_{n+1} - x_n| < \varepsilon$ .

Алгоритму Герона можно придать следующий геометрический смысл (рис. 1.2). Изобразим график функции  $y = x^2$  и проведем прямую  $y = a$ . Если в точке  $(x_n, x_n^2)$  провести касательную, то ее пересечение с прямой  $y = a$  даст значение  $x_{n+1}$ . Обозначая  $\Delta x = x_n - x_{n+1}$  и  $\Delta y = x_n^2 - a$ , получаем  $\Delta y / \Delta x = \tan \alpha$ . По формулам дифференциального исчисления  $\tan \alpha = dy / dx = 2x_n$ , поэтому  $(x_n^2 - a) / (x_n - x_{n+1}) = 2x_n$ , откуда и получается формула алгоритма Герона.

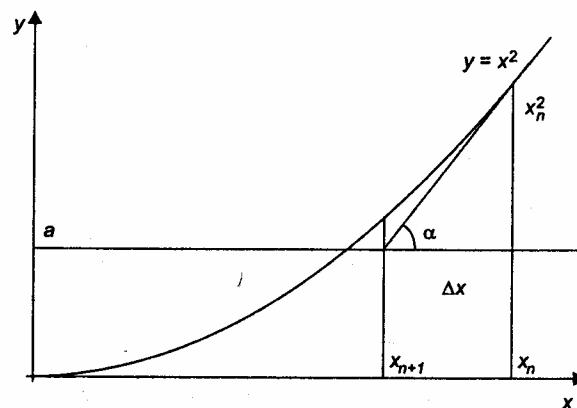


Рис. 1.2. Геометрическая иллюстрация алгоритма Герона

Цикл `while` в подпрограмме-функции `geron` выполняется до тех пор, пока не нарушится заданное в нем условие (неравенство). Невозможно заранее предугадать, когда это произойдет, и, следовательно, в такой ситуации цикл `for` был бы бесполезен. Подпрограмма `geron` вызывается в операторе `WriteLn`. Такой

вызовов функции в операторе вывода допускается. Во втором операторе вывода содержится обращение к встроенной функции вычисления квадратного корня. Таким образом, можно сравнить оба результата.

#### Листинг 1.9. Вычисление квадратного корня методом Герона

```
{$N+}
program geron_sqrt;

function geron(x: Real): Extended;
const
  eps = 1.0E-100;
var
  y, z: Real;
begin
  y := 1.0;
  {Цикл while выполняется до тех пор, пока не будет достигнута
  заданная точность вычисления квадратного корня}
  while Abs(z - y) >= eps do
    begin
      z := y;
      y := (y + x / y) / 2;
    end;
  geron := y;
end;

begin
  WriteLn('Алгоритм Герона для вычисления квадратного корня
  из двойки:');
  WriteLn('Geron(2.0)   =', geron(2.0));
  WriteLn('Sqrt(2.0)   =', Sqrt(2.0));
  Write('Нажмите <Enter>');
  ReadLn;
end.
```

В подпрограмме-функции `geron` в условии оператора цикла `while...do...` используется функция вычисления абсолютного значения `Abs`. Здесь же мы встречаемся с отношением нестрогого неравенства. При программировании логических отношений в Паскале используются четыре специальных символа для обозначения строгих и нестрогих неравенств: `<`, `>`, `<=` и `>=`. Отношение неравенства программируется при помощи пары символов `<>`.



#### ПРИМЕЧАНИЕ

Функция `geron` имеет свой собственный набор описаний — константы `eps` и переменных `y, z`. Эти величины локальны по отношению к `geron`, они «невидимы» из других процедур.

В следующем примере речь идет о решении неравенства  $b^n \leq a \leq b^{n+1}$  относительно  $n$  при условии  $a \geq 1, b > 1$ . Неравенство решается перебором значений  $n$ , метод решения реализован в функции `largest_power`, аргументами которой являются значения  $a$  и  $b$ .

**Листинг 1.10.** Решение неравенства

```
program solve_inequality;
function largest_power(a, b: LongInt): Word;
var
  n: Word;
  x: LongInt;
begin
  x := b;
  n := 0;
  while x <= a do
  begin
    x := b*x;
    Inc(n);
  end;
  largest_power := n;
end;
begin
  WriteLn('3^n <= 10000 < 3^(n+1)');
  WriteLn('n = ', largest_power(10000, 3));
  Write('Нажмите <Enter>:');
  ReadLn;
end.
```

В следующем примере запрограммировано вычисление корня функции методом деления отрезка пополам. Напомню, что корень функции  $F(x)$  — это такое значение ее аргумента  $x^*$ , при котором выполняется условие  $F(x^*) = 0$ . Известно, что для решения такого уравнения надо вначале задать интервал  $[a, b]$ , на котором будет искаться решение. Если решение действительно существует, принадлежит заданному интервалу и является на этом интервале единственным, функция  $F(x)$  принимает на границах интервала значения противоположных знаков. Другими словами, произведение значений функции на границах интервала отрицательно:  $F(a)F(b) < 0$ . Далее исходный интервал делится средней точкой  $c = (a+b)/2$  на две равные части, из которых выбирается лишь та, которая содержит решение уравнения. Процедура деления отрезка пополам повторяется до тех пор, пока корень функции не будет найден с заданной точностью. Оценкой погрешности в данном случае может быть величина последнего интервала  $|a-b|$  или значение  $|F(x)|$ . Последний критерий используется в программе, приведенной в листинге 1.11.

**Листинг 1.11.** Решение нелинейного уравнения методом деления отрезка пополам

```
{$N+}
program bisection;

type
  real_function = function (x: Real): Real;

function G(x: Real): Real; far;
begin
  G := Sqr(x) - 2.0;
end;

function zero(F: real_function; x, y: Real): Real;
```

```

const
  eps = 1.0e-12;

var
  mid, fx, fy, fm: Real;

begin
  fx := F(x); fy := F(y);
  if fx * fy > 0.0 then
    Halt: {Прекращение работы подпрограммы,
           если на выбранном участке нет корня}
  repeat
    if Abs(fx) < eps then
      begin
        zero := x;
        Exit;
      end
    else if Abs(fy) < eps then
      begin
        zero := y;
        Exit;
      end
    else
      begin
        mid := 0.5 * (x + y); fm := F(mid);
        if fx*fm <= 0 then
          begin
            y := mid;
            fy := fm;
          end
        else
          begin
            x := mid;
            fx := fm;
          end;
        end
      until false;
  end;

  begin
    WriteLn('G(', zero: 1.0, 2.0):10:10, ') = 0');
    ReadLn;
  end.

```

Остановимся подробнее на этой программе. Вторая строка содержит оператор `type`, разговор о котором пойдет позже. Сейчас же скажу, что этот оператор позволяет программисту ввести свой собственный тип. Слева от знака равенства записывается название нового типа данных, а справа задается сам этот тип. В нашем случае это функциональный тип. В третьей строке программы используется новый для нас элемент — ключевое слово `far`. Это директива компилятора, которая используется в описаниях подпрограмм, передаваемых в качестве параметров другим подпрограммам. Такой подпрограммой в нашем случае является подпрограмма-функция `G`. Она используется в качестве аргу-

мента другой подпрограммы-функции `zero`. В функции `zero` запрограммировано вычисление корня произвольной непрерывной функции, имеющей на границах исходного интервала противоположные знаки. Использование функции `G` в качестве аргумента подпрограммы-функции `zero` делает программу более универсальной, ведь для решения другого нелинейного уравнения достаточно будет подставить в данную программу описание другой функции.

Оператор цикла `repeat...until...` в программе сделан бесконечным, так как условие завершения — просто логическая константа `false`. Если окажется, что корень функции найден с заданной точностью, будет выполняться условный оператор, следующий сразу же за заголовком цикла, а его выполнение завершится вызовом процедуры `Exit`. Эта процедура завершает выполнение функции. Если условие противоположности знаков нарушено, процедура `Halt` остановит выполнение программы. В этом случае у функции на заданном интервале нет корня.

## ВВОД И ВЫВОД

Компьютер является устройством, которое принимает данные, обрабатывает их и выводит результат. Для ввода и вывода информации используются файлы. При выполнении программы на Паскале автоматически открываются два файла со специальными именами `Input` и `Output`. Файл `Input` обеспечивает ввод символов с клавиатуры, а файл `Output` — вывод символов и графических элементов на экран. Клавиатура и экран являются стандартными устройствами ввода/вывода. Обращение к файлам `Input` и `Output` происходит автоматически, без дополнительных усилий программиста. Процедуры ввода `Read` и `ReadLn` обычночитывают ввод с клавиатуры, а процедуры вывода `Write` и `WriteLn` обычно посыпают вывод на экран. У компьютера могут быть и другие устройства ввода и вывода: дисководы, принтер, последовательный порт и т. д. К каждому из этих устройств можно обратиться из программы путем определения соответствующих файлов. При сопоставлении файлов устройствам используются обычные имена, принятые в MS-DOS, например `prn` — принтер, `com1` — последовательный порт, `c:\users\ivanov\res.dat` — файл на жестком диске и т. д.

Следующие две программы предназначены для вывода таблицы значений функции  $F(x) = x/(1+x)$  на экран и в файл на диске. В программе 1.12 вывод производится на экран.

**Листинг 1.12.** Вывод таблицы значений функции на экран

```
program table_of_values;
var
  x: Real;
  k: Word;

function F(x: Real): Real;
begin
  F := x/(1.0 + x);
```

```

end;

begin
  x := 0.0;
  WriteLn('Таблица значений функции F(x) = x/(1 + x)');
  WriteLn;
  WriteLn('x':10, 'F(x)':20);
  WriteLn;

  for k := 0 to 50 do
  begin
    WriteLn(x:10:4, F(x):20:10);
    x := x + 0.1;
    if k mod 10 = 9 then ReadLn;
  end;

  WriteLn;
  Write('Нажмите <Enter>');
  ReadLn;
end.

```

Обратите внимание на то, что программа приостанавливает вывод через каждые 10 строк и ожидает нажатия клавиши Enter. Для этого при каждом проходе цикла вычисляется остаток от деления (операция `mod`) счетчика строк (переменная `k`) на 10. В операторах вывода после элементов вывода идут целые беззнаковые значения, разделяемые двоеточиями. Эти значения задают формат вывода. Первое значение, отделяемое двоеточием от элемента вывода (идентификатора переменной), определяет количество позиций в строке, отводимое под данное значение. Второе число, если оно присутствует, определяет количество позиций, отводимых под *мантиссу* числа (его дробную часть).

В программе 1.13 вывод этой же таблицы значений направляется в файл.

#### Листинг 1.13. Вывод таблицы значений функции в файл

```

program file_of_values;
var
  x : Real;
  k : Word;
  out_file : Text; {тип Text соответствует файлу,
                     состоящему из строк текста}

function F(x: Real): Real;
begin
  F := x/(1.0 + x);
end;

begin
  Assign(out_file, 'c:\user\ivanov\table.dat');
  Rewrite(out_file);
  x := 0.0;
  WriteLn(out_file, 'Таблица значений функции F(x) = x/(1 + x)');
  WriteLn(out_file);
  WriteLn(out_file, 'x':9, 'F(x)':19);

```

```

WriteLn(out_file);
for k := 0 to 50 do
begin
  WriteLn(out_file, x:9:3, F(x):19:9);
  x := x + 0.1;
  if k mod 10 = 9 then WriteLn(out_file);
end;
Close(out_file);
end.

```

Здесь `Text` обозначает тип файла, состоящего из символов, организованных в строки. Такой файл создается обычным текстовым редактором, содержит текст и называется текстовым файлом. Процедура `Assign` модуля `System` связывает специальную файловую переменную `out_file` с файлом на диске, имеющим то имя, которое задает программист. В нашем случае это файл `table.dat`, который находится в каталоге `user\ivanov` на диске С:. В этот момент файловая переменная только связывается с именем файла — ничего больше не происходит. Процедура `Rewrite` открывает файл для записи и, если файл `table.dat` уже существует на диске, удаляет его содержимое. Об этом не следует забывать! В дальнейшем процедура `Close` закрывает файл, выполнив предварительно его полное обновление, и размещает его на диске.

Обратите особое внимание на то, как используется оператор `WriteLn`, когда вывод направляется в файл, отличный от задаваемого по умолчанию файла `Output`. Процедура `WriteLn` в общем случае имеет вид

```
WriteLn(par_1, par_2, ... , par_n);
```

с произвольным числом параметров `par_i`. Если первым параметром является файловая переменная, то все остальные значения будут записываться в соответствующий файл, в противном случае они будут выводиться на экран (то есть на стандартное устройство вывода).

Следующая программа предназначена для ввода данных в программу из внешнего текстового файла `table.dat`, созданного программой `file_of_values`, и вывода их на принтер.

#### Листинг 1.14. Ввод данных из внешнего файла

```

program get_file;

const
  FF = #12; {Код управляющего символа «прогон страницы»}
var
  in_file, out_file: Text;
  k, count: Word;
  ss: string [80];

begin
  Assign(in_file, 'c:\user\ivanov\table.dat');
  Reset(in_file);
  Assign(out_file, 'prn');
  Rewrite(out_file);

```

```

count := 0;

{Вывод на печать пустых строк}
for k := 1 to 3 do
    WriteLn(out_file);

{В следующем цикле считывается из входного файла
и выводится на печать заголовок таблицы (4 строки)}
for k := 1 to 4 do
begin
    ReadLn(in_file, ss);
    WriteLn(out_file, ss);
end;
count := 0;

{Считывание из внешнего файла
и вывод на печать таблицы значений}
while not Eof(in_file) do
begin
    ReadLn(in_file, ss);
    WriteLn(out_file, ss);
    Inc(count);
    if count = 44 then
begin
        count := 0;
        Write(out_file, FF);
        for k := 1 to 4 do
            WriteLn(out_file);
    end;
end;
Write(out_file, FF);
Close(out_file);
Close(in_file);
end.

```

Считывание файла происходит в цикле `while`. Данный цикл выбран потому, что он допускает крайний случай — если файл пуст, то цикл не выполняется вообще. Функция `Eof` модуля `System` возвращает значение `true`, если текущая позиция в файле находится за последним его элементом. Таким образом, операторы цикла будут выполняться до тех пор, пока мы не доберемся до конца файла `in_file`.

Зарезервированное слово `string` определяет строковые переменные. Объявление `ss` как переменной типа `string[80]` означает, что `ss` является строкой, содержащей до 80 символов. Без параметра `[n]` максимально допустимая длина строки по умолчанию составляет 255 символов. Таким образом, предложение

```
var SS: string;
```

объявляет `ss` строкой, длина которой может быть до 255 символов.

В этой программе через каждые 45 строк в текст вставляется специальный управляющий символ «прогон страницы». Код этого символа содержится в константе `FF`.

**ВНИМАНИЕ**

Каждый файл, открытый программой, следует закрывать процедурой `Close`. Это защитит внешние файлы от аварийных ситуаций. Можно сказать, что это — признак хорошего тона в программировании.

## Модули

*Библиотечный модуль* содержит описания и подпрограммы, которые могут использоваться в различных программах. Подпрограмму имеет смысл включить в состав модуля в том случае, когда она реализует действие, которое приходится выполнять достаточно часто. Такую подпрограмму можно написать и отладить один раз, а использовать многоократно. Это позволяет ускорить процесс разработки программного обеспечения. Файл, содержащий модуль, обязан иметь имя, совпадающее с именем модуля.

Разберем в качестве примера модуль с описаниями гиперболических функций. Напомню их определение:

$$\begin{aligned}\sinh(x) &= (e^x - e^{-x})/2, \\ \cosh(x) &= (e^x + e^{-x})/2, \\ \tanh(x) &= \sinh(x)/\cosh(x).\end{aligned}$$

Гиперболических функций нет в числе встроенных функций языка Паскаль, но эти функции достаточно часто появляются в прикладных задачах, и поэтому имеет смысл включить их в состав библиотечного модуля. Доступ к функциям из этого модуля обеспечивает оператор использования `uses`, в котором указывается имя модуля. Итак, сам модуль выглядит следующим образом.

**Листинг 1.15.** Модуль с гиперболическими функциями

```
{$N+}
unit hyp_fun;

interface

type
  Float = Extended;

function sinh(x: Float): Float;
function cosh(x: Float): Float;
function tanh(x: Float): Float;

implementation

var
  t: Float;
```

```

function sinh(x: Float): Float;
begin
  t := Exp(x);
  sinh := 0.5*(t - 1.0/t);
end;

function cosh(x: Float): Float;
begin
  t := Exp(x);
  cosh := 0.5*(t + 1.0/t);
end;

function tanh(x: Float): Float;
begin
  t := Exp(2.0*x);
  tanh := (t - 1.0) / (t + 1.0);
end;

end.

```

Зарезервированные слова `interface` и `implementation` здесь играют важную роль. Каждый модуль имеет части (секции), озаглавленные этими словами. Секция `interface` (она называется интерфейсной секцией) содержит описания констант, типов, переменных и процедур, доступных из вызывающей программы или модуля. Секция `implementation` (секция реализации) содержит исходный код подпрограмм. Она может также содержать локальные описания, такие как `var t: Real;` из нашего примера.

Каждый модуль начинается с зарезервированного слова `unit` и заканчивается словом `end`, за которым следует точка. Для этого `end` не требуется соответствующего слова `begin`, хотя можно и поставить его непосредственно перед `end`. Оператор `type` в начале нашего модуля определяет тип `Float`, который в данном случае эквивалентен типу `Extended`. Указав справа от знака равенства любой другой вещественный тип, можно изменить точность вычисления гиперболических функций.

В следующей программе используется только что рассмотренный модуль `hyp_fun`. Проверьте результат ее работы с помощью калькулятора.

#### Листинг 1.16. Пример использования модуля `hyp_fun`

```

{$N+}
program test_hyperbolic_funs;
uses
  CRT, hyp_fun;
begin
  ClrScr;
  WriteLn('sinh( 0.5 ) =', sinh(0.5));
  WriteLn('cosh(-0.5) =', cosh(-0.5));
  WriteLn('tanh( 1.5 ) =', tanh(1.5));
  Write('Нажмите <Enter>: ');

```

```
ReadLn;
end.
```

 **ПРИМЕЧАНИЕ**

Результатом компиляции модуля `xuz.pas` с заголовком `unit xuz` будет файл `xuz.tpu`.

## Упражнения

В упражнениях 1.1–1.8 приведены только разделы операторов программы и предполагается наличие в программе следующих описаний:

```
var
  i, j, k, m, n: Word;
  a, b, c: LongInt;
  s, x, y, z: Real;
```

Не прибегая к помощи компьютера, определите результат выполнения каждой программы.

### Упражнение 1.1

```
begin
  n := 0; x := 1.0;
  repeat
    Inc(n);
    x := 2.0 * x;
  until x > 1.0e5;
  WriteLn(n, ' ', x);
  ReadLn;
end.
```

### Упражнение 1.2

```
begin
  s := 0.0; n := 1;
  while n <= 100 do
  begin
    x := 3.0 * n + 2.0; x := 1.0/x;
    s := s + x;
    Inc(n);
  end;
  WriteLn(n, ' ', s);
  ReadLn;
end.
```

### Упражнение 1.3

```
begin
```

Упражнения

```
s := 0; n := 0;
while n < 100 do
begin
    Inc(n); x := 3 * n + 2;
    x := 1.0/x; s := s + x;
end;
WriteLn(n, ' ', s);
ReadLn;
end.
```

**Упражнение 1.4**

```
begin
n := 25; a := 1; b := 1;
for j := 3 to n do
begin
    c := b; b := a + b; a := c;
end;
WriteLn('F', n, ' = ', b);
ReadLn;
end.
```

**Упражнение 1.5**

```
begin
n := 25; a := 1; b := 1;
for j := 2 to n do
begin
    b := a + b; a := b;
end;
WriteLn('F(', n - 1, ') = ', b);
ReadLn;
end.
```

**Упражнение 1.6**

```
begin
x := Pi; y := 0.0;
for n := 1 to 20 do y := y * x + n;
WriteLn('G', x, ' ) = ', y);
ReadLn;
end.
```

**Упражнение 1.7**

```
begin
c := 2; s := 0;
for n := 1 to 99 do
begin
    c := 6 - c; y := n/(100 + n); s := s + c * y
end;
s := s + 0.5;
WriteLn('I = ', s/300.0);
ReadLn;
end.
```

**Упражнение 1.8**

```

begin
  for k := 1 to 100 do
begin
  j := 2 + k * k; m := 1; n := 1;
  while n <= j do
begin
  if n = j then
    WriteLn('(', k, ', ', m, ')');
    Inc(m); n := m * m * m;
  end;
end;
  ReadLn;
end.

```

В упражнениях 1.9–1.13 вам предлагается написать программы для решения соответствующих задач.

**Упражнение 1.9**

В настоящее время используются пять температурных шкал. Это шкалы Цельсия, Фаренгейта, Кельвина, Ренкина и международная термодинамическая шкала температур. Наиболее популярная из них — это шкала Цельсия, официально утвержденная в качестве международной шкалы в 1950 году. Шкала Фаренгейта используется в англоязычных странах. Опорные точки для этих шкал приведены в комментарии к программе 1.2. В шкале Кельвина за точку отсчета принят абсолютный ноль — это  $-273,15^{\circ}\text{C}$ , или  $-459,67^{\circ}\text{F}$ . В шкале Ренкина отсчет температуры также идет от абсолютного нуля, а один градус равен градусу по Фаренгейту. Точка замерзания воды по шкале Ренкина —  $492^{\circ}\text{R}$ , а точка ее кипения —  $672^{\circ}\text{R}$ . И, наконец, в 1933 году ученые приняли международную шкалу температур, в которой используются дополнительные опорные точки.

Дополните программу 1.2 таким образом, чтобы она выводила таблицу соответствия между температурными шкалами Цельсия, Фаренгейта, Кельвина и Ренкина.

**Упражнение 1.10**

Найдите все целочисленные решения неравенства  $x^2 - 4xy + y^2 < 100$ .

**Упражнение 1.11**

Выведите на экран таблицу квадратов целых чисел от 0 до 999. Таблица должна состоять из 100 строк по 10 значений в каждой строке.

**Упражнение 1.12**

Выполните таблицу из упражнения 11 в файл.

**Упражнение 1.13**

Напишите программу для печати таблицы из упражнения 11 на принтере.

## Что нового мы узнали?

- Познакомились с алфавитом языка и основными зарезервированными словами.
- Узнали, как устроена программа на Паскале.
- Познакомились с основными операторами языка.
- Познакомились с подпрограммами-функциями и модулями.
- Познакомились с программированием ввода и вывода данных на Паскале.

## **УРОК**

# Программирование на языке Паскаль

- 
- Процедуры**
  - Предопределенные типы переменных**
  - Символьный тип**
  - Строковый тип**
  - Массивы**
  - Множества**
  - Целый и логический (булев) типы**
  - Вещественные типы**
-

**В** этом уроке мы познакомимся с программированием процедур. Процедуры, как и функции, являются важнейшей составной частью языка, так как они обеспечивают модульное программирование. Затем мы разберем предопределенные типы переменных Паскаля — целые, вещественные, символьный, строковый, массив и множество. Эти типы являются основными и могут использоваться при определении более сложных типов, задаваемых пользователем.

## Процедуры

В первом уроке мы познакомились с подпрограммами-функциями, создаваемыми с помощью ключевого слова `function`. Подпрограмму-функцию можно определить как фрагмент программы или модуля, начинающийся заголовком `function`, имеющий раздел описаний и раздел операторов, заканчивающийся зарезервированным словом `end` и возвращающий при вызове единственное значение, носителем которого является имя функции. Понятно, что возможности функций ограничены. Ведь иногда результат работы подпрограммы — это целый *набор* значений, а иногда результат ее выполнения не сводится к вычислениям.

Подпрограммы, создаваемые с помощью ключевого слова `procedure`, могут выполнять все виды действий, включая вычисление одного или более результатов. Процедура похожа на функцию. Это фрагмент программы (или модуля), начинающийся заголовком `procedure`, имеющий разделы описаний и операторов и завершающийся словом `end`. Заголовок процедуры имеет следующий вид:

```
procedure name_of_procedure(list_of_parameters);
```

Здесь `list_of_parameters` представляет собой список параметров процедуры. Обмен данными между процедурой и вызывающей программной единицей производится с помощью параметров. Имя процедуры, в отличие от функции, уже не является носителем результата, поэтому тип процедуры в заголовке не описывается.

В следующей программе используются процедуры. Задача состоит в том, чтобы вычислить суммарное количество секунд, соответствующее заданному числу часов, минут и секунд, и, наоборот, определить, сколько часов, минут и секунд содержится в заданном числе секунд.

**Листинг 2.1.** Программа пересчета часов, минут и секунд в секунды и обратно

```
program time_conversion;
uses crt;
var
  choice: integer;
procedure Menu;
begin
  writeln('1. Преобразовать часы, минуты и секунды в секунды');
  writeln('2. Преобразовать секунды в часы, минуты и секунды');
  writeln('3. Завершить работу');
  writeln;
  writeln('Введите номер (1-3)');
end;

procedure seconds_to_time;
var
  total_seconds: LongInt;
  hours, minutes, seconds: LongInt;
  temp: LongInt;
begin
  ClrScr;
  WriteLn('Введите суммарное количество секунд:');
  ReadLn(total_seconds);
  WriteLn;
  temp := total_seconds div 60;
  seconds := total_seconds mod 60;
  hours := temp div 60;
  minutes := temp mod 60;
  WriteLn;
  WriteLn(total_seconds, ' секунд - это');
  WriteLn;
  WriteLn(hours, ' часов, ', minutes, ' минут, ', seconds, ' секунд');
  WriteLn;
  WriteLn('Для продолжения работы нажмите <ENTER>');
  ReadLn;
end;

procedure time_to_seconds;
var
  total_seconds: LongInt;
  hours, minutes, seconds: LongInt;
begin
  ClrScr;
  WriteLn('Введите часы: ');
  ReadLn(hours);
  WriteLn;
  WriteLn('Введите минуты: ');
  ReadLn(minutes);
```

```

WriteLn;
WriteLn('Введите секунды: ');
ReadLn(seconds);
WriteLn;
total_seconds := hours * 3600 + minutes * 60 + seconds;
WriteLn;
WriteLn(hours, ' часов, ', minutes, ' минут, ', seconds,
       ' секунд это ', total_seconds, ' секунд');
WriteLn;
WriteLn('Для продолжения работы нажмите <ENTER>');
ReadLn;
end;

begin
  choice := 0;
  while choice <> 3 do
  begin
    ClrScr;
    Menu;
    ReadLn(choice);
    case choice of
      1: time_to_seconds;
      2: seconds_to_time;
    end;
  end;
end.

```

В программе используется текстовое меню с возможностью выбора трех вариантов работы. Вариант задается пользователем путем ввода с клавиатуры соответствующего целого значения. Поскольку число вариантов — три, условный оператор `if...then...else...` здесь не подходит, поэтому в программе используется оператор выбора `case`.



#### **ВНИМАНИЕ**

Вызов процедуры производится путем указания ее имени со списком параметров, если таковой имеется.

В данной программе используются процедуры `ClrScr` из модуля `Crt` и процедуры ввода/вывода из модуля `System`, а также две собственные процедуры `time_to_seconds` и `seconds_to_time`.

## **Область видимости идентификаторов**

Мы уже знаем, что функции и процедуры имеют собственные разделы описаний. Все переменные (и прочие объекты, используемые в программе), описанные в подпрограмме, являются *локальными* и действуют только внутри этой подпрограммы. Никакой связи между ними и объектами вызывающей программы, имеющими (возможно, случайно) такие же имена (идентификаторы), нет. Они полностью независимы.

С другой стороны, в подпрограмме можно использовать идентификаторы, описанные только в вызывающей программе (но не в самой подпрограмме). Смысл и значение этих идентификаторов одинаковы и там, и там. Такие идентификаторы называются *глобальными*. Область действия описания конкретного идентификатора называется еще его *областью видимости*.

Сразу же замечу, что использования глобальных переменных в подпрограммах следует избегать. Тому есть несколько причин. Одна заключается в том, что подпрограмма, использующая глобальные переменные, становится менее универсальной, чем замкнутая, «самодостаточная» подпрограмма. При ее переносе в другую программу придется тщательно проследить за обменом данными между программными единицами с помощью глобальных переменных. Вторая причина связана с тем, что при использовании глобальных переменных возрастает риск ошибок, подчас трудно обнаружимых, вызванных «несанкционированным» или неучтенным изменением значения глобальной переменной в теле подпрограммы.

Такие процедуры (или функции) могут иметь неожиданные побочные эффекты. Программа `side_effect` показывает, как это может произойти. В ней подпрограмма-функция `f` делает то, что ей положено делать, и, кроме того, «тайком» изменяет значение глобальной переменной `m`. Вследствие этого результат выполнения программы внешне выглядит несколько странно. Но в данном случае найти ошибку не составляет труда, так как программа небольшая, а в реальной много-трудной программистской жизни на поиск такой ошибки может уйти довольно много времени.

**Листинг 2.2.** Программа, демонстрирующая побочные эффекты, связанные с глобальными переменными

```
program side_effect;

var
  m: Integer;

function f(n: Integer): Integer;
begin
  f := Sqr(n) + 5;
  m := m + 1;
end;

begin
  m := 2;
  WriteLn('Произведение меняется при перестановке сомножителей:');
  WriteLn(' m * f(4) = ', m, ' * ', f(4));
  WriteLn(' f(4) * m = ', f(4), ' * ', m);
  Write('Нажмите <Enter>: ');
  ReadLn;
end.
```

Если в подпрограмме описаны другие процедуры или функции, то область видимости описанных в ней переменных распространяется на вложенные подпрограммы, если в них не описаны переменные с такими же именами.

## Формальные и фактические параметры

В заголовке процедуры или функции, как мы уже знаем, может содержаться список параметров. Параметры представляют собой идентификаторы переменных и служат для обмена значениями между подпрограммой и вызывающей ее программной единицей. При этом в описании подпрограммы, поскольку оно включается в текст программы один раз, имена параметров выбираются определенным образом и безотносительно к именам переменных, используемых в других частях программы. Такие параметры, имена которых указаны в заголовке подпрограммы, называются *формальными*.

С другой стороны, при каждом новом обращении к подпрограмме в нее могут передаваться значения разных переменных. Такие переменные, имена которых подставляются в оператор вызова подпрограммы при фактическом обращении к ней, называются *фактическими* параметрами.



### ВНИМАНИЕ

При вызове процедуры или функции количество и тип фактических параметров должны соответствовать количеству и типу формальных параметров. Из последнего, впрочем, есть исключения.

## Параметры-значения, параметры-переменные и нетипизированные параметры

В Турбо Паскале имеется три типа параметров: *параметры-значения*, *параметры-переменные* и *нетипизированные параметры*.

Формальный параметр-значение при вызове подпрограммы получает свое начальное значение путем копирования соответствующего ему фактического параметра. При изменении формального параметра-значения фактический параметр *не меняется*.

Фактическое значение параметра-значения, соответствующее формальному, должно быть выражением, а его тип должен быть совместим по присваиванию (о совместимости типов мы поговорим позже) с типом формального параметра-значения. Перед списком параметров-значений в заголовке подпрограммы отсутствует какое-либо специальное ключевое слово, а за ним следует указание типа. Вот пример списка параметров-значений:

```
function Compare(first, second: Word): Boolean;
```

Параметр-переменная используется в том случае, когда значение должно передаваться из процедуры или функции вызывающей программе. Соответствующий фактический параметр в операторе вызова процедуры или функции должен быть *ссылкой на переменную*. При вызове процедуры или функции формальный параметр-переменная замещается фактической переменной. При любых изменениях значения формального параметра-переменной изменится и фактический параметр.

**ВНИМАНИЕ**

Тип фактического параметра должен совпадать с типом формального параметра-переменной.

В заголовке подпрограммы перед списком параметров-переменных следует ключевое слово `var`, а после списка указывается тип. Вот пример списка параметров-переменных:

```
function Compare(var first, second: Word): Boolean;
```

И наконец, если формальный параметр является нетипизированным параметром-переменной, то соответствующий ему фактический параметр может представлять собой любую ссылку на переменную, независимо от ее типа. В заголовке подпрограммы группа параметров, перед которыми стоит ключевое слово `var` и за которыми не следует указание типа, является списком нетипизированных параметров-переменных. Вот пример списка нетипизированных параметров:

```
function Compare(var first, second): Boolean;
```

## Пример программы

Вернемся к проблеме измерения времени. При отладке и оптимизации программы иногда оказывается полезным определение времени выполнения программы в целом или отдельных ее частей. Такое исследование называют *профилированием* программы. Известно, что любой компьютер имеет встроенные часы. Эти часы измеряют время в сотых долях секунды и их можно использовать для измерения времени выполнения программы. Библиотечный модуль DOS содержит процедуру `GetTime`, позволяющую считывать показания системных часов. На основе данного модуля разработан модуль `timer`, который содержит три процедуры. Первые две — `get_time_start` и `get_time_stop` — позволяют избавиться от необходимости запоминания формы обращения к процедуре `GetTime`. Честно признаюсь, что здесь не соблюдаются ранее данные рекомендации. Процедура `compute_elapsed_time` намеренно производит побочный эффект. Она изменяет значения глобальных переменных секции реализации этого модуля, таких, например, как `hour1`, `min1`, `sec1`. Связано это с необходимостью учета изменения показаний часов в полночь.

Процедура `put_time` обеспечивает отформатированный вывод результатов. Она включает в себя две другие процедуры (`format` и `compute_elapsed_time`), одна из которых (`format`), в свою очередь, также содержит процедуру (`print`). Наличие этих процедур позволяет избежать дублирования кода.

**Листинг 2.3.** Модуль timer

```
unit timer;

interface

procedure get_time_start; {Определяет время начала работы}
procedure get_time_stop; {Определяет время окончания работы}
procedure put_time; {Процедура вывода результата}

implementation

uses DOS; {Модуль DOS содержит процедуру GetTime}

var
    hour_start, min_start, sec_start, hund_start,
    hour_stop, min_stop, sec_stop, hund_stop,
    hour, min, sec, hund: Word;

procedure get_time_start;
begin
    GetTime(hour_start, min_start, sec_start, hund_start);
end;

procedure get_time_stop;
begin
    GetTime(hour_stop, min_stop, sec_stop, hund_stop);
end;

procedure put_time;
procedure format(hour, min, sec, hund: Word);
procedure print(w: Word);
begin
    if w < 10 then Write('0');
    Write(w);
end:{print}

begin
    print(hour); Write(':');
    print(min); Write(':');
    print(sec); Write('.');
    print(hund);
    WriteLn;
end:{format}

procedure compute_elapsed_time;
begin
    {Учет изменения отсчета часов при переходе через полночь}
    if hour_stop < hour_start then Inc(hour_stop, 24);
    {Теперь hour_stop >= hour_start}
```

```
if hour_stop > hour_start then
begin
  Dec(hour_stop);
  Inc(min_stop, 60);
end;
{Теперь hour_stop >= hour_start, min_stop >= min_start}
if min_stop > min_start then
begin
  Dec(min_stop);
  Inc(sec_stop, 60);
end;
{Теперь hour_stop >= hour_start, min_stop >= min_start,
sec_stop >= sec_start}
if sec_stop > sec_start then
begin
  Dec(sec_stop);
  Inc(hund_stop, 100);
end;
{Теперь hour_stop >= hour_start, ..., hund_stop >= hund_start}
hour := hour_stop - hour_start;
min := min_stop - min_start;
sec := sec_stop - sec_start;
hund := hund_stop - hund_start;
if hund >= 100 then
begin
  Dec(hund, 100);
  Inc(sec);
end;
if sec >= 60 then
begin
  Dec(sec, 60);
  Inc(min);
end;
if min >= 60 then
begin
  Dec(min, 60);
  Inc(hour);
end;
end:{compute_elapsed_time}

begin
  {Вывод результатов}
  Write('Момент остановки: ');
  format(hour_stop, min_stop, sec_stop, hund_stop);
  Write('Момент начала работы:');
  format(hour_start, min_start, sec_start, hund_start);
  compute_elapsed_time;
  Write('Затраченное время:');
  format(hour, min, sec, hund);
end:{put_time}

end.
```

Процедура `format` имеет локальные переменные с теми же именами, что и переменные, объявленные вне этой процедуры, но это *разные* переменные. Ее параметры — это параметры-значения типа `Word`.

Сделаю еще несколько замечаний по поводу различий между параметрами-значениями и параметрами-переменными. Предположим, что речь идет о программировании процедуры вычисления суммы двух значений. На входе у нее два числа  $F$  и  $G$ , а результатом является их сумма  $H = F + G$ . Допустимы по крайней мере два варианта описания этих значений в качестве параметров процедуры:

```
procedure sum (F, G: Extended; var H: Extended);
procedure sum (var F, G, H: Extended);
```

Допустим, что используется первое описание. В этом случае процедура `sum` может вызываться таким образом, что в качестве  $F$  и  $G$  подставляются *выражения*, а в качестве  $H$  — имя переменной. Кроме того, это описание подразумевает, что для внешних параметров, которые подставляются при вызове `sum` в качестве  $F$  и  $G$ , создаются *копии*. Это дополнительные затраты времени и места в памяти. В нашем простом примере такими дополнительными затратами можно пренебречь, однако при решении больших вычислительных задач затраты могут быть довольно значительными. Таким образом, второе описание может оказаться предпочтительнее в смысле как скорости выполнения программы, так и ее объема.

Существует еще один тип параметров в процедурах и функциях — *параметры-константы*. Вот пример их использования:

```
procedure sum (const F, G: Extended; var H: Extended);
```

Для параметров-констант копии не создаются, поэтому их использование в определенных ситуациях позволяет сэкономить память. Значения таких параметров не могут изменяться в теле процедуры. Это обеспечивает некоторую безопасность по отношению к случайным присваиванием значений параметров.

Следует придерживаться правила: фактические переменные, замещающие несколько параметров-переменных, должны быть различными. Дело в том, что передача параметров в этом случае производится *по ссылке* (а не по значению), поэтому возникает ситуация, при которой различные формальные параметры указывают на одну и ту же область памяти, что может служить источником трудно выявляемых ошибок, связанных с косвенным изменением значений переменных.

## Упражнения

Используя модуль `timer`, выполните следующие задания.

### Упражнение 2.1

Сравните скорость выполнения циклов `for..to..do..`, `while..do..` и `repeat..until..`

### Упражнение 2.2

Сравните скорость выполнения операции сложения для данных типа `Integer`, `Word`, `LongInt` и `Extended`.

**Упражнение 2.3**

Сравните скорость выполнения операции умножения для данных типа Integer, Word, LongInt и Extended.

**Упражнение 2.4**

Сравните скорость выполнения операций div и (/) для данных типа Integer, Word и LongInt.

**Упражнение 2.5**

Сравните время выполнения операторов Inc(n) и  $n := n + 1$  для переменной n, относящейся к одному из целых типов.

**Упражнение 2.6**

Используя процедуру GetDate, установите для одной из ваших программ 15-дневный «пробный» период использования, по окончании которого программа перестанет запускаться.

## Предопределенные типы переменных

Все имеющиеся в Паскале типы принято делить на группы. Типы, принадлежащие одной группе, имеют определенное сходство. Прежде всего выделяют *простые* и *структурные* типы. Простые типы, в свою очередь, подразделяются на *порядковые* и *вещественные* типы. В табл. 2.1 приведена классификация *предопределенных* типов Паскаля. Предопределенные типы «встроены» в Паскаль в отличие от типов, задаваемых программистом.

Порядковые типы называются так потому, что их допустимые значения представляют собой множество, состоящее из конечного числа элементов. В этом множестве есть первый и последний элементы. Кроме того, каждый элемент порядкового типа имеет предшествующий ему и следующий за ним элементы. Так, например, у целого значения 12 есть предшественник (значение 11) и преемник (значение 13). Очевидным исключением являются первый (у него нет предшественника) и последний (нет преемника) элементы. Элементы порядкового типа можно пронумеровать, расположив их в определенном порядке, например по возрастанию.

В отличие от порядковых, вещественные типы представляют вещественные числа (числа, имеющие как целую, так и дробную части), а множество вещественных чисел даже из ограниченного диапазона пронумеровать невозможно. Строго говоря, следует учитывать то обстоятельство, что компьютер может оперировать лишь с *конечным* набором чисел. Это связано с его конечной *разрядностью* (то есть количеством двоичных разрядов, отводимых под хранение данных). Но для

вещественных чисел количество их допустимых значений весьма велико, поэтому не имеет смысла относить их к группе порядковых типов.

**Таблица 2.1.** Классификация предопределенных типов языка Паскаль

Группа	Подгруппа	Название	Идентификатор
Простой	Порядковый	Корткий целый Байтовый Слово Целый Длинный целый Символьный Булев	ShortInt Byte Word Integer LongInt Char Boolean
	Вещественный	Вещественный С одинарной точностью С двойной точностью С повышенной точностью Сложный	Real Single Double Extended Comp
	Строковый		String
	Структурный	Массив Множество Файл Запись	Array Set File Record
	Сылочный		Pointer
	Процедурный	Процедура Функция	Procedure Function
	Объектный		Object

В табл. 2.2 приводятся диапазоны допустимых значений порядковых типов Паскаля.

**Таблица 2.2.** Порядковые типы языка Паскаль

Идентификатор	Описание типа	Множество допустимых значений
Shortint	8-битный целый со знаком	-128..127
Integer	16-битный целый со знаком	-32768..32767
Longint	32-битный целый со знаком	-2147483648..2147483647
Byte	8-битный целый без знака	0..255
Word	16-битный целый без знака	0..65535
Boolean	Логический	False, True
Char	Символьный	Символы из расширенного набора символов кода ASCII

В Паскале имеется процедура `Ord`, которая вычисляет порядковый номер аргумента, если он имеет порядковый тип. Так, например,

```
Ord(false) = 0
Ord(true) = 1
Ord(A) = 65
Ord(Z) = 90
Ord(a) = 97
Ord(z) = 122
```

В Паскале имеются пять видов вещественных типов. Вещественные типы различаются диапазоном и точностью связанных с ними значений. Эти типы перечислены в табл. 2.3. Действия над типами с одинарной точностью, с двойной точностью и с повышенной точностью, а также над сложным типом могут выполняться только при наличии математического сопроцессора.

Таблица 2.3. Вещественные типы языка Паскаль

Идентификатор	Описание типа	Диапазон
Real	6-байтовый вещественный, 11–12 значащих цифр	$-1.7 \times 10^{38} \dots -2.9 \times 10^{-39}$ , $2.9 \times 10^{-39} \dots 1.7 \times 10^{38}$
Single	4-байтовый вещественный с одинарной точностью, 7–8 значащих цифр	$-3.4 \times 10^{38} \dots -1.5 \times 10^{-45}$ , $1.5 \times 10^{-45} \dots 3.4 \times 10^{38}$
Double	8-байтовый вещественный с двойной точностью, 15–16 значащих цифр	$-1.7 \times 10^{308} \dots -5.0 \times 10^{-324}$ , $5.0 \times 10^{-324} \dots 1.7 \times 10^{308}$
Extended	10-байтовый вещественный с повышенной точностью, 19–20 значащих цифр	$-1.1 \times 10^{4932} \dots -1.9 \times 10^{-4951}$ , $1.9 \times 10^{-4951} \dots 1.1 \times 10^{4932}$
Comp	8-байтовый сложный тип	$-2^{63} + 1 \dots 2^{63} - 1$

Замечу, что сложный тип `Comp` может представлять только целочисленные значения в диапазоне от  $-2^{63} + 1$  до  $2^{63} - 1$ , что приблизительно составляет от  $-9.2 \times 10^{18}$  до  $9.2 \times 10^{18}$ .

Более подробное обсуждение особенностей применения различных типов следует далее.

## Символьный тип

Тип данных `Char` используется для описания символьных переменных. Допустимые значения принадлежат расширенному набору символов кода ASCII. ASCII – это аббревиатура от American Standard Code for Information Interchange (американский стандартный код для обмена информацией). Согласно стандарту ASCII, каждому символу и некоторым управляющим инструкциям соответствует свой числовой код, принимающий значения от 0 до 127. В двоичном представлении ASCII-код использует 7 разрядов. Коды этих символов приведены в

табл. 2.4. Расширенная таблица ASCII использует 8 двоичных разрядов и состоит из двух частей. Первая, в которую входят символы с кодами 0–127, является универсальной, а вторая (коды 128–255) предназначена для специальных символов и букв национальных алфавитов (в том числе и русского).

Таблица 2.4. Символы ASCII с кодами 0–127

Код	Символ	Код	Символ	Код	Символ	Код	Символ
0	NUL	32	SP	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	23	>	94	^	126	~
31	US	63	?	95	_	127	DEL

Переменная типа `Char` хранит один символ. Символьная константа задается указанием символа, который записывается между апострофами. Если переменная `Symbol` описана как переменная типа `Char`, то допустимы следующие операторы присваивания:

```
Symbol := 'S';
Symbol := '?';
Symbol := '';
Symbol := ''';
```

После выполнения второго оператора присваивания переменная `Symbol` имеет значение символа «знак вопроса», третьего — значение «пробел» (в этом случае пробел является полноправным символом), а в четвертом первый и последний апострофы являются ограничителями символьного значения, а два апострофа между ними трактуются как один апостроф.

Значение переменной символьного типа можно задать в операторе присваивания с помощью символьной константы или функции `Chr`. Функция `Chr` устанавливает соответствие между однобайтовыми целыми значениями кода и символами.

Противоположной по отношению к `Chr` является функция `Ord(x)`, которая возвращает код символьного аргумента.

Два символа можно сравнивать посредством отношений `<`, `>`, `<=`, `>=` и т. д. При сравнении символьных значений сравниваются их коды. Справедливы, например, такие отношения:

```
'A' < 'B' < 'C' < ... < 'Y' < 'Z'
'A' < 'a'
```

Существует функция `UpCase`, которая преобразует строчные буквы латинского алфавита в прописные, но не изменяет другие символы, например:

```
UpCase('p') = 'P'
UpCase('B') = 'B'
UpCase('+') = '+'
```

## Управляющие символы

В табл. 2.4 кроме обычных символов имеются специальные *управляющие символы*. Они представляют собой команды, вывод которых на стандартное выходное устройство приводит к выполнению определенных действий. Эти символы имеют мнемонические двух- или трехбуквенные сокращения, пришедшие к нам из эры телеграфа. К управляющему символу можно обратиться по его ASCII-коду или по Ctrl-последовательности. Последняя представляет собой код, порождаемый одновременным нажатием клавиши `Ctrl` и какой-либо другой клавиши.

Предположим, например, что в программе имеется описание

```
var ch: Char;
```

Тогда операторы

```
ch: = chr(7);
ch: = #7;
ch: = ^G
```

присваивают символьной переменной `ch` одно и то же символьное значение. Здесь `^G` обозначает Ctrl-последовательность `Ctrl+G` (управляющий символ BEL – звуковой сигнал). Знак `#` и следующая за ним целая беззнаковая константа обозначают код символа.

Из тридцати двух управляющих символов вам, скорее всего, могут понадобиться те, которые сведены в табл. 2.5.

Таблица 2.5. Некоторые управляющие символы набора ASCII

Код	Ctrl-последовательность	Использование функции Chr	Мнемоническое обозначение	Действие
#7	^G	Chr(7)	BEL	Звуковой сигнал динамика
#8	^H	Chr(8)	BS	Возврат курсора на одну позицию
#9	^I	Chr(9)	HT	Горизонтальная табуляция
#10	^J	Chr(10)	LF	Перевод строки
#12	^L	Chr(12)	FF	Прогон страницы
#13	^M	Chr(13)	CR	Возврат каретки
#26	^Z	Chr(26)	SUB	Конец файла
#27	^[	Chr(27)	ESC	Символ Escape

## ВВОД СИМВОЛОВ С КЛАВИАТУРЫ

До сих пор мы занимались вопросами вывода символов на экран. Обратимся теперь к их вводу с клавиатуры. При нажатии на символьную клавишу естественно ожидать увидеть на экране в месте расположения курсора тот символ, который был введен. Ниже приводится текст программы `testread`. Эта программа использует оператор `ReadLn` для того, чтобы считать с клавиатуры один символ и вывести в следующей строке то, что было считано. В данном случае при вводе символа вы увидите его на экране еще до того, как будет нажата клавиша `Enter`. Нажмем клавишу `a`, а затем `Enter`, и программа сообщит, что она считала символ «`a`». Предлагаю провести следующий эксперимент. Запустите программу `testread` и в ответ на приглашение «Ведите символ» нажмите любую символьную клавишу и держите ее не отпуская. На экране при этом будут отображаться символы, и происходит это будет до тех пор, пока количество этих символов не достигнет 127. После этого нажатие клавиш уже не будет приводить к отображению новых символов, только динамик компьютера будет недовольно пощелкивать. Это сигнал о том, что полностью заполнен *буфер клавиатуры* – рабочая область памяти, в которой временно может храниться до 127 символов, введенных с клавиатуры.

**Листинг 2.4.** Ввод символов с клавиатуры

```
program testread;
var
  ch: Char;
begin
  WriteLn('Введите символ: '); ReadLn(ch);
  WriteLn('Введен символ: ', ch);
  Write('Нажмите <Enter>: '); ReadLn;
end.
```

Можно ли прочитать значение нажатой клавиши так, чтобы она не отображалась при этом на экране? Да, можно — модуль CRT содержит функцию ReadKey, которая именно это и делает. При вводе символа эта функция не сдвигает курсор и поэтому дает возможность вместо введенного символа вывести любой другой. Программа `test_readkey` использует эту функцию, заменяя каждую строчную букву заглавной.

**Листинг 2.5.** Ввод символов с преобразованием строчных латинских букв в прописные

```
program test_readkey;
uses CRT;
var
  ch: Char;
begin
  WriteLn('Введите строчные латинские буквы или z для того, чтобы выйти');
  repeat
    ch := ReadKey;
    Write(UpCase(ch));
  until ch = 'z';
end.
```

Нажатию каждой символьной клавиши в данной программе можно сопоставить вывод другого символа, сделав из нее, таким образом, простейшую программу шифрования. Пример приведен в следующем листинге.

**Листинг 2.6.** Простейшая программа шифрования

```
program encrypt;
uses CRT;
var
  ch: Char;
begin
  WriteLn('Введите строчные латинские буквы или z для того, чтобы выйти');
```

```

repeat
  ch := ReadKey;
  Write(Char(Ord(ch)+1));
  until ch = 'z';
end.

```

Проведем еще один опыт. Запустите программу `test_readkey` и нажмите клавишу `F1`. На экране при этом появятся два символа, первый из них — «пробел», а второй — еще какой-нибудь символ. Аналогичная реакция будет на нажатие клавиш `PgUp` и клавиш управления курсором. Очевидно, реакция на нажатие специальных клавиш отличается от реакции на нажатие алфавитно-цифровых клавиш. Почему?

Познакомимся с тем, как работает клавиатура. При нажатии клавиши формируются два кода — код символа и код сканирования (называемый также расширенным кодом). Код сканирования важен для тех клавиш, которым не сопоставлены обычные символы. Это, например, клавиши перемещения курсора, функциональные клавиши и т. д. Расширенный код является двухбайтовым. При считывании функцией `ReadKey` клавиши `F1` сначала возвращается код символа, равный 0, а второй вызов `ReadKey` возвращает значение кода сканирования. В табл. 2.6 приведены коды сканирования, а также коды, не распознаваемые Турбо Паскалем.

Таблица 2.6. Коды сканирования

Коды, распознаваемые Турбо Паскалем			
Нажатые клавиши	Код сканирования	Нажатые клавиши	Код сканирования
<code>Ctrl+@ Ctrl+3</code>	3	<code>PgDn</code>	81
<code>Shift+Tab</code>	15	<code>Ins</code>	82
<code>Alt+1..Alt+=</code>	120..131	<code>Del</code>	83
<code>Alt+Q..Alt+P</code>	16..25	<code>F1..F10</code>	59..68
<code>Alt+A..Alt+L</code>	30..38	<code>Shift+F1..Shift+F10</code>	84..93
<code>Alt+Z..Alt+M</code>	44..50	<code>Ctrl+F1..Ctrl+F10</code>	94..103
<code>Alt+Enter</code>	28	<code>Alt+F1..Alt+F10</code>	104..113
<code>Home</code>	71	<code>Ctrl+Print Screen</code>	114
<code>↑</code>	72	<code>Ctrl+Home</code>	119
<code>PgUp</code>	73	<code>Ctrl+PgUp</code>	132
<code>←</code>	75	<code>Ctrl+←</code>	115
<code>→</code>	77	<code>Ctrl+→</code>	116
<code>End</code>	79	<code>Ctrl+End</code>	117
<code>↓</code>	80	<code>Ctrl+PgDn</code>	118

Коды, не распознаваемые Турбо Паскалем			
Нажатые клавиши	Код сканирования	Нажатые клавиши	Код сканирования
<code>F11..F12</code>	133..134	5 (цифровая клавиатура)	76
<code>Shift+F11..Shift+F12</code>	135..136	<code>Ctrl+5 (цифровая клавиатура)</code>	143
<code>Ctrl+F11..Ctrl+F12</code>	137..138	<code>Alt+/-</code>	53
<code>Alt+F11..Alt+F12</code>	139..140	<code>Alt+/\</code>	43

Программа `scan_codes` предназначена для определения кода сканирования, соответствующего нажатой клавише. При работе с этой программой не рекомендуется нажимать комбинации клавиш `Ctrl+PrtScr` или `Alt+Ctrl+Del`, а нажатие `Ctrl+Break` приведет к прекращению выполнения программы.

#### **Листинг 2.7. Программа определения кода сканирования**

```
program scan_codes;
uses CRT;
var
  ch: Char;
begin
  WriteLn('Программа предназначена для определения символьного кода
и кода сканирования');
  WriteLn('Прекратить работу можно нажатием клавиш <Ctrl+Break>');
repeat
  Write('Следующая клавиша:');
  ch := ReadKey;
  WriteLn;
  if ch <> #0 then
    WriteLn('Обычная клавиша. Ord(ch) = ', Ord(ch))
  else
    begin
      Write('Символьный код: #0. ');
      ch := ReadKey;
      WriteLn('Код сканирования:', Ord(ch));
    end;
  WriteLn;
until false;
end.
```

При нажатии на некоторые клавиши, например на `F11`, ничего не происходит. У этих клавиш имеется и символьный код, и код сканирования, но функция DOS, которую Паскаль использует для `ReadKey`, настроена не на расширенную, а на обычновенную клавиатуру PC.

#### **Упражнение 2.7**

В модуле `System` имеется функция `UpCase`, выполняющая преобразование строчных латинских букв в прописные (небуквенные символы не преобразуются). Запрограммируйте свой вариант этой функции, используя тот факт, что `Ord('A') = Ord('a') - 32`.

#### **Упражнение 2.8**

Запрограммируйте обратную по отношению к `UpCase` (и отсутствующую в стандартных библиотеках) функцию `LowCase` преобразования прописных латинских букв в строчные. Используйте тот факт, что `Ord('A') = Ord('a') + 32`.

**Упражнение 2.9**

Напишите программу, выводящую на экран символы из второй половины кодовой таблицы.

## СтРОКОВЫЙ ТИП

Строковый тип описывается с помощью зарезервированного слова `string`. Допустимыми значениями переменных или иных объектов строкового типа являются строки символов. В каком-то смысле строковая переменная является массивом символов, но, в отличие от обычного массива с его фиксированной длиной, длина строковой переменной может меняться. Пример описания строковой переменной:

```
var pet: string[4];
```

Целое число в квадратных скобках задает максимальную длину строки. Это *атрибут длины* строковой переменной. В данном примере максимальная длина переменной `SS` равна 4, поэтому присваивание вида

```
pet := 'dog';
```

разрешено, а присваивание

```
pet := 'kitty';
```

является ошибочным, так как фактическое значение переменной `pet` будет '`kit`'. Максимальная длина строки составляет 255 символов, и именно эта длина назначается строковой переменной по умолчанию (автоматически), если параметр длины не указан вовсе. Таким образом, предложение

```
var TT: string;
```

описывает переменную `TT` как строку длиной до 255 символов.

**Описание**

```
var St: string[80];
```

резервирует для `St` 81 байт памяти. Если учесть, что на каждый символ отводится один байт, возникает естественный вопрос, а откуда взялся еще один, 81-й, а точнее, нулевой байт? Дело в том, что после того, как переменной `St` присвоено какое-то значение, нулевой байт содержит фактическую длину строки `St`. Длина строки в байтах при этом равна значению кода символа, находящегося в `St[0]`. Например, присваивание

```
St := 'abcdefghijklm';
```

дает такой результат:

```
St[0] = Chr(8),
```

```
St[1] = 'a',
```

```
... ,
```

```
St[8] = 'h'
```

Замечу, что символ с кодом 8 является управляющим. Фактическая длина строковой переменной St равна Ord(St[0]). Значение 255 для верхнего предела длины строки объясняется тем, что одиночный байт может принимать 256 различных значений, от 0 до 255.

У переменных строкового типа есть интересная особенность — программист имеет доступ к их отдельным байтам. Вот пример.

**Листинг 2.8.** Программа, демонстрирующая доступ к отдельным байтам строковой переменной

```
program Shakespeare;
```

```
var
    Juliet: string[51];

begin
    Juliet := 'What''s he that follows there,
              that would not dance?';
    writeln(Juliet);
    Juliet[36] := ' ';
    Juliet[37] := 'h';
    Juliet[38] := 'a';
    Juliet[39] := 's';
    Juliet[40] := ' ';
    Juliet[41] := 'n';
    Juliet[42] := 'o';
    Juliet[43] := 't';
    Juliet[44] := ' ';
    Juliet[45] := 'c';
    Juliet[46] := 'h';
    writeln(Juliet);
end.
```

В этой программе имеется описание строковой переменной Juliet:

```
var
    Juliet: string[51];
```

Эта переменная инициализируется, то есть ей присваивается определенное значение:

```
Juliet := 'What''s he that follows there, that would not dance?';
```

Замечу, что неинициализированные строковые переменные могут быть источником труднообнаружимых ошибок. Операторы присваивания изменяют значение отдельных байтов, так что в результате изменяется и значение строковой переменной. В первом операторе присваивания апостроф повторяется два раза подряд, в этом случае он воспринимается не как ограничитель строки символов, а как обычный символ (одиночный апостроф). Результат выполнения программы — строка

```
What's he that follows there, that has not chance?
```

## Операции над строками

В Паскале имеется набор процедур и функций для работы со строками. К строкам можно применять операцию *конкатенации*, которая обозначается знаком «плюс». Конкатенация — это объединение строк:

```
SS1 := 'ABC';
SS2 := SS1 + 'DEF';
```

Результатом такой последовательности операторов будет строка 'ABCDEF'. Несмотря на то, что конкатенация выглядит как арифметическое сложение, результат этой операции зависит от порядка слагаемых и меняется при их перестановке. Результатом выполнения последовательности операторов

```
SS1 := 'ABC';
SS2 := 'DEF'+SS1;
```

будет значение 'DEFABC'. Длина строковой переменной SS2 должна назначаться с учетом суммарной длины слагаемых.

Строчковая константа в исходном тексте программы должна размещаться в пределах одной строки. Простой перенос части символьного значения на другую строку приведет к ошибке компиляции «Error 8: String constant exceeds line». Чтобы задать длинное строковое значение, занимающее в тексте программы несколько строк, можно воспользоваться операцией конкатенации, например:

```
Hamlet := 'To be, or not to be: that is the question: ' +
'Whether ''tis nobler in the mind to suffer';
```

Операцию слияния строк str1 и str2 выполняет и функция Concat(str1, str2). Приведенная ниже программа counter является примером практического применения процедур для работы со строками. Она предназначена для подсчета запятых, точек и пробелов в текстовых файлах. Будем считать, что предложением в текстовом файле называется любая последовательность символов, завершающаяся точкой.

**Листинг 2.9.** Программа подсчета некоторых символов в текстовых файлах

```
program counter;
uses Crt;
var
  infile: text;
  file_name, s: string;
  i, commas, points, blanks, lines: integer;
begin
  ClrScr;
  commas := 0;
  points := 0;
  blanks := 0;
```

```

lines := 0;
Write('Введите имя файла:');
ReadLn(file_name);
Assign(infile, file_name);
Reset(infile);
while not Eof(infile) do
begin
  ReadLn(infile,s);
  for i := 1 to Length(s) do
  begin
    case s[i] of
      ',' : inc(commas);
      '.' : inc(points);
      ' ' : inc(blanks);
    end;
  end;
  inc(lines);
end;
close(infile);
GotoXY(1,3);
WriteLn('Количество запятых : ', commas);
WriteLn('Количество предложений: ', points);
WriteLn('Количество пробелов : ', blanks);
WriteLn('Количество строк : ', lines );
WriteLn('Нажмите <Enter>:');
ReadLn;
end.

```

Процедура `ClrScr` (модуль `Crt`) очищает экран, а `GotoXY(column, line)` (модуль `Crt`) устанавливает курсор в позицию `column` на строке `line`. В программе `counter` используются процедуры для работы с внешними файлами, поскольку анализируемый файл именно таким и является. Работу с внешними файлами мы будем разбирать позже, а сейчас обратимся к процедурам и функциям для работы со строками символов.

Функция `Length(str)` определяет длину аргумента `str` строкового типа. Процедура `Delete(str, istart, nrem)` используется для того, чтобы удалить `nrem` символов в строковой переменной `str`, начиная с позиции `istart`. Например, последовательность операторов

```

abra := '123abcde4567890';
Delete(abra, 4, 5);

```

даст результат `abra = '1234567890'`. Имеется также процедура `Insert` для вставки одной строки в другую. Чтобы вставить строку (константу или значение строковой переменной) `instr` в строковую переменную `str`, начиная с позиции `istart`, данную процедуру следует вызывать следующим образом: `Insert(instr, str, istart)`. Например, последовательность операторов

```

digits := '1237890';
Insert('456', num, 4);

```

даст результат `digits = '1234567890'`.

Функция `Copy(str1, iStart, n)` копирует  $n$  символов строки `str1`, начиная с позиции `iStart`. Это строковое значение можно присвоить другой строковой переменной. Например, последовательность операторов

```
str := Copy('1234567xyzw890', 8, 4);
```

даст результат `str = 'xyzw'`.

Функция `Pos(substr, str)` определяет начальную позицию подстроки `substr` в строке `str`. Например,

```
Pos( ' F(x) ', 'Let F(x) = 2x' ) = 5
```

Значение `Pos` равно нулю, если подстрока не встретилась в строке.

Имеются две процедуры преобразования числовых значений в строковые и наоборот: `Str` и `Val`. Процедура `Str` при обращении к ней вида

```
Str(num, strnum);
```

где `num` — значение числового типа, а `strnum` — переменная строкового типа, присваивает переменной `strnum` строковое значение, представляющее собой символьное изображение значения переменной `num`. В первом параметре функции `Str` можно использовать спецификаторы формата (см. раздел «Ввод и вывод» первого урока). Процедура `Val` выполняет обратное преобразование. Обращение к ней имеет вид:

```
Val(strnum, num, errcode);
```

Третий параметр в этой процедуре равен нулю при успешном выполнении преобразования. В том случае, когда первый параметр содержит символы, недопустимые при записи числа, значение параметра `errcode` равно номеру позиции с ошибочно заданным символом. Пример использования процедур `Str` и `Val` дается ниже:

```
program str_and_val;
var
  i, errcode: Integer;
  S: string;

begin
  Str(2000, S);
  Writeln('Строковое значение ', S);
  Readln;
  Val(S, i, errcode);
  if errcode <> 0 then
    Writeln('Ошибка ввода в позиции: ', errcode)
  else
    Writeln('Числовое значение = ', i);
  Readln;
end.
```

В этом примере вначале выполняется преобразование целочисленного значения `2000` в строковое (обращение к процедуре `Str`), а затем, наоборот, строка символов `«2000»` преобразуется в значение типа `Integer` (процедура `Val`).

И наконец, две строки можно сравнивать при помощи операций `>`, `<`, `<=`, `>=` и т. д. Сравниваются при этом коды символов, начиная с первых символов строк. Справедливы, например, следующие соотношения:

```
'Alexei' < 'Sergei'  
'100' < '110'  
'Boy' < 'girl'
```

Последний пример здесь демонстрирует правило, что более короткая строка всегда «меньше» более длинной.

#### Упражнение 2.10

Напишите процедуру, которая меняет в строке текста порядок следования символов на обратный.

#### Упражнение 2.11

Напишите программу, удаляющую в текстовом файле все отступы.

#### Упражнение 2.12

Предположим, что некий текст на иностранном языке, хранящийся в обычном текстовом файле, должен быть подготовлен для перевода на русский язык. Для выполнения подстрочного перевода удобно иметь по две копии каждой строки так, чтобы можно было вторую строку текста заменить переводом. Напишите программу, которая для любого входного текстового файла создает текстовый файл с повторением каждой строки.

#### Упражнение 2.13

Напишите программу, которая подставляет номера строк (в виде комментария языка Паскаль, например `{32}`) в конец каждой строки файла с исходным текстом программы.

## Массивы

Массивы, как и циклы, — величайшее изобретение программирующего человечества. Массивы приходят на помощь тогда, когда приходится иметь дело с наборами однотипных и однородных данных. Такой набор, имеющий общее для всех своих элементов имя, и называется массивом. Тип «массив» является представителем группы структурных типов Паскаля.

Элементы массива пронумерованы, и обратиться к каждому из них можно, указав его *индекс*. Можно сказать, что индекс — это обобщенное понятие номера. Индекс может быть один, но их может быть и несколько. В математических вычислениях часто приходится иметь дело с векторами. *Вектор* — это пример массива, в котором элементы нумеруются одним индексом. Если речь идет о хранении в массиве *таблицы* значений (матрицы), его элементы нумеруются двумя индексами.

Массив определяется прежде всего общим типом его элементов и их количеством. Количество элементов массива, в свою очередь, определяется количеством индексов и диапазоном их изменения. В программировании количество индексов массива называют его *размерностью (рангом)*, количество допустимых значений каждого индекса — его *диапазоном*, а совокупность размерности и диапазонов — *формой* массива. Последний термин связан с простой аналогией. Геометрическим образом массива можно считать многомерный параллелепипед. Каждое измерение этого параллелепипеда соответствует некоторому индексу, а его протяженность в каком-либо измерении — диапазону соответствующего индекса.

При обращении к элементу массива индекс указывается в квадратных скобках после имени массива. Он может быть выражением. Описание массива вида

```
var space_time: array [1..4] of Real;
```

резервирует память под 4 вещественных числа, являющихся элементами массива `space_time`. Здесь в квадратных скобках указан диапазон `1..4` — упорядоченный набор из четырех натуральных чисел `1, 2, 3, 4`. Ссылка на эти компоненты имеет вид

```
space_time [1]
...
space_time [4]
```

В общем случае `1..4` можно заменить любым (конечным) скалярным типом. Уточнение в скобках объясняется очевидной причиной — ограниченным объемом памяти компьютера. *Скалярный тип* — это любой тип Паскаля, состоящий из упорядоченных элементов (более подробно о скалярных типах рассказывается дальше в этом уроке).

Если речь идет о двумерных (в общем случае — многомерных) массивах, то в описаниях должны быть заданы диапазоны изменения каждого индекса. Это можно делать по-разному. Описания

```
var twodim: array[1..6, 1..10] of Word;
var twodim: array[1..6] of array [1..10] of Word;
```

полностью эквивалентны; обращаться к элементам получившегося массива можно тоже по-разному: `Ar2D[4..8]` или `Ar2D[4][8]`.

Массив описывается в разделе `var` с использованием конструкции

```
array[n..m] of type;
```

где вместо `type` должен быть указан общий для всех элементов тип (*базовый тип*).

В Турбо Паскале есть ограничение по памяти, отводимой под любую переменную. Это ограничение составляет 64 Кбайта. Например, описание

```
var too_large: array [1..200, 1..100] of Extended;
```

отводит под массив `too_large`  $200 \times 100 \times 10 = 200\ 000$  байт, что превышает допустимый предел. Чтобы обойти это ограничение, приходится использовать специальные приемы. Данная особенность Турбо Паскаля несколько ограничивает его возможности для программирования сложных вычислительных задач.

Идентификаторы массивов можно использовать в обеих частях оператора присваивания, если они имеют одинаковую форму и совместимы по типу.

Перейдем к примерам. Полином

$$a_k x^k + a_{k-1} x^{k-1} + \dots + a_1 x + a_0$$

можно описать массивом его коэффициентов  $[a_0, a_1, \dots, a_{k-1}, a_k]$ , в котором роль индекса играет степень переменной  $x$ . Рассмотрим умножение полиномов  $P(x) = A(x) \times B(x)$  в случае, когда их степени ограничены, а слагаемые в произведении, соответствующие степеням, большим максимальной, отбрасываются. Программа `polynom_product` предназначена для перемножения двух полиномов, заданных массивами своих коэффициентов.

**Листинг 2.10.** Программа перемножения полиномов

```
program polynom_product;

const
  max = 10;

var
  A, B, P : array [0..max] of Real;
  j, k     : Word;

begin
  for k := 0 to max do
    A[k] := k;
  B := A;
  for k := 0 to max do
    P[k] := 0.0;
  for j := 0 to max do
    for k := 0 to max - j do
      P[j + k] := P[j + k] + A[j]*B[k];
  for k := 0 to max do
    WriteLn(P[k]);
  ReadLn;
end.
```

Следующая программа — `twirl` — выводит на экран маленькую вращающуюся стрелку. Имитацию вращения стрелки можно запрограммировать, используя массив, состоящий из четырех символьных элементов: |, \, - и /. Эти символы последовательно, один за другим и с некоторой задержкой по времени, выводятся в заданном месте экрана. Эффект «смены кадров» и создает иллюзию вращения стрелки. Алгоритм реализован в процедуре `rot`. Первый параметр этой процедуры (`j`) задает номер позиции в строке, второй (`k`) — номер строки вывода, третий (`color`) — это цвет стрелки и четвертый (`speed`) — временная задержка (скорость вращения). Курсор устанавливается в заданное положение в результате вызова процедуры `GotoXY`. Цвет стрелки задается с помощью процедуры `TextColor`. Индексы элементов массива `line` циклически перебираются до тех

шор, пока не будет нажата любая клавиша. Временная задержка между выводом последовательных элементов массива обеспечивается процедурой `Delay`.

#### Листинг 2.11. Программа «вращение стрелки»

```
program twirl;
uses dos, crt;
procedure rot(j, k: integer; color, speed: word);
const
line: array[0..3] of char = ('|', 'V', '^', '/');
var
m: integer;
begin
textcolor(color);
m := 0;
repeat
gotoxy(j, k);
write(line[m]);
gotoxy(1, 1);
delay(speed);
m := (m + 1) mod 4;
until keypressed;
textcolor(LightGray);
end;
begin
rot(40, 13, Yellow, 100);
end.
```

В операторах присваивания можно использовать не только элементы массивов, но и массивы в целом. Оператор вида `A := B` лаконичнее и быстрее цикла `for i:=1 to n do A[i] := B[i]`. С другой стороны, необходимо помнить, что оператор вида `A := B` приводит к копированию всего массива, и поэтому использовать его следует очень аккуратно. Массивы в целом можно использовать и в логических отношениях равенства (`=`) и неравенства (`<>`). Другие операции отношения применяются только к элементам массива.

#### Упражнение 2.14

Напишите функцию, которая проверяет, является ли квадратная матрица, состоящая из элементов целочисленного типа, симметричной.

#### Упражнение 2.15

Пусть

```
const
max = 10000;
var
vector: array[1..max] of Integer;
```

Напишите процедуру, которая определяет значение максимального компонента вектора и его индекс.

### Упражнение 2.16

Пусть заданы следующие описания:

```
const
  max_deg = 30;

type
  polynomial = array[-1..max_deg] of Integer;

var
  F, G, H: polynomial;
```

Напишите процедуру суммирования для полиномов с целочисленными коэффициентами.

### Упражнение 2.17

Даны последовательность  $x_1 < x_2 < \dots < x_n$  вещественных чисел и вещественное число  $y$ . Напишите программу для определения такого  $k$ , что  $x_k < y < x_{k+1}$ .

## Множества

Следующим представителем группы структурных типов является *множество*. Тип «множество» задает интервал значений, который является множеством всех подмножеств базового типа. Множество описывается в разделе var с помощью зарезервированного слова set. Базовым типом множества может быть любой скалярный (конечный) тип, состоящий не более чем из 256 элементов. В силу этого базовый тип множества не может быть коротким целым (ShortInt), целым (Integer), длинным целым (LongInt) или словом (Word). Вот пример описания множества:

```
var
  S: set of Char;
```

Для того чтобы пояснить особенность данного типа, приведу пример. Если переменная типа «множество» описана как set of 1..3, то она может принимать значения из следующего множества: {(1, 2, 3), (1, 2), (1, 3), (2, 3), (1), (2), (3), ()}.

Константы множественного типа записываются с помощью квадратных скобок [ ] и списка элементов. Примеры:

```
const
  alphabet = ['A'..'Z', 'a'..'z'];
  empty = [];
  digits = [0..9];
```

Отсутствие списка элементов в квадратных скобках обозначает пустое множество. Зарезервированное слово `in` используется для определения принадлежности элемента множеству:

```
if ch in alphabet then ...
```

Множества языка Паскаль обладают свойствами математических множеств. В частности, над ними можно выполнять те же операции. Если  $S_1$  и  $S_2$  — константы или переменные множественного типа, то  $S_1 + S_2$  будет их объединением,  $S_1 * S_2$  — пересечением,  $S_1 - S_2$  — разностью. Объединением математических множеств  $s_1$  и  $s_2$  называется множество, состоящее из элементов, принадлежащих или множеству  $s_1$ , или множеству  $s_2$ , или обоим множествам. Пересечение множеств состоит из элементов, одновременно принадлежащих обоим множествам, а разностью называется множество, состоящее из элементов, принадлежащих первому, но не принадлежащих второму множеству. Операции отношений  $=$  (равенство),  $\neq$  (неравенство),  $\subset$  (является подмножеством),  $\supset$  (является надмножеством) к множествам применяются, а отношения строгого включения  $<$  и  $>$  — не применяются.

Иногда в операторе `if` гораздо удобнее использовать множественный тип:

```
if ch in alphabet then ...
```

чем, например, пару операций сравнения:

```
if ( (ch >= 'A') and (ch <= 'Z') ) or  
    ( (ch >= 'a') and (ch <= 'z') ) then ...
```

Чтобы добавить в множество какой-нибудь элемент, можно либо добавить множество, состоящее из единственного элемента, либо использовать процедуру `Include`.

Следующий фрагмент программы формирует множество всех заглавных букв с нечетными кодами `['A', 'C', ..., 'Y']`:

```
var  
  S : set of 'A'..'Z';  
  ch : Char;  
  
begin  
  S := [];  
  ch := 'A';  
  repeat  
    S := S + [ch];  
    Inc(ch);  
    Inc(ch);  
  until ch >= 'Z';
```

Альтернативой оператору `S := S + [ch]`; является оператор `Include (S, ch)`. Имеется и обратная процедура `Exclude` исключения элемента из множества. У этой процедуры два параметра, первый указывает множество, а второй — исключаемый элемент.

# Целый и логический (булев) типы

Вернемся к основным строительным блокам — числовым и некоторым другим типам данных — и более подробно рассмотрим некоторые их свойства.

## Целые типы

Мы уже знаем, что в Паскале имеется несколько целых типов. Целыми типами со знаком являются `ShortInt`, `Integer` и `LongInt`. Беззнаковые целые типы — это `Byte` и `Word`. Однобайтовые типы `ShortInt` и `Byte` особенно полезны в больших массивах, если их элементы принимают значения из ограниченного диапазона. Их применение в этом случае позволяет уменьшить требования программы к оперативной памяти. Например:

```
var
  short_1000 : array[1..1000] of ShortInt;
  int_1000   : array[1..1000] of Integer;
```

Несложно проверить, что

```
SizeOf (short_1000) = 1000
SizeOf (int_1000) = 2000
```

Здесь `SizeOf(x)` — это функция, возвращающая значение типа `Integer`, равное количеству байт памяти, занимаемых аргументом. Таким же образом можно сравнить массив элементов типа `Byte` с массивом элементов типа `Word`.

Типы `Integer` и `Word` являются двухбайтовыми, диапазоны их значений составляют  $[-32\ 768...32\ 767] = [-2^{15}...2^{15} - 1] = [-\text{MaxInt} - 1...\text{MaxInt}]$  и  $[0...65\ 535] = [0...2^{16} - 1]$  соответственно. Встроенная константа `MaxInt` принимает максимально допустимое значение типа `Integer`.

`LongInt` является 4-байтовым типом с диапазоном  $[-2\ 147\ 483\ 648...2\ 147\ 483\ 647] = [-\text{MaxLongInt} - 1...\text{MaxLongInt}] = [-2^{31}...2^{31} - 1]$ .

Встроенная (предопределенная) константа `MaxLongInt` равна максимальному значению типа `LongInt`. Границы диапазонов значений типов `Integer` и `LongInt` простираются в отрицательную область дальше, чем в положительную. Это является следствием того, что семейство компьютеров РС использует для хранения отрицательных чисел дополнительный код.

Основными *бинарными* (то есть выполняющимися над двумя значениями) операциями являются сложение (+), вычитание (-), умножение (\*), деление (/). В том случае, когда они применяются к значениям, имеющим разные целые типы, будет производиться преобразование типа. При этом значение, имеющее более «простой» (короткий) тип, вначале преобразуется в более «сложный» (длинный) формат. В связи с этим следует сделать несколько предостережений. Предположим, например, что дано описание переменных

```
var
  a, b : Word;
  x: Integer;
```

и где-то в программе имеется присваивание

```
a := 3;
b := 5;
x := a - b;
```

Такая конструкция приводит к сбою с сообщением об ошибке хода выполнения программы «Error 215: Arithmetic overflow». Почему? Дело в том, что выражение  $a - b$  будет вычислено до присвоения значения результата переменной  $x$ . Это выражение имеет тип *Word*, а тип *Word* не может принимать отрицательные значения. Тем не менее выполнение оператора

```
x := - b + a;
```

не приводит к ошибке! Это еще раз подтверждает вывод о том, что в машинной арифметике при перестановке слагаемых результат может измениться. Гораздо более безопасный способ выполнения вычитания таков:

```
x := a;
x := x - b;
```

Мы уже имели дело с операцией целочисленного деления *div* и вычисления остатка *mod*. Первый operand в конструкциях  $m \text{ div } n$  и  $m \text{ mod } n$  должен быть неотрицательным, а второй operand — положительным.

В модуле *System* есть две *унарных* (с единственным аргументом) операции (функции) над целыми числами: *Abs* и *Sqr*. Здесь же имеется и булева функция *Odd*, которая возвращает значение «истина», если аргумент — нечетное значение.

Наряду с привычными арифметическими операциями в Паскале есть операции, выполняющие более сложные преобразования целых значений. Операции *shl* (левый сдвиг) и *shr* (правый сдвиг) в тех случаях, когда они применяются к беззнаковым целым, дают тот же результат, что умножение (*shl*) или деление (*shr*) на степень 2. Вот пример использования этих операций:

$7 \text{ shl } 3$  (результат —  $7 \times 2^3 = 56$ );

$56 \text{ shr } 3$  (результат —  $56 \text{ div } 2^3 = 7$ ).

Действие этих операций можно рассмотреть и с точки зрения двоичного представления операнда. Пусть  $n$  является переменной типа *Word*. В двоичном представлении для  $n$  отводится 16 бит, при этом считают, что младший бит расположен справа, а старший — слева. Операции сдвига смещают все биты вправо (*shr*) или влево (*shl*), при этом лишние биты отбрасываются, а освободившиеся места заполняются нулями:

```
00000111 shl 3 => 00111000
00111000 shr 3 => 00000111
```

Применяя операции сдвига к целым со знаком, следует иметь в виду, что старшим битом является бит знака. Любой правый сдвиг присваивает этому биту нулевое значение, что означает установку знака «плюс». При левом сдвиге знак результата может оказаться произвольным. Замечу также, что в операциях `shl` и `shr` применяются быстрые алгоритмы вычисления и оптимизированные машинные коды. В определенных ситуациях операции сдвига полезно использовать для замены операций умножения или деления на степени двойки.

Существуют четыре «логических» операции, которые можно применять к целым типам: `not`, `and`, `or` и `xor`. Они действуют побитно согласно описанию, данному в табл. 2.7.

Таблица 2.7. Логические операции

Операнды		Результат операции			
<code>b1</code>	<code>b2</code>	<code>b1 and b2</code>	<code>b1 or b2</code>	<code>b1 xor b2</code>	<code>not b1</code>
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Операция `not` является унарной операцией, она заменяет в каждом бите ноль на единицу и наоборот. Примеры:

```
not 00110000 => 11001111
00110000 and 00101000 => 00100000
00110000 or 00101000 => 00111000
00110000 xor 00101000 => 00011000
```

В дальнейшем нам придется использовать целые значения, записанные в шестнадцатеричном формате. Цифрами в этом случае являются символы 0...9 и A...F, причем допустимы маленькие буквы. Перед значением в этом случае ставится знак доллара, например, `$FF` — это 255 в десятичной системе.

Преобразование числовых значений из одного представления в другое — малоприятная операция (во всяком случае, мне так кажется), но иногда таким преобразованием приходится заниматься. Решим для себя эту проблему раз и навсегда. Назначение следующей программы — преобразование десятичного целого в шестнадцатеричное. Она может обрабатывать целые числа длиной до 200 цифр. Разумеется, это строковые данные, так как целых чисел длиной более 6 или 7 знаков в Паскале нет. Программа `dec2hex` дает возможность поупражняться в работе со строками и целыми значениями, в ней используется многое из того, что мы уже знаем, поэтому разобраться с этой программой и выполнить ее будет легко. Отметим использование процедуры `Val` для преобразования строковых представлений чисел в числовые значения.

**Листинг 2.12.** Преобразование десятичного значения в шестнадцатеричное

```
program Dec2hex;

{Вводится строка до 200 десятичных цифр. Выводится число
в шестнадцатеричном формате}

uses CRT;

const
  h15 = '0123456789ABCDEF';

var
  hold_in, dec_str, hex_str: string;
  hex_digits: array [0..15] of Char;
  ch: Char;
  k: Integer;

procedure initialize;
var
  j: Word;
  temp: string [Length(h15)];
begin
  temp := h15;
  for j := 0 to 15 do hex_digits[j] := temp[j + 1];
end;

procedure get_decimal(var dec_str: string );
var
  k: Word;

begin
  repeat
    WriteLn('Введите строку десятичных цифр:');
    ReadLn(dec_str);
    if dec_str = '' then
      Halt;

    {Удалить недесятичные символы}
    for k := Length(dec_str) downto 1 do
      if not (dec_str[k] in ['0'..'9']) then
        Delete(dec_str, k, 1);

    {Удалить нули в старших разрядах}
    while (Length(dec_str) > 1) and (dec_str[1] = '0') do
      Delete(dec_str, 1, 1);
    if Length(dec_str) > 0 then
      Exit;
    WriteLn('Ошибка при вводе, повторите');
    WriteLn;
    until false;
end;
```

```
procedure convert(var dec_str, hex_str: string );
var
  remainder: Integer;

procedure divide_by_16;
{Строка dec_str, состоящая из десятичных цифр, делится на 16.
Остаток далее будет преобразован в следующую шестнадцатеричную цифру.
Частное преобразуется в строку dec_str}

var
  j, k,
  dividend: Word;
  error: Integer;

begin
  remainder := 0; {0 <= remainder <= 15}
  for j := 1 to Length(dec_str) do
    begin
      if KeyPressed then
        Halt;
      Val(dec_str[j], k, error);
      dividend:= 10*remainder + k;

      {0 <= dividend<= 159}
      remainder := dividend mod 16;

      {0 <= remainder <= 15}
      dec_str[j] := hex_digits[dividend div 16];
    end;
  Delete(dec_str, 1, 1);
end;{divide_by_16}

begin
  hex_str := '';
  while Length(dec_str) > 0 do
  begin
    if KeyPressed then Halt;
    divide_by_16;
    hex_str := hex_digits[remainder] + hex_str;
  end;
end;{convert}

begin
  initialize;
  repeat
    get_decimal(dec_str);
    hold_in := dec_str;
    convert(dec_str, hex_str);
    k := Length(hold_in) - 2;
    while k > 1 do
```

```

begin
  Insert(' ', hold_in, k);
  Dec(k, 3);
end;

Write(hold_in, ' = $');
while Length(hex_str) mod 4 <> 0 do
  hex_str := '0' + hex_str;
  k := Length(hex_str) - 3;
  while k > 1 do
    begin
      Insert(' ', hex_str, k);
      Dec(k, 4);
    end;
  WriteLn(hex_str); WriteLn;

WriteLn('Для продолжения работы нажмите <Enter>,
а для окончания <Esc>');

ch := ReadKey;
if ch = #27 then Break;
WriteLn;
until false;
end.

```

Поясню действие процедур `Break`, `Exit` и `Halt`. Процедура `Break` используется в цикле для того, чтобы остановить его дальнейшее выполнение. Она прерывает выполнение только одного цикла, самого внутреннего. Процедура `Exit` останавливает выполнение процедуры или функции, из которой она вызывается. Это обстоятельство может оказаться важным в рекурсивных программах. Процедура `Halt` безусловно прекращает выполнение программы.

## Логический (булев) тип

Логический (булев) тип имеет два возможных значения — «истина» (`True`) и «ложь» (`False`). Эти значения считаются упорядоченными, так что `False < True`. Под булево значение отводится один байт памяти.

Булевыми операциями являются `not`, `and`, `or` и `xor`. Их действие объяснялось в предыдущем параграфе.

Булевые операции используются в *логических выражениях*. Логические выражения могут включать в себя *отношения* (`<`, `>` и т. д.) и булевые операции.

Отметим важное обстоятельство, касающееся порядка вычислений в булевых выражениях. Если речь идет о вычислении сложного логического выражения вида

`Expr1 and Expr2`

то вначале вычисляются значения выражений Expr1 и Expr2, а потом уже применяется операция `and`. Но довольно часто оказывается, что результат очевиден уже после вычисления первого операнда. В выражении

`False and Expr2`

неважно, какое значение принимает Expr2. Как обойтись в таком случае без лишних вычислений?

Интегрированная среда Турбо Паскаля позволяет установить опцию, которая называется вычислением булевых выражений по короткой схеме. Такая схема вычислений включается по умолчанию при запуске интегрированной среды. Переключение этого режима производится с помощью флагка `Complete Boolean eval`, расположенного в окне настроек компилятора, вызываемом командой меню `Options ▶ Compiler`.

Следующая программа является примером использования логических выражений — как отношений, так и более сложных.

#### Листинг 2.13. Булевые выражения

```
program Boolean_Expressions;

var
  bool1, bool2, bool3, bool4,
  bool5, bool6, bool7, bool8, bool9: Boolean;
  num1, num2 : Word;

begin
  num1 := 13;
  num2 := 7;

  bool1 := num1 = num2;
  bool2 := num1 = (num2 + 21);
  bool3 := num1 < num2;
  bool4 := num1 > num2;

  Writeln(num1,' = ',num2,'      ', bool1);
  Writeln(num1,' = ',num2 + 21,'      ', bool2);
  Writeln(num1,' < ',num2,'      ', bool3);
  Writeln(num1,' > ',num2,'      ', bool4);
  Writeln;

  bool5 := bool2 and bool3 and bool4;
  bool6 := bool2 and bool3 and not bool4;
  bool7 := bool2 or bool3 or bool4;
  bool8 := (bool2 and bool3) or not (bool3 and bool4);
  bool9 := (num1 = num2 - 1) or (num1 = num2);

  Writeln('bool5 = ', bool5);
  Writeln('bool6 = ', bool6);
  Writeln('bool7 = ', bool7);
  Writeln('bool8 = ', bool8);
```

```

Writeln('bool9 = ', bool9);
WriteLn;
WriteLn('Нажмите <Enter>');
ReadLn;
end.

```

### Упражнение 2.18

Определите результат вычисления булевых переменных `bool5`, `bool6`, `bool7`, `bool8` и `bool9` в программе 2.13.

## Скалярные типы

Скалярными часто называют перечисляемые типы и отрезки типов. Перечисляемые типы определяют упорядоченные наборы значений. Эти наборы задаются перечислением идентификаторов, которые обозначают эти значения. Например:

```

type
  rainbow = (red, orange, yellow, green, blue, indigo, violet);

```

Порядок значений в таком наборе определен последовательностью, в которой перечисляются идентификаторы. Первая перечисленная константа имеет порядковый номер 0.

При применении функции `Ord` к значению перечисляемого типа она возвращает целое число, которое показывает, какое положение занимает это значение в списке. В нашем примере `Ord(red)` возвращает значение 0, `Ord(orange)` возвращает значение 1 и т. д. Кроме того, справедливы следующие соотношения:

```

orange < violet
yellow = Pred(green)
blue = Succ(green)

```

Напомню, что функция `Pred` возвращает значение, предшествующее аргументу порядкового типа, а функция `Succ` — следующее за аргументом. Можно считать, что мы просто присваиваем имена последовательным натуральным числам от 0 до некоторого максимального значения и в дальнейшем обращаемся к этим числам по именам.

Можно было бы просто описать такой тип:

```

type
  prism = 0..6;

```

Это пример описания *отрезка типа*. Отрезок типа представляет собой диапазон значений из порядкового типа, называемого *главным типом*. Определение отрезка типа включает наименьшее и наибольшее значения в поддиапазоне. В нашем примере это 0 и 6. Обе константы должны иметь один и тот же порядковый тип, и, кроме того, отрезки типов, имеющие вид `a..b`, предполагают, что `a` меньше или равно `b`.

Переменная отрезка типа имеет все свойства переменных главного типа, однако ее значение на этапе выполнения должно принадлежать указанному интервалу.

### Упражнение 2.19

Напишите свой вариант функции `Odd`, возвращающей значение «истина», если ее целочисленный аргумент имеет нечетное значение.

### Упражнение 2.20

Напишите функцию `roundup(x: LongInt; n: Word): LongInt;` для округления вещественного значения  $x$  до ближайшего целого, кратного  $n$ .

### Упражнение 2.21

Напишите функцию, которая определяет количество единиц в двоичном представлении переменной типа `Word`.

### Упражнение 2.22

Напишите функцию `power(x: Real; n: Integer): Real;`, которая вычисляет  $x^n$ .

## Вещественные типы

В Паскале имеется несколько типов вещественных данных. Эти типы приведены в табл. 2.3. Для работы со всеми вещественными типами кроме `Real` требуется математический сопроцессор 80x87. Если сопроцессора нет, то компилировать программу следует в режиме эмуляции, задав директиву компилятора `{$E}`. Тип `Comp` особенный, его назначение — обеспечивать программирование деловой арифметики с применением сопроцессора 80x87. Допустимые значения — целые числа из указанного в табл. 2.3 диапазона.

Наряду с хорошо знакомыми нам бинарными арифметическими операциями `+`, `-`, `*`, `/` в модуле `System` имеются следующие элементарные математические функции:

`Abs Sqr Sqrt Exp Ln Sin Cos Tan ArcTan`

Любую из них можно применять и к аргументам целого типа. При вычислении функции в этом случае ее аргументы будут автоматически преобразованы в вещественный тип. В Турбо Паскале имеется встроенная константа `Pi`, но нет константы — основания натуральных логарифмов  $e$ .

Функции из следующей группы:

`Round Trunc Int Frac`

имеют другую природу. `Round` преобразует вещественное значение в *ближайшее* к нему значение типа `LongInt`. Функция `Trunc` отсекает дробную часть вещественного числа, оставляя длинное целое. `Int` и `Frac` — это, соответственно, целая

(округляемая «по направлению» к нулю) и дробная части вещественного числа, также имеющие вещественный тип.

Из многообразия вещественных типов стоит обратить внимание на тип **Extended**. Он обеспечивает большую точность, кроме того, **Extended** является «родным» типом сопrocessора 80х87, поэтому его применение позволяет максимизировать скорость вычислений с вещественными значениями.

Прежде чем перейти к следующей программе, обратимся к проблеме представления численных значений в компьютере. Двоичное представление различных вещественных типов следует стандарту IEEE. Стандарт IEEE 754 определяет формат хранения вещественных значений с различной точностью (одинарная, двойная и т. д.) Согласно этому стандарту, представление вещественного значения типа **Extended** (в описании стандарта это **double-extended**) в компьютере с процессором Intel состоит из следующих четырех полей (табл. 2.8): 63-битной дробной части (биты с номерами 0–62)<sup>f</sup>, старшего бита мантиссы — специального бита, используемого для обработки ситуации с потерей значимости *j*, 15-битного порядка со смещением (порядок здесь — показатель степени двойки) *e* и бита знака *s*. Значение типа **Extended** хранится в десяти последовательно расположенных байтах, то есть занимает 80 бит.

**Таблица 2.8.** IEEE-представление для типа Extended

Биты			
79	78...64	63	62...0
Бит знака ( <i>s</i> )	Порядок ( <i>e</i> )	Бит потери значимости ( <i>j</i> )	Дробная часть ( <i>f</i> )

Численное значение, соответствующее данному представлению, определяется следующим образом:

если  $0 \leq e < 32\ 767$ , то  $x = (-1)^s \times 2^{(e - 16383)} \times j.f$ ;

если  $e = 32\ 767$  и  $f = 0$ , то  $x = (-1)^s \times \text{Inf}$ . Здесь Inf обозначает «машинную бесконечность»;

если  $e = 32\ 767$  и  $f \neq 0$ , то  $x = \text{NaN}$ . NaN (Not a Number) является признаком нечислового значения.

Программа **ieee** предназначена для отображения введенного значения в формате IEEE.

**Листинг 2.14.** Отображение десятичного значения в формате IEEE

```

{$N+}

program ieee;
uses crt;
const
```

```
mask = 127;

type
  byte_array = array[0..9] of byte;

var
  x, y, m: extended;
  b: byte_array;
  k: word;
  e: integer;
  s, u: 0..1;

begin
  ClrScr;
  WriteLn('Введите вещественное число:');
  ReadLn(x);
  b := byte_array(x);

  WriteLn('Байты введенного числа x:');
  Write('Номер байта:');
  for k := 9 downto 0 do write(k:5);
  WriteLn;
  Write('Значение:   ');

  for k := 9 downto 0 do write(b[k]:5);
  WriteLn;
  WriteLn;

  e := 256*(b[9] and mask) + b[8];
  s := b[9] shr 7;
  u := b[7] shr 7;
  m := 0.0;

  for k := 0 to 6 do m := m / 256.0 + b[k];
  m := (m / 256.0 + (b[7] and mask)) / 128.0;
  WriteLn('Показатель      e = ', e);
  WriteLn('Бит знака        s = ', s);
  WriteLn('Бит потери значимости  u = ', u);
  WriteLn('Мантисса        m = ', m);
  WriteLn;

  y := u + m;
  e := e - 16383;
  if e > 0 then
    for k := 1 to e do
      y := 2.0 * y
  else
    for k := 1 to -e do
      y := y / 2.0;

  if s = 1 then
    y := -y;
```

```

    WriteLn('Формула реконструкции численного значения:');
    WriteLn('x = [(-1)^s]*[2^(e - 16383)]*[u.m]');
        WriteLn('Реконструкция      x = ', y:26);
        WriteLn('Исходное значение   x = ', x:26);
    WriteLn('Нажмите <Enter>:');
    ReadLn;
end.

```

Программа `display_binary` отображает двоичное представление целого числа.

#### Листинг 2.15. Двоичное представление целого числа

```

program display_binary;

var
  r: LongInt;

procedure Binary(var x: LongInt);

var
  hold: array[0..7] of boolean;
  b: byte;
  size, j, k: word;
  bits: array[1..32] of byte;

begin
  size := SizeOf(x);
  move(x, bits, size);

  for k := size downto 1 do
  begin
    b := bits[k];
    for j := 0 to 7 do
    begin
      hold[j] := odd(b);
      b := b shr 1;
    end;

    for j := 7 downto 0 do
    if hold[j] then
      write(1)
    else
      write(0);
    end;
  end;

  begin
    WriteLn('Введите целое число');
    ReadLn(r);
    WriteLn('Двоичное представление:');
    Binary(r);
    WriteLn;
  end;
end.

```

```

WriteLn('Нажмите <Enter>:');
ReadLn;
end.

```

Использованная в программе процедура `Move(x, bits, size)` перемещает `size` смежных байтов, начинающихся с первого байта переменной `x`, на место, начинающееся с первого байта переменной `bits`.

## Старшинство операций

Мы знаем теперь, что Паскаль допускает использование сложных выражений, в которых могут объединяться арифметические, логические и некоторые другие операции. В этом случае следует учитывать порядок выполнения операций. В первом уроке уже шла речь о порядке выполнения арифметических операций.

Порядок выполнения операций определяется их старшинством, или *приоритетом*. В табл. 2.9 приведены приоритеты операций. Первый уровень приоритетов является наивысшим, а четвертый — низшим.

**Таблица 2.9.** Старшинство операций

Операция	Приоритет
<code>not</code>	Первый
<code>*, /, div, mod, and, shl, shr</code>	Второй
<code>+, -, or, xor</code>	Третий
<code>=, &lt;&gt;, &lt;, &gt;, &lt;=, &gt;=, in</code>	Четвертый

При определении порядка выполнения операций следует учитывать следующее:

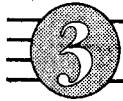
- Операнд, находящийся между двумя операциями с различными приоритетами, относится к операции, имеющей более высокий приоритет.
- Операнд, находящийся между двумя операциями с равными приоритетами, относится к той операции, которая находится слева от него.
- Выражение, заключенное в скобки, перед использованием вычисляется как отдельный operand.
- Операции с равным приоритетом выполняются слева направо, если этот порядок не изменен с помощью круглых скобок.

### Упражнение 2.23

Напишите программу для отображения IEEE-представления для чисел типа `Double`. В стандарте IEEE 754 представление вещественного значения типа `Double` состоит из трех полей: 52-битной дробной части (биты с номерами 0–51)  $f$ , 11-битного порядка со смещением (смещение равно 2047)  $e$  и бита знака  $s$ .

## Что нового мы узнали?

- Познакомились с программированием процедур на Паскале.
- Познакомились с новыми типами данных языка Паскаль: целые, вещественные, булев, символьный, строковый, массив, множество.
- Узнали, как обрабатываются сигналы от клавиатуры.
- Познакомились с приоритетами операций.

**УРОК**

## Типы данных в языке Паскаль

- 
- Работа с типами данных
  - Работа с файлами
  - Записи
  - Указатели
  - Связные списки
  - Модуль для вычислений с полиномами
  - Работа с памятью
  - Типизированные константы
-

**В** этом уроке мы познакомимся с описанием типов данных, вводимых программистом, и обсудим вопросы совместимости типов. Затем нам предстоит знакомство с новыми типами — файлами и записями. Мы разберем работу с файлами, познакомимся с указателями и программированием динамических структур данных. Рассмотрим функции, предназначенные для работы с оперативной памятью компьютера. В заключение обсуждаются типизированные константы.

## Работа с типами данных

Мы уже знаем, что язык программирования Паскаль содержит довольно большой набор *предопределенных* (то есть встроенных) типов данных. В этом уроке нам предстоит знакомство еще с несколькими. Несмотря на значительное многообразие предопределенных типов, программисту в работе часто приходится вводить свои собственные. Для этого программист должен располагать соответствующими средствами. Определение нового типа, разумеется, базируется на встроенных типах, используя их в качестве строительных блоков новой, иногда довольно сложной, конструкции.

### Типы, определяемые пользователем

Наряду с разделами описания переменных, констант, процедур и функций, в программе или в каком-либо ее блоке может присутствовать *раздел описания типов*.

Раздел описания типов начинается зарезервированным словом `type` и имеет следующий вид:

```
type
  type1_ID = type1_description;
  type2_ID = type2_description;
```

где `type1_ID` и `type2_ID` — *идентификаторы* типов, определяемых пользователем, а `type1_description` и `type2_description` представляют собой *описания* соответствующих типов. Вот пример описания типов:

```
type
  Index = 1..100;
```

```

List121 = array[0..120] of Word;
List121by8 = array[1..8] of List121;
PersonData = record
  name,firstName: string[40];
  age: Word;
  married : boolean;
  IQ : (high, medium, low, to_be_neglected);
end;

```

В этом примере программист определяет свои собственные типы: `Index` — интервал целочисленных значений от 1 до 100, `List121` — массив из 121 значения типа `Word`, `List121by8` — двумерный массив, представляющий собой таблицу  $121 \times 8$ , `PersonData` — тип «запись». Далее идентификаторы типов можно использовать для описания переменных в разделе `var`:

```

var
  in1, in2 : Index;
  data: List121;
  Ivanoff, Petroff : PersonData;

```

## Совместимость типов

Думаю, что читатель, набирая и компилируя программы в интегрированной среде Турбо Паскаля, на протяжении первых двух уроков уже имел «удовольствие» видеть сообщение компилятора об ошибке «Type mismatch». Это сообщение появляется в том случае, когда нарушается *условие совместимости типов данных*. В выражениях допускается одновременное использование только совместимых типов. Говорят, что два типа являются совместимыми, если выполняется, по крайней мере, одно из следующих условий:

- оба типа являются одинаковыми;
- оба типа являются вещественными типами;
- оба типа являются целочисленными типами;
- один тип является поддиапазоном другого;
- оба типа являются отрезками одного и того же базового типа;
- оба типа являются множественными типами с совместимыми базовыми типами;
- один тип — это тип `Pointer`, а другой — любой ссылочный тип.

Упомянутое выше сообщение компилятора может появляться, если нарушено *условие совместимости по присваиванию*. Согласно стандарту ISO7185, значение типа `T1` является совместимым по присваиванию с типом `T2` (то есть, допустим, оператор присваивания `T1:=T2`), если выполняется одно из следующих условий:

- `T1` и `T2` имеют тождественные типы, и ни один из них не является файловым типом или структурным типом, содержащим компонент с файловым типом на одном из своих уровней.

- T1 и T2 являются совместимыми порядковыми типами, и значения типа T2 попадают в диапазон возможных значений T1.
- T1 и T2 являются вещественными типами, и значения типа T2 попадают в диапазон возможных значений T1.
- T1 является вещественным типом, а T2 является целочисленным типом.
- T1 и T2 являются строковыми типами.
- T1 является строковым типом, а T2 является символьным типом (**Char**).
- T1 и T2 являются совместимыми множественными типами, и все члены значения типа T2 попадают в диапазон возможных значений T1.
- T1 и T2 являются совместимыми типами указателей.

Строгая дисциплина типов — это несомненное преимущество языка Паскаль, поскольку несоответствие типов в определенных ситуациях может приводить к неверной работе программы, а обнаружить в программе ошибку такого рода часто очень и очень сложно.

Иногда все-таки возникает необходимость использовать в операторах присваивания переменные разных типов. В Паскале есть механизм совместного использования в выражениях данных, имеющих разный тип. Этот механизм называется *приведением типов*. Пусть, например, типы **type1** и **type2** имеют одинаковый размер (занимают в памяти одинаковое число байт). Допустим, что **x** является переменной типа **type1**, а **y** — переменной типа **type2**. Предположим, наконец, что **x** присвоено какое-то значение. Тогда приведение типа

```
y := type2(x)
```

просто интерпретирует данные, хранящиеся в **x**, как значение типа **type2**, на которое ссылаются по имени **y**. В частности, если **N** имеет тип **Byte**, а **ch** — тип **Char**, то допустимы следующие операторы присваивания:

```
ch := Char(N);
N := Byte(ch);
```

В Паскале допускается и приведение типа *значений*. Приведение типа значений выглядит следующим образом:

```
Type_ID(expression)
```

где **Type\_ID** — идентификатор типа, а в круглых скобках стоит выражение **expression**, которое может представлять собой просто константу. Следует иметь в виду, что тип выражения и задаваемый тип должны быть перечисляемыми типами или указателями. Для перечисляемых типов результирующее значение получается путем преобразования выражения и его проверки на нахождение в допустимых границах. Преобразование может привести к усечению или увеличению размера исходного значения в том случае, если результирующий тип отличается от типа выражения. В том случае, когда значение расширяется, его знак всегда сохраняется. Приведение типа значений не должно встречаться в левой части оператора присваивания. Результатом работы программы

```
program cast;
begin
  writeln(integer('A'):10, integer('B'):10);
  writeln(Char(48):10, Char(49):10);
  writeln(Boolean(0):10, Boolean(1):10);
end.
```

будут значения

65	66
0	1
FALSE	TRUE

Первые два из них являются ASCII-кодами символов «А» и «В», значения во второй строке — символы «0» и «1», имеющие ASCII-коды 48 и 49. Последние два значения являются булевыми константами.

## Работа с файлами

В первом уроке мы познакомились с основами работы с текстовыми файлами. В этом разделе нам предстоит более подробное знакомство с теми средствами, которые имеются в Турбо Паскале для работы с файлами. В Паскале имеются три класса файлов: текстовый файл, типизированный файл и нетипизированный файл. Начнем с текстовых файлов.

### Текстовые файлы

Мы уже знаем, что текстовый файл содержит последовательность символов, организованных в строки, причем каждая строка заканчивается специальным символом возврата каретки CR=#13 и перевода строки LF=#10. Заканчивается текстовый файл признаком конца файла. Специальные символы обычно не отображаются программами просмотра текстовых файлов. Работу с текстовыми файлами обеспечивает модуль `System` Турбо Паскаля, не требующий использования оператора `uses`.

В Паскале имеется стандартный файловый тип `text`. Прежде чем приступить к операциям над текстовыми файлами, необходимо ввести переменные (одну или несколько) типа `text`:

```
var in_file: Text;
```

Переменная `in_file` сопоставляется внешнему файлу на диске или какому-нибудь устройству процедурой `Assign`:

```
Assign(in_file, 'C:\user\Newton\my_file');
Assign(my_text, 'prn');
```

Процедура `Assign` связывает имя *внешнего файла* (это именно тот файл, с которым мы собираемся работать) с *файловой переменной*. Внешний файл может

быть обычным файлом, расположенным на жестком диске компьютера, но это может быть и файл, связанный с каким-либо устройством, например, дисплеем или принтером. Файловая переменная далее используется в программе в качестве параметра процедур работы с файлами.

Следующий шаг заключается в том, что внешний файл надо *открыть* для записи или чтения из него какой-то информации. При открытии файла выполняются необходимые системные операции, подготавливающие файл к записи или считыванию информации. Текстовый файл `my_file` открывается процедурой `Reset(my_file)` *только для чтения* и процедурой `Rewrite(my_file)` — *только для записи*.

При завершении обработки файла программой он должен *закрываться*. После закрытия файла связанный с ним внешний файл обновляется. Затем файловая переменная может быть связана с другим внешним файлом. Закрывается файл с помощью процедуры `Close(my_file)`.

**ВНИМАНИЕ**

После завершения работы с файлом он *должен быть закрыт*, иначе вся записанная в него информация будет потеряна!

После того как текстовый файл открыт, с ним можно выполнять определенные действия, и прежде всего это запись в файл и чтение из него. Доступ к текстовому файлу организуется последовательно — это означает, что программа не может в любой момент времени считывать из него произвольную порцию информации или произвести запись в произвольное место файла. Любой файл представляет собой *линейную* последовательность элементов, каждый из которых имеет свой номер. Можно считать, что имеется указатель, который при считывании очередного элемента файла перемещается к следующему элементу (то есть становится ближе к концу файла).

Для чтения из текстового файла или записи в текстовый файл можно использовать процедуры `Write`, `WriteLn`, `Read` и `ReadLn`, но в качестве первого параметра в этих процедурах должна быть указана файловая переменная:

```
Read(in_file, a, x);
WriteLn(out_file, 'Urgent message!');
```

Первая процедура присваивает `a` и `x` значения очередных двух элементов из `in_file`.

Следует отметить, что, несмотря на то, что текстовый файл является набором символьных значений, он может использоваться (и часто используется) для хранения численных значений. При считывании значений или их записи в файл происходит автоматическое преобразование из числового формата в символьный и наоборот.

Оператор вывода допускает описание формата вывода. Если `a` является выражением целого, булевого или строкового типа, то

```
WriteLn(a:n);
```

означает запись *a* в *правые* позиции поля размером в *n* позиций. Если указанная длина поля *n* меньше, чем длина значения *a*, то заданная длина игнорируется, например,

```
WriteLn('12345', 3);
```

приведет к выводу всего символьного значения 12345. Значение *n* может быть и отрицательным, в этом случае значение записывается в *левые* *n* позиций.

В том случае, когда *a* имеет вещественный тип, должны быть описаны два поля формата:

```
Write(a:10:3);
```

Такое обращение к процедуре вывода означает запись *a* в форме с фиксированной точкой, с тремя десятичными разрядами и выравниванием по правой границе поля размером 10 позиций.

Мы уже знаем, что в Паскале имеются две стандартных файловых переменных текстового типа — *Input* и *Output*. Стандартная файловая переменная *Input* представляет собой доступный только для чтения файл, связанный со стандартным файлом ввода операционной системы (клавиатура). Вторая стандартная файловая переменная *Output* — это доступный только для записи файл, связанный со стандартным файлом вывода (дисплей). Перед началом выполнения программы DOS эти файлы автоматически открываются. Имя файла в процедурах *Read* и *Write* не указывается, если работа ведется со стандартным файлом.

При обращениях к стандартным функциям и процедурам ввода/вывода автоматически производится проверка на наличие ошибок. При обнаружении ошибки программа прекращает работу и выводит на экран сообщение. Иногда это неудобно. С помощью директив компилятора {\$I+} и {\$I-} автоматическую проверку ошибок ввода/вывода можно включить или выключить. Если автоматическая проверка отключена, ошибки ввода/вывода, возникающие при работе программы, не приводят к ее останову. Стандартная функция *IOResult* возвращает код ошибки. Нулевое значение кода ошибки означает нормальное завершение операции ввода/вывода. В табл. 3.1 приведены процедуры и функции для работы с файлами (не обязательно текстовыми).

Таблица 3.1. Процедуры и функции для работы с файлами

Функция	Описание
procedure Append(var f: text)	Открывает существующий файл, связанный с файловой переменной <i>f</i> , для добавления в него новых записей
procedure Assign(var f, String)	Связывает внешний файл, имя которого указано в строковой константе <i>String</i> , с файловой переменной <i>f</i>
procedure BlockRead(var F: File; var Buf; Count: Word)	Считывает из нетипизированного файла, связанного с файловой переменной <i>f</i> , одну или несколько записей (их количество задается целочисленным выражением <i>Count</i> ) в переменную <i>Buf</i>

Функция	Описание
procedure BlockWrite(var F: File; var Buf; Count: Word)	Записывает в нетиปизированный файл, связанный с файловой переменной F, одну или несколько записей (их количество задается целым выражением Count) из переменной Buf
procedure ChDir(S: String)	Выполняет смену текущего каталога на каталог, маршрут к которому указан в текстовой переменной S
procedure Close(var F)	Закрывает открытый файл, связанный с файловой переменной F
procedure Erase(var F)	Стирает внешний файл, связанный с файловой переменной F
function Eof(var F): boolean	Возвращает для файла, связанного с файловой переменной F, состояние End-of-file (конец файла): True — если текущее положение указателя находится в конце файла или файл пустой; False — во всех остальных случаях
function FilePos(var F): LongInt	Возвращает текущую позицию для файла, связанного с файловой переменной F. При положении текущего указателя в начале файла возвращает нулевое значение. Для текстовых файлов не используется
function FileSize(var F): LongInt	Возвращает текущий размер файла, связанного с файловой переменной F. Если файл пустой, возвращает нулевое значение. Для текстовых файлов не используется
procedure Flush(var F: text)	Сбрасывает буфер текстового файла, связанного с файловой переменной F и открытого для вывода процедурами Rewrite или Append. Это дает гарантию того, что вся информация, записываемая в файл, будет сохранена во внешнем файле. Не влияет на файлы, открытые для ввода
procedure GetDir(D: Byte; var S: String);	Возвращает текущий каталог на заданном диске. Имя каталога находится в строковой переменной S, а диск задается значением параметра D: 0 — текущий диск; 1 — диск А; 2 — диск В; 3 — диск С и т. д. Если значение, заданное в параметре D, неверное, возвращается результат «X:\»
function IOResult: Integer	Возвращает целое значение, являющееся состоянием последней выполненной операции ввода/вывода. Нулевое значение соответствует нормальному завершению операции
procedure Mkdir(S: String)	Создает подкаталог, имя для которого задается строковой переменной S
procedure Read(var F: text; ...)	Считывает одно или несколько значений из файла, связанного с файловой переменной F, в одну или несколько переменных v1...

Таблица 3.1. (Продолжение)

Функция	Описание
procedure ReadLn(var F: text; v1, ...)	То же, что и Read, но выполняет пропуск до начала следующей строки текстового файла
procedure Rename(var F, S)	Переименовывает внешний файл, связанный с файловой переменной F, присваивая ему имя, содержащееся в строковой переменной S
procedure Reset(var F)	Открывает существующий файл, связанный с файловой переменной F. Текущий указатель устанавливается в начало файла. Текстовый файл открывается только для чтения
procedure Rewrite(var F)	Создает и открывает новый файл, связанный с файловой переменной F. Если файл с указанным именем уже существует, старый файл будет стерт, а на его месте создается новый пустой файл. Текущий указатель устанавливается в начало файла
procedure RmDir(S: String)	Удаляет пустой подкаталог, маршрут которого указан в строковой переменной S
procedure Seek(var F; N: Longint)	Перемещает текущую позицию в файле, связанном с файловой переменной F, на заданный элемент. Началу файла соответствует нулевое значение N. Для текстовых файлов не используется
function SeekEof(var F: text): Boolean	Возвращает для текстового файла, связанного с файловой переменной F, состояние «конец файла»
function SeekEoln(var F: text): Boolean	Возвращает для текстового файла, связанного с файловой переменной F, состояние «конец строки»
procedure SetTextBuf(var F: Text; var Buf)	Назначает для текстового файла, связанного с файловой переменной F, буфер ввода-вывода. Никогда не применяется к открытым файлам, поскольку в этом случае возможны потери данных
procedure Truncate(var F)	Усекает размер файла, связанного с файловой переменной F, до текущей позиции. Вся информация после текущего положения указателя теряется. Для текстовых файлов не используется
procedure Write(var F: text; v1, ...)	Записывает в файл, связанный с файловой переменной F, одно или несколько значений, хранящихся в переменных v1... Файловая переменная может быть связана не только с текстовым файлом. В случае текстового файла значения могут иметь целый, вещественный, строковый, символьный или булев типы. В случае типизированного файла тип значений должен совпадать с типом компонентов файла
Procedure Writeln(var F: text; v1, ...)	Делает то же, что Write, но затем записывает в текстовый файл признак конца строки

Вооружившись теоретическими знаниями, перейдем к практике и познакомимся с модулем `fileunit`, содержащим процедуры для работы с файлами. Текст интерфейсной секции этого модуля приведен в листинге 3.1.

#### Листинг 3.1. Интерфейсная секция модуля fileunit

```
unit fileunit;

interface

uses dos;

type
    message = string[60];

var
    name: namestr;
    file_done: Boolean;

procedure get_file_name(prompt: message; var name: pathstr);
procedure test_error(var fname: PathStr);
procedure file_input(mess: PathStr; var GG: text);
procedure file_output(mess: PathStr; var GG: text);
procedure file_copy(file_src, file_dest: PathStr;
    OverWrite:Boolean);
procedure file_merge(file_src,file_dest: PathStr);
procedure file_delete(file_dest: PathStr);
procedure file_rename(file_dest,file_src: PathStr;
    OverWrite:Boolean);

function file_exist_check(Name: PathStr):Boolean;
function get_file_extension(S: PathStr):String;
function get_file_prefix(S: PathStr):String;
function get_file_dir:String;
```

Перейдем к секции реализации модуля `fileunit`. Процедура `get_file_name(prompt: message; var name: PathStr)` (ее текст приведен в листинге 3.2) действует следующим образом. При обращении к ней выдается приглашение, вид которого определяется пользователем в параметре `prompt` (напомню, что это — параметр-значение), предлагающее ввести имя файла. Это может быть, например, такое приглашение:

```
prompt := 'Введите имя файла';
get_file_name(prompt, path);
```

Имя файла вводится пользователем. Параметр-переменная `path`, которая должна быть описана как переменная типа `PathStr` (этот тип описан в модуле `Dos`), содержит результат выполнения процедуры — полное имя файла, состоящее из маршрута и собственно имени. Расширение имени файла до полного выполняется функцией `FExpand` (модуль `DOS`).

**Листинг 3.2.** Процедура get\_file\_name модуля fileunit

```

implementation

{$I+}
procedure get_file_name(prompt: message; var name: PathStr);
begin
  WriteLn(prompt);
  ReadLn(name);
  name := FExpand(name);
end;

```

Процедура test\_error(var fname: PathStr) (листинг 3.3) позволяет проверить выполнение операций с файлами на наличие некоторых ошибок. В случае возникновения ошибки на экран будет выведено сообщение о причине ошибки, а выполнение программы будет остановлено. Работа процедуры основана на анализе кода, возвращаемого функцией IOResult. Успешной операции ввода/вывода соответствует нулевое значение кода ошибки. При обращении к этой функции код ошибки «сбрасывается» в ноль. Следует учесть также, что перед вызовом процедуры test\_error следует поместить директиву {\$I-}, отключающую проверку ввода/вывода. Для прекращения выполнения программы используется процедура halt.

**Листинг 3.3.** Процедура test\_error модуля fileunit

```

procedure test_error(var fname: PathStr);
var
  messg: message;
begin
  case IOResult of
    0: Exit;
    2: messg := 'Файл не найден';
    3: messg := 'Путь не найден';
    4: messg := 'Открыто более 15 файлов';
    5: begin
        writeln('Одна из следующих ошибок:');
        writeln('Попытка записи в файл, открытый только для чтения');
        writeln('Файл с указанным именем существует');
        writeln('Попытка перезаписи файла, открытого только для чтения');
        writeln('Попытка удаления каталога или файла, открытого только для
              чтения');
        writeln('Каталог с указанным именем существует');
        messg := 'Файл ввода/вывода не был открыт';
      end;
    6: messg := 'Файл поврежден';
    8: messg := 'Недостаточно памяти';
    12: messg := 'Неправильный тип файла';
    15: messg := 'Неправильное устройство';
    16: messg := 'Удаление текущего каталога';
    17: messg := 'Попытка переноса файла на другое устройство';
    100: messg := 'Попытка чтения после признака конца файла ' + fname;
    101: messg := 'Нет места на диске';
  end;

```

```

102: messg := fname + ' не был назначен';
103..105: messg := fname + ' не был открыт';
106: messg := 'Ошибка ввода числового значения';
end;
writeln('Файл ', fname, ' не может быть обработан');
writeln(messg);
write('Нажмите <Enter> для прекращения работы программы');
readln;
halt;
end;

```

Листинг 3.4 содержит исходный текст процедур `file_input` и `file_output`. Процедура `file_input(mess: PathStr; var GG: text)` открывает текстовый файл для чтения и устанавливает текущий указатель в начало файла. При этом выполняется проверка на наличие ошибок. Пример использования этой процедуры:

```

file_input(path, fname);
ReadLn(fname, v);

```

Здесь открывается текстовый файл, имя которого указано в переменной `path`, а затем процедурой `ReadLn` производится считывание значения, которое присваивается переменной `v`.

Процедура `file_output(mess: PathStr; var GG: text)` открывает текстовый файл для записи, устанавливая при этом текущий указатель в начало файла. Пример использования процедуры:

```

file_output(path, fname);
WriteLn(fname, '1999');

```

Процедура `file_output` открывает для записи текстовый файл, имя которого указано в переменной `path`, а затем процедура `WriteLn` производит запись символьного значения «1999».

#### Листинг 3.4. Процедуры `file_input` и `file_output` модуля `fileunit`

```

procedure file_input(mess: PathStr; var GG: text);
begin
  assign(GG, mess);
  {$I-}
  reset(GG);
  test_error(mess);
  {$I+}
end;

procedure file_output(mess: PathStr; var GG: text);
begin
  assign(GG, mess);
  {$I-}
  rewrite(GG);
  test_error(mess);
  {$I+}
end;

```

Исходные тексты процедур копирования, объединения, удаления и переименования файлов приведены в листинге 3.5.

**Листинг 3.5.** Процедуры file\_copy, file\_merge, file\_delete и file\_rename модуля fileunit

```

procedure file_copy(file_src, file_dest: PathStr; OverWrite: Boolean);
var
  F_dest, F_src: File;
  NumRead, NumWritten: Word;
  Buf: Array[1..2048] Of Char;
begin
  file_done := False;
  {$I-}
  test_error(file_src);
  if file_exist_check(file_dest) and not OverWrite then
    Exit;

  Assign(F_dest, file_dest);
  Rewrite(F_dest, 1);
  test_error(file_dest);
  Assign(F_src, file_src);
  Reset(F_src, 1);
  test_error(file_src);
  {$I+}

  repeat
    BlockRead(F_src, Buf, SizeOf(Buf), NumRead);
    BlockWrite(F_dest, Buf, NumRead, NumWritten);
  until (NumRead = 0) or (NumWritten <> NumRead);
  Close(F_dest);
  Close(F_src);
  file_done := True;
end;

procedure file_merge(file_src, file_dest: PathStr);
var
  F_dest, F_src: File;
  NumRead, NumWritten: Word;
  Buf: Array[1..2048] Of Char;
begin
  file_done := False;
  {$I-}
  Assign(F_src, file_src);
  Reset(F_src, 1);
  test_error(file_src);
  if file_exist_check(file_dest) then
  begin
    Assign(F_dest, file_dest);
    Reset(F_dest, 1);
    Seek(F_dest, FileSize(F_dest));
  end
  else

```

```
begin
  Assign(F_dest, file_dest);
  Rewrite(F_dest, 1);
end;
test_error(file_dest);
repeat
  BlockRead(F_src, Buf, SizeOf(Buf), NumRead);
  BlockWrite(F_dest, Buf, NumRead, NumWritten);
until (NumRead = 0) or (NumWritten <> NumRead);
Close(F_dest);
Close(F_src);
file_done := True;
{$I+}
end;

procedure file_delete(file_dest: PathStr);
var
  F: File;
begin
  {$I-}
  Assign(F, file_dest);
  Erase(F);
  test_error(file_dest);
  {$I+}
  file_done := (IOResult = 0);
end;

procedure file_rename(file_src, file_dest: PathStr; OverWrite : Boolean);
var
  F: File;
begin
  file_done := False;
  {$I-}
  if not file_exist_check(file_src) then
    begin
      writeln('Файл ', file_src, ' не существует');
      Exit;
    end;
  if file_exist_check(file_dest) then
    if not OverWrite then Exit;
    else
      begin
        Assign(F, file_dest);
        Erase(F);
      end;
  Assign(F, file_src);
  Rename(F, file_dest);
  test_error(file_dest);
  file_done := True;
  Close(F);
  {$I+}
end;
```

Процедура `file_copy(file_src, file_dest: PathStr; OverWrite:Boolean)` предназначена для копирования файла с именем `file_src` в файл с именем `file_dest`. Третий параметр булева типа позволяет определить действие в том случае, когда файл-назначение уже существует. Если значение этого параметра `true`, файл будет замещен копией файла `file_src`, если `false`, то копирование не выполняется. При выполнении процедуры открываются два файла — файл «источник» для чтения и файл «назначение» для записи. Затем процедура `BlockRead` выполняет считывание очередного элемента данных из файла источника в буфер, которым является символьный массив. Процедура `BlockWrite` производит запись содержимого массива в файл назначения. Копирование производится до тех пор, пока не будет исчерпано содержимое источника. После этого оба файла закрываются, а их содержимое сохраняется на диске.

Процедура `file_merge(file_src, file_dest: PathStr)` предназначена для «склеивания» файлов. Содержимое файла-источника `file_src` дописывается в конец файла-назначения `file_dest`. Данная процедура похожа на процедуру копирования файлов, однако в ней после открытия файла, в который будет производиться запись, текущий указатель устанавливается не на начало файла, а в его конец. Номер последнего элемента определяется функцией `FileSize`.

Процедура `file_delete(file_dest: PathStr)` удаляет файл, имя которого задано в качестве параметра. Удаление файла производится процедурой `Erase`.

Процедура `file_rename(file_src, file_dest: PathStr; OverWrite:Boolean)` производит переименование файла. Выполняется это обращением к процедуре `Rename`. Кроме этого имеются проверки корректности переименования. При этом может оказаться, что файл с указанным (новым) именем уже существует. Если третий параметр процедуры при этом имеет значение `true`, имеющийся файл будет замещен новым, в противном случае переименование не производится.

В некоторых процедурах и функциях модуля `fileunit` используется функция `file_exist_check(Name: PathStr)` (листинг 3.6), возвращающая значение `true`, если указанный файл существует, и `false`, если его нет. Можно считать, что это «облегченный» вариант процедуры `test_error`.

#### Листинг 3.6. Процедуры `file_exist_check`, `get_file_extension`, `get_file_prefix` и `get_file_dir` модуля `fileunit`

```
function file_exist_check(Name: PathStr): Boolean;
var F: file;
begin
  Assign(F, Name);
  {$I-}
  Reset(F);
  test_error(Name);
  {$I+}
  if IOResult = 0 then
  begin
    file_exist_check := True;
    Close(F);
  end
end
```

```
        else
            file_exist_check := False;
end;

function get_file_extension(S: PathStr): String;
var
    mess: String;
begin
    mess := S;
    Delete(mess, 1, Pos('. ', mess));
    if mess = S Then mess := '';
    get_file_extension := mess;
end;

function get_file_prefix(S: PathStr): String;
var
    mess: String;
begin
    mess := S;
    Delete(mess, Pos('. ', mess), Length(mess) - Pos('. ', mess) + 1);
    get_file_prefix := mess;
end;

function get_file_dir(S: PathStr): String;
begin
    repeat
        Delete(S, Length(S), 1);
        until S[Length(S)] = '\';
        get_file_dir := S;
    end;
begin
end.
```

И, наконец, модуль `fileunit` содержит функции `get_file_extension`, `get_file_prefix` и `get_file_dir` (см. листинг 3.6). Функция `get_file_extension(S: PathStr)` возвращает строковое значение, представляющее собой часть имени файла после точки, отделяющей собственно имя от расширения. Для этого вначале определяется положение точки в строковой переменной, содержащей имя файла (функция `Pos`). Затем из строкового значения удаляется все, кроме расширения.

Функция `get_file_prefix(S: PathStr)` возвращает имя файла без расширения, а устроена она аналогично функции `get_file_extension`. Отмету, что эти функции работают неправильно в том случае, когда при указании полного имени файла точка имеется в имени одного из каталогов.

Функция `get_file_dir` возвращает имя каталога, в котором находится файл, имя которого указано в качестве параметра. Работает она аналогично функциям `get_file_prefix` и `get_file_extension`, но символом-разделителем в этом случае является не точка, а обратная наклонная черта \.

### Упражнение 3.1

В модуле DOS содержится процедура FSplit(Path, Dir, Name, Ext), возвращающая для файла, имя которого указано в качестве первого параметра, три компонента его имени: имя каталога, собственно имя и расширение. Переделайте процедуры `get_file_extension`, `get_file_prefix` и `get_file_dir`, используя эту функцию.

## Типизированные файлы

Описание типизированного файла имеет вид

```
var ftable: file of type_ID;
```

где `type_ID` может быть любым типом за исключением файлового. Элементами типизированного файла являются значения указанного типа.

При работе с типизированными файлами можно использовать уже знакомые нам процедуры `Assign`, `Reset` и `Rewrite`. Следует заметить, что текстовый файл, открытый процедурой `Reset`, доступен только для чтения, а типизированный — еще и для записи. Процедуры `Read` и `Write` здесь используются по-другому. Отличие заключается в том, что каждый из параметров в рассматриваемом случае должен быть переменной типа `type_ID`, а выражения и константы недопустимы. Процедуры `ReadLn` и `WriteLn` применяются *только* к текстовым файлам.

Типизированные файлы используются для хранения однородной (по типу) информации. Если речь идет о хранении числовых данных, следует использовать типизированные файлы. Почему? Рассмотрим программу, которая производит запись числовых данных в текстовый и типизированный файлы.

**Листинг 3.7.** Программа, демонстрирующая работу с текстовым и типизированным файлами

```
program file_of_extended;
var
  extfile: file of extended;
  textfile: text;
  x, y: extended;
  i: word;
begin
  assign(textfile, 'table.txt');
  rewrite(textfile);
  x := 0.0;
  for i := 1 to 1000 do
  begin
    y := sin(x);
    WriteLn(textfile, y);
    x := x + 0.001;
  end;
  close(textfile);
```

```

assign(extfile, 'table.ext');
rewrite(extfile);
x := 0.0;
for i := 1 to 1000 do
begin
  y := sin(x);
  write(extfile, y);
  x := x + 0.001;
end;
close(extfile);
Write('Работа программы закончена, нажмите <Enter>;');
ReadLn;
end.

```

После выполнения этой программы на диске (в рабочем каталоге программы) появятся два файла, `table.txt` и `table.ext`. Первый из них является текстовым файлом, второй — типизированным. Несмотря на то, что эти файлы содержат одинаковую информацию, между ними имеется существенное различие. Если просмотреть содержимое обоих файлов, то в первом случае (текстовый файл) мы увидим столбик цифр. Во втором случае (типованный файл) на экране отобразится хаотический набор символов. Если мы сравним, кроме того, размер обоих файлов, то обнаружим, что файл `table.ext` имеет длину 10 000 байт, а файл `table.txt` — 25 000 байт. Почему? Переменная типа `Extended` занимает 10 байт, поэтому типизированный файл, содержащий 1000 чисел типа `Extended`, имеет размер 10 000 байт. Но если переменная типа `Extended` хранится в виде текстовой строки, то эта строка состоит из 23 символов, так как она включает показатель степени «`E+nnnn`» или «`E-nnnn`». Добавьте сюда еще управляющие символы `CR` и `LF`, которыми оканчивается каждая строка при вызове процедуры `WriteLn`, и вы получите 25 символов на каждое значение типа `Extended`, записанное в текстовый файл. В итоге это даст 25 000 байт. По этой причине для хранения числовых данных экономнее использовать тип `file of Extended`.

## Нетипизированные файлы

Для более эффективного выполнения операций ввода/вывода из внешних файлов в Паскале имеются нетипизированные файлы. При работе с ними можно использовать быстрые дисковые операции низкого уровня. Нетипизированные файлы дают возможность прямого доступа к любому файлу на диске независимо от его типа и структуры. Описание нетипизированной файловой переменной имеет вид

```
var untypfile: file;
```

Такая файловая переменная связывается с внешним файлом обычным образом. В числе параметров процедур `Reset` и `Rewrite` для нетипизированных файлов кроме файловой переменной имеется необязательный второй параметр типа `Word`:

```
Reset(untypfile, n);
Rewrite(untypfile, n);
```

Дополнительный параметр *n* описывает размер индивидуальной записи в файле (в байтах). Если параметр *n* отсутствует, его значение по умолчанию принимается равным 128, однако рекомендуется явно указывать значение 1. Это связано с тем, что при любом другом значении величины записи в файле могут присутствовать неполные записи, а это не всегда удобно.

Теоретического багажа и практических навыков программирования на Паскале у нас уже достаточно для того, чтобы приступить к программе, имеющей достаточно большое практическое значение. Проблема состоит в том, чтобы прочитать текстовый файл, созданный в операционной системе UNIX, и записать его в формате текстового файла операционной системы MS-DOS. Можно представить себе ситуацию, когда некий файл был создан на сервере, работающем под управлением UNIX, а затем скопирован на компьютер пользователя, работающий под управлением MS-DOS. Не исключено также, что вы получали такой файл в UU-кодированном виде по электронной почте. Разница между форматами хранения текстовых файлов в упомянутых операционных системах состоит в том, что в UNIX каждая строка текстового файла оканчивается одним управляемым символом LF, а в DOS каждая строка оканчивается двумя последовательными управляемыми символами CR и LF. Таким образом, входной файл имеет формат, отличный от текстового формата Паскаля.

**Листинг 3.8.** Программа преобразования текстовых файлов из формата UNIX в формат DOS

```
program Unix2Dos;

const
  LF = #10;
  CR = #13;

var
  unix_file, dos_file: file;
  buffer: array[1..4] of char;
  input_bytes, output_bytes, i: word;

begin
  assign(unix_file, paramstr(1));
  reset(unix_file, 1);
  assign(dos_file, paramstr(2));
  rewrite(dos_file, 1);
  writeln('Выполняется преобразование файла Unix ', paramstr(1),
         ' в файл DOS ', paramstr(2));
  repeat
    blockread(unix_file, buffer, sizeof(buffer) - 2, input_bytes);
    for i := input_bytes downto 1 do
      begin
        if buffer[i] = LF then
          begin
            inc(input_bytes);
            move(buffer[i], buffer[i + 1], input_bytes - i);
            output_bytes := output_bytes + 1;
            if output_bytes > 255 then
              begin
                writeln(dos_file);
                output_bytes := 0;
              end;
            write(dos_file, buffer[i]);
          end;
      end;
  until input_bytes = 0;
end.
```

```
        buffer[i] := CR;
      end;
    end;
    blockwrite(dos_file, buffer, input_bytes, output_bytes);
  until (input_bytes = 0) or (output_bytes <> input_bytes);
  close(dos_file);
  close(unix_file);
end.
```

Итак, задача заключается в том, чтобы считать файл UNIX и заменить в нем каждый символ LF на пару символов CR и LF, не изменяя все остальные символы. Программа вызывается в командной строке с двумя параметрами, которыми являются имена входного и выходного файлов:

**unix2dos fileunix filedos**

Вначале открываются два файла — входной и тот, в который будет производиться запись. Затем производится считывание «порции» данных из входного файла. Процедурами для чтения нетипизированных файлов и для записи в них являются **BlockRead** и **BlockWrite**. Программа выделяет в качестве буфера область памяти, куда будет производиться запись и откуда будет производиться чтение. Роль буфера играет символьный массив **buffer**, размер которого 4 байта. В силу того что мы будем добавлять символы CR, размер буфера должен быть больше, чем число считанных байт. В переменной **input\_bytes** содержится количество фактически считанных байт. Это значение будет равно 2 до тех пор, пока процедура **BlockRead** не будет вызвана последний раз. Замечу, что **BlockRead** имеет четыре параметра — файловую переменную, переменную буфера, количество байт, которые следует считать, и количество байт, считанных в действительности. Обращение к **BlockWrite** имеет аналогичный вид. Функция **paramstr(i)** возвращает значение *i*-го параметра командной строки.

### Упражнение 3.2

Напишите программу для проверки процедур и функций модуля **fileunit**, а также его сообщений об ошибках (проверку советую проводить на чужом компьютере!).

### Упражнение 3.3

Не прибегая к помощи компьютера, определите результат работы следующей программы:

```
program file_of_string;

type
  Str80 = string[80];

var
  FF: file of Str80;
  GG: Text;
  ss: Str80;
  k: Word;
```

```

begin
  Assign(FF, 'strings.str');
  Rewrite(FF);
  Assign(GG, 'strings.txt');
  Rewrite(GG);
  ss := 'abc';
  for k := 1 to 100 do
    begin
      Write(FF, ss);
      WriteLn(GG, ss);
    end;
  Close(FF);
  Close(GG);
end.

```

#### Упражнение 3.4

Пусть FF — это внешний файл, состоящий из вещественных чисел. Напишите программу, которая считывает значения из файла и находит в FF четвертое по величине значение.

#### Упражнение 3.5

Пусть FF — это внешний файл, описанный как `file of Char`. Напишите программу, которая считывает значения из файла и находит, сколько раз каждый алфавитно-цифровой символ встречается в FF.

#### Упражнение 3.6

Пусть FF — это внешний файл, описанный как `file of Char`. Необходимо считывать из файла FF символьные значения до тех пор, пока не встретится его конец или не будут считаны по крайней мере по одному разу все символы «A»...«Z».

## Записи

### Записи с фиксированными частями

Зарезервированное слово `record` позволяет задавать такой тип данных, который содержит определенное число элементов — *полей* и представляет собой смесь разных типов. Определение типа «запись» имеет вид

```

type type_name = record
  <описание переменных>
end;

```

В описании указываются идентификатор каждого поля и его тип.

Займемся математикой. Вещественный моном от 4 переменных имеет вид  
 $c x^n y^m z^k t^p$

и определяется коэффициентом — вещественным числом  $c$  — и степенями каждой переменной. Естественным способом хранения вещественного монома в какой-либо математической программе является использование определения типа «запись», объединяющего вместе одну переменную вещественного типа и массив показателей степеней:

```
program monomial;

const
  max = 4;

type
  term = record
    coeff: Real;
    deg: array [1..max] of Word;
  end;
```



### ВНИМАНИЕ

Переменные типа «запись» могут участвовать в операторах присваивания, но никакие операции над ними выполняться не могут. Арифметические или какие-либо другие операции могут выполняться только над отдельными полями записи.

Если теперь  $M1$  является переменной типа  $term$  и в программе необходимо обращаться к ее коэффициенту, сделать это можно посредством использования точки в идентификаторе этой переменной, например

```
x := M1.coeff;
```

Есть и другой способ — использование зарезервированного слова  $with$ :

```
with M1 do x := coeff;
```

В операторе, стоящем после  $do$ , к полям переменной типа «запись» можно обращаться просто по имени поля, не указывая имя переменной. Это облегчает выполнение многократных ссылок к полям записи.

Продолжим наш пример процедурой перемножения двух мономов:

```
procedure product(M1, M2: term; var MResult: term);
var
  d: 1..max;
begin
  with MResult do
  begin
    coeff := M1.coeff*M2.coeff;
    for d := 1 to max do
      deg[d] := M1.deg[d] + M2.deg[d];
  end;
end;
```

Следующий наш математический «подвиг» — довольно серьезный проект, а именно, модуль для комплексной арифметики. Дело в том, что в Паскале нет

встроенного типа для комплексных чисел, поэтому нам придется самим создать его. Конечно, комплексное число можно было бы описать следующим образом:

```
type complex = array[1..2] of Double;
```

но как-то неудобно ссылаться, например, на вещественную часть *z* как *z*[1]. Применение типа «запись» с двумя вещественными полями в данном случае представляется более удачным.

Рассмотрим исходный код модуля для работы с комплексными числами *cmplx*. Исходный текст его интерфейсной секции приведен в листинге 3.9. Особенностью данного модуля является использование несколько необычного способа хранения комплексных чисел. Для этого вводится тип *Complex\_Type*, который формально является строковым, но фактически применяется для хранения двух числовых значений — вещественной и мнимой частей комплексного числа. Отсюда значение параметра длины строки *2\*SizeOf(float)*. Такое представление дает возможность определить функции, возвращающие комплексные значения. Получить доступ к вещественной и мнимой частям комплексных значений позволяет другой, введенный в интерфейсной части модуля тип *Cmplx\_Type*. Наличие первого поля *leng* здесь дает возможность использовать механизм приведения типов, а второе и третье поля, собственно говоря, и представляют вещественную и мнимую части. Здесь необходимо сделать оговорку. В данном случае используется нестандартный прием, цель которого — сделать программу более эффективной. Так инженер-электронщик иногда использует нестандартные режимы работы микросхем для того, чтобы повысить эффективность их работы, а бизнесмен использует недостаточно эффективную работу налоговой инспекции для повышения своей прибыли. Но микросхема может «сгореть», а бизнесмен... В программировании тоже приходится прибегать к «трюкам», но делать это следует аккуратно. Впрочем, вернемся к нашему модулю.

**Листинг 3.9.** Интерфейсная секция модуля *cmplx* для работы с комплексными числами

```
{$N+}
unit cmplx;

interface

type
  float = double;
  Complex_Type = string[2*sizeof(float)];
  Cmplx_Type = record
    leng : byte;
    r, i : float;
  end;

  function Re(z : Complex_Type) : Float;
  function Im(z : Complex_Type) : Float;
  function Complex(x, y : float) : Complex_Type;
  function Complex_to_Str(z : Complex_Type; d : ShortInt) : string;
  function CSum(x, y : Complex_Type) : Complex_Type;
  function CDiff(x, y : Complex_Type) : Complex_Type;
```

```

function CProd(x, y : .Complex_Type) : Complex_Type;
function CModulus(z: complex): Real;
procedure CPolar(z: complex; var r, theta: Real);
procedure CPrint (z: complex; n: Word);

```

Обратимся к секции реализации модуля `cmplx`. Функция `Complex` (листинг 3.10) выполняет преобразование двух вещественных значений `x` и `y` (вещественная и мнимая части) в одно комплексное.

#### Листинг 3.10. Функция Complex модуля cmplx

```

implementation

function Complex(x, y : float) : Complex_Type;
var
    result : Cmplx_Type;
begin
    with result do
    begin
        lengl := 2 * SizeOf(float);
        r := x;
        i := y;
    end;
    Complex := Complex_Type(result);
end;

```

Функция `Complex_to_Str` (листинг 3.11) выполняет преобразование численного значения в строковое. Данная операция может оказаться полезной, например, при отображении комплексных чисел в графическом режиме. Первым параметром функции `Cx2Str` является комплексное значение, а второй определяет формат вывода. Если  $d > 0$ , значения вещественной и мнимой частей выводятся в формате без экспоненты с  $d$  знаками после десятичной точки. При  $d < 0$  значения выводятся с экспонентой, то есть в виде `bbbb.sssssEpp`, где `b` — пробелы, `s` — знак, `v.aaaa` — значение, `E` — символ основания 10, `s` — знак показателя экспоненты и `pp` — показатель. Назначение полей формата `:0:d` в первых аргументах процедуры `Str` то же, что и в параметрах процедур вывода `Write` и `WriteLn`. Равенство нулю первого значения означает автоматический выбор числа позиций, отводимых под все значение.

#### Листинг 3.11. Функция Complex\_to\_Str модуля cmplx

```

function Complex_to_Str(z : Complex_Type; d : ShortInt) : String;
var
    result1, result2 : string;
begin
    if d > 0 then
    begin
        Str(Re(z) : 0 : d, result1);
        Str(Im(z) : 0 : d, result2);
    end
    else

```

```

begin
  Str(Re(z) : (d+6), result1);
  Str(Im(z) : (d+6), result2);
end;
if Im(z) < 0 then
  Complex_to_Str := result1 + result2 + 'i'
else
  Complex_to_Str := result1 + '+' + result2 + 'i';
end;

```

Функции `Re` и `Im` выделяют, соответственно, вещественную и мнимую части комплексного значения. К компонентам комплексного числа можно было бы обратиться и как к полям записи, но функции `Re` и `Im` — более наглядный способ выполнения данной операции.

#### Листинг 3.12. Функции `Re` и `Im` модуля `cmplx`

```

function Re(z : Complex_Type) : float;
begin
  Re := Cmplx_Type(z).r;
end;

function Im(z : Complex_Type) : float;
begin
  Im := Cmplx_Type(z).i;
end;

```

Следующая группа функций (`Csum`, `Cdiff` и `Cprod`, см. листинг 3.13) предназначена для вычисления суммы, разности и произведения комплексных значений.

#### Листинг 3.13. Функции комплексной арифметики модуля `cmplx`

```

function CSum(x, y : Complex_Type) : Complex_Type;
begin
  CSum := Complex(Re(x) + Re(y), Im(x) + Im(y));
end;

function CDiff(x, y : Complex_Type) : Complex_Type;
begin
  CDiff := Complex(Re(x) - Re(y), Im(x) - Im(y));
end;

function CProd(x, y : Complex_Type) : Complex_Type;
begin
  CProd := Complex(
    Re(x) * Re(y) - Im(x) * Im(y),
    Re(x) * Im(y) + Im(x) * Re(y)
  );
end;

```

Листинг 3.14 содержит исходные тексты остальных функций и процедур нашего модуля. Функция `CAbs` предназначена для вычисления абсолютной величины комплексного числа.

Функция `CPolar_form` позволяет получить экспоненциальное представление комплексного значения  $c = r e^{i\theta}$ . Входным параметром является комплексное значение, а результат работы —  $r$  и  $\theta$ .

И наконец, последняя процедура `CPrint` предназначена для вывода комплексных значений на экран.

#### Листинг 3.14. Функции `CAbs`, `CPolar_form` и `CPrint` модуля `cmplx`

```
function CAbs(z: Complex_Type): float;
begin
    CAbs := Sqrt(Sqr(Re(z)) + Sqr(Im(z)));
end;

procedure CPolar_form(z: Complex_Type; var r, theta: float);
begin
    r := CAbs(z);
    if Re(z) > 0 then
        theta := Arctan(Im(z) / Re(z))
    else
        if Re(z) < 0 then
            theta := Pi + Arctan(Im(z) / Re(z))
        else
            if Im(z) >= 0 then
                theta := 0.5*Pi
            else
                theta := - 0.5 * Pi
    end;

    procedure CPrint(z: Complex_Type; n: Word);
    begin
        if Re(z) < 0.0 then
            Write(' - '. Abs(Re(z)):(n + 2):n)
        else
            Write(' + '. Re(z):(n + 2):n);
        if Im(z) < 0.0 then
            Write(' - '. Abs(Im(z)):(n + 2):n)
        else
            Write(' + '. Im(z):(n + 2):n);
        Write('*i');
        WriteLn;
    end;
begin
    CPrint(z, 10);
end.
```

## Записи с вариантами

До сих пор мы рассматривали записи с фиксированными частями. Называются они так потому, что в различных ситуациях имеют одинаковую структуру. В Паскале имеется и другая разновидность записей — *записи с вариантами* (с перемен-

*ными или вариантными* частями). В различных ситуациях такие записи могут иметь разную структуру.

Описание типа «запись с вариантами» имеет следующий вид:

```
type
  rec = record
    {Описания фиксированных частей}
    v1, v2 : Integer;
    {Описание вариантной части}
    case n: Word of
      0: (Список полей – описаний переменных);
      1: (Список полей – описаний переменных);
      .....
      7: (Список полей – описаний переменных);
  end;
```

В этом случае описание включает вариантную часть, в которой используется оператор `case`. Описание завершается единственным зарезервированным словом `end`. В качестве селектора оператора `case` используется идентификатор типа. В нашем случае это `n: Word of`. Переменная часть должна следовать после фиксированной части, если таковая имеется. Пример записи с вариантами приводится ниже.

#### Листинг 3.15. Использование записи с вариантами

```
program rec_var_example;

type
  seasons = (spring, summer, autumn, winter);
  recvar = record
    Date : Byte;
    case time_for_vacations : seasons of
      spring : (ma : (March, April, May));
      summer : (grandpa : (June, July, August));
      autumn : (grandma : (September, October, November));
      winter : (pa : (December, January, February));
  end;

var
  my_family : recvar;

begin
  my_family.time_for_vacations := spring;
  my_family.ma := May;

  my_family.time_for_vacations := winter;
  my_family.pa := December;

  my_family.time_for_vacations := summer;
  my_family.grandpa := August;
```

```
my_family.time_for_vacations := autumn;
my_family.grandma := September;
end.
```

Здесь вначале определяется перечисляемый тип *seasons*, который затем используется в описании вариантовой части записи. Тип *recvar* представляет собой запись, которая состоит из фиксированной и вариантовой частей. Присваивая полю записи *my\_family.time\_for\_vacations* одно из допустимых значений типа *seasons*, мы формируем запись, у которой поле *Date* присутствует всегда, а другие поля будут зависеть от значения вышеупомянутого поля.

Говоря о вариантовых полях, следует отметить, что это всего лишь разные способы представления одних и тех же данных. Значение селектора не ограничивает возможности обращения к разным вариантам, он вообще может не являться полем, а быть пустым.

### Упражнение 3.7

Напишите программу для проверки модуля *strlx*.

## Указатели

Все те переменные, с которыми мы до сих пор имели дело, являлись *статическими*. Такие переменные размещаются в памяти при запуске программы и находятся там постоянно во время ее выполнения. Использование статических переменных не позволяет эффективно использовать память компьютера. *Указатели* позволяют создавать переменные во время выполнения программы, то есть *динамически*. При необходимости можно размещать в памяти новые переменные и освобождать память, когда необходимость в них отпадает. Таким образом программист получает возможность более гибко использовать память компьютера, но платой за это является потеря наглядности программы и увеличение ее сложности.

Обращение к динамическим переменным производится не по имени, а по их адресу в памяти. Для хранения адреса динамической переменной используется специальный *ссыпочный* тип, а переменная ссылочного типа называется *указателем*. Так как указатель — это адрес памяти, он занимает двойное слово (сегмент и смещение). Описание ссылочного типа имеет следующий вид:

```
type zilch_ptr = ^type_ID;
```

где *type\_ID* — это идентификатор типа, который называется *базовым*. Например:

```
type
  Int_Point = ^Integer;
var
  What, Where, When : ^Integer;
  Pt1, Pt2, Pt3 : Int_Point;
```

В данном случае базовый тип — `Integer`, а в предложении описания типа определяется тип `Int_Point` — указатель на `Integer`. Обе строки в предложении `var` эквивалентны.

С помощью указателей можно создавать графоподобные структуры данных, работать с векторами, полиномами, матрицами и другими объектами, размер которых заранее не известен, а определяется во время выполнения программы.

Чтобы обратиться к переменной через указатель, нужно произвести разыменование последнего, поставив справа от идентификатора ссылочной переменной символ «шляпки» (^).

Ниже приводится пример простой программы, в которой используются указатели. Программа позволяет определить день недели по дате.

**Листинг 3.16.** Программа определения дня недели по дате

```
program day_of_week;

uses crt;

const
  DayOfWeek : array [0..6] of string = (
    'понедельник', 'вторник', 'среда', 'четверг',
    'пятница', 'суббота', 'воскресенье'
  );

type
  pString = ^string;

var
  year, month, day : word;

function IntDow(yyyy, mm, dd : integer) : integer;
var
  AddVal : shortint;
begin
  if mm < 3 then
    AddVal := 1
  else
    AddVal := 0;
  IntDow := (((3 * yyyy - (7 * (yyyy + (mm + 9) div 12)) div 4 +
    (23 * mm) div 9 + dd + 2 + ((yyyy - AddVal) div 100 + 1) *
    3 div 4 - 16) mod 7));
end;

function StrDow(yyyy, mm, dd : integer): pString;
begin
  StrDow := @DayOfWeek[IntDow(yyyy, mm, dd)];
end;

begin
```

```

clrscr;
textcolor(7);
WriteLn;
Write(' Год : ');
ReadLn(year);
Write(' Месяц : ');
ReadLn(month);
Write(' День : ');
ReadLn(day);
WriteLn;
textcolor(11);
WriteLn(day,'.',month,'.',year,' - это ', StrDow(year, month, day)^);
WriteLn('Нажмите <Enter>:');
ReadLn;
end.

```

Вот еще один пример программы с применением указателей. В этой программе следует обратить внимание на то, что тип указателя определяется *до* определения базового типа.

#### Листинг 3.17. Использование указателей

```

program ptr_prog;

const
  max = 1000;

type
  vector_ptr = ^vector;
  vector = array [1..max] of LongInt;

var
  VP: vector_ptr;
  i, j: LongInt;
  k: Word;

begin
  i := MemAvail;
  WriteLn('Перед размещением свободно ', i:7, ' байт');
  New(VP);
  j := MemAvail;
  WriteLn('После размещения свободно ', j:7, ' байт');
  k := i - j;
  WriteLn('Размещено ', k:7, ' байт');
  for k := 1 to max do VP^ [k] := 2 * k + 1;
  Dispose(VP);
  VP := nil;
  i := MemAvail;
  WriteLn('После освобождения свободно ', i:7, ' байт');
  WriteLn('Нажмите <Enter>:');
  ReadLn;
end.

```

Программа размещает в памяти динамическую переменную и сообщает о размере свободной памяти до и после размещения, а также о том, какое место в памяти резервируется под эту переменную. Первое, на что следует обратить внимание в данной программе, — это стандартная процедура `New(P)`. Она отводит память в динамически распределяемой области под переменную того типа, на который указывает ее параметр-указатель `P`. `P` присваивается значение адреса этой переменной, а `P^` становится новой переменной. Получение значения переменной, на которую ссылается данный указатель, называют *разыменованием*.

Первое описание типа в программе — это `vector_ptr`, определяемый как указатель на пока еще не определенный тип `vector`. В следующей строке `vector` описывается как массив, состоящий из `max` значений типа `LongInt`.

Основная часть программы содержит вычисления, предназначенные для того, чтобы показать, что происходит с доступной памятью. Если опустить эти вычисления, в теле программы останутся следующие операторы:

```
begin
  New(VP);
  for k := 1 to max do
    VP^ [k] := 3*k;
  Dispose(VP);
  VP := nil;
end.
```

После того как процедура `New` отвела память под переменную, `VP^` является именем массива, и чтобы подчеркнуть это обстоятельство, в следующем операторе его элементам присваиваются значения. Действие процедуры `Dispose` обратно по отношению к действию процедуры `New` — она освобождает память, отведенную под динамическую переменную, и делает эту память доступной для другого использования. Следующий оператор присваивает `VP` значение `nil`. Эта встроенная константа представляет значение указателя, который ни на что не указывает. После того как к указателю применена процедура `Dispose`, обязательно следует присвоить ему значение `nil`.



### ВНИМАНИЕ

Работать с указателями надо аккуратно. Вот несколько советов. Следует использовать оператор `Dispose` сразу же после того, как необходимость в использовании динамической переменной отпадает. Это правило имеет даже более серьезное значение, чем требование сопровождать каждое слово `begin` словом `end`, так как без `end` компилятор выдаст сообщение об ошибке и не даст вам двинуться дальше, но он не отметит отсутствия `Dispose`. Нельзя обращаться к уже уничтоженной динамической переменной. Попытка такого обращения может привести к краху операционной системы.

А сейчас перейдем к более серьезному использованию указателей. Напишем программу для перемножения прямоугольных целочисленных матриц, размер которых заранее не известен, но может быть определен динамически. Сообщу (а читателям, знакомым с высшей математикой, напомню) некоторые сведения из матема-

тиki. *Матрицей* называется совокупность вещественных или комплексных чисел, расположенных в виде прямоугольной таблицы:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$$

Произведение матриц представляет собой матрицу, элементы которой вычисляются по следующей формуле:

$$b_{ij} = \sum_{k=1}^n a_{ik} c_{kj}.$$

Размеры перемножаемых матриц должны быть согласованы (количество столбцов первого сомножителя равно количеству строк второго). Процедуры *New* и *Dispose* в этой ситуации не подходят, так как размер переменных заранее не известен. Здесь требуется что-нибудь другое, и это другое — стандартные процедуры *GetMem* и *FreeMem*. Процедура *GetMem(var P: Pointer; Size: Word)* создает динамическую переменную указанного размера и присваивает значение адреса переменной *P* типа «указатель». Может оказаться, что свободного места в памяти не хватает для размещения переменной. В этом случае при работе программы появляется сообщение об ошибке. Процедура *FreeMem(var P: Pointer; Size: Word)* освобождает область памяти, отведенную под динамическую переменную. После вызова этой процедуры попытка обращения к указателю приведет к ошибке. Следует иметь в виду, что, как и в случае процедур *New* и *Dispose*, каждый вызов *GetMem* должен сопровождаться вызовом *FreeMem*.

Раздел описаний программы *matrices* приведен в листинге 3.18.

#### Листинг 3.18. Раздел описаний программы вычисления суммы и произведения матриц

```
program matrices;

type
  row_index = array[1..100] of ^row_vector;
  row_vector = array[1..100] of Integer;
  matrix = record
    no_rows, no_cols: Word;
    index_ptr: ^row_index;
  end;

  var
    A, B, C: matrix;
    inds_size, elems_size, i, k: Word;
    m, n: LongInt;
```

Здесь предполагается, что наибольший размер матрицы равен 100×100. Фактический размер переменной типа *matrix* составляет 8 байт (2 значения типа *Word* и один указатель). Указатель является ссылкой на вектор-столбец, длина которого

равна числу строк матрицы. Каждый элемент столбца сам является указателем на вектор-строку. Таким образом, если  $A$  — матрица, то ссылка на ее элемент  $a_{ik}$  имеет вид

`A.index_ptr^[i]^k`

Матрицы хранятся в переменных  $A$ ,  $B$  и  $C$  типа `matrix`. Элементам матриц  $A$  и  $B$  начальные значения присваиваются случайным образом. Матрица  $C$  является произведением матриц  $A$  и  $B$ .

Перейдем к обсуждению процедур, используемых программой `matrices`. Первые две процедуры — `create_matrix` и `delete` — предназначены для создания и уничтожения матрицы размера  $r \times c$ , здесь используются процедуры `GetMem` и `FreeMem`. Их тексты приведены в листинге 3.19.

#### Листинг 3.19. Процедуры `create_matrix` и `delete` программы `matrices`

```
procedure create_matrix(var A: matrix; r, c: Word);
var
  j, k: Word;
begin
  with A do
    begin
      no_rows := r; no_cols := c;
      j := no_rows * sizeof(Pointer);
      inc(inds_size, j);
      GetMem(index_ptr, j);
      for j := 1 to no_rows do
        begin
          k := no_cols * sizeof(integer);
          inc(elems_size, k);
          GetMem(index_ptr^[j], k);
          for k := 1 to no_cols do
            index_ptr^[j]^k := -5 + random(11);
        end;
    end;
end;

procedure delete(var A: matrix);
var
  j: Word;
begin
  with A do
    begin
      for j := 1 to no_rows do
        FreeMem(index_ptr^[j], no_cols * sizeof(integer));
      FreeMem(index_ptr, no_rows * sizeof(Pointer));
      index_ptr := nil;
    end;
end;
```

Поскольку при вызове процедур `GetMem` и `FreeMem` необходимо задавать объем памяти в байтах, который следует выделить или освободить, полезно использовать функцию `SizeOf` для вычисления необходимого объема памяти.

Процедура `create_matrix` формирует матрицу и размещает в памяти массив, используемый для ее хранения, а процедура `delete` удаляет этот массив из памяти. Замечу, что здесь для процедуры удаления матрицы выбрано не совсем удачное имя, поскольку оно совпадает с именем встроенной процедуры модуля `System`. Такое дублирование имен допускается, и если в программе, тем не менее, понадобится обращение к процедуре `delete` из модуля `System`, это обращение должно иметь вид `System.delete(...)`. Но «стилистически» более правильным решением было бы присвоение нашей процедуре имени `delete_matrix`. Следующие две процедуры — `product_matrix` и `print_matrix` — предназначены для перемножения матриц и для вывода матрицы на экран.

**Листинг 3.20.** Процедуры `product_matrices` и `print_matrix` программы `matrices`

```
procedure product_matrix(var A, B, C: matrix);
var
  i, j, k: Word;
  sum: LongInt;
begin
  if A.no_cols <> B.no_rows then halt;
  create_matrix(C, A.no_rows, B.no_cols);
  for i := 1 to A.no_rows do
    for k := 1 to B.no_cols do
      begin
        sum := 0;
        for j := 1 to A.no_cols do
          sum := sum +
            A.index_ptr^[i]^*[j] * B.index_ptr^[j]^*[k];
        C.index_ptr^[i]^*[k] := sum;
      end;
  end;

procedure print_matrix(var A: matrix; ss: string);
var
  i, k: Word;
begin
  with A do
  begin
    writeln(ss);
    for i := 1 to no_rows do
    begin
      for k := 1 to no_cols do
        write(index_ptr^[i]^*[k]:4);
      writeln;
    end;
  end;
end;
```

В теле программы (листинг 3.21) демонстрируется применение вышеперечисленных процедур.

#### Листинг 3.21. Тело программы matrices

```

begin
  inds_size := 0; elems_size := 0;
  m := MemAvail;
  WriteLn('Перед размещением свободно ', m:8, ' байт');
  randomize;
  create_matrix(A, 4, 5);
  create_matrix(B, 5, 3);
  product_matrix(A, B, C);
  n := MemAvail;
  WriteLn('После размещения свободно ', n:8, ' байт');
  WriteLn('Размещено ', m-n:8, ' байт');
  WriteLn('Указатели ', inds_size:8, ' байт');
  WriteLn('Матричные элементы ', elems_size:8, ' байт');
  m := m - n - inds_size - elems_size;
  print_matrix(A, 'Матрица A = ');
  print_matrix(B, 'Матрица B = ');
  print_matrix(C, 'Произведение матриц A*B = ');
  delete(C);
  delete(B);
  delete(A);
  m := MemAvail;
  WriteLn('После освобождения свободно ', m:8, ' байт');
  WriteLn('Нажмите <Enter>');
  ReadLn;
end.

```

Переменные `inds_size` и `elems_size` используются для подсчета памяти, отводимой под указатели и матричные элементы.

#### Упражнение 3.8

Пусть  $A$  является матрицей размера  $m \times n$ ,  $v$  —  $m$ -компонентным вектором-строкой, а  $w'$  —  $n$ -компонентным вектором-столбцом. Напишите функцию для вычисления произведения  $vA w'$ .

#### Упражнение 3.9

Для симметричной матрицы размера  $n \times n$  достаточно хранить не  $n^2$  символов, а только  $n(n+1)/2$ . Учитывая это, модифицируйте программу `matrices` для работы с симметричными матрицами.

## Связные списки

Связный список представляет собой цепочку записей («узлов»), в которой каждая запись содержит, кроме основных данных, ссылку на следующую запись в

цепочке. Во главе списка находится указатель, который часто называется «корнем» и который указывает на первую запись в списке (рис. 3.1).

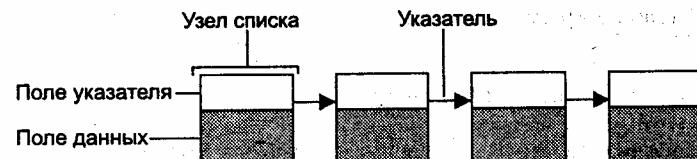


Рис. 3.1. Структура связного списка

Указателю в последней записи списка обычно присваивается значение `nil`, служащее признаком конца списка. Такая структура является динамической, она может изменяться в процессе выполнения программы.

Изменение списка обеспечивается операциями включения записи в список и удаления записи из списка. Для того чтобы добавить в связный список новый узел, достаточно изменить один указатель, сами узлы при этом перемещаться не должны (рис. 3.2).

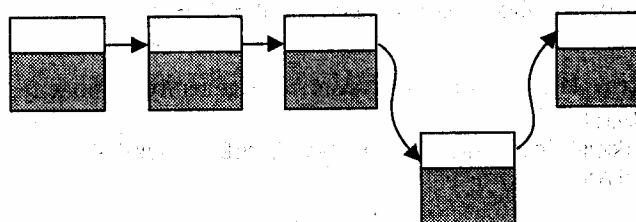


Рис. 3.2. Включение нового узла в связный список

Удаление узла также осуществляется изменением соответствующего указателя так, чтобы он ссылался на узел, следующий за удаляемым (рис. 3.3).

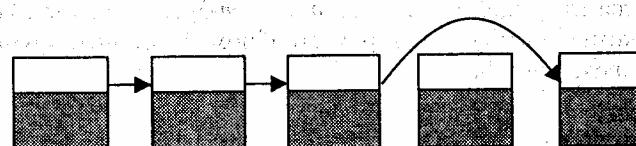


Рис. 3.3. Удаление узла из связного списка

Ниже приводится исходный код модуля для работы со связанными списками. Начнем с интерфейсной секции, текст которой приведен в листинге 3.22.

### Листинг 3.22. Интерфейсная секция модуля list для работы со связанными списками

```

unit list;
interface
type

```

```

node_ptr = ^node;
node = record
  coefficient: Real;
  degree: Word;
  next: node_ptr;
end;

var
  root: node_ptr;

procedure create_list(n: Word);
procedure destroy_list;
procedure insert_after(var q, s: node_ptr);
procedure insert_before(var q, s: node_ptr);
procedure insert_at_end(var s: node_ptr);
procedure remove_after(q: node_ptr);
procedure remove(var q: node_ptr);

```

Здесь тип `node` представляет собой запись с полями `coefficient`, `degree` и `next`. Назначение этих полей я объясню позже.

Процедура `create_list` (листинг 3.23) создает и загружает список, состоящий из `n` узлов с корнем во главе. Она добавляет каждый новый узел в начало списка.

Процедура `destroy_list` освобождает память, выделенную процедурой `create_list`.

#### Листинг 3.23. Процедуры создания и удаления списков модуля list

```

implementation

procedure create_list(n: Word);
var
  temp: node_ptr;
  k: Word;
begin
  root := nil;
  for k := 0 to n - 1 do
  begin
    temp := root;
    New(root);
    with root^ do
    begin
      coefficient := 100.0*Random - 50.0;
      degree := k;
      next := temp;
    end;
  end;
end;

procedure destroy_list;
var
  temp: node_ptr;

```

```

begin
  while root <> nil do
    begin
      temp := root^.next;
      Dispose(root);
      root := temp;
    end;
  end;
end;

```

Далее идут процедуры подстановки (листинг 3.24). Первая, `insert_after`, подставляет узел `s^` в список сразу же за узлом `q^`, на который указывает переменная `q` типа `node_ptr`.

Процедура `insert_before` для данного `q`, указывающего на узел, содержащийся в списке, подставляет `s^` перед `q^`. Делается это следующим образом. Вначале содержимое указателя на узел `q` сохраняется в локальной переменной `temp`. Затем этот узел замещается новым (`s`), а следующим становится узел, находящийся в `temp` (то есть `q`).

Для того чтобы добавить узел в конец списка, необходимо вначале пройти весь список, чтобы найти этот конец, и лишь затем добавить указатель. Этот алгоритм реализован в процедуре `insert_at_end`.

#### Листинг 3.24. Процедуры подстановки узлов модуля `list`

```

procedure insert_after(var q, s: node_ptr);
begin
  s^.next := q^.next;
  q^.next := s;
end;

procedure insert_before(var q, s: node_ptr);
var
  temp: node_ptr;
begin
  New(temp);
  temp^.next := q^.next;
  q^.next := temp^.next;
  temp^.next := s;
  Dispose(s); s := nil;
end;

procedure insert_at_end(var s: node_ptr);
var
  run: node_ptr;
begin
  if root = nil then
    root := s
  else
    begin
      run := root;
      while run^.next <> nil do
        run := run^.next;
      run^.next := s;
    end;
end;

```

```

    run := run^.next;
  run^.next := s;
end;
s^.next := nil;
end;

```

Далее (листинг 3.25) следуют описания процедур удаления заданного узла, а также узла, идущего за заданным узлом.

#### Листинг 3.25. Процедуры удаления узлов модуля list

```

procedure remove(var q: node_ptr);
var
  temp: node_ptr;
begin
  temp := q^.next;
  if temp <> nil then
    begin
      q^.next := temp^.next;
      Dispose(temp);
    end
  else
    begin
      if q = root then
        begin
          root := nil;
          Dispose(q);
        end
      else
        begin
          temp := root;
          while temp^.next <> q do temp := temp^.next;
          remove_after(temp);
        end;
      q := nil;
    end;
end;

procedure remove_after(q: node_ptr);
var
  temp: node_ptr;
begin
  temp := q^.next;
  if temp <> nil then
    begin
      q^.next := temp^.next;
      Dispose(temp);
    end;
end;

```

#### Упражнение 3.10

Напишите программу для проверки процедур модуля list.

**Упражнение 3.11**

Добавьте в модуль `list` процедуру

```
procedure skip(var p, q: node_ptr);
```

Здесь `p` и `q` являются указателями в списке, причем `q` — потомок `p`. Процедура должна удалять все узлы между `p` и `q`.

## Модуль для вычислений с полиномами

Следующий пример содержит набор процедур для операций с вещественными полиномами с любым числом членов любых степеней. При работе с полиномами, если заранее не известен их порядок (максимальная степень), нельзя использовать статические переменные, например массивы. В этом случае пришлось бы заранее зарезервировать определенный объем памяти, исходя из максимального порядка полинома, который может и не встретиться в процессе работы. Связный список в этом случае представляется естественной структурой данных. По этой причине мы будем использовать представление, в котором полином является связным списком членов, расположенных по убыванию степени, начиная от корня.

В процедурах и функциях модуля `realpoly` полином будет представлен указателем на запись, содержащую степень, коэффициент и указатель на следующий член. Для этого вводится тип `polynomial`. В последнем члене используется стандартный указатель `nil`. Полином `P = nil` интерпретируется как нулевой полином. Степень задается целой неотрицательной константой. Отрицательные значения степени, например `-1`, используются для завершения работы процедур ввода. Вначале обратимся к интерфейсной части модуля `realpoly`. Ее исходный текст содержится в листинге 3.26.

**Листинг 3.26.** Интерфейсная секция модуля `realpoly` для вычислений с вещественными полиномами

```
{$N+}
unit realpoly;
interface
type
  float = extended;
  polynomial = ^term;
  term = record
    degree: integer;
    coefficient: float;
    next: polynomial
  end;
  function degree_poly(p: polynomial): integer;
  function leading_coeff(p: polynomial): float;
```

```

procedure insert_term(t: term; var p: polynomial; augment: boolean);
procedure load(var p: polynomial);
procedure recycle(var p: polynomial);
function copy_poly(p: polynomial): polynomial;
procedure print_poly(p: polynomial);
function mult_by_scalar(c: float; p: polynomial): polynomial;
function sum_poly(p, q: polynomial): polynomial;
function product_poly(p, q: polynomial): polynomial;
function derivative_poly(p: polynomial): polynomial;
procedure division_poly(f, g: polynomial; var q, r: polynomial);
function value_poly(p: polynomial; c: float): float;
procedure evaluate_poly(p: polynomial; c: float; var D0, D1: float);

```

Первые две процедуры — `degree_poly` и `leading_coeff` (листинг 3.27) — предназначены для определения порядка полинома и значения коэффициента при члене с наибольшей степенью.

#### Листинг 3.27. Процедуры `degree_poly` и `leading_coeff` модуля `realpoly`

```

implementation
function degree_poly(p: polynomial): integer;
begin
  if p = nil then
    degree_poly := -1
  else
    degree_poly := p^.degree;
end;
function leading_coeff(p: polynomial): real;
begin
  if p = nil then
    leading_coeff := 0.0
  else
    leading_coeff := p^.coefficient;
end;

```

Процедура `insert_term` (листинг 3.28) подставляет в список новый узел, соответствующий новому добавляемому члену. Первый параметр процедуры представляет добавляемый член, второй является полиномом. Третий параметр булев, значение `true` для него означает подстановку нового члена, а `false` — замещение имеющегося члена новым.

#### Листинг 3.28. Процедура `insert_term` модуля `realpoly`

```

procedure insert_term(t: term; var p: polynomial; augment: boolean);
var
  run, temp: polynomial;
begin
  if p = nil then
  begin
    new(p);
    p^. := t;
  end
  else
  begin
    if augment then
      run := p^.next;
    else
      run := p;
    temp := run^.next;
    run^.next := t;
    t^.next := temp;
  end;
end;

```

```

p^.next := nil;
end
else
begin
    run := p;
    while (run^.degree > t.degree) and (run^.next <> nil) do
        run := run^.next;
    if run^.degree < t.degree then
    begin
        new(temp);
        temp^ := run^;
        run^ := t;
        run^.next := temp;
    end
    else
        if run^.degree = t.degree then
        begin
            if augment then
                run^.coefficient := run^.coefficient + t.coefficient
            else
                run^.coefficient := t.coefficient;
        end
        else
            begin
                new(run^.next);
                run := run^.next;
                run^ := t;
                run^.next := nil;
            end;
    end;
end;

```

Листинг 3.29 содержит исходные тексты процедур `load`, `recycle`, `copy_poly` и `print_poly`. Процедуры `load` и `recycle` предназначены для создания нового полинома и для удаления уже существующего полинома соответственно.

**Листинг 3.29.** Процедуры `load`, `recycle`, `print_poly` и функция `copy_poly` модуля `realpoly`

```

procedure load(var p: polynomial);
var
    t: term;
    k: integer;
begin
    p := nil;
    t.next := nil;
    repeat
        write('Введите степень нового члена (или -1 для выхода): ');
        readln(k);
        if k >= 0 then
        begin
            with t do
            begin

```

```
degree := k;
write('Введите коэффициент: ');
readln(coefficient);
end;
insert_term(t, p, false);
end;
until k < 0;
end;
procedure recycle(var p: polynomial);
begin
  if p <> nil then
    begin
      recycle(p^.next);
      dispose(p);
    end;
  p := nil;
end;
function copy_poly(p: polynomial): polynomial;
var
  s: polynomial;
begin
  if p = nil then
    s := nil
  else
    begin
      new(s);
      s^. := p^.;
      s^.next := copy_poly(p^.next);
    end;
  copy_poly := s;
end;
procedure print_poly(p: polynomial);
var
  run: polynomial;
  k: integer;
begin
  if p = nil then
    writeln(0:2)
  else
    begin
      with p^ do
        begin
          write(coefficient:5:5);
          if degree > 0 then write(' x^', degree);
          run := next;
        end;
      k := 4;
      while run <> nil do
        begin
          dec(k);
          if k = 0 then
            begin
```

```

k := 5;
writeln;
write(' ':15);
end;
with run^ do
begin
  if coefficient < 0.0 then
    write(' - ', -coefficient:5:5)
  else
    write(' + ', coefficient:5:5);
  if degree > 0 then
    write(' x^', degree);
  run := next;
end
end;
writeln;
end
end;

```

В процедуре `recycle` используется рекурсия, более подробный разговор о которой нам еще предстоит. Ну а пока достаточно заметить, что рекурсия — это вызов процедурой самой себя.

Функция `copy_poly` создает копию полинома. В ней также используется рекурсия. При копировании связного списка необходимо скопировать все его узлы.

Процедура `print_poly` используется для отображения на экране полинома, заданного в качестве ее параметра. Степень обозначается символом «шляпка» (^).

Функции `mult_b_scalar` и `sum_poly` (листинг 3.30) предназначены для умножения полинома на скаляр и суммирования полиномов. Здесь следует напомнить, что умножение полинома на число выполняется путем умножения каждого коэффициента на это число, степени при этом не затрагиваются. Суммой двух полиномов считается новый полином, который содержит члены, имеющиеся как в первом, так и во втором полиномах, причем, если члены определенной степени есть в обоих полиномах, то их коэффициенты суммируются.

**Листинг 3.30.** Функции умножения полинома на скаляр и суммирования полиномов модуля `realpoly`

```

function mult_by_scalar(c: float; p: polynomial): polynomial;
var
  r, s: polynomial;
begin
  s := copy_poly(p);
  r := s;
  while r <> nil do
    with r^ do
    begin
      coefficient := c*coefficient;
      r := next
    end;
end;

```

```

mult_by_scalar := s;
end;
function sum_poly(p, q: polynomial): polynomial;
var
  p_run, s: polynomial;
  t: term;
begin
  s := copy_poly(p);
  p_run := q;
  while p_run <> nil do
  begin
    t := p_run^;
    t.next := nil;
    insert_term(t, s, true);
    p_run := p_run^.next;
  end;
  sum_poly := s;
end;

```

Функция `product_poly` (листинг 3.31) вычисляет произведение двух полиномов. При программировании этой операции следует помнить о том, что произведение двух полиномов состоит из большего числа членов, чем каждый из сомножителей.

**Листинг 3.31.** Функция вычисления произведения полиномов модуля `realpoly`.

```

function product_poly(p, q: polynomial): polynomial;
var
  p_run, q_run, s: polynomial;
  t: term;
begin
  s := nil;
  t.next := nil;
  p_run := p;
  while p_run <> nil do
  begin
    q_run := q;
    while q_run <> nil do
    begin
      t.degree := p_run^.degree + q_run^.degree;
      t.coefficient :=
        p_run^.coefficient*q_run^.coefficient;
      insert_term(t, s, true);
      q_run := q_run^.next;
    end;
    p_run := p_run^.next;
  end;
  product_poly := s;
end;

```

Другой важной операцией над полиномами является деление одного полинома  $F(x)$  на другой  $G(x)$ . Результатом такой операции будут частное  $Q(x)$  и остаток  $R(x)$ :

$$F(x) = G(x) \cdot Q(x) + R(x).$$

Первые два параметра процедуры `division_poly` (листинг 3.32) — делимое и делитель, а следующие — частное и остаток соответственно.

#### Листинг 3.32. Функция `division_poly` модуля `realpoly`

```
procedure division_poly(f, g: polynomial;
  var q, r: polynomial);
var
  run, h, temp: polynomial;
  t: term;
  gap, d: word;
  a: float;
begin
  q := nil;
  d := degree_poly(g);
  r := copy_poly(f);
  while degree_poly(r) >= d do
    begin
      gap := degree_poly(r) - d;
      a := leading_coeff(r) / leading_coeff(g);
      h := copy_poly(g); run := h;
      repeat
        with run^ do
          begin
            degree := gap + degree;
            coefficient := -a * coefficient;
            run := next;
          end
        until run = nil;
      temp := sum_poly(r, h);
      recycle(r);
      recycle(h);
      r := nil;
      t.degree := gap; t.coefficient := a;
      insert_term(t, q, false);
    end;
  end;
```

Функция `derivative_poly` (листинг 3.33) выполняет дифференцирование полинома. Читатель, знакомый с элементами высшей математики, знает, что результат этой операции — новый полином, полученный в результате почленного дифференцирования исходного.

#### Листинг 3.33. Функция `derivative_poly` модуля `realpoly`

```
function derivative_poly(p: polynomial): polynomial;
var
  run: polynomial;
begin
  if degree_poly(p) = 0 then
```

```

derivative_poly := nil
else
begin
  run := copy_poly(p);
  derivative_poly := run;
repeat
  with run^ do
begin
  coefficient := degree * coefficient;
  dec(degree);
  if next = nil then
    exit;
  if next^.degree = 0 then
  begin
    dispose(next);
    next := nil;
  end;
  run := next;
end;
until run = nil;
end;
end;

```

Листинг 3.34 содержит тексты функций и процедур, предназначенные для вычисления численных значений полиномов.

В функции `value_poly` реализовано вычисление значения полинома в заданной точке `c`. Вторая процедура — `evaluate_poly` — вычисляет значение полинома (параметр `d0`) и его первой производной (параметр `d1`) в заданной точке.

**Листинг 3.34.** Функции `value_poly` и `evaluate_poly` модуля `realpoly`

```

function value_poly(p: polynomial; c: float): float;
var
  i, k: integer;
  s: float;
  run: polynomial;
begin
  if p = nil then
    s := 0.0
  else
    s := p^.coefficient;
  run := p;
  while run <> nil do
  begin
    if run^.next = nil then
      k := degree_poly(run)
    else
      k := degree_poly(run) - degree_poly(run^.next);
    for i := k downto 1 do
      s := s*c;
    run := run^.next;
  end;
end;

```

```

        if run <> nil then
            s := s + run^.coefficient;
        end;
        value_poly := s;
    end;
procedure evaluate_poly(p: polynomial; c: float;
    var D0, D1: float);
var
    i, gap: integer;
    run: polynomial;
begin
    D1 := 0;
    if p = nil then
    begin
        D0 := 0.0;
        exit;
    end;
    D0 := p^.coefficient;
    run := p;
    while run <> nil do
    begin
        if run^.next = nil then
            gap := degree_poly(run)
        else
            gap := degree_poly(run) - degree_poly(run^.next);
        for i := gap downto 1 do
        begin
            D1 := D1*c + D0;
            D0 := D0*c;
        end;
        run := run^.next;
        if run <> nil then
            D0 := D0 + run^.coefficient;
    end;
end;
end.

```

### Упражнение 3.12

Напишите программу для проверки процедур и функций модуля `realpoly`.

### Упражнение 3.13

Очевидным недостатком модуля `realpoly` является то, что если в полином подставляется член с нулевым коэффициентом, он обрабатывается наравне с прочими членами. Добавьте процедуру, удаляющую из списка те узлы, которые соответствуют членам полинома с коэффициентами  $|a_i| < e$ , где  $e$  — заданное число.

### Упражнение 3.14

Добавьте в модуль `realpoly` функцию вычисления второй производной полинома.

**Упражнение 3.15**

Добавьте в модуль `realpoly` функцию вычисления производной заданного порядка.

## Работа с памятью

В Паскале имеются функции, используемые для адресации памяти и сбора информации о памяти и адресах различных объектов. Соответствующий перечень дан в табл. 3.2.

**Таблица 3.2.** Функции, используемые для сбора информации об адресах различных объектов

Функция	Описание
<code>@</code>	Действие этой операции заключается в получении указателя, ссылающегося на операнд. Операндом является переменная, процедура или идентификатор функции. Тип результата — указатель. Так, например, применение операции <code>@</code> к переменной возвращает указатель на эту переменную. Для процедур и функций операция <code>@</code> возвращает указатель на точку входа в подпрограмму
<code>@@</code>	Возвращение физического адреса операнда. Так, например, переменная процедурного типа <code>P</code> , как и всякая другая переменная, находится по определенному адресу в памяти. Операция <code>@P</code> , примененная к процедурной переменной, возвращает указатель на точку входа в процедуру. Операция <code>@@P</code> возвращает адрес в памяти, где находится процедурная переменная
<code>Addr</code>	Вычисление адреса переменной, процедуры или функции. Операндом является ссылка на переменную, процедуру или идентификатор функции. Тип результата — указатель, ссылающийся на operand
<code>Seg</code>	Вычисление сегмента в адресе переменной, функции или процедуры. Результат — целое типа <code>Word</code>
<code>Ofs</code>	Вычисление смещения в адресе переменной, функции или процедуры. Результат — целое типа <code>Word</code>
<code>Ptr</code>	Преобразование значений сегмента и смещения в указатель на соответствующий адрес

Моделью памяти DOS является линейная последовательность байтов. Каждый байт этой последовательности имеет адрес, который состоит из адреса сегмента и смещения. Размер сегмента составляет 64 килобайта. Адрес задается следующим образом: `$ssss:$oooo`. Здесь знак доллара \$ обозначает шестнадцатеричное число. При определении адресов обычно используются шестнадцатеричные константы, поэтому для успешной работы в дальнейшем стоит потренироваться в переводе десятичных значений в шестнадцатеричное представление и наоборот. Для того чтобы получить настоящий адрес объекта, необходимо сегментную часть адреса `$ssss` умножить на `$10` и к полученному значению прибавить смещение `$oooo`. Пример такого расчета дан ниже:

\$0FFF:	\$0010
\$0FFF0	
+\$ 0010	
<hr/>	
	\$10000

При работе в DOS следует помнить, что диапазон адресов обычной памяти простирается от \$0 до \$FFFF.

Функция `Addr` создает указатель на адрес своего операнда. То же самое делает и оператор `@`. Функции `Seg` и `Ofs` вычисляют значения сегмента и смещения адреса переменной. Функция `Ptr` выполняет обратное действие — она создает указатель на адрес памяти с заданными значениями сегмента и смещения. Каждая программа имеет свой сегмент данных, и адреса определяются относительно этого сегмента.

Все вышеприведенные операторы могут быть применены и к адресам точек входа процедур и функций.

Листинг 3.35 содержит текст программы, демонстрирующей действие всех упомянутых функций работы с памятью.

#### Листинг 3.35. Работа с памятью

```
program address_demo;
uses crt;
var
  p: pointer;
  x: Real;
begin
  ClrScr;
  WriteLn('=====');
  WriteLn('Демонстрация функций Seg, Ofs, @ и Addr:');
  WriteLn('-----');
  WriteLn('(Сегмент адреса x, Смещение адреса x)');
  WriteLn(Seg(x):13, Ofs(x):13);
  WriteLn('-----');
  p := Addr(x);
  WriteLn('p := Addr(x);');
  WriteLn('(Сегмент адреса p^, Смещение адреса p^)');
  WriteLn(Seg(p^):13, Ofs(p^):13);
  WriteLn('-----');
  p := @x;
  WriteLn('p := @x;');
  WriteLn('(Сегмент адреса p^, Смещение адреса p^)');
  WriteLn(Seg(p^):13, Ofs(p^):13);
  WriteLn('-----');
  WriteLn('(Сегмент адреса p, Смещение адреса p)');
  WriteLn(Seg(p):13, Ofs(p):13);
  WriteLn('=====');
  Write('Нажмите <Enter>:'); ReadLn;
  WriteLn('-----');
  WriteLn('Демонстрация функции Ptr:');
  p := @x;
  WriteLn('(Сегмент адреса x, Смещение адреса x)');
```

```

WriteLn(Seg(x):13, 0fs(x):13);
WriteLn('-----');
p := ptr(Seg(x), 0fs(x));
WriteLn('p := Ptr(Seg(x), 0fs(x))');
WriteLn('(Сегмент адреса p^. Смещение адреса p^)');
WriteLn(Seg(p^):13, 0fs(p^):13);
WriteLn('=====');
Write('Нажмите <Enter>:'); ReadLn;
end.

```

Сначала вычисляются значения сегмента и смещения адреса переменной x с помощью обращений к функциям Seg(x) и 0fs(x). Затем указателю P присваивается значение Addr(x) и проверяется тождественность (по адресу) p<sup>^</sup> и x. Затем выполняется аналогичное сравнение, но с использованием операции @. Далее демонстрируется использование функции Ptr.

## Типизированные константы

В первом уроке мы уже познакомились с использованием *нетипизированных* констант. Нетипизированной константе присваивается определенное значение, которое не может быть изменено в ходе выполнения программы. Отличительной чертой *типованных* констант является то, что они могут использоваться и в левой части оператора присваивания, то есть их значение может быть изменено. Описание типизированных констант выглядит следующим образом:

```

const
  c_1 : type_ID_1 = val_1;
  c_2 : type_ID_2 = val_2;

```

Здесь c\_1, c\_2 — имена констант, type\_ID\_i — идентификаторы типа, а val\_i — значения этих констант. Вот пример описания типизированной константы:

```

const
  last_year : Integer = 1999;

```

Можно считать, что типизированные константы являются переменными, которым начальное значение присваивается в начале выполнения программы. Если типизированная константа используется в процедуре или функции и при этом является локальной, при повторных вызовах ее инициализация выполняться уже не будет. Мы видим, таким образом, что типизированная константа занимает как бы промежуточное положение между «настоящими» константами и «настоящими» переменными. Типизированные константы, в частности, нельзя использовать при указании границ диапазона в отрезках типов.

Типизированные константы могут быть простого типа, а могут быть структурными, например массивами. Вот пример описания одномерного массива — типизированной константы:

```

const
  digits : array[0..9] of char =
    ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

```

Элемент массива в этом случае может быть любого типа, кроме файлового. Если речь идет о многомерных массивах, то в предложении описания такого массива константы его элементы для каждой размерности заключаются в скобки и разделяются запятыми:

```
const
  ar2d : array[1..2, 1..2] of integer = ((22,5623),(1,101));
  ar3d : array[1..2, 1..2, 1..2] of integer = (((0.1),(2,3)),((4.5),(6,7)));
```

Описание константы типа «запись» содержит идентификатор записи и список значений полей, заключенный в скобки. Значение каждого поля предваряется именем поля и двоеточием, а разделяются поля точками с запятой, например:

```
type
  point = record
    x,y: real;
  end;
  const
    center : point = (x: 0.0; y: 0.0);
```

При работе с типизированными константами типа «запись» следует помнить, что в предложении описания константы поля должны указываться в том же порядке, как и в описании типа «запись». Если запись содержит поля файлового типа, то для нее нельзя описать константу.

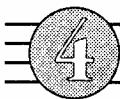
Описание константы множественного типа может содержать несколько элементов, заключенных в квадратные скобки и разделенных запятыми. Каждый элемент такой константы представляет собой константу или отрезок типа:

```
const
  SomeDigits: set of 0..9 = [0, 1, 2, 4];
```

Описание константы ссылочного типа может содержать значение `nil` или указатель, созданный из числовых констант сегмента и смещения с помощью функции `Ptr`.

## Что нового мы узнали?

- Познакомились с определением типов, задаваемых программистом.
- Научились работать с внешними файлами.
- Познакомились с новыми типами данных языка Паскаль: записями и указателями.
- Научились программировать динамические структуры данных (связные списки).
- Познакомились с функциями, используемыми для работы с адресами переменных, процедур и функций.
- Познакомились с типизированными константами.



**УРОК**

## Элементы системного программирования на языке Паскаль

- 
- Операционная система MS-DOS —  
краткий курс
  - Модуль DOS
  - Программирование для MS-DOS и BIOS
  - Мышь
  - Другие устройства
-

**П**од системным программированием мы будем понимать разработку программ, которые выполняют действия, возлагаемые на операционную систему. Это операции с файловой системой, управление выполнением программ, работа с устройствами, жесткими и гибкими дисками, манипулятором «мышь», принтерами и т. д. В этом уроке мы узнаем, как из программы на Паскале можно получить доступ к системным функциям. Мы познакомимся со встроенными процедурами и функциями Турбо Паскаля, позволяющими выполнять некоторые системные операции, научимся создавать и свои программы, напрямую работающие с операционной системой MS-DOS.

## **Операционная система MS-DOS — краткий курс**

Прежде чем мы перейдем к изучению программирования системных операций на Турбо Паскале, кратко рассмотрим операционную систему MS-DOS, познакомимся с терминологией, принципами работы и ее «внутренним устройством». Подробному рассказу об операционной системе можно посвятить целую книгу, но нам придется уделить этой теме всего несколько страниц нашего курса.

Несмотря на то, что MS-DOS современному пользователю и современному программисту представляется устаревшей операционной системой, сохраняющийся интерес к ней можно объяснить рядом причин. За те годы, когда MS-DOS была одной из доминирующих операционных систем для персональных компьютеров, для нее было создано довольно большое количество всевозможных программ, качественных, полезных и, по меркам сегодняшнего дня, нетребовательных к возможностям компьютера. MS-DOS содержится в более поздних продуктах фирмы Microsoft, которые либо являются надстройками над MS-DOS (Windows 3.xx и др.), либо дают пользователю возможность выполнения DOS-приложений. MS-DOS, наконец, является относительно простой (подчеркну — относительно!) операционной системой, и поэтому знакомство с приемами и возможностями системного программирования полезно начинать именно с нее. Для того чтобы в полной мере овладеть всеми возможностями своего компьютера, необходимо понимать принципы работы его операционной системы.

Для чего нужны операционные системы? Не прибегая к строгим формулировкам, можно сказать, что назначение операционной системы заключается в обеспечении удобства управления компьютером. Операционная система — это наиболее важная программа (точнее говоря, набор программ) любого компьютера. Значительная часть работы операционной системы заключается в том, чтобы скрыть от пользователя детали выполнения некоторых операций. При обычном копировании файла, например, системе приходится выполнять до двух десятков действий «низкого уровня», в числе которых проверка наличия копируемого файла, а также файла с таким же именем на целевом устройстве, проверка наличия свободного места на диске и т. д. Если бы пользователю приходилось каждый раз самостоятельно выполнять все необходимые действия низкого уровня, жизнь пользователя превратилась бы в сущий ад, а профессия программиста была бы сродни профессии шахтера или лесоруба.

К числу основных функций операционной системы относится обслуживание файловой системы, устройств ввода/вывода, справочный сервис, загрузка программ и распределение между ними ресурсов (то есть памяти, процессорного времени и т. д.).

Один из принципов организации операционных систем — это их модульность. Разработчики операционной системы, придерживаясь принципа модульности, разделяют ее функции на группы. Эти группы организуются в тщательно спланированную иерархическую систему, в которой каждый уровень иерархии выполняет присущие ему функции.

MS-DOS тоже является модульной операционной системой, которая состоит из нескольких основных модулей и составных частей:

- *Система BIOS* (Basic Input Output System — базовая система ввода/вывода) в ПЗУ (постоянном запоминающем устройстве). Эта система поставляется вместе с персональным компьютером и может использоваться любой операционной системой. Задача BIOS заключается в обслуживании основных, базовых операций. BIOS является составной частью не только MS-DOS, но и любой операционной системы персонального компьютера.
- *Загрузчик системы*. Загрузчик системы представляет собой короткую программу, находящуюся в первом секторе загрузочного диска. Функция этой программы заключается в запуске процесса загрузки операционной системы после включения компьютера или его перезапуска (происходящего при нажатии клавиш **CTRL+Alt+Del**). Загрузчик считывает следующие модули операционной системы в память.
- *Модуль расширения BIOS*, содержащийся в дисковом файле IO.SYS. Его задача заключается в определении состояния оборудования и установке включенных периферийных устройств в исходное состояние, конфигурировании системы по указаниям в файле CONFIG.SYS, запуске базового модуля операционной системы. BIOS, модуль расширения IO.SYS и драйверы устройств образуют «физический» уровень операционной системы.
- *Базовый модуль MS-DOS*, содержащийся в файле MSDOS.SYS. Функция базового модуля состоит в инициализации внутренних таблиц, векторов прерыва-

ний, считывании в память и запуске интерпретатора команд. Данный модуль реализует основные услуги MS-DOS и образует «логический» уровень операционной системы.

- *Интерпретатор команд*, содержащийся в файле COMMAND.COM. Главная функция интерпретатора заключается в обработке команд пользователя. Команды MS-DOS, считающиеся внутренними, такие, например, как COPY или DIR, реализуются процедурами COMMAND.COM. Интерпретатор команд при загрузке также выполняет файл AUTOEXEC.BAT.
- *Утилиты MS-DOS*. Утилиты содержатся в отдельных файлах и загружаются в память только в случае необходимости. Они не являются неотъемлемой частью MS-DOS, но поставляются вместе с ней. Примером такой утилиты является программа форматирования дисков FORMAT.

Приведу термины, которые имеют прямое отношение к теме данного урока и которые будут использоваться в дальнейшем (да и ранее уже использовались).

- *Логический диск* — это диск, формируемый каким-либо драйвером. Логические диски обычно именуются первыми буквами латинского алфавита с последующим двоеточием А:, В:, С:, Д:... На одном физическом диске может быть несколько логических.
- *Составное имя файла* — это совокупность имени файла и его расширения (последовательности из 1-3 символов, следующих после точки). Имя файла содержит не более восьми символов. Расширение является необязательной частью имени файла и часто характеризует тип файла, то есть показывает, является ли он текстовым (.txt), например, или выполняемым (.exe). В именах файлов нельзя использовать символы «==», «+», «[», «]», «|», «\», «/», «;», «:», «>», «<» и «>>», а символы «\*» и «?» играют особую роль (см. ниже). Использование в именах файлов русских букв допускается только в русифицированной версии MS-DOS.
- *Исполняемый файл* в MS-DOS имеет расширение .EXE, .COM или .BAT. Последний из перечисленных типов выполняемых файлов является *командным файлом*, то есть содержит последовательность команд операционной системы MS-DOS.
- *Шаблон* имени файла — это обозначение группы файлов, имена которых могут содержать один или несколько произвольных символов. Символ «?» означает не более одного (произвольного) символа, а звездочка «\*» — любое количество любых символов. Так, например, запись вида \*.PAS обозначает все файлы текущего каталога, имеющие расширение .PAS, то есть являющиеся файлами с исходными текстами программ на языке Паскаль. Шаблон вида A?.\* обозначает все файлы, имя которых начинается с символа А и содержит не более двух символов, а расширение — произвольное (оно может и отсутствовать).
- *Полное имя файла* состоит из следующих трех частей:
  - имя логического диска, на котором находится данный файл;
  - маршрут — ведущая к файлу последовательность имен каталогов, разделенных символом «\» («обратный слэш»);
  - составное имя файла.

Если в имени файла не указан логический диск, используется текущий. Если не указан маршрут, считается, что файл находится в текущем каталоге.

- Абсолютный маршрут* начинается с корневого каталога, имеющего имя \.
- Относительный маршрут* начинается с любого другого каталога кроме корневого.

## Память

Первые версии MS-DOS были разработаны для процессоров 8086/8088, позволяющих адресовать память с суммарным объемом 1 Мбайт. Размещение этой памяти показано на рис. 4.1.



Рис. 4.1. Логическая структура первого мегабайта оперативной памяти

Первые 640 Кбайта относятся к так называемой *стандартной памяти*. Следующие 384 Кбайт занимает *область верхней памяти*, недоступная для прикладных программ MS-DOS и зарезервированная для использования BIOS. Далее следует *область расширенной памяти*, не представленная на рисунке. Для ее использования требуются специальные драйверы, и здесь мы не будем заниматься вопросами, связанными с использованием расширенной памяти.

Итак, мы теперь знаем, что MS-DOS использует для своих собственных нужд и пользовательских программ 640 Кбайт памяти. Эта память распределяется так, как показано на рис. 4.2.

Детали распределения памяти зависят от версии MS-DOS, конфигурации системы и опций, указываемых пользователями в файлах конфигурации системы

**CONFIG.SYS** и **AUTOEXEC.BAT**. Следует заметить, что интерпретатор команд **COMMAND.COM** загружается в две отдельные области памяти. Область, размещенная выше драйверов устройств, сохраняется в памяти постоянно и называется *резидентной частью*. Эта часть отвечает за обработку ошибок дисковых операций ввода/вывода, завершение программ пользователя и реакцию на нажатие комбинации клавиш **Ctrl+Break**. Другая часть **COMMAND.COM** обеспечивает взаимодействие пользователя с **MS-DOS**. Эта часть называется *нерезидентной*, потому что она присутствует только тогда, когда не выполняются программы пользователя или когда программа пользователя пытается загрузить другую программу. Нерезидентная часть обрабатывает *внутренние* команды **MS-DOS** и содержит загрузчик программ. Различие между внутренними и *внешними* командами заключается в том, что последние реализованы в виде отдельных программ, в то время как реализация внутренних команд «зашита» в интерпретатор.

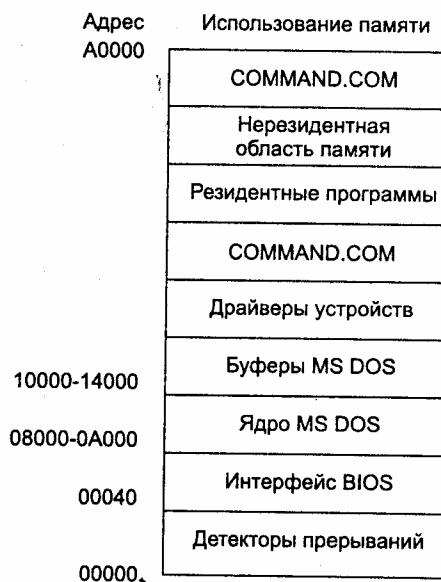


Рис. 4.2. Распределение стандартной памяти

Раздел, помеченный как «Резидентные программы», содержит *резидентные* программы, завершаемые и оставляемые в памяти (**TSR** – *Terminate and Stay Resident*).

Раздел «драйверы устройств» относится к устанавливаемым драйверам устройств, то есть к тем драйверам, которые указываются с помощью команды **DEVICE** = в файле **CONFIG.SYS**. С термином «драйвер» мы еще не знакомы. В общем случае драйвером называют специальную программу, которая управляет определенным устройством. Это может быть видеoadаптер, манипулятор «мышь», звуковая карта и т. д. Драйверы устройств, назначаемые и применяемые в системе по умолчанию, содержатся в разделе «интерфейс BIOS», где они используются во время загрузки или инициализации системы **MS-DOS**.

«Ядро MS-DOS» – это раздел MS-DOS, который обрабатывает различные функции операционной системы. Он является «посредником» между программами пользователя или **COMMAND.COM** и различными драйверами устройств, а также аппаратными средствами.

Раздел «векторы прерываний» содержит описание 256 векторов прерываний системы. Прерывания будут рассматриваться позже в этом уроке. «Нерезидентная область программ» – это область, куда загружаются программы пользователя.

## Среда

При загрузке программы в память MS-DOS включает в начало программы блок операционной среды. Блок среды программы содержит собственную копию операционной среды MS-DOS. *Среда* (или *окружение*) MS-DOS представляет собой область, в которой запоминаются путь поиска файлов – переменная среды **PATH**, путь к файлу **COMMAND.COM** – переменная среды **COMSPEC**, приглашение – переменная среды **PROMPT**, а также любые переменные, назначаемые по команде **SET**. Переменная среды задается с помощью инструкции **SET NAME = строковое значение** в коде ASCII. Формат блока среды представлен на рис. 4.3.

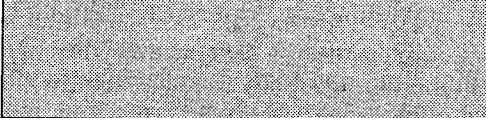
COMSPEC=C:\COMMAND.COM0	0
INCLUDE=D:\DJGPP\CPP\INCLUDE	0
LIB=D:\f32\LIB	0
ECHO=OFF	0
PROMPT=\$p4g	0
PATH=C:\DOS;C:\WINDOWS;C:\ARCH;	0
0	
	

Рис. 4.3. Блок среды MS-DOS

Каждый элемент в блоке среды завершается нулевым байтом. Весь список элементов заканчивается еще одним нулевым байтом. Этот байт является признаком (маркером) конца списка. Элементы, предшествующие этому маркеру, отображаются всякий раз при использовании команды **SET** операционной системы MS-DOS без параметра.

## Прерывания

Как операционная система управляет программами, периферийными устройствами, словом, всем тем, чем она должна управлять? Важнейшую роль здесь играют *прерывания*. Прерывание представляет собой сигнал процессору,рабатываемый программой, каким-либо устройством или самим процессором. Этот сигнал приостанавливает выполнение программы и запускает соответствующую функцию операционной системы для выполнения необходимых системных действий. Необходимость прерываний вызвана двумя причинами — специальными запросами на выполнение системных действий, например по вводу и выводу данных, а также возникновением ошибок при выполнении программ. Прерываниям присваиваются номера, например \$10. Прерывания до \$1F включительно зарезервированы для BIOS, а начиная с \$20 и до \$62 — для операций MS-DOS. Прерывания хранятся в специальной таблице прерываний. В табл. 4.1 приведен перечень некоторых прерываний BIOS и MS-DOS.

Таблица 4.1. Перечень некоторых прерываний BIOS и MS-DOS

Прерывание	Владелец	Назначение
\$00	BIOS	Прерывание из-за ошибки деления на нуль
\$04	BIOS	Прерывание из-за переполнения при умножении
\$05	BIOS	Функция печати копии экрана
\$06	BIOS	Исключительная ситуация «неопределенный код операции»
\$07	BIOS	Исключительная ситуация «код операции ESC»
\$10	BIOS	Обслуживание видеокарты
\$11	BIOS	Список установленного оборудования
\$12	BIOS	Размер памяти
\$13	BIOS	Обслуживание дискового ввода/вывода
\$14	BIOS	Обслуживание последовательного ввода/вывода
\$16	BIOS	Обслуживание ввода/вывода с клавиатуры
\$17	BIOS	Обслуживание принтера
\$19	BIOS	Программа начальной загрузки системы
\$1A	BIOS	Обслуживание системного таймера и часов
\$1B	BIOS	Клавиши Control+Break клавиатуры
\$1C	BIOS	Часы таймера пользователя
\$21	MS-DOS	Вызов функции MS-DOS
\$22	MS-DOS	Адрес завершения программы
\$23	MS-DOS	Адрес выхода Control+C
\$24	MS-DOS	Адрес аварийного завершения из-за фатальной ошибки
\$25	MS-DOS	Функция чтения по абсолютному адресу на диске
\$26	MS-DOS	Функция записи по абсолютному адресу на диске
\$27	MS-DOS	Функция «завершить и оставить резидентной»

В приложении Б дан более подробный перечень прерываний.

# Модуль DOS

Модуль DOS содержит большое число процедур, обеспечивающих возможность выполнения некоторых системных операций. Среди них, например, прямой доступ к различным устройствам и, в частности, к дискам. Некоторые из этих процедур мы уже использовали ранее. Сейчас рассмотрим процедуры модуля более подробно.



## ВНИМАНИЕ!

Многие программы, рассматриваемые в этом уроке, потенциально опасны, так как они напрямую обращаются к операционной системе и выполняют системные операции. Неаккуратное обращение с этими функциями может привести к потере части информации на дисках вашего компьютера и другим неприятным последствиям.

## Работа с файловой системой

Вначале обратимся к процедурам и функциям для работы с файловой системой. В модуле DOS имеется встроенный тип PathStr, который определяется следующим образом:

```
type PathStr: string[79];
```

Кроме этого есть и другие встроенные типы, которые приведены в табл. 4.2.

**Таблица 4.2.** Встроенные типы модуля DOS для работы с файловой системой

Имя и описание типа	Назначение
ComStr = string[127]	Командная строка
DirStr = string[67]	Логический диск и каталог
NameStr = string[8]	Имя файла
ExtStr = string[4]	Расширение, включая точку

Напомню, что функция Fexpand расширяет имя файла до полного имени с указанием маршрута. Ее описание в модуле DOS имеет следующий вид:

```
function Fexpand(path: PathStr): PathStr;
```

Результатом обращения к этой функции вида

```
Fexpand('my_file.pas')
```

будет, например, строка 'D:\USERS\Petrov\my\_file.pas'.

Процедура Fsplit уже встречалась нам в третьем уроке. Напомню, что она разбивает значение типа PathStr на компоненты:

```
procedure FSplit(path: PathStr; var dir: DirStr;
  var name: NameStr; var ext: ExtStr);
```

Первым параметром этой процедуры является имя файла, вторым — имя каталога, в котором находится этот файл, третьим — собственное имя файла и, наконец, последним, четвертым — расширение имени файла, включая точку.

В модуле DOS есть две процедуры поиска файла в каталоге. Результатом каждой является переменная типа `SearchRec` с полной информацией о каталоге, содержащем данный файл:

```
type
  SearchRec = record
    fill: array[1..21] of Byte;
    attr: Byte;
    time, size: LongInt;
    name: string[12];
  end;
```

Здесь поле `fill` зарезервировано для операционной системы, и его лучше не трогать. Поле `time` содержит время и дату создания файла, упакованные в четырехбайтовое число типа `LongInt`. При необходимости доступ к этим полям обеспечивает процедура `UnpackTime`, которая распаковывает значение в запись `DateTIme`. Имеется, кроме того, процедура для установки времени последней модификации файла `SetFTime`. Поле `size` содержит размер файла, а `name` — его имя. Размер последнего поля складывается из 8 символов имени файла, точки и 3 символов расширения имени файла.

Поле `attr` содержит информацию о типе файла, то есть о том, является ли он обычным файлом, подкаталогом, скрытым или архивным файлом и т. д. Поскольку файл может иметь несколько атрибутов, то значения могут быть установлены для нескольких битов этого поля. Назначение каждого бита указано в табл. 4.3. В модуле DOS определены константы со значениями, соответствующими этому набору битов. Эти константы также приведены в таблице.

Таблица 4.3. Атрибуты файла

Бит	Атрибут	Константа
0	Файл открыт только для чтения	ReadOnly = \$01
1	Скрытый файл	Hidden = \$02
2	Системный файл	SysFile = \$04
3	Метка тома	VolumeID = \$08
4	Каталог	Directory = \$10
5	Архив	Archive = \$20
0–5	Произвольный набор атрибутов	AnyFile = \$3F

Последняя константа `AnyFile` представляет собой сумму всех остальных, то есть в ней установлены в единичное значение все биты от нулевого до пятого. Она может оказаться полезной при обработке всех элементов каталога. Файл с атрибутом `VolumeID` (идентификатор тома) может существовать только в корневом каталоге диска.

Поиск в каталоге начинается с вызова процедуры **FindFirst**. Ее описание в модуле DOS имеет вид

```
procedure FindFirst(path: string; attr: Word;
  var srec: SearchRec);
```

После того как первый файл в каталоге найден, поиск может быть продолжен посредством вызовов функции **FindNext**:

```
function FindNext(var srec: SearchRec);
```

В случае ошибки некоторые процедуры модуля DOS возвращают отличное от нуля значение переменной **DOSError**, которая определена в модуле DOS. Замечу, что функция **IOResult** входит в состав модуля **System**, а переменная **DOSError** — в состав модуля DOS.

Процедуры работы с файлами имеются и в модуле **System** — это **Rename**, **Erase**, **MkDir**, **RmDir** и **ChDir**. Модуль **System** содержит и процедуру **GetDir**, которая определяет текущий каталог для указанного диска.

Перейдем к примеру. Программа **delempy** (листинг 4.1) предназначена для удаления всех пустых файлов из указанного каталога. Имя каталога является параметром, с которым вызывается данная программа, например:

```
delempy c:\users\emptyman
```

#### Листинг 4.1. Программа удаления пустых файлов из заданного каталога

```
{$I-}
program delempy;

uses dos, crt;

var
  old_dir, new_dir: dirstr;

procedure get_name;
begin
  new_dir := paramstr(1);
  normalize;
  if new_dir = '' then
    begin
      clrscr;
      writeln('Программа DelEmpty.exe используется так:');
      writeln;
      writeln('DelEmpty Path <Enter>');
      writeln;
      write('Path - полное имя очищаемого каталога');
      readln(new_dir);
      normalize;
      writeln;
    end;
  writeln;
end;
```

```
procedure process_directory(dir: DirStr);
var
  serch_rec: SearchRec;
  io_result: word;
  path: pathstr;
  gg: file;
  attr: word;
begin
  dir := FExpand(dir);
  ChDir(dir);
  if IOResult <> 0 then
    begin
      writeln('Каталог ', dir, ' не найден');
      write('Нажмите <Enter>');
      readln;
      exit;
    end;
  FindFirst('*.*', AnyFile, serch_rec);
  while doserror = 0 do
    begin
      if serch_rec.name = '.' then
        begin
          findnext(serch_rec);
          findnext(serch_rec);
        end
      else
        begin
          if (serch_rec.attr <> Directory) and (serch_rec.size=0) then
            begin
              path := fexpand(serch_rec.name);
              assign(gg, path);
              erase(gg);
              write('Файл ', path, ' удален, нажмите <Enter>');
              readln;
            end;
          attr := serch_rec.attr;
          if attr and Directory = Directory then
            process_directory(serch_rec.name);
          findnext(serch_rec);
        end;
      end;
      if dir <> new_dir then
        chdir('..');
    end;
  begin
    ClrScr;
    writeln('Программа удаления пустых файлов .delempy.exe');
    writeln('=====');
    GetDir(0, old_dir);
    old_dir := FExpand(old_dir);
    get_name;
```

```

writeln;
process_directory(new_dir);
ChDir(old_dir);
write('Работа закончена, нажмите <Enter>');
readln;
end.

```

Разберем эту программу. Процедура `GetDir` определяет имя текущего каталога на заданном диске. Ее первый параметр имеет байтовый тип и принимает значения, определяющие логический диск (табл. 4.4).

**Таблица 4.4. Соответствие значений первого параметра процедуры `GetDir` логическим дискам**

Значение параметра	Диск
0	Текущий диск
1	A:
2	B:
3	C:

Второй параметр имеет строковый тип и содержит имя текущего каталога. Таким образом, с помощью этой процедуры определяется имя текущего каталога, затем оно расширяется до полного имени (функция `Fexpand`) и в игру вступает процедура пользователя `get_name`. Эта процедура считывает имя каталога, очищаемого от пустых файлов. Если программа запущена без параметра, на экран будет выведена краткая информация о формате ее вызова. Затем вызывается процедура `process_directory`, которая и выполняет основную работу — работу чистильщика. В процессе работы ей приходится переходить из одного каталога в другой, но по завершении всей программы она обязательно должна вернуться в тот каталог, из которого была запущена на выполнение. Для этого и используется имя, определенное процедурой `GetDir`. Это имя передается в процедуру `ChDir`, выполняющую переход в каталог с указанным именем.

Процедура `process_directory` вызывается с параметром `new_dir`, который она дополняет до полного имени каталога с маршрутом, а затем переходит в этот каталог.

В процедуре `process_directory` предусмотрены диагностика и вывод сообщений об ошибках.

Далее, чтобы определить первый элемент каталога, вызывается процедура `FindFirst`. Как известно, MS-DOS при выводе содержимого каталога, если это не корневой каталог данного диска, в качестве первых двух элементов выводит «..» и «..», и это следует учесть в программе, пропустив первые две записи. После перехода к первому файлу исследуемого каталога этот файл открывается, а затем определяется его размер. Нулевой размер файла является основанием для удаления файла из каталога с помощью процедуры `Erase`. Для обработки подкаталогов, если такие имеются, используется рекурсивное обращение к процедуре `process_directory` из самой себя.

`searchRec` является встроенным типом модуля DOS, содержащим поля, в которых хранится информация о найденном файле.

Программа `test_exec` (листинг 4.2) демонстрирует возможности, предоставляемые модулем DOS по выяснению конфигурации операционной системы и запуску других программ MS-DOS. Конфигурация операционной системы определяется *переменными окружения*. Одной из таких переменных окружения является переменная `PATH`, которая содержит список путей для поиска файлов. Другая переменная окружения — `COMSPEC` — указывает на файл `COMMAND.COM` — *интерпретатор команд* операционной системы. Напомню, что при работе в MS-DOS пользователь, подавая команды операционной системе, общается именно с интерпретатором команд. Функция `GetEnv` возвращает значение переменной окружения, указанной в качестве ее параметра. Если переменной с указанным именем не существует, результат выполнения функции — пустая строка.

#### Листинг 4.2. Программа `test_exec`

```
{$M $4000.0.0}
program test_exec;

uses crt, dos;

var
  ss, tt: string;
  k: word;

begin
  clrscr;
  ss := GetEnv('PATH');
  writeln('Переменная окружения DOS PATH:');
  k := length(ss);
  writeln(ss);
  writeln;
  writeln('Командный интерпретатор: ', GetEnv('COMSPEC'));
  readln;
  writeln('Выполнение команды из среды DOS');
  SwapVectors;
  Exec(GetEnv('COMSPEC'), '/C dir | more');
  SwapVectors;
  writeln('Нажмите <Enter>');
  readln;
end.
```

Перед заголовком программы находится директива компилятора `{$M...}`. Данная директива позволяет задать параметры распределения памяти для программы на этапе ее выполнения. Кратко поясню назначение параметров. Первый из них задает размер стека. Он должен быть целым числом в диапазоне от 1024 до 65 520. Стек представляет собой область оперативной памяти, выделяемую программе и используемую для хранения локальных переменных в процедурах. Второй и третий параметры задают размер динамически распределяемой област-

ти памяти (минимальное и максимальное значения). Они должны быть числами в диапазоне от 0 до 655 360. Без этой директивы наша программа может работать неправильно, что связано с необходимостью резервирования памяти для процедуры Exec.

Процедура Exec имеет два строковых параметра:

```
function Exec(path: PathStr; parameters: string);
```

Первый содержит имя запускаемой программы, в нашем примере это интерпретатор COMMAND.COM. Второй параметр содержит команду. У нас это команды вывода содержимого текущего каталога и постраничного вывода списка файлов на экран. Параметр /C означает выполнение команды с последующим завершением работы командного интерпретатора.

При вызове процедуры Exec следует иметь в виду, что процедура SwapVectors должна вызываться непосредственно перед и сразу же после обращения (или последовательности обращений) к Exec. Таким образом сохраняется состояние обработчиков прерываний, которое было до вызова процедуры Exec.

В табл. 4.5 приводятся коды ошибок DOSError.

Таблица 4.5. Коды ошибок DOSError

Код ошибки	Ошибка
1	Несуществующая функция DOS
2	Несуществующий файл
3	Несуществующий путь или подкаталог
4	Слишком много открытых файлов
5	Ошибка доступа к файлу или каталогу
6	Неправильный обработчик файла, возможно, файл поврежден
8	Недостаточно памяти
12	Неправильное значение FileMode
15	Неправильно указан диск
16	Невозможно удалить текущий каталог
17	Нельзя перенести файл на другой диск
18	Больше файлов не найдено

## Программирование для MS-DOS и BIOS

В операционной системе MS-DOS имеются специальные адреса памяти, которые используются системой и с помощью которых можно выполнять, например, опе-

рации по обслуживанию периферийных устройств. Эти адреса, расположенные в сегменте, для доступа к которому можно использовать константу Seg0040, приведены в табл. 4.6.

Таблица 4.6. Некоторые специальные адреса MS-DOS

Смещение	Размер	Назначение
\$0010	word	Список оборудования
\$0017	word	Статус клавиатуры
\$0019	word	Альтернативный ввод с клавиатуры
\$001A	word	Вершина буфера клавиатуры
\$001C	word	Нижняя часть буфера клавиатуры
\$001E	16xword	Буфер клавиатуры — кольцевой список
\$003F	byte	Статус мотора флоппи-дисковода (биты 0..3)
\$0040	byte	Обратный отсчет до момента остановки мотора дисковода. Начальное значение равно 37 (~2 с)
\$0041	byte	Статус флоппи-дисковода (ошибки)
\$0042	7xbyte	Статус контроллера флоппи-дисковода
\$0049	byte	Видеорежим
\$004A	word	Ширина текстового экрана
\$004C	word	Количество байт в экранной странице
\$004E	word	Смещение экранной страницы
\$0050	8xword	Положение курсора для 8 текстовых страниц (col, row)
\$0060	word	Форма текстового курсора (bottom, top)
\$0062	byte	Номер видеостраницы
\$0063	word	Адрес порта для видеомикросхемы 6845 (обычно \$03D4)
\$0065	byte	Установка режима CRT
\$0066	byte	Маска цветовой палитры
\$0070	byte	Переход системных часов через полночь
\$0071	byte	Индикация нажатия комбинации клавиш Ctrl+Break
\$0072	word	Устанавливается значение \$1234 при выполнении перезагрузки по нажатию клавиш Alt+Ctrl+Del
\$00F0	16xbyte	Область межпрограммных коммуникаций (Inter-Application Communication Area)
\$0100	byte	Статус клавиши PrintScreen
\$0104	byte	Статус диска (0 = A; 1 = B:)

Назначение битов по адресу \$0010 (список оборудования) приведено в табл. 4.7.

Назначение битов по адресу \$0041 (статус флоппи-дисковода) дано в табл. 4.8.

Таблица 4.7. Назначение битов в слове списка оборудования

Биты	Назначение
0	Установлен в 1, если имеется хотя бы один флоппи-дисковод
1	Установлен в 1 при наличии математического сопроцессора
2–3	Количество банков оперативной памяти
4–5	Исходный видеорежим (значение 10 — цветной режим 80 столбцов; значение 11 — монохромный режим 80 столбцов)
6–7	Количество флоппи-дисководов
8	Равен 0, если установлена микросхема DMA
9–11	Количество последовательных портов
12	Установлен в 1 при наличии джойстика
13	Установлен принтер (или модем) на последовательный порт
14–15	Количество установленных на параллельные порты принтеров

Таблица 4.8. Назначение битов в байте статуса флоппи-дисковода

Биты	Назначение
0	Запрос на неправильную команду
1	Не найдена адресная отметка
2	Сектор не найден (дискета повреждена или не отформатирована)
3	Ошибка DMA
4	Ошибка CRC (Cyclic Redundancy Check) в данных
5	Ошибка микросхемы контроллера
6	Ошибка установки считывающей головки на дорожку
7	Тайм-аут при считывании или записи

В качестве примера использования специальных адресов DOS рассмотрим программу, которая обеспечивает доступ к байту состояния клавиатуры. Его абсолютный адрес в памяти в обычной нотации *сегмент:смещение* имеет вид \$0040:\$0017. Программа *numlock\_off* (листинг 4.3) отключает режим NumLock, если он был включен.

Листинг 4.3. Программа выключения режима NumLock

```
program numlock_off;
var
  pp: ^byte;
begin
  pp := ptr(Seg0040, $0017);
  pp^ := (pp^ and $DF);
end.
```

В табл. 4.9 приведено описание байта состояния клавиатуры (адрес \$0040:\$0017). Установка каждого бита в этом байте соответствует наступлению определенного события.

**Таблица 4.9.** Байт состояния клавиатуры

Бит	Событие
0	Нажата правая клавиша Shift
1	Нажата левая клавиша Shift
2	Нажата клавиша Ctrl
3	Нажата клавиша Alt
4	Нажата клавиша Scroll Lock
5	Нажата клавиша Num Lock
6	Нажата клавиша Caps Lock
7	Нажата клавиша Insert

Шестнадцатеричная константа \$DF, в двоичной записи имеющая вид 11011111, играет роль маски — побитовое выполнение логической операции *and* над этой константой и байтом состояния клавиатуры устанавливает значение 5-го бита («режим NumLock включен») в 0, то есть отключает его.

В программе *numlock\_off* используется переменная типа «указатель», а процедура *Ptr* применяется для того, чтобы сделать ее указателем на интересующий нас байт состояния клавиатуры. Эта процедура полезна для получения доступа к произвольному адресу памяти. Ее описание в модуле *System* имеет вид

```
procedure Ptr(segment, offset: Word);
```

Турбо Паскаль имеет и другие средства для работы с памятью «напрямую». Программу *numlock\_off* можно написать, используя предопределенный массив *Mem*. Элементы этого массива имеют тип *Byte*, а индекс элемента задается в виде *segment:offset*. При необходимости можно использовать предопределенные массивы *MemW* и *MemL*, элементы которых имеют, соответственно, тип *Word* и *LongInt*. Обращение к элементу массива *Mem* с некоторым индексом равнозначно обращению к соответствующей ячейке памяти. Ниже приводится исходный текст программы *numlock\_off*, написанной с использованием массива *Mem*:

```
program numlock_off;
var
  pp: Byte;
begin
  pp := Mem[$0040:$0017] and $DF;
  Mem[$0040:$0017] := pp;
end.
```

Для доступа к памяти можно использовать и так называемые *абсолютные* переменные, описываемые с помощью зарезервированного слова *absolute*:

```
var
  pp: type_ID absolute Seg:0fs;
```

Здесь `type_ID` представляет собой тип, а `Seg:0fs` определяет адрес памяти, по которому и будет размещена данная переменная. Третий вариант программы `numlock_off` использует абсолютную переменную и является, пожалуй, наиболее лаконичным:

```
program numlock_off;

var
  pp: Byte absolute $0040:$0017;

begin
  pp := pp and $DF;
end.
```

## Прерывания

Мы уже знаем, что прерывания — это вызовы процедур низкого уровня операционной системы MS-DOS и BIOS. В Паскале есть две процедуры для прямого обращения к функциям DOS и BIOS — `MSDos` и `Intr`. Прерывания зависят от состояния регистров центрального процессора и заносят в эти регистры результат своей работы. *Регистр* — это внутреннее запоминающее устройство процессора для временного хранения обрабатываемой или управляющей информации. В модуле `System` имеется специальный тип `Registers`, который описывается следующим образом:

```
type
  Registers = record
    case Integer of
      0: (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags: Word);
      1: (AL, AH, BL, BH, CL, CH, DL, DH: Byte);
  end;
```

Тип `Registers` представляет собой запись, состоящую только из варианной части. В табл. 4.10 дан список регистров процессора 80x86.

Введем еще одно важное понятие системного программирования — сегмент. *Сегментом* называется область памяти размером до 64 Кбайт, которая начинается по любому адресу, кратному 16. В области памяти, выделяемой программе, имеется три основных сегмента:

- Сегмент кода* содержит машинные команды, которые будут выполняться. Первая выполняемая команда находится в начале кодового сегмента и операционная система для выполнения программы передает управление по адресу данного сегмента.
- Сегмент данных* содержит данные и рабочие области, необходимые программе.
- Сегмент стека* содержит адреса для возврата в операционную систему и для возврата из подпрограмм в главную программу.

Таблица 4.10. Перечень регистров процессора 80x86

<b>Регистры общего назначения</b>	
<b>Регистры данных (16-битные = 8 + 8 бит)</b>	
<b>Регистр</b>	<b>Назначение</b>
AX = AH AL	Сумматор
BX = BH BL	Базовый (индексный) регистр
CX = CH CL	Счетный регистр
DX = DH DL	Регистр данных
<b>Индексные регистры и регистровые указатели (16 бит)</b>	
<b>Регистр</b>	<b>Назначение</b>
SP	Указатель стека
BP	Указатель базы регистра
SI	Регистр-индекс источника
DI	Регистр-индекс назначения
<b>Сегментные регистры (16 бит)</b>	
<b>Регистр</b>	<b>Назначение</b>
CS	Сегмент кода
DS	Сегмент данных
SS	Сегмент стека
ES	Дополнительный сегмент
<b>Регистр командного указателя (16 бит)</b>	
<b>IP</b>	
<b>Флаговый регистр</b>	

Особенность регистров общего назначения заключается в том, что возможна их адресация как в целом (то есть одним словом), так и каждой однобайтовой части (например, AL — младшая половина, а AH — старшая половина). Регистр AX является сумматором и применяется для всех операций ввода/вывода, некоторых арифметических и строковых операций. Регистр BX может использоваться для расширенной адресации и вычислений. Регистр CX необходим для управления числом повторений циклов и операций сдвига. Регистр DX применяется для некоторых операций ввода/вывода и тех арифметических операций над большими числами, которые используют два регистра DX:AX. Регистровые указатели SP и BP обеспечивают системе доступ к данным в сегменте стека, реже они используются для вычислений. Индексные регистры SI (индекс источника) и DI (индекс назначения) могут применяться для расширенной адресации и операций сложения и

вычитания. Регистр командного указателя содержит смещение указателя на команду, которая должна быть выполнена.

*Флагом* называют признак, устанавливаемый программой с целью указания особенностей выполнения стандартных операций, — это одноразрядный индикатор, отражающий выполнение или невыполнение определенного условия. Флаги занимают 16 бит, из которых биты с номерами 1, 3, 5, 13 и 15 не используются. Назначение некоторых других битов приведено в табл. 4.11.

Таблица 4.11. Флаги

Бит	Флаг	Описание
0	CF	<i>Carry Flag</i> — признак переноса или переполнения. Разряд слова состояния процессора (PSW — Processor Status Word), принимающий значение 1, если при выполнении команды произошел перенос из старшего разряда, и значение 0 в противном случае. Признак переноса используется командами условного перехода
2	PF	<i>Parity Flag</i> — флаг четности. Проверяются младшие восемь битов результата операции. При нечетном числе битов флаг устанавливается в 0, а при четном — в 1
4	AF	<i>Auxilliary Carry</i> — вспомогательный флаг переноса. Этот флаг имеет отношение к операциям над символами кода ASCII
6	ZF	<i>Zero Flag</i> — признак нуля. Разряд слова состояния процессора, устанавливаемый в единичное состояние при равенстве нулю результата последней арифметической или логической операции
7	SF	<i>Sign Flag</i> — флаг знака. Устанавливается в 0, если результат арифметической операции положительный, и в 1, если результат отрицательный
8	TF	<i>Trap Flag</i> — флаг трассировки. При установке этого флага в 1 процессор переходит в режим пошагового выполнения команд. При этом в каждый момент выполняется одна команда под управлением пользователя
9	IF	<i>Interrupt Flag</i> — флаг прерывания. При нулевом значении этого флага прерывания запрещены, а при единичном — разрешены
10	DF	<i>Direction Flag</i> — флаг направления. Используется в операциях над строками для определения направления передачи данных
11	OF	<i>Overflow Flag</i> — флаг переполнения. Определяет арифметическое переполнение

Рассмотрим процедуры, предназначенные для работы с регистрами процессора и прерываниями `MSDOS` и `Intr` модуля DOS:

```
procedure MSDos(var regs: Registers);
procedure Intr(number: Byte; var regs: Registers);
```

Процедура `Intr` предназначена для обращения к прерыванию с указанным номером (первый параметр). Перед выполнением прерывания процедура загружает регистры процессора значениями соответствующих полей записи `regs`. После выполнения прерывания содержимое этих регистров вновь записывается в соответствующую переменную. Процедура `MSDOS` эквивалентна вызову процедуры `Intr` для прерывания \$21.

Для прерывания \$21 имеется несколько сотен функций MS-DOS, предназначенных для работы с файлами и дисками. Процедуры Паскаля, использующие все эти низкоуровневые операции, дают гораздо более удобные и безопасные способы их выполнения. Поэтому к работе с прерываниями следует прибегать, если в этом действительно есть необходимость.

В качестве примера работы с прерываниями возьмем пример простой программы, в которой используется функция \$2C для считывания показаний системных часов (программа `gettime`, исходный текст которой приведен в листинге 4.4).

#### Листинг 4.4. Программа считывания показаний системных часов

```
program gettime;
uses dos, crt;
var
  regs: registers;
begin
  ClrScr;
  regs.AH := $2C;
  Intr($21, regs);
  Write('Системные часы показывают ');
  with regs do
  begin
    Write(CH, ' час. ');
    Write(CL, ' мин. ');
    Write(DH, '.');
    if DL < 10 then
      write(0);
    WriteLn(DL, ' сек.');
  end;
  WriteLn('Нажмите <Enter>');
  ReadLn;
end.
```

## Прерывания BIOS

В этом разделе мы рассмотрим использование двух прерываний BIOS, \$16 и \$10. Первое из них мы используем для развлечения. Программа в листинге 4.5 превращает клавиатуру вашего компьютера в светомузыкальную установку. При запуске этой программы на выполнение динамик компьютера играет мелодию известной песни «Happy birthday...», а на клавиатуре вспыхивают и гаснут индикаторы нажатия клавиш NumLock, CapsLock и ScrollLock.

Прежде чем мы перейдем к обсуждению программы, познакомимся подробнее с тем, как работает клавиатура персонального компьютера. Стандартная клавиатура содержит свой собственный микропроцессор (микросхема Intel 8048 или ее эквивалент). Нажатие или отпускание клавиши посылает сигнал контроллеру

прерываний, который вызывает подпрограмму обслуживания прерывания \$09 для обработки этого запроса. Приоритет прерываний от клавиатуры – второй после прерываний от таймера. Постоянное запоминающее устройство на системной плате содержит программу обработки прерывания \$09 по умолчанию. Эта программа читает и декодирует считываемый код, отслеживает нажатие специальных клавиш (Control, Shift, Alt и др.) и преобразует коды сканирования во внутренние коды. Каждое нажатие клавиши вырабатывает два кода сканирования – для нажатия и отпускания клавиши. Выбор кода сканирования зависит от информации о состоянии клавиатуры. Например, нажатие клавиши A генерирует код сканирования \$61 (код ASCII строчной латинской буквы *a*). Если при нажатии клавиши A нажата еще и управляющая клавиша Ctrl, код сканирования трансформируется в \$01 (код ASCII для Ctrl+A). Если при нажатии клавиши A нажата клавиша Shift, код сканирования будет \$41 (код ASCII для заглавной латинской буквы *A*). В результате нажатия таких клавиш как Shift и Alt программа обработки прерывания \$09 обновляет байт состояния клавиатуры внутри сегмента данных BIOS. Сегмент данных BIOS начинается с адреса \$0040 и содержит динамические переменные, используемые различными подпрограммами BIOS. В табл. 4.9 приводилось описание байта состояния клавиатуры. Определенные комбинации клавиш имеют специальное назначение. Такой специальной комбинацией является, например, Ctrl+Alt+Del, которая генерирует прерывание \$19. При вводе символов они заносятся в буфер клавиатуры. Если буфер полон, программа обработки выдает звуковой сигнал и отвергает символ, в противном случае символ добавляется в конец буфера.

С помощью определенных комбинаций клавиш (например, Alt+буква или Alt+цифра) и специальных клавиш (например, функциональных) воспроизводятся символы расширенного кода ASCII. Программное обеспечение доступно аппаратным средствам клавиатуры через BIOS. Прерывание \$16 BIOS позволяет осуществить доступ к клавиатуре.

#### Листинг 4.5. Программа keyboard\_last\_song

```
program keyborad_last_song;

uses dos, crt;

var
  keyb_status_byte : byte absolute $40:$17;
  num, caps, scroll : boolean;
  old_num, old_caps, old_scroll : boolean;
  loop : byte;
  k : Integer;

const
  max = 51;

procedure get_leds_status(var n, c, s : boolean);
begin
  s := true;
  if (keyb_status_byte and $10) = 0 then s := false;
```

```

n := true;
if (keyb_status_byte and $20) = 0 then n := false;
c := true;
if (keyb_status_byte and $40) = 0 then c := false;
end;

procedure set_leds_status(n, c, s : boolean);
var
reg : registers;
begin
if s then
keyb_status_byte := keyb_status_byte or $10
else
keyb_status_byte := keyb_status_byte and not $10;
if n then
keyb_status_byte := keyb_status_byte or $20
else
keyb_status_byte := keyb_status_byte and not $20;
if c then
keyb_status_byte := keyb_status_byte or $40
else
keyb_status_byte := keyb_status_byte and not $40;
reg.ax := $0200;
intr($16, reg);
end;

procedure keyb_song(tone : integer);
const
melody : array[0..max, 1..2] of integer =(
(0, 0), (1046, 200), (1046, 200), (1174, 400), (1046, 400),
(1398, 400), (1318, 800), (1046, 200), (1046, 200), (1174, 400),
(1046, 400), (1568, 400), (1398, 800), (1046, 200), (1046, 200),
(2092, 400), (1760, 400), (1398, 400), (1318, 400), (1174, 800),
(1864, 200), (1864, 200), (1760, 400), (1398, 400), (1568, 400),
(1398, 800), (1046, 200), (1046, 200), (1174, 400), (1046, 400),
(1398, 400), (1318, 800), (1046, 200), (1046, 200), (1174, 400),
(1046, 400), (1568, 400), (1398, 800), (1046, 200), (1046, 200),
(2092, 400), (1760, 400), (1398, 400), (1318, 400), (1174, 800),
(1864, 200), (1864, 200), (1760, 400), (1398, 400), (1568, 400),
(1398, 800), (0, 0));
begin
sound(melody[tone][1]);
delay(melody[tone][2]);
nosound;
end;

begin
get_leds_status(old_num, old_caps, old_scroll);
num:= false;
caps := false;
scroll := false;
loop := 0;

```

```

k := 0;
writeln('Нажмите <enter> для остановки программы');
repeat
  keyb_song(k);
  k := (k + 1) mod max;
  loop := random(4);
  case loop of
    0 : scroll := false;
    1 : num   := true;
    2 : begin
      num  := false;
      caps := true;
    end;
    3 : begin
      caps := false;
      scroll := true;
    end;
  end;
  set_leds_status(num, caps, scroll);
until keypressed;

set_leds_status(old_num, old_caps, old_scroll);
end.

```

Процедура `get_led_status` определяет состояние индикаторов в момент запуска программы на выполнение и сохраняет это состояние в переменных `old_num`, `old_caps` и `old_scroll`. Чтобы выяснить состояние индикаторов, используется один из специальных адресов MS-DOS, содержащий статус клавиатуры (см. табл. 4.5). Переменная `keyb_status_byte` указывает на байт статуса клавиатуры. Изменение значения определенных битов этой переменной позволяет включать или выключать индикаторы клавиатуры. Доступ к битам обеспечивается масками `$10`, `$20` и `$40`, имеющими двоичное представление `00010000`, `00100000` и `01000000` соответственно. Напомню, что установка единичного значения для 4-го бита (отсчет идет справа налево, начиная с нуля) означает включение режима `ScrollLock`, 5-го — включение режима `NumLock` и 6-го — включение режима `CapsLock`. Установить в необходимый бит единичное значение можно с помощью побитной логической операции `or`, operandами которой являются байт состояния клавиатуры и соответствующая маска. Установка нулевого значения бита производится заменой маски результатом ее обращения посредством логической операции `not`, при этом вместо операции `or` следует использовать `and`.

В начале программы, после определения состояния клавиатуры, переменным `num`, `caps` и `scroll` присваиваются значения `false`, соответствующие выключенным индикаторам. Затем в игру вступает цикл `repeat...until`, в котором имеется параметр — целая переменная `k`, значение которой изменяется циклически от 0 до 51. Именно 52 ноты в мелодии, которую проигрывает динамик компьютера при вызове процедуры `keyb_song`. Мелодия хранится в виде двумерного массива, каждая нота описывается высотой и продолжительностью звучания. Выбор комбинации включенных индикаторов производится случайным образом, а их переключение происходит в процедуре `set_leds_status`. Выполнение цикла прекращается при

нажатии любой клавиши, после чего восстанавливается исходное состояние индикаторов клавиатуры и программа завершает свою работу.

В программе **test16** (листинг 4.6) для считывания символов клавиатуры используется прерывание \$16. Функция \$10 выполняет считывание кода нажатой клавиши.

#### Листинг 4.6. Использование прерывания \$16 для считывания символов клавиатуры

```
program test16;
uses Dos;
var
  ch : Char;
  regs : Registers;
  ext : Boolean;

function ExReadKey(var Extended_code : boolean): char;
begin
  regs.ah := $10;
  Intr($16, regs);
  Extended_code := (regs.al = 0) or (regs.al > 127);
  if Extended_code then
    ExReadKey := Chr(regs.ah)
  else
    ExReadKey := Chr(regs.al);
end;

begin
  repeat
    ch := ExReadKey(ext);
    WriteLn(ch, ' ', ext, ' ', ord(ch),
      ' Нажмите <Esc> для выхода из программы');
    until Ch = #27;
end.
```

Функция **ExReadKey** анализирует код нажатой клавиши, и если это расширенный код, параметру **ext** булева типа присваивается значение **true**. Выход из цикла **repeat...until** производится нажатием клавиши **Esc**.

Прерывание BIOS \$10 связано с выполнением экранных функций низкого уровня. Многие из этих функций могут использоваться как в текстовом, так и в графическом режимах. Рассмотрим пример использования прерывания \$10 при работе в текстовом режиме. Листинг 4.7 содержит исходный текст программы **execute\_at\_time** («программа отложенного запуска команд»), которая предназначена для запуска заданной программы или команды в указанное пользователем время. Это стандартная задача, с которой сталкиваются как рядовые пользователи, так и системные администраторы. Рядовой пользователь, например, может «завести» свой компьютер на проигрывание какой-нибудь мелодии в определенное время, превратив его таким образом в будильник. Системный администратор же может установить запуск программы удаления временных файлов. Честно

признаюсь, что наша программа вряд ли ему поможет, так как такая сервисная программа не должна мешать другим пользователям запускать свои программы, а для этого она должна была бы запускаться в «фоновом» режиме. Мы этого делать не умеем, так что с практической точки зрения программе `attime` в лучшем случае остается скромная роль будильника.

**Листинг 4.7.** Программа отложенного запуска команд

```
{$M 4096,0,0}
program execute_at_time;

uses crt, dos;

var
  time_to_run, program_to_run, cur_time : string;
  h_str, m_str, s_str, hnd_str: string[2];
  ch : char;
  hours, minutes, seconds, hunds : Byte;

procedure current_time(var hrs, min, sec, hnd : Byte);
var
  regs : registers;
begin
  with regs do
  begin
    ax := 44 shl 8;
    msdos(regs);
    hrs := ch;
    min := cl;
    sec := dh;
    hun := dl;
  end;
end;

procedure cursor_off;
var
  regs : registers;
begin
  regs.ax :=$0100;
  regs.cx :=$2607;
  intr($10, regs);
end;

procedure cursor_on;
var
  regs : registers;
begin
  regs.ax :=$0100;
  regs.cx :=$0506;
  intr($10, regs);
end;
```

```
procedure input(var time, command : string);
var
  loop : integer;
  p1,p2 : string;
  found_time : boolean;
begin
  if (paramcount <> 2) then
  begin
    writeln;
    writeln(' использование: attime ЧЧ:ММ:СС команда');
    writeln;
    cursor_on;
    halt;
  end
  else
  begin
    found_time := false;
    p1 := paramstr(1);
    p2 := paramstr(2);
    loop := Pos(':', p1);
    if loop <> 0 then
    begin
      found_time := true;
      time := p1;
      command := p2;
    end;
    if not found_time then loop := Pos(':', p2);
    if loop = 0 then
    begin
      found_time := true;
      time := p2;
      command := p1;
    end;
    if not found_time then
    begin
      writeln;
      writeln('использование: attime ЧЧ:ММ:СС команда');
      writeln;
      cursor_on;
      halt;
    end;
  end;
end;

function time_rest(r_time : string) : string;
var
  hours_to_run, minutes_to_run, seconds_to_run : Byte;
  code : integer;
  str1, str2, str3 : string[2];
  tmp1, tmp2, tmp3, time_rest_to_start : LongInt;
const
  seconds_in_a_day : LongInt = 86400;
```

```

begin
  val(copy(r_time, 1, 2), hours_to_run, code);
  val(copy(r_time, 4, 2), minutes_to_run, code);
  val(copy(r_time, 7, 2), seconds_to_run, code);

  tmp3 := seconds_to_run - seconds;
  tmp2 := minutes_to_run - minutes;
  tmp1 := hours_to_run - hours;

  time_rest_to_start := tmp1 * 3600 + tmp2 * 60 + tmp3;

  if time_rest_to_start < 0 then
    time_rest_to_start := time_rest_to_start + seconds_in_a_day;
  tmp1 := time_rest_to_start div 3600;
  tmp2 := time_rest_to_start mod 3600 div 60;
  tmp3 := time_rest_to_start - tmp1 * 3600 - tmp2 * 60;

  str(tmp3, str3);
  if length(str3) = 1 then str3 := '0' + str3;

  str(tmp2, str2);
  if length(str2) = 1 then str2 := '0' + str2;

  str(tmp1, str1);
  if length(str1) = 1 then str1 := '0' + str1;

  time_rest := str1 + ':' + str2 + ':' + str3;
end;

procedure beep;
begin
  sound(500);
  delay(1000);
  nosound;
end;

begin
  checkbreak := false;
  cursor_off;
  input(time_to_run, program_to_run);
  clrscr;
  gotoxy(20, 10); writeln('-----');
  gotoxy(20, 11); writeln('|');
  gotoxy(20, 12); writeln('| Текущее время : |');
  gotoxy(20, 13); writeln('| Время старта : |');
  gotoxy(20, 14); writeln('| До запуска осталось : |');
  gotoxy(20, 15); writeln('| Программа : |');
  gotoxy(20, 16); writeln('|');
  gotoxy(20, 17); writeln('-----');
  gotoxy(20, 18); writeln('| alt-x : выход |');
  gotoxy(20, 19); writeln('-----');
  gotoxy(44, 13); writeln(time_to_run);

```

```

gotoxy(44, 15); writeln(program_to_run);

repeat
  current_time(hours, minutes, seconds, hunds);
  str(hours, h_str);
  if hours = 0 then h_str:='00';
  str(minutes, m_str);
  str(seconds, s_str);
  str(hunds, hnd_str);

  if length(m_str) = 1 then m_str := '0' + m_str;
  if length(s_str) = 1 then s_str := '0' + s_str;
  cur_time := h_str + ':' + m_str + ':' + s_str + ':' + hnd_str;

  if keypressed then
    begin
      ch := readkey;
      if ch = #0 then
        begin
          ch := readkey;
          if ch = 'X' then
            begin
              cursor_on;
              clrscr;
              halt;
            end;
          end;
        end;
      gotoxy(44, 12);
      writeln(cur_time);
      gotoxy(44, 14);
      writeln(time_rest(time_to_run));
      until time_to_run = copy(cur_time, 1, 8);

      clrscr;
      exec(getenv('comspec'), '/c ' + program_to_run);
      cursor_on;
    end.

```

Разберем программу `execute_at_time`. Назначение процедуры `current_time` заключается в определении текущего времени по системным часам. Эта процедура возвращает четыре параметра байтового типа — `hours`, `minutes`, `seconds` и `hunds` — час, минуты, секунды и сотые доли секунды. Для считывания системного времени используется прерывание \$21 с кодом команды `AH=$2C`. Оператор `ax := 44 sh1 8`; обеспечивает выполнение этой команды, занося ее код в старшую половину регистра `AX`. Следующая группа операторов считывает количество часов, минут, секунд и сотых долей секунды из соответствующих регистров.

Процедуры `cursor_off` и `cursor_on` включают и выключают отображение курсора на текстовом экране. Для этого используется прерывание \$10.

Процедура `input` иллюстрирует важный элемент программирования — какого бы высокого мнения ни был разработчик программного обеспечения о потенциальном пользователе своих программ, он должен иметь в виду возможность ошибки при вводе данных или неосведомленность пользователя о формате запуска данной программы. По этой причине программа должна быть настроена к пользователю и дружелюбно, и строго. Строгость заключается в проверке корректности вводимых данных. Вводимые данные не должны приводить к аварийному завершению программы. Ну а дружелюбность — это информирование пользователя об основных правилах обращения с программой. Процедура `input` основана на материале прошлых уроков, и читатель сможет разобраться в ее работе самостоятельно.

Функция `time_rest` определяет время, оставшееся до момента запуска указанной команды, и переводит это значение в строку в соответствии с принятым представлением — `чч:мм:сс:дд`, где `чч` — это часы, `мм` — минуты, `сс` — секунды и `дд` — сотые доли секунды. При этом следует особо позаботиться о наличии в полях часов, минут и секунд нулей в левой позиции, если соответствующее значение находится в диапазоне `0...9`. Процедура `beep` в момент запуска указанной команды на выполнение дает непродолжительный звуковой сигнал с помощью динамика компьютера.

В теле программы содержатся операторы, выполняющие вывод заставки программы с отображением текущего времени (с точностью до сотых долей секунды), назначенного времени запуска программы (с точностью до секунд), времени, оставшегося до намеченного момента и имени запускаемой команды или программы. «Опрос» системного таймера и вывод текущего времени производится в цикле `repeat...until`. Работу программы можно остановить, нажав комбинацию клавиш `Alt+x`. В тело программы добавлена проверка этой комбинации клавиш с завершением работы программы в случае их нажатия. В назначенный момент времени процедура `exec` запускает командный интерпретатор и собственно программу. Во время ожидания курсор отключен и вновь включается уже при завершении работы программы.

Следует иметь в виду, что в отличие от процедур для работы с текстовым экраном, которые использует Паскаль, значения параметров прерывания `$10` отчитываются от нуля, а не от единицы. В Паскале предполагается, что координаты текстового экрана увеличиваются от верхнего левого угла `(1, 1)` к нижнему правому `(80, 25)`. Таким образом, процедура `GotoXY(80, 1)` перемещает курсор в верхний правый угол текстового экрана. Для прерывания `$10` в текстовом режиме верхний левый угол имеет координаты `(0, 0)`, а нижний правый — `(79, 24)`. Еще одна важная деталь заключается в том, что при работе в текстовом режиме поддерживается 8 страниц видеопамяти. На экране дисплея в каждый момент отображается лишь одна из них, но вывод текстовой информации может происходить на любую из восьми страниц.

#### Листинг 4.8. Программа `test_intr_10`

```
program test_intr_10;
uses dos, crt;
```

```
var
    top, bottom, x, y, attr, k: byte;
    ss: string;
    reg : registers;

procedure display_page(page: byte);
begin
    reg.AH := $05;
    reg.AL := page;
    intr($10, reg);
end;

procedure set_cursor_position(page, x, y: byte);
begin
    with reg do
    begin
        AH := $02;
        BH := page;
        DH := y;
        DL := x;
    end;
    intr($10, reg);
end;

procedure write_chars(page, x, y: byte; ch0: char; attr: byte;
multiplicity: word);
begin
    set_cursor_position(page, x, y);
    with reg do
    begin
        AH := $09;
        AL := ord(ch0);
        BH := page;
        BL := attr;
        CX := multiplicity;
    end;
    intr($10, reg);
end;

procedure write_string(page, x, y: byte; ss: string; attr: byte);
var
    j: word;
begin
    for j := 1 to length(ss) do
    begin
        write_chars(page, x, y, ss[j], attr, 1);
        inc(x);
    end;
end;

begin
    clrscr;
```

```

writeln('Демонстрация использования прерывания 10');
writeln('для работы с текстовым экраном');
write('Нажмите <Enter>');
readln;
for k := 0 to 7 do
begin
  write_chars(k, 0, 0, ' ', black, 80);
  attr := 128*(k mod 2) + k shl 4 + k + 8;
  write_chars(k, 0, 1, chr(177), attr, 80*24);
  write_string(k, 0, 0, 'Видеостраница ' + chr(48 + k) + ' ', k + 8);
  display_page(k);
  delay(1000);
end;
write_string(7, 40, 4, 'Нажмите <Enter>: ', Yellow);
readln;
end.

```

Процедура `display_page` отображает на экране видеостраницу с указанным в качестве аргумента номером.

Процедура `set_cursor_position` устанавливает курсор в положение на странице с указанным номером (первый параметр) в заданной позиции на заданной строке (второй и третий параметры).

Следующие две процедуры (`write_chars` и `write_string`) предназначены для вывода символов и строк. У процедуры `write_chars` 6 параметров. Первый из них (`page`) задает номер страницы. Затем идут параметры, определяющие номер позиции в строке и номер строки (`x, y`), в которой будет выведен символ, задаваемый четвертым параметром (`ch0`). Далее идут параметры атрибута символа (`attr`) и повторитель (`multiplicity`). Байт атрибута содержит информацию о цвете символа, цвете фона и признаком мерцания:

- бит 7 — мерцание;
- биты 6–4 — цвет фона. Возможны следующие варианты:

Двоичное значение	Цвет
000	Черный
001	Синий
010	Зеленый
011	Голубой
100	Красный
101	Ярко-красный
110	Коричневый
111	Светло-серый

- биты 3–0 — цвет символа (цвет задается аналогично цвету фона).

Повторитель указывает, сколько раз, начиная с заданной позиции, должен быть выведен символ. В назначении параметров процедуры `write_string` нетрудно разобраться самостоятельно. Процедуры `write_chars` и `write_string` не влияют на положение курсора и трактуют управляющие символы не как инструкции, а как печатаемые символы. Замечательно то, что можно незаметно подготовить целую страницу, а затем вывести ее на экран.

В разделе операторов программы вывод в текстовом режиме выполняется на все 8 видеостраниц, которые заполняются символами псевдографики с изменением атрибутов символов.

Программа `test_text_video` (листинг 4.9) демонстрирует использование «прокрутки» экрана. В этой программе используются некоторые функции прерывания \$10. Номера этих функций заносятся в регистр AH. Кратко поясню назначение этих функций. Функция \$02 используется для установки курсора на определенной странице видеопамяти в заданной позиции (см. текст процедуры `set_cursor_position`). Функция \$06 — это функция прокрутки окна вверх на заданное количество строк (количество строк заносится в регистр AL). Границы окна заносятся в регистры CL, CH (верхний левый угол) и DL, DH (нижний правый угол). Содержимое регистра BH определяет атрибуты вывода пустых строк в нижней части окна при его прокрутке. Функция \$07 используется для прокрутки окна вниз. И наконец, функция \$09 выводит символ с заданными атрибутами в заданной позиции видеостраницы.

#### Листинг 4.9. Программа `test_text_video`

```
program test_text_video;
uses dos, crt;
const
  ss: array[1..12] of string[40] =(
    'HAPPY HALLOWEEN!!!!',
    '          00000000000000000000000000000000',
    '          00000000000000000000000000000000',
    '          00000000  0000000000  00000000',
    '          0000000000  0000000000  0000000000',
    '          000000000000  0000000000  0000000000',
    '          000000000000000000000000000000000000',
    '          000000000000000000000000000000000000',
    '          000000000000000000000000000000000000',
    '          000000000000000000000000000000000000',
    '          000000000000000000000000000000000000',
    '          000000000000000000000000000000000000');
var
  x, y, n, attr, k, col, row: byte;
  reg : registers;
procedure up(x0, y0, x1, y1, n, attr: byte);
begin
```

```
with reg do
begin
    ah := $06;
    al := n;
    bh := attr;
    cl := x0; ch := y0;
    dl := x1; dh := y1;
end;
intr($10, reg);
end;

procedure down(x0, y0, x1, y1, n, attr: byte);
begin
    with reg do
    begin
        ah := $07;
        al := n;
        bh := attr;
        cl := x0; ch := y0;
        dl := x1; dh := y1;
    end;
    intr($10, reg);
end;

procedure set_cursor_position(page, x, y: byte);
begin
    with reg do
    begin
        AH := $02;
        BH := page;
        DH := y;
        DL := x;
    end;
    intr($10, reg);
end;

procedure write_chars(page, x, y, ch0: char; attr: byte; multiplicity: word);
begin
    set_cursor_position(page, x, y);
    with reg do
    begin
        AH := $09;
        AL := ord(ch0);
        BH := page;
        BL := attr;
        CX := multiplicity;
    end;
    intr($10, reg);
end;

procedure write_string(page, x, y: byte; ss: string; attr: byte);
var
    j: word;
```

```
begin
    for j := 1 to length(ss) do
    begin
        write_chars(page, x, y, ss[j], attr, 1);
        inc(x);
    end;
end;

begin
    up(0, 0, 79, 24, 0, black);
    attr := yellow;
    for row := 1 to 12 do
        write_string(0, 20, row, ss[row], attr);
    for row := 1 to 12 do
        write_chars(0, 0, row, ' ', black, col);
    k := 0;
repeat
    inc(k);
    if odd(k) then
        for row := 0 to 9 do
        begin
            delay(100);
            down(0, 0, 79, 24, 1, black);
        end
    else
        for row := 0 to 9 do
        begin
            delay(100);
            up(0, 0, 79, 24, 1, green);
        end;
    sound(200); delay(10); nosound;
    until keypressed;
end.
```

#### Упражнение 4.1

Самостоятельно разберите программу 4.9, обратите особое внимание на процедуры *up* и *down*.

## Мышь

Мышь — это хорошо всем известный манипулятор, устройство для ввода информации, дающее программисту восхитительное чувство единения с компьютером! Программная поддержка манипулятора «мышь» основана на использовании прерывания \$33. Это прерывание MS-DOS, а не BIOS, то есть оно обрабатывается не где-то в глубинах микросхемы BIOS, а в программном обеспечении операционной системы. Сейчас мы займемся программированием мыши при работе с текстовым экраном.

Каждому режиму экрана компьютера соответствует виртуальный экран мыши. Можно считать, что виртуальный экран представляет собой сетку, покрывающую экран, размер которой может совпадать, а может и не совпадать с размерами экрана в пикселях. Для тех режимов экрана, которые мы, возможно, будем использовать, виртуальные экраны такие:

- 16-цветный текстовый режим 80×25 (режим \$02) — виртуальный экран 640×200;
- 16-цветный графический режим 640×350 (режим \$10) — виртуальный экран 640×350;
- 16-цветный графический режим 640×480 (режим \$12) — виртуальный экран 640×480.

Рассмотрим модуль, содержащий довольно представительный набор процедур для работы с мышью. Листинг 4.10 содержит исходный текст интерфейсной секции этого модуля.

**Листинг 4.10.** Интерфейсная секция модуля mouse

```
unit mouse;

interface

type
  Str6 = string [6];

procedure reset_mouse(var mouse_OK: Boolean; var button: Byte);
procedure show_cursor;
procedure hide_cursor;
procedure get_mouse_status(var button: Byte; var x, y: Word);
procedure mouse_gotoXY(x, y: Word);
procedure get_mouse_button_press(var button: Byte; var count, x, y: Word);
function double_click(timeout: Word): Boolean;
procedure get_mouse_button_release(var button: Byte; var count, x, y: Word);
procedure set_cursor_x_lim(min_x, max_x: Word);
procedure set_cursor_y_lim(min_y, max_y: Word);
procedure set_graph_cursor_shape(hot_x, hot_y: Integer; address: Pointer);
procedure set_text_cursor_shape(screen_mask, cursor_mask: Word);
procedure get_relative_move(var x, y: Integer);
procedure set_hide_cursor_window(x0, y0, x1, y1: Word);
procedure get_mouse_state_size(var size: Word);
procedure save_mouse_driver_state(address: Pointer);
procedure restore_mouse_driver_state(address: Pointer);
procedure get_mouse_page(var page: Word);
procedure set_mouse_page(page: Word);
procedure reset_mouse_software(var installed: Boolean; var no_buttons: Word);
procedure get_mouse_info(var ver: Str6; var port, IRQ: Byte);
```

Обзор модуля начнем с тех процедур, которые переприсваивают значения параметров мыши, размещают курсор мыши в центре экрана, отключают и включают его.

**Листинг 4.11.** Процедуры reset\_mouse, show\_cursor и hide\_cursor модуля mouse

implementation

uses DOS, CRT;

var

regs: Registers;

procedure reset\_mouse(var mouse\_OK: Boolean; var button: Byte);  
begin

regs.AX := \$00;

Intr(\$33, regs);

mouse\_OK := Odd(regs.AX);

button := regs.BX;

end;

procedure show\_cursor;

begin

regs.AX := \$01;

intr(\$33, regs);

end;

procedure hide\_cursor;

begin

regs.AX := \$02;

intr(\$33, regs);

end;

Процедура reset\_mouse устанавливает значения параметров «по умолчанию», что означает в нашем случае размещение курсора в центре экрана, прямоугольную его форму в текстовом режиме или форму стрелки в графическом режиме, отображение курсора на нулевой видеостранице, доступность курсору всего экрана и некоторые другие. У этой процедуры два параметра. Первый параметр принимает значение «истина», если мышь установлена, а второй возвращает количество кнопок.

Следующие две процедуры — show\_cursor и hide\_cursor — с помощью вызова соответствующих функций прерывания \$33 позволяют включить или выключить отображение курсора на текущей (нулевой) видеостранице.

**Листинг 4.12.** Процедуры get\_mouse\_status, mouse\_gotoXY и get\_mouse\_button\_press модуля mouse

procedure get\_mouse\_status(var button: Byte; var x, y: Word);  
begin

regs.AX := \$03;

intr(\$33, regs);

with regs do

begin

button := BL;

x := CX;

```

y := DX;
end;
end;

procedure mouse_gotoXY(x, y: Word);
begin
  with regs do
  begin
    AX := $04;
    CX := x;
    DX := y;
  end;
  intr($33, regs);
end;

procedure get_mouse_button_press(var button: Byte; var count, x, y: Word);
begin
  regs.AX := $05;
  regs.BL := button;
  intr($33, regs);
  with regs do
  begin
    button := AL;
    count := BX;
    x := CX;
    y := DX;
  end;
end;

```

Листинг 4.12 содержит тексты трех процедур модуля `mouse`. Процедура `get_mouse_status` определяет состояние кнопок мыши и положение ее курсора. Ниже перечислено назначение битов переменной, описывающей статус кнопок мыши:

- биты 7–3 – не используются;
- бит 2 – нажата центральная кнопка;
- бит 1 – нажата правая кнопка;
- бит 0 – нажата левая кнопка.

Процедура `mouse_gotoXY` перемещает курсор в нужное положение, заданное двумя параметрами этой процедуры, а процедура `get_mouse_button_press` аналогична `get_mouse_status`, за исключением того, что она учитывает историю нажатий кнопок мыши со времени последнего к ней обращения. В параметре `count` возвращается количество нажатий заданной кнопки со времени ее последнего вызова.

#### Листинг 4.13. Функция `double_click` модуля `mouse`

```

function double_click(timeout: Word): Boolean;
var
  k, x, y: Word;
  button: Byte;
begin

```

```

double_click := false;
repeat
    get_mouse_status(button, x, y);
until button = 0;
repeat
    get_mouse_status(button, x, y);
until button = 1;
repeat
    get_mouse_status(button, x, y);
until button = 0;
k := 0;
repeat
    Delay(1);
    Inc(k);
    get_mouse_status(button, x, y);
until (button = 1) or (k = timeout);
if k = timeout then
    Exit;
repeat
    Delay(1);
    Inc(k);
    get_mouse_status(button, x, y);
until (button = 0) or (k = timeout);
double_click := (k < timeout);
end;

```

Прерывание \$33 не содержит функций проверки двойного нажатия кнопок, поэтому такая проверка оформлена в виде специальной процедуры `double_click` (см. листинг 4.13), которая возвращает значение «истина» в том случае, когда в течение `timeout` миллисекунд имело место двойное нажатие левой кнопки мыши («двойной клик» на программистском жаргоне).

#### Листинг 4.14. Процедура `get_mouse_button_release` модуля `mouse`

```

procedure get_mouse_button_release(var button: Byte;
    var count, x, y: Word);
begin
    regs.AX := $06;
    regs.BL := button;
    intr($33, regs);
    with regs do
    begin
        button := AL;
        count := BX;
        x := CX;
        y := DX;
    end;
end;

```

Процедура `get_mouse_button_release` (листинг 4.14) аналогична процедуре `mouse_get_mouse_button_press`, но она сохраняет историю освобождений кнопки.

**Листинг 4.15.** Процедуры set\_cursor\_x\_lim и set\_cursor\_y\_lim модуля mouse

```

procedure set_cursor_x_lim(min_x, max_x: Word);
begin
  with regs do
  begin
    AX := $07;
    CX := min_x;
    DX := max_x;
  end;
  intr($33, regs);
end;

procedure set_cursor_y_lim(min_y, max_y: Word);
begin
  with regs do
  begin
    AX := $08;
    CX := min_y;
    DX := max_y;
  end;
  intr($33, regs);
end;

```

Процедуры `set_cursor_x_lim` и `set_cursor_y_lim` (листинг 4.15) используются для того, чтобы ограничить движение курсора мыши прямоугольной областью на экране, из которой она не может выйти до повторного вызова данной процедуры. Каждая из этих процедур имеет два параметра, определяющих минимальное и максимальное допустимые значения координат курсора мыши.

**Листинг 4.16.** Процедуры установки формы курсора модуля mouse

```

procedure set_graph_cursor_shape(hot_x, hot_y: Integer; address: Pointer);
begin
  with regs do
  begin
    AX := $09;
    BX := Word(hot_x);
    CX := Word(hot_y);
    ES := Seg(address^);
    DX := Ofs(address^);
  end;
  intr($33, regs);
end;

procedure set_text_cursor_shape(screen_mask, cursor_mask: Word);
begin
  with regs do
  begin
    AX := $0A;
    BX := $00;
    CX := screen_mask;
  end;
  intr($33, regs);
end;

```

```

    DX := cursor_mask;
end;
intr($33, regs);
end;

```

Процедуры `set_graph_cursor_shape` и `set_text_cursor_shape` (листинг 4.16) задают форму курсора и положение его активной зоны в графическом и текстовом режимах соответственно, а также поведение курсора при его перемещении по экрану. Что это значит? Один из возможных вариантов поведения курсора мыши, наиболее привычный, но не единственный, заключается в том, что он имеет форму прямоугольника, а при размещении над символом в текстовом режиме этот символ отображается инверсной подсветкой (проглядывает темным силуэтом). Практически этого можно добиться, используя *спецификацию курсора*. Спецификация курсора состоит из двух частей: маски экрана и маски курсора. При движении курсора по произвольной области экрана к этой области и маске экрана применяется логическая операция `and`, а затем к полученному результату и маске курсора применяется логическая операция `xor`. Используя специально подобранные маски, можно сделать курсор непрозрачным, можно сделать и так, что при его размещении над символом инверсной подсветкой будет отображаться совершенно другой символ!

Графический курсор представляет собой графическое изображение размером  $16 \times 16$  пикселов (и, соответственно, бит). Переменная `address` указывает на массив, состоящий из 32 чисел типа `Word`. Это 16 чисел для экранной маски и 16 — для маски курсора. При движении курсора мыши в области `A` на экране отображается результат следующих логических операций:  $(A \text{ and } \text{screen\_mask}) \text{ xor } \text{cursor\_mask}$ . Активная зона (`hot_x, hot_y`) представляет собой точку, выделяемую нажатием кнопки мыши. Верхний левый угол курсора мыши (размером  $16 \times 16$ ) имеет координаты  $(0, 0)$ . Для курсора мыши, имеющего по умолчанию форму стрелки,  $(\text{hot}_x, \text{hot}_y) = (0, -1)$ .

Маска состоит из 16 значений типа `Word`, каждое из которых задает 16 пикселов в строке. По умолчанию маска экрана `screen_mask` = `$77FF`, маска курсора `cursor_mask` = `$7700`.

#### Листинг 4.17. Процедуры `get_relative_move` и `set_hide_cursor_window` модуля `mouse`

```

procedure get_relative_move(var x, y: Integer);
begin
  regs.AX := $0B;
  intr($33, regs);
  x := Integer(regs.CX);
  y := Integer(regs.DX);
end;

procedure set_hide_cursor_window(x0, y0, x1, y1: Word);
begin
  with regs do
  begin
    AX := $10;
  end;
end;

```

```

CX := x0;
DX := y0;
SI := x1;
DI := y1;
end;
intr($33, regs);
end;

```

Процедура `get_relative_move` позволяет определить относительное смещение мыши, а процедура `set_hide_cursor_window` — задать окно, при попадании в которое курсор мыши пропадает. Текст обеих процедур приведен в листинге 4.17.

Следующие три процедуры — `get_mouse_state_size`, `save_mouse_driver_state` и `restore_mouse_driver_state` (листинг 4.18) — позволяют определить состояние мыши, сохранить и восстановить его. Зачем это нужно? Если прерывается выполнение одной процедуры, использующей мышь, и запускается другая, то следует сохранить состояние драйвера мыши и восстановить его при возобновлении работы первой процедуры. Процедура `get_mouse_state_size` использует функцию \$15 прерывания \$33, которая возвращает размер области памяти, необходимой для сохранения состояния драйвера мыши. Процедура `save_mouse_driver_state` сохраняет состояние драйвера, и, наконец, последняя из них — `restore_mouse_driver_state` — это состояние восстанавливает. Перечисленные процедуры используют, соответственно, функции \$16 и \$17 прерывания \$33.

**Листинг 4.18.** Процедуры `get_mouse_state_size`, `save_mouse_driver_state` и `restore_mouse_driver_state` модуля `mouse`

```

procedure get_mouse_state_size(var size: Word);
begin
  regs.AX := $15;
  intr($33, regs);
  size := regs.BX;
end;

procedure save_mouse_driver_state(address: Pointer);
begin
  with regs do
  begin
    AX := $16;
    ES := Seg(address^);
    DX := Ofs(address^);
  end;
  intr($33, regs);
end;

procedure restore_mouse_driver_state(address: Pointer);
begin
  with regs do
  begin
    AX := $17;
    ES := Seg(address^);
  end;
end;

```

```

DX := Ofs(address^);
end;
intr($33, regs);
end;

```

При переключении между видеостраницами «мышиные» страницы не меняются автоматически, и это изменение необходимо запрограммировать. В этом, собственно, и заключается назначение процедур `set_mouse_page` и `get_mouse_page` (листинг 4.19).

**Листинг 4.19.** Процедуры `set_mouse_page` и `get_mouse_page` модуля `mouse`

```

procedure set_mouse_page(page: Word);
begin
  regs.AX := $1D;
  regs.BX := page;
  intr($33, regs);
end;

procedure get_mouse_page(var page: Word);
begin
  regs.AX := $1E;
  intr($33, regs);
  page := regs.BX;
end;

procedure reset_mouse_software(var installed: Boolean; var no_buttons: Word);
begin
  regs.AX := $21;
  intr($33, regs);
  installed := regs.AX = $FFFF;
  no_buttons := regs.BX;
end;

```

Процедура `reset_mouse_software` возвращает два значения. Первое имеет булев тип и принимает значение «истина», если драйвер мыши установлен. Второй — число кнопок мыши. И наконец, последняя процедура модуля — `get_mouse_info` (листинг 4.20) — позволяет определить номер версии драйвера мыши, порт, к которому она подключена, и IRQ — номер запроса прерывания, назначенный мыши.

**Листинг 4.20.** Процедура `get_mouse_info` модуля `mouse`

```

procedure get_mouse_info(var ver: Str6; var port, IRQ: Byte);
var
  ss: Str6;
begin
  regs.AX := $24;
  intr($33, regs);
  with regs do
    begin

```

```

    Str(BH, ver); Str(BL, ss);
    ver := ver + ',' + ss;
    port := CH; IRQ := CL;
  end;
end;

```

## Пример использования модуля mouse

Листинг 4.21 содержит текст программы `test_text_mouse`, предназначенный для демонстрации некоторых процедур для работы с мышью в текстовом режиме из модуля `mouse`. Поработайте с этой программой, модифицируя ее и наблюдая, что произошло в результате такой модификации. Попробуйте, в частности, задавать различные маски.

**Листинг 4.21.** Программа, демонстрирующая использование модуля `mouse`

```

program test_text_mouse;

uses crt, dos, mouse;

var
  ss: string;
  mouse_OK: boolean;
  count, x, y: word;
  button, port, IRQ: byte;
  ver: str6;

procedure display_page(page: byte);
var
  reg : registers;
begin
  reg.AH := $05;
  reg.AL := page;
  intr($10, reg);
end;

begin
  clrscr;
  writeln('Мышь в текстовом режиме');
  reset_mouse(mouse_OK, button);
  if not mouse_OK then
    halt;
  show_cursor;
  get_mouse_info(ver, port, IRQ);
  writeln('Версия драйвера мыши:', ver);
  case port of
    1: Write('bus');
    2: Write('serial');
    3: Write('mouse');
    4: Write('PS/2');
  end;
end.

```

```
5: Write('HP');
end;
writeln('порт IRQ ', IRQ, ', ', button, ' кнопки');
writeln('Нажмите левую кнопку мыши:');
repeat
  button := 0;
  get_mouse_button_press(button, count, x, y);
until button = 1;
clrscr;
writeln('Подвигайте мышь!');
writeln('Нажмите правую кнопку мыши:');
gotoxy(1, 8);
textcolor(yellow);
textbackground(brown);
for x := 1 to 80*16 do
  write(chr(x mod 224 + 31));
textcolor(lightgray);
textbackground(black);
repeat
  button := 0;
  get_mouse_button_press(button, count, x, y);
until button = 2;
show_cursor;
gotoxy(1, 3);
writeln('Нажмите левую кнопку мыши, и курсор пропадет!');
repeat
  button := 0;
  get_mouse_button_press(button, count, x, y);
until button = 1;
hide_cursor;
writeln('Нажмите правую кнопку мыши, и курсор появится!');
repeat
  button := 1;
  get_mouse_button_press(button, count, x, y);
until button = 2;
show_cursor;
set_text_cursor_shape($FFAB, $A200);
writeln('Нажмите обе кнопки мыши');
repeat
  button := 0;
  get_mouse_button_press(button, count, x, y);
until button = 3;
window(1, 1, 80, 6);
clrscr;
window(1, 1, 80, 25);
set_mouse_page(0);
display_page(0);
clrscr;
reset_mouse(mouse_OK, button);
show_cursor;
gotoxy(1, 1);
writeln('Видеостраница 0');
```

```

program mouse;
begin writeln('Для завершения работы нажмите левую кнопку мыши'); clrscr; repeat
  button := 0;
  get_mouse_button_press(button, count, x, y);
  until button = 1;
  reset_mouse(mouse_OK, button);
  show_cursor;
  clrscr;
end.

```

## Другие устройства

В качестве примера программы, использующей функции операционной системы для работы с внешними устройствами, приведем программу cdtest (листинг 4.22), которая позволяет определить, является ли логический диск, имя которого указано в качестве параметра, CD-ROM-дисководом. Речь идет об устройстве считывания с лазерных дисков, обслуживаемом драйвером MSCDEX.

**Листинг 4.22.** Программа cdtest

```

program cdtest;
uses dos;
var
  cd : boolean;
  drive_letter : char;
  in_param : string;

function cdrom (drive : char) : boolean;
var
  regs : registers;
begin
  cdrom := false;
  drive := UpCase(drive);
  if (drive < 'A') or (drive > 'Z') then
    exit;
  FillChar(regs, SizeOf(regs), 0);
  regs.cx := ord(drive) - ord('A');
  regs.ax := $150B;
  intr ($2F, regs);
  cdrom := (regs.ax <> 0) and (regs.bx = $ADAD);
end;

begin
  in_param := paramstr(1);
  drive_letter := in_param[1];
  cd := cdrom(drive_letter);
  writeln(cd);
end.

```

Параметром программы является строковое значение, состоящее из одной буквы — имени диска, эта литера извлекается из первой позиции вводимой строки. Затем в работе приступает функция `cdrom`. Если окажется, что введенное значение не является буквой, результатом будет булево значение `false`. Если же параметр — буква, функция использует прерывание `$2F` для определения того, является ли указанный дисковод CD-ROM-дисководом.

В приложении Б приведен перечень некоторых прерываний DOS и BIOS и их функций. Этот материал может оказаться полезным при самостоятельной разработке программ.

## ЧТО НОВОГО МЫ УЗНАЛИ?

- Познакомились с основами архитектуры операционной системы MS-DOS.
- Познакомились с процедурами модуля DOS, предназначенными для выполнения системных операций.
- Научились программировать обработку событий, связанных с клавиатурой и системным таймером.
- Научились использовать системные функции для работы с экраном.
- Познакомились с программированием манипулятора «мышь».
- Познакомились с программированием для внешних устройств (CD-ROM-дисковод).



# **УРОК**

## **Основы программирования графики**

- 
- 
- Работа в графическом режиме
  - Модуль Graph
  - Программа «Игла Бюффона»
  - Программа «Жизнь»
  - Принципы программирования графики
  - Использование модуля mouse  
для программирования мыши  
в графическом режиме
- 
-

**В** этом уроке мы будем заниматься программированием вывода графических изображений на экран. Вначале познакомимся с основными принципами работы в графическом режиме, а также с теми возможностями, которые предоставляют процедуры и функции графического модуля Турбо Паскаля Graph. Затем мы используем эти возможности в программе моделирования опыта «Игла Бюффона» и программе «Жизнь». Далее перейдем к подробному обсуждению программирования различных графических элементов, таких как точки, отрезки прямых и окружности. В завершение приводится пример использования мыши в графическом режиме.

## Графика

### Текстовый и графический режимы

Прежде чем мы перейдем к основам программирования графики на Турбо Паскале, давайте разберемся, что же такое графический вывод. Известно, что основным устройством для вывода информации, в том числе и результатов работы программы, является монитор компьютера. Монитор внешне очень похож на телевизор, но у него имеется важная особенность. Эта особенность заключается в том, что у телевизора один-единственный (с точки зрения вывода изображения) режим работы, а у компьютерного монитора их два. Это *текстовый и графический режимы*.

Различие между текстовым и графическим режимами работы монитора заключается в возможностях управления выводом визуальной информации. В текстовом режиме минимальным объектом, отображаемым на экране, является *символ*, алфавитно-цифровой или какой-либо иной. В обычных условиях экран монитора, работающего в режиме алфавитно-цифрового дисплея, может содержать не более 80 символов по горизонтали и 25 символов по вертикали, то есть всего 2000 визуальных объектов. При этом имеются ограниченные возможности по управлению цветом символов. Конечно, в таком режиме можно выводить на экран не только обычный текст, но и некие графические изображения, однако понятно, что качество таких изображений будет вне всякой критики. Тем не менее, в «героическую» эпоху компьютерной эры этот метод был единственным и поэто-

му очень популярным способом вывода графиков и целых картин на экран (и на принтер). Программистам иногда удавалось создавать настоящие шедевры «компьютерной псевдографики». Но для серьезной работы с изображениями текстовый режим дисплея абсолютно не подходит.

В графическом режиме минимальным объектом, выводом которого может управлять программист, является так называемый *пиксел* (от английского pixel, возникшего в результате объединения слов «рисунок» (picture) и «элемент» (element)). Пиксел имеет меньшие размеры по сравнению с символом (на один символ в текстовом режиме отводится площадка размером в несколько пикселов). Его геометрические размеры определяются разрешением монитора. Разрешение монитора обычно задается в виде  $rx \times ry$ , где  $rx$  — количество пикселов на экране по горизонтали, а  $ry$  — количество пикселов по вертикали. На практике используются не произвольные, а некоторые определенные значения разрешения. Такими разрешениями являются, например, 320×200, 640×480, 800×600, 1024×768, 1280×1024 и т. д.

Даже в случае самого грубого разрешения изображение в графическом режиме формируется с помощью 64 000 графических элементов, что намного превышает возможности текстового режима.

Можно рассуждать и геометрически. Размер экрана — величина фиксированная. Если величина диагонали экрана 14 дюймов, его геометрические размеры составляют примерно 28×20 см. Размер пикселя можно приблизительно получить, разделив размер экрана на разрешение. Геометрические размеры пикселя определяют степень детализации изображения, его качество. Имеется, правда, минимально допустимое значение размера пикселя, определяемое техническими параметрами монитора.

## Графические координаты

Любое изображение формируется из достаточно простых геометрических фигур. Это точки, отрезки прямых, окружности и т. д. Из геометрии известно, что положение геометрического объекта и его форма задаются координатами его точек. Следовательно, для того чтобы запрограммировать графический вывод, надо научиться задавать координаты графических объектов.

*Графические координаты* задают положение точки на экране дисплея. Поскольку минимальным элементом, к которому имеет доступ программист, является пиксел, естественно в качестве графических координат использовать порядковые номера пикселов. Допустимый диапазон изменения графических координат составляет  $[0, rx - 1]$  для  $x$ -координаты и  $[0, ry - 1]$  для  $y$ -координаты. Точной отсчета является верхний левый угол экрана (рис. 5.1). Значения  $x$ -координаты отсчитываются слева направо, а  $y$ -координаты — сверху вниз. Последнее отличает графические координаты от обычных декартовых координат, принятых в математике, и служит неиссякающим источником ошибок для начинающего программиста.

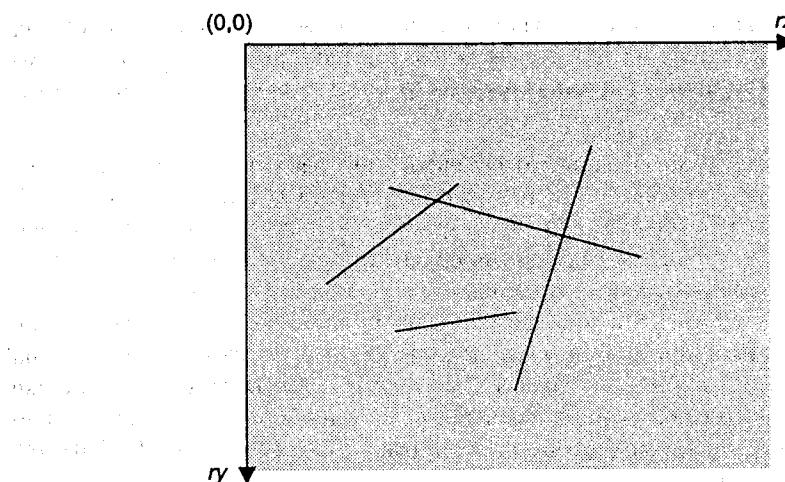


Рис. 5.1. Графические координаты

Проблема заключается в том, что при разработке программы график или другое изображение обычно проектируется в привычной для нас декартовой системе координат. Но для правильного отображения такого графика на экране необходимо учесть различие между декартовой и графической системами координат. Подчеркну, что таких различий три:

1. Графические координаты принимают только целочисленные значения.
2. Графические координаты принимают значения, ограниченные как снизу (нулевым значением), так и сверху (значением разрешения).
3. Графическая координата  $y$  отсчитывается сверху вниз.

Таким образом, геометрические декартовы координаты точки  $(x, y)$  для отображения ее на экране следует пересчитать в графические  $(xg, yg)$  по формулам

$$\begin{aligned} xg &= \lfloor sx \times x \rfloor + dx, \\ yg &= ry - \lfloor sy \times y \rfloor - dy, \end{aligned}$$

где  $\lfloor x \rfloor$  — целая часть  $x$ ;  $sx$  и  $sy$  — масштабные множители, выбираемые из условия

$$\begin{aligned} rx &= \lfloor sx \times x_{\max} \rfloor + 1, \\ ry &= \lfloor sy \times y_{\max} \rfloor + 1. \end{aligned}$$

Здесь  $x_{\max}$  и  $y_{\max}$  — максимальные значения геометрических координат. Пересчет координат  $y$  по такой же формуле, что и для  $x$ , привел бы к зеркально отраженному относительно горизонтальной линии изображению. Слагаемые  $dx$  и  $dy$  обеспечивают смещение изображения относительно левого верхнего угла экрана. Изображение будет смещено в центр экрана при

$$\begin{aligned} dx &= \lfloor rx / 2 \rfloor, \\ dy &= \lfloor ry / 2 \rfloor. \end{aligned}$$

## Переключение между текстовым и графическим режимами

Работа дисплея невозможна без специальных микросхем, управляющих его работой. *Видеoadаптер* должен поддерживать работу дисплея в графическом режиме. Турбо Паскаль обеспечивает работу со следующими видеоадаптерами: CGA, MCGA, EGA, VGA, Hercules, AT&T 400, 3270 PC, IBM-8514.

Работой видеоадаптера управляет специальная программа, которая называется *драйвером*. Драйвер хранится в отдельном файле на диске и содержит как исполняемый код, так и необходимые ему для работы данные. Признак файла с драйвером — расширение *.bgi* имени файла. Имя файла с драйвером соответствует типу видеоадаптера вашего компьютера.

Переключение в графический режим и работа в нем реализованы в Турбо Паскале в виде набора процедур, находящихся в специальном модуле *graph.tpi*. Этот модуль должен явно подключаться к программе с помощью *оператора использования uses*. В модуль *graph* входит примерно 50 процедур. В этом уроке мы познакомимся с некоторыми из них.

При работе в графическом режиме следует иметь в виду, что большинство видеоадаптеров могут работать в нескольких *графических режимах*. Эти режимы различаются прежде всего разрешением и набором доступных цветов.

Итак, программа при переключении в графический режим должна определить тип видеоадаптера. Это можно сделать, явно указав в программе тип видеоадаптера или дав программе возможность самостоятельно определить значение соответствующих параметров. Для этого необходимо ввести переменную целого типа, пусть ее идентификатор будет *gd*. При явном определении видеоадаптера в программе должен присутствовать оператор присваивания

```
gd := value;
```

где *value* — это либо некоторое число, либо встроенная константа (встроенные константы не надо описывать специально, так как их описания содержатся в модулях). Некоторые возможные значения *value* приведены в табл. 5.1.

**Таблица 5.1.** Встроенные константы видеоадаптеров

Константа	Значение
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMono	5
HercMono	7
ATT400	8
VGA	9
PC3270	10

При автоматическом распознавании видеоадаптера в правой части оператора присваивания используется константа `Detect` (или нулевое значение).

Второе, что должна сделать программа, — задать определенный графический режим. Для этого следует ввести еще одну переменную целого типа, назовем ее `gm`, и присвоить ей значение. Некоторые допустимые значения приведены в табл. 5.2.

Таблица 5.2. Графические режимы

Константа	Значение	Описание графического режима
EGALo	0	640×200, 16 цветов, 4 страницы
EGAHi	1	640×350, 16 цветов, 2 страницы
EGA64Lo	0	640×200, 16 цветов, 1 страница
EGA64Hi	1	640×350, 4 цвета, 1 страница
HercMonoHi	0	720×348, 2 страницы
VGALo	0	640×200, 16 цветов, 4 страницы
VGAMed	1	640×350, 16 цветов, 2 страницы
VGAHi	2	640×480, 16 цветов, 1 страница
IBM8514Lo	0	640×480 точек, 256 цветов
IBM8514Hi	1	1024×768 точек, 256 цветов

В столбце «описание графического режима» приведены разрешение, количество цветов и количество страниц. Последний параметр связан с тем, что графическое изображение формируется в специальной видеопамяти. Можно считать, что видеопамять состоит из набора областей — страниц. Изображение формируется на отдельной странице. Страницу организацию видеопамяти можно использовать для программирования динамических изображений.

Переключение в графический режим работы дисплея выполняется вызовом процедуры `InitGraph` из модуля `graph`:

```
InitGraph(gd, gm, 'c:\tp\bgi');
```

Первый параметр в этой процедуре задает тип видеоадаптера, второй определяет режим, а третий представляет собой строку с указанием расположения драйвера на диске. Пустая строка означает, что графический драйвер находится в том же каталоге, что и программа. Для большинства современных видеоадаптеров можно использовать драйвер `egavga.bgi` (или более современный `svga.bgi`). Процедура `InitGraph` переводит систему в графический режим, а затем возвращает управление вызывающей программе.



#### ВНИМАНИЕ

Следует помнить, что в графическом режиме в определенных ситуациях могут некорректно работать операторы вывода `Write` и `WriteLn`. Вместо них надо использовать специальные процедуры для вывода текста.

Завершение работы в графическом режиме производится с помощью процедуры `CloseGraph`, которая выгружает драйвер из памяти и восстанавливает предыдущий видеорежим.

С помощью процедур `RestoreCrtMode` и `SetGraphMode` можно переключаться между текстовым и графическим режимами, не закрывая графический режим.

## Примеры программ

Переключение в графический режим (*инициализация графического режима*) обычно сопровождается обработкой возможных ошибок инициализации. Эти ошибки могут быть связаны с отсутствием графического драйвера или неправильными значениями параметров. При наличии ошибки процедура `InitGraph` вырабатывает соответствующий, отличный от нуля, результат — *код ошибки*. Этот код можно получить с помощью функции `GraphResult`. Однако не очень-то удобно каждый раз записывать в программе целый набор операторов инициализации. Поэтому в дальнейшем нам пригодится графический модуль, использующий «стандартный» модуль `Graph` и скрывающий от нас детали переключения в графический режим и обратно. Исходный текст этого модуля приводится в листинге 5.1.

**Листинг 5.1.** Модуль `graphs`

```
unit graphs;

interface

procedure open_graph;
procedure close_graph;
function gx(x : Extended; sx : Integer) : Integer;
function gy(y : Extended; sy : Integer) : Integer;

implementation

uses Graph;

var
  x, y : Extended;
  sx, sy : Integer;

procedure open_graph;
var
  graph_device, graph_mode : Integer;
begin
  graph_device := Detect;
  InitGraph(graph_device, graph_mode, '');
  if GraphResult <> 0 then
    begin
      Writeln('Ошибка инициализации графического режима');
      ReadLn;
      Halt;
    end;
end;

procedure close_graph;
begin
  CloseGraph;
end;

function gx(x : Extended; sx : Integer) : Integer;
begin
  x := x * 1000;
  if x < -1000 then
    sx := -sx;
  if x > 1000 then
    sx := sx;
  gx := sx;
end;

function gy(y : Extended; sy : Integer) : Integer;
begin
  y := y * 1000;
  if y < -1000 then
    sy := -sy;
  if y > 1000 then
    sy := sy;
  gy := sy;
end;

```

```
    end;
end;

procedure close_graph;
begin
  CloseGraph;
  if GraphResult <> 0 then
  begin
    WriteLn(' Ошибка завершения графического режима ');
    ReadLn;
    Halt;
  end;
end;

function gx(x : Extended; sx : Integer) : Integer;
begin
  gx := trunc(sx * x) + GetMaxX div 2;
end;

function gy(y : Extended; sy : Integer) : Integer;
begin
  gy := GetMaxY div 2 - trunc(sy * y);
end;

end.
```

Этот модуль содержит описания двух процедур, `open_graph` и `close_graph`. Они используются без параметров. Сюда же включены и функции `gx` и `gy` для преобразования математических координат в графические. У каждой из этих функций два параметра, первый — математическая координата, а второй — масштаб. В дальнейшем для инициализации или завершения графического режима достаточно скопировать драйвер в рабочий каталог программы и разместить в программе обращение к соответствующим процедурам. Если в процессе работы процедуры возникла ошибка, на экран будет выведено соответствующее сообщение и программа в целом завершится.

Настало время попробовать возможности графики и познакомиться с некоторыми процедурами модуля `Graph` для установки цветов и рисования линий. Программа `web_in_blue` («Паутина», см. листинг 5.2) выводит на экран простое изображение, составленное из отрезков прямых и концентрических окружностей. Общий центр точки пересечения отрезков и окружностей находится в центре экрана независимо от установленного разрешения. Чтобы изображение не зависело от разрешения, здесь используются функции `GetMaxX` и `GetMaxY`, возвращающие наибольший номер пикселя по горизонтали и по вертикали соответственно. Графические координаты правого нижнего угла экрана равны (`GetMaxX`, `GetMaxY`). Процедура `SetBkColor` устанавливает цвет фона, а процедура `SetColor` — цвет графического объекта. Параметрами этих процедур являются либо цифровые коды цветов, либо встроенные константы, обозначающие цвет. В табл. 5.3 приведены встроенные константы Турбо Паскаля, обозначающие цвета и соответствующие цифровые коды.

**Таблица 5.3.** Встроенные константы Турбо Паскаля, обозначающие цвета, и соответствующие им цифровые коды

Цвет	Константа	Код
Черный	Black	0
Синий	Blue	1
Зеленый	Green	2
Бирюзовый	Cyan	3
Красный	Red	4
Розовый	Magenta	5
Коричневый	Brown	6
Светло-серый	LightGray	7
Темно-серый	DarkGray	8
Светло-синий	LightBlue	9
Светло-зеленый	LightGreen	10
Светло-бирюзовый	LightCyan	11
Светло-красный	LightRed	12
Светло-розовый	LightMagenta	13
Желтый	Yellow	14
Белый	White	15

Процедура `Line(x1, y1; x2, y2)` выводит отрезок прямой, задаваемый начальной точкой  $(x_1, y_1)$  и конечной точкой  $(x_2, y_2)$ . Окружность выводится процедурой `Circle(x, y, r)`, параметрами которой являются графические координаты центра  $x$  и  $y$ , а также радиус  $r$  (в пикселях).

В программе используется процедура `Delay` модуля `CRT`. Эта процедура приостанавливает выполнение программы на указанное количество миллисекунд.

Программа прекращает свою работу при нажатии клавиши `Enter`.

#### Листинг 5.2. Программа «Паутина»

```
program web_in_blue;
uses CRT, Graph, graphs;
var
  i: Word;
begin
  open_graph;
  SetBkColor(Blue);
  SetColor(LightCyan);
  Line(0, 0, GetMaxX, GetMaxY);
  Delay(1000);
```

```

SetColor(Yellow);
Line(0, GetMaxY, GetMaxX, 0);
Delay(1000);
SetColor(LightGreen);
Line(0, GetMaxY div 2, GetMaxX, GetMaxY div 2);
Delay(1000);
SetColor(LightGray);
Line(GetMaxX div 2, 0, GetMaxX div 2, GetMaxY);
Delay(1000);
SetColor(LightRed);
for i:=2 to 20 do
begin
  SetColor(16 - i div 2);
  Circle(GetMaxX div 2, GetMaxY div 2, GetMaxY div i);
  Delay(500 - 15 * i);
end;
ReadLn;
close_graph;
end.

```

Программа *oscillators* (листинг 5.3) позволяет поточечно строить довольно интересные кривые. Эти кривые задаются параметрически, то есть их уравнения имеют вид

$$\begin{aligned}x &= \sin(\omega_x t), \\y &= \cos(\omega_y t),\end{aligned}$$

где  $t$  — вещественный параметр, изменяющийся в определенном диапазоне значений. Задача может быть дана и наглядное физическое истолкование. Речь идет о построении траектории движения точки, которая совершает гармонические колебания с некоторой частотой  $\omega_x$  в горизонтальном направлении и гармонические же колебания с частотой  $\omega_y$  в вертикальном направлении (с некоторым сдвигом по фазе). Задавая различные частоты, можно получить самые разнообразные кривые.

#### Листинг 5.3. Программа *oscillators*

```

program oscillators;
uses graph, graphs, crt;
var
  x, y, t, h, OmegaX, OmegaY : Extended;
  sx, sy : Integer;
begin
  WriteLn('Введите значение частоты колебаний по x: ');
  ReadLn(OmegaX);
  WriteLn(' Введите значение частоты колебаний по y: ');
  ReadLn(OmegaY);
  open_graph;
  SetBkColor(Blue);

```

```

sx := GetMaxX div 3 - 10;
sy := GetMaxY div 3 - 10;
t := 0;
h := 0.001;
repeat
  x := sin(OmegaX * t);
  y := cos(OmegaY * t);
  t := t + h;
  delay(5);
  putpixel(gx(x, sx), gy(y, sy), yellow);
until keypressed;
readln;
close_graph;
end.

```

В этой программе для преобразования математических координат в графические используются функции `gx` и `gy` из нашего модуля `graphs`. Разумеется, поточечное построение траектории — не самое лучшее решение, ведь траектория является плавной кривой. Но построение плавных кривых — тема другого урока. В качестве условия выхода из цикла `repeat` используется вызов процедуры `keypressed` из модуля `CRT`. Эта процедура возвращает значение «истина», если нажата любая клавиша. Таким образом, построение траектории прекращается при нажатии любой клавиши, а нажатие клавиши `Enter` завершает работу программы.

И наконец, вершина нашего мастерства. Программа `polygon` (листинг 5.4) строит довольно сложную фигуру — равносторонний многоугольник. Позволим себе небольшой эстетический изыск. Для того чтобы сделать изображение более интересным, стороны этого многоугольника мы рисовать не будем, но зато изобразим все его диагонали. Диагонали изображаются разными цветами.

#### Листинг 5.4. Программа `polygon`

```

program polygon;
uses CRT, Graph, graphs;
const
  max = 21;
  recip_max = 1.0 / max;
  two_pi = 2.0 * Pi;
var
  x, y : array [1..max] of Integer;
  color, radius, j, k: Word;
begin
  open_graph();
  radius := GetMaxY div 2 - 10;
  for j := 1 to max do
    begin
      x[j] := gx(Cos(two_pi * j * recip_max), radius);
      y[j] := gy(Sin(two_pi * j * recip_max), radius);
    end;
  for j := 1 to max do
    begin
      for k := j + 1 to max do
        if (j + k) mod 2 = 0 then
          color := red
        else
          color := blue;
        line(x[j], y[j], x[k], y[k], color);
    end;
  close_graph();
end.

```

```

y[j] := gy(Sin(two_pi * j * recip_max), radius);
end;
color := 13;
for j := 2 to max do
begin
  SetColor(color);
  for k := 1 to j - 2 do Line(x[j], y[j], x[k], y[k]);
Dec(color);
if color = 10 then
  color := 13;
end;
SetColor(White);
OutTextXY(10, 10, 'Press <Enter>: ');
ReadLn;
close_graph;
end.

```

Обратим внимание на некоторые моменты. Во-первых, в описании констант (раздел `const`) используются арифметические выражения. Далее, для хранения координат вершин многоугольника используются два массива — `x` и `y`.

Процедура `OutTextXY` служит для вывода текста на графический экран. Первые два параметра задают графические координаты начала текста, а третий параметр содержит собственно текст. В данном случае верхний левый угол текста начинается в точке (10, 10). Замечу, что, в отличие от оператора `Write` текстового режима, данная процедура предназначена только для вывода текста, поэтому если все-таки возникает необходимость вывести на экран численное значение, его следует вначале преобразовать в строку символов (для этого можно использовать процедуру `Str`).

### **Упражнение 5.1**

(Любителям игры в бильярд). Напишите программу для вывода в графическом режиме траектории движения шарика в прямоугольном бильярде. Шарик можно считать точечным объектом, трением можно пренебречь. Столкновение шарика со стенками бильярда происходит по закону зеркального отражения. Входные параметры программы — скорость и направление начального удара по шарику.

### **Упражнение 5.2**

Добавьте в программу из упражнения 5.1 возможность попадания шарика в лузу.

### **Упражнение 5.3**

(Любителям истории средних веков). В камере, находящейся во внутренней тюрьме замка Ноттингем, находится узник. Робин Гуд собирается спасти узника, и для этого он должен передать узнику план побега. Единственный способ — привязать послание к стреле и выстрелить из лука так, чтобы стрела попала в окно камеры. Между тюрьмой и стрелком стоит стена замка, в которой имеется бойница. Стена высокая, поэтому выстрелить надо так, чтобы стрела пролетела как через бойницу, так и через окно камеры. Известны: расстояние от стрелка до стены замка, толщина стены, расстояние от внутренней поверхности стены до

стены тюрьмы, высота бойницы и высота окна камеры над поверхностью земли, а также высота бойницы и окна. Написать программу, которая строит траекторию полета стрелы, и определить методом подбора угол выстрела и начальную скорость стрелы, необходимые для успеха плана Робин Гуда.

#### Упражнение 5.4

(Любителям физики). Написать программу для построения траектории полета тела в однородном поле тяжести.

#### Упражнение 5.5

Написать программу для построения траектории полета тела в фантастической ситуации, когда сила притяжения в каждый момент времени изменяется случайным образом.

#### Упражнение 5.6

(Любителям фантастики). Написать программу для построения траектории полета тела в фантастической ситуации, когда сила притяжения изменяется так, что ускорение свободного падения зависит от высоты  $h$  по закону:

$$g = 9,8 \sin(a h),$$

где  $a$  задано.

## Модуль Graph

Модуль **Graph** Турбо Паскаля содержит около пятидесяти различных процедур и функций, предназначенных для работы с графическим экраном. В этом же модуле описаны некоторые встроенные константы и переменные, которые могут быть использованы в графических программах. Напомню, что для того, чтобы воспользоваться всеми возможностями модуля **Graph**, в начале программы (после заголовка) необходимо разместить оператор использования

```
uses Graph;
```

Основную часть модуля составляют процедуры вывода базовых графических элементов, таких как точки, отрезки прямых линий, дуги и целые окружности и т. д. Такие элементы называются *графическими примитивами*. Другая группа процедур предназначена для управления графическим режимом.

Программа **test\_Graph\_unit** демонстрирует некоторые возможности модуля **Graph**. Ее исходный текст приведен в листингах 5.5 – 5.10.

**Листинг 5.5.** Описания переменных, процедуры **test\_fill\_styles** и **fill\_bar** программы **test\_Graph\_unit**

```
program test_Graph_unit;
uses crt, dos, Graph;
```

```

var
  x, y, xx, yy: integer;
  ch : char;
  s : string;
  ptr : pointer;
  size : word;

procedure test_fill_styles;
procedure fill(x, y, style : integer; color : word; ss : string);

var
  prev_color : word;

begin
  prev_color := getcolor;
  setcolor(white);
  outtextxy(x, y - 15, ss);
  setcolor(prev_color);
  setfillstyle(style, color);
  rectangle(x, y, x + 90, y + 90);
  bar(x + 1, y + 1, x + 89, y + 89);
end;{fill}

begin
  fill( 50, 70, emptyfill,      1, 'EmptyFill');
  fill(200, 70, solidfill,      2, 'SolidFill');
  fill(350, 70, linefill,       3, 'LineFill');
  fill(500, 70, ltslashfill,    4, 'LtSlashFill');
  fill( 50, 215, slashfill,     5, 'SlashFill');
  fill(200, 215, bkslashfill,   6, 'BkSlashFill');
  fill(350, 215, ltbkslashfill, 7, 'LtBkSlashFill');
  fill(500, 215, hatchfill,     8, 'HatchFill');
  fill( 50, 345, xhatchfill,    9, 'XHatchFill');
  fill(200, 345, interleavefill, 10, 'InterleaveFill');
  fill(350, 345, widedotfill,   11, 'WideDotFill');
  fill(500, 345, closedotfill,  12, 'CloseDotFill');
end;{test_fill_styles}

procedure fill_bar(x, y, xx, yy : integer; color, raster : byte);

const
  fil_pat :array[0..27] of fillpatterntype =(
    ($ff,$ff,$ff,$ff,$ff,$ff,$ff,$ff),
    (0,$fb,$fb,$fb,0,$df,$df,$df),
    (0,$10,$28,$44,$28,$10,0,0),
    ($22,0,$88,0,$22,0,$88,0),
    ($cc,$33,$cc,$33,$cc,$33,$cc,$33),
    ($aa,$55,$aa,$55,$aa,$55,$aa,$55),
    ($94,$84,$48,$30,0,$c1,$22,$14),
    ($aa,$aa,$aa,$aa,$aa,$aa,$aa,$aa),
    ($ff,0,$ff,0,$ff,0,$ff,0),
    ($ff,$1,$7d,$45,$5d,$41,$7f,0),
    
```

```

($01,$82,$44,$28,$10,$20,$40,$80),
(0,$3c,$42,$42,$42,$42,$3c,0),
(0,$7e,$7e,$7e,$7e,$7e,$7e,0),
($81,$42,$24,$18,$18,$24,$42,$81),
(0,$ec,$2a,$2a,$2a,$aa,$ec,0),
(0,$08,$18,$3f,$3f,$18,$08,0),
(0,0,$7e,$42,$7e,$42,$7e,$42),
($80,$7f,$41,$41,$41,$41,$7f),
(0,$5d,$3e,$6b,$7f,$63,$36,$5d),
(0,0,$04,$08,$90,$a0,$c0,$f0),
($92,$24,$49,$92,$24,$49,$92,$24),
($b1,$22,$14,$14,$22,$91,$48,$24),
($18,$3c,$3c,$7e,$7e,$3c,$3c,$18),
($e7,$c3,$81,$18,$18,$81,$c3,$e7),
($2,$91,$68,$8,$10,$16,$89,$40),
($ff,$81,$81,$81,$81,$81,$81,$ff),
(0,0,$81,$81,$42,$24,$18,$18),
($c3,$42,$5a,$7e,$5a,$42,$c3));

```

```

begin
  if raster < 28 then
    begin
      setfillpattern(fil_pat[raster], color);
      setfillstyle(12, color);
      bar(x, y, xx, yy);
    end;
  end::{fill_bar}

```

Процедуры `test_fill_styles` и `fill_bar` демонстрируют вывод геометрических фигур с замкнутой границей (в нашем случае это прямоугольники) и заполнение их внутренней области определенным шаблоном. Прямоугольник строится процедурой `Rectangle(x1, y1, x2, y2)`, первые два параметра которой задают графические координаты верхнего левого угла прямоугольника, а последние два параметра — координаты нижнего правого угла. Процедура `Bar(x1, y1, x2, y2)` также предназначена для вывода прямоугольника и отличается от процедуры `Rectangle` тем, что в этом случае внутренняя часть прямоугольника закрашивается заранее определенным образом. В случае процедуры `Rectangle` для заполнения внутренней области фигуры приходится использовать дополнительный вызов процедуры модуля `Graph`. Цвет заполнения может быть задан с помощью процедуры `SetColor(Color)`. Аргументом этой процедуры является константа или переменная типа `Word`, которая является кодом цвета. Этот код может принимать значения из интервала `0..15`. Удобнее использовать встроенные константы модуля `Graph`, имена которых соответствуют выводимым цветам. Перечень цветов, кодов и соответствующих им встроенных констант был дан в табл. 5.3. Способ закраски определяется не только цветом, но и *стилем* заполнения. В Турбо Паскале имеется 12 стилей заполнения, а выбрать любой из них можно с помощью процедуры `SetFillStyle(Style, Color)`. Первый параметр этой процедуры задает стиль заполнения, а второй — цвет. В числе допустимых стилей закраски — сплошная, штриховая и т. д. (табл. 5.4).

Таблица 5.4. Стили заполнения геометрических фигур

Константа	Код	Описание
EmptyFill	0	Сплошное заполнение цветом фона
SolidFill	1	Сплошное заполнение заданным цветом
LineFill	2	Заполнение горизонтальными линиями
LtSlashFill	3	Диагональное заполнение (///)
SlashFill	4	Диагональное заполнение толстыми линиями (///)
BkSlashFill	5	Обратное диагональное заполнение толстыми линиями (\\\)
LtBkSlashFill	6	Обратное диагональное заполнение (\\\)
HatchFill	7	Клетчатое заполнение
XhatchFill	8	Косое клетчатое заполнение
InterleaveFill	9	Чередующееся линейное заполнение
WideDotFill	10	Редко расположенные точки
CloseDotFill	11	Часто расположенные точки
UserFill	12	Стиль определен пользователем

В процедуре `fill_bar` используются стили заполнения, заданные пользователем. Первые четыре параметра этой процедуры задают левый верхний и правый нижний углы прямоугольной области. Пятый параметр определяет цвет заполнения, а шестой представляет наибольший интерес. Это индекс, позволяющий выбрать из внутреннего массива процедуры `fil_pat` элемент, содержащий пользовательский стиль заполнения. Каждый элемент этого массива имеет тип `FillPatternType`, являющийся встроенным типом модуля `Graph`. Переменная типа `FillPatternType`, в свою очередь, является массивом, состоящим из восьми элементов типа `Byte`. Каждая такая переменная описывает шаблон размером 8×8 пикселов. Шаблон можно рассматривать как матрицу (таблицу) соответствующего размера, элементы которой принимают единичное или нулевое значение. Единичное значение соответствует «включенному» пиксели, а нулевое — «погашенному». Так, например, элементу `fil_pat[1]` соответствует сплошная закраска, а второму, как это видно из табл. 5.5, — заполнение «кирпичками»:

В листинге 5.6 приведен текст процедур `shadow_text` и `info`. Процедура `shadow_text` выводит заданный текст. Для вывода надписи используется процедура `OutTextXY`, первые два параметра которой определяют графические координаты начала текста, а последний параметр является строковым значением, выводимым на экран. Используя эту процедуру, мы идем на маленькую хитрость — выводим на экран один и тот же текст два раза — с небольшим смещением и разными цветами. Такой незамысловатый трюк создает иллюзию выпуклости надписи. Просто, но красиво! Параметры процедуры `shadow_text` позволяют задать не только расположение текста (`x, y`), но и его цвет (`bg`), цвет «тени» (`vg`), а также шрифт надписи (о шрифтах разговор у нас пойдет позже) и его размер (`font, size`).

Таблица 5.5. Шаблон заполнения, соответствующий элементу fil\_pat[1]

Биты								Байт
7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	0	\$00
1	1	1	1	1	0	1	1	\$fb
1	1	1	1	1	0	1	1	\$fb
1	1	1	1	1	0	1	1	\$fb
0	0	0	0	0	0	0	0	\$00
1	1	0	1	1	1	1	1	\$df
1	1	0	1	1	1	1	1	\$df
1	1	0	1	1	1	1	1	\$df

Процедуре info в нашей программе отведена простая, но почетная роль — она очерчивает прямоугольную пустую область, в которой будут отображаться графические элементы, и выводит надпись в верхней части экрана.

Листинг 5.6. Процедуры shadow\_text и info программы test\_Graph\_unit

```

procedure shadow_text(x, y : integer; text : string;
  bg, vg, font, size : byte);
begin
  settextstyle(font, 0, size);
  setcolor(bg);
  outtextxy(x, y, text);
  setcolor(vg);
  outtextxy(x + 1, y + 1, text);
  settextstyle(0, 0, 0);
end;

procedure info(s : string);
begin
  fill_bar(0, 0, 639, 49, 0, 0);
  shadow_text(20, 5, s, 14, 6, 2, 6);
end;

```

В процедуре test\_get\_put\_image (листинг 5.7) используются две процедуры модуля Graph – GetImage и PutImage. Первая из них копирует битовый образ указанной области графического экрана в память. Размер прямоугольной области копирования задается первыми четырьмя параметрами, а последний представляет собой нетипизированный параметр, который хранит размеры изображения и само изображение. Размер области памяти, отводимой под последний параметр, определяется правилом «6 + размер изображения». Вторая процедура (PutImage) выполняет обратное действие — выводит на экран сохраненное в памяти изобра-

жение. Первые два параметра этой процедуры задают графические координаты вывода изображения на экран, третий параметр указывает на хранимое в памяти изображение, а последний (типа Word) определяет способ вывода изображения на экран. Этот параметр задает побитовую логическую операцию, которая применяется к битам изображения на экране и изображения, хранимого в памяти. Результат этой операции и отображается в конечном счете на экране. Вместо числовых значений в качестве четвертого параметра удобнее использовать константы модуля Graph. Список этих констант приводится ниже:

Константа	Значение
CopyPut	0
XORPut	1
OrPut	2
AndPut	3
NotPut	4

Для ссылки на область памяти, в которой сохраняется изображение, используется указатель. Копирование изображений в память и их вывод на экран можно использовать для программирования движущихся (динамических) изображений, поскольку такой способ эффективнее простой перерисовки экрана.

Функция `ImageSize` вычисляет размер буфера, необходимого для хранения требуемой части экрана.

#### Листинг 5.7. Процедура `test_get_put_image` программы `test_Graph_unit`

```
procedure test_get_put_image;
var
  ptr : pointer;
  size : word;
begin
  setfillstyle(solidfill, lightcyan);
  fillellipse(100, 210, 20, 20);
  circle(140, 210, 20);
  rectangle(80, 190, 160, 230);
  size := imagesize(80, 190, 160, 230);
  getmem(ptr, size);
  getimage(80, 190, 160, 230, ptr^);
  setcolor(white);
  setfillstyle(solidfill, blue);
  bar(280, 60, 400, 440);
  putimage(300, 100, ptr^, CopyPut);
  putimage(300, 160, ptr^, orput);
  putimage(300, 220, ptr^, xorput);
  putimage(300, 280, ptr^, notput);
  putimage(300, 340, ptr^, andput);
  outtextxy(410, 120, 'CopyPut');
  outtextxy(410, 180, 'OrPut');
```

```

outtextxy(410, 240, 'XorPut');
outtextxy(410, 300, 'NotPut');
outtextxy(410, 360, 'AndPut');
freemem(ptr, size);
end;

```

При работе с цветными изображениями большое значение имеет набор цветов, доступных программисту. Количество доступных для воспроизведения цветов определяется возможностями аппаратуры, то есть дисплея и видеoadаптера, а также видеорежимом. Набор цветов составляет цветовую *палитру*. В программировании обычно используется схема цветового представления, в которой считается, что каждый цвет является смесью трех основных цветов: красного (Red), зеленого (Green) и синего (Blue), а оттенок определяется интенсивностью компонентов. Эта схема называется *RGB-представлением*. Использование RGB-представления связано с конструктивными особенностями электронно-лучевой трубки, в которой каждая графическая точка состоит из трех компонентов: красного, зеленого и синего. Для каждого основного цвета имеется 256 возможных интенсивностей. Нетрудно подсчитать, что всего таким образом можно задать  $256 \times 256 \times 256 = 16\ 777\ 216$  цветов и оттенков.

При инициализации видеорежима компьютер создает цветовую палитру, которую можно считать таблицей, где каждому цвету сопоставлен свой номер. Для описания цвета в системе RGB используются 3 байта:

<b>Синий</b>	<b>Зеленый</b>	<b>Красный</b>
bbbbbbbb	gggggggg	rrrrrrrr

Каждый байт описывает интенсивность соответствующего основного цвета, таким образом интенсивно-красный цвет имеет номер \$0000FF (шестнадцатеричное представление), а белому цвету соответствует значение \$FFFFFF.

При работе с цветом в режиме VGA интенсивность основного цвета задается значением типа *Word*, в котором реально используются только 6 младших битов. Шестнадцатицветная палитра VGA приведена в табл. 5.6.

**Таблица 5.6. Цветовая палитра VGA**

Индекс	Цвет	RGB	Индекс	Цвет	RGB
0	Черный	\$000000	8	Серый	\$202020
1	Синий	\$200000	9	Светло-синий	\$3F0000
2	Зеленый	\$002000	10	Светло-зеленый	\$003F00
3	Голубой	\$202000	11	Светло-голубой	\$3F3F00
4	Красный	\$000020	12	Светло-красный	\$00003F
5	Фиолетовый	\$200020	13	Светло-фиолетовый	\$3F003F
6	Коричневый	\$002020	14	Желтый	\$003F3F
7	Белый	\$303030	15	Ярко-белый	\$3F3F3F

В режиме VGA можно задать  $64 \times 64 \times 64 = 262\,144$  цвета, причем одновременно не более 256 (при самом грубом разрешении), обычно же лишь 16.

Процедура `SetRGBPalette` позволяет изменить в используемой цветовой палитре цвет с заданным индексом. Вызывается она следующим образом:

```
procedure SetRGBPalette(ColorNum, RedValue, GreenValue, BlueValue : Integer)
```

Первый аргумент является индексом цвета, а второй, третий и четвертый параметры позволяют задать интенсивности основных цветов для данного цвета.

Процедура `test_palette` (листинг 5.8) изменяет интенсивности цветов случайным образом. Обратите внимание на то, что `test_palette` начинается с описания константы типа «массив», использование которой позволяет автоматизировать выбор индекса цвета, заданного при помощи встроенной константы модуля `Graph`.

#### Листинг 5.8. Процедура `test_palette` программы `test_Graph_unit`

```
procedure test_palette;
begin
  randomize;
  SetRGBPalette(col_num[lightred], 63, 10, 10);
  setfillstyle(solidfill, lightred);
  bar(30, 50, 179, 450);
  SetRGBPalette(col_num[lightgreen], 9, 63, 9);
  setfillstyle(solidfill, lightgreen);
  bar(180, 50, 329, 450);
  SetRGBPalette(col_num[lightblue], 0, 0, 63);
  setfillstyle(solidfill, lightblue);
  bar(330, 50, 479, 450);
  SetRGBPalette(col_num[brown], 40, 30, 0);
  setfillstyle(solidfill, brown);
  bar(480, 50, 629, 450);
  setcolor(white);
  outtextxy(30, 50, 'LightRed');
  outtextxy(180, 50, 'LightGreen');
  outtextxy(330, 50, 'LightBlue');
  outtextxy(480, 50, 'Brown');
  delay(1000);
  repeat
    SetRGBPalette(col_num[lightred], 63, random(40), random(40));
    delay(100);
    SetRGBPalette(col_num[lightgreen], random(40), 63, random(40));
    delay(100);
```

```

SetRGBPalette(col_num[lightblue], random(40), random(40), 63);
delay(100);
SetRGBPalette(col_num[brown], 30 + random(20), 20 + random(20), 0);
delay(100);
until keypressed;
end;

```

Процедуры `horizontal_reflection` и `vertical_reflection` (листинг 5.9) выполняют зеркальное отражение изображения относительно горизонтальной и вертикальной осей, проходящих через центр экрана. Зеркальное отражение реализовано перестановкой пикселов, расположенных симметрично относительно соответствующей линии. Для этого используются функция `GetPixel(x, y)` и процедура `PutPixel(x, y, color)`. Первая из них определяет цвет пикселя в точке с заданными графическими координатами, а вторая нам уже знакома — она выводит пикセル в заданной точке и с указанным цветом.

**Листинг 5.9.** Процедуры `horizontal_reflection` и `vertical_reflection` программы `test_Graph_unit`

```

procedure horizontal_reflection(mx, my, mxx, myy : integer);
var
  pix1, pix2 : integer;
begin
  if (mx < 0) or (my < 0) or (mxx > GetMaxX) or
    (myy > GetMaxY) then exit;
  for x := mx to mxx do
    for y := my to (my + myy) div 2 do
      begin
        pix1 := getpixel(x, y);
        pix2 := getpixel(x, my + myy - y);
        putpixel(x, my + myy + y, pix1);
        putpixel(x, y, pix2);
      end;
end;

procedure vertical_reflection(mx, my, mxx, myy : integer);
var
  pix1, pix2 : integer;
begin
  if (mx < 0) or (my < 0) or (mxx > GetMaxX) or
    (myy > GetMaxY) then exit;
  for y := my to myy do
    for x := mx to (mx + mxx) div 2 do
      begin
        pix1 := getpixel(x, y);
        pix2 := getpixel(mx + mxx - x, y);
        putpixel(mx + mxx - x, y, pix1);
        putpixel(x, y, pix2);
      end;
end;

```

В разделе операторов программы (листинг 5.10) инициализируется графический режим. При этом используется драйвер VGA, а режим `VGAhi` соответствует разрешению 640x480 пикселов и поддержке 16 цветов.

#### Листинг 5.10. Раздел операторов программы `test_Graph_unit`

```

begin
  x := vga; y := vgahi;
  InitGraph(x, y, '');
  ClearViewPort;
  setcolor(15);
  rectangle(10, 50, 639, 450);
  info('Press <Enter>');
  ReadLn;

  info('Fill_Bar. Press <Enter>');
  fill_bar(11, 51, 638, 449, 3, 5);
  fill_bar(50, 100, 280, 180, 4, 4);
  fill_bar(450, 80, 580, 150, 0, 0);
  fill_bar(500, 350, 600, 430, 14, 20);
  ReadLn;

  info('Shadow_text. Press <Enter>');
  Shadow_text(100, 200, 'SHADOWTEXT...', 15, 7, 0, 2);
  Shadow_text(120, 230, 'HELLO!HELLO!!!!', 13, 5, 0, 3);
  Shadow_text(140, 260, 'good bye, good bye...', 0, 11, 2, 6);
  ReadLn;

  info('Test_get_put_image. Press <Enter>');
  test_get_put_image;
  ReadLn;

  clearviewport;
  info('Test_fill_styles. Press <Enter>');
  rectangle(10, 50, 639, 450);
  fill_bar(11, 51, 638, 449, 3, 5);
  test_fill_styles;
  ReadLn;

  info('Test_palette. Press <Enter>');
  test_palette;
  ReadLn;

  info('FloodFill. Circle, Ellipse, Setfillstyle. Press <Enter>');
  setcolor(15);
  circle(100, 350, 50);
  setfillstyle(1, 8);
  floodfill(100, 350, 15);
  setcolor(13);
  fillellipse(150, 350, 30, 60);
  setfillstyle(2, 4);
  floodfill(150, 350, 13);

```

```

    setcolor(10); // цвет для отображения, который определяет цвета символов
    setfillstyle(4, 5); // стиль заливки, определяет способ заполнения пикселей
    fillellipse(250, 150, 90, 60); // рисует эллипс с центром в точке (250, 150), радиусами 90 и 60
    ReadLn; // ожидает нажатия клавиши Enter для продолжения выполнения программы
    info('Vertical reflection. Press <Enter>');
    vertical_reflection(0, 50, 639, 450);
    ReadLn;
    info('Horizontal reflection. Press <Enter>');
    horizontal_reflection(0, 50, 639, 450);
    ReadLn;
end.

```

Приятно, что в Турбо Паскале есть поддержка.

Обратимся теперь к другой проблеме, часто возникающей при программировании графического вывода в Турбо Паскале. Эта проблема — вывод текста. Очевидно, чтобы иметь возможность вывода текстовой информации, система программирования должна поддерживать работу не менее чем с одним шрифтом. Шрифт представляет собой набор символов, используемых для отображения текстовой информации. С точки зрения «потребителя», шрифты различаются начертанием символов. Программисту важен еще и способ хранения информации о форме символов. При работе с графикой Турбо Паскаля используются два вида шрифтов, различающихся своим внутренним форматом — *растровые* (он один) и *векторные* (их несколько).

Растровый символ задается с помощью матрицы элементов изображения этого символа. Матрица имеет размер 8×8 пикселов. Векторный шрифт задается набором векторов, которые указывают графической системе, как рисовать символ.

Разница между растровым и векторными шрифтами становится очевидной при отображении символов увеличенного размера (рис. 5.2).



Рис. 5.2. Растворный (верхняя надпись) и векторный (нижняя надпись) шрифты

Поскольку векторный шрифт задается векторами, то при увеличении шрифта качество и разрешение остаются хорошими. При использовании растрового шрифта для отображения увеличенных символов битовая матрица умножается на масштабный коэффициент, а когда этот масштабный коэффициент большой, разрешение становится более грубым. Для вывода мелких надписей можно использовать растворный шрифт, но для больших символов следует использовать векторные шрифты.

Использование различных шрифтов демонстрирует программа fonts (см. листинг 5.11). Она содержит процедуры, управляющие размещением текста на экране: SetTextStyle и SetTextJustify. Векторные шрифты существуют в виде отдельных файлов (хотя имеется способ их включения в исполняемый код программы), имеющих расширение .chr и расположенных в том же каталоге BGI Турбо Паскаля, что и графические драйверы. Чтобы программа выполнялась правильно, ей должны быть доступны файлы со всеми необходимыми шрифтами. Если графическая программа не сможет найти необходимый шрифт, то это не приведет к сбоям в ее работе, будет просто использоваться шрифт, заданный по умолчанию. В нашем случае для нормальной работы программы в рабочем каталоге программы должны находиться файлы, перечисленные в табл. 5.7.

**Таблица 5.7. Векторные шрифты Турбо Паскаля**

Константа	Код	Файл
TriplexFont	1	trip.chr
SmallFont	2	litt.chr
SansSerifFont	3	sans.chr
GothicFont	4	goth.chr

Следует отметить, что «стандартные» (то есть входящие в исходную поставку Турбо Паскаля) векторные шрифты не содержат русских букв. Впрочем, существуют векторные шрифты, созданные программистами и поддерживающие работу с русским алфавитом. Их можно попытаться найти, используя возможности глобальной компьютерной сети Интернет. Для вывода текста используется процедура OutTextXY. Способ вывода текста можно задать, вызвав предварительно процедуру SetTextStyle. Первый параметр этой процедуры задает шрифт, второй — направление текста, горизонтальное (встроенная константа HorizDir) или вертикальное (VertDir), и, наконец, третий — размер шрифта (масштабный множитель). Для растрового шрифта значение 1 масштабного множителя соответствует битовой матрице 8×8, а 2, например, — битовой матрице 16×16. Процедура SetTextJustify(horJ, verJ) устанавливает выравнивание в горизонтальном (первый параметр) и вертикальном (второй параметр) направлениях. Типы выравнивания приведены в табл. 5.8.

Процедура ClearDevice устанавливает текущий указатель в исходное положение (точка с координатами (0, 0)) и очищает экран, заполняя его цветом фона.

**Таблица 5.8. Типы выравнивания текста в процедуре SetTextJustify**

Горизонтальное		Вертикальное			
Константа	Код	Выравнивание	Константа	Код	Выравнивание
LeftText	0	Влево	BottomText	0	Вниз
CenterText	1	По центру	CenterText	1	По центру
RightText	2	Вправо	TopText	2	Вверх

**Листинг 5.11.** Программа, демонстрирующая работу со шрифтами

```

program fonts;
uses crt, Graph, Graphs;
procedure show_font(font : word; ss : string);
var
  j : word;
begin
  for j := 1 to 7 do
  begin
    settextstyle(font, horizdir, j);
    outtextxy(0, 50 * j, ss);
  end;
  settextstyle(font, vertdir, 3);
  outtextxy(600, 0, 'Press <Enter>');
  readln;
  cleardevice;
end;

procedure justify;
const
  ss='JUSTIFY';
var
  j, k, l : word;
begin
  settextstyle(sansseriffont, horizdir, 1);
  setfillstyle(solidfill, lightgray);
  for j:= 0 to 2 do
    for k := 0 to 2 do
    begin
      rectangle(50+j * 150, 15 + 150 * j + 50 * k,
                200+j * 150, 60 + 150 * j + 50 * k);
      floodfill(60 + j * 150, 20 + 150 * j + 50 * k, white);
      settextjustify(j, k);
      outtextxy(120+j * 150, 30 + 150 * j + 50 * k , ss);
    end;
  outtextxy(200, 430, 'Press <Enter>');
end;

begin
  open_Graph;
  show_font(defaultfont, 'Default Font');
  show_font(triplexfont, 'TriplexFont');
  show_font(smallfont, 'SmallFont');
  show_font(sansseriffont, 'SansSerifFont');
  show_font(gothicfont, 'GothicFont');
  justify;
  ReadLn;
  close_Graph;
end.

```

При использовании различных шрифтов, находящихся в файлах \*.chr, неудобным является то, что эти дополнительные файлы должны сопровождать файл с программой. К счастью, как уже упоминалось, имеется способ включения файлов со шрифтами и графическими драйверами в исполняемый файл. Для этого можно, например, использовать модуль `bgi_chr` (листинг 5.12). Перед компиляцией этого модуля следует воспользоваться утилитой `BINOBJ.EXE`, которая находится в подкаталоге `/BIN` каталога Турбо Паскаля. Эта утилита преобразует двоичные файлы драйвера или шрифта в формат объектного файла, который может быть включен в исполняемый код при компиляции программы. Утилита вызывается следующим образом:

```
BINOBJ EGAVGA.BGI EGAVGA EGAVGADriverProc <Enter>
BINOBJ LITT.CHR LITT SmallFontProc <Enter>
```

Здесь первый параметр — это исходный файл, второй — имя объектного файла, третий — имя процедуры. После запуска утилиты `BINOBJ.EXE` полученные объектные файлы следует разместить в том же каталоге, где находится файл с исходным текстом модуля `bgi_chr`, и откомпилировать этот модуль.

#### Листинг 5.12. Модуль `bgi_chr`

```
unit bgi_chr;

interface

procedure EgaVgaDriverProc;
procedure SmallFontProc;

implementation

procedure EgaVgaDriverProc; external;
{$L EGAVGA.OBJ }

procedure SmallFontProc; external;
{$L LITT.OBJ }

end.
```

В модуле `bgi_chr` используется директива компилятора `{$L..}`. Эта директива позволяет использовать на этапе компоновки и создания исполняемого кода внешний объектный файл (имеющий расширение .OBJ), имя которого указывается в директиве. Чтобы использовать этот модуль в графической программе, необходимо в начале программы разместить оператор использования

```
uses bgi_chr;
```

а в разделе операторов программы — процедуры «регистрации» драйвера и/или шрифта:

```
if RegisterBGIDriver(@EGAVGADriverProc) < 0 then
begin
  writeln('No EGAVGA Driver!');
```

```

    write('Press <Enter>: '); readln;
    halt;
end;
begin
  if RegisterBGIFont(@SmallFontProc) < 0, then
  begin
    writeln('No Small Font!');
    write('Press <Enter>: '); readln;
    halt;
  end;
end.

```

Процедура `RegisterBGIfont` модуля `Graph` передает графической системе указатель на шрифт, который был включен в исполняемый код на этапе компиляции. Она возвращает отрицательное целое значение при возникновении ошибки. В случае нормального выполнения возвращается код шрифта, и этот шрифт можно использовать в графических процедурах.

Процедура `RegisterBGIdriver` модуля `Graph` передает графической системе указатель на драйвер видеoadаптера, включенный в исполняемый код программы. В остальном действие этой процедуры аналогично действию процедуры `RegisterBGIfont`.

#### Упражнение 5.7

Определите и нарисуйте стили заполнения в процедуре `fil_bar` программы 5.1 для всех элементов массива `fil_pat`.

#### Упражнение 5.8

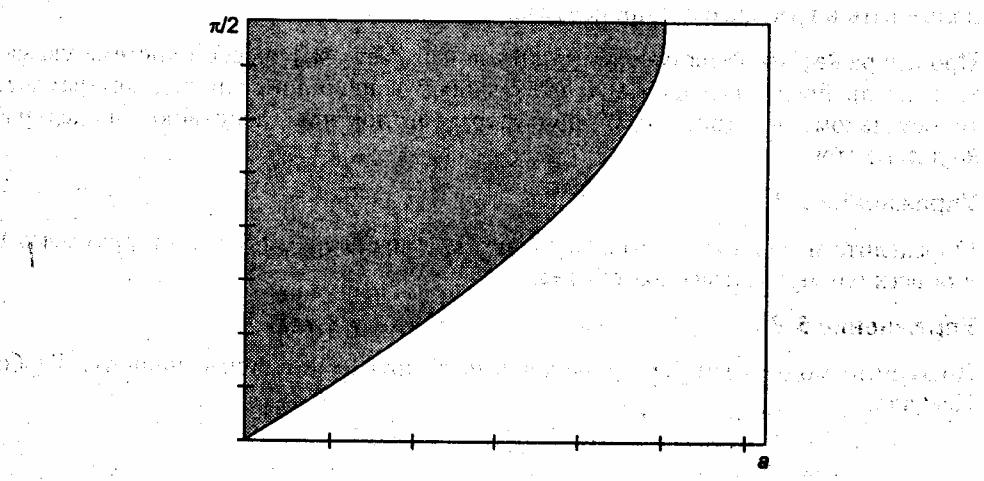
Дополните модуль `bgi_chr`, включив в него другие векторные шрифты Турбо Паскаля.

## Программа «Игла Бюффона» (бросаем иголки по-научному)

Программа `buffon` (листинг 5.13) предназначена для моделирования классического опыта, описание которого можно найти во многих учебниках по теории вероятности. Этот опыт позволяет экспериментально определить приближенное значение числа «пи» (отношение длины окружности к ее диаметру).

Начнем с краткого изложения теории. На плоскость с нанесенными на ней на расстоянии  $2a$  друг от друга параллельными прямыми случайно бросается прямолинейный отрезок («игла») длиной  $2l$ . Требуется найти вероятность того, что отрезок пересечет хотя бы одну из прямых. Решать задачу можно следующим образом. Положение отрезка относительно параллельных прямых характеризуется двумя величинами: расстоянием его центра от ближайшей прямой  $x$  ( $0 \leq x \leq a$ ) и

углом его наклона по отношению к прямым  $\theta$  ( $0 \leq \theta \leq \pi / 2$ ). Игла пересекает хотя бы одну из прямых только в том случае, когда выполняется условие  $x \leq l \sin(\theta)$ . Если рассматривать  $\theta$  и  $x$  как прямоугольные декартовы координаты некоторой точки на плоскости, то область всех допустимых значений этих координат будет представлять собой прямоугольник со сторонами  $a$  и  $\pi / 2$  (рис. 5.3). Пересечению иглы с одной из параллельных линий соответствует серая область на рисунке, ограниченная частью синусоиды. Считая все положения точки  $(\theta, x)$  равновероятными в пределах прямоугольника, мы можем вычислить вероятность пересечения иглы с линией как отношение площади серой области к площади всего прямоугольника. При  $l \leq a$  это даст  $2l/\pi a$ . Эту вероятность, с другой стороны, можно определить опытным путем, выбрасывая иглу на систему параллельных линий и оценивая вероятность как отношение числа пересечений к полному числу испытаний. Зная  $l$  и  $a$ , таким образом можно найти приближенное значение числа  $\pi$ .



**Рис. 5.3.** Геометрическая иллюстрация к опыту «Игла Бюффона»

**Листинг 5.13.** Программа «Игра Бюффона»

```

program buffon;
uses crt, Graph, Graphs;
var crossed, thrown, drawlinecount, ltemp, rescalecount : longint;
    half_length: double;
    nline, offset, step : word;
procedure drawlines;
var
    row : word;
begin
    setlinestyle(0, 0, -1);

```

```
setcolor(white);
row := offset;
nline := 0;
while (row <= getmaxy) do
begin
    line(0, row, getmaxx, row);
    inc(row, step);
    inc(nline)
end
end;

procedure setuplines;
begin
    clearviewport;
    offset := getmaxy div 10;
    step := getmaxy div 5;
    half_length := 0.5 * step;
    drawLines
end;

procedure beep;
begin
    sound(500); delay(2); nosound;
end;

procedure thrown_needle;
var
    alpha, cos_theta, hypotenuse, sin_theta, x, y : real;
    x_delta, y_delta, x_mid, y_mid : integer;
begin
    alpha := random - 0.5;
    x := 2.0 * (random - 0.5);
    y := random;
    if x = 0.0 then
begin
    sin_theta := 1.0;
    cos_theta := 0.0
end
else
begin
    hypotenuse := sqrt(sqr(x) + sqr(y));
    sin_theta := y / hypotenuse;
    cos_theta := x / hypotenuse
end;
inc(thrown);
if abs(alpha) <= 0.5 * sin_theta then
begin
    inc(crossed);
    beep;
end;
x_mid := random(getmaxx + 1);
y_mid := offset + step * random(1 + nline) + round(alpha * step);
```

```

x_delta := round(half_length * cos_theta);
y_delta := round(half_length * sin_theta);
setcolor(1 + random(getmaxcolor));
line (x_mid - x_delta, y_mid - y_delta, x_mid + x_delta, y_mid + y_delta);
end;

procedure buffon_finished;
var
  pi_approximate : real;
begin
  clrscr;
  writeln;
  writeln ('Опыт "Игра Бюффона" - экспериментальное определение
           числа пи');
  writeln;
  writeln('Брошено иголок      ', thrown:9);
  write('Пересечений с линиями ', crossed:9);
  writeln;
  if crossed > 0 then
    begin
      pi_approximate := 2.0 * thrown / crossed;
      writeln('Приближенное значение числа пи ', pi_approximate:8:6);
    end;
  writeln;
end;

begin
  randomize;
  thrown := 0;
  crossed := 0;
  rescalecount := 2000;
  drawlinecount := rescalecount div 5;
  open_Graph;
  setviewport (0.0, getmaxx, getmaxy, clipon);
  setuplines;
  repeat
    thrown_needle;
    delay(200)
  until keypressed;
  close_Graph;
  buffon_finished;
  WriteLn('Нажмите <Enter>');
  readln; readln;
end.

```

Разберем работу этой программы. Прежде всего напомню, что речь идет о моделировании эксперимента со случным исходом, поэтому в данной программе не обойтись без моделирования случайных чисел с так называемым равномерным распределением. В этом случае вначале необходимо вызвать процедуру инициализации датчика случайных чисел `Randomize` и лишь затем можно вызывать саму функцию генерации случайных чисел `Random(n)`. Если эта функция вызывается

без параметра, она возвращает случайное вещественное число в интервале от 0 до 1. При вызове с целочисленным параметром результат — случайное целое число (типа Word) в диапазоне от 0 до числа, на единицу меньшего, чем заданный параметр.

После выполнения операторов и процедур инициализации, в том числе графического режима (последний инициализируется и закрывается процедурами `open_Graph` и `close_Graph` нашего модуля `Graphs`), вызывается процедура `setup_lines`, которая определяет параметры рисования параллельных линий и рисует эти линии на экране. Графическая процедура `SetViewPort` устанавливает размеры и положение области для графического вывода. Все последующие процедуры графического вывода позиционируются относительно границ этой области.

После вывода на экран изображения параллельных линий начинает свою работу цикл `repeat...until`, который прекращается при нажатии на произвольную клавишу. В этом цикле разыгрываются случайные выбрасывания иглы на плоскость. При каждом выбрасывании иглы проверяется, выполняется ли условие ее пересечения с одной из параллельных линий. При этом ведется учет числа пересечений и общего числа выбрасываний иглы.

Итоги подводит процедура `buffon_finished`, которая выводит статистику эксперимента — количество испытаний и количество пересечений, а также приближенную оценку числа  $\pi$ .

## Программа «Жизнь»

А теперь перейдем к программированию известной игры Дж. Конвея «Жизнь». Это, собственно, не игра, а простая модель эволюции сообщества виртуальных организмов. Пассивное наблюдение над совокупностью большого числа поколений этой игры может доставить некоторое удовольствие (и даже определенное ощущение собственной значимости, ведь судьбы виртуального сообщества в ваших руках!).

Игровым полем, то есть жизненным пространством в этой игре является двумерная поверхность (у нас это поверхность тора, которая не имеет границ: наличие границ всегда связано с проблемами), разделенная на квадратные ячейки. У каждой ячейки имеется 8 соседей. Ячейка может быть заселена одним организмом («живая» ячейка), а может быть пустой («мертвая»). Популяция в первом поколении задается случайным образом. Это значит, что решение о том, будет ли каждая конкретная ячейка заселена, принимается с некоторой вероятностью.

Распределение населенных ячеек в следующем поколении определяется правилами, которые применяются одновременно ко всем ячейкам:

- живая ячейка, имеющая не более одной живой соседки, погибает от одиночества;
- живая ячейка, имеющая 4 или больше живых соседей, погибает от перенаселения;

- мертвая ячейка с тремя живыми соседями возрождается;
- во всех прочих случаях состояние ячейки не изменяется.

В программе *life* (листинг 5.14) при отображении эволюции сообщества виртуальных организмов применен прием *анимации*, то есть вывода динамического (изменяющегося) изображения. Эффект здесь основан на последовательной смене кадров — графических изображений. Для быстрого перехода от одного «кадра» с изображением популяции к другому мы будем использовать две графические страницы. Графическая страница представляет собой область видеопамяти, которая хранит изображение. При этом, если режим работы видеoadаптера поддерживает работу только с одной страницей, ее содержимое отображается на экране дисплея. В некоторых режимах работы могут поддерживаться несколько графических страниц. Содержимое одной из них («визуальной» страницы) отображается на экране, а на второй («активной»), невидимой глазу пользователя, в это время может строиться новое изображение. Подготовленное на активной странице изображение можно вывести на экран. Такой метод возможен, например, при работе в режиме VGA с разрешением 640×350, так как в этом режиме есть две графические страницы.

В разделе описаний констант заданы параметры программы, которые если и придется менять, то редко, а потому неудобно их вводить заново при каждом запуске программы:

- hor* — количество ячеек по горизонтали;
- ver* — количество ячеек по вертикали;
- cell\_width*, *cell\_height* — ширина и высота ячейки;
- prob\_factor* — параметр, определяющий вероятность заселения ячеек при формировании начальной популяции.

Размеры приведены в пикселях.

#### Листинг 5.14. Программа «Жизнь»

```
program life;
uses crt, dos, Graph;
const
  hor = 100;
  ver = 70;
  cell_width = 8;
  cell_height = 6;
  prob_factor = 0.5;
var
  old_gen, new_gen : array[0..ver, 0..hor] of 0..1;
  prob : real;
  ch : char;
  x_center : array[0..hor] of word;
```

```
y_center : array[0..ver] of word;
gen_count, radius, page : word;
ss : string[10];

procedure init_cells;
var
  j, k : word;
begin
  gen_count := 0;
  for j := 0 to ver do
    for k := 0 to hor do
      begin
        old_gen[j, k] := 0;
        if random <= prob then
          new_gen[j, k] := 1
        else
          new_gen[j, k] := 0;
      end;
end;

procedure next_generation;
var
  j, k, m, prev_j, next_j, prev_k, next_k : word;
begin
  old_gen := new_gen;
  for j := 0 to ver do
    begin
      if j = 0 then
        prev_j := ver
      else
        prev_j := j - 1;
      if j = ver then
        next_j := 0
      else
        next_j := j + 1;
      for k := 0 to hor do
        begin
          if k = 0 then
            prev_k := ver
          else
            prev_k := k - 1;
          if k = hor then
            next_k := 0
          else
            next_k := k + 1;
          m := old_gen[prev_j, prev_k]
            + old_gen[prev_j, k]
            + old_gen[prev_j, next_k]
            + old_gen[j, prev_k]
            + old_gen[j, next_k]
            + old_gen[next_j, prev_k]
            + old_gen[next_j, k]
        end;
    end;
end;
```

```
    + old_gen[next_j, next_k];
    if (old_gen[j, k] = 1) and ((m <= 1) or (m >= 4)) then
        new_gen[j, k] := 0
    else
        if (old_gen[j, k] = 0) and (m = 3) then
            new_gen[j, k] := 1
        else
            new_gen[j, k] := old_gen[j, k];
    end;
end;
end;

procedure init_screen;
var
    GraphDriver, GraphMode : integer;
    j, k : word;
begin
    GraphDriver := VGA; GraphMode := VGAMed;
    page := 0;
    InitGraph(GraphDriver, GraphMode, '');
    if GraphResult <> grOK then
        halt;
    for k := 0 to hor do
        x_center[k] := k * cell_width + cell_width div 2;
    for j := 0 to ver do
        y_center[j] := j * cell_height + cell_height div 2;
    radius := 4;
end;

procedure display;
var
    j, k : word;

procedure rule_plane;
var
    j, k : word;

begin
    setviewport(0, 0, getmaxx, getmaxy, clipon);
    setfillstyle(SolidFill, Blue);
    bar(0, 0, getmaxx, 10);
    SetColor(white);
    outtext('Generation: ');
    OutTextXY(250, 0, 'Q: Quit');
    OutTextXY(450, 0, 'Any other key: renew');
    str(gen_count, ss); outtext(ss);
    SetBkColor(DarkGray);
end;{rule_plane}

begin
    if gen_count <> 0 then
        next_generation;
```

```

inc(gen_count);
page := 1 - page;
setactivepage(page);
cleardevice;
setcolor(yellow);
for j := 0 to ver do
  for k := 0 to hor do
    if new_gen[j, k] = 1 then
      circle(x_center[k], y_center[j], radius);
rule_plane;
setvisualpage(page);
end:{display}

begin
  init_screen;
  repeat
    randomize;
    prob := 0.1 + prob_factor * random;
    outtextxy(0, 0, 'Conway''s Game of Life');
    writeln;
    outtextxy(0, 15, 'Live cells inserted at random.');
    str(prob:3:3, ss);
    outtextxy(0, 30, 'with probability ' + ss);
    outtextxy(0, 60, 'Press any key to start: ');
    ch := readkey;
    cleardevice;
    init_cells;

    repeat
      display;
      if keypressed then
        begin
          ch := readkey;
          break;
        end;
    until false;

    setviewport(0, 0, getmaxx, getmaxy, clipon);
    cleardevice;
    setcolor(white);
    if upcase(ch) = 'Q' then
      break;
    until false;
    CloseGraph;
end.

```

Процедура `init_cells` формирует исходную популяцию, присваивая элементам массива `new_gen` значения 0 или 1 с учетом выбранного значения вероятности заселения.

Процедура `next_generation` строит в массиве `new_gen` следующую популяцию в соответствии с правилами игры.

В процедуре `init_screen` инициализируется графический режим и заполняются массивы графических координат центров ячеек `x_center` и `y_center`.

Процедура `display` выполняет основную работу по построению изображения рабочего поля игры на скрытой активной странице и выводу ее на экран. Это обеспечивают процедуры `SetActivePage` и `SetVisualPage`, вызываемые с параметром 0 или 1, определяющим номер графической страницы. Первая процедура при вызове устанавливает активную страницу, а вторая — визуальную.

## Принципы программирования графики

У нас уже есть опыт программирования графических изображений на Турбо Паскале. Этот опыт позволяет создавать графические программы средней сложности. А вот чтобы стать чародеем графики, следует хорошо овладеть ее основами. Не могу обещать читателю быстрого превращения в чародея, но сейчас мы вернемся к основам и вновь рассмотрим некоторые важнейшие принципы графики. При этом мы часто будем работать с видеoadаптером на уровне базовой системы ввода/вывода (BIOS) и на аппаратном уровне.

Прежде всего следует понимать, что графические возможности персонального компьютера ограничены. Это связано с тем, что непрерывное поле заменяется дискретной сеткой пикселов, и даже простые рисунки вроде окружностей, если на них смотреть через увеличительное стекло, имеют весьма прерывистый вид. Однако несмотря на это, для решения разнообразных задач компьютерная графика просто неоценима. Ведь за миллионы лет биологической эволюции мозг человека выработал высокоэффективные алгоритмы обработки именно визуальной информации. Поэтому гораздо легче определить особенности поведения какой-нибудь зависимости по графику, чем по заданному в виде таблицы массиву чисел.

До сих пор для программирования графики мы использовали процедуры и функции, содержащиеся в модуле `Graph`. Работа этого модуля основана на использовании так называемого графического интерфейса Борланд (BGI — Borland Graphics Interface), специализированной графической библиотеки. Основное достоинство этого интерфейса заключается в том, что он прост в изучении и вполне подходит для программирования несложных графических программ. Недостатками программирования с использованием BGI-интерфейса являются сравнительно низкое быстродействие графических программ и его ограниченные возможности. Если учсть, что роль графического вывода в современных программах велика, понятным оказывается то, что следующим шагом в программировании графики после использования собственных процедур Турбо Паскаля является работа с видеоподсистемой компьютера «напрямую».

## Инициализация графического режима. Пиксели

Итак, вернемся к проблеме инициализации графического режима. Самый простой графический режим поддерживает работу с 256 цветами, линейную модель видеопамяти и разрешение 320×200. Чтобы включить этот режим, надо воспользоваться прерыванием \$10 и функцией \$00 «установка режима». При этом в регистр AL должно быть записано значение \$13. Процедура `init_Graph` программы `pixel` (листинг 5.6) инициализирует данный графический режим именно таким образом. При инициализации происходит очистка экрана. Завершение работы с графическим экраном и возвращение в текстовый режим выполняется с помощью функции \$00 того же прерывания при записи в регистр AL значения \$03 (процедура `close_Graph`).

При разрешении 320×200 размер буфера экрана должен быть  $320 \times 200 \times 1 = 64\,000$  байт, так как на каждый пиксель отводится 1 байт, содержащий информацию о цвете данного пикселя. Количество возможных цветов составляет 256, поэтому одного байта вполне достаточно. Ранее уже говорилось о том, что используется линейная модель видеопамяти. Это значит, что относительный (отсчитываемый от начала области видеопамяти) адрес первого пикселя равен 0, второго пикселя — 1 и т. д. вплоть до 63 999 (табл. 5.9).

**Таблица 5.9.** Адресация пикселов изображения при линейной модели видеопамяти

	x = 0	x = 1	...	...	x = 318	x = 319
y = 0	0	1	...	...	318	319
y = 1	320	321	...	...	638	639
...	...	...	...	...	...	...
y = 199	63681	63682	...	...	63998	63999

Все эти адреса определены относительно абсолютного сегмента памяти \$A000, то есть адрес первого пикселя равен \$A000:0000, второго — \$A000:0001 и т. д. Адрес пикселя с графическими координатами  $x$  и  $y$  можно определить по формуле

$$\text{Address} = 320 \times y + x.$$

Процедура `set_pixel` программы `pixel` (листинг 5.15) устанавливает цвет пикселя с заданными графическими координатами, занося номер цвета в соответствующий элемент массива экранного буфера. Координаты пикселов и их цвета выбираются случайным образом. Выполнение программы прекращается при нажатии клавиши `Enter`.

### Листинг 5.15. Программа `pixel`

```
program pixel;
uses crt, dos;
type
```

```

screenbuffertype = array[0..63999] of byte;
screenbufferptr = ^screenbuffertype;

var
  screen : screenbufferptr;
  hres, vres : word;
  color : byte;

procedure init_Graph;
var
  regs : registers;
begin
  regs.ah := $00;
  regs.al := $13;
  intr($10, regs);
  screen := ptr($a000, 0);
end;

procedure close_Graph;
var
  regs : registers;
begin
  regs.ah := $00;
  regs.al := $03;
  intr($10, regs);
end;

procedure set_pixel(x, y : word; color : byte);
var
  address : word;
begin
  address := y * 320 + x;
  screen^[address] := color;
end;

begin
  hres := 320; vres := 200; color := 255;
  randomize;
  init_Graph;
  repeat
    set_pixel(random(hres), random(vres), random(color+1));
  until keypressed;
  close_Graph;
end.

```

Процедура `set_pixel` имеет один недостаток — низкое быстродействие, которое объясняется использованием арифметического умножения. Умножение является одной из наиболее трудоемких арифметических операций, включающей в общем случае несколько десятков машинных команд. Эффективность процедуры `set_pixel` можно увеличить, например, заменив арифметическое умножение операциями сдвига:

Address := (Y shl 8) + (Y shl 6) + X;

Такая замена сокращает время, затрачиваемое на вычисление адреса, почти в два раза. Следующим шагом по усовершенствованию процедуры вывода пикселя на экран могло бы послужить предварительное вычисление всех возможных значений произведения  $320 \times y$  и сохранение их в одномерном массиве, проиндексированном значениями  $y$ .

## Отрезки прямых

Отрезок прямой линии — одна из простейших геометрических фигур, но несмотря на это вывод отрезка в графическом режиме представляет собой достаточно сложную задачу. Начнем с простого — вывода горизонтальных и вертикальных отрезков. Отрезок можно было бы строить поточечно, воспользовавшись процедурой вывода пикселя и циклом `for`. Это незэффективное решение. Мы воспользуемся линейной моделью видеопамяти. Очевидно, что адреса графических точек, составляющих отрезок горизонтальной прямой, изменяются с шагом, равным единице, от некоторого минимального до максимального значения. Минимальное и максимальное значения определяют положение и длину горизонтального отрезка. Процедура `hline` программы `line` (листинг 5.16) использует эту идею. Здесь, кроме того, применяется заранее подготовленная таблица адресов пикселов.

Отображение вертикальной линии — более сложная задача, так как последовательные точки вертикального отрезка уже не располагаются последовательно в видеопамяти. Шаг изменения значения адреса в этом случае равен ширине экрана в пикселях. Процедура `vline` выводит вертикальный отрезок.

**Листинг 5.16.** Вывод отрезков прямых линий

```
program line;
uses crt, dos;

type
  screenbuffertype = array[0..63999] of byte;
  screenbufferptr = ^screenbuffertype;
  screentype = record
    buffer : screenbufferptr;
    ytable : array[0..199] of word;
    width : word;
    color : word;
  end;

var
  screen : screentype;
  screen_y : array[0..199] of word;
  i, j : word;

procedure calcscreeny(width : word);
var
```

```
i : integer;
begin
    for i := 0 to 199 do
        screen_y[i] := i * width;
end;

procedure init_Graph;
var
    regs : registers;
begin
    regs.ah := $00;
    regs.al := $13;
    intr($10, regs);
    screen.buffer := ptr($a000, 0);
    screen.width := 320;
    calcscreeny(320);
end;

procedure close_Graph;
var
    regs : registers;
begin
    regs.ah := $00;
    regs.al := $03;
    intr($10, regs);
end;

procedure hline(x1, y, x2 : word; color : byte);
var
    x, address : word;
begin
    address := screeny[y];
    for x := x1 to x2 do
        screen.buffer^[address+x] := color;
end;

procedure vline(x, y1, y2 : word; color : byte);
var
    y, address : word;
begin
    address := screen_y[y1]+x;
    for y := y1 to y2 do
    begin
        screen.buffer^[address] := color;
        address := address + screen.width;
    end;
end;

begin
    init_Graph;
    for i := 0 to 199 do
        hline(0, i, 319, i mod 255);
```

```

readln;
for i := 0 to 319 do
  vline(i, 0, 199, i mod 255);
readln;
close_Graph;
end.

```

Следующий шаг — построение наклонных линий. Имеется несколько алгоритмов построения отрезков наклонных линий. Прежде всего вспомним, что уравнение произвольной прямой имеет вид

$$y = m x + b,$$

где  $m$  — тангенс угла наклона прямой к горизонтальной оси (в дальнейшем просто «наклон»), а  $b$  — точка пересечения прямой с осью  $y$  прямоугольных координат. Это уравнение нам удобно переписать в другом виде:

$$y = m (x - x_1) + y_1.$$

В этом случае, задав значение  $x$ , которое должно находиться между  $x_1$  и  $x_2 = x_1$ , получим координаты пикселя  $(x, y)$ , лежащего на отрезке наклонной прямой. Наклон прямой можно найти, если заданы координаты точек начала и конца отрезка. Непосредственная реализация такого алгоритма неэффективна, так как при этом приходится использовать операции с вещественными числами, требующие больших затрат процессорного времени. Неплохим было бы решение, основанное на целочисленной арифметике, так как операции с целыми числами являются самыми быстрыми.

Одним из алгоритмов такого рода является алгоритм Брезенхама, предложенный в середине шестидесятых годов. Этот алгоритм реализован в процедуре `bresenhamline`, которая используется в программе `bresenline` (листинг 5.17). Исходный текст опущенных фрагментов (процедуры `calcscreeny`, `init_Graph`, `close_Graph` и `set_pixel`) был приведен в листингах 5.15 и 5.16 (программы `pixel` и `line`).

#### Листинг 5.17. Использование алгоритма Брезенхама

```

program bresenline;

procedure calcscreeny(width : word);
...
procedure init_Graph;
...
procedure close_Graph;
...
procedure set_pixel(x, y : word; color : byte);
...

procedure bresenhamline(x1, y1, x2, y2 : word; color : byte);
var
  deltax, deltay, incl1, inc2, d, x, y : integer;

```

```

begin
    deltax := x2 - x1;
    deltay := y2 - y1;
    d := 2 * deltay - deltax;
    inc1 := deltay shl 1;
    inc2 := (deltay - deltax) shl 1;
    x := x1; y := y1;
    set_pixel(x, y, color);
    while x < x2 do
    begin
        if d < x1 then
        begin
            d := d + inc1;
            inc(x);
        end
        else
        begin
            d := d + inc2;
            inc(x);
            inc(y);
        end;
        set_pixel(x, y, color);
    end;
end;

begin
    init_Graph;
    bresenhamline(0, 0, 319, 199, 15);
    readln;
    close_Graph;
end.

```

Рассмотрим алгоритм вывода отрезка прямой более подробно. Итак, известны координаты двух точек — начала отрезка  $P_1 = (x_1, y_1)$  и его конца  $P_2 = (x_2, y_2)$ . Чтобы построить изображение отрезка, необходимо найти координаты всех принадлежащих ему пикселов. Все осложняется тем, что пиксель не может иметь произвольные значения координат. Можно считать, что прямоугольные пиксели образуют прямоугольную решетку. Следовательно, математическая задача заключается в том, чтобы найти те ячейки этой решетки, которые расположены на отрезке или в непосредственной близости от него (прямоугольники серого цвета на рис. 5.4).

Таких пикселов может быть очень много, и применение целочисленной арифметики при вычислении их координат является единственно приемлемым решением. Будем считать, что выполняются следующие условия:

$$\begin{aligned}x_1 &< x_2, \\y_1 &< y_2, \\y_2 - y_1 &< x_2 - x_1.\end{aligned}$$

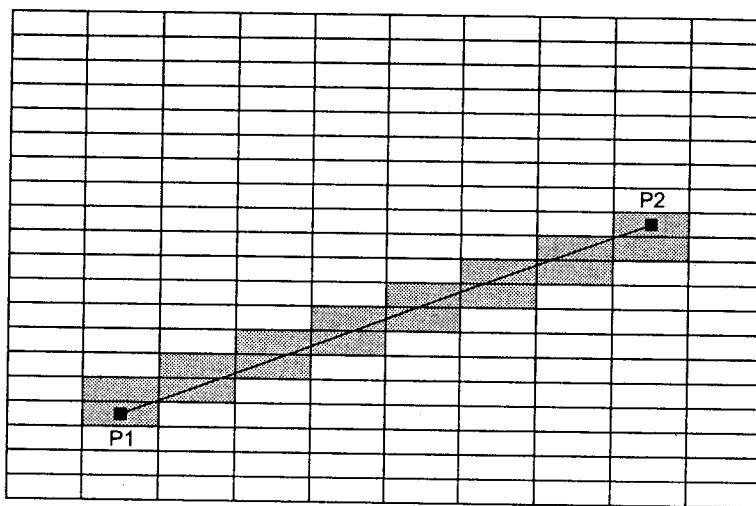


Рис. 5.4. Графический образ отрезка прямой линии

В этом случае необходимо организовать цикл по переменной  $x$ , изменяющейся от  $x_1$  до  $x_2$  с шагом 1. В общем случае (то есть в случае произвольного наклона отрезка) в качестве независимой управляющей переменной цикла следует выбрать ту координату ( $x$  или  $y$ ), которая соответствует наибольшему значению из  $|x_2 - x_1|$  и  $|y_2 - y_1|$ . Пусть такой переменной выбрана  $x$ . В цикле для каждого значения  $x$  определяются пиксели, находящиеся на минимальном расстоянии от отрезка. Вначале положим  $x = x_1$  и  $y = y_1$ . В цикле будем задавать для переменной  $x$  приращение 1, а  $y$  будем либо оставлять неизменным, либо увеличивать на единицу. Выбор осуществляется, как уже упоминалось, таким образом, чтобы новая ячейка сетки ( $x, y$ ) располагалась на минимальном расстоянии от прямой линии, идущей из точки  $P_1$  в точку  $P_2$ . Это означает, что расстояние по вертикали между новой выбранной точкой и отрезком не должно превышать 0,5.

Введем переменную для обозначения этого расстояния:

$$\Delta = y_{\text{exact}} - y_{\text{pixel}}$$

и потребуем, чтобы выполнялось соотношение

$$|\Delta| < 0,5.$$

Это значение вначале устанавливается равным нулю. Затем оно пересчитывается на каждом шаге цикла. Отклонение  $\Delta$  показывает, насколько ниже точной линии находится вычисленная точка. При увеличении значения переменной  $x$  на единицу при неизменном значении  $y$  отклонение увеличивается на значение наклона отрезка. Если окажется, что значение отклонения превышает 0,5, величину  $y$  следует увеличить на единицу с одновременным уменьшением значения отклонения на единицу.

В только что описанном методе подразумевалось использование арифметики с плавающей точкой, мы же решили, что алгоритм должен быть сформулирован в

целочисленной арифметике. Вещественный тип имеют переменные отклонения и наклона:

$$t = \frac{y_2 - y_1}{x_2 - x_1}.$$

В этой дроби числитель и знаменатель — целые числа, а сама дробь является рациональной. Величина отклонения  $\Delta$  вычисляется как сумма элементов, каждый из которых равен либо  $t$ , либо  $-1$ , поэтому отклонение также может быть записано в виде рациональной дроби со знаменателем  $(x_2 - x_1)$ . К целочисленным значениям можно перейти, умножив значения  $\Delta$  и  $t$  на величину  $s$ :

$$\begin{aligned}s &= 2(x_2 - x_1), \\ T &= s t, \\ D &= s \Delta.\end{aligned}$$

Коэффициент 2 используется в первой формуле для того, чтобы избавиться от нецелочисленной константы 0,5. При построении эффективного алгоритма рисования отрезка прямой учитывается еще одно обстоятельство. Оно заключается в том, что сравнение с нулем выполняется быстрее, чем с ненулевым вещественным значением. Поэтому вводится переменная

$$E = D - s / 2.$$

Для нее условие  $\Delta < s / 2$  заменяется условием  $E < 0$ . Начальное значение  $E = -s / 2$ .

В программе `segment_demo` (листинг 5.18) демонстрируется применение алгоритма Брезенхама для построения отрезка прямой линии в общем случае. Некоторые процедуры, используемые здесь, нам уже знакомы по другим программам этого урока, поэтому их исходный текст здесь опущен. Основной процедурой программы является процедура `segment`.

**Листинг 5.18.** Использование алгоритма Брезенхама для построения отрезка прямой линии

```
program segment_demo;

uses crt, dos;
type
  screenbuffertype = array[0..63999] of byte;
  screenbufferptr = ^screenbuffertype;
  screentype = record
    buffer : screenbufferptr;
    ytable : array[0..199] of word;
    width : word;
    color : word;
  end;

var
  screen : screentype;
```

```
screen_y : array[0..199] of word;
i, j : word;

procedure calcscreeny(width : word);
...
procedure init_Graph;
...
procedure close_Graph;
...
procedure hline(x1, y, x2 : word; color : byte);
...
procedure vline(x, y1, y2 : word; color : byte);
...
procedure set_pixel(x, y : word; color : byte);
...
procedure segment(x0, y0, x1, y1 : integer);
var
  dx, two_dx, dy, two_dy, abs_slope,
  Error, dError, eps, sum, t : integer;

procedure bresenham_big_y;
begin
  if x1 < x0 then
    begin
      t := x0;
      x0 := x1;
      x1 := t;
      t := y0;
      y0 := y1;
      y1 := t;
    end;
  if y0 < y1 then
    eps := 1
  else
    eps := -1;
  Error := -dy;
  while abs(y1 - y0) >= 2 do
    begin
      set_pixel(x0, y0, white);
      set_pixel(x1, y1, white);
      Error := Error + two_dx;
      if Error >= 0 then
        begin
          Error := Error - two_dy;
          inc(x0); dec(x1);
        end;
    end;
end;
```

```
    y0 := y0 + eps;
    y1 := y1 - eps;
end;
if y0 = y1 then
    set_pixel(x0, y0, white)
else
begin
    set_pixel(x0, y0, white);
    set_pixel(x1, y1, white);
end;
end:{bresenham_big_y}

procedure bresenham_big_x;
begin
    if y1 < y0 then
begin
    t := x0;
    x0 := x1;
    x1 := t;
    t := y0;
    y0 := y1;
    y1 := t;
end;
    if x0 < x1 then
        eps := 1
    else
        eps := -1;
    Error := -dx;
    while abs(x1 - x0) >= 2 do
begin
    set_pixel(x0, y0, white);
    set_pixel(x1, y1, white);
    Error := Error + two_dy;
    if Error >= 0 then
begin
        Error := Error - two_dx;
        inc(y0); dec(y1);
end;
    x0 := x0 + eps;
    x1 := x1 - eps;
end;
    if x0 = x1 then
        set_pixel(x0, y0, white)
    else
begin
        set_pixel(x0, y0, white);
        set_pixel(x1, y1, white);
end;
end:{bresenham_big_x}

begin
    if x0 = x1 then
```

```

    vline(x0, y0, y1, white)
else
  if y0 = y1 then
    hline(x0, x1, y0, white)
  else
begin
  dx := abs(x1 - x0);
  dy := abs(y1 - y0);
  two_dx := 2 * dx;
  two_dy := 2 * dy;
  if dx <= dy then
    bresenham_big_y
  else
    bresenham_big_x;
end;
end;{segment}

begin
  init_Graph;
  segment(10, 180, 300, 30);
  readln;
  close_Graph;
end.

```

В процедуре `segment` в зависимости от наклона отрезка применяются различные варианты алгоритма Брезенхама. В общем случае можно использовать две стратегии. При наклоне от 0 до 1 отрезок строится процедурой `bresenham_big_x`, а при наклоне от 1 и больше — процедурой `bresenham_big_y`. В первом случае пиксели перебираются в цикле по координате  $x$ , а во втором — по  $y$ .

На проверки в процедуре `segment` затрачивается основное время. Чтобы минимизировать число проверок, алгоритм применяется симметрично с обоих концов отрезка со смещением к его середине.

## Отсечение линий

До сих пор мы рассматривали ситуации, когда начало и конец отрезка прямой линии находились в области графического вывода. Как быть, если это не так? Решением данной проблемы мы сейчас и займемся.

При отображении графических объектов обычно задается прямоугольная область на экране, которая и определяет размеры изображения. Эта область называется *областью вывода* и задается минимальными и максимальными значениями координат:

```

type
  RectType = record
    x_min, y_min, x_max, y_max : integer;
  end;

```

На рис. 5.5 показаны прямоугольная область вывода  $ABCD$  и отрезок прямой  $PQ$ , который начинается и оканчивается вне области вывода.

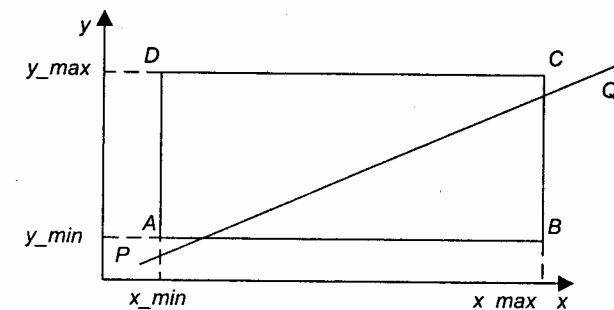


Рис. 5.5. Общий случай вывода изображения прямолинейного отрезка

Одним из алгоритмов отсечения линий является алгоритм Коэна—Сазерленда. Любой точке сопоставляется четырехбитовый код  $b_3 b_2 b_1 b_0$ . Этот код содержит информацию о положении точки относительно прямоугольной области вывода. Значения битов определяются следующим образом:

$$\begin{aligned} b_3 &= (x < x_{\min}), \\ b_2 &= (x > x_{\max}), \\ b_1 &= (y < y_{\min}), \\ b_0 &= (y > y_{\max}). \end{aligned}$$

Значение равно единице, если условие выполнено, и нулю, если не выполнено. Из 16 возможных битовых комбинаций допустимы только 9, и они показаны на рис. 5.6.

1001		0001		0101
	D		C	
1000		0000		0100
	A		B	
1010		0010		0110

Рис. 5.6. Схема кодирования в алгоритме Коэна—Сазерленда

Алгоритм Коэна—Сазерленда основан на анализе кодов точек отрезка. Вначале анализ выполняется для начальной и конечной точек отрезка. Если хотя бы один

из кодов содержит единичный бит, конечная (начальная) точка перемещается к одной из границ окна или его продолжению. Для определения координат новой точки используются формулы линейной интерполяции

$$x = x_1 + (y - y_1) \frac{x_2 - x_1}{Y_2 - Y_1},$$

$$y = y_1 + (x - x_1) \frac{y_2 - y_1}{x_2 - x_1}.$$

Реализация алгоритма Коэна—Сазерленда дана в программе `lineclipping` (листинг 5.19).

#### Листинг 5.19. Использование алгоритма Коэна—Сазерленда

```
program lineclipping;
uses crt, dos;

type
  recttype = record
    x_min, y_min, x_max, y_max : integer;
  end;
  screenbuffertype = array[0..63999] of byte;
  screenbufferptr = ^screenbuffertype;
  screentype = record
    buffer : screenbufferptr;
    dbuffer : boolean;
    ytable : array[0..199] of word;
    width : word;
    clip : recttype;
  end;

var
  x1, y1, x2, y2, tmp : integer;
  screen : screentype;
  screen_y : array[0..199] of word;
  i, j : word;

procedure calcscreeny(width : word);
...
procedure init_Graph;
...
procedure close_Graph;
...
procedure setclipboundary(x1, y1, x2, y2 : integer);
begin
  screen.clip.x_min := x1;
```

```
screen.clip.y_min := y1;
screen.clip.x_max := x2;
screen.clip.y_max := y2;
end;

procedure hline(x1, y, x2 : word; color : byte);
...

procedure vline(x, y1, y2 : word; color : byte);
...

procedure set_pixel(x, y : word; color : byte);
...

procedure bresenhamline(x1, y1, x2, y2 : word; color : byte);
...

procedure linec(x1, y1, x2, y2 : integer; color : byte);

const
  codebottom = 1;
  codetop    = 2;
  codeleft   = 4;
  coderight  = 8;

function compute_code(x, y : integer) : byte;
var
  code : byte;
begin
  code := 0;
  if y > screen.clip.y_max then
    code := codebottom
  else
    if y < screen.clip.y_min then
      code := codetop;
    if x > screen.clip.x_max then
      code := code + coderight
    else
      if x < screen.clip.x_min then
        code := code + codeleft;
  compute_code := code;
end:{compute_code}

var
  outcode0, outcode1, outcodeout : byte;
  x, y : integer;
begin
  outcode0 := compute_code(x1, y1);
  outcode1 := compute_code(x2, y2);
  while (outcode0 <> 0) or (outcode1 <> 0) do
  begin
    if (outcode0 and outcode1) <> 0 then
```

```

    exit
else
begin
  if outcode0 > 0 then
    outcodeout := outcode0
  else
    outcodeout := outcode1;
  if (outcodeout and codebottom) = codebottom then
begin
  y := screen.clip.y_max;
  x := x1 + longint(x2 - x1) * longint(y - y1) div (y2 - y1);
end
else
  if (outcodeout and codetop) = codetop then
begin
  y := screen.clip.y_min;
  x := x1 + longint(x2 - x1) * longint(y - y1) div (y2 - y1);
end
else
  if (outcodeout and coderight) = coderight then
begin
  x := screen.clip.x_max;
  y := y1 + longint(y2 - y1) * longint(x - x1) div (x2 - x1);
end
else
  if (outcodeout and codeleft) = codeleft then
begin
  x := screen.clip.x_min;
  y := y1 + longint(y2 - y1) *
    longint(x - x1) div (x2 - x1);
end;
if (outcodeout = outcode0) then
begin
  x1 := x; y1 := y;
  outcode0 := compute_code(x1, y1);
end
else
begin
  x2 := x; y2 := y;
  outcode1 := compute_code(x2, y2);
end;
end;
end;
line(x1, y1, x2, y2, color);
end:{linec}

begin
  randomize;
  init_Graph;
  setClipboundary(30, 30, 190, 150);
  vline(30, 30, 150, 7);
  vline(190, 30, 150, 7);

```

```

hline(30, 30, 190, 7);
hline(30, 150, 190, 7);
repeat
  x1 := random(320);
  y1 := random(200);
  x2 := random(320);
  y2 := random(200);
  if x1 > x2 then
    begin
      tmp := x2; x2 := x1; x1 := tmp
    end;
  if y1 > y2 then
    begin
      tmp := y2; y2 := y1; y1 := tmp
    end;
  linec(x1, y1, x2, y2, 7);
until readkey = #27;
close_Graph;
end.

```

## Окружность

Рисование окружности с центром в точке  $(x_0, y_0)$  и радиусом  $r$  является другой фундаментальной проблемой компьютерной графики. Есть несколько способов рисования окружности, но наиболее эффективным из них является алгоритм Брезенхама. Чтобы уменьшить число проверок, вычисления проводятся для одной восьмой окружности, а для построения остальной части используется симметрия. Для того чтобы пояснить идею алгоритма Брезенхама, предположим, что  $x_0 = 0$ ,  $y_0 = 0$ , а точка  $(x, y)$  принадлежит первой «восьмушке» (нумерация идет от оси абсцисс против часовой стрелки). Тогда следующей точкой образа окружности будет ближайшая к ней точка из двух —  $(x, y + 1)$  или  $(x - 1, y + 1)$ . В первом случае величина

$$s = [x^2 + (y + 1)^2 - r^2] + [(x - 1)^2 + (y + 1)^2 - r^2]$$

отрицательна, а во втором положительна. Определив знак этой величины, определим таким образом и координаты очередного пикселя.

Процедура `circle` в программе `circle_demo` (листинг 5.20) предназначена только для рисования полной окружности, находящейся целиком в области рисования.

**Листинг 5.20.** Использование алгоритма Брезенхама для построения окружности

```
program circle_demo;
```

```
uses crt, dos;
```

```
const
```

```
no_rows = 200;
```

```
no_cols = 320;
```

```
x_max = no_cols div 2;
```

```

x_min = - x_max + 1;
y_max = no_rows div 2;
y_min = - y_max + 1;

type
  screenbuffertype = array[0..63999] of byte;
  screenbufferptr = ^screenbuffertype;
  screentype = record
    buffer : screenbufferptr;
    ytable : array[0..199] of word;
    width : word;
    color : word;
  end;

var
  screen : screentype;
  screen_y : array[0..199] of word;
  i, j : word;

procedure calcscreeny(width : word);
...
procedure init_Graph;
...
procedure close_Graph;
...
procedure set_pixel(x, y : word; color : byte);
...
procedure circle(x0, y0, r : integer);
var
  x, y, s : integer;
  xx, yy : array[1..8] of integer;
  k : word;
begin
  if (x0 + r > x_max) or (x0 - r < x_min) or
    (y0 + r > y_max) or (y0 - r < y_min) then
    exit;
  set_pixel(x0 + r, y0, white);
  set_pixel(x0 - r, y0, white);
  set_pixel(x0, y0 + r, white);
  set_pixel(x0, y0 - r, white);
  x := r;
  y := 0;
  s := 3 - 2 * r;
  for k := 1 to 4 do
    begin
      yy[k] := y0;
      xx[4 + k] := x0;
      if s <= 0 then
        begin
          yy[4 + k] := yy[k] + 1;
          xx[4 + k] := xx[k] - 1;
          s := s + 2 * yy[k] + 1;
        end
      else
        begin
          yy[4 + k] := yy[k] + 1;
          xx[4 + k] := xx[k];
          s := s + 2 * yy[k] + 2 * xx[k] + 1;
        end
      end;
end;

```

```

xx[1] := x0 + r;
xx[2] := xx[1];
xx[3] := x0 - r;
xx[4] := xx[3];
yy[5] := y0 + r;
yy[6] := y0 - r;
yy[7] := yy[5];
yy[8] := yy[6];
while y < x - 1 do
begin
  inc(yy[1]);
  yy[3] := yy[1];
  dec(yy[2]);
  yy[4] := yy[2];
  inc(xx[5]);
  xx[6] := xx[5];
  dec(xx[7]);
  xx[8] := xx[7];
  if s <= 0 then
    s := s + 4 * y + 6
  else
    begin
      s := s + 4 * (y - x) + 10; dec(x);
      dec(xx[1]); xx[2] := xx[1];
      inc(xx[3]); xx[4] := xx[3];
      dec(yy[5]); yy[7] := yy[5];
      inc(yy[6]); yy[8] := yy[6];
    end;
  inc(y);
  if x = y then
    for k := 1 to 4 do
      set_pixel(xx[k], yy[k], white)
  else
    for k := 1 to 8 do
      set_pixel(xx[k], yy[k], white);
  end;
end;

begin
  init_Graph;
  circle(140, 60, 20);
  'readln';
  close_Graph;
end.

```

В заключение замечу, что в большинстве случаев для вывода основных графических элементов в программах можно ограничиться использованием процедур модуля Graph. Иногда работа этих процедур может показаться программисту неудовлетворительной, например, по быстродействию. Иногда возникают нестандартные задачи компьютерной графики. В этих случаях приходится заниматься разработкой и программированием основных графических алгоритмов, что тре-

бует более глубокого понимания принципов отображения графических примитивов.

#### Упражнение 5.9

Разберите работу программы `segment_demo` и `circle_demo`.

#### Упражнение 5.10

Напишите процедуру рисования эллипса.

#### Упражнение 5.11

Напишите процедуру рисования параболы.

#### Упражнение 5.12

Напишите процедуру рисования спирали.

## Использование модуля `mouse` для программирования мыши в графическом режиме

Рассмотрим пример программирования мыши в графическом режиме. Программа `Graph_mouse` (листинг 5.21) использует модули `Graphs` из данного урока и `mouse` из четвертого урока. Обратите особое внимание на типизированную константу — двумерный массив `masks`, который определяет форму курсора мыши в графическом режиме. В нашем случае это таинственный указующий перст (см. комментарий в начале программы).

**Листинг 5.21.** Работа с мышью в графическом режиме

```
program Graph_mouse;
uses
  crt, Graph, Graphs, mouse;

const
  x_min = 220;
  x_max = 420;
  y_min = 290;
  y_max = 390;
  masks : array[0..1, 0..15] of word =
    ($E1FF, $EDFF, $EDFF, $EDFF,
     $EDFF, $EC00, $EDB6, $EDB6,
     $0DB6, $6FFE, $6FFE, $6FFE,
     $7FFE, $7FFE, $7FFE, $0000),
    ($1E00, $1200, $1200, $1200,
```

```

$1200, $13FF, $1249, $1249,
$F249, $9001, $9001, $9001,
$8001, $8001, $8001, $FFFF) );
{ Маска экрана:      Маска курсора:      Изображение:
1110000111111111  0001111000000000      XXXX
1110110111111111  0001001000000000      X  X
1110110000000000  0001000111111111      X  XXXXXXXXXXXX
1110110110110110  0001001001001001      X  X  X  X  X
1110110110110110  0001001001001001      X  X  X  X  X
0000110110110110  1111001001001001      XXXX  X  X  X  X
0110111111111110  1001000000000001      X  X      X
0110111111111110  1001000000000001      X  X      X
0110111111111110  1001000000000001      X  X      X
0111111111111110  1000000000000001      X      X
0111111111111110  1000000000000001      X      X
0111111111111110  1000000000000001      X      X
0000000000000000  1111111111111111      XXXXXXXXXXXXXXXX }
var
  ss, tt : string;
  not_mouse : boolean;
  x, y, k, count : word;
  button : byte;

begin
  gotoxy(1, 6);
  write('Нажмите обе кнопки мыши');
  repeat
    button := 2;
    get_mouse_button_press(button, count, x, y);
  until button = 3;
  open_Graph;
  setcolor(yellow);
  setbkcolor(darkgray);
  setfillstyle(solidfill, red);
  outtextxy(0, 0, 'Testing Graphics mouse. Move around!');
  reset_mouse(not_mouse, button);
  bar(x_min, y_min, x_max, y_max);
  show_cursor;
  outtextxy(0, 20, 'Click left to hide cursor');
  repeat
    button := 0;
    get_mouse_button_press(button, count, x, y);
  until button = 1;
  hide_cursor;
  outtextxy(0, 40, 'Click right for the finger; move it into the red!');
  repeat
    button := 1;
    get_mouse_button_press(button, count, x, y);
  until button = 3;
end.

```

```

until button = 2;
set_Graph_cursor_shape(4, 0, masks);
show_cursor;
repeat
    get_mouse_status(button, x, y);
    until (x >= x_min) and (x <= x_max) and (y >= y_min) and (y <= y_max);
    set_hide_cursor_window(x_min, y_min, x_max, y_max);
    outtextxy(0, 60, 'Click left: restrict and show cursor');
repeat
    button := 0;
    get_mouse_button_press(button, count, x, y);
until button = 1;
set_cursor_x_lim(x_min, x_max);
set_cursor_y_lim(y_min, y_max);
outtextxy(0, 80, 'Try to move the cursor out of red field');
show_cursor;
outtextxy(0, 100, 'Double click left to quit:');
repeat until _double_click(500);
close_Graph;
reset_mouse(not_mouse, button);
clrscr;
end.

```

Чтобы разобраться с работой этой программы, придется вернуться к материалу четвертого урока, обращая особое внимание на процедуры, предназначенные для работы в графическом режиме. Это прежде всего процедура `set_Graph_cursor_shape`, устанавливающая форму курсора и положение его активной зоны (у нас это указательный палец изображения курсора).

Как на экран выводится курсор мыши? Сначала к образу, находящемуся на экране в области курсора размером 16×16 пикселов, и экранной маске применяется операция `and`, затем к результату и маске курсора применяется операция `xor`. Применение операции `and` изменит цвет границы руки на черный, а все остальное останется неизменным. Маска курсора противоположна экранной маске в том смысле, что значения битов 1 и 0 меняются местами, поэтому применение операции `xor` с маской курсора изменит цвет границы руки на белый.

Графический курсор, используемый по умолчанию, представляет собой стрелку — это стандартная форма, например, для MS Windows. К использованию мыши в графическом режиме мы еще вернемся в уроке, посвященном программированию звука. Мышь и графический вывод там будут использованы в симпатичной демонстрационной программе.

#### Упражнение 5.13

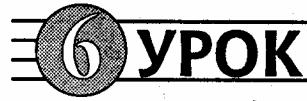
Добавьте в программу `Graph_mouse` фрагмент, в котором движение курсора мыши ограничено окружностью.

#### Упражнение 5.14

Напишите программу, в которой курсор мыши играет роль карандаша (или кисти), то есть при перемещении с нажатой левой кнопкой на экране оставляет след в виде белой линии.

## Что нового мы узнали?

- Познакомились с основными принципами работы в графическом режиме.
- Познакомились с основными процедурами графического модуля Graph.
- Рассмотрели процесс инициализации графического режима.
- Рассмотрели алгоритмы построения основных графических элементов.
- Рассмотрели пример программирования мыши в графическом режиме.



## **6 УРОК**

# **Графика VGA, программирование трехмерных и динамических изображений**

- 
- Технические подробности
  - Графика VGA
  - Программирование трехмерных изображений
  - Программирование динамических изображений
-

**В** этом уроке мы продолжим изучение приемов программирования графического режима. Основными темами будут: использование возможностей видеoadаптера VGA, программирование трехмерных и динамических изображений.

## Технические подробности

Чтобы успешно программировать в графическом режиме, полезно иметь представление о том, как формируется изображение на экране дисплея. Видеоподсистему персонального компьютера составляют видеокарта и дисплей. Обычный дисплей устроен и работает примерно так же, как и обычный телевизор. Изображение формируется электронным лучом, который пробегает по внутренней поверхности экрана, имеющей специальное покрытие. Покрытие светится под воздействием электронного луча. Благодаря изменению интенсивности подсветки, быстрому движению луча и некоторым техническим ухищрениям и создается цветное статическое (неподвижное) или динамическое (движущееся) изображение.

Основным функциональным узлом компьютера, обеспечивающим вывод на дисплей, является видеокарта. Типовая видеокарта состоит из четырех основных устройств: видеопамяти, видеоконтроллера, цифроаналогового преобразователя (ЦАП) и постоянного запоминающего устройства (ПЗУ).

*Видеопамять* служит для хранения изображения. От ее объема зависят «изобразительные» возможности видеокарты — максимально возможное разрешение и количество возможных цветов каждого пикселя. Для разрешения 640×480 при 16 цветах требуется 256 Кбайт видеопамяти, для разрешения 800×600 и 256 цветов — 512 Кбайт, для разрешения 1024×768 и 65 536 цветов (режим High Color, другое его обозначение — 64k цветов) требуется 2 мегабайта и т. д. Поскольку для хранения цветов отводится целое число разрядов, количество цветов всегда является степенью двойки.

*Видеоконтроллер* управляет выводом изображения из видеопамяти на экран дисплея, регенерацией (сохранением и обновлением) содержимого памяти, формированием сигналов развертки для монитора и обработкой запросов центрального процессора. Для исключения конфликтов при обращении к памяти со стороны видеоконтроллера и центрального процессора первый имеет отдельный буфер,

который в свободное от обращений центрального процессора время заполняется данными из видеопамяти. Если конфликта избежать не удается, то видеоконтроллеру приходится задерживать обращение процессора к видеопамяти, что снижает производительность системы в целом.

*Цифроаналоговый преобразователь* (ЦАП, DAC – Digital-Analog Converter) служит для преобразования потока данных, формируемого видеоконтроллером, в уровни интенсивности цветовых сигналов, подаваемых на монитор. Современные мониторы используют аналоговый (то есть непрерывно изменяющийся) видеосигнал, поэтому возможный диапазон цветности изображения определяется только параметрами ЦАП. Большинство ЦАП имеют разрядность  $8 \times 3$  – три канала для основных цветов (красного, синего и зеленого) по 256 уровней яркости на каждый цвет. Это обеспечивает поддержку до 16,7 миллионов цветов.

Видео ПЗУ содержит видео BIOS, экранные шрифты, служебные таблицы и т. д. К видео ПЗУ обращается только центральный процессор, и в результате выполнения им программ из ПЗУ происходят обращения к видеоконтроллеру и видеопамяти. ПЗУ необходимо только для первоначального запуска видеoadаптера и для работы в режиме MS-DOS.

В современных видеокартах используются дополнительные устройства, такие как, например, графические ускорители, которые позволяют ускорить вывод на экран сложных и реалистических изображений.

В персональных компьютерах используются следующие типы видеoadаптеров:

- **MDA** (Monochrome Display Adapter – монохромный адаптер дисплея) – простейший видеoadаптер, применявшийся в старых персональных компьютерах. Работает в текстовом режиме с разрешением  $80 \times 25$ , поддерживает пять атрибутов текста: обычный, яркий, инверсный, подчеркнутый и мигающий. Частота строчной развертки 15 кГц.
- **HGC** (Hercules Graphics Card – графическая карта Hercules) представляет собой расширение MDA, разработанное фирмой Hercules.
- **CGA** (Color Graphics Adapter – цветной графический адаптер). Это первый адаптер с графическими возможностями. Работает в текстовом режиме с разрешениями  $40 \times 25$  и  $80 \times 25$ , а также в графическом режиме с разрешением  $320 \times 200$  или  $640 \times 200$ . В текстовых режимах доступно 256 атрибутов символа – 16 цветов символа и 16 цветов фона. В графических режимах доступно четыре палитры по четыре цвета каждая в режиме  $320 \times 200$ . Графический режим с разрешением  $640 \times 200$  является монохромным. Частота строчной развертки 15 кГц.
- **EGA** (Enhanced Graphics Adapter – улучшенный графический адаптер) представляет собой дальнейшее развитие CGA. По сравнению с CGA добавлено разрешение  $640 \times 350$ . В текстовых режимах поддерживается разрешение  $80 \times 25$  и  $80 \times 43$ . Количество одновременно отображаемых цветов равно 16, однако палитра расширена до 64 цветов. Структура видеопамяти сделана на основе так называемых битовых *слоев*, или *плоскостей*, каждая из которых в графическом режиме содержит биты только своего цвета, а в текстовых ре-

жимах по плоскостям разделяются собственно текст и данные знакогенератора. Совместим с MDA и CGA. Частоты строчной развертки 15 и 18 кГц.

- **MCGA** (Multicolor Graphics Adapter — многоцветный графический адаптер) был введен фирмой IBM в ранних моделях персональных компьютеров PS/2 (сейчас эти компьютеры — уже музейная редкость). По сравнению с предыдущими моделями добавлено разрешение в текстовом режиме 80×25 и 80×50. Количество воспроизводимых цветов увеличено до 262 144 (по 64 уровня на каждый из основных цветов). Помимо палитры введено понятие *таблицы цветов*, через которую выполняется преобразование 64-цветного пространства цветов EGA в пространство цветов MCGA. Введен также видеорежим с разрешением 320×200 и 256 цветами, в котором вместо битовых плоскостей используется представление экрана непрерывной областью памяти объемом 64 000 байт, где каждый байт описывает цвет соответствующей ему точки экрана. Частота строчной развертки 31 кГц.
- **VGA** (Video Graphics Array — массив визуальной графики) — расширение MCGA, совместимое с EGA. Впервые введен фирмой IBM в средних моделях PS/2. Фактический стандарт видеоадаптера с конца 80-х годов. Добавлен графический режим 640×480 с доступом через битовые плоскости. В режиме 640×480 используется так называемая квадратная точка (соотношение количества точек по горизонтали и вертикали совпадает со стандартным соотношением сторон экрана, а именно 4:3).



#### **ВНИМАНИЕ**

Думаю, что здесь полезно напомнить некоторые правила безопасной работы с монитором. Работающий монитор является источником излучения. Это рентгеновское излучение, идущее из кинескопа, а также переменное электромагнитное поле, идущее от катушек строчной и кадровой развертки. Рентгеновское излучение поглощается на расстоянии в 20-30 сантиметров от электронно-лучевой трубы, а электромагнитное поле катушек распространяется во все стороны, особенно вбок и назад. В направлении вперед оно ослабляется теневой маской и арматурой кинескопа. Медики считают, что именно электромагнитное излучение низкой частоты представляет наибольшую опасность для здоровья, поэтому санитарные нормы определяют минимальное расстояние от экрана до оператора около 50-70 см, а ближайших рабочих мест от боковой и задней стенок монитора — не менее 1,5 м. Один из стандартов на допустимые уровни электромагнитных излучений — стандарт MPR II, устанавливающий безопасные уровни излучений на расстоянии 50 см от монитора, — этому стандарту удовлетворяют практически все современные мониторы. Более жесткий стандарт TCO 92 устанавливает безопасные уровни на расстоянии 30 см от монитора. В настоящее время действует стандарт TCO 95.

- **IBM 8514/а** — специализированный адаптер для работы с высокими разрешениями (640×480 и 256 цветов, а также 1024×768 и 256 цветов), имеющий элементы графического ускорителя. Не поддерживает видеорежимы VGA.

- **IBM XGA** – специализированный адаптер IBM. Расширено цветовое пространство (режим 640×480 и 64 к цветов), добавлен текстовый режим 132×25.
- **SVGA** (Super VGA) – расширение VGA с добавлением более высоких разрешений и дополнительных возможностей. Добавлены видеорежимы 800×600, 1024×768, 1152×864, 1280×1024, 1600×1200 (все, кроме режима 1280×1024 – с соотношением 4:3). Цветовое пространство расширено до 65 536 цветов (режим High Color) или 16,7 миллионов цветов (режим True Color). Добавлены текстовые режимы 132×25, 132×43, 132×50. Фактический стандарт видеоадаптера с начала 90-х годов.

## Графика VGA

Для работы в графическом режиме VGA мы воспользуемся прерыванием \$10 и некоторыми особенностями аппаратных средств VGA. Программа `demo_VGA` (листинг 6.1) выводит на графический экран любой текст, в том числе символы ASCII с кодами 1–31.

**Листинг 6.1.** Программа `demo_VGA`

```
program demo_VGA;
uses crt, graph, dos;
var
  GraphDriver, GraphMode: integer;
  k: word;
  x, y, attr, xl, yl, top, bottom, mode, cols, page: byte;
  ss, tt: string[80];
  reg: registers;

procedure set_cursor_position(page, x, y: byte);
begin
  with reg do
  begin
    AH := $02;
    BH := page;
    DH := y;
    DL := x;
  end;
  intr($10, reg);
end;

procedure get_cursor(page: byte; var x, y : byte);
begin
  reg.AH := $03;
  reg.BH := page;
  intr($10, reg);
end;
```

```

        with reg do
begin
y := DH;
x := DL;
end;
end;

procedure write_chars(page, x, y: byte; ch0: char; attr: byte;
multiplicity: word);
begin
set_cursor_position(page, x, y);
with reg do
begin
AH := $09;
AL := ord(ch0);
BH := page;
BL := attr;
CX := multiplicity;
end;
intr($10, reg);
end;

procedure write_string(page, x, y: byte; ss: string; attr: byte);
var
j: word;
begin
for j := 1 to length(ss) do
begin
write_chars(page, x, y, ss[j], attr, 1);
inc(x);
end;
end;
end;

begin
GraphDriver := detect;
InitGraph(GraphDriver, GraphMode, "");
setbkcolor(darkgray);
setcolor(lightblue);
ss := "Graphics mode";
fillellipse(320, 240, 20, 20);
x := 0;
y := 0;
write_string(0, x, y, ss, white);
set_cursor_position(0, 0, 5);
y := 5;
attr := 16 * brown + lightcyan;
for k := 0 to 15 do
begin
str(k:3, ss);
x := 3 * k;
write_string(0, x, y, ss, attr);
end;

```

```

for k := 0 to 15 do
    write_chars(0, 3 * k + 2, 6, chr(k), attr, 1);
    set_cursor_position(0, 0, 8);
    y := 8;
    for k := 0 to 15 do
    begin
        str((16 + k):3, ss);
        x := 3 * k;
        write_string(0, x, y, ss, attr);
    end;
    for k := 0 to 15 do
        write_chars(0, 3 * k + 2, 9, chr(16 + k), attr, 1);
    ss := "0123456";
    x := 0;
    y := 20;
    write_string(0, x, y, ss, 11);
    get_cursor(0, x, y, top, bottom);
    str(x, ss);
    str(y, tt);
    inc(y);
    ss := "At end of «0123456». (x, y) = (" + ss + ", " + tt + ")";
    write_string(0, x, y, ss, white);
    ss := "Press <Enter>: ";
    x := 0;
    y := 25;
    attr := 16 * darkgray + yellow;
    write_string(0, x, y, ss, attr);
    get_cursor(0, x, y, top, bottom);
    repeat
        write_chars(0, x, y, "J", yellow, 1);
        delay(100);
        write_chars(0, x, y, "=>", lightblue, 1);
        delay(100);
    until keypressed;
    readln;
    CloseGraph;
end.

```

Процедура `set_cursor_position` уже знакома нам по четвертому уроку. Напомню, что она задает положение курсора на указанной видеостранице. Процедура `get_cursor` определяет параметры курсора — номер видеостраницы и его положение (параметры `page`, `x` и `y`). В обоих случаях используются функции десятого прерывания.

Процедура `write_chars` предназначена для вывода на экран произвольного символа. С этой процедурой мы также знакомы по четвертому уроку. Напомню, что ее параметрами являются номер страницы, положение выводимого символа, сам символ, атрибут и «кратность». Кратность позволяет задать число повторений заданного символа и в определенных ситуациях может оказаться полезной. Обращаю внимание читателя на то, что и здесь используется прерывание \$10.

Процедура `write_string` является «надстройкой» над процедурой `write_chars`. Ее назначение — вывод символьной строки.

При запуске программы `demo_VGA` на выполнение вначале происходит инициализация графического режима. Затем с помощью процедуры `FillEllipse` модуля `Graph` рисуется (вообще говоря) эллипс, внутренняя часть которого имеет сплошную закраску. Далее следует демонстрация процедур вывода и работы с курсором. Здесь, в частности, следует обратить внимание на вывод результата выполнения процедуры `get_cursor`. Отметчены также два вызова процедур `write_chars` в конце программы, которые создают иллюзию мигающего двухцветного курсора. Мигание обеспечивается процедурами задержки `Delay` после каждого вызова процедуры `write_chars` с символами псевдографики.

В режиме VGA средней разрешающей способности (640×350) имеются две графические страницы (в режиме с высоким разрешением — только одна). Программа `test_graph_pages` (листинг 6.2) демонстрирует некоторые возможности, связанные с двухстраничной организацией памяти. Опущенные фрагменты исходного кода аналогичны фрагментам из предыдущего листинга.

#### Листинг 6.2. Использование графических страниц

```
program test_graph_pages;
uses Crt, Dos, Graph;
var
  GraphDriver, GraphMode, k: integer;
  ss, tt: string;
  reg: registers;
procedure set_cursor_position(page, x, y: byte);
...
procedure write_chars(page, x, y: byte; ch0: char; attr: byte;
  multiplicity: word);
...
procedure write_string(page, x, y: byte; ss: string; attr: byte);
...
begin
  GraphDriver := VGA;
  graphmode := VGAMed;
  InitGraph(GraphDriver, GraphMode, "");
  write_string(0, 20, 0, "Page 0", Yellow);
  write_string(0, 20, 1, "Press <Enter>: ", Yellow);
  write_string(1, 30, 0, "Page 1", Magenta);
  write_string(1, 30, 1, "Press <Enter>: ", Magenta);
  setvisualpage(0);
  setactivepage(1);
  ss := "XX";
end.
```

```
for k := 0 to 25 do
begin
  str(k, tt);
  tt := tt + " " + ss;
  if k <= 9 then
    tt := " " + tt;
  write_string(1, 0, k, tt, LightGreen);
  line(0, 13 + 14 * k, 8 * length(tt), 13 + 14 * k);
  ss := ss + "XX";
end;
setcolor(white);
setfillstyle(ltbkslashfill, LightRed);
bar(500, 0, 600, 349);
readln;
setvisualpage(1);
setactivepage(0);
setcolor(LightMagenta);
setfillstyle(xhatchfill, LightCyan);
fillellipse(500, 175, 137, 100);
readln;
setvisualpage(0);
readln;
setvisualpage(1);
readln;
RestoreCRTMode;
end.
```

Процедура модуля `Graph SetVisualPage(0)` позволяет вывести на экран страницу с номером 0, а `SetActivePage(1)` определяет в качестве рабочей видеостраницу с номером 1. Изображение строится на рабочей странице, но она может и не отображаться в данный момент времени на экране. В программе `test_graph_pages` при запуске программы, после инициализации графического режима, изображение строится сразу на двух видеостраницах, а на экран содержимое этих страниц выводится по очереди. Затем «невидимая» и отображаемая страницы меняются местами. В последнем параграфе этого урока мы будем использовать двухстраничную организацию видеопамяти для программирования движущихся изображений. Процедура `RestoreCRTMode` выполняет переключение в текстовый режим.

Познакомимся ближе с аппаратными средствами VGA. В модуле `Graph` имеются процедуры `GetImage` и `PutImage`, предназначенные для записи части графического экрана в видеобуфер и последующего восстановления этой части, возможно, в другом месте экрана. Обе эти процедуры неплохо работают в случае копирования маленькой области. Однако при сохранении и восстановлении целого графического экрана использование этих процедур дает не слишком хорошие результаты, в частности, по скорости выполнения. Эти операции можно сделать гораздо более быстрыми, если работать непосредственно с аппаратными средствами.

Такой подход иллюстрирует программа `save_VGA`. Она предназначена для сохранения содержимого графического экрана VGA и его восстановления. При реше-

нии этой задачи сразу возникает вопрос — где хранить изображение? Конечно, запись в память выполняется намного быстрее, чем сохранение экрана в файле на диске, однако для записи графической информации, содержащейся на экране VGA, требуется 153 600 байт, так что для большого рисунка памяти может не хватить. Программа `save_VGA` (листинг 6.3) записывает изображение на диск. Процедура `fill_screen` используется для заполнения экрана 16 цветами.

**Листинг 6.3.** Программа для записи изображения на диск

```
program save_VGA;
uses dos, crt, graph;
const
  pixels = 38400;
  file_name = "picture";
type
  plane = array[1..pixels] of byte;
var
  storage: file;
  result: word;
  graph_driver, graph_mode: integer;
procedure fill_screen;
const
  dx = 640 div 4;
  dy = 480 div 4;
var
  i, j, x, y: integer;
  color: word;
begin
  color := 0; y := 0;
  for i := 0 to 3 do
    begin
      x := 0;
      for j := 0 to 3 do
        begin
          setfillstyle(solidfill, color);
          bar(x, y, x + dx, y + dy);
          inc(x, dx); inc(color);
        end;
      inc(y, dy);
    end;
end;
procedure copy_screen;
var
  i: byte;
begin
  assign(storage, file_name);
```

```
rewrite(storage, 1);
port[$3CE] := 4;
for i := 3 downto 0 do
begin
  port[$3CF] := i;
  BlockWrite(storage, mem[segA000:$0000], pixels, result);
  if result <> pixels then
  begin
    writeln("Ошибка : i = ", i);
    readln;
    halt;
  end;
end;
close(storage);
end;

procedure restore_screen;
var
  i, j, k: byte;
begin
  assign(storage, file_name);
  reset(storage, 1);
  port[$3C4] := 2;
  j := 8;
  repeat
    port[$3C5] := j;
    blockread(storage, mem[segA000:$0000], pixels, result);
    if result <> pixels then break;
    j := j shr 1;
  until j = 0;
  close(storage);
  port[$3C5] := $0F;
end;

begin
  clrscr;
  graph_driver := detect;
  initgraph(graph_driver, graph_mode, "");
  if graphresult <> 0 then halt;
  fill_screen;
  copy_screen;
  readln;
  closegraph;
  writeln(" Текстовый режим, нажмите <Enter>:");
  readln;
  initgraph(graph_driver, graph_mode, "");
  restore_screen;
  outtext("Graph screen");
  readln;
  closegraph;
end.
```

Процедура `copy_screen` копирует экран на диск. Видеопамять VGA организована в виде 4 цветовых плоскостей: красной, зеленой, синей и серой. Любой пикселу соответствует в каждой плоскости определенный бит. Если включены все 4 бита, то пиксел будет белого цвета. Если установлен только красный бит, это соответствует красному пиксели. Если установлены бит красного цвета и серый бит, это даст светло-красный пиксель и т. д.

Для дальнейшего нам придется познакомиться с таким понятием, как *порт*. Термин «порт» в программировании может иметь разный смысл. Здесь мы будем иметь в виду специальный регистр процессора, через многоразрядные входы и выходы которого передаются команды и данные для управления работой других устройств. Порт ввода/вывода с адресом \$03CE содержит индекс регистров управления графикой VGA. Для считывания цветовой плоскости следует установить этот индекс равным 4. Затем задается считываемая плоскость (0...3) для порта ввода/вывода \$03CF. В этом случае фактические данные начинаются с абсолютного адреса \$A000:\$0000. Для считывания плоскости процедурами `Block-Write` (при работе с файлом) или `Move` (при копировании в память) просто копируются 38 400 байт ( $38\ 400 = 640 \times 480 \text{ div } 8$ ).

Обратный процесс требует обращения к контроллеру VGA, который управляет синхронизацией функций VGA. *Контроллер* — это специализированный процессор, предназначенный для управления внешним устройством, дисплеем, принтером и т. д. Наличие контроллера освобождает центральный процессор от выполнения этих функций. Индекс регистров контроллера VGA находится в порте ввода/вывода \$03C4. Установка этого значения равным 2 дает возможность записи на цветовые плоскости. Порт ввода/вывода \$03C4 определяет, на какую страницу производится запись. Запись можно производить более чем на одну страницу одновременно. Каждый из 4 младших битов порта \$03C4 управляет одной страницей, и битам от 0 до 4 могут быть присвоены определенные значения. Значение 6 (в двоичном представлении — 0110), к примеру, вызывает запись на страницы 1 и 2 одновременно.

Копирование графического экрана производится в файл `picture`. Изображение можно восстановить из этого же файла.

## Трехмерная графика

Освоив методы вывода плоских (двумерных) изображений, мы сможем перейти к решению более сложной задачи — задачи отображения сложных геометрических объектов, имеющих протяженность в трех измерениях. Основная трудность здесь может быть сформулирована на математическом языке следующим образом. Необходимо установить соответствие между точками поверхности *трехмерного* геометрического объекта и точками *плоской* поверхности экрана дисплея. При этом следует учитывать то обстоятельство, что некоторые части поверхности могут быть не видны. При построении более реалистических изображений приходится, кроме того, учитывать разные условия освещения, тип поверхности (ее «текстуру») и другие параметры.

Читателю уже стало ясно, что без маленькой лекции по математике нам не обойтись. Поэтому наберемся терпения и поговорим о геометрии.

## Векторы и операции над векторами

Прежде всего вспомним, что положение точки в пространстве задается с помощью координат. Значения координат зависят от выбора системы координат и положения точки отсчета. В компьютерной графике обычно используется *прямоугольная* (декартова) система координат, которая состоит из трех взаимно перпендикулярных осей, проходящих через общую точку, принимаемую за начало отсчета. В прямоугольной системе координат положение точки задается тремя числами: координатами  $x$ ,  $y$  и  $z$ . Эти значения отсчитываются от начала системы координат к основаниям перпендикуляров, проведенных из точки на соответствующие координатные оси (рис. 6.1).

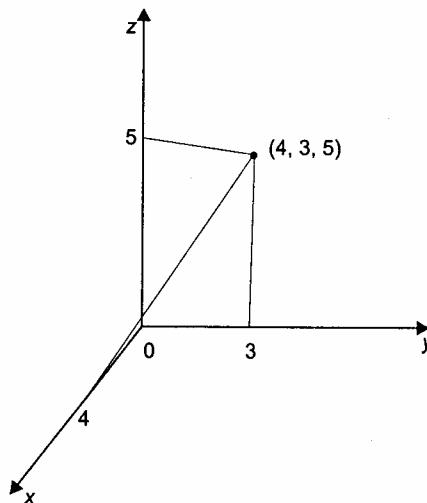


Рис. 6.1. Декартова система координат

В фиксированной системе координат одной точке соответствует единственная тройка значений координат, а любой тройке чисел соответствует единственная точка. В дальнейшем мы будем использовать векторные обозначения. При этом положение точки в пространстве будет задаваться вектором, который можно считать отрезком, проведенным из начала системы координат в заданную точку и имеющим направление из начала координат в заданную точку (рис. 6.2).

Взаимное положение двух точек тоже задается вектором, начинающимся в одной точке и направленным в другую точку. Вектор задается следующим образом:

$$\vec{v} = (x, y, z),$$

где  $x, y, z$  – это координаты точки. Для векторов определены действия и преобразования. Это прежде всего определение длины (нормы) вектора:

$$|\vec{v}| = \sqrt{x^2 + y^2 + z^2}.$$

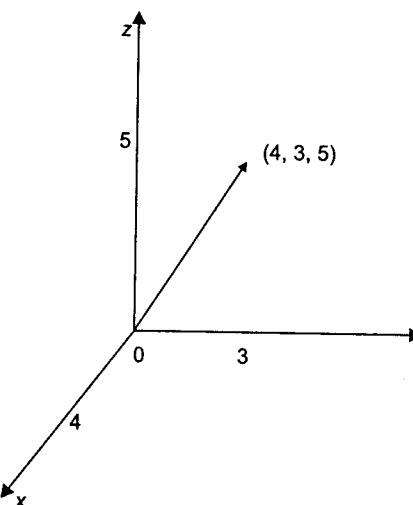


Рис. 6.2. Вектор в декартовой системе координат

Следующая векторная операция — это сложение векторов (рис. 6.3).

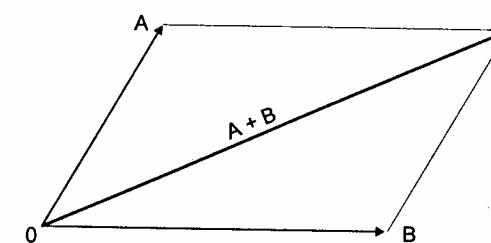


Рис. 6.3. Сложение векторов

Сумма двух векторов — также вектор, координаты которого являются суммой соответствующих координат обоих векторов:

$$A + B = (x_A + x_B, y_A + y_B, z_A + z_B).$$

Вычитание векторов можно рассматривать как сложение двух векторов, когда второе слагаемое представляет собой вектор, направление которого изменено на противоположное, что соответствует умножению на  $-1$ .

Умножение вектора на число (скаляр)  $a$  в результате дает вектор, имеющий то же (или противоположное, если множитель отрицательный) направление, но другую длину:

$$aA = (ax, ay, az).$$

*Скалярное произведение* – одна из важнейших операций в аналитической геометрии (и компьютерной графике). Скалярное произведение двух векторов является числом и определяется следующим образом:

$$\langle A \cdot B \rangle = x_A x_B + y_A y_B + z_A z_B$$

или

$$\langle A \cdot B \rangle = |A| |B| \cos(\theta),$$

где  $\theta$  – угол между векторами.

В геометрических построениях, связанных с компьютерной графикой, иногда приходится строить проекции одного вектора на другой (рис. 6.4).

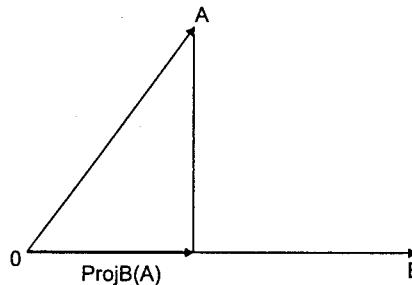


Рис.6.4. Проекция вектора

*Проекция* представляет собой вектор, имеющий то же (или противоположное) направление, что и вектор, на который производится проецирование, а длина проекции определяется как расстояние между основаниями перпендикуляров, проведенных из начальной и конечной точек проецируемого вектора на второй вектор. В общем случае проекция вектора  $A$  на вектор  $B$  может быть рассчитана как

$$\text{Proj}_B(A) = \vec{B} \frac{\langle \vec{A}, \vec{B} \rangle}{|\vec{B}|^2}$$

И наконец, определено *векторное произведение* двух векторов, которое представляет собой вектор:

$$A \times B = (x_A z_B - z_A y_B, z_A x_B - x_A z_B, x_A y_B - y_A x_B).$$

Полезно знать, что результирующий вектор в данном случае направлен перпендикулярно к обоим сомножителям. Это свойство векторного произведения используется для построения вектора, перпендикулярного к двум заданным.

## Векторные преобразования

Среди всех возможных векторных преобразований чаще всего приходится иметь дело с переносами и поворотами. Если имеется вектор

$$\vec{v} = (x, y, z),$$

то *переносом* называется следующее преобразование:

$$v \rightarrow v' = (x + \Delta x, y + \Delta y, z + \Delta z).$$

Чтобы ввести преобразование поворота, нам придется познакомиться с элементами матричной алгебры. *Матрицей* называют прямоугольную (в общем случае) таблицу, элементами которой являются числа. Определены операции сложения и умножения матриц. Будем обозначать элемент матрицы переменной с двумя индексами, определяющими номер строки и номер столбца:  $a_{ij}$ . Тогда результат *умножения* матрицы  $A$  на вектор  $c$  — тоже вектор, который задается следующим образом:

$$b_i = \sum_{j=1}^n a_{ij} c_j.$$

*Произведение* двух матриц  $A$  и  $C$  является матрицей  $B$ , элементы которой вычисляются по формуле

$$b_{ik} = \sum_{j=1}^n a_{ij} c_{jk}.$$

*Сумма* двух матриц также является матрицей, элементы которой получаются суммированием соответствующих элементов матриц-слагаемых. При вычислении суммы и произведения матриц должно соблюдаться соответствие размеров слагаемых или сомножителей. Это значит, что размеры матриц-слагаемых должны совпадать, а число столбцов в первом сомножителе должно быть равно числу строк во втором сомножителе.

Еще одно важное понятие линейной алгебры — матрица, *обратная* к данной. Матрицей, обратной по отношению к матрице  $A$ , называется матрица  $A^{-1}$ , удовлетворяющая условию

$$A^{-1} A = A A^{-1} = I,$$

где  $I$  — единичная матрица, то есть матрица, у которой на главной диагонали стоят единичные элементы ( $I_{kk} = 1$ ), а все остальные элементы равны нулю.

Особую роль в машинной графике играют вращения, которые могут быть описаны в терминах матричных операций. Прежде всего замечу, что при описании трехмерных объектов в компьютерной графике обычно используется так называемая *правая система координат*. Пусть имеется набор из трех единичных векторов, направленных, соответственно, вдоль координатных осей  $Ox$ ,  $Oy$  и  $Oz$ . Если система координат является правой, это означает, что в том случае, когда поворот от вектора к вектору на  $90^\circ$  соответствует повороту винта с правой резьбой, направление вектора совпадает с направлением перемещения винта. Повороты относительно каждой координатной оси на произвольный угол  $\alpha$  могут быть описаны *матрицами поворота*:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{bmatrix}$$

$$R_y(\alpha) = \begin{bmatrix} \cos \alpha & 0 & \sin \alpha \\ 0 & 1 & 0 \\ -\sin \alpha & 0 & \cos \alpha \end{bmatrix}$$

$$R_z(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

При этом угол поворота считается положительным, если вращение соответствует положительному направлению оси по правилу правого винта (рис. 6.5).

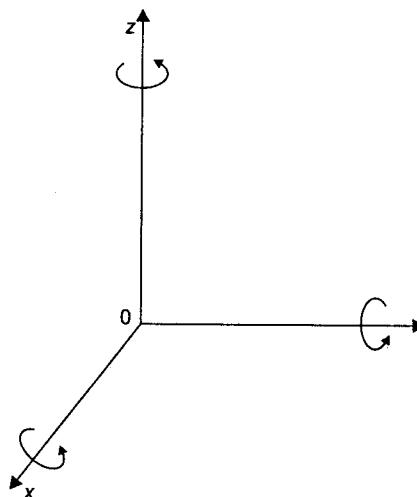


Рис. 6.5. Положительные направления вращения вокруг координатных осей

Здесь, правда, есть исключение. Читатель, искушенный в математике, вероятно, обратит внимание на то, что для оси  $y$  направление отсчета угла противоположное. Дело вот в чем. При вращении вокруг этой оси положительным направлением измерения углов является направление от оси  $z$  к оси  $x$ . Однако традиционно углы измеряются в противоположном направлении! Таким образом, вращение на угол  $\alpha$  в правой системе координат эквивалентно вращению на угол  $-\alpha$ , изменивший в направлении от оси  $x$  к оси  $z$ .

В результате поворота вектора  $v$  относительно оси  $x$ , например, получим новый вектор, координаты которого получаются в результате умножения матрицы поворота на исходный вектор:

$$v' = R_x(\alpha) v.$$

В общем случае поворот вектора вокруг произвольной оси, проходящей через точку начала координат, можно свести к этим трем типам поворота. Поступают следующим образом. Выбирают такую последовательность поворотов исходной системы координат, чтобы в конечном итоге новое направление положительной оси  $z$  совпало с осью вращения. Изменение координат вектора при поворотах системы координат относительно ее осей описывается матрицами, обратными к матрицам  $R_x$ ,  $R_y$  и  $R_z$ .

Произвольная ось вращения задается своими координатами:

$$\vec{r} = (x_r, y_r, z_r).$$

Она образует с осью  $z$  угол  $\varphi$  (рис. 6.6), а ее проекция на плоскость  $xOy$  образует с осью  $x$  угол  $\theta$ .

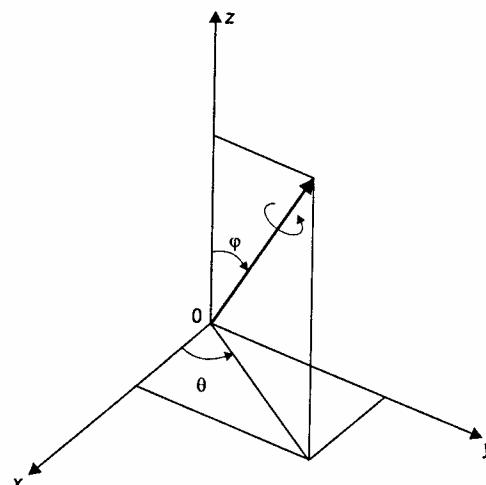


Рис. 6.6. Поворот вокруг произвольной оси вращения

Эти углы вычисляются по следующим формулам:

$$\varphi = \arccos\left(\frac{z_r}{|\vec{r}|}\right),$$

$$\theta = \begin{cases} \arctan\left(\frac{y_r}{x_r}\right), & x_r > 0, \\ \pi + \arctan\left(\frac{y_r}{x_r}\right), & x_r < 0, \\ \frac{\pi}{2}, & x_r = 0, \quad y_r \geq 0, \\ \frac{3\pi}{2}, & x_r = 0, \quad y_r < 0. \end{cases}$$

Начнем с поворота осей  $x$  и  $y$  вокруг оси  $z$  на угол  $\theta$ . Этот поворот задается матрицей

$$R_z^{-1}(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Затем оси  $x'$  и  $z'$  поворачиваем вокруг оси  $y'$  на угол  $\phi$  до совпадения оси  $z''$  с вектором  $r$ . Этот поворот описывается следующей матрицей:

$$R_y^{-1}(\phi) = \begin{bmatrix} \cos \phi & 0 & -\sin \phi \\ 0 & 1 & 0 \\ \sin \phi & 0 & \cos \phi \end{bmatrix}$$

Фактический поворот вокруг вектора  $r$  на угол  $\alpha$  теперь можно выполнить как поворот вокруг оси  $z''$ :

$$R_r(\alpha) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Теперь систему координат необходимо вернуть в прежнее положение, выполнив обратные повороты. Таким образом, оказывается, что поворот вокруг произвольного вектора можно описать матрицей поворота, которая является произведением матриц поворотов для отдельных осей:

$$R = R_z R_y R_r R_y^{-1} R_z^{-1}$$

Преобразование сдвига тоже можно записать в форме матричного умножения, если добавить в запись вектора четвертый компонент, равный единице:

$$r = (x_r, y_r, z_r, 1)$$

и перейти от матриц  $3 \times 3$  к матрицам  $4 \times 4$ :

$$R^* = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Сдвиг при этом описывается матрицей

$$T = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Общее преобразование, включающее не только поворот, но и сдвиг, может быть представлено в виде матрицы

$$R_{\text{gen}} = T R^* T^{-1}.$$

Преобразование *масштабирования* также может быть описано матрицей:

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Это преобразование заключается в изменении масштаба по каждой координате.

Четырехмерные координаты, введенные выше, в вычислительной геометрии принято называть *однородными координатами*, их использование позволяет формально объединить такие разнородные преобразования, как повороты, сдвиги и масштабирование.

## Перспективные изображения и проекции

Теперь поговорим немного о тех особенностях восприятия человеком окружающего (трехмерного) мира, которые следует учитывать при программировании трехмерных изображений. Впервые серьезно об этом задумались художники итальянского Возрождения, которые стремились к максимальному реализму в своих произведениях. Реализм, кроме всего прочего, предполагает воспроизведение пространственной глубины объекта — природного ландшафта, строения, человеческого тела. Именно художники Возрождения ввели ряд важнейших геометрических понятий, которые затем легли в основу *проективной геометрии*. Читатель, утомленный математическими выкладками этого урока, возможно, вздрогнул, прочитав это название. Успокою читателя — проективной геометрией мы заниматься не будем, это, как говорится, уже другая история, но с некоторыми основными идеями нам придется познакомиться.

Одно из важнейших наблюдений состояло в том, что даже строго параллельные линии человек воспринимает как параллельные только в редких, специальных случаях. Обычно такие линии (или их продолжения) кажутся человеку сходящимися. Это явление получило название *эффекта перспективы*. Если отметить все точки схождения параллельных линий, нанесенных на участок поверхности Земли, который можно считать локально плоским, получим *линию горизонта*.

С наблюдателем связаны *точка наблюдения* и *направление наблюдения*. Замечу, что если речь идет о наблюдении какого-либо объекта (например, куба), то эффект перспективы будет обратно пропорционален расстоянию от объекта до точки наблюдения. Если глаз расположен далеко от объекта, то параллельные линии объекта (параллельные ребра куба) будут казаться параллельными и наблюдателю. С другой стороны, если приблизиться к объекту достаточно близко, эффект перспективы усиливается и ребра куба будут казаться расходящимися.

Положение объекта обычно задается в системе так называемых *мировых координат*. Мировые координаты точек объекта в конечном итоге надо преобразовать в экранные координаты. Экран — плоскость, следовательно, экранные координаты являются двумерными. Еще говорят о системе *видовых координат*, связанной с наблюдателем. Мировые координаты преобразуются в видовые (видовое преобразование), а видовые координаты преобразуются в экранные (перспективное преобразование).

Положение наблюдателя (его глаза или фотокамеры) задается координатами точки наблюдения. Удобно считать, что объект находится недалеко от начала мировой системы координат. Вектор наблюдения  $E0$  (рис. 6.7) определяет направление наблюдения; кроме того, считают, что точки объекта можно видеть только внутри конуса, вершина которого совпадает с точкой наблюдения, а ось — с линией  $E0$ .

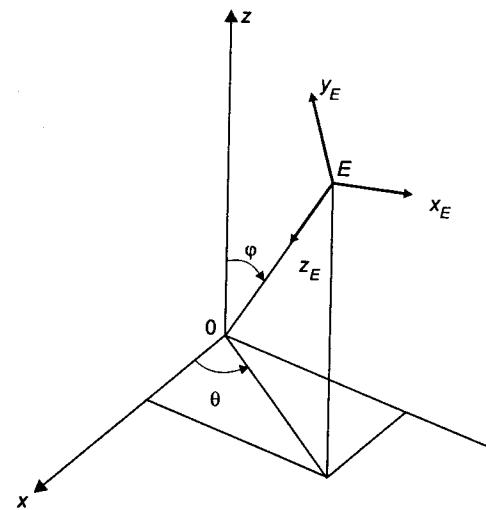


Рис. 6.7. Система видовых координат

Система координат  $x_Ey_Ez_E$ , изображенная на рис. 6.7, является видовой системой координат. Ее начало расположено в точке  $E$ , положительная ось  $Ey_E$  направлена вверх, а положительная ось  $Ex_E$  — вправо. Такой выбор осей позволит определить экранные оси в тех же направлениях. Экран расположен между точками  $E$  и  $O$  перпендикулярно линии наблюдения.

Прямоугольные координаты точки наблюдения в мировой системе координат связаны со сферическими координатами  $(\rho, \phi, \theta)$  следующими соотношениями:

$$\rho = \sqrt{x_E^2 + y_E^2 + z_E^2},$$

$$x_E = \rho \sin \phi \cos \theta,$$

$$y_E = \rho \sin \phi \sin \theta,$$

$$z_E = \rho \cos \phi.$$

Преобразование координат точки из мировой системы координат в видовую выполняется по формуле

$$\vec{r}_E = V \vec{r}_v,$$

где  $V$  — матрица преобразования, имеющая вид

$$V = \begin{bmatrix} -\sin \theta & \cos \theta & 0 & 0 \\ -\cos \varphi \cos \theta & -\cos \varphi \sin \theta & \sin \varphi & 0 \\ -\sin \varphi \cos \theta & -\sin \varphi \sin \theta & -\cos \varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Остается сделать последний шаг — перейти к экранным координатам. Это, собственно, и есть проецирование.

Различают два типа проецирования — *параллельное* и *перспективное*. В параллельных проекциях игнорируется видимое нарушение пропорций объекта, возникающее вследствие различной удаленности его частей. Это соответствует удалению точки наблюдения на бесконечно большое расстояние. В перспективных проекциях эффект перспективы учитывается.

Примером параллельной проекции является *ортогональная* проекция, которая получается простым игнорированием  $z$ -координаты после перехода в видовую систему координат. Известны и другие проекции такого вида, например изометрическая. В параллельных проекциях параллельные геометрические линии объекта остаются параллельными и после проецирования.

В случае перспективной проекции следует определить экранные координаты  $X$  и  $Y$ , учитывая то, что точка удалена на расстояние  $z$ , а экран расположен на расстоянии  $d$  от точки наблюдения (рис. 6.8).

Точка  $S$  является началом экранной системы координат. Точка  $P'(x, 0, z)$  — это точка, изображение которой надо спроектировать на плоскость экрана, для простоты  $y$ -координата взята равной нулю. Нам следует, таким образом, вычислить значение координаты  $X$  — точки пересечения отрезка  $EP'$  с плоскостью экрана. Для этого вновь придется вспомнить геометрию — свойства подобных треугольников, из которых следует соотношение

$$\frac{PS}{ES} = \frac{P'Q}{EQ}.$$

Отсюда получаем

$$\frac{X}{d} = \frac{x}{z} \text{ и } X = d \frac{x}{z}.$$

Аналогичную формулу можно записать и для экранной координаты  $Y$ . Замечу, что в своих рассуждениях мы предполагали, что начало экранной системы координат находится в центре экрана. Если это не так, в правую часть формул для  $X$

и  $Y$  надо добавить слагаемые, задающие смещение начала экранной системы координат относительно центра экрана.

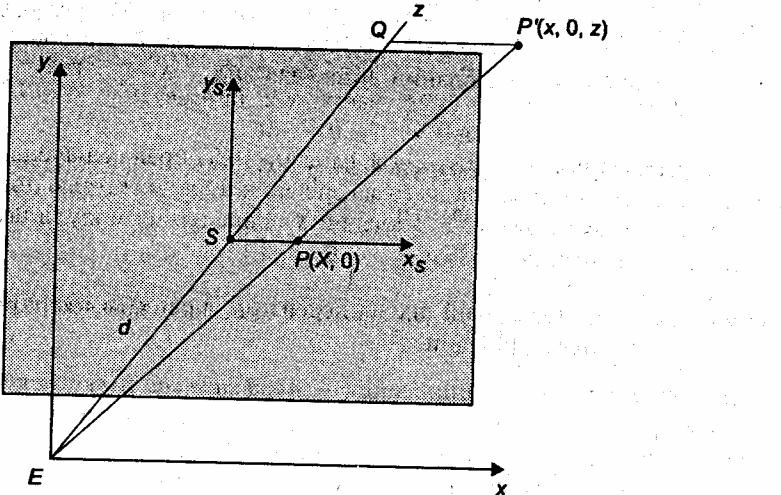


Рис. 6.8. Определение экранных координат для перспективной проекции

Математические и, в значительной степени, технические рисунки выполняются в ортогональных проекциях.

## Модуль graphs3d

Модуль graphs3d предназначен для вывода трехмерных изображений. В нем реализованы некоторые из тех идей, которые обсуждались выше. Листинг 6.4 содержит исходный текст интерфейсной секции этого модуля.

### Листинг 6.4. Интерфейсная секция модуля graphs3d

```
unit graphs3d;

interface

uses dos, crt, graph, bgi_chr;

type
  vector = array[1..3] of real;
  projection = array[1..2] of vector;

var
  abs_x_center, abs_y_center : integer;

procedure open_graph();
procedure close_graph();
procedure out_text_XY(ss : string; x, y : integer; color : word);
```

```

procedure put_pixel(x, y : integer; color : word);
procedure norm_line(x0, y0, x1, y1 : integer; color : word);
procedure compute_isometric_matrix(var P : projection);
procedure compute_dimetric_matrix(alpha : real; var P : projection);
procedure compute_oblique_matrix(alpha : real; var P : projection);
procedure project(const P : projection; const x, y, z : real;
  const u0, v0 : integer; var u, v : integer);

```

Процедуры `open_graph` и `close_graph` (листинг 6.5) предназначены для инициализации графического режима и его завершения. Если читателю покажется, что эти процедуры в чем-то беднее аналогичных процедур из модуля `Graphs`, он сможет заменить их своими версиями.

**Листинг 6.5.** Процедуры инициализации и завершения графического режима модуля `graphs3d`

```

implementation

var
  hold_color : word;

procedure open_graph;
var
  GraphDriver, GraphMode : integer;
begin
  GraphDriver := VGA;
  GraphMode := VGAHi;
  InitGraph(GraphDriver, GraphMode, "");
  SetBkColor(black);
  SetColor(white);
  SetTextStyle(SansSerifFont, HorizDir, 4);
  SetTextJustify(LeftText, BottomText);
  abs_x_center := (GetMaxX + 1) div 2;
  abs_y_center := (GetMaxY + 1) div 2;
end;

procedure close_graph;
var
  gr: integer;
begin
  CloseGraph();
  gr := GraphResult;
  if gr <> 0 then
    begin
      writeln("GraphResult = ", gr);
      readln;
      halt;
    end;
end;

```

Далее следуют процедуры вывода текста в графическом режиме, вывода пикселя и линии — `out_text`, `put_pixel` и `norm_line` (листинг 6.6). Их главная особен-

ность состоит в том, что они позиционируются (то есть ведут отсчет графических координат) относительно центра экрана. В процедуру `norm_line` добавлен, кроме того, еще один параметр, определяющий цвет линии.

#### Листинг 6.6. Процедуры `out_text_XY`, `put_pixel` и `norm_line` модуля `graphs3d`

```

procedure out_text_XY(ss : string; x, y : integer; color : word);
begin
    hold_color := getcolor;
    setcolor(color);
    OutTextXY(x + abs_x_center, abs_y_center - y, ss);
    setcolor(hold_color);
end;

procedure put_pixel(x, y : integer; color : word);
begin
    PutPixel(x + abs_x_center, abs_y_center - y, color);
end;

procedure norm_line(x0, y0, x1, y1 : integer; color : word);
begin
    hold_color := GetColor;
    SetColor(color);
    Line(x0 + abs_x_center, abs_y_center - y0,
        x1 + abs_x_center, abs_y_center - y1);
    SetColor(hold_color);
end;

```

Следующие процедуры модуля `graphs3d` составляют его основную часть. Именно они и являются реализацией тех методов построения проекций, о которых мы говорили в теоретической части данного параграфа. Мы уже знаем, что имеется несколько типов проекций — в модуле `graph3d` реализованы ортогональные и косоугольные проекции. В ортогональных проекциях лучи из точки проецирования, расположенной в бесконечности, перпендикулярны плоскости экрана. В косоугольных проекциях лучи не ортогональны плоскости рисунка, а пространственные оси *y* и *z* обычно выбираются в этой плоскости.

Процедуры `compute_isometric_matrix` и `compute_dimetric_matrix` вычисляют матрицы для двух основных видов ортогонального проецирования. Замечу, что далее мы для наглядности все-таки будем разделять преобразования сдвига и поворота. Массив *P* содержит матрицу проецирования:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} + \begin{bmatrix} P_{11} & P_{12} & P_{13} \\ P_{21} & P_{22} & P_{23} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

С точки зрения матричной алгебры, проекция называется ортогональной, если векторы, соответствующие строкам (двум) матрицы проецирования, являются ортогональными (проще говоря, взаимно перпендикулярными). Математически

условие ортогональности записывается как равенство нулю скалярного произведения двух векторов.

В изометрической проекции считается, что оси  $x$ ,  $y$  и  $z$  мировой системы координат составляют равные углы с плоскостью экрана. Эта проекция называется изометрической потому, что расстояния по трем координатным осям сокращаются на одну и ту же величину. Чтобы построить матрицу изометрической проекции, можно рассуждать следующим образом. Ось  $z$  должна проецироваться только на ось  $v$ :

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ c \\ c \end{bmatrix}$$

Для этого надо положить  $P_{13} = 0$ ,  $P_{23} = c$ . Проекции двух других осей должны иметь равные длины и равные проекции на ось  $v$ . Отсюда вытекают следующие условия:

$$P_{11}^2 + P_{21}^2 = P_{12}^2 + P_{22}^2 = c^2,$$

$$P_{21} = P_{22} \equiv a.$$

Добавим сюда условие ортогональности векторов

$$P_{11}P_{21} + P_{12}P_{22} + P_{13}P_{23} = 0$$

и решим полученную систему уравнений. Результат будет такой:

$$P = \begin{bmatrix} -\sqrt{3}a & \sqrt{3}a & 0 \\ -a & -a & 2a \end{bmatrix}.$$

Остается найти значение масштабного множителя  $a$ . От величины этого множителя зависят геометрические размеры проекции фигуры (масштаб изображения). Принято выбирать его следующим образом:

$$a = \frac{1}{\sqrt{6}}.$$

Обоснование такого выбора можно найти в специальной литературе по аналитической геометрии или машинной графике. В результате получаем матрицу изометрической проекции:

$$P = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & \sqrt{\frac{2}{3}} \end{bmatrix}$$

Вычисление матрицы изометрической проекции реализовано в процедуре `compute_isometric_matrix` (листинг 6.7).

**Листинг 6.7.** Процедура вычисления матрицы изометрической проекции модуля graphs3d

```
procedure compute_isometric_matrix(var P : projection);
begin
  P[1, 1] := -1.0 / sqrt(2.0);
  P[1, 2] := -P[1, 1];
  P[1, 3] := 0.0;
  P[2, 1] := -1.0 / sqrt(6.0);
  P[2, 2] := P[2, 1];
  P[2, 3] := -2.0 * P[2, 1];
end;
```

В диметрической проекции, которая также относится к числу ортогональных проекций, ось  $z$  проецируется на ось  $v$  экранной системы координат, а проекции осей  $x$  и  $y$  составляют некоторый заданный угол  $\alpha$  с отрицательной полуосью  $u$  и положительной полуосью  $u$  соответственно. Диметрическая проекция включает изометрическую как частный случай. Угол в процедуре `compute_dimetric_matrix` (листинг 6.8) задается в градусах.

**Листинг 6.8.** Процедура вычисления матрицы диметрической проекции модуля graphs3d

```
procedure compute_dimetric_matrix(alpha : real; var P : projection);
var
  t : real;
begin
  alpha := pi * alpha / 180.0;
  P[1, 1] := -1.0 / sqrt(2.0);
  P[1, 2] := -P[1, 1];
  P[1, 3] := 0.0;
  t := sin(alpha) / cos(alpha);
  P[2, 1] := t * P[1, 1];
  P[2, 2] := P[2, 1];
  P[2, 3] := sqrt(1.0 - sqr(t));
end;
```

В косоугольной проекции оси  $y$  и  $z$  мировой системы координат совпадают с осями  $u$  и  $v$  экранной системы, а проекция оси  $x$  составляет заданный угол  $\alpha$  с отрицательной полуосью  $v$ . Текст соответствующей процедуры приведен в листинге 6.9.

**Листинг 6.9.** Процедура вычисления матрицы косоугольной проекции модуля graphs3d

```
procedure compute_oblique_matrix(alpha : real; var P : projection);
begin
  alpha := pi * alpha / 180.0;
  P[1, 1] := -sin(alpha);
  P[1, 2] := 1.0;
  P[1, 3] := 0.0;
  P[2, 1] := -cos(alpha);
  P[2, 2] := 0.0;
```

```
P[2, 3] := 1.0;
end;
```

В процедуре `project` (листинг 6.10) производится пересчет мировых координат в экранные с учетом выбранной проекции. Первый параметр здесь — матрица проекции, которая должна быть вычислена перед вызовом процедуры `project`, следующие три параметра — координаты проецируемой точки. Затем следуют два параметра, задающие сдвиг вдоль осей экранных координат, а последние два параметра возвращаются процедурой и являются экранными координатами точки.

#### Листинг 6.10. Процедура `project` модуля `graphs3d`

```
procedure project(const P : projection; const x, y, z : real;
                  const u0, v0 : integer; var u, v : integer);
begin
  u := u0 + round(P[1, 1] * x + P[1, 2] * y + P[1, 3] * z);
  v := v0 + round(P[2, 1] * x + P[2, 2] * y + P[2, 3] * z);
end;
```

Секция инициализации (листинг 6.11) при запуске основной программы регистрирует графический драйвер и шрифты, то есть загружает их в динамически распределяемую область памяти.

#### Листинг 6.11. Секция инициализации модуля `graphs3d`

```
begin
  if RegisterBGI_driver(@EGAVGADriverProc) < 0 then
    begin
      clrscr;
      writeln(GraphErrorMsg(GraphResult));
      write("Press <Enter>; "); readln;
      halt;
    end;

  if RegisterBGIFont(@SmallFontProc) < 0 then
    begin
      clrscr;
      writeln(GraphErrorMsg(GraphResult));
      write("Press <Enter>; "); readln;
      halt;
    end;

  if RegisterBGIFont(@SansSerifFontProc) < 0 then
    begin
      clrscr;
      writeln(GraphErrorMsg(GraphResult));
      write("Press <Enter>; "); readln;
      halt;
    end;
end.
```

Пример использования модуля `graphs3d` дан в программе `projections` (листинг 6.12).

#### Листинг 6.12. Программа `projections`

```

program projections;

uses graph, crt, graphs3d;

var
  x, z, y, xstep, ystep, theta : real;
  i, j, xold, yold, xnew, ynew : integer;
  P : projection;

const
  xCount = 50;
  yCount = 50;
  xMin = -100;
  xMax = 100;
  yMin = -100;
  yMax = 100;

function Fun(x, y : real) : real;
begin
  Fun:=cos(sqrt(x * x + y * y));
end;

procedure init;
begin
  clearviewport;
  SetColor(14); setbkcolor(1);
end;

procedure FindScreenCoordinates(var x, y :real; var xp, yp : integer);
begin
  z:=10 * Fun(0.1 * x,0.1 * y);
  project(P, x, y, z, 0, xnew, ynew);
end;

procedure draw_cube;
var
  xp, yp : array[1..8] of integer;
begin
  project(P, 50, 50, 50, 0, 0, xp[1], yp[1]);
  project(P, -50, 50, 50, 0, 0, xp[2], yp[2]);
  project(P, -50, -50, 50, 0, 0, xp[3], yp[3]);
  project(P, 50, -50, 50, 0, 0, xp[4], yp[4]);
  project(P, 50, 50, -50, 0, 0, xp[5], yp[5]);
  project(P, -50, 50, -50, 0, 0, xp[6], yp[6]);
  project(P, -50, -50, -50, 0, 0, xp[7], yp[7]);
  project(P, 50, -50, -50, 0, 0, xp[8], yp[8]);

  norm_line(xp[2], yp[2], xp[1], yp[1], White);

```

```
norm_line(xp[3], yp[3], xp[2], yp[2], White);
norm_line(xp[4], yp[4], xp[3], yp[3], White);
norm_line(xp[1], yp[1], xp[4], yp[4], White);
norm_line(xp[5], yp[5], xp[1], yp[1], White);
norm_line(xp[6], yp[6], xp[5], yp[5], White);
norm_line(xp[7], yp[7], xp[6], yp[6], LightGray);
norm_line(xp[8], yp[8], xp[7], yp[7], LightGray);
norm_line(xp[5], yp[5], xp[8], yp[8], White);
norm_line(xp[2], yp[2], xp[6], yp[6], White);
norm_line(xp[7], yp[7], xp[3], yp[3], LightGray);
norm_line(xp[4], yp[4], xp[8], yp[8], White);

end;

procedure draw_surf;
begin
  xStep := (xMax - xMin) / xCount;
  yStep := (yMax - yMin) / yCount;
  for i := 0 to xCount do
    begin
      x := xmin + i * xstep;
      y := ymin;
      findscreencordinates(x, y, xnew, ynew);
      xold := xnew;
      yold := ynew;
      for j := 0 to yCount do
        begin
          y := ymin + j * ystep;
          findscreencordinates(x, y, xnew, ynew);
          norm_line(xnew, ynew, xold, yold, Yellow);
          xold := xnew;
          yold := ynew;
        end;
    end;
  for i := 0 to yCount do
    begin
      y := ymin + i * ystep;
      x := xmin;
      findscreencordinates(x, y, xnew, ynew);
      xold := xnew;
      yold := ynew;
      for j := 0 to xCount do
        begin
          x := xmin + j * xstep;
          findscreencordinates(x, y, xnew, ynew);
          norm_line(xnew, ynew, xold, yold, Yellow);
          xold := xnew;
          yold := ynew;
        end;
    end;
  end;
```

```

abs_x_center := 0; abs_y_center := 0;
open_graph;

compute_isometric_matrix(P); init;
out_text_XY("Isometric Projection", -150, 150, Yellow);
draw_cube; readln;

theta:=45;
compute_oblique_matrix(theta, P); init;
out_text_XY("Oblique Projection", -150, 150, Yellow);
draw_cube; readln;

theta:=15;
compute_dimetric_matrix(theta, P); init;
out_text_XY("Dimetric Projection", -150, 150, Yellow);
draw_cube; readln;

compute_isometric_matrix(P); init;
out_text_XY("Isometric Projection", -150, 150, Yellow);
draw_surf; readln;

theta:=45;
compute_oblique_matrix(theta, P); init;
out_text_XY("Oblique Projection", -150, 150, Yellow);
draw_surf; readln;

theta:=15;
compute_dimetric_matrix(theta, P); init;
out_text_XY("Dimetric Projection", -150, 150, Yellow);
draw_surf; readln;

close_graph;
end.

```

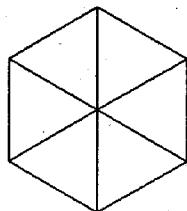
Эта программа строит три проекции куба и изображение более сложной поверхности, описываемой уравнением

$$z = \cos(\sqrt{x^2 + y^2}).$$

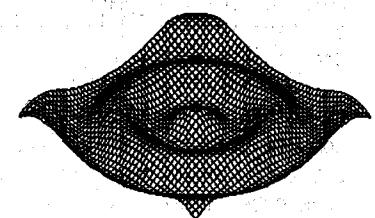
Куб задается координатами восьми своих вершин, которые соединяются отрезками прямых, поэтому для того, чтобы построить проекцию куба, необходимо спроецировать вершины, а затем соединить их между собой. Сложная поверхность аппроксимируется (то есть заменяется другой, более простой, но похожей) поверхностью, собранной из прямолинейных отрезков, соединяющих достаточно близкие точки исходной поверхности. В этом случае на плоскость экрана проецируются начальная и конечная точки отрезка.

Результат (рис. 6.9) похож на изображения проволочных каркасов (их, кстати, так иногда и называют). А можно ли изобразить сплошной, непрозрачный объект? Да, можно, и об этом мы поговорим в следующем параграфе.

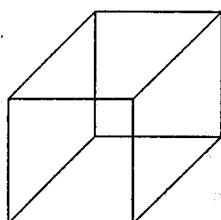
## Isometric Projection



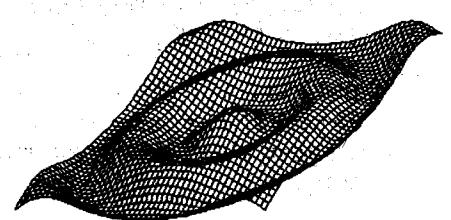
## Isometric Projection



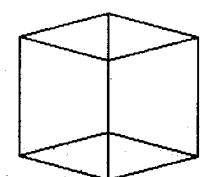
## Oblique Projection



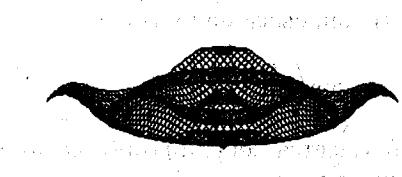
## Oblique Projection



## Dimetric Projection



## Dimetric Projection



**Рис. 6.9.** Графический вывод программы *projections*: «isometric» — изометрическая, «oblique» — косоугольная и «dimetric» — диметрическая проекции

## Построение непрозрачных объектов

Чем отличается непрозрачный объект от прозрачного? Тем, что некоторые участки его поверхности не видны наблюдателю, так как загораживаются другими

частями этого (а возможно, и другого) объекта. Следовательно, чтобы научиться строить непрозрачные, сплошные геометрические объекты, нам прежде всего надо научиться удалять невидимые части.

Имеется несколько алгоритмов удаления невидимых частей. Среди них алгоритм художника,  $z$ -буфер, порталы и т. д. Мы ограничимся простым случаем построения изображения непрозрачного куба. Куб удобен для анализа потому, что он состоит из шести квадратных граней. В компьютерной графике даже сложные поверхности заменяют похожими поверхностями, «собранными» из полигонов — плоских многоугольных граней. Будем считать, что каждая грань куба содержит 5 «опорных» точек — четыре вершины и центр. В центральной точке построим нормаль — перпендикуляр к поверхности. Если считать, что нормаль (в нашем случае) направлена из объекта, то в видовой системе координат проекция нормали на ось  $z$ , которую мы будем считать идущей к плоскости экрана и наблюдателю, будет положительной в том случае, когда грань видна наблюдателю, и отрицательной, если она не видна. Так мы получаем достаточно простой алгоритм определения видимых и невидимых граней. Этот алгоритм можно применить и в общем случае — в случае поверхности, собранной из полигонов. При этом, конечно, возникает вопрос: как построить нормаль? Ответ простой — если считать, что две стороны полигона, выходящие из одной общей вершины, являются векторами, то их векторное произведение можно взять за нормаль к поверхности этого полигона. Направление нормали будет однозначно определено, если задан порядок векторных сомножителей.

Описанный алгоритм используется в программе `cubes`. Исходный текст этой программы приведен в листинге 6.13.

**Листинг 6.13.** Программа сибе

```

((x : -25; y : -25; z : -25),
(x : +25; y : -25; z : -25),
(x : +25; y : -25; z : +25),
(x : -25; y : -25; z : +25),
(x : 0; y : -25; z : 0)),

((x : -25; y : +25; z : +25),
(x : +25; y : +25; z : +25),
(x : +25; y : +25; z : -25),
(x : -25; y : +25; z : -25),
(x : 0; y : +25; z : 0)),

((x : -25; y : -25; z : -25),
(x : -25; y : -25; z : +25),
(x : -25; y : +25; z : +25),
(x : -25; y : +25; z : -25),
(x : -25; y : 0; z : 0)),

((x : +25; y : -25; z : +25),
(x : +25; y : -25; z : -25),
(x : +25; y : +25; z : -25),
(x : +25; y : +25; z : +25),
(x : +25; y : 0; z : 0)),

((x : -25; y : +25; z : -25),
(x : +25; y : +25; z : -25),
(x : +25; y : -25; z : -25),
(x : -25; y : -25; z : -25),
(x : 0; y : 0; z : -25))));

procedure rotx;
var
  y1, z1 : real;
  i, j : integer;
begin
  for i := 1 to 6 do
    for j := 1 to 5 do
      begin
        y1 := pcos * cube[i, j].y - psin * cube[i, j].z;
        z1 := psin * cube[i, j].y + pcos * cube[i, j].z;
        cube[i, j].y := y1;
        cube[i, j].z := z1;
      end
  end;

procedure roty;
var
  x1, z1 : real;
  i, j : integer;
begin
  for i := 1 to 6 do
    for j := 1 to 5 do

```

```

begin
  x1 := pcos * cube[i, j].x - psin * cube[i, j].z;
  z1 := psin * cube[i, j].x + pcos * cube[i, j].z;
  cube[i, j].x := x1;
  cube[i, j].z := z1;
end
end;

procedure rotz;
var
  y1, x1 : real;
  i, j : integer;
begin
  for i := 1 to 6 do
    for j := 1 to 5 do
      begin
        y1 := pcos * cube[i, j].y - psin * cube[i, j].x;
        x1 := psin * cube[i, j].y + pcos * cube[i, j].x;
        cube[i, j].y := y1;
        cube[i, j].x := x1;
      end
end;
end;

procedure display_cube(col : byte);
var
  i, j, a : integer;
  c : byte;
  vertex:array[1..4] of pointtype;
begin
  for i := 1 to 6 do
    if cube[i, 5].z > 0 then
      begin
        for a := 1 to 3 do
          begin
            if col > 0 then
              c := white
            else
              c := 0;
            setcolor(c);
            for j := 1 to 4 do
              begin
                vertex[j].x := a * 150 + round(cube[i, j].x);
                vertex[j].y := getmaxy div 2 + round(cube[i, j].y);
              end;
            line(vertex[1].x, vertex[1].y, vertex[2].x, vertex[2].y);
            line(vertex[2].x, vertex[2].y, vertex[3].x, vertex[3].y);
            line(vertex[3].x, vertex[3].y, vertex[4].x, vertex[4].y);
            line(vertex[4].x, vertex[4].y, vertex[1].x, vertex[1].y);
          end;
      end;
end;
end;

```

```
procedure init;
begin
  pcos := cos(6 * 2 * pi / 360);
  ncos := cos(-6 * 2 * pi / 360);
  psin := sin(6 * 2 * pi / 360);
  nsin := sin(-6 * 2 * pi / 360);
end;

begin
  open_graph; init;
  repeat
    display_cube(1);
    repeat
      c := readkey;
      until(uppercase(c) in ["X", "Y", "Z", "J", "K", "L", "U", "I"])
        or (c = #27);
      display_cube(0);
      case uppercase(c) of
        "Z": rotz;
        "X": rotx;
        "Y": roty;
        "J": begin
          rotx;
          roty;
        end;
        "L": begin
          rotx;
          rotz;
        end;
        "K": begin
          roty;
          rotz;
        end;
        "I": begin
          rotx;
          roty;
          rotz;
        end;
      end;
    end;
    until c = #27;
    repeat
      display_cube(1);
      delay(100);
      display_cube(0);
      rotx;
      roty;
      rotz;
    until keypressed;
    close_graph;
  end.
```

В этой программе процедуры `rotx`, `roty` и `rotz` выполняют «элементарные» повороты относительно соответствующих осей. Типизированная константа `cube` описывает куб. Это описание включает в себя описание шести граней, причем каждая грань задается пятью своими опорными точками. Углы поворота задаются в процедуре `init`. Процедура рисования куба `display_cube` перебирает все его грани и в зависимости от значения `z`-проекции нормали каждой грани рисует ее или не рисует. Отмечу использование встроенного типа `PointType`, представляющего собой запись с двумя полями типа `Integer`. Чтобы работа с этой программой была интереснее, в нее включена возможность управления поворотом кубиков с помощью нажатия на соответствующие клавиши. При первом нажатии на клавишу `Esc` управление передается второму циклу `repeat...until`, в котором кубики врачаются уже самостоятельно.

В *алгоритме художника* объект, который требуется нарисовать, также рассматривается как набор граней. Для каждой грани вводится характеристика ее удаленности от точки наблюдения. Это может быть, например, среднее арифметическое расстояний до всех ее вершин. Затем грани сортируются по удаленности и рисуются, начиная с самых удаленных и заканчивая ближними, которые закрывают удаленные грани. Этот алгоритм, очевидно, неэффективен, так как включает построение невидимых частей геометрической фигуры. По этой причине обычно используются более эффективные алгоритмы, познакомиться с которыми можно в специальной литературе.

#### Упражнение 6.1

Напишите программу, выводящую на экран цилиндр в разных проекциях и с разной ориентацией.

#### Упражнение 6.2

Напишите программу для вывода трехмерных объектов в перспективной проекции.

## Программирование динамических изображений

В программе `cubes` был реализован эффект *анимации*, то есть вывод изменяющегося, динамического изображения. Сам этот термин можно перевести как «одушевление» или «оживление» статичного (неподвижного) изображения. «Живые картинки» в компьютерной графике могут использоваться для развлечения, например в компьютерных играх. Современное кино уже трудно представить без использования спецэффектов, построенных с помощью анимации. У анимации есть и более серьезные приложения в научной и инженерной графике. При отображении результатов компьютерного моделирования в физике, химии, автомобилестроении и авиастроении часто используются приемы анимации. Ведь включение в

изобразительные средства «четвертого», временного измерения, позволяет увеличить их информативность.

В англоязычной специальной литературе используют термины *computer-assisted animation* и *computer-generated animation*. В первом случае имеется в виду подход, когда человек традиционными средствами создает ключевые кадры анимации, а компьютер «дорисовывает» все промежуточные кадры. Нас будет интересовать второй случай, в котором движущееся изображение строит сам компьютер.

Иллюзию движущегося изображения создает просмотр последовательности неподвижных изображений (кадров), показывающих разные фазы движения. Смена этих изображений должна происходить достаточно быстро. Зрительное восприятие человека обладает некоторой инерцией, вследствие чего последовательность кадров и воспринимается как плавное движение. Если скорость смены изображений чуть меньше времени реакции зрительных рецепторов, иллюзия движения сохраняется, но это движение кажется человеку прерывистым. Хорошой скоростью считается скорость показа около 30 кадров в секунду.

Чтобы вывести на экран дисплея динамическое изображение, надо запрограммировать вывод последовательности кадров. Скорость смены кадров при этом должна быть достаточно большой. В этом, собственно, и заключается основная проблема при программировании динамических изображений. Если каждый кадр содержит сложное изображение, состоящее из большого числа графических элементов, да еще со своими атрибутами, такими как условия освещения, характер отражающей поверхности и т. д., решить эту проблему непросто.

Простейший способ включения анимации в программу — обычная перерисовка графического экрана. Этот метод применяется и в программе *cubes*. Здесь мы имеем дело с простейшими изображениями, но даже в этом случае глаз замечает мерцание изображения и его «подергивание» при перерисовке.

В программе *flag* (листинг 6.14) увеличение скорости при перерисовке достигается за счет прямого обращения к памяти при выводе графических элементов. С этим приемом мы уже знакомы (материал пятого урока). Программа выводит изображение развевающегося флага. Каждый горизонтальный слой полотнища развевается по синусоиде.

#### Листинг 6.14. Программа, выводящая изображение развевающегося флага

```
program flag:  
  
uses crt, dos;  
  
const  
  spd = 1;  
  size = 3;  
  curve = 125;  
  xmax = 250 div size;  
  ymax = 150 div size;  
  sofs = 30;  
  samp = 10;
```

```
slen = 255;

type
  screenbuffertype = array[0..63999] of byte;
  screenbufferptr = ^screenbuffertype;

var
  buffer : screenbufferptr;
  screen_y : array[0..199] of word;
  stab : array[0..slen] of word;

procedure calcscreeny(width : word);
var
  i : integer;
begin
  for i := 0 to 199 do
    screen_y[i] := i * width;
end;

procedure init_graph;
var
  regs : registers;
begin
  regs.ah := $00;
  regs.al := $13;
  intr($10, regs);
  buffer := ptr($a000, 0);
  calcscreeny(320);
end;

procedure close_graph;
var
  regs : registers;
begin
  regs.ah := $00;
  regs.al := $03;
  intr($10, regs);
end;

procedure csin;
var
  i : byte;
begin
  for i := 0 to slen do
    stab[i] := round(sin(i * 4 *pi / slen) * samp) + sofs;
end;

procedure display_flag;
type
  scrarray = array[0..xmax, 0..ymax] of byte;
var
  postab : array[0..xmax, 0..ymax] of word;
```

```

bitmap : scrarray;
x, y, xp, yp, sidx : word;
begin
  sidx := 0;
  for x := 0 to xmax do
    for y := 0 to (ymax div 3) do
      bitmap[x, y] := white;
  for x := 0 to xmax do
    for y := (ymax div 3) to 2 * (ymax div 3) do
      bitmap[x, y] := lightblue;
  for x := 0 to xmax do
    for y := 2 * (ymax div 3) to ymax do
      bitmap[x, y] := lightred;
  for x := 0 to xmax do
    for y := 0 to ymax do
      postab[x, y] := 0;
repeat
  for x := 0 to xmax do
    for y := ymax downto 0 do
      begin
        buffer^[postab[x, y]] := 0;
        xp := size * x + stab[(sidx + curve * (x + y)) mod slen];
        yp := size * y + stab[(sidx + 4 * x + (curve + 1) * y) mod slen];
        postab[x, y] := xp + screen_y[yp];
        buffer^[postab[x, y]] := bitmap[x, y];
      end;
      sidx := (sidx + spd) mod slen;
      until keypressed;
end;

begin
  csin;
  init_graph;
  display_flag;
  close_graph;
end.

```

Дам краткий комментарий о назначении процедур программы `flag`. Процедуры `calcscreen`, `init_graph` и `close_graph` обеспечивают работу в графическом режиме. Это «облегченные» версии аналогичных процедур пятого урока.

Процедура `csin` формирует вспомогательный массив, содержащий синусоиду, по которой развевается горизонтальный слой флага.

Процедура `display_flag` выводит динамическое изображение развевающегося флага. Цвета флага (в нашем случае это российский «триколор») хранятся в массиве `bitmap`. Массив `postab` содержит адреса видеопамяти, соответствующие отдельным пикселям. Эффект смены кадров достигается тем, что изображение предыдущего кадра закрашивается цветом фона, а поскольку при этом используется прямое обращение к памяти, смена кадров происходит быстро. При этом создается иллюзия плавного движения полотнища.

Разберем еще один пример анимации. В этом примере используются все достижения шестого урока. Программа *rotating\_sphere* (листинг 6.15) выводит на экран изображение вращающегося «проволочного каркаса» сферы. Вначале вводятся два типа, *Point3D* и *Base2D*, первый из которых предназначен для хранения мировых координат точки, а второй — экранных. Массив *data* типа *Point3D* содержит мировые координаты точек пересечения «меридианов» и «параллелей» сферы.

**Листинг 6.15.** Программа «Вращающаяся сфера»

```
program rotating_sphere;

uses crt, graph, graphs;

type
  Point3D = record
    X, Y, Z: Real;
  end;
  Base2D = record
    U, V: Integer;
  end;

var
  data: array[0..25, 0..40] of Point3D;
  m, t1, t2 : integer;
  ch : char;
  Base : Base2D;
  E : Real;
  Perspective : Boolean;

const
  rad = Pi / 180;

procedure initialize(radius, Hor, Ver : Integer; Hx, Hy, Hz : Integer);

var
  P1, P2 : Point3d;
  A, B, hor_step, ver_step : Real;
  i, j : byte;

begin
  for i := 0 to 25 do
  begin
    for j := 0 to 40 do
    begin
      Data[i, j].x := 0.0;
      Data[i, j].y := 0.0;
      Data[i, j].z := 0.0
    end
  end;
  t1 := 0;
  hor_step := 180 / Hor;
```

```

ver_step := 360 / Ver;
for i := 0 to Hor do
begin
  B := 0;
  A := -90 + i * hor_step;
  spher_to_cartesian(radius, trunc(A), trunc(B), P1);
  Rotate(Hx, Hy, Hz, P1);
  t2 := 0;
  inc(t1);
  Data[t1, t2] := P1;
  for j := 0 to Ver do
  begin
    inc(t2);
    B := j * ver_step;
    spher_to_cartesian(radius, trunc(A), trunc(B), P2);
    Rotate(Hx, Hy, Hz, P2);
    Data[t1, t2] := P2;
  end;
end;
end;

procedure spher_to_cartesian(radius, A, B : Integer; var P : Point3D);

var
  S1, C1, S2, C2, s3, c3 : Real;

begin
  S1 := Sin(A * Rad); C1 := Cos(A * Rad);
  S2 := Sin(B * Rad); C2 := Cos(B * Rad);
  P.X := radius * C1 * C2;
  P.Y := radius * C1 * S2;
  P.Z := radius * S1
end;

procedure Rotate(AngleX, AngleY, AngleZ : Integer; var P : Point3D);

var
  S1, S2, S3, C1, C2, C3 : Real;
  Puit : Point3D;

begin
  S1 := Sin(AngleX * Rad);
  C1 := Cos(AngleX * Rad);
  S2 := Sin(AngleY * Rad);
  C2 := Cos(AngleY * Rad);
  S3 := Sin(AngleZ * Rad);
  C3 := Cos(AngleZ * Rad);
  Puit.X := P.X * (C2 * C3) + P.Y * (C2 * S3) - P.Z * S2;
  Puit.Y := P.X * (S1 * S2 * C3 - C1 * S3) + P.Y * (S1 * S2 * S3 +
  C1 * C3) + P.Z * (S1 * C2);
  Puit.Z := P.X * (C1 * S2 * C3 + S1 * S3) + P.Y * (C1 * S2 * S3 -
  S1 * C3) + P.Z * (C1 * C2);

```

```

P := PUIT
end;

procedure Rotation(x, y, z : real);
var
  S1, S2, S3, C1, C2, C3 : Real;
  P, Puit : Point3D;
  u, v : Byte;

begin
  S1 := Sin(X * Rad);
  C1 := Cos(X * Rad);
  S2 := Sin(Y * Rad);
  C2 := Cos(Y * Rad);
  S3 := Sin(Z * Rad);
  C3 := Cos(Z * Rad);
  for u := 0 to t1 do
    for v := 0 to t2 do
      begin
        P := Data[u, v];
        PUIT.X := P.X * (C2 * C3) + P.Y * (C2 * S3) - P.Z * S2;
        PUIT.Y := P.X * (S1 * S2 * C3 - C1 * S3) + P.Y * (S1 * S2 * S3 +
          C1 * C3) + P.Z * (S1 * C2);
        PUIT.Z := P.X * (C1 * S2 * C3 + S1 * S3) + P.Y * (C1 * S2 * S3 -
          S1 * C3) + P.Z * (C1 * C2);
        Data[u, v] := PUIT;
      end;
    end;
  end;

procedure Transform(Pt3D : Point3D; var X2D, Y2D : Integer);

var
  Thit : real;

begin
  if Perspective then
    begin
      Thit := 1 / (1 - Pt3D.X / E);
      X2D := trunc(Base.U + Pt3D.Y * Thit);
      Y2D := trunc(Base.V - Pt3D.Z * Thit)
    end
  else
    begin
      X2D := Base.U + Round(Pt3D.Y);
      Y2D := Base.V - Round(Pt3D.Z)
    end
  end;

procedure Line3(Point1, Point2 : Point3D; color : Integer);

var

```

```

X1, Y1, X2, Y2 : Integer;

begin
    Transform(Point1, X1, Y1);
    Transform(Point2, X2, Y2);
    LineC(X1, Y1, X2, Y2, Color)
end;

procedure LineC( x1, y1, x2, y2 : integer; color : byte);
begin
    SetColor(color); Line(x1, y1, x2, y2)
end;
procedure draw(Color : byte);
var
    u, v: Byte;
begin
    for u := 1 to t2 - 1 do
        for v := 1 to t1 - 1 do
            begin
                Line3(Data[v, u], Data[v + 1, u], Color);
                Line3(Data[v, u], Data[v, u + 1], Color)
            end;
    end;

begin
    open_graph;
    setgraphmode(vgamed);
    setbkcolor(blue);
    base.u := 320;
    base.v := 175;
    perspective := false;
    e := 150;
    m := 1;
    initialize(120, 10, 10, 30, -75, 14);
    ch := #32;
repeat
    m := 1 - m;
    setactivepage(m);
    cleardevice;
    rotation(-0.0, 0.0, 0.50);
    draw(14);
    setvisualpage(m);
    if keypressed then
        ch := readkey;
    until ch = #27;
    close_graph;
End.

```

Процедура `initialize` выполняет построение сферы, заполняя массив `data`. При этом используется вспомогательная процедура преобразования сферических координат в прямоугольные. Сферическую систему координат мы уже использовали — в ней положение точки задается длиной радиус-вектора и двумя углами.

Процедуры `rotate` и `rotation` выполняют преобразование вращения на заданные углы вокруг каждой из осей *x*, *y* и *z*. В первом случае поворот выполняется для отдельной точки, а во втором — для сферы в целом (то есть для всех ее опорных точек).

Процедура `transform` выполняет преобразование мировых координат точки в экранные. Если глобальная булева переменная `perspective` принимает значение `true`, строится перспективная проекция. В этом случае необходимо задать параметр *e* (это тоже глобальная переменная), который указывает расстояние от точки наблюдения до объекта наблюдения. Остальные процедуры предназначены для формирования изображения.

В программе `rotating_sphere` используется наличие в некоторых видеорежимах нескольких страниц видеопамяти (вспомним начало данного урока). Очередной кадр отображается на экране, а в это время на невидимой глазу странице строится следующий кадр. Затем страницы меняются ролями. Этот прием анимации мы уже использовали в программе `life` из пятого урока.

## Спрайты

Одним из распространенных приемов программирования динамических изображений является использование *спрайтов*. Особенно часто спрайты используются при программировании компьютерных игр. В литературе спрайт определяется как аппаратное или программное средство формирования динамического графического изображения. Спрайт представляет собой растровое графическое изображение небольшого размера (например, 32×32 пикселя), которое может перемещаться по экрану независимо от остального изображения. Спрайт накладывается на основное изображение, перекрывая его.

Рассмотрим пример программирования спрайтов. Прежде всего следует заметить, что со спрайтом связана структура данных такого вида:

```
type
  SpriteType = record
    Height, Width : integer;
    Data          : pointer;
  end;
```

Поля этой записи содержат информацию о размерах спрайта (его высоте и ширине) и указатель на участок памяти, содержащий изображение. Спрайт надо создавать и надо его удалять. Для выполнения этих действий используются процедуры `CreateSprite` и `KillSprite`. В первой из них предполагается, что для хранения одного пикселя отводится один байт памяти:

```
procedure CreateSprite(var Sprite : SpriteType; Width, Height : integer);

begin
  if Sprite.Data <> nil then
    KillSprite(Sprite);
```

```

    Sprite.Width := Width;
    Sprite.Height := Height;
    getmem(Sprite.Data, Width * Height);
end;

procedure KillSprite(var Sprite : SpriteType);
begin
  with Sprite do
begin
  FreeMem(Data, Width * Height);
  Width := 0;
  Height := 0;
  Data := nil;
end;
end;

```

Операции создания и удаления спрайта подразумевают выделение и освобождение памяти для размещения соответствующей данному спрайту структуры данных. Добавим к уже имеющимся процедуры записи прямоугольного фрагмента экрана в память в виде спрайта (*GetSprite*), а также его вывода на экран (*DrawSprite*):

```

procedure getsprite(var sprite : spritetyp; x1, y1, x2, y2 : integer);
var
  index, x, y : integer;
begin
  createsprite(sprite, x2-x1+1, y2-y1+1);
  index := 0;
  for y := y1 to y2 do
    for x := x1 to x2 do
      begin
        sprite.data^[index] := getpixel(x, y);
        inc(index);
      end;
end;

procedure drawsprite(var sprite : spritetyp; x, y : integer);
var
  index, xi, yi : integer;
begin
  index := 0;
  for yi := 0 to sprite.height - 1 do
    for xi := 0 to sprite.width - 1 do
      begin
        putpixel(xi + x, yi + y, sprite.data^[index]);
        inc(index);
      end;
end;

```

Первая из этих процедур просто записывает в память цвет каждого пикселя из заданной области экрана, а вторая выполняет противоположное действие. Выше перечисленные процедуры используются в программе `sprite` (листинг 6.16).

**Листинг 6.16.** Программа `sprite`

```
program sprite;

uses
  crt, graph, graphs;

type
  bytearraytype = array[0..65534] of byte;
  bytearrayptr = ^bytearraytype;
  spritetype = record
    height, width : integer;
    data          : bytearrayptr;
  end;

var
  backgroundsprite, testsprite : spritetype;
  xd, yd, ox, oy, i, x, y   : integer;

procedure killsprite(var sprite : spritetype);
...

procedure createsprite(var sprite : spritetype; width, height : integer);
...

procedure getsprite(var sprite : spritetype; x1, y1, x2, y2 : integer);
...

procedure drawsprite(var sprite : spritetype; x, y : integer);
...

begin
  open_graph;
  setgraphmode(ega64lo);
  line(10, 1, 10, 20);
  line(1, 10, 20, 10);
  line(1, 1, 20, 20);
  line(20, 1, 1, 20);
  getsprite(testsprite, 0, 0, 21, 21);
  x := 0;
  y := 0;
  xd := 1;
  yd := 1;
  while not keypressed do
  begin
    drawsprite(testsprite, x, y);
    ...
  end;
end.
```

```
inc(x, xd);
inc(y, yd);
if (x = getmaxx - 20) or (x = -20) then
    xd := -xd;
if (y = getmaxy - 20) or (y = -20) then
    yd := -yd;
end;
readln;

clearviewport;
setcolor(green);
for i := 0 to 100 do
    line(random(getmaxx), random(getmaxy), random(getmaxx), random(getmaxy));
x := 0;
y := 0;
xd := 1;
yd := 1;
while not keypressed do
begin
    drawsprite(testsprite, x, y);
    inc(x, xd);
    inc(y, yd);
    if (x = getmaxx - 20) or (x = -20) then
        xd := -xd;
    if (y = getmaxy - 20) or (y = -20) then
        yd := -yd;
end;
readln;

clearviewport;
setcolor(green);
for i := 0 to 100 do
    line(random(getmaxx), random(getmaxy), random(getmaxx),
         random(getmaxy));
x := 0;
y := 0;
xd := 1;
yd := 1;
while not keypressed do
begin
    getsprite(backgroundsprite, x, y, x + 22, y + 22);
    drawsprite(testsprite, x, y);
    ox := x;
    oy := y;
    inc(x, xd);
    inc(y, yd);
    if (x = getmaxx - 20) or (x = -20) then
        xd := -xd;
    if (y = getmaxy - 20) or (y = -20) then
        yd := -yd;
```

```
drawsprite(backgroundsprite, ox, oy);
end;
drawsprite(testsprite, x, y);
readln;
close_graph;
end.
```

В начале программы, после инициализации графического режима, строится спрайт, состоящий из четырех скрещенных линий. Это изображение сохраняется в памяти и в цикле `while not keypressed do...` выводится на экран в движении. Дополнительные условные операторы в цикле обрабатывают «соударения» спрайта с границами области вывода с учетом зеркального отражения от границ. В первом цикле движение происходит по пустому полю, во втором — по полю, заполненному хаотическим набором прямолинейных отрезков. Даже на компьютере с небольшим быстродействием движение выглядит плавным. Неприятность заключается в том, что во втором случае спрайт оставляет за собой темный след, «затирая» фоновое изображение. Третий, последний цикл показывает, что можно сделать, если это изображение представляет ценность. Предлагаемое здесь решение заключается в том, чтобы ввести второй спрайт, содержащий область фона. Двигающийся спрайт при этом становится «прозрачным», но затраты времени на перерисовку возрастают и изображение становится мерцающим. Избавиться от этого недостатка можно, оптимизируя процедуры работы со спрайтами. Решительный шаг в этом направлении — это... отказ от программирования на языке Паскаль и использование процедур, написанных на ассемблере. Турбо Паскаль допускает использование вставок ассемблерного кода, но чтобы использовать эту возможность, надо уметь программировать на ассемблере.

#### Упражнение 6.3

Переделайте программу `flag` так, чтобы выводилось изображение французского, голландского, немецкого флагов. (Учитель может дать особенно назойливому ученику задание запрограммировать движение, например, австралийского флага. Ценитель прекрасного может поместить на полотнище изображение «Данаи» Рембрандта, а любитель поесть — один из натюрмортов Снейдерса.)

#### Упражнение 6.4

Напишите программу, которая создает иллюзию быстрого движения среди звезд.

#### Упражнение 6.5

Напишите программу, выводящую случайный набор точек, который начинает вращаться по часовой стрелке при нажатии на клавишу `C` и против часовой стрелки — при нажатии на клавишу `V`.

#### Упражнение 6.6

Запрограммируйте свой вариант игры в виртуальный бильярд. Используйте приемы анимации, процедуры работы с мышью и т. д.

## Что нового мы узнали?

- Кратко познакомились с техническими средствами формирования и вывода видеоизображений.
- Рассмотрели некоторые вопросы программирования графического режима для видеоадаптера VGA.
- Рассмотрели алгоритмы построения трехмерных изображений.
- Рассмотрели некоторые приемы программирования динамических изображений.



## Рекурсия и рекурсивные алгоритмы

- 
- Рекурсия
  - Примеры программ с использованием рекурсии
  - Перебор с возвратами
  - Рекурсивные графические алгоритмы
  - Комбинаторные вычисления
-

**Н**екоторые программисты считают (и не без оснований), что рекурсия — это сердце и душа языка Паскаль. В этом уроке мы рассмотрим применение рекурсии в программах на Паскале. Здесь рассматриваются примеры рекурсивных алгоритмов и программирование комбинаторных вычислений. Приводится также пример применения алгоритмов перебора с возвратами.

## Рекурсия

Подпрограмма называется *рекурсивной*, если она вызывает саму себя. Это простейшее определение рекурсии. Рекурсивной также будет процедура, вызывающая другую процедуру, которая, в свою очередь, обращается к первой процедуре. Возможны и более сложные конструкции. В первом случае рекурсия называется *прямой*, а во втором — *косвенной*. Иногда бывает трудно написать программу по-другому, без использования рекурсии. Часто оказывается, что рекурсивные программы и выполняются быстрее, и используют меньше памяти. Очевидно, при написании рекурсивных программ следует соблюдать определенные правила предосторожности.

Следует помнить, прежде всего, что при рекурсивном вызове процедурой самой себя (или другой процедуры) ее состояние должно быть сохранено во временной области памяти. *Состояние* при этом определяется адресом, где произошла остановка вызывающей процедуры, и значением всех локальных переменных вызывающей процедуры. Программисту, как правило, специально заботиться о сохранении состояния вызывающей подпрограммы нет необходимости.

Рассмотрим пример использования рекурсивного вызова. Задача заключается в вычислении  $n$ -го по счету числа Фибоначчи. Числа Фибоначчи составляют последовательность, очередной элемент которой вычисляется по двум предыдущим значениям:

$$F_n = F_{n-1} + F_{n-2}$$

Нулевое и первое значения должны быть заданы, их значения равны единице. Последовательности такого рода применяются, например, в программных генераторах случайных чисел. Вычисление 20-го числа Фибоначчи реализовано в программе *Fibonacci* (листинг 7.1). Впрочем, номер числа можно изменить, задав в описании константы другое значение.

**Листинг 7.1.** Программа вычисления 20-го числа Фибоначчи

```
program fibonacci;
uses crt;
const
  n = 20;
function F(n : word): longint;
begin
  if keypressed then
    halt;
  if (n = 0) or (n = 1) then
    F := 1
  else
    F := F(n - 2) + F(n - 1);
end;

function G(n : word): longint;
var
  x, y, t : longint;
  k : word;
begin
  x := 1;
  y := 1;
  for k := 2 to n do
  begin
    t := y;
    y := x + y;
    x := t;
  end;
  G := y;
end;

begin
  clrscr;
  textcolor(lightgray + 128);
  write('Считаю... ');
  textcolor(lightgray);
  writeln('--Ждите ответа--');
  writeln;
  writeln('Рекурсивный алгоритм : F(', n, ') = ', F(n));
  writeln;
  writeln('Итерационный алгоритм: F(', n, ') = ', G(n));
  gotoxy(1, 1);
  cleol;
  gotoxy(1, 7);
  write('Нажмите <Enter>');
  readln;
end.
```

В этой программе реализованы два метода решения задачи вычисления числа Фибоначчи. Один назовем итерационным методом — он заключается в прямом программировании итерационной формулы для чисел Фибоначчи. В функции `G` для этого используются три вспомогательные переменные типа `LongInt`.

Решение с использованием рекурсивных вызовов запрограммировано с помощью функции `F`. Оператор, вычисляющий ее значение, два раза вызывает саму эту функцию. Текст рекурсивной функции короче, лаконичнее итерационной функции. Процедура `C1rEo1` из модуля `Crt` удаляет все символы строки от текущего положения курсора до ее конца.



### СОВЕТ

При рекурсивном программировании, во всяком случае, на первых порах, велика вероятность ошибок, которые во время выполнения программы могут даже «подвесить» компьютер. По этой причине следует соблюдать некоторые правила безопасности. Во-первых, рекомендуется компилировать программу с директивой `{$S+}`. Эта директива включает проверку переполнения стека (то есть именно той области памяти, в которой хранится состояние вызывающей подпрограммы). Если в процессе выполнения программы происходит переполнение стека, вызов процедуры или функции, откомпилированной с опцией `{$S+}`, приводит к завершению работы программы и на дисплей выводится сообщение об ошибке. Полезно также использовать директиву `{$R+}`, включающую проверку диапазона переменных. Во-вторых, можно разместить в начале каждой процедуры (функции), вызываемой рекурсивно, строку `if keypressed then Halt`. В этом случае при зависании программы вместо перезагрузки достаточно будет нажать любую клавишу.

Рассмотрим еще один пример рекурсивной программы. В первом уроке мы уже занимались решением нелинейного уравнения — приближенным вычислением корня функции (см. листинг 1.11). Сейчас мы вернемся к этой задаче, но решим ее с использованием рекурсии (программа `bisection`, листинг 7.2). В качестве функции выберем  $F(x) = x^2 - 2$ .

**Листинг 7.2.** Решение нелинейного уравнения методом деления отрезка пополам (рекурсивная версия)

```

{$S+}
program bisection;

uses crt;

type
  real = extended;

const
  eps = 1.0e-17;

```

```

var
  x0, x1, midpt : real;
  depth : word;

function F(x : real): real;
begin
  F := x * x - 2.0;
end;

procedure bisect(var x0, x1 : real);
begin
  if keypressed then
    halt;
  inc(depth);
  midpt := 0.5 * (x0 + x1);
  if F(x1) - F(x0) < eps then
    exit;
  if F(midpt) >= 0.0 then
    x1 := midpt
  else
    x0 := midpt;
  bisect(x0, x1);
end;

begin
  depth := 0;
  x0 := 0.0;
  x1 := 5.0;
  bisect(x0, x1);
  clrscr;
  writeln('Решение уравнения F(x) = 0 приблизительно ', midpt:22:18);
  writeln('Глубина рекурсии ', depth);
  writeln('Нажмите <Enter>');
  readln;
end.

```

Эта программа не нуждается в подробных комментариях. Здесь учтены все те советы, касающиеся «правил безопасности», которые давались ранее. Константа `eps` определяет точность вычисления корня функции. В качестве эксперимента можно задать значение константы `eps = 1.0e-19`. Результат будет плачевый, но не фатальный — аварийный останов программы с сообщением о переполнении стека. Переполнение стека в этом случае вызвано тем, что для достижения требуемой точности требуется слишком большое число рекурсивных вызовов. Программа выводит не только приближенное значение корня, но и количество рекурсивных вызовов (*глубину рекурсии*).

В следующем примере (программа `gcd`, листинг 7.3) запрограммирован алгоритм Евклида нахождения наибольшего общего делителя двух целых чисел. В этом алгоритме используются только операции вычитания. Пусть имеются два целых положительных числа *m* и *n*. Тогда наибольший общий делитель (Greatest Common Divisor) находится с помощью следующего рекурсивного алгоритма:

$m, n > 0$ ,

$$\text{gcd}(m, n) = \begin{cases} \text{gcd}(n, m), & m < n; \\ n, & m = 0; \\ \text{gcd}(m - n, n), & m > n. \end{cases}$$

Здесь вновь предлагаются два варианта — один рекурсивный, а другой итерационный. Имеется также переменная-счетчик, предназначенная для вывода сообщений о глубине рекурсивных вызовов.

Листинг 7.3. Программа вычисления наибольшего общего делителя

```
{S+}
program gcd;

uses crt;
var
  m, n : longint;
  depth : word;

function gcd_rec(m, n : longint): longint;
begin
  inc(depth);
  if n = 0 then
    gcd_rec := m
  else
    if m < n then
      gcd_rec := gcd_rec(n, m)
    else
      gcd_rec := gcd_rec(m - n, n);
end;

function gcd_iter(m, n : longint): longint;
var
  t : longint;
begin
  while n > 0 do
  begin
    if m < n then
    begin
      t := m;
      m := n;
      n := t;
    end;
    t := m - n;
    m := n;
    n := t;
  end;
  gcd_iter := m;
end;
```

```

begin
  clrscr;
  depth := 0;
  writeln('Рекурсивная функция : ', gcd_rec(200, 62140));
  writeln('Глубина рекурсии : ', depth);
  writeln('Итерационная функция : ', gcd_iter(200, 62140));
  writeln;
  writeln('Нажмите <Enter>');
  readln;
end.

```

Вновь хотелось бы обратить внимание на то, что решение с использованием рекурсивного вызова функции оказывается короче.

В тех примерах, которые мы успели рассмотреть, у нас все-таки была свобода выбора — использовать рекурсию или другие, пока что более привычные нам, методы. В следующем примере использование рекурсии — единственный способ решения. Задача состоит в том, чтобы вычислить элементы последовательности, определенной следующим образом:

$$\begin{aligned}a(1) &= 1, \\a(n) &= n - a(a(n - 1)), \quad n > 1.\end{aligned}$$

Попробуйте-ка в качестве развлечения вычислить первый десяток элементов этой последовательности вручную! Программа seq\_1 (листинг 7.4) строит таблицу из первых 300 элементов последовательности.

#### Листинг 7.4. Программа seq\_1

```

program seq_1;

const
  n0 = 300;

var
  i, j, k, n : word;

function a(n : word): word;
begin
  if n = 1 then
    a := 1
  else
    a := n - a(a(n - 1));
end;

begin
  n := 1;
  for i := 0 to (n0 div 150) - 1 do
  begin
    write(' ': 5);
    for k := 1 to 15 do
      write(k:4);

```

```
writeln;
for j := 0 to 9 do
begin
    write(150 * i + 15 * j : 3, ' ');
    for k := 1 to 15 do
    begin
        write(a(n):4);
        inc(n);
    end;
    writeln;
end;
writeln;
end;
end.
```

Программа кажется довольно простой. Но это с использованием рекурсивных вызовов функции. Попробуйте написать аналогичную программу без рекурсии!

Вот еще один вариант этой программы (листинг 7.5). Программа seq\_2 выполняется быстрее, чем предыдущая, и связано это с тем, что в программе seq\_1 используется рекурсивный вызов функции, а в программе seq\_2 — рекурсивное обращение к элементам массива a. В первом случае больше «накладные» расходы на организацию рекурсивного вызова функции.

#### Листинг 7.5. Программа seq\_2

```
program seq_2;

const
  n0 = 300;

var
  i, j, k, n : word;
  a: array[1..n0] of word;

procedure fill_a;
begin
  a[1] := 1;
  for n := 2 to n0 do
    a[n] := n - a[a[n - 1]];
end;

begin
  fill_a;
  n := 1;
  for i := 0 to (n0 div 150) - 1 do
  begin
    write(' ': 5);
    for k := 1 to 15 do
      write(k: 4); writeln;
    for j := 0 to 9 do
    begin
```

```

write(150 * i + 15 * j : 3, ' ');
for k := 1 to 15 do
begin
  write(a[n]:4);
  inc(n);
end;
writeln;
end;
writeln;
end;
readln;
end.

```

**Упражнение 7.1**

Определите (с помощью карандаша и бумаги) результат выполнения следующей программы:

```

program what_is;

procedure explain_me(n : Word);
begin
  Write(n:4);
  if n < 150 then
    explain_me(2 * n);
  Write(n:4);
end;

begin
  explain_me(1);
  writeln;
  Write('Нажмите <Enter>');
  ReadLn;
end.

```

**Упражнение 7.2**

Запрограммируйте с использованием рекурсии вычисление функции  $F(x) = x^n$ .

**Упражнение 7.3**

Напишите рекурсивную процедуру для решения уравнений вида  $F(x) = x$  методом простых итераций. Проверьте ее работу на функциях  $\cos(x)$  и  $\sqrt{x+1}$ .

**Упражнение 7.4**

Напишите рекурсивную процедуру для вычисления значения полинома Лежандра порядка  $n$  в точке  $x$ . Полиномы Лежандра определяются следующим образом:

$$\begin{aligned}
 P_0(x) &= 1, \\
 P_1(x) &= x, \\
 P_n(x) &= ((2n-1)P_{n-1}(x) - (n-1)P_{n-2}(x))/n.
 \end{aligned}$$

# Примеры программ с использованием рекурсии

Начну с вопроса. О чём вам, читатель, говорит, например, такой набор символов: **MCMXCIX**? Может быть, ничего, а может быть, вы догадались, что это некое число, записанное римскими цифрами. Но вот какое — далеко не каждый сможет определить. Поручим перевод числа из одной системы записи в другую компьютеру.

Рассмотрим более простую задачу о переводе арабских чисел в римские. Программа `arab_to_roman` решает эту задачу. Ее исходный текст приведен в листинге 7.6.

**Листинг 7.6.** Программа перевода арабских чисел в римские

```
{$S+,R+}
program arab_to_roman;

uses crt;

var
  n : word;
  ch : char;

procedure convert(n : word);
const
  no = 12;
  index : array[0..no] of word =
    (1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000);
  strings : array[0..no] of string[2] =
    ('I', 'IV', 'V', 'IX', 'X', 'XL', 'L', 'XC', 'C', 'CD', 'D', 'CM', 'M');

  var
    i : word;

begin
  if n = 0 then
    exit;
  if keypressed then
    halt;
  if n >= index[no] then
  begin
    write(strings[no]);
    n := n - index[no];
  end
  else
  begin
    i := no;
    repeat
```

```

    if keypressed then
        halt;
        dec(i);
    until (n >= index[i]) and (n < index[i+1]);
    write(strings[i]);
    n := n - index[i];
end;
convert(n);
end;

begin
clrscr;
repeat
    write('Введите положительное целое число: ');
    readln(n); write(n, ' = ');
    convert(n);
    writeln;
    writeln('Чтобы завершить работу, нажмите Q:');
    writeln('Для продолжения работы нажмите любую другую клавишу:');
    ch := upcase(readkey);
    if ch = 'Q' then
        break;
    writeln;
    writeln;
until false;
end.

```

Массивы `index` и `strings` устанавливают соответствие между двенадцатью «опорными» значениями в арабской и римской записях. Идея алгоритма заключается в том, чтобы вначале выделить во введенном числе составную часть, равную наибольшему опорному значению. На экран выводятся соответствующие символы римской записи. Эта часть вычитается, и поиск продолжается для остатка. В алгоритме используется поиск наибольшего значения элемента массива арабских чисел `index`, не превышающего значения введенного числа `n`.

## Разностные уравнения

Так уж получается, что большая часть программ этого урока предназначена для решения тех или иных математических задач. Вот и следующая программа предназначена для решения так называемого *разностного уравнения*. Разностные уравнения — это естественные кандидаты на применение приемов рекурсивного программирования. Рассмотрим, например, такое уравнение:

$$A(0) = 1,$$

$$A(n) = A(n \text{ div } 2) + A(n \text{ div } 3).$$

Простого и очевидного способа итерационного программирования функции  $A(n)$  не существует. Здесь вновь на помощь приходит рекурсия (программа `diff_eq`, листинг 7.7).

**Листинг 7.7. Программа diff\_eq**

```

program diff_eq;

uses crt;

var
  k, n : longint;
  row, col, depth, max_depth : word;

function A(n : longint): longint;
begin
  if keypressed then
    halt;
  inc(depth);
  if depth > max_depth then
    inc(max_depth);
  if n = 0 then
    A := 1
  else
    A := A(n div 2) + A(n div 3);
  dec(depth);
end;

begin
  repeat
    max_depth := 0;
    col := 0;
    writeln;
    writeln('Введите положительное целое число или 0
для завершения работы');
    readln(n);
    if n = 0 then
      halt;
    for k:=1 to n do
    begin
      depth := 0;
      writeln('A(', k, ') = ', A(k), ' Глубина рекурсии = ', max_depth);
    end;
    writeln;
    until false;
end.

```

При запуске программы `diff` необходимо ввести положительное число, и на экран будут выведены значения решения разностного уравнения от 1 до указанного номера.

**Упражнение 7.5**

Пусть в алгебраической записи выражения имеется всего одна операция, умножение, обозначаемое обычным образом (два сомножителя следуют непосредственно друг за другом). Выражение состоит из строки символов и скобок-ограничителей: () [] и {}. Напишите программу, которая выполняет проверку на

предмет соответствия закрывающих и открывающих скобок. Например, запрещены выражения типа  $(ab)$  или  $a(b[c)d)$ .

### Упражнение 7.6

Функция Аккермана определяется следующим образом:

$$A(0, y) = y + 1,$$

$$A(x, 0) = A(x - 1, 1),$$

$$A(x, y) = A(x - 1, A(x, y - 1)).$$

Здесь  $x$  и  $y$  — целые неотрицательные числа. Функция возрастает настолько быстро, что вскоре «выбивает» из работы любой компьютер. Определим «модульную функцию Аккермана» как  $A \bmod m$ , где значение параметра  $m$  вводится. Постройте таблицу значений этой функции.

### Упражнение 7.7

Напишите программу для решения головоломки «Ханойская башня». Правила этой головоломки таковы. Имеется доска с тремя колышками. На первом из них нанизано несколько дисков убывающего диаметра (самый большой находится внизу). Требуется расположить диски в том же порядке на другом колышке, причем диски можно перекладывать с колышка на колышек по одному, а класть больший диск на меньший не разрешается.

## Перебор с возвратами

В этом параграфе мы познакомимся с примером рекурсивного алгоритма. Сразу признаюсь, что возможности рекурсивных вызовов подпрограмм не будут использованы в его программной реализации. Тем не менее, алгоритм (точнее говоря, целое семейство алгоритмов) достоин хотя бы беглого с ним знакомства. Любителю головоломок и игр могу предложить такое определение — *алгоритмы перебора с возвратами* представляют собой эффективные итерационные алгоритмы прохождения лабиринтов. Идея таких алгоритмов состоит в следующем. Предположим, что из каждого узла лабиринта можно пойти налево, прямо или направо. Возможно также, что мы оказались в тупике. В методе перебора с возвратами начинают с выбора левого направления и двигаются так до тех пор, пока не достигнут тупика. Всякий раз при достижении тупика отступают на один шаг назад. Если до этого был сделан шаг влево, то следующая попытка делается в прямом направлении. Если шли прямо, придется повернуть направо. Если шли вправо, то отступаем на один шаг и повторяем всю последовательность действий заново.

Можно дать этому алгоритму более строгое определение. Перебор с возвратами — способ поиска (например, по дереву решений), когда при возврате после рассмотрения «пробного» варианта решения на один шаг назад все переменные программы восстанавливают свои значения.

Применим перебор с возвратами для решения известной задачи о расстановке восьми ферзей. Требуется разместить 8 ферзей на шахматной доске таким образом, чтобы ни один из них не нападал на любого другого. Всего у этой задачи 92 решения, правда, некоторые из них, как говорят математики, «эквивалентны в силу симметрии шахматной доски относительно преобразований вращения и отражения». Учет симметрии оставит всего 12 неэквивалентных решений. Мы симметрию учитывать не будем и честно найдем все решения данной задачи (листинг 7.8, программа `eight_queens`).

Как известно, шахматная доска содержит 8 вертикалей и 8 горизонталей. Присвоим им номера от 0 до 7 и определим массивы. Массивы, используемые в программе, имеют следующее назначение. Первый, `solution[col] = row`, если ферзь находится на поле `(row, col)`, где `row` — номер горизонтали, а `col` — номер вертикали. Элемент массива `Q_row[row]` принимает значение `true` тогда и только тогда, когда имеется ферзь на горизонтали `row`. Элементы массивов `Q_up_diag[diag]` и `Q_down_diag[diag]` принимают значение `true`, если имеется ферзь на диагоналях с соответствующим номером. Если обе диагонали, горизонталь и вертикаль, проходящие через некоторое поле, заняты, ставить ферзя на это поле уже нельзя. Массив `solution` для каждой вертикали содержит номер занятой диагонали.

Будем просматривать шахматную доску от нулевой до седьмой вертикали. При переходе к очередной вертикали имеется 8 вариантов размещения ферзя: от нулевой до седьмой горизонтали. Для каждой из горизонталей, в свою очередь, производится проверка, не находится ли поле «под боем», пока не будет найдено правильное положение ферзя на этой вертикали, иначе происходит возврат.

#### Листинг 7.8. Программа «Восемь ферзей»

```
program eight_queens;
uses crt;
const
  max = 7;
  m = 2 * max;

type
  vector = array[0..max] of word;
var
  solution : vector;
  Q_row : array[0..max] of boolean;
  Q_up_diag : array[-max..max] of boolean;
  Q_down_diag : array[0..m] of boolean;
  done : boolean;
  row, col, diag : integer;
  total, count : word;
  out_file : file of vector;

procedure remove_queen;
begin
```

```
Q_row[row] := false;
Q_down_diag[col + row] := false;
Q_up_diag[col - row] := false;
end;
```

```
procedure backtrack;
begin
  if keypressed then
    halt;
  dec(col);
  row := solution[col];
  while (row = max) and (col > 0) do
  begin
    remove_queen;
    dec(col);
    row := solution[col];
  end;
  if row < max then
  begin
    remove_queen;
    inc(row);
  end
  else
    done := true;
end:
```

```
procedure print;
begin
  write('Решение ', count:2, ': ');
  write('[');
  for col := 0 to max do
    write(solution[col]:3);
  writeln(']'':3);
  write(out_file, solution);
end:
```

```
begin
  assign(out_file, '8queens.dat');
  rewrite(out_file);
  clrscr;
  total := 0;
  count := 0;
  writeln('Все решения задачи о ', max + 1, ' ферзях');
  writeln;
```

```
done := false;
for row := 0 to max do
  Q_row[row] := false;
for diag := -max to max do
  Q_up_diag[diag] := false;
for diag := 0 to m do
  Q_down_diag[diag] := false;
```

```

row := 0;
col := 0;

repeat
    if Q_row[row] or Q_down_diag[col + row] or
        Q_up_diag[col - row] then
        if row = max then
            backtrack
        else
            inc(row)
    else
        begin
            solution[col] := row;
            Q_row[row] := true;
            Q_down_diag[col + row] := true;
            Q_up_diag[col - row] := true;
            if col = max then
                begin
                    inc(total);
                    inc(count);
                    print;
                    remove_queen;
                    backtrack;
                end
            else
                begin
                    inc(col);
                    row := 0;
                end;
        end;
    until done;

writeln;
writeln('Число решений: ', total);
writeln;
write('Нажмите <Enter>: ');
readln;
close(out_file);
end.

```

Процедура `remove_queen` производит удаление ферзя, занося в массивы `Q_row`, `Q_down_diag` и `Q_up_diag` значения `false`.

Процедура `backtrack` является программной реализацией алгоритма перебора с возвратами. Первый цикл `while...do` выполняет удаление ферзя и переход к предыдущей вертикали. Условный оператор предназначен для удаления ферзя и перехода на следующую горизонталь, оставаясь на той же вертикали.

Процедура `print` выводит найденное решение на экран и в файл `8queens.dat`. Каждое решение выводится в виде массива 8 целых значений — каждое значение представляет собой номер занятой горизонтали для соответствующей вертикали.

При запуске программы выполняется инициализация массивов и открывается для записи файл 8queens.dat. Затем в работу вступает цикл `repeat...until`, в котором производится расстановка ферзей. Первый условный оператор применяет алгоритм перебора с возвратами до тех пор, пока не окажется, что на вертикалях, горизонталях и обеих диагоналях, проходящих через рассматриваемое поле, нет ферзей. В этом случае поле занимается и происходит переход к следующей вертикали.

#### Упражнение 7.8

Программа `queens` создает файл 8queens.dat, содержащий решения задачи о 8 ферзях. Напишите графическую программу для отображения этих решений.

## Рекурсивные графические алгоритмы

В этом параграфе мы займемся рекурсивным построением геометрических узоров. Программы, описываемые здесь, основаны на идеях «черепашьей графики». «Черепаха» — это вектор с началом в текущем указателе, показывающий направление очередного относительного перемещения из текущей точки и оставляющий при перемещении позади себя след. Что можно делать с «черепахой»? Ее можно поворачивать на заданный угол и перемещать на заданное расстояние.

Начнем с рекурсивной программы рисования кривой Коха (см. листинг 7.9). В ней используются только повороты на  $+90^\circ$  и  $-90^\circ$  и смещения вдоль прямолинейных отрезков. Алгоритм рисования кривой Коха можно записать следующим образом:

- нарисовать прямолинейный отрезок заданной длины;
- повернуть на  $+90^\circ$ ;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на  $-90^\circ$ ;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на  $-90^\circ$ ;
- нарисовать прямолинейный отрезок заданной длины;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на  $+90^\circ$ ;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на  $+90^\circ$ ;
- нарисовать прямолинейный отрезок заданной длины;

- повернуть на  $-90^\circ$ ;
- нарисовать прямолинейный отрезок заданной длины.

Затем эта последовательность действий вновь применяется к каждому прямолинейному участку построенной ломаной линии. В бесконечном по числу повторений пределе получим кривую Коха — так называемый *фрактал*. Интересной особенностью фрактальных объектов является повторение особенностей их структуры при переходе к любому другому масштабу. Для непосвященного в математические тайны человека фракталы — просто красивые геометрические узоры.

#### Листинг 7.9. Программа «Кривая Коха»

```
program koch;
uses crt, graph, graphs;
var
  ch : char;
  min : word;
procedure rotate_plus(var x, y : integer);
var
  t : integer;
begin
  t := x;
  x := y;
  y := -t;
end;
procedure rotate_minus(var x, y : integer);
var
  t : integer;
begin
  t := x;
  x := -y;
  y := t;
end;
procedure draw(x, y : integer; length : word);
var
  t : word;
begin
  if keypressed then
    exit;
  if length >= min then
    begin
      if length >= 32 then
        setlinestyle(solidln, 0, thickwidth)
      else
        setlinestyle(solidln, 0, 1);
      line(x, y, x + t * cos(ch), y + t * sin(ch));
      draw(x + t * cos(ch), y + t * sin(ch), length / 3);
      draw(x + t * cos(ch) * 2, y + t * sin(ch) * 2, length / 3);
    end;
end;
begin
  min := 10;
  repeat
    ch := readkey;
    if ch = 'q' then
      exit;
    if ch = 'l' then
      draw(100, 100, 100);
    if ch = 'r' then
      draw(100, 100, 100);
    if ch = 'd' then
      draw(100, 100, 100);
    if ch = 'u' then
      draw(100, 100, 100);
  until false;
end.
```

```

    setlinestyle(solidln, 0, normwidth);
t := length div 4;
setcolor(white);
draw(x, y, t);
setcolor(yellow);
rotate_plus(x, y);
draw(x, y, t);
setcolor(magenta);
rotate_minus(x, y);
draw(x, y, t);
rotate_minus(x, y);
setcolor(red);
draw(x, y, t);
setcolor(cyan);
draw(x, y, t);
rotate_plus(x, y);
setcolor(green);
draw(x, y, t);
setcolor(blue);
rotate_plus(x, y);
draw(x, y, t);
setcolor(lightgray);
rotate_minus(x, y);
draw(x, y, t);
end
else
  LineRel(length * x, length * y);
end;

begin
open_graph;
min := 256;
while min >= 4 do
begin
  moveto(20, 200);
  draw(1, 0, 512);
  outtextxy(200, 460, 'Press <Space>');
  ch := readkey;
  min := min div 4;
  cleardevice;
end;
close_graph;
end.

```

Прямолинейные отрезки, начинающиеся из текущей точки и оканчивающиеся в точке с заданными координатами, в программе koch рисуются процедурой `LineRel` модуля `Graph`. Процедура `draw(x, y : Integer; Length : Word)` является рекурсивной, она строит кривую Коха. Параметрами этой процедуры являются `x` и `y` (эти два числа задают единичный вектор в направлении очередного смещения) и величина смещения `length`.

Следующая программа (Cantor, листинг 7.10) строит двумерное множество Кантора. Множество Кантора строится следующим образом. Имеется квадрат, внутренняя часть которого закрашена каким-то цветом. Этот квадрат делится на 16 равных частей (тоже квадратных). Затем удаляются 4 средних квадрата, причем изображения их границ остаются. Далее процедура повторяется для каждого оставшегося квадрата.

**Листинг 7.10.** Программа «Множество Кантора»

```
program cantor;

uses crt, graph, graphs;

var
    ch : char;

procedure draw(x, y : integer; size : word);
var
    s : word;

procedure solid_rectangle(x, y, size : integer);
begin
    rectangle(x - size, y - size, x + size, y + size);
    bar(x - size + 1, y - size + 1, x + size - 1, y + size - 1);
end;

begin
    if size > 1 then
    begin
        s := size div 2;
        draw(x - size, y + size, s);
        draw(x - size, y - size, s);
        draw(x + size, y + size, s);
        draw(x + size, y - size, s);
    end;
    solid_rectangle(x, y, size);
end;

begin
    open_graph;
    setfillstyle(solidfill, black);
    setcolor(yellow);
    draw(getmaxx div 2, getmaxy div 2, getmaxy div 4);
    outtextxy(265, 235, 'Press <Enter>');
    readln;
    close_graph;
end.
```

Следующий знаменитый геометрический объект — кривая Пеано. Строится она так же, как и предыдущие, — имеется базисная форма, и на каждом шаге каждый элемент этой формы заменяется базисной формой с соответствующим изменением масштаба. Для кривой Пеано базисная форма похожа на квадратную букву Z или двойку, отображаемую на жидкокристаллическом дисплее (рис. 7.1).

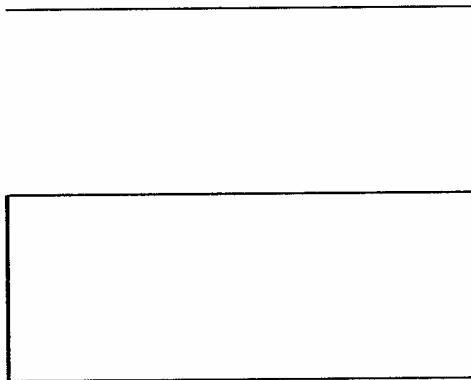


Рис. 7.1. Базисная форма для кривой Пеано

«Черепаший» алгоритм построения базисной формы можно записать в виде следующей последовательности действий:

- нарисовать прямолинейный отрезок заданной длины;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на +90°;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на +90°;
- нарисовать прямолинейный отрезок заданной длины;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на -90°;
- нарисовать прямолинейный отрезок заданной длины;
- повернуть на -90°;
- нарисовать прямолинейный отрезок заданной длины;
- нарисовать прямолинейный отрезок заданной длины.

Обратной по отношению к этой форме будет фигура, похожая на квадратную букву S. Это еще одна форма, используемая при построении кривой Пеано. Кривая Пеано степени  $n$  состоит из трех рядов Z- и S-форм, соединенных вместе. Мы можем включить рисование Z и S в одну процедуру, добавив булеву переменную, описывающую направление движения. Программной реализацией этих идей является рекурсивная программа `peano` (листинг 7.11), ключевую роль в которой играет процедура `draw_Z`. Параметр  $n$  задает порядок кривой.

**Листинг 7.11.** Программа «Кривая Пеано»

```
program peano;

uses crt, graph, graphs;

var
  ch : char;
  x, y : integer;
procedure draw_Z(n : word; plus : boolean);

procedure rotate(plus : boolean);

var
  t : integer;

begin
  if plus then
    begin
      t := x;
      x := -y;
      y := t;
    end
  else
    begin
      t := x;
      x := y;
      y := -t;
    end;
  end;

begin
  if n > 0 then
    begin
      draw_Z(n - 1, plus);
      linerel(x, y);
      draw_Z(n - 1, not plus);
      linerel(x, y);
      draw_Z(n - 1, plus);
      rotate(plus);
      linerel(x, y);
      rotate(plus);
      draw_Z(n - 1, not plus);
      linerel(x, y);
      draw_Z(n - 1, plus);
      linerel(x, y);
      draw_Z(n - 1, not plus);
      rotate(not plus);
      linerel(x, y);
      rotate(not plus);
      draw_Z(n - 1, plus);
      linerel(x, y);
    end;
end;
```

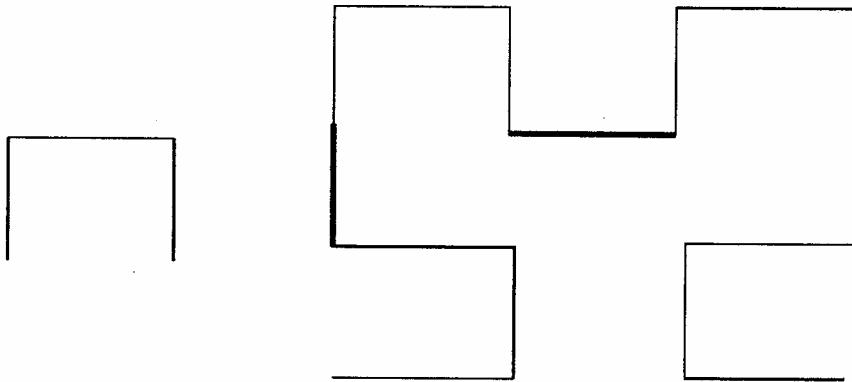
```

draw_Z(n - 1, not plus);
lineref(x, y);
draw_Z(n - 1, plus);
end;
end;

begin
open_graph;
moveto(0, 0);
x := 5;
y := 0;
draw_Z(4, true);
outtextxy(500, 460, 'Press <Enter>');
readln;
close_graph;
end.

```

Рекурсию можно использовать и для построения кривой Гильберта. Базисной формой кривой Гильберта является квадратная буква П. Кривая Гильберта первого порядка совпадает с базисной формой. Построение кривой второго порядка показано на рис. 7.2.



**Рис. 7.2.** Построение кривой Гильберта второго порядка

Кривая второго порядка изображена справа. Толстыми линиями изображены линии-связки (на реальном рисунке они не выделяются). Базисные формы изображены более тонкими линиями. Каждая сторона мёньшей буквы П имеет ту же длину, что и связка, то есть она в три раза меньше стороны квадрата, в который вписывается кривая второго порядка. Как запрограммировать вывод кривой Гильберта произвольного порядка? Попробуйте выполнить это в качестве упражнения.

#### Упражнение 7.9

Запрограммируйте вывод кривой Гильберта произвольного порядка.

# Комбинаторные вычисления

Программирование комбинаторных вычислений — большая и серьезная тема. Здесь мы лишь бегло познакомимся с применением рекурсии для решения некоторых комбинаторных задач. Что же такое комбинаторика? Предлагаю свое определение этой науки. Имеется набор предметов, которым присвоены номера. Эти предметы можно разложить в определенном или случайном порядке. Можно выбрать из всего набора лишь некоторые предметы и т. д. Математическое описание таких действий и составляет предмет комбинаторики.

Из всех комбинаторных операций здесь мы будем заниматься только *сочетаниями*. Если имеется набор из  $n$  предметов (множество, если говорить более строго), то количество меньших наборов (подмножеств), которые можно составить из  $k$  предметов, описывается величиной

$$C_n^k = \binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Эта величина произносится как «число сочетаний из  $n$  по  $k$ ». Должно выполняться условие  $n \leq k$ . Известна следующая формула:

$$C_n^0 = 1, \quad C_n^k = C_{n-1}^{k-1} + C_{n-1}^k.$$

Таблица значений этой величины называется «треугольником Паскаля», так как длина ее строк увеличивается с ростом номера строки. Треугольник Паскаля неизбежно должен был появиться в книге, посвященной программированию на языке Паскаль! В программе Pascal (листинг 7.12) напрямую запрограммирована вышеприведенная формула.

**Листинг 7.12.** Программа «Треугольник Паскаля»

```
{$S+}
program Pascal;

uses crt;

var
  n, k : word;

function n_choose_k(n, k : word): longint;
begin
  if keypressed then
    halt;
  if k = 0 then
    n_choose_k := 1
  else
    if k = n then
```

```

n_choose_k := 1
else
  n_choose_k := n_choose_k(n - 1, k - 1) +
    n_choose_k(n - 1, k);
end;

begin
  clrscr;
  for n := 0 to 12 do
  begin
    write(n_choose_k(n, 0):(40 - 3 * n));
    for k := 1 to n do
      write(n_choose_k(n, k):6);
    writeln;
  end;
  writeln;

  n := 16;
  k := 10;
  writeln(' Ждите - идут вычисления для числа сочетаний из ',
    n, ' по ', k);
  writeln(' Ответ ', n_choose_k(n, k));
  writeln;
  write('Нажмите <Enter>:');  readln;
end.

```

Более сложной задачей, чем подсчет числа подмножеств, является их перечисление. Принадлежность  $k$ -го элемента множества некоторому подмножеству будем описывать с помощью массива  $S$ . Если этот элемент принадлежит подмножеству, значение соответствующего элемента массива устанавливается в единицу, в противном случае оно равно нулю. Задача заключается в том, чтобы найти все подмножества множества целых чисел  $\{1..m\}$ . Рассмотрим рекурсивную программу subsets решения этой задачи (листинг 7.13).

#### Листинг 7.13. Программа subsets

```

program subsets;

uses crt;

const
  m = 5;

var
  S : array[1..m] of 0..1;
  count : word;

procedure initialize;
var
  k : word;

```

```
begin
  for k := 1 to m do
    S[k] := 0;
  writeln('Номер', ':4, 'Код':m, ':4, 'Подмножество');
  count := 0;
end;

procedure print(direct : boolean);
var
  j, k, size : word;

begin
  size := 0;
  inc(count);
  write(count:5, ':4);
  if direct then
    for k := 1 to m do
    begin
      write(S[k]);
      size := size + S[k];
    end
  else
    for k := m downto 1 do
    begin
      write(S[k]);
      size := size + S[k];
    end;
  write(' ':4, '{');

  if size = 0 then
    writeln('}')
  else
begin
  j := 0;
  k := 1;
  while j < size do
  begin
    if S[k] = 1 then
    begin
      inc(j);
      write(k);
      if j < size then
        write(', ');
    end;
    inc(k);
  end;
  writeln('}');
end;
if count mod 20 = 0 then
begin
  write('Для продолжения нажмите <Enter>: ');
end;
```

```

readln;
end;
end;

procedure subset(n : word);

begin
  if n = 0 then
    print(false)
  else
    begin
      S[n] := 1;
      subset(n - 1);
      S[n] := 0;
      subset(n - 1);
    end;
end;

begin
  clrscr;
  initialize;
  subset(m);
  write('Для завершения работы нажмите <Enter>: ');
  readln;
end.

```

Процедура `initialize` присваивает начальные значения элементам массива `S` и выводит заголовок таблицы.

Процедура `print` предназначена для вывода результатов работы основной программы. В качестве упражнения самостоятельно разберите ее работу.

Процедура `subset` — сердце нашей программы. Здесь используется достаточно «прозрачная» рекурсия. В список сначала вносятся все подмножества, содержащие  $n$ , а затем те, которые не содержат этот элемент. Далее следуют рекурсивные вызовы процедуры.

Процедуру `subset` можно сократить на один оператор, если записать ее в следующем виде:

```

procedure subset(n : Word);

begin
  if n = 0 then
    print(false)
  else
    begin
      subset(n - 1);
      S[n] := 1 - S[n];
      subset(n - 1);
    end;
end;

```

## Что нового мы узнали?

- Познакомились с использованием в программах на Паскале рекурсивных вызовов подпрограмм.
- Познакомились с алгоритмами перебора с возвратами и их применением.
- Рассмотрели программирование рекурсивных графических алгоритмов.
- Рассмотрели применение рекурсии в программировании комбинаторных вычислений.



## Программирование звучка

- 
- Использование встроенного динамика
  - Программа «Виртуальное пианино»
  - Программирование для SoundBlaster
-

**В** этом уроке мы будем заниматься программированием звука. Вначале рассмотрим, как программируется вывод звука для встроенного динамика. Затем познакомимся с основными принципами программирования для звуковой карты. Для чтения материала данного урока полезным (но не обязательным) является знание основ музыкальной грамоты.

## Использование встроенного динамика

Невозможно представить себе современный персональный компьютер без устройств вывода звука. Нормой стало оснащение компьютера специальным устройством — звуковой картой, позволяющей творить со звуком самые настоящие чудеса. «Озвученный» компьютер — это почти полноценный музыкальный центр. Он дает пользователю возможность превратиться в исполнителя на различных музыкальных инструментах, он может даже стать неплохим рассказчиком!

Свое знакомство с программированием звука на Турбо Паскале мы начнем с использования встроенного динамика. Прежде всего замечу, что модуль `Crt` содержит две процедуры, предназначенные для работы с динамиком. Первая из них — `Sound(Hz)` — включает динамик на звуковой частоте, задаваемой значением параметра `Hz` (типа `Word`) в герцах. Динамик генерирует звук до тех пор, пока он не будет отключен вызовом процедуры `NoSound`, не имеющей параметров. Процедура `Sound` обеспечивает довольно бедную интонацию, потому что она использует только целочисленные частоты. Да и встроенный динамик, к сожалению, дает звук самого низкого качества. Он не допускает управления громкостью, очень по-разному резонирует на разных частотах. Не ждите от него многоного! Тем не менее, мы постараемся хотя бы частично исправить положение и разработаем свои собственные процедуры для модуля `Speaker`.

Прежде всего, познакомимся поближе с работой встроенного динамика. Источником звуковых акустических колебаний является его диффузор. Он может находиться в одном из двух возможных положений. Для управления работой динамика ему посылаются прямоугольные импульсы определенной частоты. В

течение первой половины каждого цикла диффузор перемещается в одно положение, а затем, в течение второй половины цикла, в другое. Это вызывает движение воздуха. В процессе генерации акустических колебаний прямоугольный импульс, который порождает неприятно звучащую ноту, несколько сглаживается. Так возникает звуковой сигнал.

Теперь несколько слов о технических деталях. Компьютер содержит микросхему 8255, которую называют «программируемым периферийным интерфейсом» (PPI – Programmable Peripheral Interface). PPI управляет работой динамика, используя микросхему таймера 8254 (PIT – Programming Interval Timer). Микросхема 8255 имеет различные порты, включая порт ввода/вывода \$61 размером («шириной») в один байт с битами b7..b0. Запись впорт \$61 нулевого либо единичного значения бита b0 позволяет установить режим управления динамиком. Если b0 = 1, то динамиком управляет второй канал микросхемы таймера. Микросхема таймера имеет 3 канала, каждый из которых предназначен для решения своих задач. Нулевой канал обслуживает системные часы, первый канал обслуживает микросхему прямого доступа к памяти (DMA). Второй канал связан со встроенным динамиком. Обычно он запрограммирован на генерацию последовательности прямоугольных импульсов, что дает при включении динамика непрерывный тон определенной частоты. Управление встроенным динамиком связано прежде всего с программированием второго канала таймера.

Для работы с таймером используются порты (всего 4), из которых для нас интерес представляют порт \$42 (порт второго канала таймера) и порт \$43 (порт управляющего слова — открыт только для записи). Каждый канал имеет свой счетчик, содержимое которого уменьшается от некоторого максимального значения. Для второго канала определена константа \$1234DD. Это значение представляет собой число, которое нужно разделить на частоту в герцах, чтобы получить значение счетчика для таймера.

Итак, мы выяснили, что единичное значение управляющего бита b0 означает, что динамик подключен к каналу, а нулевое значение — что он отключен от канала. Динамик включен, если бит b1 равен единице. Таким образом, прежде всего надо научиться «включать» биты b0 и b1, записывая в них единичные значения, не изменяя значений всех прочих битов. Чтобы сообщить таймеру о том, что следующие два байта являются значением типа Word, обратно пропорциональным частоте, и что на выходе должен быть прямоугольный импульс, в порт микросхемы таймера \$43 должно быть записано значение \$B6. После этого таймер будет включать и выключать динамик с определенной частотой.

Наша первая программа — speaker1 — предназначена для генерации последовательности прямоугольных звуковых импульсов (листинг 8.1).

#### Листинг 8.1. Программа speaker1

```
program speaker1;
uses crt;
const
  speaker_port = $61;
```

```

var
    portval : byte;

begin
    portval := port[speaker_port] and $FC;
    while not KeyPressed do
    begin
        port[speaker_port] := portval or 2;
        delay(5);
        port[speaker_port] := portval;
        delay(5);
    end;
    ReadKey;
end.

```

Двоичное представление шестнадцатеричного значения `#FC` имеет вид 11111100. Вначале определяется значение, находящееся в порте динамика, и два младших бита обнуляются (благодаря использованию операции `and` и маски `$FC`). В цикле динамик включается (в бит b1 записывается единица), затем, после небольшой задержки, отключается и так повторяется до нажатия произвольной клавиши. После нажатия клавиши выполнение цикла прекращается. Важно то, что оба младших бита будут содержать нулевые значения, и при всех последующих обращениях к динамику других программ он будет работать правильно. Программа генерирует тон частотой примерно 100 Гц.

У этой программы есть существенный недостаток. Обработка центральным процессором прерываний во время ее выполнения влияет на точность соблюдения временных задержек. Попробуйте подвигать мышью во время работы программы, и вы услышите перерывы в звучании динамика. По этой причине использование процедуры задержки `delay` является нежелательным. Процедура вывода звука, кроме того, «захватывает» центральный процессор, и одновременное выполнение других процедур оказывается невозможным. Улучшенный вариант программы использует второй канал таймера (программа `speaker2`, листинг 8.2).

Напоминаю, что для использования таймера необходимо сначала подключить динамик ко второму каналу PPI, а затем записать единичные значения двух младших битов в порт динамика.

#### Листинг 8.2. Программа `speaker2`

```

program speaker2;

uses crt;

const
    speaker_port = $61;
    pit_control = $43;
    pit_channel_2 = $42;
    pit_freq = $1234dd;

procedure sound(frequency : word);

```

```

var
  counter : word;
begin
  counter := pit_freq div frequency;
  port[pit_control] := $b6;
  port[pit_channel_2] := lo(counter);
  port[pit_channel_2] := hi(counter);
  port[speaker_port] := port[speaker_port] or 3;
end;

procedure nosound;
begin
  port[speaker_port] := port[speaker_port] and $fc;
end;

begin
  sound(200);
  repeat until keypressed;
  nosound;
end.

```

Теперь манипуляции с мышью уже не будут приводить к прерывистому звучанию динамика. Суммируем наши достижения в модуле speaker (листинг 8.3). В отличие от разобранных ранее процедур, здесь допускается вещественное значение частоты.

#### Листинг 8.3. Модуль speaker

```

unit speaker;

interface

procedure sound(freq : real);
procedure no_sound;

implementation

procedure sound(freq : real);
const
  freq0 = 1.19318e6;
var
  count : word;
  b : byte;
begin
  count := round(freq0/freq);
  b := port[$61];
  if b and $03 = 0 then
    begin
      b := b or $03;
      port[$61] := b;
      port[$43] := $B6;
    end;
end;

```

```

b := Lo(count);
port[$42] := b;
b := Hi(count);
port[$42] := b;
end;

procedure no_sound;
var
  x : byte;
begin
  port[$61] := port[$61] and $FC;
end;

end.

```

## Программа «Виртуальное пианино»

Читатель! Наступает торжественный момент. Сейчас мы применим все свои знания и весь свой опыт программирования на Турбо Паскале и перейдем к серьезной программе — имитатору пианино (*virtual\_piano*, листинг 8.4). Раздел описаний этой программы включает описание формы курсора мыши, который будет играть роль руки вашего виртуального двойника — исполнителя на виртуальном пианино. Клавиатура нашего пианино охватывает почти 5 октав, а 12 частот каждой октавы хранятся в двумерном массиве значений вещественного типа. В группе клавиш каждой октавы имеется 7 белых и 5 черных клавиш, и отдельные массивы для этих частот удобно хранить раздельно, по цветам клавиш.

**Листинг 8.4.** Программа «Виртуальное пианино»

```

program virtual_piano;

uses crt, speaker, dos, mouse, graph, graphs;

const
  masks : array[0..1..0..15] of word = (
    ($E1FF, $EDFF, $EDFF, $EDFF,
     $EDFF, $EC00, $EDB6, $EDB6,
     $0DB6, $6FFE, $6FFE, $6FFE,
     $7FFE, $7FFE, $7FFE, $0000),
    ($1E00, $1200, $1200, $1200,
     $1200, $13FF, $1249, $1249,
     $F249, $9001, $9001, $9001,
     $8001, $8001, $8001, $FFFF));

```

```

const
  width = 18;

```

```
x1 = 35 * width;
y1 = 160;
black_key_low = 100;
half_black_width = width div 3;
quit_x = 500;
quit_y = 400;

var
    frequency : array[0..4, 0..11] of real;
    white_freq : array[0..4, 0..6] of real;
    black_freq : array[0..4, 0..6] of real;
    mouse_OK : boolean;
    x, y : word;
    button : byte;

procedure init_frequency;
const
    white_map : array[0..6] of word = (0, 2, 4, 5, 7, 9, 11);
    black_map : array[0..6] of word = (0, 1, 3, 0, 6, 8, 10);
var
    x, semitone_ratio : real;
    i, j : word;
begin
    semitone_ratio := exp(ln(2.0) / 12.0);
    frequency[1, 9] := 440.0;
    x := 440.0;
    for j := 8 downto 0 do
    begin
        x := x / semitone_ratio;
        frequency[1, j] := x;
    end;
    x := 440.0;
    for j := 10 to 11 do
    begin
        x := x * semitone_ratio;
        frequency[1, j] := x;
    end;
    for j := 0 to 11 do
    begin
        frequency[0, j] := 0.5 * frequency[1, j];
        frequency[2, j] := 2.0 * frequency[1, j];
        frequency[3, j] := 4.0 * frequency[1, j];
        frequency[4, j] := 8.0 * frequency[1, j];
    end;
    for i := 0 to 4 do
    begin
        for j := 0 to 6 do
        begin
            white_freq[i, j] := frequency[i, white_map[j]];
            black_freq[i, j] := frequency[i, black_map[j]];
        end;
    end;
end;
end;
```

```

procedure draw_keyboard;
const
    black : array[1..5] of integer = (-3, 16, 52, 70, 87);
var
    pp, z, p : integer;
begin
    x := 0;
    y := 10;
    for pp := 1 to 5 do
    begin
        for p := 1 to 8 do
        begin
            setcolor(11);
            for z := y to y + 150 do
                line(x, z, x + 15, z);
            putpixel(x, y + 150, 0);
            putpixel(x + 15, y + 150, 0);
            inc(x, 18);
        end;
        dec(x, 131);
        for p := 1 to 5 do
        begin
            setcolor(8);
            for z := y to y + 80 do
                line(x + black[p], z, x + black[p] + 11, z);
            setcolor(0);
            rectangle(x + black[p] + 1, y, x + black[p] + 12, y + 80);
            setcolor(7);
            for z := y + 77 to y + 79 do
                line(x + black[p] + 2, z, x + black[p] + 11, z);
        end;
        inc(x, 114);
    end;
    rectangle(quit_x, quit_y, 639, 479);
    outtextxy(quit_x + 40, quit_y + 35, "Quit");
end;

procedure poll;
begin
repeat
    get_mouse_status(button, x, y);
    if button and $01 <> 0 then
        exit
    else
        no_sound;
    until false;
end;

procedure play;
var
    key, octave : word;
    z, w : integer;

```

```

begin
    key := x div width;
    octave := key div 7;
    z := x mod (7 * width);
    w := round(z / width);
    if (y <= black_key_low) and
        (abs(width * w - z) <= half_black_width) and
        (w in [1, 2, 4..6]) then
        sound(black_freq[octave, w])
    else
        sound(white_freq[octave, key mod 7]);
repeat
    get_mouse_status(button, x, y);
until button and $01 = $00;
no_sound;
end;

begin
    init_frequency;
    reset_mouse(mouse_OK, button);
    if not mouse_OK then
        halt;
    open_graph;
    reset_mouse(mouse_OK, button);
    set_graph_cursor_shape(7, 0, @masks);
    show_cursor;
    mouse_gotoXY(320, 300);
    draw_keyboard;
repeat
    poll;
    if (x = 0) or (y = 0) then
        begin
            no_sound;
            poll;
        end;
    if (x <= x1) and (y <= y1) then
        play
    else
        if (x >= quit_x) and (y >= quit_y) then
            break;
    no_sound;
until false;
close_graph;
end.

```

В разделе описаний констант программы `virtual_piano` кроме масок курсора мыши содержатся описания размеров графических изображений клавиш (`width` — ширина белой клавиши), а также положение и размеры прямоугольной области экрана, щелчок мышью в которой приводит к завершению работы программы.

В процедуре `init_frequency` происходит вычисление частот нот музыкального звукоряда. При этом используется основной принцип настройки — в равномерно

темперированной шкале все отношения частот двух соседних нот (полутонов) должны быть равны. Поскольку в октаве имеется 12 нот, а частота звука при переходе к следующей октаве удваивается, каждая частота равна корню двенадцатой степени из двойки, умноженному на предыдущую частоту.

Процедура `draw_keyboard` выводит изображение клавиатуры и инициализирует остальную часть экрана. В частности, она выделяет область, в которой следует щелкнуть кнопкой мыши, чтобы выйти из программы.

Процедура `poll` проверяет, нажата ли левая кнопка мыши, а процедура `play` включает звучание ноты и продолжает его до тех пор, пока левая кнопка мыши не отпущена.

Наконец, основная часть программы соединяет все эти процедуры вместе.

## Программирование SoundBlaster

Прежде чем мы перейдем к программированию звуковой карты, полезно разобраться с тем, что же такое звук и какими свойствами он обладает.

### Звук и его свойства

Наше ухо реагирует на звуковые (акустические) колебания, представляющие собой волны сжатия и разрежения воздуха. Если подключить микрофон к осциллографу, любой звук, направляемый в микрофон, изображается последовательностью максимумов и минимумов. Чем выше максимум давления в звуковой волне, тем громче звук и тем выше будут пики на осциллограмме.

Звук обладает тремя основными характеристиками. Это — *основной тон*, или *высота* звука, его *тембр* и *амплитуда*. Высота звука определяется частотой колебаний (количеством циклов в секунду) в звуковой волне. Чем больше частота, тем выше звук. Можно привести такой пример. Струны контрабаса толще и массивнее, чем струны скрипки, поэтому частота их колебаний меньше и, как следствие, у контрабаса звук ниже (рис. 8.1, *а*), чем у скрипки (рис. 8.1, *б*).

*Тембр* является качественной характеристикой звука, позволяющей определить на слух различие между звуками одинаковой высоты, издаваемыми разными источниками. Так, например, настроенный камертон дает почти чистую синусоидальную волну, а флейта, издающая ту же ноту, дает звук более сложного состава (рис. 8.2, *а*). На рис. 8.2, *б* приведена осциллограмма звука гитарной струны, настроенной на ту же ноту, ее вид еще сложнее.

*Амплитуда* колебаний определяет громкость звука. Амплитуда — это расстояние между максимумом и нулевой линией осциллограммы звука.

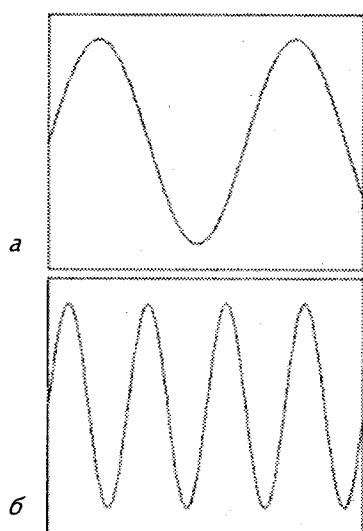


Рис. 8.1. Осциллограммы: *а* — низкого, *б* — высокого тона

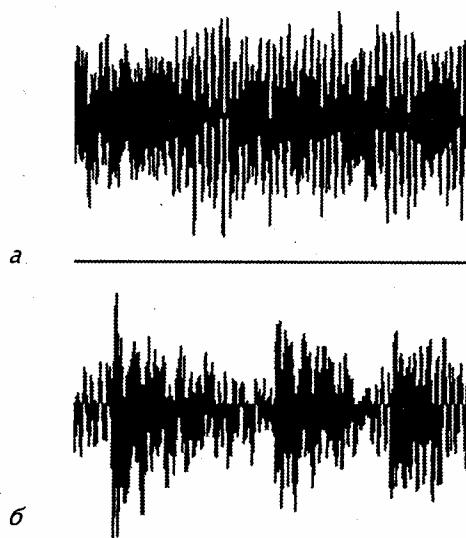


Рис. 8.2. Осциллограммы: *а* — звука флейты, *б* — гитарной струны

Звуковые карты для синтеза музыкальных звуков используют *амплитудные огибающие*, которые управляют громкостью звука на протяжении его существования. Амплитудная огибающая представляет собой линию, проведенную через максимумы осциллограммы звука. Анализ формы амплитудных огибающих показывает, что в развитии музыкального звука от его возникновения до полного затухания можно выделить несколько этапов. Эти этапы схематически изображены на рис. 8.3, где показана амплитудная огибающая звука.

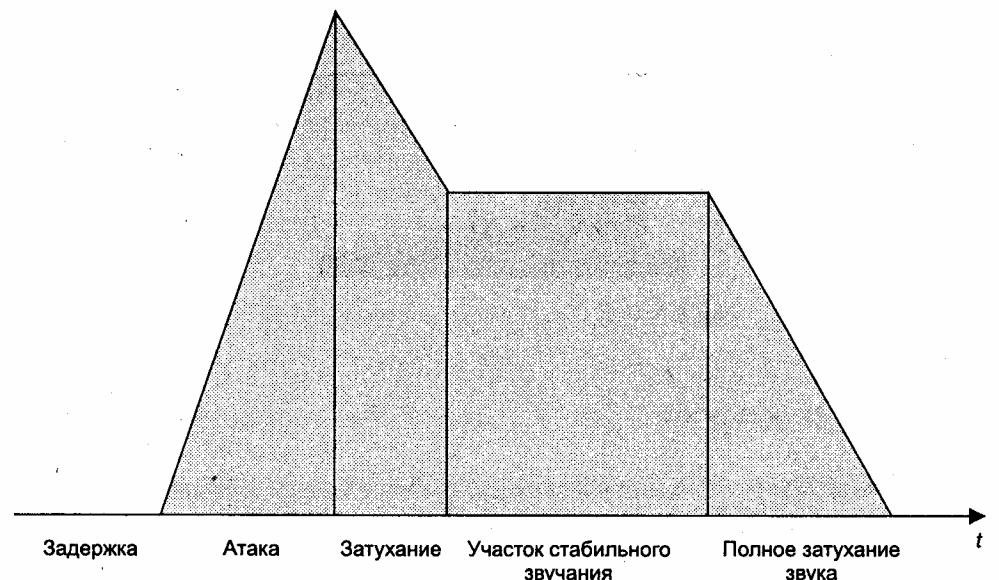


Рис. 8.3. Амплитудная огибающая музыкального звука

- Задержка* — это промежуток между началом звукоизвлечения (нажатием на клавишу музыкального инструмента) и появлением первого звука.
- Атака* — промежуток между появлением звука и достижением им максимальной громкости.
- Затухание* — промежуток, на котором звук затухает от максимального значения до уровня участка стабильного звучания.
- Участок стабильного звучания* — промежуток относительно устойчивого звучания, который обычно продолжается до тех пор, пока нажата клавиша музыкального инструмента.
- Полное затухание звука* — наступает после отпускания клавиши музыкального инструмента.

Форма огибающей для звуков, порождаемых разными музыкальными инструментами, различна. Для церковного органа, например, характерна произвольная длительность стабильного звучания. У барабана или фортепиано эта длительность фиксирована. На рис. 8.4 приведена осциллограмма звука, извлеченного на гитаре. Видно, что участок атаки здесь имеет практически нулевую длину, затем следует непродолжительный участок затухания, а потом наступает стабилизация. Полное затухание звука струны длится довольно долго.

Огибающие могут использоваться для создания специальных звуковых эффектов. В этом случае обычно применяется *модуляция* звука. Модуляция — это динамическое изменение какой-нибудь характеристики звука, например, его амплитуды или частоты.

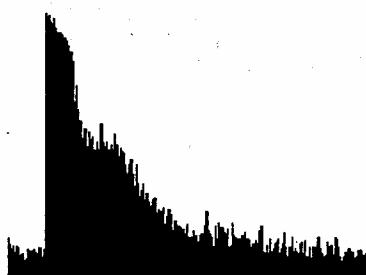


Рис. 8.4. Оциллограмма звука гитарной струны

## Звуковая карта

Звук высокого качества может обеспечить, конечно, только *звуковая карта*. Звуковая карта предназначена для воспроизведения и записи звука на компьютере. В число ее дополнительных функций входит поддержка некоторых устройств (джойстик, CD-ROM). Диапазон звуковых карт широк — от простейших, предназначенных для невзыскательного слушателя, до звуковых карт для профессиональной работы. В программировании для различных звуковых карт есть много общего, и с некоторыми приемами такого программирования мы сейчас познакомимся. Для определенности будем заниматься звуковыми картами, принадлежащими к семейству карт, совместимых с картой SoundBlaster. В эту группу попадает большое число современных звуковых карт, которые могут отличаться друг от друга техническими характеристиками, но их архитектура и методы программирования примерно одинаковы.

Звуковая карта состоит из нескольких основных модулей. Это:

- *Модуль цифровой обработки звука*, называемый также цифровым звуковым процессором (DSP — Digital Sound Processor). Осуществляет преобразования аналогового сигнала в цифровой и наоборот. DSP состоит из узла, непосредственно выполняющего аналого-цифровые преобразования, и узла управления.
- *FM-синтезатор* (FM — Frequency Modulation, частотная модуляция). Построен на базе микросхем FM-синтеза OPL2 или OPL3.
- *Микширующее устройство*. Модуль микширования включает в себя собственно микшер и усилители. Микшер предназначен для смешивания сигналов от различных устройств звуковой карты. Амплитуда, а следовательно, и громкость каждого сигнала управляются независимо для каналов воспроизведения и записи. В канале воспроизведения сигнал идет вначале на выходной усилитель, которым можно управлять, а затем на колонки или наушники. В канале записи сигнал сначала поступает на аналого-цифровой преобразователь (АЦП), где происходит преобразование аналогового (непрерывно изменяющегося) сигнала в цифровой (представленный последовательностью чисел).

- **Блок MPU** (MIDI Processing Unit – устройство MIDI-обработки) Осуществляет прием и передачу данных по внешнему MIDI-интерфейсу.

Здесь мы будем заниматься только программированием для FM-синтезатора. Что такое частотный синтез? Напомню, что чистый звуковой тон – это синусоидальная волна с определенной частотой, определяющей высоту тона. Реальный звук редко имеет чистый тон, обычно это сложная смесь синусоидальных сигналов различных частоты и амплитуды. Смесь может получаться простым суммированием различных сигналов, при частотной же модуляции один сигнал определяет закон изменения частоты для другого сигнала.

С точки зрения аппаратной реализации, частотный синтез – синтез при помощи нескольких генераторов синусоидального сигнала со взаимной модуляцией. Каждый генератор снабжается схемой управления частотой и амплитудой сигнала и образует базовую единицу синтеза, которая называется *оператором*. В звуковых картах применяются двухоператорный (OPL2) и четырехоператорный (OPL3) синтез. Большинство современных звуковых карт поддерживает режим OPL3, но из соображений совместимости во многих программах используется двухоператорный режим.

Алгоритм взаимодействия операторов и параметры каждого из них – частота, амплитуда и закон их изменения во времени – определяют тембр звука. Количество операторов и возможность гибкого ими управления определяет предельное количество синтезируемых тембров. Основными достоинствами метода частотного синтеза принято считать достаточно большое количество синтезируемых звуков и минимальные проблемы с совместимостью. Основной недостаток – отчетливо выраженный «электронный» характер звука и очень грубая имитация звучания реальных музыкальных инструментов. Значительно большие возможности дает метод синтеза, основанный на использовании заранее записанных образцов звука (синтез по волновым таблицам – Wave Tables).

В FM-синтезаторе звуковой карты SoundBlaster, собранном на основе микросхемы OPL3, для создания звука используются 4 сигнала. Звуковая карта SoundBlaster поддерживает стереозвучание, для чего многие из вышеперечисленных устройств дублируются для левого и правого каналов.

## Программирование звуковой карты

Итак, мы выяснили, что звуковая карта SoundBlaster поддерживает работу с FM-синтезированным и оцифрованным звуком. Как уже было сказано, рассматривая основные вопросы программирования для SoundBlaster, мы ограничимся программированием музыкальных FM-синтезированных звуков.

Программирование для звуковой карты – это прежде всего программирование DSP, и осуществляется оно посредством записи данных в порты ввода/вывода:

порт \$0388 (Address/Status port) – адресный порт монорежима, открытый как для чтения, так и для записи;

порт \$0389 (Data port) – порт данных, открытый только для записи;

порт \$0220 — адресный порт для левого стереоканала;  
 порт \$0221 — порт данных для левого стереоканала;  
 порт \$0222 — адресный порт для правого стереоканала;  
 порт \$0223 — порт данных для правого стереоканала.

Вывод через порты \$0388 и \$0389 идет на два канала одновременно (монорежим).

Звуковые карты имеют массив из 244 регистров. Для записи в нужный регистр в адресный порт посыпается его номер (от \$01 до \$F5), а нужное значение записывается в порт данных. Регистры открыты только для записи.

Чтобы синтезатор зазвучал, вначале надо записать в регистры нулевые значения. Это достаточно грубый, но надежный метод инициализации звуковой карты. Инициализация производится один раз при запуске программы. Затем следует «настроить» синтезатор, записав в регистры соответствующие значения, определяющие параметры генерации звука. Краткое описание некоторых регистров, используемых при программировании звуковой карты, приведено ниже.

Все биты регистра \$01, как правило, устанавливаются в 0. Интерес представляет 5-й бит. Если для него установлено единичное значение, то это позволяет микросхеме FM-синтезатора контролировать форму волны. Далее идет группа регистров, которые мы не будем использовать.

Перейдем к описанию регистров, сгруппированных по 22. Это связано с тем, что FM-синтезатор содержит 9 каналов, причем на каждый канал (или «FM голос») приходится 2 оператора. Табл. 8.1 устанавливает соответствие между каналами, операторами и номерами регистров.

**Таблица 8.1.** Смещения для управляющих регистров звуковой карты

Канал	1	2	3	4	5	6	7	8	9
1-й оператор	\$00	\$01	\$02	\$08	\$09	\$0A	\$10	\$11	\$12
2-й оператор	\$03	\$04	\$05	\$0B	\$0C	\$0D	\$13	\$14	\$15

В этой таблице приведены смещения для регистров. Таким образом, например, параметры атаки и затухания звукового сигнала для первого канала заносятся в регистр \$60 для первого оператора и в регистр \$63 — для второго.

Регистры \$20...\$35 хранят управляющие байты, каждый из которых содержит биты, позволяющие включить или отключить определенный режим генерации звука. Седьмой (старший) бит позволяет применить амплитудную модуляцию, то есть управление амплитудой звукового сигнала с помощью другого сигнала. Шестой бит отвечает за эффект «вибратор» — высокочастотное «дрожание» звука. Для нас наибольший интерес здесь представляют младшие биты с нулевого по третий включительно. Их значение определяет сдвиг по высоте генерируемого звука относительно установленной частоты. Ниже приводится соответствие между значением битов и смещением по частоте:

0 — вниз на одну октаву;

- 1 — нулевое смещение (высота звука совпадает с заданной частотой);
- 2 — вверх на одну октаву;
- 3 — вверх на одну октаву и квинту;
- 4 — вверх на две октавы;
- 5 — вверх на две октавы и большую терцию;
- 6 — вверх на две октавы и квинту;
- 7 — вверх на две октавы и малую септиму;
- 8 — вверх на три октавы;
- 9 — вверх на три октавы и большую секунду;
- \$A, \$B — вверх на три октавы и большую терцию;
- \$C, \$D — вверх на три октавы и квинту;
- \$E, \$F — вверх на три октавы и большую септиму.

Регистры \$40...\$55 управляют уровнем масштабирования и громкостью звука. Старшие биты (7 и 6) позволяют определить изменение громкости при возрастании частоты — значение \$00 означает отсутствие такого изменения, а все прочие комбинации задают разную скорость изменения громкости при возрастании частоты. Биты с 0 по 5 определяют громкость звука, причем максимальной громкости соответствует установка для всех битов нулевых значений, а минимальной — единичных.

Регистры \$60...\$75 позволяют установить продолжительность участков атаки и затухания. Биты 0–3 «отвечают» за затухание, причем нулевому значению соответствует самое медленное затухание, а значению \$F — самое быстрое. Биты 4–7 определяют скорость атаки. Здесь также нулевому значению соответствует самое медленное затухание, а значению \$F — самое быстрое.

Регистры \$80...\$95 определяют параметры участка стабильного звучания (громкость — от \$00 для минимальной до \$F0 для максимальной громкости) и полного затухания звука (скорость затухания — от \$00 для минимальной до \$0F для максимальной).

Важнейшими для нас являются регистры \$A0...\$B8. Прежде чем перейти к пояснению назначения отдельных битов, необходимо ввести понятие *F-числа*. F-число задает определенную ноту. В табл. 8.2 приведены значения F-чисел для первой октавы, соответствующие хроматическому звукоряду.

F-число состоит из двух байтов. Старший принимает одно из двух возможных значений \$01 или \$02, а младший принимает значение, соответствующее определенной ноте. Группа регистров \$A0...\$A8 содержит младший байт F-числа. В регистрах \$B0...\$B8 запись в пятый бит единицы включает звук (то есть соответствует нажатию клавиши музыкального инструмента), а нулевое значение выключает его. Биты со второго по четвертый задают октаву — самой низкой присвоен номер 0, а самой высокой — 7. Два младших бита содержат старшие значимые биты F-числа.

Таблица 8.2. F-числа для первой октавы

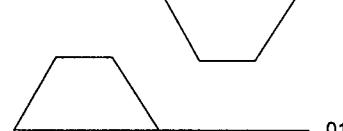
F-число	Частота (Гц)	Нота
\$156	261,6	C (до)
\$16B	277,2	C# (до-диез)
\$181	293,7	D (ре)
\$198	311,1	D# (ре-диез)
\$1B0	329,6	E (ми)
\$1CA	349,2	F (фа)
\$1E5	370,0	F# (фа-диез)
\$202	392,0	G (соль)
\$220	415,3	G# (соль-диез)
\$241	440,0	A (ля)
\$263	466,2	A# (ля-диез)
\$287	493,9	H (си)

Регистр \$BD управляет глубиной амплитудной модуляции (бит 7), глубиной вибрато (бит 6) и «ритмической секцией». При этом пятый бит включает/отключает ритмическую секцию, а оставшиеся биты позволяют задать ритмический инструмент.

Регистры с \$E0 по \$E5 позволяют внести в генерируемый звук искажение с помощью одной из четырех временных диаграмм (рис. 8.5).



00



01



10



11

Рис. 8.5. Временная диаграмма искажений, задаваемых регистрами \$E0...\$E5

Код диаграммы записывается в нулевой и первый биты регистров. В дальнейшем мы будем использовать первый канал, выбирая из каждой группы первый регистр.

Рекомендуемые начальные установки регистров приведены в табл. 8.3.

**Таблица 8.3.** Начальные установки регистров FM-синтезатора

Регистр	Значение	Краткое описание
\$20	\$01	Смещение по частоте для модулирующего сигнала отсутствует
\$40	\$10	Уровень модулирующего сигнала 40 дБ
\$60	\$F0	Для модулирующего сигнала атака быстрая, а затухание долгое
\$80	\$77	Для модулирующего сигнала уровень стабильного звучания средний, скорость полного затухания тоже средняя
\$A0	\$98	Частота звука соответствует ноте D#
\$20	\$01	Смещение по частоте для несущей отсутствует
\$40	\$00	Установлен максимальный уровень для несущей
\$60	\$F0	Для несущей атака быстрая, затухание медленное
\$80	\$77	Для несущей уровень стабильного звучания средний, скорость полного затухания тоже средняя
\$B0	\$31	Включить звук и установить первую октаву

Чтобы выключить звук, необходимо обнулить пятый бит регистра \$B0.

В модуле sbfm реализованы некоторые звуковые функции FM-синтезатора, позволяющие воспроизводить с помощью звуковой карты музыкальные звуки. В интерфейсной секции модуля (листинг 8.5) описаны константы Mono, Left\_Stereo и Right\_Stereo, которые содержат адреса портов для режимов моно- и стереозвучания (левый и правый каналы), o1..o8 — коды октав, ключи key\_on и key\_off (клавиша нажата/отпущена), f\_most1 и f\_most2 — старшие биты кодов нот и, наконец, сами ноты note\_\*.

**Листинг 8.5.** Интерфейсная секция модуля sbfm

```
unit sbfm;

interface

uses crt;

var
  port_num : word;

const
  Left_Stereo = $0220;
  Right_Stereo = $0222;
  Mono = $0388;
```

```

o1 = $00;
o2 = $04;
o3 = $08;
o4 = $0C;
o5 = $10;
o6 = $14;
o7 = $18;
o8 = $1C;
key_on = $20;
key_off = $00;
f_most1 = $0100;
f_most2 = $0200;
note_do_diez = $6B + f_most1;
note_re = $81 + f_most1;
note_re_diez = $98 + f_most1;
note_mi = $B0 + f_most1;
note_fa = $CA + f_most1;
note_fa_diez = $E5 + f_most1;
note_so1 = $02 + f_most2;
note_sol_diez = $20 + f_most2;
note_la = $41 + f_most2;
note_la_diez = $63 + f_most2;
note_si = $87 + f_most2;
note_do = $AE + f_most2;

procedure port_init(port_num : Word);
procedure play_note(port_num, octav, note, duration : Word);
procedure play_note_mono(octav, note, duration : Word);
procedure play_note_stereo(octav, note, duration : Word);
procedure play_pause(duration : word);
procedure sound_on(port_num, freq : Word);
procedure set_modulator_mult(port_num : Word; mult : Byte);
procedure set_modulator_level(port_num : Word; level : Byte);
procedure set_modulator_attack(port_num : Word; attack : Byte);
procedure set_modulator_sustain(port_num : Word; sustain : Byte);
procedure set_carrier_mult(port_num : Word; mult : Byte);
procedure set_carrier_volume(port_num : Word; volume : Byte);
procedure set_carrier_attack(port_num : Word; attack : Byte);
procedure set_carrier_sustain(port_num : Word; sustain : Byte);
procedure sound_off(port_num : word);

```

Рассмотрим процедуры модуля sbfm. Процедура `port_init` (листинг 8.6) предназначена для инициализации порта и регистров. Ее параметром является адрес порта для устанавливаемого режима звуковоспроизведения.

#### Листинг 8.6. Процедура инициализации порта и регистров (модуль sbfm)

```

implementation
procedure port_init(port_num : Word);
var
  i, k : byte;
begin

```

```

port[port_num] := $01;
port[port_num] := $00;
for i := $01 to $F5 do
begin
    port[port_num] := i;
    port[port_num + $0001] := $00;
end;
end;

```

Процедура `play_note` (листинг 8.7) проигрывает в заданном режиме и указанной октаве определенную ноту. Последний параметр задает длительность звучания ноты в миллисекундах. Воспроизведение ноты производится в соответствии с предварительно сделанными установками, в том числе касающимися способа звукоизвлечения (атака, затухание и т. д.).

#### **Листинг 8.7.** Процедура `play_note` модуля `sbfm`

```

procedure play_note(port_num, octav, note, duration : Word);
begin
    port[port_num] := $A0;
    port[port_num + $0001] := Lo(note);
    port[port_num] := $B0;
    port[port_num + $0001] := key_on + octav + Hi(note);
    delay(duration);
    port[port_num] := $B0;
    port[port_num + $0001] := key_off + octav + Hi(note);
end;

```

Процедуры `play_note_mono` и `play_note_stereo` (листинг 8.8) аналогичны `play_note`, но подразумевают использование моно- и стереорежимов.

#### **Листинг 8.8.** Процедуры воспроизведения ноты в моно- и стереорежимах

```

procedure play_note_mono(octav, note, duration : Word);
begin
    port[mono] := $A0;
    port[mono + $0001] := Lo(note);
    port[mono] := $B0;
    port[mono + $0001] := key_on + octav + Hi(note);
    delay(duration);
    port[mono] := $B0;
    port[mono + $0001] := key_off + octav + Hi(note);
end;

procedure play_note_stereo(octav, note, duration : Word);
begin
    port[Left_Stereo] := $A0;
    port[Left_Stereo + $0001] := Lo(note);
    port[Right_Stereo] := $A0;
    port[Right_Stereo + $0001] := Lo(note);
    port[Left_Stereo] := $B0;
    port[Left_Stereo + $0001] := key_on + octav + Hi(note);

```

```

port[Right_Stereo] := $B0;
port[Right_Stereo + $0001] := key_on + octav + Hi(note);
delay(duration);
port[Left_Stereo] := $B0;
port[Left_Stereo + $0001] := key_off + octav + Hi(note);
port[Right_Stereo] := $B0;
port[Right_Stereo + $0001] := key_off + octav + Hi(note);
end;

```

Листинг 8.9 содержит исходные тексты трех процедур модуля. Простенькая процедура `play_pause` «играет» паузу заданной длительности.

Процедура `sound_on` позволяет воспроизводить произвольный звук (разумеется, в той мере, в какой это возможно с FM-синтезатором звуковой карты), а не обязательно ноту. Ее параметры — это порт и частота. Частота задается байтовым значением, а соответствие между этим значением и значением частоты в герцах можно установить с помощью табл. 8.2.

Выключение звука производится процедурой `sound_off`. Эта процедура необходима лишь при генерации произвольного звука процедурой `sound_on`, поскольку процедуры воспроизведения ноты `play_note...` уже содержат в себе выключение звука. Процедура `sound_off` обязательно должна вызываться, если в программе использовалась процедура `sound_on`.

#### **Листинг 8.9.** Процедуры `play_pause`, `sound_on` и `sound_off` модуля sbfm

```

procedure play_pause(duration : word);
begin
  delay(duration);
end;

procedure sound_on(port_num, freq : Word);
begin
  port[port_num] := $A0;
  port[port_num + $0001] := freq;
  port[port_num] := $B0;
  port[port_num + $0001] := $31;
end;

procedure sound_off(port_num : word);
begin
  port[port_num] := $B0;
  port[port_num + $0001] := $11;
end;

```

Далее идут процедуры, с помощью которых можно установить режимы звуко-воспроизведения (листинг 8.10). Процедура `set_modulator_mult` устанавливает смещение по частоте для модулирующего сигнала. Процедура `set_modulator_level` устанавливает уровень модулирующего сигнала. Процедура `set_modulator_attack` устанавливает атаку при звукоизвлечении. Процедура `set_modulator_sustain` устанавливает уровень участка стабильного звучания.

**Листинг 8.10.** Процедуры установки режимов звуковоспроизведения модуля sbfm

```

procedure set_modulator_mult(port_num : Word; mult : Byte);
begin
    port[port_num] := $20;
    port[port_num + $0001] := mult;
end;

procedure set_modulator_level(port_num : Word; level : Byte);
begin
    port[port_num] := $40;
    port[port_num + $0001] := level;
end;

procedure set_modulator_attack(port_num : Word; attack : Byte);
begin
    port[port_num] := $60;
    port[port_num + $0001] := attack;
end;

procedure set_modulator_sustain(port_num : Word; sustain : Byte);
begin
    port[port_num] := $80;
    port[port_num + $0001] := sustain;
end;

```

Следующая группа процедур (листинг 8.11) задает параметры несущей.

**Листинг 8.11.** Процедуры установки параметров несущей модуля sbfm

```

procedure set_carrier_mult(port_num : Word; mult : Byte);
begin
    port[port_num] := $23;
    port[port_num + $0001] := mult;
end;

procedure set_carrier_volume(port_num : Word; volume : Byte);
begin
    port[port_num] := $43;
    port[port_num + $0001] := volume;
end;

procedure set_carrier_attack(port_num : Word; attack : Byte);
begin
    port[port_num] := $63;
    port[port_num + $0001] := attack;
end;

procedure set_carrier_sustain(port_num : Word; sustain : Byte);
begin
    port[port_num] := $83;
    port[port_num + $0001] := sustain;
end;

```

```
end;
```

```
begin
end.
```

В модуле `sbfm`, разумеется, не хватает некоторых достаточно важных функций. Одной из них является процедура определения наличия звуковой карты. Алгоритм автоматического определения наличия звуковой карты зависит от типа карты, и его программирование требует углубленного изучения технических деталей ее устройства и функционирования. Изучение этих деталей не входит в наши планы, поэтому перейдем к программе `sbfm_demo`, демонстрирующей работу процедур модуля `sbfm` (листинг 8.12). Эта программа при запуске позволяет выбрать один из двух режимов работы (моно или стерео). В соответствии со сделанным выбором производится инициализация портов. Затем идет серия вызовов процедур, задающих режимы звучания синтезатора. Программа содержит две «звучашие» части. В первой воспроизводится гамма до-мажор, а во второй генерируется звук, высота которого изменяется по синусоидальному закону. В этом случае интересный эффект возникает при работе в стереорежиме, поскольку в левый и правый каналы подаются сигналы, слегка отличающиеся друг от друга. Различие между ними состоит в том, что один синусоидальный звук имеет небольшой фазовый сдвиг относительно другого. Это создает эффект объемности звучания.

#### Листинг 8.12. Программа, демонстрирующая использование модуля `sbfm`

```
program sbfm_demo;
uses crt, sbfm;
var
  i, sound_left, sound_right : word;
  Ch : char;

begin
  clrscr;
  WriteLn("Введите:");
  WriteLn(" 1 для работы в режиме моно;");
  WriteLn(" 2 для работы в режиме стерео;");
  WriteLn(" 0 для прекращения работы.");

  ch := readkey;
  while not ((ch = "1") or (ch = "2")) or (ch = "0") do
    case ch of
      "1" : port_init(Mono);
      "2" : begin
        port_init(Left_Stereo);
        port_init(Right_Stereo);
      end;
      "0" : exit;
    else ch := readkey;
```

```
end;
clrscr;
WriteLn("Нажмите любую клавишу для продолжения работы");

set_modulator_mult(Left_Stereo, $01);
set_modulator_mult(Right_Stereo, $01);
set_modulator_mult(Mono, $01);

set_modulator_level(Left_Stereo, $10);
set_modulator_level(Right_Stereo, $10);
set_modulator_level(Mono, $10);

set_modulator_attack(Left_Stereo, $f0);
set_modulator_attack(Right_Stereo, $f0);
set_modulator_attack(Mono, $f0);

set_modulator_sustain(Left_Stereo, $77);
set_modulator_sustain(Right_Stereo, $77);
set_modulator_sustain(Mono, $77);

set_carrier_mult(Left_Stereo, $01);
set_carrier_mult(Right_Stereo, $01);
set_carrier_mult(Mono, $01);

set_carrier_volume(Left_Stereo, $00);
set_carrier_volume(Right_Stereo, $00);
set_carrier_volume(Mono, $00);

set_carrier_attack(Left_Stereo, $f0);
set_carrier_attack(Right_Stereo, $f0);
set_carrier_attack(Mono, $f0);

set_carrier_sustain(Left_Stereo, $77);
set_carrier_sustain(Right_Stereo, $77);
set_carrier_sustain(Mono, $77);

repeat
  play_note_stereo(o4, note_do, 200);
  play_note_stereo(o5, note_re, 200);
  play_note_stereo(o5, note_mi, 200);
  play_note_stereo(o5, note_fa, 200);
  play_note_stereo(o5, note_sol, 200);
  play_note_stereo(o5, note_la, 200);
  play_note_stereo(o5, note_si, 200);
until keypressed;
readln;
clrscr;
WriteLn("Нажмите любую клавишу для прекращения работы");

i := 1;
repeat
  inc(i);
```

```

sound_left := trunc(128 * sin(0.0001 * i)) + 127;
sound_right := trunc(128 * sin(0.0001 * i + 0.11)) + 127;
if ch = "1" then
  sound_on(Mono, sound_left)
else
begin
  sound_on(Left_Stereo, sound_left);
  sound_on(Right_Stereo, sound_right);
end;
until keypressed;
case ch of
"1" : sound_off(Mono);
"2" : begin
  sound_off(Left_Stereo);
  sound_off(Right_Stereo);
end;
end;
end.

```

Вновь обращаю внимание читателя на то, что в конце работы программы звук должен быть отключен, иначе после завершения работы программы колонки вашего компьютера будут продолжать звучать. С программой `sbfm_demo` полезно повозиться, задавая разные режимы и способы воспроизведения как музыкальных, так и прочих звуков. Здесь открывается большой простор для самостоятельных экспериментов.



#### ВНИМАНИЕ

Если ваш компьютер расположен недалеко от стены, отделяющей вашу квартиру от соседней, не включайте колонки компьютера на полную мощность! Слишком громкие эксперименты со звуком могут огорчить ваших соседей.

## Другие возможности

Мы коснулись, да и то, надо признаться, бегло, лишь некоторых возможностей программирования одного из функциональных узлов звуковой карты — FM-синтезатора. Микросхема DSP звуковой карты Sound Blaster имеет 16 портов, которые используются для FM-синтезатора, микшера, доступа к CD-ROM. Программисту на Турбо Паскале открыты все эти возможности. Ведь програмировать можно не только воспроизведение FM-звука, но и запись с микрофона. Звуковая карта дает возможность воспроизведения звуков, записанных в специальных форматах. Одним из наиболее популярных является формат WAV. Инструментальные музыкальные произведения записываются в специальном MIDI-формате, поэтому и любитель, и поклонник настоящей музыки наверняка заинтересовались бы программированием MIDI. Но это — особый разговор.

#### Упражнение 8.1

Постарайтесь извлечь с помощью программы `sbfm_demo` такие характерные звуки, как звук падающего мяча, звук стрельбы из пистолета или пулемета и т. д.

**Упражнение 8.2**

Переделайте программу *virtual\_piano* для звуковой карты SoundBlaster.

## Что нового мы узнали?

- Познакомились с программированием для встроенного динамика.
- Познакомились с основами программирования для звуковой карты и основными принципами ее работы.



## Введение в объектно- ориентированное программирование

- 
- Что такое объектно-ориентированное программирование
  - Модуль matrices
-

**Ц**ель настоящего урока — познакомить читателя с основными идеями и принципами объектно-ориентированного программирования (ООП) в том объеме, в котором оно поддерживается Турбо Паскалем. Вводятся и разъясняются базовые понятия ООП — объект, метод, инкапсуляция, наследование и полиморфизм. Читатель познакомится с виртуальными методами. Во второй части урока разбирается пример использования ООП.

## Что такое объектно-ориентированное программирование

Элементы объектно-ориентированного программирования появились в начале 70-х в языке моделирования Симула, затем получили свое развитие, и в настоящее время ООП принадлежит к числу ведущих технологий программирования. В Турбо Паскале поддержка этой технологии появилась, начиная с версии 5.5 (1989 год).

Основная цель ООП, как и большинства других подходов к программированию, — повышение эффективности разработки программ. Идеи ООП оказались плодотворными и нашли применение не только в языках программирования, но и в других областях Computer Science, например в области разработки операционных систем.

Появление объектно-ориентированного программирования было связано с тем наблюдением, что компьютерные программы представляют собой описание *действий*, выполняемых над различными *объектами*. В роли последних могут выступать, например, графические объекты, записи в базах данных или совокупности числовых значений. В традиционных методах программирования изменение данных или правил их обработки часто приводило к необходимости значительного изменения программы. Всякое существенное изменение программы — это большая неприятность для программиста, так как при этом увеличивается вероятность ошибок, вследствие чего возрастает время, необходимое для «доводки» программы. Использование ООП позволяет выйти из такой ситуации с ми-

нимальными потерями, сводя необходимую модификацию программы к ее расширению и дополнению. Сразу замечу, что ООП не является панацеей от всех программистских бед, но его ценность как передовой технологии программирования несомненна. Изучение идей и методов объектно-ориентированного программирования — не очень простая задача, однако освоение ООП может существенно упростить разработку и отладку сложных программ.

Мы уже привыкли использовать в своих программах процедуры и функции для программирования тех сложных действий по обработке данных, которые приходится выполнять многократно. Использование подпрограмм в свое время было важным шагом на пути к увеличению эффективности программирования. Подпрограмма может иметь формальные параметры, которые при обращении к ней заменяются фактическими параметрами. В этом случае есть опасность вызова подпрограммы с неправильными данными, что может привести к сбою программы и ее аварийному завершению при выполнении. Поэтому естественным обобщением традиционного подхода к программированию является объединение данных и подпрограмм (процедур и функций), предназначенных для их обработки.

## Объекты

Базовым в объектно-ориентированном программировании является понятие *объекта*. Объект имеет определенные свойства. Состояние объекта задается значениями его признаков. Объект «знает», как решать определенные задачи, то есть располагает методами решения. Программа, написанная с использованием ООП, состоит из объектов, которые могут взаимодействовать между собой.

Ранее отмечалось, что программная реализация объекта представляет собой объединение данных и процедур их обработки. В Турбо Паскале имеется тип *object*, который можно считать обобщением структурного типа *record*. Переменные объектного типа называются *экземплярами* объекта. Здесь требуется уточнение — экземпляр лишь формально можно назвать переменной. Его описание дается в предложении описания переменных, но в действительности экземпляр — нечто большее, чем обычная переменная. Почему? Это будет понятно из дальнейшего.

В отличие от типа «запись», объектный тип содержит не только поля, описывающие данные, но также процедуры и функции, описания которых содержатся в описании объекта. Эти процедуры и функции называются *методами*. Методам объекта доступны его поля. Следует заметить, что методы и их параметры определяются в описании объекта, а их реализация дается вне этого описания, в том месте программы, которое предшествует вызову данного метода. В описании объекта фактически содержатся лишь *шаблоны* обращений к методам, которые необходимы компилятору для проверки соответствия количества параметров и их типов при обращении к методам. Вот пример описания объекта:

```
type
  Location = object
    X,Y: Integer;
    procedure Init(InitX, InitY: Integer);
```

```

        function GetX: Integer;
        function GetY: Integer;
    end;

```

Здесь описывается объект, который может использоваться в дальнейшем, скажем, в графическом режиме и который предназначен для определения положения на экране произвольного графического элемента. Объект описывается с помощью зарезервированных слов `object...end`, между которыми находятся описания полей и методов. В нашем примере объект содержит два поля для хранения значений графических координат, а также описания процедуры и двух функций — это *методы* данного объекта. Процедура предназначена для задания первоначального положения объекта, а функции — для считывания его координат.

Зарезервированное слово `private` позволяет ограничить доступ к полям объекта. В следующем примере доступ к переменным `X` и `Y` возможен только через методы объектного типа `Location`:

```

type
    Location = object
        procedure Init(InitX, InitY: Integer);
        function GetX: Integer;
        function GetY: Integer;
    private
        X, Y: Integer;
    end;

```

В секции `private` могут находиться и методы объекта.

Полное описание методов, то есть описание их реализации, должно находиться после описания объекта. Имена методов составные и складываются из имени объекта и имени метода, разделенных точкой:

```

procedure Location.Init(InitX, InitY: Integer);
begin
    X := InitX;
    Y := InitY;
end;

function Location.GetX: Integer;
begin
    GetX := X;
end;

function Location.GetY: Integer;
begin
    GetY := Y;
end;

```

После того как объект описан, в программе можно использовать его экземпляры, то есть переменные указанного объектного типа:

```

var
    GrMarker : Location;

```

## Инкапсуляция

*Инкапсуляция* является важнейшим свойством объектов, на котором строится объектно-ориентированное программирование. Инкапсуляция заключается в том, что объект скрывает в себе детали, которые несущественны для использования объекта. В традиционном подходе к программированию с использованием глобальных переменных программист не был застрахован от ошибок, связанных с использованием процедур, не предназначенных для обработки данных, связанных с этими переменными. Предположим, например, что имеется «не-ООП» программа, предназначенная для начисления заработкающей платы сотрудникам некоей организации, а в программе той имеются два массива. Один массив хранит величину заработкающей платы, а другой — телефонные номера сотрудников (для составления отчета для налоговой инспекции). Что произойдет, если программист случайно перепутает эти массивы? Очевидно, для бухгалтерии начнутся тяжелые времена. «Жесткое» связывание данных и процедур их обработки в одном объекте позволяет избежать неприятностей такого рода. Инкапсуляция и является средством организации доступа к данным *только через соответствующие методы*.

В нашем примере описания объекта процедура инициализации `Init` и функции `GetX` и `GetY` уже не существуют как отдельные самостоятельные объекты. Это неотъемлемые части объектного типа `Location`. Если в программе имеется описание нескольких переменных указанного типа, то для каждой переменной резервируется своя собственная область памяти для хранения данных, а указатели на точки входа в процедуру и функции — общие. Вызов каждого метода возможен только с помощью составного имени, явно указывающего, для обработки каких данных предназначен данный метод.

## Наследование

*Наследование* — это еще одно базовое понятие ООП. Наследование позволяет определять новые объекты, используя свойства прежних, дополняя или изменяя их. Объект-наследник получает все поля и методы «родителя», к которым он может добавить свои собственные поля и методы или заменить («перекрыть») их своими методами. Пример описания объекта-наследника дается ниже:

```
type
  Point = object(Location)
    Visible : Boolean;
    procedure Init(InitX, InitY : Integer);
    procedure Show;
    procedure Hide;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;
```

Наследником здесь является объект `Point`, описывающий графическую точку, а родителем — объект `Location`. Наследник не содержит описания полей и методов

родителя. Имя последнего указывается в круглых скобках после слова `object`. Из методов наследника можно вызывать методы родителя. Для создания наследника не требуется иметь исходный текст объекта-родителя. Объект-родитель может быть в составе уже оттранслированного модуля.

В Турбо Паскале имеется зарезервированное слово `inherited`, которое используется для определения объекта-родителя. Соответствующее предложение размещается в описании объекта-наследника, например:

```
constructor NumField.Init(X_coord, Y_coord, Len : Integer; LMin, LMax : Longint);
begin
  inherited Init(X_coord, Y_coord, Len);
  Min := LMin;
  Max := LMax;
end;
```

С назначением зарезервированного слова `constructor` мы познакомимся позже. Предложение `inherited`, разумеется, нельзя использовать в тех объектах, которые не имеют родителя.

В чем привлекательность наследования? Если некий объект уже был определен и отложен, он может быть использован и в других программах. При этом может оказаться, что новая задача отличается от предыдущей, и возникает необходимость некоторой модификации как данных, так и методов их обработки. Программисту приходится решать дилемму — создавать объект заново или использовать результаты предыдущей работы, применяя механизм наследования. Первый путь менее эффективен, так как требует дополнительных затрат времени на отладку и тестирование. Во втором случае часть этой работы оказывается выполненной, что сокращает время на разработку новой программы. Программист при этом может и не знать деталей реализации объекта-родителя.

В нашем примере к объекту, связанному с определением положения графического элемента, просто добавилось новое поле, описывающее признак видимости графической точки, и несколько новых методов, связанных с режимами отображения точки и ее преобразованиями.

## Виртуальные методы

Наследование позволяет создавать иерархические, связанные отношениями подчинения, структуры данных. Следует, однако, заметить, что при использовании этой возможности могут возникать проблемы. Предположим, что в нашей графической программе необходимо определить объект `Circle`, который является потомком объекта `Point`:

```
type
  Circle = object (Point)
    Radius: Integer;
    procedure Show;
    procedure Hide;
```

```

procedure Expand(ExpandBy: Integer);
procedure Contract(ContractBy: Integer);
end;

```

Новый объект **Circle** соответствует окружности. Поскольку свойства окружности отличаются от свойств точки, в объекте-наследнике придется изменить процедуры **Show** и **Hide**, которые отображают окружность и удаляют ее изображение с экрана. Может оказаться, что метод **Init** объекта **Circle**, унаследованный от объекта **Point**, также использует методы **Show** и **Hide**, ведь во время трансляции объекта **Point** использовались ссылки на старые методы. Очевидно, в объекте **Circle** они работать не будут. Можно, конечно, попытаться «перекрыть» метод **Init**. Чтобы это сделать, нам придется полностью воспроизвести текст метода. Это усложняет работу, да и не всегда возможно, поскольку исходного текста может не оказаться под рукой (если объект-родитель находится в уже оттранслированном модуле).

Для решения этой проблемы используются *виртуальные методы*. Связь между виртуальными методами и вызывающими их процедурами устанавливается не во время трансляции (это называется *ранним связыванием*), а во время выполнения программы (*позднее связывание*).

Чтобы использовать виртуальные методы, необходимо в описании объекта после заголовка метода добавить ключевое слово **virtual**. Заголовки виртуальных методов родителя и наследника должны в точности совпадать. Инициализация экземпляра объекта, имеющего виртуальные методы, должна выполняться с помощью специального метода — *конструктора*. Конструктор обычно присваивает полям объекта начальные значения и выполняет другие действия по инициализации объекта. В заголовке метода-конструктора слово **procedure** заменяется словом **constructor**. Действия, обратные действиям конструктора, выполняет еще один специальный метод — *деструктор*. Он описывается словом **destructor**. С учетом всех этих замечаний описания объектов **Point** и **Circle** следуют изменить следующим образом:

```

type
  Point = object (Location)
    Visible : Boolean;
    constructor Init(InitX, InitY : Integer);
    destructor Done; virtual;
    procedure Show; virtual;
    procedure Hide; virtual;
    function IsVisible : Boolean;
    procedure MoveTo(NewX, NewY : Integer);
  end;

type
  Circle = object (Point)
    Radius: Integer;
    constructor Init(InitX, InitY : Integer; InitRadius : Integer);
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure Expand(ExpandBy : Integer); virtual;
  end;

```

```
procedure Contract(ContractBy : Integer); virtual;
end;
```

Конструктор выполняет действия по подготовке позднего связывания. Эти действия заключаются в создании указателя на *таблицу виртуальных методов*, которая в дальнейшем используется для поиска методов. Таблица содержит адреса всех виртуальных методов. При вызове виртуального метода по его имени определяется адрес, а затем по этому адресу передается управление.

У каждого объектного типа имеется своя собственная таблица виртуальных методов, что и позволяет одному и тому же оператору вызывать разные процедуры. Если имеется несколько экземпляров объектов одного типа, то недостаточно вызвать конструктор для одного из них, а затем просто скопировать этот экземпляр во все остальные. Каждый объект должен иметь свой собственный конструктор, который вызывается для каждого экземпляра. В противном случае возможен сбой в работе программы.

Замечу, что конструктор и деструктор могут быть и «пустыми», то есть не содержать операторов. Весь необходимый код в этом случае создается при трансляции ключевых слов `constructor` и `destructor`.

## Динамическое создание объектов

Переменные объектного типа могут быть *динамическими*, то есть размещаться в памяти только на время их использования. Для работы с динамическими объектами используется расширенный синтаксис процедур `New` и `Dispose`. Обе процедуры в этом случае содержат в качестве второго параметра вызов конструктора или деструктора для выделения или освобождения памяти переменной объектного типа:

`New(P, Construct)`

или

`Dispose(P, Destruct)`

где `P` — указатель на переменную объектного типа, а `Construct` и `Destruct` — конструктор и деструктор этого типа.

Действие процедуры `New` в случае расширенного синтаксиса равносильно действию следующей пары операторов:

```
New(P);
P^.Construct;
```

Эквивалентом `Dispose` является следующее:

```
P^.Dispose;
Dispose(P);
```

Применение расширенного синтаксиса не только улучшает читаемость исходного кода, но и генерирует более короткий и эффективный исполняемый код.

## Полиморфизм

**Полиморфизм** заключается в том, что одно и то же имя может соответствовать различным действиям в зависимости от типа объекта. В тех примерах, которые рассматривались ранее, полиморфизм проявлялся в том, что метод `Init` действовал по-разному в зависимости от того, являлся объект точкой или окружностью. Полиморфизм напрямую связан с механизмом позднего связывания. Решение о том, какая операция должна быть выполнена в конкретной ситуации, принимается во время выполнения программы.

Следующий вопрос, связанный с использованием объектов, заключается в совместности объектных типов. Следует заметить, что строгие правила языка Паскаль, касающиеся дисциплины типов, несколько ослаблены в применении к объектам. Это — следствие наследования. Полезно знать следующее. Наследник сохраняет свойства совместности с другими объектами своего родителя. В правой части оператора присваивания вместо типов родителя можно использовать типы наследника, но не наоборот. Таким образом, в нашем примере допустимы присваивания

```
var
  Alocation : Location;
  Apoint    : Point;
  Acircle   : Circle;

  Alocation := APoint;
  APoint   := ACircle;
  Alocation := ACircle;
```

Дело в том, что наследник может быть более сложным объектом, содержащим дополнительные поля и методы, поэтому присваивание значения экземпляра объекта-родителя экземпляру объекта-наследника может оставить некоторые поля неопределенными и, следовательно, представляет потенциальную опасность. При выполнении оператора присваивания копируются только те поля данных, которые являются общими для обоих типов.

## Модуль `matrices`

Изучая методы программирования трехмерных изображений, мы столкнулись с необходимостью матрично-векторных вычислений. Здесь в качестве примера применения технологии объектно-ориентированного программирования мы рассмотрим модуль, содержащий процедуры матричной алгебры.

Будем считать, что с основными операциями матричной алгебры мы уже знакомы. Будем различать матрицы с рациональными (представленными в виде отношения двух целых чисел), вещественными и комплексными коэффициентами. Собственно, проблема заключается в том, чтобы запрограммировать действия с

матрицами один раз и навсегда, независимо от происхождения коэффициентов матриц.

Эта задача решается в модуле `matrices`. Модуль демонстрирует основные особенности ООП, о которых речь шла выше. Интерфейсная секция этого модуля приведена в листинге 9.1. Замечу, что в данном модуле предполагается, что матрицы имеют размер  $3 \times 3$  (константа `max`). Читатель может подумать о возможности модификации модуля `matrices` на случай матриц произвольной размерности.

В начале интерфейсной секции описываются скалярные типы `rational` (запись, поля которой `num` и `den` описывают, соответственно, числитель и знаменатель произвольного элемента рациональной матрицы) и `complex` (запись, поля которой соответствуют вещественной и мнимой частям комплексного значения элемента), а также матричные типы с рациональными, комплексными и вещественными коэффициентами. Используя приведение типов для указателей, мы сможем в дальнейшем ссылаться на вещественные, комплексные элементы или элементы других типов, которые мы собираемся использовать в матричных расчетах.

Далее следуют описания объектов. Объектный тип `matrix_obj` содержит поле `M` типа `matrix`. Оно предназначено для хранения матрицы. Кроме этого поля, в описании типа перечислены методы, соответствующие основным матричным операциям и операциям над скалярными значениями, а также инициализация матрицы нулевыми значениями. Каждый экземпляр `matrix_object` содержит единственное поле `M`.

Обратим внимание на связанные между собой методы: виртуальный `scalar_sum` и не виртуальный `matrix_sum`. Входными параметрами процедуры `scalar_sum` являются два указателя, пока еще не ссылающиеся ни на какой тип данных, а выходным — другой указатель, которым можно пользоваться для указания на сумму значений. На входе `matrix_sum` имеются две матрицы `A` и `B` — слагаемые, а результат их матричного сложения заносится в поле `M`. При реализации метода `matrix_sum` будет вызываться метод `scalar_sum`, который предназначен для вычисления суммы указателей на вещественные, рациональные и комплексные значения. Поскольку метод `scalar_sum` объявлен как виртуальный, метод `matrix_sum` будет использовать его описание. Это позволит нам определить сумму двух матриц, не зная заранее значений их коэффициентов.

Аналогично, процедура `matrix_product` вызывает виртуальный метод `scalar_product`. Чтобы инициализировать нулевым значением сумму произведений перед вычислением произведения матриц, процедура обращается к виртуальному методу `set_to_zero`. Метод `print` печатает массив строк. Он вызывает виртуальный метод `convert_to_string`, преобразующий скалярное значение в строковое.

Перейдем к объектному типу `real_matrix_obj`, обращая внимание на начало его описания: `real_matrix_obj = object (matrix_obj)`. Это означает, что объект `real_matrix_obj` является наследником объекта `matrix_obj` и в таком качестве наследует все поля и методы родителя. Объекты, соответствующие рациональным и комплексным матрицам, также являются наследниками объекта `matrix_obj`.

**Листинг 9.1. Интерфейсная секция модуля matrices**

```

{$r+}
unit matrices;

interface

uses crt;

const
  max = 3;

type
  str10 = string[10];
  str30 = string[30];
  matrix = array[1..max, 1..max] of pointer;
  rational = record
    num, den: longint;
  end;
  rational_ptr = ^rational;
  rational_matrix = array[1..max, 1..max] of rational;
  real_ptr = ^real;
  real_matrix = array[1..max, 1..max] of real;
  complex = record
    re, im : real;
  end;
  complex_ptr = ^complex;
  complex_matrix = array[1..max, 1..max] of complex;
  matrix_obj = object
    M : matrix;
    constructor init;
    procedure set_to_zero(var p : pointer); virtual;
    procedure scalar_sum(const p, q : pointer; var r : pointer); virtual;
    procedure scalar_product(const p, q : pointer;
                           var r : pointer); virtual;
    procedure matrix_sum(const A, B : matrix);
    procedure matrix_product(const A, B : matrix);
    procedure convert_to_string(p : pointer; var tt : str30); virtual;
    procedure print(name : str10);
    destructor done;
  end;

  real_matrix_obj = object(matrix_obj)
    constructor init(M0 : real_matrix);
    procedure set_to_zero(var p : pointer); virtual;
    procedure scalar_sum(const p, q : pointer; var r : pointer); virtual;
    procedure scalar_product(const p, q : pointer;
                           var r : pointer); virtual;
    procedure convert_to_string(p : pointer; var tt : str30); virtual;
    procedure print(name : str10);
    destructor done;
  end;

```

```

end;

rational_matrix_obj = object(matrix_obj)
constructor init(M0 : rational_matrix);
procedure set_to_zero(var p : pointer); virtual;
procedure scalar_sum(const p, q : pointer; var r : pointer); virtual;
procedure scalar_product(const p, q : pointer;
var r : pointer); virtual;
procedure convert_to_string(p : pointer; var tt : str30); virtual;
procedure print(name : str10);
destructor done;
end;

complex_matrix_obj = object(matrix_obj)
constructor init(M0 : complex_matrix);
procedure set_to_zero(var p : pointer); virtual;
procedure scalar_sum(const p, q : pointer; var r : pointer); virtual;
procedure scalar_product(const p, q : pointer;
var r : pointer); virtual;
procedure convert_to_string(p : pointer; var tt : str30); virtual;
procedure print(name : str10);
destructor done;
end;

```

Перейдем к реализации методов и начнем с методов `matrix_obj`. Соответствующие исходные тексты приведены в листинге 9.2. Обратим внимание на то, что методы с одинаковыми именами различаются посредством использования составных имен: `matrix_obj.init` для метода `init` объектного типа `matrix_obj` и т. д. Конструктор `init` и деструктор `done`, хотя и не содержат операторов, выполняют большую скрытую работу по управлению памятью. Три виртуальных метода — `set_to_zero`, `scalar_sum` и `scalar_product` — не выполняют никаких действий. Они просто являются шаблонами для последующих описаний. Полезно разобрать реализацию методов `matrix_sum` и `matrix_product`, вызывающих эти виртуальные методы. Аналогично, метод `convert_to_string` пока не выполняет никакой работы, но он будет использоваться в методе `print`. В `matrix_obj.print` используется функция `WhereY` (модуль CRT), которая возвращает значение *y*-координаты текущего положения курсора.

**Листинг 9.2.** Реализация методов `matrix_obj` и `real_matrix_obj` модуля `matrices`

```

implementation

constructor matrix_obj.init;
begin
end;

procedure matrix_obj.set_to_zero(var 'p : pointer);
begin
end;

procedure matrix_obj.scalar_sum(const p, q : pointer;

```

```
var r : pointer);
begin
end;

procedure matrix_obj.scalar_product(const p, q : pointer; var r : pointer);
begin
end;

procedure matrix_obj.matrix_sum(const A, B : matrix);
var
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      scalar_sum(A[i, j], B[i, j], M[i, j]);
end;

procedure matrix_obj.matrix_product(const A, B : matrix);
var
  i, j, k : word;
  prod, sum : pointer;
begin
  getmem(prod, SizeOf(M[1, 1]));
  for i := 1 to max do
    for k := 1 to max do
      begin
        sum := M[i, k];
        set_to_zero(sum);
        for j := 1 to max do
          begin
            scalar_product(A[i, j], B[j, k], prod);
            scalar_sum(sum, prod, sum);
          end;
        end;
      end;
  freemem(prod, SizeOf(M[1, 1]));
end;

procedure matrix_obj.convert_to_string(p : pointer; var tt : str30);
begin
end;

procedure matrix_obj.print(name : str10);
var
  i, j, k, hold_y : word;
  temp : str30;
begin
  hold_y := WhereY;
  gotoxy(1, hold_y + (max + 1) div 2);
  write(name, ' = ');
  k := length(name) + 4;
  gotoxy(k, hold_y); write('v');
  for i := 1 to max do
```

```
begin
  gotoxy(k, hold_y + i);
  write('|');
  for j := 1 to max do
    begin
      convert_to_string(M[i, j], temp);
      write(temp);
    end;
    write('|');
  end;
  gotoxy(k, max + hold_y + 1); write('x');
  gotoxy(k + length(temp) * max + 1, hold_y);
  write('J');
  gotoxy(k + length(temp) * max + 1, max + hold_y + 1);
  writeln('^');
end;

destructor matrix_obj.done;
begin
end;

constructor real_matrix_obj.init(M0 : real_matrix);
var
  i, j : word;
begin
  inherited init;
  for i := 1 to max do
    for j := 1 to max do
      begin
        new(real_ptr(M[i, j]));
        real_ptr(M[i, j])^ := M0[i, j];
      end;
end;

procedure real_matrix_obj.set_to_zero(var p : pointer);
begin
  real_ptr(p)^ := 0.0;
end;

procedure real_matrix_obj.scalar_sum(const p, q : pointer; var r : pointer);
begin
  if not Assigned(r) then
    new(real_ptr(r));
  real_ptr(r)^ := real_ptr(p)^ + real_ptr(q)^;
end;

procedure real_matrix_obj.scalar_product(const p, q : pointer;
  var r : pointer);
begin
  if not Assigned(r) then
    new(real_ptr(r));
  real_ptr(r)^ := real_ptr(p)^ * real_ptr(q)^;
end;
```

```

procedure real_matrix_obj.convert_to_string(p : pointer; var tt : str30);
begin
  str(real_ptr(p)^:9:4, tt);
end;

destructor real_matrix_obj.done;
var
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      begin
        dispose(real_ptr(M[i, j]));
      end;
  inherited done;
end;

```

Остается рассмотреть реализацию методов `real_matrix_obj`, `rational_matrix_obj` и `complex_matrix_obj` (листинг 9.3). Остановимся только на том из них, который предназначен для работы с рациональными матрицами. Остальные оставим читателю для самостоятельной работы. Вначале рассмотрим реализации конструктора `init` и деструктора `done`. Замечу, что `init` прежде всего вызывает своего родителя `matrix_obj.init`, который в общем случае выделяет объекту память. Для объявления объекта-родителя используется зарезервированное слово `inherited`. Затем `init` выделяет для данных память посредством обращений к процедуре `new`. Деструктор `done` высвобождает эту память. Далее, метод `set_to_zero` инициализирует рациональное число, на которое ссылается указатель, значением 0.

Вспомогательная процедура `lowest_terms` сокращает числитель и знаменатель рационального значения на общий делитель. Обратите внимание на реализации виртуальных методов `scalar_sum` и `scalar_product`, которые, в отличие от своих родителей, здесь стали содержательными.

**Листинг 9.3.** Реализация методов `rational_matrix_obj` и `complex_matrix_obj` модуля `matrices`

```

constructor rational_matrix_obj.init(M0 : rational_matrix);
var
  i, j : word;
begin
  inherited init;
  for i := 1 to max do
    for j := 1 to max do
      begin
        new(rational_ptr(M[i, j]));
        rational_ptr(M[i, j])^ := M0[i, j];
      end;
end;

procedure rational_matrix_obj.set_to_zero(var p : pointer);
begin
  rational_ptr(p)^.num := 0;
  rational_ptr(p)^.den := 1;
end;

```

```

procedure lowest_terms(var x, y : longint);
var
  a, b, r : longint;
begin
  a := x;
  b := y;
  a := Abs(a);
  repeat
    r := a mod b;
    a := b;
    b := r;
  until r = 0;
  x := x div a;
  y := y div a;
end;

procedure rational_matrix_obj.scalar_sum(const p, q : pointer;
                                         var r : pointer);
var
  x, y, z : rational;
begin
  x := rational_ptr(p)^;
  y := rational_ptr(q)^;
  z.num := x.num * y.den + y.num * x.den;
  z.den := x.den * y.den;
  lowest_terms(z.num, z.den);
  if not Assigned(r) then
    new(rational_ptr(r));
  rational_ptr(r)^ := z;
end;

procedure rational_matrix_obj.scalar_product(const p, q : pointer;
                                              var r : pointer);
var
  x, y, z : rational;
begin
  x := rational_ptr(p)^;
  y := rational_ptr(q)^;
  z.num := x.num * y.num;
  z.den := x.den * y.den;
  lowest_terms(z.num, z.den);
  if not Assigned(r) then
    new(rational_ptr(r));
  rational_ptr(r)^ := z;
end;

procedure rational_matrix_obj.convert_to_string(p : pointer; var tt : str30);
var
  t1, t2 : string[11];
  j : word;
begin
  str(rational_ptr(p)^.num, t1);
  str(rational_ptr(p)^.den, t2);
  tt := t1 + '/' + t2;
  for j := length(t1) + length(t2) to 12 do tt := ' ' + tt;
end;

```

```
destructor rational_matrix_obj.done;
var
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      begin
        dispose(rational_ptr(M[i, j]));
      end;
  inherited done;
end;

constructor complex_matrix_obj.init(M0 : complex_matrix);
var
  i, j : word;
begin
  inherited init;
  for i := 1 to max do
    for j := 1 to max do
      begin
        new(complex_ptr(M[i, j]));
        complex_ptr(M[i, j])^ := M0[i, j];
      end;
end;

procedure complex_matrix_obj.set_to_zero(var p : pointer);
begin
  complex_ptr(p)^.re := 0.0;
  complex_ptr(p)^.im := 0.0;
end;

procedure complex_matrix_obj.scalar_sum(const p, q : pointer; var r : pointer);
var
  x, y, z : complex;
begin
  x := complex_ptr(p)^;
  y := complex_ptr(q)^;
  z.re := x.re + y.re;
  z.im := x.im + y.im;
  if r = nil then
    new(complex_ptr(r));
  complex_ptr(r)^ := z;
end;

procedure complex_matrix_obj.scalar_product(const p, q : pointer;
  var r : pointer);
var
  x, y, z : complex;
begin
  x := complex_ptr(p)^;
  y := complex_ptr(q)^;
  z.re := x.re * y.re - x.im * y.im;
  z.im := x.re * y.im + x.im * y.re;
  if not Assigned(r) then
    new(complex_ptr(r));
  complex_ptr(r)^ := z;
end;
```

```

end;

procedure complex_matrix_obj.convert_to_string(p : pointer; var tt : str30);
var
  t1, t2 : string[6];
begin
  str(complex_ptr(p)^.re:6:4, t1);
  str(complex_ptr(p)^.im:6:4, t2);
  tt := ' ' + t1 + ' + ' + t2 + 'i';
end;

destructor complex_matrix_obj.done;
var
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      begin
        dispose(complex_ptr(M[i, j]));
      end;
    inherited done;
end;
end.

```

Программа `obj_demo` (листинг 9.4) демонстрирует возможности модуля `matrices` для работы с вещественными, рациональными и комплексными матрицами.

**Листинг 9.4.** Программа, демонстрирующая использование модуля `matrices`

```

{$r+}
program obj_demo;
uses crt, matrices;

procedure test_real;
var
  A0, n0, C0 : real_matrix;
  A, B, C : real_matrix_obj;
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      begin
        A0[i, j] := i * j;
        n0[i, j] := i + j;
        C0[i, j] := 0.0;
      end;
  A.init(A0);
  B.init(n0);
  C.init(C0);
  A.print('A');
  B.print('B');
  C.matrix_sum(A.M, B.M);
  C.print('A + B');
  C.matrix_product(A.M, B.M);
  C.print('A * B');

```

```
A.done;
B.done;
C.done;
end;

procedure test_rational;
var
  A0, B0, C0 : rational_matrix;
  A, B, C : rational_matrix_obj;
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      begin
        A0[i, j].num := i + j - 1;
        A0[i, j].den := 1;
        B0[i, j].num := 1;
        B0[i, j].den := i + j - 1;
        C0[i, j].num := 1;
        C0[i, j].den := 1;
      end;
  A.init(A0);
  B.init(B0);
  C.init(C0);
  A.print('A');
  B.print('B');
  C.matrix_sum(A.M, B.M);
  C.print('A + B');
  C.matrix_product(A.M, B.M);
  C.print('A * B');
  A.done;
  B.done;
  C.done;
end;

procedure test_complex;
var
  A0, B0, C0 : complex_matrix;
  A, B, C : complex_matrix_obj;
  i, j : word;
begin
  for i := 1 to max do
    for j := 1 to max do
      begin
        A0[i, j].re := 1.0; A0[i, j].im := 0.0;
        B0[i, j].re := i;   B0[i, j].im := j;
        C0[i, j].re := 0.0; C0[i, j].im := 0.0;
      end;
  A.init(A0);
  B.init(B0);
  C.init(C0);
  A.print('A');
  B.print('B');
  C.matrix_sum(A.M, B.M);
  C.print('A + B');
```

```
C.matrix_product(A.M, B.M);
C.print('A * B');
A.done;
B.done;
C.done;
end;
begin
  clrscr;
  writeln('Вещественные матрицы');
  test_real;
  write('Нажмите <Enter>');
  readln;
  clrscr;
  writeln('Рациональные матрицы');
  test_rational;
  write('Нажмите <Enter>');
  readln;
  clrscr;
  writeln('Комплексные матрицы');
  test_complex;
  write('Нажмите <Enter>');
  readln;
end.
```

«Наследник» Турбо Паскаля, продукт фирмы Borland (ныне Inprise) под названием Delphi — система визуальной разработки Windows-приложений, основывается на более развитом диалекте Паскаля, который называется Object Pascal. Он отличается от Borland Pascal 7.0 главным образом в части ООП.

#### Упражнение 9.1

Разберите реализацию методов в модуле `matrices`.

#### Упражнение 9.2

Разберите работу с объектами в программе `obj_demo`.

## Что нового мы узнали?

- Познакомились с основными идеями объектно-ориентированного программирования.
- Познакомились с понятиями объекта, метода, инкапсуляции, наследования и полиморфизма.
- Познакомились с виртуальными методами.
- Познакомились с примером использования ООП для решения вычислительных задач.



## Турбо Паскаль для вычислений

- 
- Вычисления, связанные с теорией чисел
  - Вычисления с полиномами
  - Линейная алгебра
  - Решение нелинейных уравнений
  - Вычисление интегралов
  - Решение дифференциальных уравнений
-

**Т**ема данного урока — решение задач вычислительной математики. Для успешной работы с материалом требуется знание основ высшей математики и численных методов.

## Вычисления, связанные с теорией чисел

Теория чисел предоставляет большие возможности для «вычислительных экспериментов» и служит богатым источником интересных вычислительных задач. Здесь мы рассмотрим несколько простых примеров.

Первая задача состоит в том, чтобы пересчитать все целые числа, находящиеся в интервале  $[m, n]$ , которые делятся нацело на 3 или на 7, но не на оба эти числа одновременно. Предлагаются два решения, которые реализованы в программе *counting* (листинг 10.1).

### Листинг 10.1. Программа *counting*

```
program counting;
uses crt;

function counter1(m, n : word): word;
var
  c, j : word;
begin
  c := 0;
  for j := m to n do
    if (j mod 3 = 0) xor (j mod 7 = 0) then
      inc(c);
  counter1 := c;
end;

function counter2(m, n : word) : word;
begin
```

```

counter2 := n div 7 + n div 3 - 2 * (n div 21)
  - (m - 1) div 7 - (m - 1) div 3 + 2 * ((m - 1) div 21);
end;

begin
  clrscr;
  writeln(counter1(100, 40000));
  writeln(counter2(100, 40000));
  writeln('Нажмите <Enter>');
  readln;
end.

```

В процедуре counter1 используется простой перебор целых значений от минимального до максимального. Для каждого значения проводится проверка сложного условия, после чего значение счетчика увеличивается на единицу, если это условие выполнено. Во втором случае (функция counter2) используется явная математическая формула, поэтому второе решение представляется оптимальным. Так ли это? Читателю предоставляется возможность это проверить, проводя измерения затрат времени на выполнение обеих функций (разумеется, программными средствами). Основной урок, который следует извлечь из этого простого примера, состоит в том, что при решении вычислительных задач предварительное математическое исследование, как правило, приводит к оптимальному программному решению. Программист, не знающий математики, — это неграмотный писатель, которому уготована разве что роль народного сказителя!

В программе gcd (листинг 10.2) тремя разными способами находится наибольший общий делитель двух неотрицательных целых чисел, отличных от нуля. В главе, посвященной рекурсии, мы уже решали эту задачу. Здесь приведены новые варианты решения. В функциях gcd1 и gcd2 используется рекурсия, но вторая из них работает быстрее, чем первая. Объясняется это тем, что в результате модификации алгоритма глубина рекурсии (то есть количество рекурсивных вызовов функции) уменьшается с 55 до 7 — почти в восемь раз! Функция gcd3 является итерационной, а работает она еще быстрее, так как цикл внутри этой функции выполняется лишь 5 раз. Неплохая математическая задача — доказать, что все три алгоритма дают одинаковый результат.

#### Листинг 10.2. Программа вычисления наибольшего общего делителя

```

program gcd;

uses crt;

function gcd1(m, n : word) : word;
begin
  if n = 0 then
    gcd1 := m
  else
    if m < n then
      gcd1 := gcd1(n, m)
    else

```

```

gcd1 := gcd1(m - n, n)
end;

function gcd2(m, n : word) : word;
begin
  if n = 0 then
    gcd2 := m
  else
    if m < n then
      gcd2 := gcd2(n, m)
    else
      gcd2 := gcd2(n, m mod n);
end;

function gcd3(m, n : word) : word;
var
  r : word;
begin
  if m < n then
    begin
      r := n;
      n := m;
      m := r;
    end;
  while n > 0 do
    begin
      r := m mod n;
      m := n;
      n := r;
    end;
  gcd3 := m;
end;

begin
  clrscr;
  writeln('Первый вариант ', gcd1(29008, 36368));
  writeln('Второй вариант ', gcd2(29008, 36368));
  writeln('Третий вариант ', gcd3(29008, 36368));
  writeln;
  writeln('Нажмите <Enter>');
  readln;
end.

```

## Простые числа

*Простым* называется число  $p$ , большее 1 и не имеющее положительных целых делителей, кроме 1 и  $p$ . Чтобы найти все простые числа вплоть до некоторого наибольшего значения  $max$ , используются алгоритмы отсева. Одним из наиболее известных алгоритмов такого рода является «решето Эратосфена». Отсев здесь происходит следующим образом. Первое простое число, согласно определению,

двойка. Отбросим все числа, кратные первому простому, то есть двойке. Наименьшее оставшееся число 3 является вторым простым числом. Затем отбросим все числа, кратные этому простому числу. Наименьшее оставшееся число (5) будет следующим простым числом и т. д. Алгоритм «решето Эратосфена» реализован в программе *eratosphen1* (листинг 10.3).

**Листинг 10.3.** Программа «Решето Эратосфена» (первый вариант)

```
program eratosphen1;
uses crt;

const max = 1000;

var
  b, j, k : word;
  flag : array[1..max] of boolean;

begin
  clrscr;
  flag[1] := false;
  for j := 2 to max do
    flag[j] := true;
  b := trunc(sqrt(max));
  k := 0;
  while k <= b do
    begin
      repeat
        inc(k)
      until flag[k];
      j := 2 * k;
      while j <= max do
        begin
          flag[j] := false;
          j := j + k;
        end;
    end;
  for j := 1 to max do
    if flag[j] then
      write(j:8);
  writeln;
  write('Нажмите <Enter>');
  readln;
end.
```

В следующей программе (*eratoshen2*, листинг 10.4) реализован другой алгоритм поиска простых чисел, который может использоваться для отыскания большого количества простых чисел, поскольку его требования к памяти пропорциональны  $\sqrt{m}$ , где  $m$  — количество простых чисел. Для предыдущей программы эта зависимость была пропорциональна  $m$ . Время выполнения программы пропорционально  $m^{3/2}$ . Входным параметром является значение  $\sqrt{m}$  (константа *n*), а результат работы —  $m-1$  простое число.

В основе программы `eratosphen2` лежит следующая теория. Пусть

$$p_1 < p_2 < \dots$$

обозначает последовательность простых чисел. Известно, что для каждого  $n \geq 2$  выполнено неравенство  $p_{n^2-1} < p_n^2$ . Таким образом, чтобы найти первые  $n^2 - 1$  простых чисел, мы должны отбросить все числа, кратные простым числам  $p_1 \dots p_n$ . Для любого из этих простых чисел  $p$  отсеивание можно начинать с  $p^2$ , так как любое меньшее кратное делится на какое-то меньшее простое число и, следовательно, уже отброшено, так что нет нужды повторять уже сделанную работу. Для каждого из обработанных ранее простых чисел соответствующее наибольшее кратное ему число хранится в массиве `mult`, объем которого определяет требования к памяти. Массив `prime` не нужен, если для сохранения результатов работы используется, например, запись в файл. И, конечно, если мы перейдем к поиску больших простых чисел, многие переменные придется переопределить из типа `Word` в тип `Longint`.

**Листинг 10.4.** Программа «Решето Эратосфена» (второй вариант)

```
program eratosphen2;
uses crt;
const
  n = 20;
  m = n * n;
var
  p, s, i, j, k : word;
  accept: boolean;
  prime : array[1..m - 1] of word;
  mult : array[1..n - 1] of word;
  old_mode : integer;
begin
  clrscr;
  prime[1] := 2;
  p := 2;
  k := 1;
  s := sqr(prime[1]);
  for i := 2 to m - 1 do
  begin
    repeat
      inc(p);
      if s <= p then
        begin
          mult[k] := s;
          inc(k);
          s := sqr(prime[k]);
        end;
    accept := true;
```

```

for j := 1 to k - 1 do
begin
  if mult[j] < p then
    mult[j] := mult[j] + prime[j];
  accept := (mult[j] > p);
  if not accept then
    break;
end;
until accept;
prime[i] := p;
end;
writeln;
writeln;
old_mode := lastmode;
textmode(C080 + font8x8);
for i := 1 to 10 do
  write(i:8); writeln;
textcolor(yellow);
for i := 1 to m - 1 do
  write(prime[i]:8);
writeln;
writeln;
textcolor(lightgray);
write('Нажмите <Enter>');
readln;
textmode(old_mode);
end.

```

В программе `eratosphen2` переменная `p` является очередным числом, проверяемым на «простоту», а переменная `s` используется для хранения квадратов простых чисел. Массив `mult[k]` содержит значения, кратные `prime[k]`, начиная с  $s = \text{prime}^2[k]$ . Чтобы разместить на экране все найденные простые числа, программа производит переключение в текстовый режим с 50 строками и 80 позициями в каждой строке. После завершения выполнения программы восстанавливается исходный режим.

### Упражнение 10.1

Напишите программу, которая находит целые числа, являющиеся суммой квадратов трех целых чисел.

### Упражнение 10.2

Булев массив `flag` в программе `eratosphen1` занимает 1000 байт. Если потребуется найти, скажем, 100 000 простых чисел, величина этого массива превысит 64-килобайтный предел для размера переменной. На самом же деле, чтобы хранить информацию о булевом значении, нужен только 1 бит, поэтому достаточно было бы всего 12 500 байт. Модифицируйте соответствующим образом программу `eratosphen1`.

## Вычисления с полиномами

В этом разделе мы разберем модуль, предназначенный для вычисления значений классических ортогональных полиномов в заданной точке. Необходимо напомнить их рекурсивные определения. Полиномы Чебышева определяются следующим образом:

$$\begin{aligned}T_0(x) &= 1, \\T_1(x) &= x, \\T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x).\end{aligned}$$

Полиномы Лежандра:

$$\begin{aligned}P_0(x) &= 1, \\P_1(x) &= x, \\P_n(x) &= [(2n - 1)x P_{n-1}(x) - (n - 1) P_{n-2}(x)]/n.\end{aligned}$$

Полиномы Эрмита:

$$\begin{aligned}H_0(x) &= 1, \\H_1(x) &= 2x, \\H_n(x) &= [x H_{n-1}(x) - (n - 1) H_{n-2}(x)]/n.\end{aligned}$$

И, наконец, полиномы Лагерра определяются так:

$$\begin{aligned}L_0(x) &= 1, \\L_1(x) &= x, \\L_n(x) &= [(2n - 1 - x) L_{n-1}(x) - (n - 1) L_{n-2}(x)]/n.\end{aligned}$$

В модуле `orthpoly` (листинг 10.5) дана реализация только полиномов Чебышева, реализация прочих ортогональных полиномов аналогична.

### Листинг 10.5. Модуль `orthpoly`

```
unit orthpoly;

interface

function Chebyshev(degree : word; x : real) : real;

implementation

function Chebyshev(degree : word; x : real) : real;
var
  Pn, Pn_1, Pn_2 : real;
  k : word;
begin
  if degree = 0 then
    Chebyshev := 1.0
  else
    begin
      Pn := 1.0;
      Pn_1 := 0.0;
      Pn_2 := 0.0;
      for k := 1 to degree do
        begin
          Pn := 2 * x * Pn_1 - Pn_2;
          Pn_1 := Pn;
          Pn_2 := Pn_1;
        end;
    end;
  end;
```

```

else
begin
  Pn_1 := 1.0;
  Pn := x;
  for k := 2 to degree do
  begin
    Pn_2 := Pn_1;
    Pn_1 := Pn;
    Pn := 2.0 * x * Pn_1 - Pn_2;
  end;
  Chebyshev := Pn;
end;
end;
end.

```

Реализация функции вычисления полинома Чебышева здесь достаточно проста и не нуждается в комментариях.

В модуле `series` (листинг 10.6 содержит интерфейсную секцию этого модуля) реализованы некоторые математические операции со степенными рядами. Напомню, что *степенным рядом* называется бесконечная сумма вида

$$S(x) = \sum_{k=0}^{\infty} a_k x^k,$$

где  $a_k$  — числовые коэффициенты. Разумеется, компьютер может оперировать лишь с конечными суммами («обрезанными» рядами), а это полиномы, поэтому данный материал и помещен в разделе, посвященном полиномам. Прежде всего опишем те операции с рядами, которые реализованы в модуле `series`. Пусть есть два ряда

$$F(x) = \sum_k f_k x^k, \quad G(x) = \sum_l g_l x^l.$$

*Композицией* рядов называется функция:

$$H(x) = F[G(x)].$$

*Обращением* ряда  $F(x)$  будем называть решение уравнения

$$x = F(y)$$

относительно  $y$ .  $n$ -й степенью ряда  $F(x)$  назовем новый ряд

$$G(x) = F(x)^n.$$

И наконец, еще одна операция — вычисление *частного* двух рядов:

$$H(x) = F(x)/G(x).$$

В модуле `series` предполагается, что максимальная степень полинома 20, это значение содержит нетипизированная константа `max_deg`. При необходимости это значение можно изменить. Другая константа — `eta` — определяет значение або-

лотной величины коэффициента, меньше которой этот коэффициент считается нулевым. Ряд (точнее, полином, соответствующий «обрезанному» ряду) хранится в виде вещественного массива значений коэффициентов для степеней от нулевой до максимальной.

#### Листинг 10.6. Интерфейсная секция модуля series

```
unit series;

interface

const
  max_deg = 20;
  eta = 1.0e-20;
type
  power_series = array[0..max_deg] of real;

var
  Zero, One: power_series;
  k: word;
procedure quotient(const F, G : power_series; var H : power_series);
procedure composition(const F, G : power_series; var H : power_series);
procedure invert(const F : power_series; var H : power_series);
procedure nth_power(const F : power_series; n : word; var G : power_series);
```

Рассмотрим реализацию ранее перечисленных математических действий с рядами. Вычисление частного двух «обрезанных» рядов основано на следующей идее. Если  $H = F/G$ , то  $F = GH$ . Если разложить это соотношение и собрать коэффициенты при одинаковых степенях  $x$ , то получим итерационные формулы для коэффициентов частного  $H$ . Этот алгоритм реализован в процедуре `quotient` (листинг 10.7).

Процедура `composition` предназначена для вычисления композиции рядов. Алгоритм, использованный в этой процедуре, основан на следующих соотношениях:

$$F(y) = \sum_{k=1}^{\infty} a_k y^k,$$

$$H(x) = F[G(x)] = \sum_{k=1}^{\infty} a_k G(x)^k.$$

#### Листинг 10.7. Процедуры quotient и composition модуля series

```
implementation

procedure quotient(const F, G : power_series; var H : power_series);
var
  r, k : word;
  sum : real;
begin
  if (F[0] <> 1.0) or (G[0] <> 1.0) then
```

```

halt;
H := One;
for r := 1 to max_deg do
begin
  sum := F[r];
  for k := 0 to r - 1 do
    sum := sum - H[k] * G[r - k];
  H[r] := sum;
end;
end;
procedure composition(const F, G : power_series; var H : power_series);
var
  j, k : word;
  P : power_series;
begin
  if (F[0] <> 0.0) or (G[0] <> 0.0) or
    (F[1] <> 1.0) or (G[1] <> 1.0) then
    halt;
  for k := 1 to max_deg
    do H[k] := F[1] * G[k];
  for k := 2 to max_deg do
    if F[k] <> 0.0 then
      begin
        nth_power(G, k, P);
        for j := k to max_deg do
          H[j] := H[j] + F[k] * P[j];
      end;
end;

```

В процедуре `invert` используется алгоритм Феттиса. Мы ищем степенной ряд  $H(x)$ , обратный по отношению к ряду  $F(x)$ , так что

$$y = H(x) = \sum_{l=2}^{\infty} h_l x^l,$$

причем

$$x = F(y) = y + \sum_{k=2}^{\infty} f_k y^k.$$

Ряд  $H(x)$  строится в виде последовательности полиномов возрастающей степени, начиная с  $H_1(x) = x$ . Необходимо найти полином  $H_m(x)$  степени  $m$ , такой что

$$H_m(x) \equiv y \pmod{x^{m+1}}, \quad (*)$$

что выполняется для  $m = 1$ .

Если найден ряд  $H_{m-1}(x)$ , то для  $k \geq 2$  выполняются следующие соотношения:

$$H_{m-1}(x) = y + x^m L(x),$$

$$H_{m-1}(x)^k = y^k + k y^{k-1} x^m L(x) + \dots \equiv y^k \pmod{x^{m+1}},$$

поскольку  $y \equiv 0 \pmod{x}$ . Здесь  $L(x)$  — такая функция  $x$ , что соблюдается условие  $(*)$ . Положим

$$H_m(x) = x - \sum_{k=2}^m a_k H_{m-1}(x)^k.$$

Тогда

$$H_m(x) \equiv x - \sum_{k=2}^m a_k y^k \equiv y \pmod{x^{m+1}},$$

что и требовалось.

В процедуре `invert` (листинг 10.8) для вычисления полинома по полиному на единицу меньшей степени используется схема Горнера:

$$H = x - (((a_m y + a_{m-1})y + a_{m-2})y + \dots + a_2)y.$$

Пусть  $P$  представляет собой полином, вычисленный вплоть до некоторой закрывающей круглой скобки. Тогда на следующем шаге вначале необходимо вычислить  $P := P - a$ , а затем  $P := P * H \pmod{x^{(m+1)}}$ . Подпрограмма `truncated_product` вычисляет произведение  $P * H$ . Внешний цикл `for` выполняется в обратном порядке, поэтому на следующем шаге заново вычисленный коэффициент  $P$  не используется.

#### Листинг 10.8. Процедура `invert` модуля `series`

```

procedure invert(const F : power_series; var H : power_series);
var
  P : power_series;
  k, m : word;
procedure truncated_product(var P : power_series; m : word);
var
  j, k : word;
  sum : real;
begin
  begin
    for j := m downto 1 do
      begin
        sum := 0;
        for k := 1 to j do
          sum := sum + P[j - k] * H[k];
        P[j] := sum;
      end;
    P[0] := 0.0;
  end;{truncated_product}

begin{invert}
  if (F[0] <> 0.0) or (F[1] <> 1.0) then
    halt;
  H := Zero;
  H[1] := 1.0;
  for m := 2 to max_deg do
    begin
      P := Zero;

```

```

for k := m downto 2 do
begin
  P[0] := -F[k];
  truncated_product(P, m);
end;
truncated_product(P, m);
P[1] := P[1] + 1.0;
H := P;
end;
end;{invert}

```

Процедура вычисления  $n$ -й степени ряда без постоянного члена — `nth_power` (листинг 10.9) — основана на следующих соотношениях:

$$\begin{aligned} G &= F^n, \\ G' &= n F^{n-1} F', \\ FG' &= n F' G. \end{aligned}$$

Последнее уравнение после разложения дает итерационные соотношения для коэффициентов искомого ряда.

#### Листинг 10.9. Процедура `nth_power` и секция инициализации модуля `series`

```

procedure nth_power(const F : power_series; n : word; var G : power_series);
var
  deg_F, deg_G, j, k, r : integer;
  p, sum, t : real;
begin
  if F[0] <> 0.0 then
    exit;
  G := Zero;
  deg_F := 0;
  repeat
    inc(deg_F)
  until (abs(F[deg_F]) >= eta) or (deg_F = max_deg);
  deg_G := deg_F * n;
  if (F[deg_F] = 0.0) or (deg_G > max_deg) then
    exit;
  p := 1.0;
  t := F[deg_F];
  for j := 1 to n do
    p := p * t;
  G[deg_G] := p;
  for r := deg_G + 1 to max_deg do
  begin
    sum := 0.0;
    k := r;
    for j := deg_F + 1 to r + deg_F - deg_G do
    begin
      dec(k);
      sum := sum + (k - n * j) * F[j] * G[k];
    end;
  end;
end;

```

```

G[r] := - sum / ( (r - n * deg_F) * F[deg_F] );
end;
end;

begin
  for k := 0 to max_deg do
    Zero[k] := 0.0;
  One := Zero;
  One[0] := 1.0;
end.

```

В секции инициализации модуля `series` формируются массивы, отвечающие «нулевому» ряду (`Zero`), все коэффициенты которого — нули, и единичному (`One`), содержащему единицу для младшего члена ряда и нулевые коэффициенты для всех прочих.

### Упражнение 10.3

Завершите реализацию модуля `orthpoly`, добавив в него функции вычисления полиномов Лежандра, Эрмита и Лагерра (`Legendre`, `Hermit`, `Laguerre`).

### Упражнение 10.4

Напишите программу для проверки работы со всеми ортогональными полиномами из модуля `orthpoly`.

### Упражнение 10.5

Напишите программу для проверки всех процедур из модуля `series`.

### Упражнение 10.6

Реализации различных процедур в модуле `series` не так эффективны, как могли быть. Рассмотрите еще раз внутренние циклы, где небольшие усовершенствования могут значительно ускорить выполнение процедуры, и постарайтесь их оптимизировать. Имейте в виду, что одинаковые вычисления не должны выполняться дважды, процедуры `inc` и `dec` работают быстрее, чем операции `+` и `-`, и т. д. Может потребоваться введение нескольких локальных переменных. Наиболее вероятные кандидаты на оптимизацию: процедура `quotient` (содержит во внутреннем цикле вычитание), внутренний цикл в процедуре `composition` (содержит многократное умножение на `F[k]` с фиксированным `k`), внутренний цикл процедуры `truncated_product` (она вложена в процедуру `invert` и содержит вычитание), целочисленное выражение, вычисляемое во внутреннем цикле процедуры `nth_power` (содержит умножение и вычитание).

# Линейная алгебра

Линейная алгебра посвящена исследованию свойств матриц, а также решению систем линейных уравнений

$$\mathbf{Bx} = \mathbf{c},$$

где **B** — матрица коэффициентов системы, **c** — вектор правой части, а **x** — вектор неизвестных, компоненты которого следует вычислить.

Первая проблема, которой мы займемся, — решение линейных алгебраических систем методом гауссова исключения. Это традиционная тема вычислительной математики. По этому вопросу имеется обширная литература и множество программ. Добавим к ним еще несколько! Модуль `lin_alg` содержит набор процедур для решения систем линейных уравнений. Входными данными является матрица  $A = [B \ C]$ , состоящая из двух блоков, причем левый блок **B** — это квадратная матрица системы, а правый блок **C** — несколько столбцов, служащих правыми частями системы уравнений. Таким образом, можно запрограммировать решение системы сразу с несколькими правыми частями. Будем считать, что блок **C** может содержать любое число столбцов от 1 до числа столбцов в блоке **B**.

Метод гауссова исключения основан на том математическом факте, что система уравнений, в которой матрица коэффициентов получена поэлементным сложением строк исходной матрицы (с учетом компонентов вектора правой части), возможно, умноженных на некоторый общий множитель, имеет то же самое решение. Используя этот прием, можно обратить в нуль все элементы матрицы, лежащие ниже главной диагонали (эта последовательность действий и называется *гауссовым исключением*, или *приведением к треугольному виду*). Система уравнений с треугольной матрицей легко решается *обратной подстановкой*, когда вначале решается последнее уравнение в системе с треугольной матрицей, затем предпоследнее и т. д. вплоть до первого. При решении систем линейных уравнений могут возникать сложности, связанные с наличием нулевых элементов на главной диагонали матрицы. В этом случае невозможно подобрать множители для гауссова исключения и применяется метод Гаусса с *выбором ведущего элемента*. В качестве ведущего выбирается либо максимальный элемент в столбце (строке), либо максимальный элемент во всей матрице. Затем путем перестановок строк (столбцов) этот элемент переводится на главную диагональ матрицы. Такая стратегия гарантирует от неприятностей, связанных с наличием на главной диагонали нулевого элемента, и делает метод устойчивым. В модуле `lin_alg` используется метод с *частичным* выбором ведущего элемента. Если убрать пробел после первой фигурной скобки, комментарий превращается в директиву условной компиляции, которая включает в компилируемый текст новые фрагменты, соответствующие использованию *полного* выбора ведущего элемента (по всей матрице). Листинг 10.10 содержит интерфейсную секцию модуля `lin_alg`.

#### Листинг 10.10. Интерфейсная секция модуля `lin_alg`

```
unit lin_alg;
{ $define complete_pivot }

interface

const
  max_no_rows = 20;
```

```

type
  matrix = array[1..max_no_rows, 1..2 * max_no_rows] of real;
  str10 = string[10];

var
  no_rows, no_cols : word;
  singular : boolean;

procedure get_system_matrix(var A : matrix);
procedure get_rhs(var A : matrix);
procedure print_matrix(const A : matrix; name: str10;
  screen_row, first_col, last_col : word);
procedure triangularize(var A : matrix);
procedure back_subtract(var A : matrix);

```

Вначале идут процедуры ввода значений элементов матрицы, векторов правой части и проверки правильности этого ввода с возможностью исправления неправильно введенных элементов. Тексты процедур здесь не приводятся, читателю предоставляется возможность запрограммировать их самостоятельно.

Далее идут процедуры, реализующие последовательность таких элементарных преобразований строк, которые приводят матрицу системы **В** к верхнему треугольному виду. Метод выбора ведущего элемента реализован следующим образом. После обработки первых  $p$  строк определяется наибольший (по величине) элемент столбца  $p+1$ , находящийся в строках от  $(p+1)$ -й до последней. Затем производится перестановка строки, в которой находится найденный элемент, с  $(p+1)$ -й строкой. После этого умноженная на некоторый коэффициент (новая)  $(p+1)$ -я строка вычитается из всех нижележащих строк, чтобы обнулить их элементы. Эти действия выполняет процедура `triangularize`. Секция реализации модуля `lin_alg` приведена в листинге 10.11.

Решение системы находится процедурой `back_subtract`, которая дополнительными элементарными преобразованиями строк выполняет приведение левого блока к единичной матрице. Она реализует обратную подстановку метода Гаусса.

#### Листинг 10.11. Секция реализации модуля `lin_alg`

```

implementation

uses crt;

const
  eps = 1.0e-10;

var
  change : boolean;
  no_vectors : word;

{$ifdef complete_pivot}
var
  s, t : word;

```

```
column_rec : array[1..max_no_rows] of word;
{$endif}

procedure correct_input(var A : matrix; first_col, last_col : word);
...

procedure get_system_matrix(var A : matrix);
...

procedure get_rhs(var A : matrix);
...

procedure print_matrix(const A : matrix; name : str10;
screen_row, first_col, last_col : word);
...

procedure subtract_rows(var A : matrix; p, q : word; x : real);
var
    j : word;
begin
    for j := p + 1 to no_cols do
        A[q, j] := A[q, j] - x * A[p, j];
end;

{$ifdef complete_pivot}
procedure change_columns(var A : matrix; p, t : word);
var
    j : word;
    x : real;
begin
    for j := 1 to no_rows do
        begin
            x := A[j, p];
            A[j, p] := A[j, t];
            A[j, t] := x;
        end;
    end;
end;
{$endif}

procedure triangularize(var A : matrix);
var
    p, q, k : word;

function partial_pivot(p : word) : word;
var
    j, q : word;
    x, y : real;
begin
    q := p;
    x := abs(A[p, p]);
    for j := p + 1 to no_rows do
        begin
```

```

y := abs(A[j, p]);
if y > x then
begin
    q := j;
    x := y
end;
end;
singular := (abs(x) < eps);
partial_pivot := q;
end;{partial_pivot}

procedure change_rows(p, q : word);
var
    j : word;
    x : real;
begin
    for j := p to no_cols do
begin
    x := A[p, j];
    A[p, j] := A[q, j];
    A[q, j] := x;
end
end;{change_rows}

{$ifdef complete_pivot}
procedure find_complete_pivot(p : word; var s, t : word);
var
    j, k : word;
    x, y : real;
begin
    s := p;
    t := p;
    x := abs(A[p, p]);
    for j := p to no_rows do
        for k := p to no_rows do
begin
    y := abs(A[j, k]);
    if y > x then
begin
    x := y;
    s := j;
    t := k;
end;
end;
singular := (abs(x) < eps);
end;{find_complete_pivot}
{$endif}

begin{triangularize}
p := 1;
repeat
{$ifdef complete_pivot}

```

```

find_complete_pivot(p, s, t);
if not singular then
begin
  if s > p then
    change_rows(p, s);
  if t > p then
  begin
    change_columns(A, p, t);
    column_rec[p] := t;
  end
  else
    column_rec[p] := p;
{$else}
  k := partial_pivot(p);
  if not singular then
  begin
    if k > p then
      change_rows(p, k);
  {$endif}
  for q := p + 1 to no_rows do
    subtract_rows(A, p, q, A[q, p] / A[p, p]);
  end;
  inc(p);
until (p > no_rows) or singular;
end;{triangularize}

procedure back_subtract(var A : matrix);
var
  p, q : word;

procedure normalize_row(p : word);
var
  k : word;
  x : real;
begin
  x := A[p, p];
  A[p, p] := 1.0;
  for k := p + 1 to no_cols do
    A[p, k] := A[p, k] / x;
end;{normalize_row}

{$ifdef complete_pivot}
procedure restore_col_order;
var
  j, k : word;
  x : real;
begin
  for j := no_rows - 1 downto 1 do
  begin
    p := column_rec[j];
    if p > j then
    begin

```

```

        for k := no_rows + 1 to no_cols do
        begin
            x := A[j, k];
            A[j, k] := A[p, k];
            A[p, k] := x;
        end;
        end;
    end;
end;{restore_col_order}
{$endif}

begin{back_subtract}
    for p := 1 to no_rows do
        normalize_row(p);
    for p := no_rows downto 2 do
        for q := p - 1 downto 1 do
            subtract_rows(A, p, q, A[q, p]);
{$ifdef complete_pivot}
    restore_col_order;
{$endif}
end;{back_subtract}
end.

```

### Упражнение 10.7

Напишите программу для проверки процедур модуля `lin_alg`.

### Упражнение 10.8

*Масштабирование* позволяет увеличить точность решения системы линейных уравнений. Один из вариантов масштабирования заключается в том, что перед решением системы каждая ее строка поэлементно делится на максимальный элемент этой строки. Дополните модуль `lin_alg` возможностью масштабирования системы по строкам.

## Симметричные матрицы и проблема собственных значений

Следующая тема — вычисление собственных значений и собственных векторов симметричных матриц. Пусть  $A$  — вещественная симметричная матрица размера  $n \times n$ . Для симметричной матрицы  $a_{ij} = a_{ji}$ . Для собственных значений  $\mu_i$  и собственных векторов  $v_i$  матрицы  $A$  выполняется следующее условие:

$$A v_i = \mu_i v_i$$

Согласно теории, все  $n$  собственных значений симметричной матрицы  $A$  вещественны, а соответствующие им собственные векторы образуют ортонормированный базис.

**Листинг 10.12.** Интерфейсная секция модуля symm\_mat

```

unit symm_mat;

interface

const
  max_dim = 50;

type
  symmetric_matrix = array[1..max_dim, 1..max_dim] of real;
  vector = array[0..max_dim] of real;
  str4 = string[4];

var
  dim : word;

procedure get_matrix(var A : symmetric_matrix);
procedure print_matrix(var A : symmetric_matrix; name : str4);
procedure iterate(const A : symmetric_matrix; var v : vector;
  var r : real; var fail : boolean);
procedure QR_algorithm(var A : symmetric_matrix;
  var char_root : vector; var W : symmetric_matrix);

```

Процедуры `get_matrix` и `print_matrix` предназначены для ввода и вывода значений элементов матрицы. Здесь вновь их программирование предоставляется читателю в качестве упражнения. В процедуру `get_matrix` следует включить контроль правильности ввода и возможность исправления неправильно введенных значений.

Алгоритм нахождения наибольшего по величине собственного значения реализован в процедуре `iterate` (листинг 10.13). Эта процедура выполняет итерации для вектор-функции

$$\phi(v) = \pm \frac{Av}{|Av|}.$$

Итерации начинаются с единичного вектора  $e_1$  и дают последовательность векторов  $e_{n+1} = \phi(e_n)$ . Если не учитывать маловероятный случай неудачного выбора начального вектора, то последовательность итераций будет сходиться к собственному вектору, соответствующему наибольшему собственному числу. Данный метод известен под названием *степенного метода*.

**Листинг 10.13.** Процедура iterate модуля symm\_mat

```
implementation
```

```
uses crt;
```

```
const
  tol = 1.0e-30;
```

```

procedure get_matrix(var A : symmetric_matrix);
...
procedure print_matrix(var A : symmetric_matrix; name : str4);
...
procedure iterate(const A : symmetric_matrix; var v : vector;
                 var r : real; var fail : boolean);
var
  x : real;
  count, j, k : word;
  w : vector;
  test : boolean;

function norm_squared(const v : vector) : real;
var
  x : real;
  j : integer;
begin
  x := 0.0;
  for j := 1 to dim do
    x := x + sqr(v[j]);
  norm_squared := x;
end;{norm_squared}

function distance_squared(const v, w : vector) : real;
var
  x : real;
  j : integer;
begin
  x := 0.0;
  for j := 1 to dim do
    x := x + sqr(v[j] - w[j]);
  distance_squared := x;
end;{distance_squared}

begin{iterate}
  count := 0;
  fail := false;
  test := false;
  repeat
    inc(count);
    w := v;
    for j := 1 to dim do
    begin
      x := 0.0;
      for k := 1 to dim do
        x := x + A[k, j] * w[k];
      v[j] := x;
    end;
    x := norm_squared(v);
    if x < tol then

```

```

    fail := true
else
begin
  r := 0.0;
  for j := 1 to dim do
    r := r + v[j] * w[j];
  x := 1.0 / sqrt(x);
  if r < 0.0 then
    x := -x;
  for j := 1 to dim do
    v[j] := x * v[j];
  test := (distance_squared(v, w) < tol);
end;
until fail or test or (count = 200);
fail := not test;
end;{iterate}

```

В процедуре `QR_algorithm` (листинг 10.14) запрограммирован алгоритм отыскания всех собственных значений и векторов произвольной вещественной несингулярной симметричной матрицы. Приведем краткое описание этого итерационного алгоритма, выполнение которого начинается с присваивания  $A_0 = A$ . Для  $n$ -го шага итерации существует единственное представление матрицы  $A_n = U_n T_n$ , где матрица  $U_n$  ортогональна, а  $T_n$  является верхней треугольной с положительными диагональными элементами. Положим  $A_{n+1} = T_n U_n$ . Тогда последовательность  $\{A_n\}$  сходится к диагональной матрице, диагональными элементами которой будут собственные значения матрицы  $A$ . Сходимость гарантирована, если все собственные значения простые (то есть имеющие единичную кратность) и имеют различные абсолютные значения. Из теории известно, что матрицы  $A_n$  и  $A_{n+1}$ , связанные соотношениями

$$A_{n+1} = U_n^{-1} A_n U_n,$$

имеют одинаковые собственные значения.

Представление матрицы в виде произведения сомножителей называется *факторизацией*. Факторизация матрицы — в общем случае медленная процедура. Она выполняется гораздо быстрее, если мы имеем дело с *трехдиагональной* матрицей, все элементы которой равны нулю за исключением главной диагонали, наддиагонали и поддиагонали. Можно показать, что если матрица  $A = UT$  с самого начала трехдиагональная, то матрица  $TU$  тоже будет трехдиагональной, а следовательно, и все матрицы из последовательности  $\{A_n\}$  тоже трехдиагональны. Реализация *QR*-алгоритма начинается с предварительного шага (преобразования Хаусхолдера), на котором матрица  $A$  заменяется эквивалентной трехдиагональной матрицей. Это производится процедурой `tridiagonalize`. *UT*-факторизация матрицы  $A$  затем выполняется последовательностью плоских вращений:

Возможна дальнейшая модификация алгоритма с целью ускорения его сходимости. Если число  $\lambda$  принимает значение в окрестности собственного значения матрицы  $A_n$ , то факторизация и итерирование производятся следующим образом (см. процедуры `translate` и `factor`):

$$\mathbf{A}_n - \lambda \mathbf{I} = \mathbf{U}_n \mathbf{T}_n,$$

$$\mathbf{A}_{n+1} = \lambda \mathbf{I} + \mathbf{T}_n \mathbf{U}_n.$$

Назначение процедуры `factor` заключается в факторизации диагонального блока от  $m$ -й до  $n$ -й строки. Затем процедура `multiply` вычисляет значения элементов соответствующего блока. Процедура `report_result` должна выводить результаты расчета. Запрограммируйте ее самостоятельно.

**Листинг 10.14.** Процедура QR\_algorithm модуля symm\_mat

```

procedure QR_algorithm(var A : symmetric_matrix;
                      var char_root : vector; var W : symmetric_matrix);
var
  diagonal, sine, cosine,
  super_diag, super_diag2 : vector;
  norm, eps : real;
  m, n : word;
  A0 : symmetric_matrix;
  c, s, x, x1, y, z, u, lambda, mu : real;

procedure initialize_QR;
var
  j, k : word;

procedure tri_diagonalize(var A, W : symmetric_matrix;
                          var diagonal, super_diag : vector);
var
  c, t, x, y : real;
  v : vector;
  i, j, k : word;

procedure Householder_step(i : word);
var
  p : vector;
  j, k : word;
  z : real;
begin
  for j := i + 1 to dim do
  begin
    t := 0.0;
    for k := i + 1 to dim do
      t := t + A[j, k] * v[k];
    p[j] := 2.0 * t;
  end;
  z := 0.0;
  for j := i + 1 to dim do
    z := z + p[j] * v[j];
  z := 2.0 * z;
  for j := i + 1 to dim do
    for k := j to dim do
      A[j, k] := A[j, k] - v[j] * p[k] - (p[j] - z * v[j]) * v[k];
  end:{Householder_step}
end;

```

```
procedure update_W(i : word);
var
  p : vector;
  j, k : word;
begin
  for j := 1 to dim do
  begin
    t := 0.0;
    for k := i + 1 to dim do
      t := t + W[j, k] * v[k];
    p[j] := 2.0 * t;
  end;
  for j := 1 to dim do
    for k := i + 1 to dim do
      W[j, k] := W[j, k] - p[j] * v[k];
end;{update_W}

begin{tri_diagonalize}
super_diag[dim] := 0.0;
for i := 1 to dim - 2 do
begin
  diagonal[i] := A[i, i];
  t := 0.0;
  for j := i + 1 to dim do
    t := t + sqr(A[i, j]);
  c := sqrt(t);
  x := A[i, i + 1];
  if x > 0.0 then
    c := -c;
  super_diag[i] := c;
  if c <> 0.0 then
  begin
    y := sqrt(0.5 * (1.0 - x / c));
    v[i + 1] := y;
    y := -1.0 / (2.0 * c * y);
    for j := i + 2 to dim do
      v[j] := y * A[i, j];
    Householder_step(i);
    update_W(i);
  end;
  end;
  k := dim - 1;
  diagonal[k] := A[k, k];
  diagonal[dim] := A[dim, dim];
  super_diag[k] := A[k, dim];
  super_diag[dim] := 0.0;
end;{tri_diagonalize}

begin{initialize_QR}
for j := 1 to dim do
begin
  for k := 1 to dim do
```

```

W[j, k] := 0.0;
W[j, j] := 1.0;
end;
tri_diagonalize(A, W, diagonal, super_diag);
norm := 0.0;
y := 0.0;
for j := 1 to dim do
begin
  x := y + abs(diagonal[j]);
  y := abs(super_diag[j]);
  x := x + y;
  if x > norm then
    norm := x;
end;
eps := norm * tol;
for j := 0 to dim do
begin
  sine[j] := 0.0;
  cosine[j] := 0.0;
end;
end;{initialize_QR}

procedure translate(a, b, c : real; var lambda, mu : real);
var
  trace, det, lambda0, b2, srt : real;
begin
  b2 := sqr(b);
  det := a * c - b2;
  trace := a + c;
  srt := sqrt(sqr(trace) - 4.0 * det);
  if trace >= 0.0 then
    lambda := trace + srt
  else
    lambda := trace - srt;
  lambda := 0.5 * lambda;
  lambda0 := det / lambda;
  if abs(lambda0 - mu) < 0.5 * abs(lambda0) then
  begin
    lambda := lambda0;
    mu := lambda0
  end
  else
    if abs(lambda - mu) < 0.5 * abs(lambda) then
      mu := lambda
    else
      begin
        mu := lambda0;
        lambda := 0.0;
      end;
end;{translate}

procedure factor(m, n : word);

```

```

var
  j : word;
begin
  diagonal[m] := diagonal[m] - lambda;
  u := super_diag[m];
  for j := m to n - 1 do
  begin
    x := diagonal[j];
    x1 := diagonal[j + 1] - lambda;
    y := super_diag[j];
    z := sqrt(sqr(x) + sqr(u));
    c := x / z;
    s := u / z;
    cosine[j] := c;
    sine[j] := s;
    diagonal[j] := z;
    diagonal[j + 1] := -s * y + c * x1;
    super_diag[j] := c * y + s * x1;
    u := super_diag[j + 1];
    super_diag[j + 1] := c * u;
    super_diag2[j] := s * u;
  end;
end:{factor}

procedure multiply(m; n : word);
var
  j, k : word;
begin
  for k := m to n - 1 do
  begin
    c := cosine[k];
    s := sine[k];
    x := diagonal[k];
    y := super_diag[k];
    super_diag[k - 1] := super_diag[k - 1] * c +
      super_diag2[k - 1] * s;
    diagonal[k] := x * c + y * s + lambda;
    super_diag[k] := -x * s + y * c;
    diagonal[k + 1] := diagonal[k + 1] * c;
    for j := 1 to dim do
    begin
      x := W[j, k];
      y := W[j, k + 1];
      W[j, k] := x * c + y * s;
      W[j, k + 1] := -x * s + y * c;
    end;
    end;
    diagonal[n] := diagonal[n] + lambda;
  end:{multiply}

procedure report_result;

```

```

begin{QR_algorithm}
    A0 := A;
    initialize_QR;
    n := dim;
    mu := 0.0;
    while n > 0 do
    begin
        if abs(super_diag[n - 1]) < eps then
        begin
            char_root[n] := diagonal[n];
            dec(n);
        end
        else
        begin
            m := n - 1;
            repeat
                dec(m)
            until abs(super_diag[m]) < eps;
            inc(m);
            translate(diagonal[n - 1], super_diag[n - 1],
                      diagonal[n], lambda, mu);
            factor(m, n);
            super_diag2[m - 1] := 0.0;
            super_diag[m - 1] := 0.0;
            multiply(m, n);
        end;
    end;
end:{QR_algorithm}

end.

```

**Упражнение 10.9**

Напишите программу для тестирования процедуры `iterate`.

**Упражнение 10.10**

Напишите программу для тестирования процедуры `QR_algorithm`.

# Решение нелинейных уравнений

Для решения нелинейных вещественных уравнений в уроке 1 мы использовали метод деления отрезка пополам, но этот метод имеет невысокую точность. Гораздо чаще для приближенного вычисления корней функции используется *метод Ньютона*, известный также как *метод касательных*. Описание и математическое обоснование этого метода можно найти в специальной и учебной литературе по численным методам. Скажу лишь, что это итерационный метод, для которого

задается начальное приближение, а затем происходит его последовательное уточнение. Сходимость метода Ньютона к простому корню зависит от выбора начального приближения. Здесь мы рассмотрим применение метода Ньютона для решения системы нелинейных уравнений.

Пусть  $\mathbf{R}^n$  обозначает пространство  $n$ -мерных вещественных векторов-столбцов, а  $\mathbf{D}$  — область этого пространства. Пусть  $F$  — гладкая функция:

$$F: \mathbf{D} \rightarrow \mathbf{R}^n; \quad \mathbf{D} \subseteq \mathbf{R}^n.$$

Обычная производная первого порядка, используемая в ньютоновских итерациях для случая одной переменной, заменяется якобиевой матрицей — матрицей первых производных вида  $\mathbf{J} = \|\partial F_i / \partial x_j\|$ . Для заданной точки  $\mathbf{x}$  в  $\mathbf{D}$  ньютонская итерация будет иметь следующий вид:

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{J}^{-1}|_{\mathbf{x}} F(\mathbf{x}),$$

где  $\mathbf{J}^{-1}$  — матрица, обратная якобиевой. Если начальное приближение  $\mathbf{x}$  выбрано достаточно близко к корню функции  $F$ , то новое значение  $\mathbf{x}$  должно быть гораздо более точным приближением.

Программа, основанная на рассматриваемом алгоритме, должна реализовывать вычисление всех частных производных первого порядка. Наиболее прямой способ — просто запрограммировать вычисленные аналитически производные функции. Рассмотрим пример с  $n = 2$ . Вычислим экстремумы вещественнозначной функции

$$F(x, y) = \cos(x) \cos(y) - 0,1x - 0,2y + 0,15xy$$

путем анализа ее первых производных, а именно, найдем общие нули двух первых частных производных (см. программу `nonlinear_system`, исходный текст которой приведен в листинге 10.15). Эта задача сводится к решению системы из двух нелинейных уравнений:

$$\begin{cases} -\sin(x)\cos(y) - 0,1 + 0,15y = 0; \\ -\cos(x)\sin(y) - 0,2 + 0,15x = 0. \end{cases}$$

Константа `max` задает максимальное количество итераций метода Ньютона, что позволяет избежать бесконечного повторения итерационного процесса, если метод по какой-то причине не сходится. Константа `eps` определяет точность решения системы. Если якобиан принимает значение, меньшее значения константы `eta`, система считается *сингулярной*. В этом случае погрешность метода становится слишком большой и дальнейшее продолжение ньютоновских итераций теряет смысл.

Функции `F`, `G` и `dG` соответствуют функции, для которой производится вычисление экстремумов, функциям, составляющим систему нелинейных уравнений, и частным производным для этих функций. Параметрами функции `F` являются значения переменных  $x$  и  $y$ . Первый параметр функции `G` определяет номер уравнения в системе, а второй и третий задают значения переменных. Метод Ньютона реализован в процедуре `solve`. Первый параметр этой процедуры содержит переменную типа `vector_fun`, в которой содержатся функции, составляющие сис-

тему уравнений. Второй и третий параметры — координаты начальной точки ньютоновских итераций.

**Листинг 10.15.** Программа решения системы двух нелинейных уравнений

```
program nonlinear_system;
uses crt;

const
  max = 20;
  eps = 1.0e-10;
  eta = 1.0e-3;

type
  vector_fun = function(j : word; x, y : real) : real;

var
  x, y : real;
  i, n : word;
  ch : char;

function F(x, y : real) : real;
begin
  F := cos(x) * cos(y) - 0.1 * x - 0.2 * y + 0.15 * x * y;
end;

function G(j : word; x, y : real) : real; far;
begin
  if j = 1 then
    G := -sin(x) * cos(y) - 0.1 + 0.15 * y
  else
    G := -cos(x) * sin(y) - 0.2 + 0.15 * x;
end;

function dG(j, k : word; x, y : real) : real; far;
begin
  if j = 1 then
    begin
      if k = 1 then
        dG := -cos(x) * cos(y)
      else
        dG := sin(x) * sin(y) + 0.15;
    end
  else
    begin
      if k = 1 then
        dG := sin(x) * sin(y) + 0.15
      else
        dG := -cos(x) * cos(y);
    end;
end;
```

```

procedure solve(G : vector_fun; var x, y : real);
var
  j, k : integer;
  r, s, t, u, v : real;
  singular : boolean;
  ch : char;

procedure Newton_step(var x, y : real);
var
  det : real;
  j, k : word;
begin
  u := G(1, x, y);
  v := G(2, x, y);
  r := dG(1, 1, x, y);
  s := dG(1, 2, x, y);
  t := dG(2, 2, x, y);
  det := r * t - s * s;
  if abs(det) < eta then
    singular := true
  else
    begin
      singular := false;
      x := x - (t * u - s * v) / det;
      y := y - (-s * u + r * v) / det;
    end
end;{Newton_step}

begin{solve}
repeat
  Newton_step(x, y);
  dec(i);
until singular or (i = 0) or (abs(u) + abs(v) < eps );
if singular then
  writeln('Решения нет')
else
  if i = 0 then
    begin
      writeln(n, ' итераций');
      writeln('F(', x, ', ', y, ') = (' , u, ', ', v, ')')
    end
  else
    writeln('Решение: (x, y) = (' , x, ', ', y, ')');
end;{solve}

begin
  clrscr;
  x := 0.0;
  y := 0.0;
  i := max;
  solve(G, x, y);
  writeln('F(x, y) = ', F(x, y));

```

```
writeln;
write('Нажмите <Enter>'); readln;
end.
```

Скорость сходимости метода Ньютона квадратичная. Имеются и другие методы приближенного решения нелинейных уравнений, в том числе и такие, которые сходятся быстрее, но обсуждать их здесь мы не будем.

## Вычисление интегралов

Обратимся теперь к проблеме численного интегрирования. Эта тема, как правило, подробно излагается в учебниках по вычислительной математике и численным методам. Приближенное вычисление интеграла

$$I = \int_{x_0}^{x_1} F(x) dx$$

основано на его замене конечной суммой

$$I_n = \sum_{k=0}^n w_k F(x_k),$$

где  $w_k$  — числовые коэффициенты, а  $x_k$  — точки отрезка  $[x_0, x_1]$ . Приближенное равенство

$$I \approx I_n$$

называется *квадратурной формулой*, точки  $x_k$  — *узлами* квадратурной формулы, а числа  $w_k$  — *коэффициентами* квадратурной формулы. Разные методы приближенного интегрирования отличаются выбором узлов и коэффициентов. От этого выбора зависит погрешность квадратурной формулы

$$R_n = |I - I_n|.$$

В модуле `integral` реализовано несколько методов численного интегрирования как для простых (одномерных), так и для кратных (многомерных) интегралов. В функции `simpson` реализован стандартный метод Симпсона для интегрирования функции  $F(x)$  по заданному промежутку, когда число разбиений интервала выбирается заранее. Функция `double_simpson` является прямым обобщением метода Симпсона на случай интегрирования функции от двух переменных  $F(x, y)$  по прямоугольной двумерной области. `Adaptive_simpson` — это функция для вычисления простых интегралов, которая корректирует число и размер разбиений интервала, чтобы ошибка вычисления интеграла попала в заранее заданный интервал. Этот метод называется *адаптивным интегрированием*. Все современные программы интегрирования так или иначе адаптивны. В функции `romberg` запрограммирован еще один метод адаптивного интегрирования — метод Ромберга, в настоящее время, вероятно, один из наиболее популярных. Имеется также функция `gauss` — одномерная версия метода интегрирования Гаусса. Интерфейсная секция модуля `integral` приведена в листинге 10.16.

**Листинг 10.16.** Интерфейсная секция модуля integral

```

unit integral;
interface
const
  max_dim = 10;
  max_deg = 96;
type
  real_fun = function(x : real) : real;
  real_fun2 = function(x, y : real) : real;
  real_vec = array[1..max_dim + 1] of real;
  index = array[1..max_dim + 1] of word;
  vec_fun = function(j: word; x : real_vec) : real;
var
  no_evaluations, highest_level : word;
function simpson(F : real_fun; x0, x1 : real; div_no : word) : real;
function double_simpson(F : real_fun2; x0, x1, y0, y1 : real;
  x_div, y_div : word) : real;
function adaptive_simpson(F : real_fun; x0, x1, eps, eta : real) : real;
function romberg(f : real_fun; x0, x1, eps, eta : real;
  min, max : word) : real;
function gauss3(F : real_fun; x0, x1 : real; n : word): real;
procedure compute_gauss_coeffs(deg : word);
function gauss(F : real_fun; x0, x1 : real; deg : word) : real;

```

Перейдем к секции реализации. Она начинается описанием функции `simpson`. Стоит сказать несколько слов о выборе узлов и коэффициентов квадратурной формулы Симпсона. Идея *трехточечного метода Симпсона* заключается в следующем. Пусть  $x_m$  — это средняя точка интервала  $[x_0, x_1]$  и пусть  $Q(x)$  — единственный полином второй степени, который интерполирует (приближает) подынтегральную функцию  $F(x)$  по точкам  $x_0, x_m$  и  $x_1$ . Искомый интеграл аппроксимируется интегралом от функции  $Q(x)$ :

$$I \approx \int_{x_0}^{x_1} Q(x) dx = \frac{x_1 - x_0}{6} [F(x_0) + 4F(x_m) + F(x_1)].$$

Эта оценка точна, если  $F(x)$  является полиномом степени 3. В функции `simpson` интервал интегрирования делится на `div_no` равных частей, а трехточечная формула Симпсона применяется к каждому такому интервалу. Параметрами функции `simpson` (листинг 10.17) являются, по порядку, подынтегральная функция, нижняя и верхняя границы интервала интегрирования и количество подинтервалов.

**Листинг 10.17.** Функция `simpson` модуля integral

```

implementation
uses crt;

```

```

var
    zero, weight : array[1..max_deg] of real;

function simpson(F : real_fun; x0, x1 : real; div_no : word) : real;
var
    x, dx, sum : real;
    j : word;
begin
    dx := (x1 - x0) / (2.0 * div_no);
    sum := F(x0) + F(x1);
    x := x0;
    for j := 1 to 2 * div_no - 1 do
begin
    x := x + dx;
    if odd(j) then
        sum := sum + 4.0 * F(x)
    else
        sum := sum + 2.0 * F(x);
end;
    simpson := dx * sum / 3.0;
end;

```

Функция `double_simpson` (листинг 10.18) является, по существу, прямым обобщением одномерного метода Симпсона на случай вычисления двойного интеграла по прямоугольной области.

**Листинг 10.18.** Функции `double_simpson` и `simple_simpson` модуля `integral`

```

function double_simpson(F : real_fun2; x0, x1, y0, y1 : real;
    x_div, y_div : word) : real;
var
    dx, dy, x, sum : real;
    i : word;

function simple_simpson(x : real) : real;
var
    y, sum : real;
    j, v : word;
begin
    sum := F(x, y0) + F(x, y1);
    y := y0;
    for j := 1 to 2 * y_div - 1 do
begin
    y := y + dy;
    if odd(j) then
        sum := sum + 4.0 * F(x, y)
    else
        sum := sum + 2.0 * F(x, y);
end;
    simple_simpson := sum;
end;{simple_simpson}

```

```

begin{double_simpson}
  dx := (x1 - x0) / (2.0 * x_div);
  dy := (y1 - y0) / (2.0 * y_div);
  x := x0;
  sum := simple_simpson(x0) + simple_simpson(x1);
  for i := 1 to 2 * x_div - 1 do
    begin
      x := x + dx;
      if odd(i) then
        sum := sum + 4.0 * simple_simpson(x)
      else
        sum := sum + 2.0 * simple_simpson(x);
    end;
  double_simpson := dx * dy * sum / 9.0;
end;{double_simpson}

```

Недостатком рассмотренных функций интегрирования является то, что они не дают возможности явно задать точность вычисления интеграла. Точность связана с количеством точек разбиения, но ее значение в этих функциях не определяется и не используется. От этого недостатка свободны методы интегрирования с адаптивным выбором шага разбиения. Такой функцией является `adaptive_simpson`. Параметры `eps` и `eta` задают, соответственно, абсолютную и относительную погрешности. Их роль поясняется следующим неравенством:

$$|I - I_n| < \text{eps} + \text{eta} \int_{x_0}^{x_1} |F(x)| dx$$

Функция `Adaptive_simpson` (листинг 10.19) использует рекурсивную процедуру `simpson3point`, которая вычисляет значение интеграла по интервалу  $[x_0, x_0+\delta x]$ , где  $x_0$  — не обязательно исходная левая граничная точка. Если трехточечный метод Симпсона не дает достаточную точность на данном интервале, этот интервал делится на 3 равные части и метод вновь применяется к каждой из полученных частей. В результате получим 7 точек разбиения, но вычислять функцию  $F(x)$  придется только в четырех из них, поскольку значения в других трех точках уже известны (рис. 10.1).

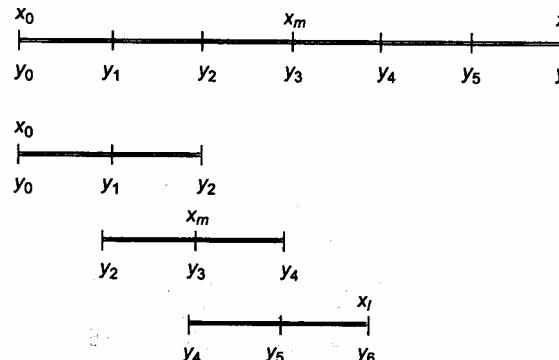


Рис. 10.1. Точки разбиения промежутка интегрирования в адаптивном методе Симпсона

При аддитивном разбиении имеется одна тонкость. При переходе к подынтервалам, составляющим одну треть от исходного, чтобы получить новые абсолютную и относительную погрешности, казалось бы, надо поделить `eps` и `eta` на 3. Оказывается, это не так, и следует в качестве делителя использовать значение  $\sqrt{3}$ .

**Листинг 10.19.** Функция `adaptive_simpson` модуля `integral`

```

function adaptive_simpson(F : real_fun; x0, xl, eps, eta : real) : real;
const
  max_level = 35;
var
  k, nest_level : word;
  integral_abs : real;

function simpson3point(x0, delta_x, estimate, integral_abs,
  eps, eta, left, middle, right : real) : real;
var
  dx3, sum, eps3, eta3, factor, left_integ,
  middle_integ, right_integ, F1, F2, F4, F5 : real;
begin
  inc(nest_level);
  dx3 := delta_x / 3.0;
  F1 := F(x0 + 0.5 * dx3);
  F2 := F(x0 + dx3);
  F4 := F(x0 + 2.0 * dx3);
  F5 := F(x0 + 2.5 * dx3);
  inc(no_evaluations, 4);
  factor := dx3 / 6.0;
  left_integ := factor * (left + 4.0 * F1 + F2);
  middle_integ := factor * (F2 + 4.0 * middle + F4);
  right_integ := factor * (F4 + 4.0 * F5 + right);
  sum := left_integ + middle_integ + right_integ;
  integral_abs := integral_abs - abs(estimate)
    + abs(left_integ) + abs(middle_integ) + abs(right_integ);
  if (nest_level > 1) and ((nest_level = max_level) or
    (abs(sum - estimate) <= eps + eta * integral_abs)) then
    simpson3point := sum
  else
    begin
      if nest_level > highest_level then
        inc(highest_level);
      eps3 := 0.577 * eps;
      eta3 := 0.577 * eta;
      left_integ := simpson3point(x0, dx3, left_integ, integral_abs,
        eps3, eta3, left, F1, F2);
      middle_integ := simpson3point(x0 + dx3, dx3,
        middle_integ, integral_abs, eps3, eta3, F2, middle, F4);
      right_integ := simpson3point(x0 + 2.0 * dx3, dx3,
        right_integ, integral_abs, eps3, eta3, F4, F5, right);
      simpson3point := left_integ + middle_integ + right_integ;
    end;
end;
```

```

dec(nest_level);
end;{simpson3point}

begin{adaptive_simpson}
nest_level := 1;
highest_level := 1;
no_evaluations := 3;
adaptive_simpson := simpson3point(x0, x1 - x0, 0.0, 0.0, eps, eta,
F(x0), F(x0 + 0.5 * (x1 - x0)), F(x1));
end;{adaptive_simpson}

```

Интегрирование следующим методом — методом Ромберга — основано на правиле трапеций, использующем кусочно-линейное приближение для интегрируемой функции. Основной факт относительно погрешности в методе трапеций следующий.

**Теорема.** Пусть  $F(x)$  — гладкая функция на интервале  $[a, b]$ , и этот интервал делится на  $n$  равных частей, каждая длиной  $h = (b - a)/n$ . Пусть  $I(h)$  обозначает соответствующее приближение метода трапеций:

$$I(h) = h[f_0 + 2f_1 + 2f_2 + \dots + 2f_{n-1} + f_n]/2,$$

где  $f_j = F(a + jh)$  — значение интегрируемой функции в точке  $a + jh$ .

Тогда

$$\int_a^b F(x)dx = I(h) - \sum_{k=1}^{\infty} a_k h^{2k},$$

где  $a_k$  — некоторая константа.

Основное здесь то, что погрешность в методе трапеций может быть выражена рядом по четным степеням шага интегрирования  $h$ . Построим таблицу значений  $T_{ik}$ :

$T_{00}$	$T_{01}$	$T_{02}$	...	$T_{0k}$
$T_{10}$	$T_{11}$	$T_{12}$	...	$T_{1,n-1}$
$T_{20}$	$T_{21}$	...	...	...
...	...	...	...	...
$T_{n0}$				

В нулевой строке  $T_{0k} = I((b - a)/2^k)$ , так что  $T_{00}, T_{01}, \dots$  являются последовательными приближениями метода трапеций для интеграла, каждое с удвоенным по сравнению с предыдущим числом интервалов. Согласно приведенной выше теореме,

$$T_{0k} = \int_a^b F(x)dx + a_1 h^2 + O(h^4),$$

где  $h = (b - a)/2^k$ .

Отсюда следует, что

$$T_{0,k+1} = \int_a^b F(x)dx + \frac{1}{4} a_1 h^2 + O(h^4),$$

поэтому положим

$$T_{1k} = \frac{4T_{0,k+1} - T_{0k}}{4 - 1} = \int_a^b F(x)dx + a_2 t^4 + O(t^6).$$

В общем случае мы строим  $j$ -ю строку таблицы Ромберга по формуле

$$T_{jk} = \frac{4^j T_{j-1,k+1} - T_{j-1,k}}{4^j - 1},$$

а оценка погрешности имеет вид

$$T_{jk} = \int_a^b F(x)dx + O(h^{2j+2}),$$

где  $h = (b - a)/2^k$ .

Для работы понадобится не целая таблица, а только последняя вычисленная строка.

Число точек выборки на каждом шаге удваивается. Обратите внимание на то, что функцию следует вычислять только в новых точках, которые являются средними точками предыдущих подинтервалов:

$$\begin{aligned} F_0 + 2F_1 + 2F_2 + \dots + 2F_{2n-1} + F_{2n} &= \\ &= (F_0 + 2F_2 + 2F_4 + \dots + 2F_{2n-2} + F_{2n}) + 2(F_1 + F_3 + \dots + F_{2n-1}). \end{aligned}$$

Таким образом, чтобы модифицировать предыдущее приближение, необходимо вычислить сумму значений функции в новых средних точках. Это делается в цикле со счетчиком  $k$ . Метод Ромберга реализован в функции `romberg` (листинг 10.20).

#### Листинг 10.20. Функция `romberg` модуля `integral`

```
function romberg(F : real_fun; x0, x1, eps, eta : real; min, max : word) :
real;
const
  abs_max = 30;
var
  p, dx, error, F_of_x0, F_of_x1, F_of_xk,
  roundoff_error, integral_abs, tolerance,
  previous_estimate, current_estimate,
  mid_sum, temp_sum, mid_sum_abs : real;
  table : array[0..abs_max] of real;
  j, n : word;
  k, r : longint;
```

```

done : boolean;
denom : array[1..abs_max] of real;
begin
  p := 1.0;
  for k := 1 to abs_max do
  begin
    p := 4.0 * p;
    denom[k] := 1.0 / (p - 1.0);
  end;
  dx := x1 - x0;
  F_of_x0 := F(x0);
  F_of_x1 := F(x1);
  current_estimate := 0.0;
  previous_estimate := 0.0;
  done := false;
  table[0] := 0.5 * dx * (F_of_x0 + F_of_x1);
  integral_abs := 0.5 * abs(dx) * (abs(F_of_x0) + abs(F_of_x1));
  n := 1;
  r := 1;
  repeat
    dx := 0.5 * dx;
    mid_sum := 0.0;
    mid_sum_abs := 0.0;
    roundoff_error := 0.0;
    for k := 1 to r do
    begin
      F_of_xk := F(x0 + (2 * k - 1) * dx);
      mid_sum_abs := mid_sum_abs + abs(F_of_xk);
      F_of_xk := F_of_xk + roundoff_error;
      temp_sum := mid_sum + F_of_xk;
      roundoff_error := (mid_sum - temp_sum) + F_of_xk;
      mid_sum := temp_sum;
      if keypressed then
        halt;
    end;
    table[n] := 0.5 * table[n - 1] + dx * mid_sum;
    integral_abs := 0.5 * integral_abs + abs(dx) * mid_sum_abs;
    for j := n - 1 downto 0 do
      table[j] := table[j + 1] + denom[n - j] * (table[j + 1] - table[j]);
    if n >= min then
    begin
      tolerance := eta * integral_abs + eps;
      error := abs(table[0] - current_estimate)
        + abs(current_estimate - previous_estimate);
      done := (error < tolerance);
    end;
    inc(n);
    done := done or (n > max);
    previous_estimate := current_estimate;
    current_estimate := table[0];
    r := r + r;
  until done;

```

```

    romberg := current_estimate;
end;

```

Теперь перейдем к *гауссовским квадратурам* — семейству правил интегрирования, основанных на неравномерном разбиении основного интервала интегрирования. Вообще, метод Гаусса с  $n$  точками точен для полиномов степени  $2n-1$ . В функции gauss3 (листинг 10.21) основной трехточечный алгоритм Гаусса применяется к каждой из  $n$  равных частей интервала. Для интервала  $[-1, 1]$  узлами квадратурной формулы являются нули полинома Лежандра третьей степени  $P_3 = (5x^3 - 3x)/2$ , а коэффициенты выбираются специальным образом.

#### Листинг 10.21. Функция gauss3 модуля integral

```

function gauss3(F : real_fun; x0, x1 : real; n : word) : real;
var
  t, sum, x, z, dx : real;
  i, k : word;
  gzero, gweight : array[1..3] of real;

procedure initialize_constants;
var
  s, t : real;
  j : word;
begin
  gzero[1] := -sqrt(0.6);
  gzero[2] := 0.0;
  gzero[3] := sqrt(0.6);
  gweight[1] := 5.0 / 9.0;
  gweight[2] := 8.0 / 9.0;
  gweight[3] := 5.0 / 9.0;
  for j := 1 to 3 do
  begin
    gzero[j] := 0.5 * (1.0 + gzero[j]);
    gweight[j] := 0.5 * gweight[j];
  end;
end:{initialize_constants}

begin{gauss3}
  initialize_constants;
  dx := (x1 - x0) / n;
  x := x0;
  sum := 0.0;
  for i := 0 to n - 1 do
  begin
    t := 0.0;
    for k := 1 to 3 do
    begin
      z := x + dx * gzero[k];
      t := t + gweight[k] * F(z);
    end;
    sum := sum + dx * t;
  end;
end;

```

```

x := x + dx;
end;
gauss3 := sum;
end;{gauss3}

```

Дадим краткий обзор некоторых свойств полиномов Лежандра. Рекурсивное определение полиномов Лежандра приводилось ранее в этом уроке. Они образуют ортогональное (но не ортонормированное) семейство на промежутке  $[-1, 1]$ , то есть

$$\int_{-1}^1 P_m(x)P_n(x)dx = 0, \quad m \neq n,$$

$$\int_{-1}^1 P_n(x)^2 dx = \frac{2}{2n+1}.$$

Величина второго интеграла определяет нормировку для этих полиномов. Имеет место также следующее представление полиномов Лежандра:

$$P_n(x) = \frac{1}{2^n} \sum_{k=0}^{n \text{ div } 2} (-1)^k \binom{n}{k} \binom{2n-2k}{n} x^{n-2k}.$$

Другая явная формула

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n.$$

Приведу несколько первых полиномов Лежандра:

$$\begin{aligned}
P_0(x) &= 1, & P_3(x) &= (5x^3 - 3x)/2, \\
P_1(x) &= x, & P_4(x) &= (35x^4 - 30x^2 + 3)/8, \\
P_2(x) &= (3x^2 - 1)/2, & P_5(x) &= (63x^5 - 70x^3 + 15x)/8.
\end{aligned}$$

Очевидно, что в общем случае полиномы Лежандра нечетной степени являются нечетными функциями, а четной степени — четными функциями.

Нам требуется найти нули полинома  $P_n(x)$ . Важно здесь то, что эти нули являются простыми и принадлежат открытому интервалу  $(-1, 1)$ . Таким образом,

$$-1 < x_1 < x_2 < \dots < x_n < 1, \quad P_n(x_j) = 0.$$

Соответствующая формула гауссовского интегрирования (с остатком) имеет следующий вид:

$$\int_{-1}^1 F(x)dx = \sum_{j=1}^n w_j F(x_j) + R_n.$$

В этой формуле

$$R_n = \frac{2^{2n+1}(n!)^4}{(2n+1)[(2n)!]^3} F^{(2n)}(\xi),$$

где  $-1 < \xi < 1$ . Веса задаются некоторыми эквивалентными формулами:

$$w_j = \frac{2}{nP_{n-1}(x_j)P'_n(x_j)} = \frac{2}{(1-x_j^2) \left[ P'_j(x_j) \right]} = \frac{2(1-x_j^2)}{(n+1)P_{n+1}(x_j)^2}.$$

Процедура `compute_gauss_coeff` (листинг 10.22) предназначена для вычисления нулей и весов квадратурной формулы Гаусса. Подпрограмма `legendre_poly` вычисляет значения  $P_n(x)$ ,  $P_{n-1}(x)$  и  $P'_n(x)$ . Последнее получается дифференцированием основной рекуррентной формулы для  $P_n(x)$ :

$$P'_n(x) = \frac{1}{n} \left[ (2n-1) (P_{n-1}(x) + xP'_{n-1}(x)) - (n-1)P'_{n-2}(x) \right].$$

Нули находятся предварительным делением интервала и применением метода секущих, после чего следуют ньютоновские итерации, в которых используются значения производных. Затем применяется первая формула для весов, в которой вновь используются значения производных. Здесь `zero` — массив нулей полинома Лежандра  $n$ -й степени, а `weight` — массив соответствующих весов. Метод вычисления нулей полинома заключается в том, чтобы поделить интервал  $[0, 1]$  на маленькие подинтервалы и проверить каждый из них на изменение знака полинома. Если изменение знака имеет место, то однократное применение метода секущих позволяет достаточно хорошо определить положение нуля. Для уточнения этого значения применяется метод Ньютона. Для обработки интервала  $[-1, 0]$  учитывается симметрия.

**Листинг 10.22.** Процедура `compute_gauss_coeff` модуля `integral`

```
procedure compute_gauss_coeff(deg : word);
const
  eps = 6.0e-20;
var
  i, index : word;
  P0k, P0k_1, D0k, P1k, P1k_1, D1k,
  x0, x1, y, z, dx, x, u : real;

procedure legendre_poly(n : word; x : real; var Pk, Pk_1, Dk : real);
var
  Pk_2, Dk_1, Dk_2 : real;
  i, j, k : word;
begin
  if n = 0 then
    begin
      Pk := 1.0;
      Dk := 0.0;
    end
  else
    begin
      Pk := 2.0*x;
      Dk := 1.0;
      for i := 1 to n-1 do
        begin
          j := i+1;
          Pk_1 := Pk;
          Dk_1 := Dk;
          Pk := (2*j+1)*x*Pk - j*Dk;
          Dk := (j+1)*(2*j+3)*x*Pk - (j+1)*Dk;
        end;
    end;
end;
```

```

else
begin
  Pk_1 := 1.0;
  Pk := x;
  Dk_1 := 0.0;
  Dk := 1.0;
  i := 3;
  j := 1;
  for k := 2 to n do
  begin
    Pk_2 := Pk_1;
    Pk_1 := Pk;
    Dk_2 := Dk_1;
    Dk_1 := Dk;
    Pk := (i * x * Pk_1 - j * Pk_2) / k;
    Dk := (i * (Pk_1 + x * Dk_1) - j * Dk_2) / k;
    inc(i, 2);
    inc(j);
  end;
end;
end:{legendre_poly}

begin{compute_gauss_coeff}
index := (deg + 1) div 2;
dx := 1.0 / (10.0 * deg);
x0 := 0.0;
x1 := x0 + dx;
if odd(deg) then
begin
  zero[index] := 0.0;
  legendre_poly(deg, x0, P0k, P0k_1, D0k);
  weight[index] := 2.0 / (P0k_1 * D0k * deg);
end;
for i := 0 to 10*deg - 1 do
begin
  x0 := x1;
  x1 := x1 + dx;
  legendre_poly(deg, x0, P0k, P0k_1, D0k);
  legendre_poly(deg, x1, P1k, P1k_1, D1k);
  if P0k * P1k <= 0.0 then
  begin
    x := x0 - P0k * dx / (P1k - P0k);
    legendre_poly(deg, x, P0k, P0k_1, D0k);
    u := P0k / D0k;
    y := x - u;
    while abs(x - y) >= eps do
    begin
      if keypressed then
      begin
        writeln(' >= eps loop: ', x:10:10, ' ', i:10:10, ' ', u:10:10,
y:10:10, ' ', abs(x-y):10);
        readln;
      end;
    end;
  end;
end;

```

```

    end;
    x := y;
    legendre_poly(deg, x, P0k, P0k_1, D0k);
    u := P0k / D0k;
    y := x - u;
  end;
  inc(index);
  legendre_poly(deg, y, P0k, P0k_1, D0k);
  zero[index] := y;
  weight[index] := 2.0 / (P0k_1 * D0k * deg);
  if index = deg then
    break;
  end;
end;
for i := 1 to deg div 2 do
begin
  zero[i] := - zero[deg - i + 1];
  weight[i] := weight[deg - i + 1];
end;
end;{compute_gauss_coeffs}

```

В функции `gauss` (листинг 10.23) запрограммирован один гауссовский шаг на заданном интервале. Конечно, серьезная прикладная программа будет делить интервал на меньшие подинтервалы и применять эту процедуру к каждому из них аддитивным способом.

#### Листинг 10.23. Функция gauss модуля integral

```

function gauss(F : real_fun; x0, x1 : real; deg : word) : real;
var
  index : word;
  a, b, sum : real;
begin
  a := 0.5 * (x1 - x0);
  b := 0.5 * (x1 + x0);
  sum := 0.0;
  for index := 1 to deg do
  begin
    sum := sum + F(a * zero[index] + b) * weight[index];
    if keypressed then
      halt;
  end;
  gauss := a * sum;
end;

```

В следующих упражнениях следует учитывать, что функции, используемые как параметры в различных функциях интегрирования модуля `integral`, должны компилироваться с директивой дальнего вызова `{$F+}`.

#### Упражнение 10.11

Напишите программу для проверки функции `simpson`.

**Упражнение 10.12**

Напишите программу для проверки функции `double_simpson`.

**Упражнение 10.13**

Напишите программу для проверки функции `adaptive_simpson`.

**Упражнение 10.14**

Напишите программу для проверки функции `romberg`.

**Упражнение 10.15**

Напишите программу для проверки функции `gauss3`.

**Упражнение 10.16**

Напишите программу для проверки функции `gauss`.

# Решение дифференциальных уравнений

Заключительная тема данного урока — решение проблемы начальных значений для систем обыкновенных дифференциальных уравнений.

Будем решать следующую задачу. На интервале  $[t_0, t_1]$  задано обыкновенное дифференциальное уравнение

$$dx/dt = F(t, x)$$

с начальным условием  $x(t_0) = x_0$ . Требуется найти  $x(t_1)$ .

Зависимая переменная  $x$  может быть скалярной функцией или, в общем случае, вектор-функцией от  $t$ . Рассмотрим численные методы, применяемые для приближенного вычисления  $x(t_1)$ , считая, что функция  $F(t, x)$  является достаточно гладкой.

## Методы Рунге—Кутта

Обширное семейство методов приближенного решения дифференциальных уравнений основано на введении сетки и замене производных их конечно-разностными аппроксимациями. Данный шаг позволяет избавиться от такого неудобного для компьютера объекта, как производная, заменив исходную задачу задачей алгебраической. Простейшим из этих методов является *метод Эйлера*:

$$x(t_1) \approx x_0 + F(t_0, x_0) dt,$$

$$dt = t_1 - t_0.$$

Геометрическое истолкование этого метода состоит в том, что он аппроксимирует график решения дифференциального уравнения касательной к графику в точке  $(t_0, x_0)$  (рис. 10.2).

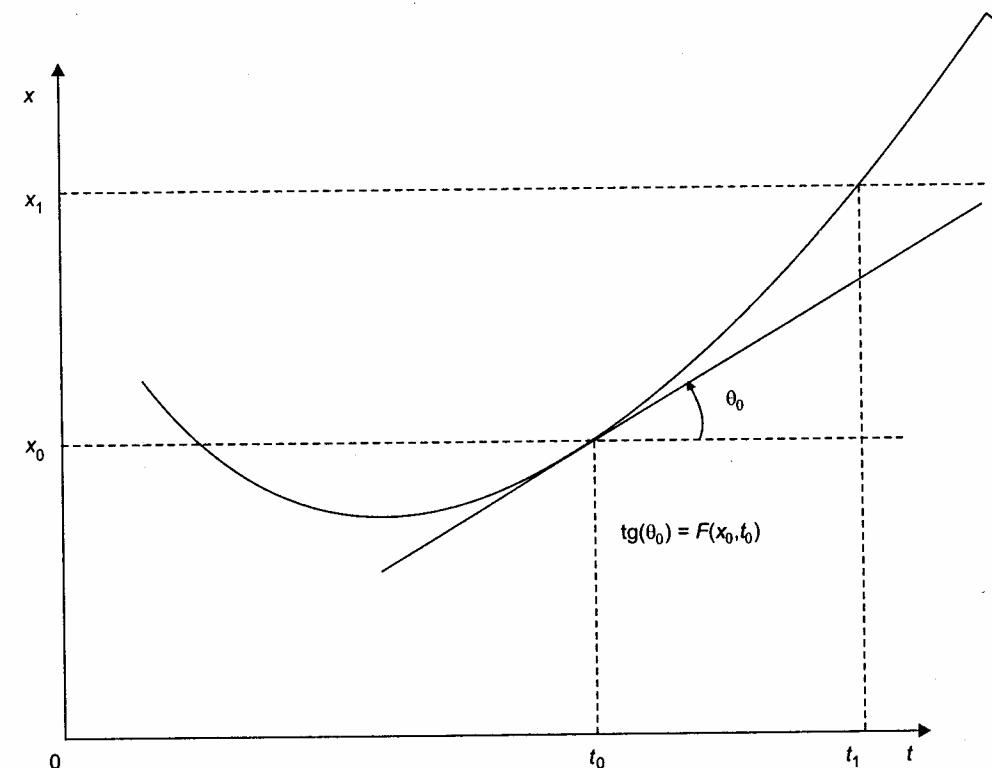


Рис. 10.2. Шаг метода Эйлера

Этот метод называется одношаговым, так как процедура получения оценки решения в некоторой точке состоит из одного шага. Погрешность простого метода Эйлера довольно велика, следовательно, он нуждается в модификации.

*Модифицированный метод Эйлера* (известный также как *метод Хьюона*) — это двухшаговый процесс (рис. 10.3).

Простой метод Эйлера в нем используется для построения первого приближения в точке  $t_1$ . В этой точке определяется тангенс угла наклона касательной к графику функции. Для получения окончательной оценки в точке  $t_1$  производится усреднение тангенсов угла наклона в обеих точках. Очевидно, в этом случае требуются две оценки функции  $F(x)$ :

$$\begin{aligned}s_1 &= F(t_0, x_0), \quad z = x_0 + s_1 dt, \\ s_2 &= F(t_1, z), \quad x_1 = x_0 + (s_1 + s_2)dt/2.\end{aligned}$$

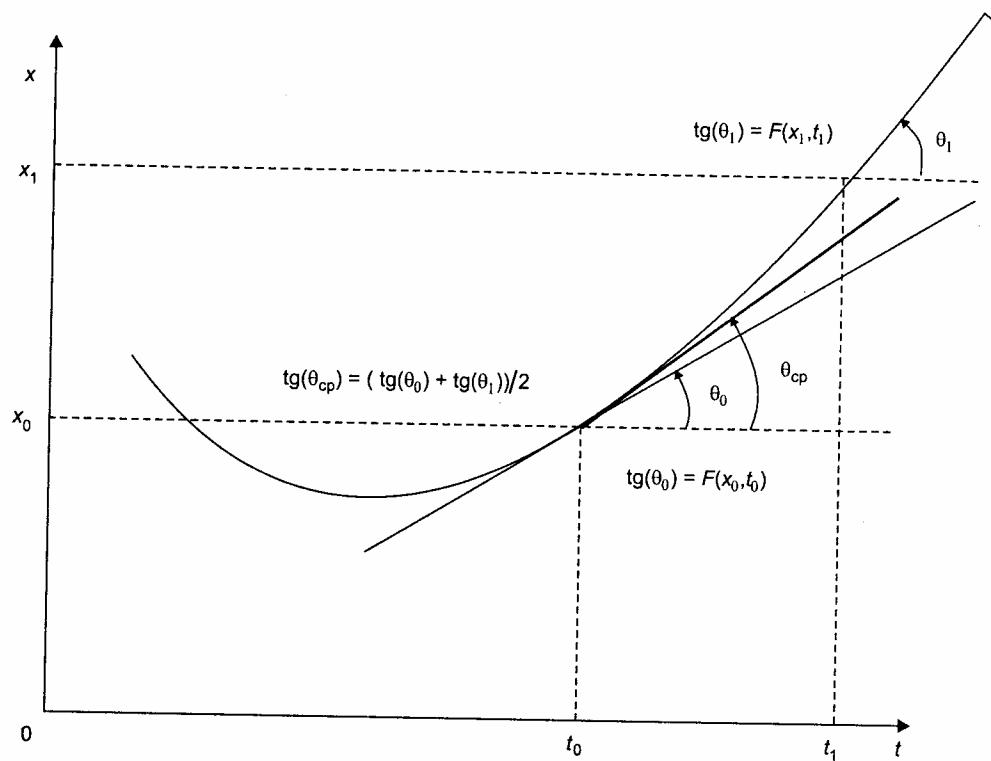


Рис. 10.3. Шаг модифицированного метода Эйлера. Жирной линией показано используемое приближение

В общем случае *метод Рунге–Кутта* является  $n$ -шаговым методом. На интервале  $[t_0, t_1]$  имеется последовательность  $n$  точек, и для этих точек вычисляются наклоны касательных к графику функции. Получив несколько значений, мы используем их взвешенное среднее для построения отрезка линии, идущего из начальной точки. Делается это для того, чтобы вычислить оценку решения в следующей точке. Затем вычисляется наклон в этой точке и т. д. Это — геометрическое истолкование методов Рунге–Кутта. Читатель может самостоятельно построить рисунок, аналогичный рисункам для метода Эйлера. Можно дать и более строгую формулировку. Пусть заданы числовые коэффициенты  $a_i, b_{ij}, i = 2, \dots, m, j = 1, \dots, m-1, \sigma_k, k = 1, \dots, m$  и последовательно вычисляются функции

$$k_1 = F(t_n, x_n),$$

$$k_2 = F(t_n + a_2 h, x_n + b_{21} h k_1),$$

$$k_3 = F(t_n + a_3 h, x_n + b_{31} h k_1 + b_{32} h k_2),$$

.....

$$k_m = F(t_n + a_m h, x_n + b_{m1} h k_1 + b_{m2} h k_2 + \dots + b_{m,m-1} h k_{m-1}).$$

Затем из формулы

$$\frac{x_{n+1} - x_n}{h} = \sum_{i=1}^m \sigma_i k_i$$

находится значение  $x_{n+1} = x(t_{n+1})$ . Числовые коэффициенты выбираются из соображений точности, методы Рунге–Кутта различаются способом выбора этих коэффициентов. Заметим, что методы Рунге–Кутта при  $m > 5$  не используются.

В расчетах обычно поступают следующим образом. Интервал, на котором необходимо найти приближенное решение дифференциального уравнения, разбивают на несколько интервалов меньшего размера (рис. 10.4) и на каждом таком интервале применяют метод Эйлера или любой другой из названных численных методов решения дифференциальных уравнений. В итоге получают, если вновь обратиться к языку геометрических построений, ломаную, которая достаточно (или недостаточно!) хорошо приближает график точного решения дифференциального уравнения (рис. 10.5).



Рис. 10.4. Разбиение интервала, на котором строится приближенное решение дифференциального уравнения

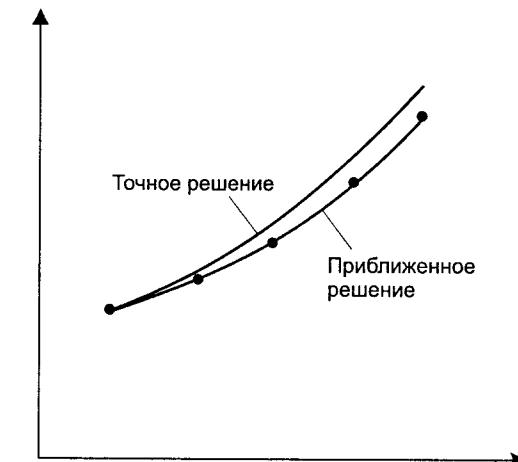


Рис. 10.5. Соответствие между приближенным и точным решениями дифференциального уравнения

Модуль `rkutta` (листинг 10.24) представляет собой небольшую коллекцию методов Рунге–Кутта. В него входят реализация модифицированного метода Эйлера, а также несколько вариантов методов Рунге–Кутта разного порядка точности (`RK4` – Runge–Kutta 4, `KM34` – Kutta–Merson, `RKN34` – Runge–Kutta–Norsett, `RKF45` – Runge–Kutta–Fehlberg).

Говорят, что метод имеет порядок  $n$ , если его погрешность составляет  $O(dt^n)$ . Каждый метод в модуле `rkutta` реализован с вычислением двух приближений.

Методы, имеющие названия с простым суффиксом, вычисляют два приближения одного и того же порядка, то есть RK4 вычисляет два приближения порядка 4. Методы с двузначными суффиксами вычисляют одну оценку для каждого порядка, то есть KM34 вычисляет приближения порядка 3 и 4. В каждом случае также вычисляется разность между двумя приближениями. Эта разность служит мерой погрешности.

В процедуре RKF45 массив abscissa содержит значения  $t$  (приведенные к единичному интервалу), для которых будут вычисляться наклоны касательных. Каждая строка массива weights суммируется с соответствующим значением из массива abscissa, что в результате дает веса, используемые для усреднения всех наклонов вплоть до текущего. С этими весами можно вычислить значение следующего наклона. Пять наклонов усреднены с весами в векторе mult1, чтобы вычислить первое приближение  $x_1$ , являющееся приближением четвертого порядка. Вычисление рациональных значений, используемых в расчете, — трудная задача, и ее здесь мы касаться не будем.

#### Листинг 10.24. Модуль rkutta

```
unit rkutta;

interface

const
  max_dim = 10;

type
  vector = array[1..max_dim] of real;
  derivative = procedure(const t : real; const x : vector;
    var w : vector);

function max(a, b : real) : real;
function sup_norm(const dim : word; const z : vector): real;
function x2y(x, y : real) : real;
procedure Euler(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);
procedure KM34(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);
procedure RKN34(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);
procedure RK4(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);
procedure RKF45(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);

implementation

function max(a, b : real) : real;
begin
  if a >= b then
    max := a
  else
    max := b;
end;
```

```
else
    max := b;
end;

function sup_norm(const dim : word; const z : vector) : real;
var
    k : word;
    w : real;
begin
    w := 0.0;
    for k := 1 to dim do
        w := max(w, abs(z[k]));
    sup_norm := w;
end;

function x2y(x, y : real) : real;
begin
    x2y := exp(y * ln(x));
end;

procedure Euler(dim : word; F : derivative; const t0, dt : real;
    const x0 : vector; var x1, y1, error : vector);
var
    slope0, slope1, z, w : vector;
    j : word;
begin
    F(t0, x0, slope0);
    for j := 1 to dim do
        z[j] := x0[j] + dt * slope0[j];
    F(t0 + dt, z, slope1);
    for j := 1 to dim do
        x1[j] := x0[j] + 0.5 * dt * (slope0[j] + slope1[j]);
    for j := 1 to dim do
        z[j] := x0[j] + 0.5 * dt * slope0[j];
    F(t0 + 0.5 * dt, z, slope1);
    for j := 1 to dim do
        w[j] := x0[j] + 0.25 * dt * (slope0[j] + slope1[j]);
    F(t0 + 0.5 * dt, w, slope0);
    for j := 1 to dim do
        z[j] := w[j] + 0.5 * dt * slope0[j];
    F(t0 + dt, z, slope1);
    for j := 1 to dim do
        begin
            y1[j] := w[j] + 0.25 * dt * (slope0[j] + slope1[j]);
            error[j] := x1[j] - y1[j];
        end;
end;

procedure KM34(dim : word; F : derivative; const t0, dt : real;
    const x0 : vector; var x1, y1, error : vector);
const
    bound = 4;
```

```

abscissa : array[1..bound] of real = (1./3., 1./3., 1./2., 1. );
weight : array[1..bound, 0..bound - 1] of real = (
  (1./3., 0., 0., 0.),
  (1./6., 1./6., 0., 0.),
  (1./8., 0., 3./8., 0.),
  (1./2., 0., -3./2., 2.));
mult1 : array[0..bound] of real =
  (1./2., 0., -3./2., 2., 0.);
mult2 : array[0..bound] of real =
  (1./6., 0., 0., 2./3., 1./6. );
error_coeff : array[0..bound] of real =
  (1./3., 0., -3./2., 4./3., -1./6.);

var
  slope, z : array[0..bound] of vector;
  s, s1, s2 : real;
  i, j, k : word;

begin{KM34}
  F(t0, x0, slope[0]);
  for i := 1 to bound do
    begin
      for j := 1 to dim do
        begin
          s := 0.0;
          for k := 0 to i - 1 do
            s := s + weight[i, k] * slope[k, j];
          z[i, j] := x0[j] + dt * s;
        end;
      F(t0 + abscissa[i] * dt, z[i], slope[i]);
    end;
  for j := 1 to dim do
    begin
      s1 := 0.0;
      s2 := 0.0;
      for k := 0 to bound do
        begin
          s1 := s1 + mult1[k] * slope[k, j];
          s2 := s2 + mult2[k] * slope[k, j];
        end;
      x1[j] := x0[j] + dt * s1;
      y1[j] := x0[j] + dt * s2;
      error[j] := x1[j] - y1[j];
    end;
end;{KM34}

procedure RKN34(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);
const
  bound = 4;
  abscissa : array[1..bound] of real =
    (3./8., 9./16., 25./32., 1. );

```

```

weight : array[1..bound, 0..bound - 1] of real = (
  (3./8.. 0.. 0.. 0.. ),
  (0.. 9./16.. 0.. 0.. ),
  (-125./672.. 325./336.. 0.. 0.. ),
  (371./891.. -200./297.. 1120./891.. 0.. 0.. );
mult1 : array[0..bound] of real =
  (37./225.. 44./117.. 0.. 448./975.. 0.. 0.. );
mult2 : array[0..bound] of real =
  (25./162.. 32./135.. 256./567.. 0.. 11./70.. 0.. 0.. );
error_coeff : array[0..bound] of real =
  (41./4050.. 244./1755.. -256./567.. 448./975.. -11./70.. 0.. 0.. );

```

```

var
  z, slope : array[0..bound] of vector;
  s, s1, s2 : real;
  i, j, k : word;

begin{RKN34}
  F(t0, x0, slope[0]);
  for i := 1 to bound do
    begin
      for j := 1 to dim do
        begin
          s := 0.0;
          for k := 0 to i - 1 do
            s := s + weight[i, k] * slope[k, j];
          z[i, j] := x0[j] + dt * s;
        end;
      F(t0 + abscissa[i] * dt, z[i], slope[i]);
    end;
  for j := 1 to dim do
    begin
      s1 := 0.0;
      s2 := 0.0;
      for k := 0 to bound do
        begin
          s1 := s1 + mult1[k] * slope[k, j];
          s2 := s2 + mult2[k] * slope[k, j];
        end;
      x1[j] := x0[j] + dt * s1;
      y1[j] := x0[j] + dt * s2;
      error[j] := x1[j] - y1[j];
    end;
end;{RKN34}

```

```

procedure RK4(dim : word; F : derivative; const t0, dt : real;
  const x0 : vector; var x1, y1, error : vector);

```

```

const
  bound = 3;
  abscissa : array[1..bound] of real = (1./2.. 1./2.. 1. );
  weight : array[1..bound, 0..bound - 1] of real = (
    (1./2.. 0.. 0.. 0.. ),
    (0.. 9./16.. 0.. 0.. ),
    (-125./672.. 325./336.. 0.. 0.. ),
    (371./891.. -200./297.. 1120./891.. 0.. 0.. );

```

```

(0.0, 1./2.. 0. ),
(0.. 0., 1. ) );
mult1 : array[0..bound] of real = (1./6., 1./3., 1./3., 1./6. );
var
slope : array[0..bound] of vector;
w : vector;
s, s1, s2 : real;
j : word;

procedure step(const t0, dt : real; const x0 : vector; var y : vector);
var
i, j, k : word;
z : array[1..bound] of vector;
begin
F(t0, x0, slope[0]);
for i := 1 to bound do
begin
for j := 1 to dim do
begin
s := 0.0;
for k := 0 to i - 1 do
s := s + weight[i, k] * slope[k, j];
z[i, j] := x0[j] + dt * s;
end;
F(t0 + abscissa[i] * dt, z[i], slope[i]);
end;
for j := 1 to dim do
begin
s1 := 0.0;
for k := 0 to bound do
s1 := s1 + mult1[k] * slope[k, j];
y[j] := x0[j] + dt * s1;
end;
end;{step}

begin{RK4}
step(t0, dt, x0, x1);
step(t0, 0.5 * dt, x0, w);
step(t0 + 0.5*dt, 0.5 * dt, w, y1);
for j := 1 to dim do
error[j] := x1[j] - y1[j];
end:{RK4}

procedure RKF45(dim : word; F : derivative; const t0, dt : real;
const x0 : vector; var x1, y1, error : vector);
const
bound = 5;
abscissa : array[1..bound] of real =
(1./4., 3./8., 12./13., 1., 1./2. );
weight : array[1..bound, 0..bound - 1] of real =
(1./4., 0., 0., 0., 0. ),
(3./32., 9./32., 0., 0., 0. ).
```

```

(1932./2197.. -7200./2197, 7296./2197.., 0., 0.),
(439./216.. -8., 3680./513.., -845./4104.., 0.),
(-8./27.., 2., -3544./2565.., 1859./4104.., -11./40. ) );
mult1 : array[0..bound] of real =
(25./216.., 0., 1408./2565.., 2197./4104.., -1./5.., 0 );
mult2 : array[0..bound] of real =
(16./135.., 0., 6656./12825.., 28561./56430.., -9./50.., 2./55. );
error_coeff : array[0..bound] of real =
(-1./360.., 0., 128./4275.., 2197./75240.., -1./50.., -2./55..);
var
slope : array[0..bound] of vector;
z : array[1..bound] of vector;
s, s1, s2 : real;
i, j, k : word;

begin{RKF45}
F(t0, x0, slope[0]);
for i := 1 to bound do
begin
for j := 1 to dim do
begin
s := 0.0;
for k := 0 to i - 1 do
s := s + weight[i, k] * slope[k, j];
z[i, j] := x0[j] + dt * s;
end;
F(t0 + abscissa[i] * dt, z[i], slope[i]);
end;
for j := 1 to dim do
begin
s1 := 0.0;
s2 := 0.0;
for k := 0 to bound do
begin
s1 := s1 + mult1[k] * slope[k, j];
s2 := s2 + mult2[k] * slope[k, j];
end;
x1[j] := x0[j] + dt * s1;
y1[j] := x0[j] + dt * s2;
error[j] := x1[j] - y1[j];
end;
end;{RKF45}
end.

```

В качестве примера применения процедур модуля rkutta рассмотрим решение следующей краевой задачи:

$$\frac{d^2x}{dt^2} = -1 - (t^2 + 1)x$$

при  $x(-1) = 0, \dots x(1) = 0$ .

Можно считать, что проблема состоит в том, чтобы выбрать такое значение производной функции  $x(t)$  в левой точке интервала, которое позволит получить известное значение  $x$  на правой границе промежутка. Сначала заменим уравнение второго порядка системой двух уравнений первого порядка:

$$\begin{cases} \frac{dx_1(t)}{dt} = x_2(t), \\ \frac{dx_2(t)}{dt} = -1 - (1 + t^2)x_1(t). \end{cases}$$

Выбор в качестве начальных значений 1 и 2 для производной дает противоположные знаки функции на правой границе промежутка (будем считать, что это — подсказка). Метод стрельбы можно объяснить, считая, что полученная выше система дифференциальных уравнений первого порядка описывает движение тела («снаряда») по некоторой траектории. Используя такую «артиллерийскую» аналогию, сформулируем задачу так. Из орудия, установленного в левой граничной точке интервала, производится обстрел правой границы интервала. При этом начальный наклон изменяется до тех пор, пока «снаряд», траектория которого описывается данным дифференциальным уравнением, не «попадет в мишень». Отклонение «снаряда» от правой граничной точки является параметром, который должен в итоге обратиться в ноль. Для построения траектории дифференциальное уравнение решается численно, а проблема подбора начальных условий движения снаряда рассматривается как проблема решения нелинейного уравнения.

В нашем случае мы будем использовать процедуру интегрирования КМ34. Полный текст программы приводится ниже (листинг 10.25). Подробный разбор исходного текста программы поручим читателю в качестве самостоятельной работы.

#### Листинг 10.25. Программа shoot

```
program shoot;
uses crt, rkutta;
const
  u26 = 26.0 * 5.5e-20;
  eps = 5.0e-15;
  eta = 5.0e-15;
var
  delta_t : real;
  x0, x1 : vector;
  fun_count, k : word;
procedure F(const t : real; const x : vector; var w : vector); far;
begin
  w[1] := x[2];
  w[2] := -1 - (1 + t*t)*x[1];
end;
begin
  rkutta(@F, 0.0, 1.0, 0.01, 1000, @x0, @x1, @delta_t);
  if abs(x1[1]) < eta then
    writeln('Success')
  else
    writeln('Failure');
  readln;
end.
```

```
w[2] := -1 - (1 + sqr(t)) * x[1];
end;

procedure solve_ode(dim : word; const t0, t1 : real; const x0 :
    vector; var x1 : vector);
const
    max_calls = 20000;
var
    dt, min_dt, t, tol, temp,
    norm_error : real;
    current_x, next_x, error, y1, der_x : vector;
    output_count : word;
    advanced, done : boolean;

procedure initialize_dt;
var
    norm_der_x : real;
begin
    fun_count := 0;
    output_count := 0;
    t := t0;
    delta_t := t1 - t0;
    current_x := x0;
    F(t, current_x, der_x);
    tol := eta * sup_norm(dim, current_x) + eps;
    dt := abs(delta_t);
    norm_der_x := sup_norm(dim, der_x);
    if tol < norm_der_x * x2y(dt, 4.0) then
begin
    dt := u26 * max(abs(t), dt);
    dt := max(dt, x2y(tol / norm_der_x, 0.25));
end;
    if delta_t < 0 then
        dt := -dt;
end:{initialize_dt}

begin{solve_ode}
    initialize_dt;
repeat
    if abs(dt) > 2.0 * abs(delta_t) then
        inc(output_count);
    if abs(delta_t) <= u26 * abs(t) then
begin
    for k := 1 to dim do
        x1[k] := current_x[k] + delta_t * der_x[k];
    exit;
end;
    done := false;
    advanced := true;
    min_dt := u26 * abs(t);
    delta_t := t1 - t;
    if abs(delta_t) < 2.0 * abs(dt) then
```

```
begin
  if 0.9 * abs(delta_t) > abs(dt) then
    dt := 0.5 * delta_t
  else
    begin
      done := true;
      dt := delta_t;
    end;
end;
repeat
  KM34(dim, F, t, dt, current_x, next_x, y1, error);
  tol := sup_norm(dim, current_x) + sup_norm(dim, next_x);
  tol := 0.5 * tol * eta + eps;
  norm_error := sup_norm(dim, error);
  if norm_error >= tol then
    begin
      advanced := false;
      done := false;
      if norm_error >= 6561.0 * tol then
        dt := 0.1 * dt
      else
        dt := 0.9 * dt / x2y(norm_error / tol, 0.25);
    end;
  until norm_error < tol;
  t := t + dt;
  current_x := next_x;
  F(t, current_x, der_x);
  inc(fun_count);
  if advanced then
    begin
      if norm_error <= (2.56e-3) * tol then
        temp := 4.0
      else
        temp := x2y(tol / norm_error, 0.25);
      temp := temp * abs(dt);
      if min_dt > temp then
        temp := min_dt;
      if dt >= 0 then
        dt := temp
      else
        dt := -temp;
    end;
  until done;
  x1 := next_x;
end:{solve_ode}

var
  dx02: real;

begin
  x0[1] := 0.0;
  x0[2] := 1.0;
```

```

clrscr;
writeln;
writeln('Абсолютная погрешность:', eps:5,
       ' Относительная погрешность:', eta:5);
writeln;
writeln('Идут вычисления:');
writeln;
dx02 := 0.1;
for k := 1 to 9 do
begin
  writeln('Номер ', k);
  repeat
    write('x0 = [', x0[1]:9:9, ', ', x0[2]:9:9, ']');
    solve_ode(2, -1.0, 1.0, x0, x1);
    writeln(' x1 = [', x1[1]:9:9, ', ', x1[2]:9:9, ']');
    x0[2] := x0[2] + dx02;
    until x1[1] >= 0.0;
    x0[2] := x0[2] - 2.0 * dx02;
    dx02 := 0.1 * dx02;
  end;
  writeln;
  write ('Нажмите <Enter>');
  readln;
end.

```

Использование в программе `shoot` константы `u26` заслуживает особого комментария. Известно, что компьютер выполняет арифметические операции над вещественными числами неточно. Количественной мерой погрешности служит ошибка округления единицы, которая определяется как такое наименьшее значение  $d$ , что компьютер различает  $1$  и  $1+d$ . Эта величина зависит от целого ряда факторов, в числе которых используемое программное обеспечение, а также используемый вещественный тип.

Программа `unit_roundoff_error` (листинг 10.26) определяет ошибку округления единицы для вещественного типа Турбо Паскаля. Это делается в два этапа. Сначала определяется порядок величины, а затем производится ее уточнение.

Вернемся к константе `u26` программы `shoot`. При численном решении дифференциального уравнения для больших значений аргумента приращение не имеет смысла делать произвольно маленьким — ограничение накладывается погрешностью вещественной арифметики. Практика показывает, что в случае конкретной программы это приращение определяется множителем `u26`, пропорциональным ошибке округления единицы.

#### Листинг 10.26. Программа определения ошибки округления единицы

```

program unit_roundoff_error;

var
  s, u: real;
  k: word;

```

```
begin
  u := 1.0;
  while 1.0 + u > 1.0 do u := 0.1 * u;
  s := u;
  for k := 1 to 3 do
  begin
    while 1.0 + u <= 1.0 do u := u + s;
    u := u - s;
    s := 0.1*s;
  end;
  writeln('Ошибка округления единицы: ', u:14);
  write('Нажмите <Enter>: ');
  readln;
end.
```

### Упражнение 10.17

Напишите программу для проверки всех процедур решения дифференциальных уравнений из модуля rkutta.

## Что нового мы узнали?

- Познакомились с программированием вычислительных задач, связанных с теорией чисел.
- Познакомились с программированием вычислений, связанных с полиномами.
- Познакомились с программированием вычислительных методов линейной алгебры.
- Познакомились с программированием методов приближенного решения нелинейных уравнений.
- Познакомились с программированием методов приближенного вычисления определенных интегралов.
- Познакомились с примерами программирования методов приближенного решения обыкновенных дифференциальных уравнений.

## **ПРИЛОЖЕНИЕ А**

### **Работа в интегрированной среде Турбо Паскаля**

В этом приложении даются основные сведения о работе в интегрированной среде Турбо Паскаля. Интегрированная среда объединяет вместе компилятор языка, редактор, который используется для набора программ и их модификации, а также отладчик. Все эти компоненты доступны из одной программы, которая запускается командой *turbo*. При этом на экране появляется окно (рис. А.1), которое состоит из нескольких частей. Эти части — строка меню в верхней части, рабочая область в центре и строка статуса внизу.

Строка меню предоставляет доступ к командам интегрированной среды. Страна меню может быть активной, и тогда имеется возможность работать с командами меню; она может быть неактивной, если, например, производится редактирование текста программы. Меню и все прочие компоненты интегрированной среды могут быть невидимыми, если идет выполнение программы или просмотр результатов ее работы, выводимых на экран. Активизировать строку меню можно, нажав клавишу F10. Можно воспользоваться для этого и мышью, указав на меню и нажав левую кнопку. Каждому пункту основного меню соответствует группа команд, с которыми связаны подпункты меню. Раскрыв любой пункт основного меню, можно видеть, что есть несколько разновидностей команд.

Если за командой меню следует знак многоточия (...), то выбор этой команды приведет к выводу диалогового окна. Если за командой следует значок >, то эта команда вызывает вложенное подменю. Все прочие команды непосредственно связаны с выполнением каких-то действий. Чтобы выбрать необходимую команду, после активизации основного меню можно воспользоваться клавишами управления курсором. Можно нажать клавишу, соответствующую выделенному символу в названии пункта меню, а можно воспользоваться, как уже было сказано, мышью. Запуск команды производится нажатием клавиши Enter. Имеется

еще один способ быстрого запуска команды. Нажмите клавишу Alt и выделенную букву в названии команды. Прервать выполняемое действие можно нажатием клавиши Esc, а прервать выполнение программы — одновременным нажатием клавиш Ctrl+Break. Некоторые команды меню могут быть недоступными (их названия выводятся серым цветом). Это происходит в том случае, когда использование команды не имеет смысла.

Многие команды интегрированной среды могут вызываться с помощью специальных клавиш или комбинаций клавиш. Перечень специальных клавиш и комбинаций клавиш для различных режимов работы в интегрированной среде дан в табл. А.1–А.6. Изучения этих таблиц достаточно для успешной работы с интегрированной средой.

```

File Edit Search Run Compile Debug Tools Options Window Help
[ * ] COUNT.PAS 1-[ F ]
program counting;
uses crt;
function counter1(m, n : word): word;
var
  c, j : word;
begin
  c := 0;
  for j := m to n do
    if (j mod 3 = 0) xor (j mod 7 = 0) then
      inc(c);
  counter1 := c;
end;
function counter2(m, n : word) : word;
begin
  counter2 := n div 7 + n div 3 - 2 * (n div 21)
  - (m - 1) div 7 - (m - 1) div 3 + 2 * ((m - 1) div 21);
end;
  1:1
Help F2 Save F3 Open Alt+F9 Compile F9 Make Alt+F10 Local menu

```

Рис. А.1. Рабочий экран интегрированной среды Турбо Паскаля

**Таблица А.1.** Назначение функциональных клавиш  
в интегрированной среде Турбо Паскаля

Клавиша	Команда меню	Функция
F1	Help	Показывает экран справки
F2	File ▶ Save	Сохранение файла, находящегося в активном окне редактора
F3	File ▶ Open	Открытие файла
F4	Run ▶ Go to Cursor	Выполнение фрагмента программы от подсвеченной строки до строки, на которой стоит курсор
F5	Window ▶ Zoom	Изменение размера активного окна
F6	Window ▶ Next	Переход к следующему окну редактирования
F7	Run ▶ Trace Into	Запуск программы в режиме отладки с заходом внутрь процедур

Клавиша	Команда меню	Функция
F8	Run ▶ Step Over	Запуск программы в режиме отладки, минуя вызовы процедур
F9	Compile ▶ Make	Компиляция программы из текущего окна
F10		Активизация строки меню

Таблица А.2. Специальные комбинации клавиш для редактирования

Клавиши	Команда меню	Функция
Ctrl+Del	Edit ▶ Clear	Удаление выбранного текста
Ctrl+Ins	Edit ▶ Copy	Копирование выбранного текста в буфер
Shift+Del	Edit ▶ Cut	Перемещение выбранного текста в буфер
Shift+Ins	Edit ▶ Paste	Запись текста из буфера в активное окно
Ctrl+L	Search ▶ Search Again	Повторяет последнюю команду Find или Replace

Таблица А.3. Специальные комбинации клавиш редактирования, не связанные с командами меню

Клавиши	Функция
Ctrl+K, затем R	Вставить в текущей позиции курсора файл с диска
Ctrl+K, затем W	Записать выделенный блок в файл
Ctrl+K, затем любая цифра	Установить маркер с номером, соответствующим использованной цифре
Ctrl+K, затем T	Выделить слово
Ctrl+Q, затем В	Перейти в начало выделенного блока
Ctrl+Q, затем К	Перейти в конец выделенного блока
Ctrl+Q, затем любая цифра	Перейти к маркеру с номером, соответствующим использованной цифре
Ctrl+Q, затем A	Найти и заменить
Ctrl+Q, затем Р	Перейти к предыдущей позиции курсора
Ctrl+Q, затем L	Восстановить строку
Ctrl+Q, затем Y	Удалить текст до конца строки
Ctrl+Q, затем W	Восстановить последнее сообщение об ошибке компиляции
Ctrl+Q, затем [	Найти парную скобку в направлении вперед
Ctrl+Q, затем ]	Найти парную скобку в направлении назад
Ctrl+Y	Удалить строку
Ctrl+N	Вставить строку
Ctrl+Home	Перейти в начало экрана
Ctrl+End	Перейти в конец экрана
Ctrl+PgUp	Перейти к началу файла
Ctrl+PgDn	Перейти к концу файла
Shift+клавиши управления курсором	Выделение блока

**Таблица А.4.** Специальные комбинации клавиш для управления окнами

Клавиши	Команда меню	Функция
Alt+цифра 1–9		Переход к окну с заданным номером
Alt+0	Window ▶ List	Показать список открытых окон
Alt+F3	Window ▶ Close	Закрыть активное окно
Alt+F5	Window ▶ User Screen	Показать экран пользователя
Shift+F6	Window ▶ Previous	Переход к предыдущему открытому окну
Ctrl+F5	Window ▶ Size/Move	Изменение размера (Shift+клавиши со стрелками) или положения (клавиши со стрелками) активного окна

**Таблица А.5.** Специальные клавиши и комбинации клавиш встроенной справочной системы

Клавиши	Команда меню	Функция
F1	Help ▶ Contents	Открывает контекстно-ориентированный экран справочной информации
F1 F1	Help ▶ Help on Help	Вызов справки по справочной системе
Shift+F1	Help ▶ Index	Вызов оглавления справочной системы
Alt+F1	Help ▶ Previous Topic	Показать предыдущий экран справочной информации
Ctrl+F1	Help ▶ Topic Search	Вызов контекстной информации по языку Паскаль

**Таблица А.6.** Специальные комбинации клавиш для запуска и отладки программ

Клавиши	Команда меню	Функция
Alt+F9	Compile ▶ Compile	Компиляция активного файла
Ctrl+F2	Run ▶ Program Reset	Сброс выполняемой программы в исходное состояние
Ctrl+F4	Debug ▶ Evaluate/Modify	Вычислить выражение или модифицировать значение переменной
Ctrl+F7	Debug ▶ Add Watch	Добавить выражение для просмотра
Ctrl+F8	Debug ▶ Toggle BreakPoint	Установить или удалить точку останова
Ctrl+F9	Run ▶ Run	Запустить программу на выполнение

В нижней части экрана находится строка статуса. Она содержит напоминание о назначении основных комбинаций клавиш, а также сообщает о выполняющемся действии.

Создание программы происходит следующим образом. После запуска интегрированной среды на экране должно появиться пустое активное окно редактирования. Если появившееся окно не пустое, то с помощью команды меню File ▶ New следует активизировать окно ввода нового текста. В верхней части окна редактирования появится название, которое среда автоматически присваивает новому файлу — NONAME00.PAS. После набора текста программы надо обязательно изме-

нить имя файла, иначе есть опасность потерять его, если он случайно будет замещен другим файлом с таким же стандартным именем. Рекомендуется также периодически производить сохранение файла с помощью нажатия клавиши F2 (через каждые 10–20 набранных строк), так как всегда имеется вероятность аварийной ситуации в работе компьютера, после которой не сохраненный на диске файл будет потерян. При первой записи файла на диск система предложит задать имя файла, причем расширение .PAS добавляется автоматически.

После завершения набора программы и ее записи на диск можно приступать к компиляции (клавиша F9). При этом компилятор может обнаружить ошибки в программе, и эти ошибки надо будет исправить. При возникновении неясных ситуаций, относящихся как к характеру ошибки, так и к правилам языка или правилам использования библиотечных подпрограмм смело пользуйтесь встроенной контекстной справочной системой Турбо Паскаля.

После завершения «борьбы» с синтаксическими ошибками можно запускать программу на выполнение (Ctrl+F9). Программа может благополучно отработать, но могут обнаружиться ошибки времени выполнения или алгоритмические ошибки. Если в программе не предусмотрена приостановка выполнения для просмотра результатов работы, выведенные на экран результаты перекрываются окном интегрированной среды. Временно убрать это окно и просмотреть результаты работы можно, нажав клавиши Alt+F5.

Для поиска ошибок времени выполнения можно использовать возможности встроенного отладчика. Сообщение о такой ошибке имеет вид

Run-time error <errnum> at <segment>:<offset>

где <errnum> — это код ошибки, а <segment>:<offset> — адрес в памяти, где произошла ошибка.

Интегрированный отладчик Турбо Паскаля дает возможность пошагового выполнения программы. При этом можно просматривать значения различных переменных, что иногда дает ценную информацию о реальной работе программы. Для запуска сеанса отладки выберите команду Run ▶ Trace Into или нажмите клавишу F7. При этом программа вначале компилируется, а затем начинается ее пошаговое выполнение. Каждый шаг заключается в выполнении очередной строки операторов, и происходит он при очередном нажатии на клавишу F7.

Во время отладки полезным может оказаться использование окна просмотра (Watch). Если нажать клавиши Alt+D для появления меню Debug и выбрать команду Add Watch (или нажать клавиши Ctrl+F7), то далее в появившемся окне можно набрать имя переменной, текущее значение которой необходимо узнать. Для добавления в окно просмотра других переменных можно повторно использовать команду Add Watch. Таким образом, используя возможность пошагового выполнения в сочетании с просмотром текущих значений переменных программы, можно провести достаточно подробный анализ ее работы. Это позволяет в случае неправильного поведения программы поставить ей достаточно точный «диагноз».

Завершить работу «зациклившейся» программы можно нажатием клавиш **Ctrl+Break**, а завершение работы интегрированной среды производится одновременным нажатием клавиш **Alt+X**.

В заключение следует заметить, что назначение «горячих клавиш» Турбо Паскаля можно переопределить. Можно также определить свои собственные макросы, облегчающие выполнение определенных «стандартных» последовательностей действий. Для этого используются специальный язык **TEM**L (Turbo Editor Macro Language) и компилятор для этого языка **TEM**C.EXE.

## **ПРИЛОЖЕНИЕ Б**

### **Список некоторых прерываний DOS и BIOS**

В этом приложении дается перечень некоторых прерываний DOS и BIOS с кратким описанием функций этих прерываний. Полный перечень прерываний и функций DOS и BIOS занимает несколько сотен страниц, поэтому выборка, сделанная для данного приложения, неизбежно субъективна. Более детальную информацию можно найти в технической документации.

#### **Прерывания BIOS**

##### **\$08 — системный таймер**

Специальные адреса памяти, связанные с этим прерыванием:

40:6C — Ежедневный счетчик (4 байта).

40:70 — Флаг 24-часового переполнения (1 байт). Используется некоторыми функциями DOS для настройки даты.

40:67 — Ежедневный счетчик для всех моделей PC после AT.

40:40 — Счетчик отключения мотора флоппи-дисковода. При установке в 0 отключает мотор.

## \$09 — клавиатура

Специальные адреса памяти, связанные с этим прерыванием:

- 40:17 — Нулевой байт флага состояния клавиатуры.
- 40:18 — Первый байт флага состояния клавиатуры.
- 40:1E — Буфер клавиатуры (32 байта).
- 40:71 — Седьмой бит байта, расположенного по этому адресу, устанавливается при нажатии клавиш Ctrl+Break.
- 40:72 — При нажатии клавиш Ctrl+Alt+Del по этому адресу записывается значение флага сброса — \$1234.
- 40:96 — Определение типа клавиатуры (AT, PS/2).
- 40:97 — Установка флагов индикаторов клавиатуры (AT, PS/2).
- FFFF:0 — Код перезагрузки, вызываемый при нажатии клавиш Ctrl+Alt+Del.

Нормальное время выполнения — 500 мкс.

## \$10 — видеорежим

Перечень функций (номер функции при вызове прерывания должен содержаться в регистре AH):

- 0 — установка видеорежима;
- 1 — установка вида курсора;
- 2 — установка положения курсора;
- 3 — считывание положения курсора;
- 5 — установка активной страницы;
- 6 — скроллинг активной страницы вверх;
- 7 — скроллинг активной страницы вниз;
- 8 — считать символ и его атрибут в позиции курсора;
- 9 — записать курсор и его атрибут в позиции курсора;
- B — установить цветовую палитру;
- C — записать пиксель в указанных координатах;
- D — считать пиксель в указанных координатах;
- F — определить текущий видеорежим;
- 10 — установить/определить регистры палитры (EGA/VGA);
- 11 — подпрограмма знакогенератора (EGA/VGA);
- 1C — запись/восстановление видеорежима (только для VGA).

## Установка видеорежима — функция 0

Входные параметры:

AH = 00.

В регистре AL — код видеорежима:

- 00 — 40×25 текстовый черно-белый (CGA, EGA, MCGA, VGA).
- 01 — 40×25 текстовый 16-цветный (CGA, EGA, MCGA, VGA).
- 02 — 80×25 текстовый 16 оттенков серого (CGA, EGA, MCGA, VGA).
- 03 — 80×25 текстовый 16-цветный (CGA, EGA, MCGA, VGA).
- 04 — 320×200 графический 4-цветный (CGA, EGA, MCGA, VGA).
- 05 — 320×200 графический 4-цветный (CGA, EGA, MCGA, VGA).
- 06 — 640×200 графический черно-белый (CGA, EGA, MCGA, VGA).
- 07 — 80×25 текстовый монохромный (MDA, HERC, EGA, VGA).
- 0D — 320×200 графический 16-цветный (EGA, VGA).
- 0E — 640×200 графический 16-цветный (EGA, VGA).
- 0F — 640×350 графический монохромный (EGA, VGA).
- 11 — 640×480 графический черно-белый (MCGA, VGA).
- 12 — 640×480 графический 16-цветный (VGA).
- 13 — 320×200 графический 256-цветный (MCGA, VGA).

## Установка вида курсора — функция 1

Входные параметры:

AH = 01;

CH = начальная строка развертки (положение верхнего края курсора, используются младшие 5 бит);

CL = конечная строка развертки (положение нижней части курсора, используются младшие 5 бит).

Отключение курсора: CX = \$2000.

## Установка положения курсора — функция 2

Входные параметры:

AH = 02;

BH = номер страницы (0 для графического режима);

DH = строка;

DL = столбец.

Положение отсчитывается от левого верхнего угла экрана, который имеет координаты (0, 0).

## **Определение положения и размера курсора — функция 3**

Входные параметры:

AH = 03;

BH = видеостраница.

Возвращает:

CH = начальная строка развертки;

CL = конечная строка развертки;

DH = строка;

DL = столбец.

## **Установка активной видеостраницы — функция 5**

Входные параметры:

AH = 05;

AL = номер новой страницы.

## **Скроллинг (прокрутка) вверх — функция 6**

Входные параметры:

AH = 06;

AL = количество строк прокрутки; предыдущие строки заполняются пробелами;

BH = атрибут пустой строки;

CH = строка верхнего левого угла окна прокрутки;

CL = столбец верхнего левого угла окна прокрутки;

DH = строка нижнего правого угла окна прокрутки;

DL = столбец нижнего правого угла окна прокрутки.

## **Скроллинг вниз — функция 7**

Входные параметры:

AH = 07.

Назначение остальных регистров то же самое, что и для скроллинга вверх.

## Считать символ и его атрибут в позиции курсора — функция 8

Входные параметры:

AH = 08;

BH = видеостраница.

Возвращает:

AH = атрибут символа (только для текстовых режимов);

AL = символ в позиции курсора.

## Записать символ и его атрибут в позиции курсора — функция 9

Входные параметры:

AH = 09;

AL = ASCII- код символа;

BH = видеостраница;

BL = атрибут символа (текстовые режимы) или цвет (графические режимы);

CX = количество записываемых символов (CX >= 1).

## Записать пиксель в указанных координатах — функция \$0c

Входные параметры:

AH = 0C;

AL = код цвета;

BH = номер страницы;

CX = номер столбца;

DX = номер строки.

## Считать пиксель в указанных координатах — функция \$0d

Входные параметры:

AH = 0D;

BH = номер видеостраницы;

CX = номер столбца;

DX = номер строки.

Возвращает:

AL = цвет считанного пикселя.

### **Определить текущий видеорежим — функция \$0f**

Входные параметры:

AH = 0F.

Возвращает:

AH = число столбцов;

AL = текущий видеорежим;

BH = видеостраница.

## **\$11 — флаги оборудования**

Возвращает в регистре AX следующие биты флагов:

- бит 1 — наличие математического сопроцессора;
- биты 4–5 — исходный видеорежим;
- биты 6–7 — количество флоппи-дисководов;
- бит 8 — 1, если установлен контроллер DMA;
- биты 9–11 — количество последовательных портов;
- биты 14–15 — количество параллельных портов.

## **\$12 — объем оперативной памяти**

Возвращает:

AX = количество 1-килобайтных блоков основной памяти.

## **\$13 — дисковые службы**

Перечень функций (номер функции при вызове прерывания должен содержаться в регистре AH):

- 0 — сброс дисковой системы;
- 1 — определение состояния диска;
- 2 — чтение секторов диска;
- 3 — запись секторов диска;

- 5 — форматирование дорожки;
- 8 — определение текущих параметров диска;
- 10 — проверка готовности диска;

При задании входных параметров обычно используется следующая схема:

- AH = номер функции;
- AL = количество секторов (1–128);
- CH = количество цилиндров (0–1023);
- CL = номер сектора (1–17);
- DH = номер головки (0–15);
- DL = номер дисковода (0 = A:, 1 = 2-й флоппи; \$80 = жесткий диск 0, \$81 = жесткий диск 1).

Возвращает:

CF = 0 в случае успеха, = 1 в случае ошибки.

## **Чтение секторов диска — функция 2**

Входные параметры:

- AH = 02;
- AL = количество считываемых секторов (1–128);
- CH = номер дорожки/цилиндра (0–1023);
- CL = номер сектора (1–17);
- DH = номер головки (0–15);
- DL = номер дисковода (0 = A:, 1 = 2-й флоппи; \$80 = жесткий диск 0, \$81 = жесткий диск 1).

## **Запись секторов диска — функция 3**

Входные параметры:

AH = 03.

Назначение остальных регистров то же, что и при чтении.

## **\$16 — клавиатура**

Перечень функций (номер функции при вызове прерывания должен содержаться в регистре AH):

- 0 — ожидание строки символов и ее считывание;
- 1 — определение статуса строки символов;

- 2 — получение статуса клавиши переключения регистров Shift;
- 3 — установка частоты опроса клавиатуры;
- 5 — запись в буфер клавиатуры.

## \$17 — службы принтера

Перечень функций (номер функции при вызове прерывания должен содержаться в регистре AH):

- 0 — печать символа;
- 1 — инициализация принтерного порта;
- 2 — считывание состояния принтерного порта.

Флаги состояния принтера (регистр AH):

- бит 0 — тайм-аут;
- бит 1 — ошибка ввода/вывода;
- бит 3 — выбранный принтер включен;
- бит 4 — нет бумаги;
- бит 6 — принтер не занят.

### Печать символа — функция 0

Входные параметры:

- AH = 00;
- AL = печатаемый символ;
- DX = используемый принтер (0–2).

Возвращает:

- AH = состояние принтера.

### Инициализация принтерного порта — функция 1

Входные параметры:

- AH = 01;
- DX = инициализируемый принтерный порт (0–2).

Возвращает:

- AH = состояние принтера.

## \$1a — часы

Перечень функций (номер функции при вызове прерывания должен содержаться в регистре AH):

- 0 — считывание счетчика системных часов;
- 1 — установка счетчика системных часов;
- 2 — считывание показаний системных часов (AT, PS/2);
- 3 — установка показаний системных часов (AT, PS/2);
- 4 — считывание даты (AT, PS/2);
- 5 — установка даты (AT, PS/2).

### Считывание счетчика системных часов — функция 0

Входные параметры:

AH = 00.

Возвращает:

- AL = флаг перехода через полночь, установлен в 1, если таковой имел место;
- CX = старшее слово счетчика;
- DX = младшее слово счетчика.

Значение счетчика увеличивается с периодом 55 мс. В полночь CX:DX равно нулю.

### Считывание показаний системных часов — функция 2

Входные параметры:

AH = 02.

Возвращает:

- CF = 0 — успешное считывание, = 1 — ошибка;
- CH = часы;
- CL = минуты;
- DH = секунды;
- DL = 1, если установлен переход на летнее время.

### Считывание даты — функция 4

Входные параметры:

AH = 04.

Возвращает:

CH = век (XIX или XX);  
 CL = год;  
 DH = месяц;  
 DL = день;  
 CF = 0 — успешное считывание, = 1 — ошибка.

## Прерывания DOS

Далее приводится краткий перечень прерываний DOS, который может оказаться полезным для последующего целенаправленного поиска более подробной информации об этих прерываниях и об их функциях.

\$20 — останов программы;  
 \$21 — диспетчер функций DOS;  
 \$22 — останов программы;  
 \$23 — адрес выхода по Ctl+Break;  
 \$24 — адрес обработчика критической ошибки;  
 \$25 — прямое считывание с диска;  
 \$26 — прямая запись на диск;  
 \$27 — перевод программы в резидентное состояние.

## \$21 — функции DOS

Перечень функций (номер функции при вызове прерывания должен содержаться в регистре AH):

0 — останов программы;  
 1 — ввод с клавиатуры с отображением вводимых символов;  
 2 — вывод на дисплей;  
 5 — вывод на принтер;  
 9 — печать строки;  
 A — буферизованный ввод с клавиатуры;  
 B — проверка стандартного состояния ввода;  
 C — очистка буфера клавиатуры и вызов функции ввода с клавиатуры;

- D — сброс файловых буферов;
- E — смена текущего диска;
- F — открытие файла;
- 10 — закрытие файла;
- 13 — удаление файла;
- 14 — последовательное чтение;
- 15 — последовательная запись;
- 16 — создание файла;
- 17 — переименование файла;
- 19 — определить текущий диск;
- 23 — определение размера файла;
- 25 — установка вектора прерываний;
- 29 — анализ имени файла;
- 2A — определение даты;
- 2B — установка даты;
- 2C — определение времени;
- 2D — установка времени;
- 30 — определение номера версии DOS;
- 31 — перевод программы в резидентное состояние;
- 33 — определение/установка состояния Ctrl+Break;
- 36 — определение размера свободного места на диске;
- 39 — создание подкаталога;
- 3A — удаление подкаталога;
- 3B — изменение текущего подкаталога;
- 41 — удаление файла;
- 43 — изменение атрибутов файла;
- 47 — определение текущего каталога;
- 56 — переименование файла;
- 57 — определение/установка даты и времени создания файла;
- 5C — запрещение/разрешение доступа к файлу;
- 60 — получение полного имени файла.

# **ПРИЛОЖЕНИЕ В**

## **Ответы и решения**

В данном приложении приведены ответы, возможные варианты решения и указания к решению некоторых упражнений курса.

### **Упражнение 1.1**

17    1.3107200000000E+0005

### **Упражнение 1.2**

101    1.47833593622636E+0000

### **Упражнение 1.3**

100    1.47833593622636E+0000

### **Упражнение 1.4**

F<sub>25</sub> = 75025

### **Упражнение 1.5**

F(24) = 16777216

### **Упражнение 1.6**

G( 3.14159265358830E+0000 ) = 6.00722071675781E+0009

### **Упражнение 1.7**

I = 3.06852819127574E-0001

### **Упражнение 1.8**

(5, 3)

**Упражнение 1.10**

```
program inequality;

const b = 100;

var
  i, j: integer;

function F(x, y: longint): longint;
begin
  F := x * x - 4 * x * y + 5 * y * y;
end;

begin
  for i := -b to b do
    for j := -b to b do
      if F(i, j) <= 100 then
        writeln(' x= ', i, ' y= ', j);
  write('Нажмите <Enter>: ');
  readln;
end.
```

**Упражнение 1.11**

```
program table_of_squares;

var
  j, s: word;

begin
  write(' ');
  for s := 0 to 9 do
    write(s:7);
  writeln;
  for j := 0 to 99 do
    begin
      write(j, ' ');
      for s := 0 to 9 do
        write(sqr(10 * j + s):7);
      writeln;
      if j mod 10 = 9 then
        begin
          writeln;
          readln;
        end;
    end;
  writeln;
  write('Нажмите <Enter>: ');
  readln;
end.
```

**Упражнение 1.12**

```

program file_of_values;

var
  j, s : word;
  out_file : text;

begin
  assign(out_file, 'table.dat');
  rewrite(out_file);
  writeln(out_file, 'Таблица квадратов');
  writeln(out_file);
  write(out_file, ' ');
  for s := 0 to 9 do
    write(out_file, s:7);
  writeln;
  writeln;
  for j := 0 to 99 do
    begin
      write(out_file, j, ' ');
      for s := 0 to 9 do
        write(out_file, sqr(10 * j + s):7);
      writeln(out_file);
    end;
  writeln(out_file);
  close(out_file);
end.

```

**Упражнение 1.13**

Самый простой способ решения этой задачи — вывод таблицы в файл со стандартным имеем prn, соответствующим принтеру:

```

assign(out_file, 'prn');
rewrite(out_file);

```

**Упражнения 2.1–2.4**

Ниже приводится пример программы для измерения затрат времени на выполнение арифметических операций с типом Integer и на выполнение циклов.

```

{$N+}
program test_time;

uses crt, timer;

var
  j, k: longint;
  m, n, p: integer;

begin
  clrscr;

```

```
m := 345;
n := 678;
get_time_start;
for j := 1 to 1000 do
    for k := 1 to 1000 do
        p := m + n;
get_time_stop;
writeln('1.000.000 сложений для типа Integer:');
put_time;
writeln;

m := 34;
n := 567;
get_time_start;
for j := 1 to 1000 do
    for k := 1 to 1000 do
        p := m * n;
get_time_stop;
writeln('1.000.000 умножений для типа Integer:');
put_time;
writeln;

m := 31760;
n := 789;
get_time_start;
for j := 1 to 1000 do
    for k := 1 to 1000 do
        p := m div n;
get_time_stop;
writeln('1.000.000 целочисленных делений для типа Integer:');
put_time;
writeln;

get_time_start;
for j := 1 to 1000 do
    for k := 1 to 1000 do
        begin
        end;
writeln('1.000.000 циклов for:');
get_time_stop;
put_time;
writeln;

get_time_start;
j := 0;
repeat
    inc(j);
    k := 0;
    repeat
        inc(k);
        until k = 1000;
until j = 1001;
```

```

get_time_stop;
writeln('1.000.000 циклов repeat:');
put_time;
writeln;

get_time_start;
j := 0;
while j < 1000 do
begin
  inc(j);
  k := 0;
  while k < 1000 do
  begin
    inc(k);
  end;
end;
get_time_stop;
writeln('1.000.000 циклов while:');
put_time;
writeln;

write('Нажмите <Enter>: ');
readln;
end.

```

**Упражнение 2.9**

```

program ascii2;

var
  k: word;

begin
  for k := 128 to 159 do
  begin
    write(k:10, chr(k) : 3);
    write((k + 32) : 10, chr(k + 32) : 3);
    write((k + 64) : 10, chr(k + 64) : 3);
    writeln((k + 96) : 10, chr(k + 96) : 3);
  end;
end.

```

**Упражнение 2.10**

```

program reverse;

uses crt;

var
  s,ss : string;
  p     : integer;
  ch    : char;

```

```

begin
  clrscr;
repeat
  textColor(7);
  write('Введите строку:');
  textColor(13);
  readln(s);
  for p := length(s) downto 1 do
begin
  ss := copy(s,p,1);
  textColor(9);
  write(ss);
end;
writeln;
textcolor(11);
writeln('Выход <Escape>, следующая строка <Enter>');
ch := readkey;
until ch=#27;
end.

```

**Упражнение 2.12**

```

program double;

var
  in_file, out_file : text;
  ss : string;

begin
  assign(in_file, 'my_file');
  assign(out_file, 'my_file dbl');
  reset(in_file);
  rewrite(out_file);
  while not eof(in_file) do
begin
  readln(in_file, ss);
  writeln(out_file, ss);
  writeln(out_file, ss);
end;
  close(in_file);
  close(out_file);
end.

```

**Упражнение 2.13**

```

program add_line_numbers;

var
  in_file, out_file : text;
  ss : string;
  n : word;
  temp, in_name, out_name : string[12];

```

```

procedure get_name;
begin
  writeln('Введите имя файла');
  readln(temp);
  n := pos('. ', temp);
  if n > 0 then delete(temp, n, length(temp) - n + 1);
  in_name := temp + '.pas';
  out_name := temp + '.pas$';
end;

begin
  get_name;
{$I-}
  assign(in_file, in_name);
  reset(in_file);
  if IOResult > 0 then halt;
  assign(out_file, out_name);
  rewrite(out_file);
  n := 0;
  while not eof(in_file) do
  begin
    inc(n);
    readln(in_file, ss);
    write(out_file, ss);
    writeln(out_file, '{ :(73 - length(ss)), n:3, ' }');
  end;
  close(in_file);
  close(out_file);
end.

```

#### Упражнение 2.14

```

program test_for_symmetry;

const
  n = 100;

type
  matrix = array[1..n, 1..n] of integer;

function symmetric(var A: matrix): boolean;

var
  j, k : integer;

begin
  symmetric := true;
  for j := 1 to n - 1 do
  begin
    for k := j + 1 to n do
    begin
      if A[j, k] <> A[k, j] then

```

```
begin
    symmetric := false;
    exit;
end;
end;
begin
end.
```

**Упражнение 2.15**

```
program find_largest_element;

const
    max = 10000;

type
    vector = array[1..max] of integer;

var
    first, last, largest : integer;
    V : vector;
    k : word;

procedure locate_largest;

var
    j : word;

begin
    first := 1;
    largest := V[1];
    for j := 2 to max do
    begin
        if V[j] > largest then
        begin
            first := j;
            largest := V[j];
        end;
    end;
end;

begin
    randomize;
    for k := 1 to max do
    begin
        V[k] := integer(random(6001)) - 3000;
    end;
    locate_largest;
    writeln('Максимальный элемент массива = ', largest);
end.
```

```
writeln('его индекс
readln;
end.
```

**Упражнение 2.18**

```
bool5 = FALSE
bool6 = FALSE
bool7 = TRUE
bool8 = TRUE
bool9 = FALSE
```

**Упражнение 2.20**

```
function round_up(x : longint; n : word) : longint;
var
  k : longint;

begin
  k := x mod n;
  if k = 0 then
    round_up := x
  else
    if x >= 0 then
      round_up := x + n - k
    else
      round_up := x - k;
end;
```

**Упражнение 2.21**

```
program no_ones;
var
  k : word;

function number_ones(w : word) : word;
var
  count, k : word;

begin
  count := 0;
  writeln('k':5, 'w shr k':10);
  for k := 0 to 15 do
    begin
      writeln(k:5, w:10);
      if odd(w) then
        inc(count);
      w := w shr 1;
    end;
  number_ones := count;
end;
```

```

begin
  randomize;
  k := number_ones(random(20000));
  writeln('Количество единиц = ', k);
  readln;
end.

```

### Упражнение 2.22

```

function power(x : real; n : integer) : real;
var
  y, p : real;
  m : word;

begin
  if n = 0 then
    begin
      power := 1.0;
      exit;
    end;
  m := abs(n);
  p := x;
  y := 1.0;
  begin
    if odd(m) then
      begin
        y := p * y;
        dec(m);
      end
    else
      begin
        p := sqr(p);
        m := m shr 1;
      end;
    end;
    if n < 0 then
      y := 1.0 / y;
  power := y;
end;

```

### Упражнение 3.3

Программа `file_of_string` создает на диске два файла — типизированный `strings.str` и текстовый `strings.txt`, состоящий из 100 строк «abc».

### Упражнение 3.5

```

program count_symbols;
uses crt;

const
  file_name = 'text.txt';

```

```

var
  ch : char;
  ff : file of char;
  aa : array[#32..#255] of longint;

begin
  for ch := #32 to #255 do
    aa[ch] := 0;
  assign(ff, file_name);
  reset(ff);
  while not eof(ff) do
  begin
    read(ff, ch);
    if ch > #31 then
      inc(aa[ch]);
  end;
  close(ff);
  clrscr;
  for ch := #32 to #255 do
    writeln('ASCII ', ord(ch), ' occurs ', aa[ch], ' times.');
  writeln;
  write('Нажмите <Enter>: ');
  readln;
end.

```

### Упражнение 3.6

```

program alphabet;
var
  F : file of char;
  ch : char;

procedure create_file;
var
  n : longint;
begin
  assign(F, 'temp');
  rewrite(F);
  randomize;
  for n := 1 to 1000 do
  begin
    ch := chr(random(256));
    write(F, ch);
    if ch >= ' ' then
      write(ch);
  end;
  writeln;
end;

procedure read_file;
var
  count : longint;

```

```

S : set of char;

begin
  reset(F);
  S := ['A'..'Z'];
  count := 0;
  while not eof(F) do
    begin
      read(F, ch);
      inc(count);
      S := S - [ch];
      if S = [] then
        break;
    end;
  write('Все символы ');
  if S = [] then
    writeln('считаны')
  else
    writeln('не считаны');
  writeln('Всего считано ', count, ' символов' );
end;

begin
  create_file;
  read_file;
  write('Нажмите <Enter>: '); readln;
end.

```

### Упражнение 3.13

```

procedure purge(var p : polynomial);
const
  min_eps = 1.0e-50;
var
  run1, run2 : polynomial;
begin
  if p <> nil then
    begin
      run1 := p;
      run2 := p^.next;
      while run2 <> nil do
        begin
          if abs(run2^.coeff) < min_eps then
            begin
              run1^.next := run2^.next;
              run2^.next := nil;
              dispose(run2);
              run2 := run1^.next;
            end
          else
            begin

```

```

    run1 := run2;
    run2 := run2^.next;
  end;
end;
if abs(p^.coeff) < min_eps then
begin
    run1 := p;
    p := p^.next;
    run1^.next := nil;
    dispose(run1);
  end;
end;
end;

```

**Упражнение 3.14**

```

procedure eval_der2(p : polynomial; c : real; var D0, D1, D2 : real);
var
  i, gap : word;
  run : polynomial;
begin
  D1 := 0.0;
  D2 := 0.0;
  if p = nil then
  begin
    D0 := 0.0;
    exit;
  end;
  D0 := p^.coeff;
  run := p;
  while run <> nil do
  begin
    if run^.next = nil then
      gap := degree(run)
    else
      gap := degree(run) - degree(run^.next);
    for i := gap downto 1 do
    begin
      D2 := D2 * c + 2.0 * D1;
      D1 := D1 * c + D0;
      D0 := D0 * c;
    end;
    run := run^.next;
    if run <> nil then
      D0 := D0 + run^.coeff;
  end;
end;

```

**Упражнение 5.1**

Для решения данной задачи требуется знание механики в объеме средней школы. Можно считать, что трение шара о поверхность бильярда отсутствует. В этом

случае шар после удара двигается равномерно и прямолинейно. При столкновении с бортом направление движения шара меняется по закону зеркального отражения. Таким образом, основное в программировании задачи — построение траектории. Можно использовать два подхода. В первом, исходя из начальных условий движения, строится отрезок траектории между точкой старта и точкой соударения с бортом. Затем определяется новая точка соударения, строится второй отрезок траектории и т. д. Во втором случае выбирается достаточно маленькое приращение по времени, и отрезки траекторий строятся для этих промежутков с одновременной проверкой, не произошло ли соударение с одним из бортов бильярда.

### **Упражнение 5.3**

Для решения задачи необходимо вспомнить формулы, описывающие движение траектории в однородном поле тяжести. Известно, что в этом случае движение происходит по параболе, параметры которой определяются начальными условиями движения стрелы. Задача решается методом подбора.

### **Упражнение 5.4**

См. комментарий к упражнению 5.3.

### **Упражнение 5.5**

Для решения этой задачи следует написать уравнения движения тела в поле тяжести. Затем уравнения решаются одним из численных методов. Хороший повод проверить физическую интуицию — попробуйте заранее предсказать особенности и характер движения тела в предлагаемой ситуации.

### **Упражнение 5.6**

См. примечание к упражнению 5.5.

### **Упражнение 7.1**

1 2 4 8 16 32 64 128 256 256 128 64 32 16 8 4 2 1

### **Упражнение 7.2**

```
function power(x : real; n : word) : real;
begin
  if n = 0 then
    power := 1.0
  else
    if odd(n) then
      power := x * power(x, n - 1)
    else
      power := sqr(power(x, n div 2))
end;
```

### **Упражнение 7.3**

```
program fixed_point_by_iterations;
var
  x : real;
```

```

function f(x : real) : real;
begin
  f := sqrt(x + 1.0)
end;

procedure solve;
var
  y : real;
begin
  y := f(x);
  if abs(y - x) >= 1.0e-9 then
    begin
      x := y;
      solve;
    end;
  end;
begin
  write('Введите начальное приближение x0: ');
  readln(x);
  if x < -1.0 then halt;
  write('Начальное приближение x0 = ', x:8:6);
  solve;
  writeln('Корень x = ', x:12:10);
  write('Нажмите <Enter>: ');
  readln;
end.

```

#### Упражнение 7.4

```

function legendre_poly(n : integer; x : real) : real;
begin
  if n > 1 then
    legendre_poly := ((2 * n - 1) * x * legendre_poly(n - 1, x)
    - (n - 1) * legendre_poly(n - 2, x)) / n
  else
    if n = 1 then
      legendre_poly := x
    else
      legendre_poly := 1.0;
end;

```

#### Упражнение 7.6

```

{$S+}
program modular_ackerman;
uses crt;
var
  m, x, y : word;
function Ack(x, y : word) : word;
begin

```

```

if keypressed then halt;
if x = 0 then
  Ack := (y + 1) mod m
else
  if y = 0 then
    Ack := Ack(x - 1, 1) mod m
  else
    Ack := Ack(x - 1, Ack(x, y - 1)) mod m;
end;

begin
  writeln;
  write('Введите m: '); readln(m);
  writeln;
  writeln(' ' : 5, 'Таблица значений функции Аккермана Ack(x, y):
  x = row, y = column');
  writeln;
  write('      ');
  for y := 0 to 9 do
    write(y : 4);
  writeln;
  writeln;
  for x := 0 to 5 do
begin
  begin
    write(x : 2, '      ');
    for y := 0 to 9 do
      begin
        write(Ack(x, y) : 4);
      end;
      writeln;
    end;
    write('Нажмите <Enter>: ');
    readln;
  end.
end.

```

### Упражнение 7.7

```

function hanoi(n : word) : longint;
begin
  if n = 1 then
    hanoi := 1
  else
    hanoi := 2 * hanoi(n - 1) + 1;
end;

function hanoi2(n : word) : longint;
var
  j : longint;
  k : word;

begin
  if n = 1 then

```

```

    hanoi2 := 1
else
begin
    j := 1;
    for k := 2 to n do
        j := 2 * j + 1;
    hanoi2 := j;
end;
end;

begin
    writeln(hanoi(20));
    writeln(hanoi2(20));
end.

```

**Упражнение 7.8**

```

program queen8_picture;

uses graph;

const
    height = 50;
    width = 50;
    max = 7;
    w0 = 100;
    h0 = 40;

type
    solution = array[0..max] of word;

var
    gd, gm : integer;
    WhiteBlank, BlackBlank,
    WhiteQueen, BlackQueen : pointer;
    size : word;
    position : solution;
    j, k, count : word;
    c_str : string[4];
    in_file : file of solution;

{$I-}
procedure initialize;
begin
    assign(in_file, '8queens.dat');
    reset(in_file);
    count := 0;
    if IOResult <> 0 then
begin
    writeln('Файл "8queens.dat" не найден');
    writeln('Сгенерируйте этот файл программой "queens"');
    write('Нажмите <Enter>: ');

```

```
readln;
halt;
end;
gd := detect;
InitGraph(gd, gm, '');
if GraphResult <> 0 then halt;
size := ImageSize(0, 0, width, height);
GetMem(WhiteBlank, size);
GetMem(BlackBlank, size);
GetMem(WhiteQueen, size);
GetMem(BlackQueen, size);
SetLineStyle(SolidLn, 0, NormWidth);
SetBkColor(Red);
SetFillStyle(SolidFill, Yellow);
SetTextStyle(DefaultFont, HorizDir, 3);
Bar(0, 0, width, height);
GetImage(0, 0, width, height, BlackBlank^);
SetColor(Blue);
OutTextXY(13, 13, 'Q');
GetImage(0, 0, width, height, BlackQueen^);

SetFillStyle(SolidFill, Green);
Bar(0, 0, width, height);
GetImage(0, 0, width, height, WhiteBlank^);
SetColor(Blue);
OutTextXY(13, 13, 'Q');
GetImage(0, 0, width, height, WhiteQueen^);
ClearDevice;
end;

{$I+}
procedure close1;

begin
  FreeMem(BlackQueen, size);
  FreeMem(WhiteQueen, size);
  FreeMem(BlackBlank, size);
  FreeMem(WhiteBlank, size);
  CloseGraph;
end;

begin
  initialize;
  SetTextStyle(DefaultFont, HorizDir, 2);
  SetColor(White);
  while not eof(in_file) do
  begin
    inc(count);
    str(count, c_str);
    read(in_file, position);
    ClearDevice;
    OutTextXY(20, 10, 'Solution ' + c_str);
  end;
end.
```

```

for j := 0 to max do
begin
  for k := 0 to max do
  begin
    if odd(j + k) then
    begin
      if position[j] = max - k then
        PutImage(w0 + j * width, h0 + k * height,
                  WhiteQueen^, NormalPut)
      else
        PutImage(w0 + j * width, h0 + k * height,
                  WhiteBlank^, NormalPut);
    end
    else
    begin
      if position[j] = max - k then
        PutImage(w0 + j * width, h0 + k * height,
                  BlackQueen^, NormalPut)
      else
        PutImage(w0 + j * width, h0 + k * height,
                  BlackBlank^, NormalPut);
    end;
  end;
end;
OutTextXY(400, 460, 'Press <Enter>: ');
readln;
end;
close1;
close(in_file);
end.

```

### Упражнение 7.9

```

program hilbert_curve;
uses crt, graph, graphs;
const
  size : array[1..7] of integer
  = (380, 127, 57, 25, 12, 6, 3);
var
  order : word;
  ch : char;
  x, y : integer;

procedure draw_U(n : word; plus : boolean);

procedure rotate(plus : boolean);
var
  t : integer;
begin

```

```
if plus then
begin
    t := x;
    x := -y;
    y := t;
end
else
begin
    t := x;
    x := y;
    y := -t;
end;
end:{rotate}

begin{draw_U}
if n > 0 then
begin
    if odd(n) then
begin
        draw_U(n - 1, not plus);
        linerel(x, y);
        rotate(plus);
        draw_U(n - 1, plus);
        linerel(x, y);
        draw_U(n - 1, plus);
        rotate(plus);
        linerel(x, y);
        draw_U(n - 1, not plus);
    end
    else
begin
        draw_U(n - 1, not plus);
        rotate(plus);
        linerel(x, y);
        draw_U(n - 1, plus);
        rotate(not plus);
        linerel(x, y);
        rotate(not plus);
        draw_U(n - 1, plus);
        linerel(x, y);
        rotate(plus);
        draw_U(n - 1, not plus);
    end;
end;
end:{draw_U}

begin
open_graph;
for order := 1 to 7 do
begin
    setcolor(8 + order);
    moveto(100, 50);
```

```
x := size[order];
y := 0;
if odd(order) then
  draw_U(order, true)
else
  draw_U(order, false);
setcolor(white);
outtextxy(30, 0, 'Press <Space>: ');
ch := readkey;
cleardevice;
end;
close_graph;
end.
```

# Алфавитный указатель

## Символы

' (апостроф), 72  
' (запятая), 30  
' (одиночная кавычка), 34  
\$ (знак доллара), 84, 144  
() (круглые скобки), 31  
\ (обратная косая черта), 111, 151  
\* (звездочка), 30, 82  
+ (знак плюс), 82  
- (знак минус), 82  
. (точка), 29, 30, 47, 117  
/ (косая черта), 31, 82  
; (точка с запятой), 17, 21, 22, 24, 26, 30, 37  
< (меньше), 39  
<= (меньше или равно), 39  
<> (не равно), 39  
= (знак равенства), 34, 41, 47  
> (больше), 39  
>= (больше или равно), 39  
@, оператор, 145  
[] (квадратные скобки), 80  
{} (квадратные скобки), 77  
^ (символ «шляпки»), 124  
\_ (символ подчеркивания), 28  
{\$E}, директива компилятора, 90  
{\$!+}, директива компилятора, 102, 106  
{\$L...}, директива компилятора, 221  
{\$M...}, директива компилятора, 161  
{\$N+}, директива компилятора, 37  
{\$R+}, директива компилятора, 307  
{\$S+}, директива компилятора, 307  
{} (фигурные скобки), 15, 18, 29, 37

## А

Abs, функция, 39, 83, 90  
absolute, зарезервированное слово, 165  
Addr, функция, 145

and, зарезервированное слово, 15, 84, 87, 165, 172, 189, 252, 335  
Arc Tan, функция, 90  
array, зарезервированное слово, 14, 77  
ASCII, 64, 100  
Assign, процедура, 44, 100, 112

## Б

Bar, процедура, 210  
begin, зарезервированное слово, 15, 37, 126  
BINOBJ.EXE, утилита, 221  
BIOS, 150  
    модуля расширения, 150  
BlockRead, процедура, 110, 115  
BlockWrite, процедура, 110, 115, 265  
Boolean, тип данных, 63  
Borland Graphics Interface, 231  
Break, процедура, 28, 87  
Byte, тип данных, 63, 82, 99, 165, 211

## С

CapsLock, режим клавиатуры, 172  
case, зарезервированное слово, 15, 23, 55, 122  
CD-ROM-дисковод, 194  
Char, тип данных, 63, 64, 66, 99  
ChDir, процедура, 160  
Chr, функция, 66  
Circle, процедура, 204  
ClearDevice, процедура, 219  
Close, процедура, 44, 45, 101  
CloseGraph, процедура, 202  
ClrEol, процедура, 307  
ClrScr, процедура, 35, 55, 74  
Сотр, тип данных, 64, 90  
const. зарезервированное слово, 14, 19, 61, 207

constructor, зарезервированное слово, 15, 363, 364, 365  
**C**  
 Continue, процедура, 28  
 Copy, функция, 75  
 Cos, функция, 90  
 CR, символ, 100  
 CRT, модуль, 55, 68, 74, 204, 206, 307, 333  
   ClrEol, процедура, 307  
   ClrScr, процедура, 35, 55, 74  
   Delay, процедура, 79, 204, 261, 335  
   GotoXY, процедура, 74, 78, 178  
   keypressed, процедура, 206  
   NoSound, процедура, 333  
   ReadKey, функция, 68, 69, 70  
   Sound, процедура, 333  
   TextColor, процедура, 78  
   WhereY, функция, 369  
 Ctrl-последовательности, 66

**D**  
 Debug, меню  
   Add Watch, команда, 441  
 Dec, функция, 35  
 Delay, процедура, 79, 204, 261, 335  
 Delete, процедура, 74, 129  
 Delphi, 377  
 destructor, зарезервированное слово, 15, 364, 365  
 Detect, константа, 201  
 Dispose, процедура, 126  
   расширенный синтаксис, 365  
 div, зарезервированное слово, 15, 31, 83  
 DMA, 334  
 do, зарезервированное слово, 15, 25, 26, 27, 117  
 DOS, модуль, 58, 105, 156, 157, 158, 161  
   Exec, процедура, 162, 178  
   FindFirst, процедура, 158  
   FindNext, процедура, 158  
   FExpand, функция, 105, 156, 160  
   FSplit, процедура, 112, 156  
   GetEnv, функция, 161  
   GetTime, процедура, 58  
   Intr, процедура, 168  
   MSDOS, процедура, 168  
   PathStr, тип данных, 105, 156  
   SearchRec, тип данных, 157  
   SetFTime, процедура, 157  
   Swap Vectors, процедура, 162  
   UnpackTime, процедура, 157  
 DOSError, переменная, 158  
   коды ошибок, 162  
 Double, тип данных, 64, 94  
 downto, зарезервированное слово, 15, 27, 32

**E**  
 else, зарезервированное слово, 24  
 end, зарезервированное слово, 24, 37, 47, 53, 122, 126, 361  
 Eof, функция, 45  
 Erase, процедура, 110, 158, 160  
 Exclude, процедура, 81  
 Exec, процедура, 162, 178  
 Exit, процедура, 42, 87  
 Exp, функция, 90  
 Extended, тип данных, 19, 37, 47, 64, 90, 91, 113  
**F**  
 False (ложь), логическое значение, 21, 87, 110, 136, 172, 195, 319  
 far, директива компилятора, 41  
 far, зарезервированное слово, 41  
 FExpand, функция, 105, 156, 160  
 file, зарезервированное слово, 14  
 File, меню  
   New, команда, 440  
 FileSize, функция, 110  
 FillEllipse, процедура, 261  
 Fill Pattern Type, тип данных, 211  
 FindFirst, процедура, 158  
 FindNext, процедура, 158  
   for, зарезервированное слово, 15, 26, 27, 28, 32, 33  
 Frac, функция, 90  
 FreeMem, процедура, 127, 129  
 FSplit, процедура, 112, 156  
   function, зарезервированное слово, 14, 53

**C**  
 GetDir, процедура, 160  
 GetEnv, функция, 161  
 GetImage, процедура, 212, 262  
 GetMaxX, функция, 203  
 GetMaxY, функция, 203  
 GetMcm, процедура, 127, 129  
 GetPixel, функция, 216  
 GetTime, процедура, 58  
 GotoXY, процедура, 74, 78, 178  
 Graph, модуль, 200, 208  
   Bar, процедура, 210  
   Circle, процедура, 204  
   ClearDevice, процедура, 219  
   CloseGraph, процедура, 202  
   Detect, константа, 201  
   FillEllipse, процедура, 261  
   FillPatternType, тип данных, 211  
   GetImage, процедура, 212, 262  
   GetMaxX, функция, 203  
   GetMaxY, функция, 203

**G**  
**Graph**, модуль (*продолжение*)  
 GetPixel, функция, 216  
 GraphResult, функция, 202  
 HorizDir, константа, 219  
 ImageSize, функция, 213  
 InitGraph, процедура, 201, 202  
 Line, процедура, 204  
 LineRel, процедура, 322  
 OutTextXY, процедура, 207, 211, 219  
 PutImage, процедура, 212, 262  
 PutPixel, процедура, 216  
 Rectangle, процедура, 210  
 RegisterBGIdriver, процедура, 222  
 RegisterBGIfont, процедура, 222  
 RestoreCRTMode, процедура, 202,  
 262  
 SetActivePage, процедура, 231, 262  
 SetBkColor, процедура, 203  
 SetColor, процедура, 203, 210  
 SetFillStyle, процедура, 210  
 SetGraphMode, процедура, 202  
 SetRGBPalette, процедура, 215  
 SetTextJustify, процедура, 219  
 SetTextStyle, процедура, 219  
 SetViewPort, процедура, 226  
 SetVisualPage, процедура, 231, 262  
 VertDir, константа, 219  
 константы  
 векторных шрифтов, 219  
 видеоадаптеров, 200  
 видеорежимов, 201  
 выравнивания текста, 219  
 логических операций, 213  
 стилей заполнения, 210  
 цветов, 203  
**GraphResult**, функция, 202

**H**  
**Halt**, процедура, 42, 87, 106, 307  
**HorizDir**, константа, 219

**I**  
 IEEE 754, стандарт, 91  
**if**, зарезервированное слово, 15, 21, 33, 81  
**ImageSize**, функция, 213  
**implementation**, зарезервированное слово,  
 15, 47  
**in**, зарезервированное слово, 81  
**Inc**, функция, 35  
**Include**, процедура, 81  
**inherited**, зарезервированное слово, 363  
**InitGraph**, процедура, 201, 202  
**Input**, файл ввода, 42, 102  
**Insert**, процедура, 74  
**Int**, функция, 90

**Integer**, тип данных, 63, 75, 80, 82, 456  
**interface**, зарезервированное слово, 15, 47  
**Intr**, процедура, 168  
**IOResult**, функция, 102, 106, 158

**K**  
**keypressed**, процедура, 206

**L**  
**Length**, функция, 74  
**LF**, символ, 100  
**Line**, процедура, 204  
**LineRel**, процедура, 322  
**Ln**, функция, 90  
**Longint**, тип данных, 35, 63, 80, 82, 90,  
 126, 157, 165, 307

**M**  
**Maxint**, константа, 82  
**MaxLongInt**, константа, 82  
**Mem**, предопределенный массив, 165  
**MemL**, предопределенный массив, 165  
**MernW**, предопределенный массив, 165  
**mod**, зарезервированное слово, 15, 31, 43, 83  
**Move**, процедура, 265  
**MS-DOS**, операционная система  
 базовый модуль, 150  
 байт состояния клавиатуры, 165  
 интерпретатор команд, 151  
 нерезидентная часть, 153  
 резидентная часть, 153  
 операционная среда, 154  
 переменные окружения, 161  
 распределение памяти, 152  
 состав, 150  
 специальные адреса памяти, 162  
 список оборудования, 163  
 статус флоппи-дисковода, 163  
 утилиты, 151  
 ядро, 154  
**MSDOS**, процедура, 168

**N**  
**New**, процедура, 126  
 расширенный синтаксис, 365  
**nil**, константа, 126  
**NoSound**, процедура, 333  
**not**, зарезервированное слово, 15, 84, 87,  
 172  
**NumLock**, режим клавиатуры, 164, 172

**O**  
**Object Pascal**, 377  
**object**, зарезервированное слово, 15, 361,  
 363

*Odd*, функция, 35, 83  
*of*, зарезервированное слово, 14, 15, 24  
*Ofs*, функция, 145  
*Options*, меню  
  Compiler, окно диалога  
    8087/80287, флагок, 38  
    Complete Boolean eval,  
      флагок, 88  
    Emulation, флагок, 38  
*or*, зарезервированное  
  слово, 15, 84, 87, 172  
*Ord*, процедура, 64, 72, 89  
*Ord*, функция, 66  
*Output*, файл вывода, 42  
*OutTextXY*, процедура, 207, 211, 219

**P**

*PathStr*, тип данных, 105, 156  
*Pi*, константа, 90  
*PIT (Programming Interval Timer)*, 334  
*Pointer*, зарезервированное  
  слово, 98  
*Pos*, функция, 75, 111  
*PPI (Programmable Peripheral Interface)*, 334  
*Pred*, функция, 89  
*private*, зарезервированное  
  слово, 361  
*prn*, файл принтера, 456  
*procedure*, зарезервированное  
  слово, 14, 53, 364  
*program*, зарезервированное  
  слово, 14, 16, 28  
*Ptr*, процедура, 165  
*Ptr*, функция, 145  
*public*, зарезервированное слово, 15  
*PutImage*, процедура, 212, 262  
*PutPixel*, процедура, 216

**Q**

*QR-алгоритм*, 400

**R**

*Random*, функция, 225  
*Randomize*, процедура, 225  
*Read*, процедура, 42, 101, 102, 112  
*ReadKey*, функция, 68, 69, 70  
*ReadLn*, процедура, 30, 42, 67, 101, 107,  
  112  
*Real*, тип данных, 19, 32, 64, 90  
*record*, зарезервированное  
  слово, 14, 116  
*Rectangle*, процедура, 210  
*RegisterBGIdriver*, процедура, 222

*RegisterBGIfont*, процедура, 222  
*Registers*, тип данных, 166  
*Rename*, процедура, 110, 158  
*repeat*, зарезервированное  
  слово, 15, 25, 28, 35, 42  
*Reset*, процедура, 101, 112, 113  
*RestoreCRTMode*, процедура, 262  
*RestoreCrtMode*, процедура, 202  
*Rewrite*, процедура, 44, 101, 112, 113  
*RGB-представление цвета*, 214  
*Round*, функция, 90

**S**

*ScrollLock*, режим клавиатуры, 172  
*SearchRec*, тип данных, 157  
*Seg*, функция, 145  
*Seg0040*, константа, 163  
*set*, зарезервированное слово, 80  
*SetActivePage*, процедура, 231, 262  
*SetBkColor*, процедура, 203  
*SetColor*, процедура, 203, 210  
*SetFillStyle*, процедура, 210  
*SetFTimc*, процедура, 157  
*SetGraphMode*, процедура, 202  
*SetRGBPalette*, процедура, 215  
*SetTextJustify*, процедура, 219  
*SetTextStyle*, процедура, 219  
*SetViewPort*, процедура, 226  
*SetVisualPage*, процедура, 231, 262  
*shi*, зарезервированное слово, 15, 83  
*Shortint*, тип данных, 63, 80, 82  
*shr*, зарезервированное слово, 15, 83  
*Sin*, функция, 90  
*Single*, тип данных, 64  
*SizeOf*, функция, 82, 118, 129  
*Sound*, процедура, 333  
*SoundBlaster*, 345  
  порты ввода/вывода, 345  
  программирование  
    воспроизведение звука  
    произвольной частоты, 352  
    воспроизведение ноты, 351  
    выключение звука, 352  
    инициализация, 350  
    параметры несущей, 353  
    режимы звуковоспроизведения,  
      352  
  регистры, 346  
*Sqr*, функция, 32, 83, 90  
*Sqrt*, функция, 90  
*Str*, процедура, 75, 119, 207  
*string*, зарезервированное слово, 14, 45  
*string*, тип данных, 71  
*Succ*, функция, 89  
*Swap Vectors*, процедура, 162

System, модуль, 55, 70, 83, 90, 100, 129, 158, 165, 166  
Abs, функция, 39, 83, 90  
Addr, функция, 145  
ArcTan, функция, 90  
Assign, процедура, 44, 100, 112  
BlockRead, процедура, 110, 115  
BlockWrite, процедура, 110, 115, 265  
Break, процедура, 28, 87  
ChDir, процедура, 160  
Chr, функция, 66  
Close, процедура, 44, 45, 101  
Continue, процедура, 28  
Copy, функция, 75  
Cos, функция, 90  
Dec, функция, 35  
Delete, процедура, 74, 129  
Dispose, процедура, 126  
расширенный синтаксис, 365  
Eof, функция, 45  
Erase, процедура, 110, 158, 160  
Exclude, процедура, 81  
Exit, процедура, 42, 87  
Exp, функция, 90  
FileSize, 41учкия, 110  
Frac, функция, 90  
FrceMcm, процедура, 127, 129  
GetDir, процедура, 160  
GetMem, процедура, 127, 129  
Halt, процедура, 42, 87, 106, 307  
Inc, функция, 35  
Include, процедура, 81  
Insert, процедура, 74  
Int, функция, 90  
IOResult, функция, 102, 106, 158  
Length, функция, 74  
Ln, функция, 90  
Maxint, константа, 82  
MaxLongInt, константа, 82  
Move, процедура, 265  
New, процедура, 126  
расширенный синтаксис, 365  
Odd, функция, 35, 83  
Ofs, функция, 145  
Ord, процедура, 64, 72, 89  
Ord, функция, 66  
Pos, функция, 75, 111  
Pred, функция, 89  
Ptr, процедура, 165  
Ptr, функция, 145  
Random, функция, 225  
Randomize, процедура, 225  
Read, процедура, 42, 101, 102, 112  
ReadLn, процедура, 30, 42, 67, 101, 107, 112

Rename, процедура, 110, 158  
Reset, процедура, 101, 112, 113  
Rewrite, процедура, 44, 101, 112, 113  
Round, функция, 90  
Seg, функция, 145  
Seg0040, константа, 163  
Sin, функция, 90  
SizeOf, функция, 82, 118, 129  
Sqr, функция, 32, 83, 90  
Sqrt, функция, 90  
Str, процедура, 75, 119, 207  
Succ, функция, 89  
Tan, функция, 90  
Trunc, функция, 35, 90  
UpCase, функция, 66  
Val, процедура, 75, 84  
Write, процедура, 30, 32, 42, 101, 102, 112, 119, 201, 207  
WriteLn, процедура, 30, 32, 34, 38, 42, 44, 101, 107, 112, 113, 119, 201

## Т

Tan, функция, 90  
TEML (Turbo Editor Macro Language), 442  
Text, тип данных, 44, 100  
TextColor, процедура, 78  
then, зарезервированное слово, 15, 21, 33  
to, зарезервированное слово, 15, 26, 32  
True (истина), логическое значение, 21, 87, 110, 136, 173, 298, 317  
Trunc, функция, 35, 90  
type, зарезервированное слово, 14, 41, 47, 97

## У

unit, зарезервированное слово, 14, 47  
UnpackTime, процедура, 157  
until, зарезервированное слово, 15, 25, 35, 42  
UpCase, функция, 66  
uses, зарезервированное слово, 35, 46, 200

## В

Val, процедура, 75, 84  
var, зарезервированное слово, 14, 18, 58, 77, 80, 98, 124  
VertDir, константа, 219  
virtual, зарезервированное слово, 15, 364

## W

Watch, окно диалога, 441  
WhereY, функция, 369

`while`, зарезервированное слово, 15, 25, 28, 38, 45  
`with`, зарезервированное слово, 117  
`Word`, тип данных, 19, 32, 61, 63, 80, 82, 83, 127, 165, 189, 210, 213, 214, 226, 333, 334  
`Write`, процедура, 30, 32, 42, 101, 102, 112, 119, 201, 207  
`WriteLn`, процедура, 30, 32, 34, 38, 42, 44, 101, 107, 112, 113, 119, 201

**X**

`xog`, зарезервированное слово, 14, 84, 87, 189, 252

**A**

абсолютные переменные, 165  
адреса памяти, 123, 144  
Аккермана *функция*, 316  
алгоритмы  
    QR-разложения, 400  
    Брезенхама, 236, 247  
    гауссова исключения, 392  
    Евклида, 308  
    Коэна—Сазерленда, 243  
    метод Гаусса, 409, 417  
    метод касательных, 405  
    метод Ньютона, 405, 406, 419  
    метод Ромберга, 409, 414  
    метод Рунге—Кутта, 424, 425  
    метод Симпсона, 409  
    метод стрельбы, 432  
    метод Хьюна, 423  
    метод Эйлера, 422  
        модифицированный, 423  
    перебора с возвратами, 316  
    перевода арабских чисел  
в римские, 313  
решето Эратосфена, 381  
схема Горнера, 389  
удаления невидимых линии, 286  
алгоритм художника, 290  
    Феттиса, 388  
алфавит языка Паскаль, 14  
анимация, 227, 290  
    спрайты, 298  
арифметические выражения, 30, 37  
арифметические операции, 30  
    вычисление остатка, 31, 83  
    вычитание, 30  
    деление, 30, 31  
    остаток от деления, 43  
    порядок выполнения, 31  
    приоритеты, 30  
    сложение, 30

арифметические операции (*продолжение*)  
    умножение, 30  
    целочисленное деление, 31, 83  
атрибуты файлов, 157

**B**

библиотечные модули, 16, 35, 46  
бинарные операции, 82  
бит знака, 84  
Брезенхама алгоритм, 236, 247  
булев тип данных, 26, 87  
булевы операции, 87  
буфер клавиатуры, 67

**V**

ввод данных  
    проверка корректности  
        ввода, 178  
ввод с клавиатуры, 67  
векторные преобразования  
    перенос, 269  
векторы, 266  
    векторное произведение, 268  
    вычитание, 267  
    масштабирование, 273  
    норма, 266  
    ортогональность, 279  
    поворот, 270  
        со сдвигом, 273  
    проекции, 268, 279  
    сдвиг, 272  
    скалярное произведение, 268, 279  
    сложение, 267  
    умножение матрицы  
на вектор, 269  
    умножение на скаляр, 267  
векторы прерываний, 154  
ветвление, 23  
вещественные типы данных, 90, 98, 99  
видеоадаптер, 200  
    CGA, 256  
    EGA, 256  
    HGC, 256  
    IBM 8514/a, 257  
    IBM XGA, 258  
    MCGA, 257  
    MDA, 256  
    SVGA, 258  
    VGA, 257  
        организация видеопамяти, 265  
видео ПЗУ, 256  
видеоконтроллер, 255, 265  
видеопамять, 255  
графические режимы, 200  
графические ускорители, 256

видеоадаптер (*продолжение*)  
драйвер, 200  
подключение к исполняемому  
файлу, 221  
устройство, 255  
цифроаналоговый преобразователь,  
256  
видеопамять, страницы, 178  
видеорежимы  
палитра, 214  
текстовые и графические, 197  
видовые координаты, 274  
вложенные подпрограммы, 56  
возврат каретки, 100  
встроенный динамик, 333  
управление, 334  
выбор, 23  
селектор, 24  
список значений, 24  
вывод  
значений переменных, 34  
перевод строки, 30  
символов на экран, 180, 181  
символьной строки, 30  
строк текста, 34  
формат вывода, 43  
данных, 101  
численного значения, 30  
вызовов процедур, 20  
выражения, 20, 37  
арифметические, 30, 37  
логические, 21, 25, 87  
ускоренное вычисление, 88

**Г**

Гаусса метод, 409, 417  
гауссова исключения алгоритм, 392  
выбор ведущего элемента  
полный, 392  
частичный, 392  
Герона метод, 38  
Гильберта кривая, 326  
гиперболические функции, 46  
глобальные переменные, 56  
головоломка «Ханойская башня», 316  
Горнера схема, 389  
графические  
координаты, 198  
отличие от декартовых, 199  
прimitивы, 208  
страницы, 227  
графический режим  
восстановление содержимого  
экрана, 262  
вывод окружностей, 247

графический режим (*продолжение*)  
вывод отрезков, 234  
вертикальных, 234  
горизонтальных, 234  
наклонных, 236  
отсечение линий, 243  
вывод текста, 258  
инициализация, 232  
копирование экрана, 265  
курсор, 261  
область вывода, 242  
сохранение содержимого экрана, 262  
страницы, 261

**Д**

датчик случайных чисел, 225  
двойные интегралы, 411  
деление отрезка пополам, 40  
диапазон значений, 36  
диапазоны, 77, 98  
динамик встроенный, 333  
управление, 334  
динамические переменные, 123  
диофантовы уравнения, 32  
линейные, 33  
нелинейные, 34  
директивы компилятора  
{\$E}, 90  
{\$!+}, 102, 106  
{\$L...}, 221  
{\$M...}, 161  
{\$N+}, 37  
{\$R+}, 307  
{\$S+}, 307  
far, 41

диски, логические, 151  
дисковод CD-ROM, 194  
дифференциальные уравнения, 422  
метод Рунге—Кутта, 424, 425  
метод Хьюона, 423  
метод Эйлера, 422  
модифицированный, 423  
драйверы устройств, 153  
дублирование имен, 129

**Е**

Евклида алгоритм, 308

**З**

заголовок  
подпрограммы-функции, 37  
программы, 28, 37  
загрузчик системы, 150  
задача о расстановке восьми ферзей, 317  
задачи краевые, 431

- записи, 116  
 ноля, 116  
 с вариантами, 121  
 зарезервированные слова, 13, 14  
 absolute, 165  
 and, 15, 84, 87, 165, 172, 189, 252, 335  
 array, 14, 77  
 begin, 15, 37, 126  
 case, 15, 23, 55, 122  
 const, 14, 19, 61, 207  
 constructor, 15, 363, 364, 365  
 destructor, 15, 364, 365  
 div, 15, 31, 83  
 do, 15, 25, 26, 27, 117  
 downto, 15, 27, 32  
 else, 15, 22, 24  
 end, 14, 15, 24, 37, 47, 53, 122, 126, 361  
 far, 41  
 file, 14  
 for, 15, 26, 27, 28, 32, 33  
 function, 14, 53  
 if, 15, 21, 33, 81  
 implementation, 15, 47  
 in, 81  
 inherited, 363  
 interface, 15, 47  
 mod, 15, 31, 43, 83  
 not, 15, 84, 87, 172  
 object, 15, 361, 363  
 of, 14, 15, 24  
 or, 15, 84, 87, 172  
 Pointer, 98  
 private, 361  
 procedure, 14, 53, 364  
 program, 14, 16, 28  
 public, 15  
 record, 14, 116  
 repeat, 15, 25, 28, 35, 42  
 set, 80  
 shi, 15, 83  
 shr, 15, 83  
 string, 14, 45, 71  
 then, 15, 21, 33  
 to, 15, 26, 32  
 type, 14, 41, 47, 97  
 unit, 14, 47  
 until, 15, 25, 35, 42  
 uses, 35, 46, 200  
 var, 14, 18, 58, 77, 80, 98, 124  
 virtual, 15, 364  
 while, 15, 25, 28, 38, 45  
 with, 117  
 xor, 84, 87, 189, 252
- звук  
 синтез по волновым таблицам, 345  
 частотный синтез, 345  
 оператор, 345  
 звука характеристики амплитуда, 341  
 высота, 341  
 модуляция, 343  
 гибающая, 342  
 атака, 343  
 задержка, 343  
 затухание, 343  
 полное затухание, 343  
 участок стабильного звучания, 343  
 основной тон, 341  
 тембр, 341  
 звуковая карта, 344  
 SoundBlaster, 345  
 состав  
 FM-синтезатор, 344  
 блок MPU, 345  
 микширующее устройство, 344  
 модуль цифровой обработки звука, 344  
 знаковый бит, 84
- И**  
 идентификатор, 16  
 идентификаторы. См. имена  
 имена  
 глобальные, 56  
 дублирование, 129  
 локальные, 55  
 область видимости, 56  
 регистр символов, 30  
 типов, 97  
 функций, 37  
 имя программы, 16, 28  
 инициализация переменных, 29, 72  
 интегралы двойные, 411  
 интегрирование численное, 409  
 адаптивное, 409, 412  
 квадратурная формула, 409  
 коэффициенты, 409  
 узлы, 409  
 метод  
 Гаусса, 409, 417  
 Ромберга, 409, 414  
 Симпсона, 409  
 интегрированная среда  
 завершение работы, 442  
 запуск программы, 441  
 клавиатурные комбинации, 438

- интегрированная среда (*продолжение*)  
компиляция программы, 441  
макросы, 442  
окно редактирования, 440  
переопределение клавиатурных комбинаций, 442  
пошаговое выполнение программы, 441  
прерывание работы программы, 442  
просмотр значений переменных, 441  
результатов работы программы, 441  
сохранение файла, 441  
строка меню, 437  
строка статуса, 440  
исполняемые операторы, 29, 37  
исполняемый файл, 16  
итерационные методы, 307, 380, 390, 400, 405  
степенной метод, 398
- К**
- карта звуковая, 344  
SoundBlaster, 345  
состав  
    FM-синтезатор, 344  
    блок MPU, 345  
    микширующее устройство, 344  
    модуль цифровой обработки звука, 344  
касательных метод, 405  
клавиатура  
буфер, 67  
буфер клавиатуры, 170  
индикаторы, 172  
код символа, 69  
код сканирования, 69, 170  
расширенный код, 69  
режим  
    CapsLock, 172  
    NumLock, 164, 172  
    ScrollLock, 172  
клавиатурные комбинации, 438  
клавиши, специальные, 69  
код ASCII, 64  
командная строка  
    параметры, 115  
комбинаторика, 326  
перечисление подмножеств, 328  
сочетания, 327  
комментарии, 18, 29  
компиляция, 16, 30  
комплексные числа, 118
- константы, 14  
    Detect, 201  
    HorizDir, 219  
    MaxInt, 82  
    MaxLongInt, 82  
    nil, 126  
    Pi, 90  
    Seg0040, 163  
    VertDir, 219  
видеоадаптеров, 200  
нетипизированные, 19  
тиปизированные, 19  
координаты видовые, 274  
графические, 198  
    отличие от декартовых, 199  
мировые, 274, 281  
однородные, 273  
сферические, 274  
экранные, 275, 281  
Коха кривая, 320  
Козна—Сазерленда алгоритм, 243  
краевые задачи, 431  
кривая  
    Гильберта, 326  
    Коха, 320  
    Пеано, 323  
курсор  
    мыши, 185  
        активная зона, 189, 252  
        область скрытия, 190  
        ограничение области  
            перемещения, 188  
        отображение, 185  
        спецификация, 189  
        установка в заданное положение, 186  
        форма, 189, 250, 252, 337  
    текстовый, отображение, 177  
        установка в заданную позицию, 180, 181
- Л**
- Лагерра полиномы, 385  
Лежандра полиномы, 312, 385, 417, 418  
линейная алгебра, 391  
логические выражения, 25, 87  
    ускоренное вычисление, 88  
логические значения  
    False (ложь), 21, 34, 42, 87, 110, 136, 172, 195, 319  
        True (истина), 21, 34, 87, 110, 136, 173, 298, 317  
    логические отношения, 66  
        равенство, 34  
логический диск, 151

- логическое выражение, 21  
 локальные  
     описания, 39, 47  
     переменные, 55
- M**
- массивы, 76  
     базовый тип, 77  
     векторы, 76  
     и логические отношения 79  
     индексы, 76
- диапазон, 77  
     матрицы, 76  
     многомерные, 77  
     размерность, 77  
     ранг, 77  
     форма, 77
- математический сопроцессор, 37, 64, 90  
     эмуляция, 38
- матрицы, 127, 269, 391  
     ввод и вывод значений  
     элементов 398  
         вывод на экран, 129  
         диагональные, 400  
         матрица Якоби, 406  
         нахождения наибольшего собственного значения, 398  
             регулярные, 400  
             обращая матрица, 269  
             перемножение, 269  
             поворота, 269  
             произведение, 127  
             симметричные, 397  
             сложение, 269  
             собственные пик горы, 397  
             собственные значения, 397
- кратность, 400  
     трехдиагональные 400  
     умножение на вектор, 269  
     факторизация, 400
- матричная алгебра, 366
- метод  
     Гаусса, 409, 417  
     Герона, 38  
     деления отрезка пополам, 40  
     касательных, 405  
     Ньютона, 405, 406, 419  
         скорое го сходимости, 409  
         сходимость, 406  
     Ромберга, 409, 414  
     Рунге Кутта, 424, 425  
     Симпсона, 409  
         трехточечный, 410  
     стрельбы, 432  
     Хьюса, 423
- метод (*продолжение*)  
     Эйлера, 422  
         модифицированный, 423
- методы итерационные, 380, 390, 400, 405  
     смешанный метод, 398
- мировые координаты, 274, 281
- многократное использование кода, 37, 46
- множества, 80, 99  
     базовый тип, 80
- операции отношений, 81  
     пустое, 81
- Кантора, 322
- модули, 16, 35, 36, 37, 46  
     CRT, 35, 55, 68, 74, 204, 206, 307,  
         333, 369
- DOS, 58, 105, 156, 157, 158, 161  
     Graph, 200, 208
- System, 28, 30, 32, 34, 35, 38, 39, 42,  
     44, 45, 55, 70, 83, 90, 100, 129, 158,  
     165, 166
- интерфейсная секция, 47
- секция реализации, 47
- модуль расширения BIOS, 150
- мономы, 116  
     перемножение, 117
- мышь, 183  
     виртуальный экран, 184  
     использование в графическом режиме, 250  
     количество нажатий кнопок, 186  
     курсор, 185  
         активная зона, 189, 252  
         область скрытия, 190  
         ограничение области  
             перемещения, 188  
             отображение, 185  
             спецификация, 189  
         установка в заданное положение, 186
- форма, 189, 250, 252, 337
- определение числа кнопок, 191
- относительное смещение, 190
- переключение между  
     videостраницами, 191  
     положение курсора, 186  
     проверка двойного нажатия  
         кнопок, 187  
     проверка наличия, 191  
     состояние, 190  
     кнопок, 186  
     считывание параметров, 191
- H**
- наибольший общий делитель, 308, 380  
     нелинейные уравнения, 307, 405

- 
- нетипизированные  
  константы, 19  
  файлы, 113  
Ньютона метод, 405, 406, 419  
  скорость сходимости, 409  
  сходимость, 406
- O**
- Output, файл вывода, 102  
объектно-ориентированное  
программирование, 359  
инкапсуляция, 362  
наследование, 362  
объекты, 360  
  взаимодействие, 360  
  виртуальные методы, 364, 369  
  деструктор, 364, 365, 369, 372  
  динамические, 365  
  конструктор, 364, 365, 369, 372  
  методы, 360, 361  
  свойства, 360  
таблица виртуальных  
  методов, 365  
  экземпляры, 360  
позднее связывание  
  методов, 364, 366  
полиморфизм, 366  
раннее связывание методов, 364  
совместимость объектных  
  типов, 366
- ограничители комментария, 29  
однородные координаты, 273  
округление, 35  
ООП, 359  
операторные скобки, 16  
операторы, 14  
@, 145  
ввода/вывода, 15  
выбора, 23, 55  
селектор, 24  
список значений, 24  
вывода, 34, 101  
исполняемые, 29, 37  
присваивания, 20, 30, 37  
значения функции, 37  
совместимость, 98  
пустые, 17  
составные, 16, 20, 21  
структурные, 21  
управляющие, 15  
условные, 21, 33, 42, 81  
вложенные, 21, 23  
оптимальная последовательность  
  вложения, 23  
  полная версия, 22
- операторы (*продолжение*)  
  цикла, 25  
  бесконечного, 25, 26, 42  
  выход из середины, 28  
  прерывание итерации, 28  
  с постусловием (*repeat*), 25, 28,  
    35, 42  
  с предусловием (*while*), 25, 28,  
    38, 45  
  со счетчиком (*for*), 26, 28, 30,  
    32, 33
- операции  
  арифметические, 30  
  вычисление остатка, 31, 83  
  вычитания, 30  
  деление, 31  
  деления, 30  
  порядок выполнения, 31  
  приоритеты, 30  
  сложения, 30  
  умножение, 30  
  целочисленное деление, 31, 83  
  бинарные, 82  
  булевы, 87  
  над множествами, 81  
  объединение, 81  
  пересечение, 81  
  разность, 81  
  приоритет, 94  
  сдвига, 83  
операционная система  
  функции, 150  
описания, 16  
  переменных, 29  
  констант, 19  
  локальные, 39, 47  
  переменных, 18, 29  
  подпрограмм-функций, 37  
  типов, 97
- остановка выполнения программы, 42  
отладка, 29, 37  
  профилирование, 58  
отношения, 87  
логические, 66  
нестрогое неравенства, 39  
отрезки типов, 89, 98  
ошибки  
  ввода/вывода, 102  
  времени выполнения, 441  
компиляции  
  String constant exceeds line, 73  
  Type mismatch, 31, 98  
округления, 435  
хода выполнения программы  
  Arithmetict overflow, 83

**П**

палитра цветовая, 214  
 видеоадаптера VGA, 214  
 параметры  
 командной строки, 115  
 нетипизированные, 58  
 параметры-значения, 57, 61  
 параметры-константы, 61  
 параметры-переменные, 58, 61  
 фактические, 57  
 формальные, 57  
 Паскаля треугольник, 327  
 Пеано кривая, 323  
 перебор, 380  
   с возвратами, 316  
 перевод строки, 100  
 переменные, 13  
 DOSError, 158  
   коды ошибок, 162  
   абсолютные, 165  
   глобальные, 56  
   динамические, 123  
   освобождение памяти, 126, 127  
   размещение в памяти, 126  
   инициализация, 72  
   локальные, 55  
   ограничение размера памяти, 77  
   строковые, 45  
    атрибут длины, 71  
    доступ к отдельным  
    байтам, 72  
    конкатенация, 73  
    сравнение, 76  
    указатели, 123  
 базовый тип, 123  
 разыменование, 124, 126  
 файловые, 44, 100, 101  
 переполнение, 36  
 перечисляемые типы, 26, 89  
 пиксели, 198  
 побочные эффекты, 56, 58  
 погрешность вычислений, 32  
 подпрограммы, 36  
 вложенные, 56  
 параметры, 57  
 нетипизированные, 58  
 параметры-значения, 57, 61  
 параметры-константы, 61  
 параметры-переменные, 58, 61  
 фактические, 57  
 формальные, 57  
 побочные эффекты, 56, 58  
 процедуры, 37, 53  
 функции, 37, 53  
 позднее связывание, 364, 366

полигоны, 286  
   нормаль, 286  
 полиномы, 135  
   вычисление значения, 142  
   деление, 140  
   дифференцирование, 141  
   Лагерра, 385  
   Лежандра, 312, 385, 417, 418  
   перемножение, 78, 140  
   суммирование, 139  
   схема Горнера, 389  
   умножение на скаляр, 139  
   Чебышева, 385  
   Эрмита, 385  
 порты ввода/вывода, 265, 334, 345  
 порядковые типы данных, 99  
 предложения списания  
 переменных, 18  
 констант, 19  
 предопределенные массивы  
 Mem, 165  
 MemL, 165  
 MemW, 165  
 представление численных значений, 91  
 преобразование  
 типа, 82  
 Хаусхолдера, 400  
 прерывание выполнения: программы, 438  
 прерывания, 155, 166, 335  
   \$08 (системный таймер), 443  
   \$09 (клавиатура), 170, 444  
   \$10 (видеосервис), 1, 73, 181, 232, 258  
   запись пикселя, 447  
   запись символа и атрибутов, 447  
   перечень функций, 444  
   прокрутка вверх, 446  
   прокрутка вниз, 446  
   считывание пикселя, 447  
   считывание положения и размера  
    курсора, 446  
    считывание символа  
    и атрибутов, 447  
    считывание текущего  
   видеорежима, 448  
   установка активной  
    видеостраницы, 446  
    установка вида курсора, 445  
    установка видеорежима, 445  
    установка положения  
   курсора, 445  
   \$11 (список оборудования), 448  
   \$12 (размер памяти), 448  
   \$13 (дисковый ввод/вывод)  
   задание входных  
   параметров, 449

- прерывания (*продолжение*)  
запись секторов диска, 449  
перечень функций, 448  
чтение секторов диска, 449  
\$16 (ввод/вывод  
с клавиатуры), 170, 173, 449  
\$17 (принтер)  
инициализация порта принтера,  
    450  
перечень функций, 450  
печатать символа, 450  
флаги состояния принтера, 450  
\$1a (системные часы)  
перечень функций, 451  
считывание даты, 451  
считывание показаний  
    системных часов, 451  
считывание счетчика системных  
    часов, 451  
\$21 (вызов функций MS-DOS), 452  
\$33 (мыши), 183  
DOS, 452  
приведение типов, 99, 118  
признак конца файла, 100  
принцип модульности, 36  
приоритет операций, 94  
присваивание, 20, 30, 37  
массивов, 79  
совместимость, 98  
программы, 16  
остановка выполнения, 42  
резидентные, 153  
проективная геометрия, 273  
проекции  
    параллельные, 275  
    диметрическая, 280  
    изометрическая, 275, 279  
    косоугольные, 278, 280  
    ортогональная, 275, 278  
    перспективные, 275  
        расчет экранных координат, 275  
прокрутка экрана, 181  
простейшая программа, 28  
простые числа, 381  
профилирование, 58  
процедуры, 20, 53  
    Assign, 44, 100, 112  
    Bar, 210  
    BlockRead, 110, 115  
    BlockWrite, 110, 115, 265  
    Break, 28, 87  
    CbDir, 160/  
    Cirde, 204  
    ClearDevice, 219  
    Close, 450  
процедуры (*продолжение*)  
CloseGraph, 202  
Ch-Eol, 307  
ClrScr, 35, 55, 74  
Continue, 28  
Delay, 79, 204, 261, 335  
Delete, 74, 129  
Dispose, 126  
    расширенный синтаксис, 365  
Erase, 110, 158, 160  
Exclude, 81  
Exec, 162, 178  
Exit, 42, 87  
FillEllipse, 261  
FmdFirst, 158  
FmdNext, 158  
FreeMem, 127, 129  
FSplit, 112, 156  
GetDir, 160  
GetImage, 212, 262  
GetMem, 127, 129  
GetTime, 58  
GotoXY, 74, 78, 178  
Halt, 42, 87, 106, 307  
Include, 81  
InitGraph, 201, 202  
Insert, 74  
Intr, 168  
keypressed, 206  
Line, 204  
Move, 265  
MSDos, 168  
New, 126  
    расширенный синтаксис, 365  
NoSound, 333  
Ord, 64, 72, 89  
OutTextXY, 207, 211, 219  
Ptr, 165  
PutImage, 212, 262  
PutPixel, 216  
Randomize, 225  
Read, 42, 101, 102, 112  
ReadLn, 30, 42, 67, 101, 107, 112  
Rectangle, 210  
RegisterBGIdver, 222  
RegisterBGIfont, 222  
Rename, 110, 158  
Reset, 101, 112, 113  
RestoreCRTMode, 202, 262  
Rewrite, 44, 101, 112, 113  
SetActivePage, 231, 262  
SetBkCoter, 203  
SetColor, 203, 210  
SetFillStyle, 210  
SetFTime, 157

процедуры (*продолжение*)

- SetGraphMode, 202
- SetRGBPalette, 215
- SetTextJustify, 219
- SetTextStyle, 219
- SetViewPort, 226
- SetVisualPage, 231, 262
- Sound, 333
- Str, 75, 119, 207
- Swap Vectors, 162
- TextColor, 78
- UnpackTime, 157
- Val, 75, 84
- Write, 30, 32, 42, 101, 102, 112, 119, 201, 207
- WntLn, 30, 32, 34, 38, 42, 44, 101, 107, 112, 113, 119, 201
- список параметров, 53
- прямой доступ к памяти, 334
- пустое множество, 81
- пустой оператор, 17

## **P**

раздел операторов, 16

раздел описаний, 16, 37

- описания переменных, 18
- описания типов, 97

разностные уравнения, 314

разрешение монитора, 198

разрядность компьютера, 62

раннее связывание, 364

регистры процессора, 166

резидентные программы, 153

рекурсия, 380

- глубина, 308

- косвенная, 305

- прямая, 305

решето Эратосфена, 381

римские цифры, 313

Ромберга метод, 409, 414

Рунге—Кутта метод, 424, 425

ряды степенные, 386

- возвведение в степень, 386, 390

- деление, 386, 387

- единичный, 391

- композиция, 386, 387

- нулевой, 391

- обращение, 386, 388

## **C**

связные списки, 130

- добавление элементов, 131, 133

- создание, 132

- удаление элементов, 131, 134

- уничтожение, 132

сегмент, 123, 144, 166

- данных, 145, 166

- кода, 166

- стека, 166

символы

- специальные, 15

- CR (возврат каретки), 113, 114

- LF (перевод строки), 113, 114

- управляющие, 66

символьный тип данных, 26, 99

Симпсона метод, 409

- трехточечный, 410

сингулярность, 406

синтез звука по волновым

- таблицам, 345

системное время

- считывание, 177

системные часы, 58

системы координат

- декартова, 266

правая и левая, 269

системы линейных уравнений, 391

- гауссово исключение, 392

с выбором ведущего

элемента, 392, 393

- масштабирование, 397

- обратная подстановка, 392, 393

приведение к треугольному

- виду, 392, 393

системы нелинейных уравнений

сингулярные, 406

системы обыкновенных

- дифференциальных уравнений, 422

скалярные типы данных, 26, 77, 89

скобки

- операторные, 16

- фигурные, 18

случайные числа, 225

смещение, 123, 144

совместимость типов данных, 98

- по присваиванию, 98

сопrocessор математический, 37, 64, 90

- эмуляция, 38

составной оператор, 16, 20, 21

состояние кнопок мыши, 186

специальные символы, 15

- CR (возврат каретки), 113, 114

- LF (перевод строки), 113, 114

- возврат каретки, 100

- перевод строки, 100

списки связные, 130

- добавление элементов, 131, 133

- создание, 132

- удаление элементов, 131, 134

- уничтожение, '132

спрайта, 298  
вывод на экран, 299  
захват с экрана, 299  
создание, 298  
сохранение фона при движении, 302  
структура данных, 298  
удаление, 298  
ссылочный тип данных, 123  
стандарт IEEE 754, 91  
степенные ряды, 386  
возвведение в степень, 386, 390  
деление, 386, 387  
единичный, 391  
композиция, 386, 387  
нулевой, 391  
обращение, 386, 388  
страницы видеопамяти, 178  
стрельбы метод, 432  
строки символов, 71  
строковые  
переменные, 45  
атрибут длины, 71  
доступ к отдельным байтам, 72  
конкатенация, 73  
сравнение, 76  
типы данных, 99  
структурная программа, 17, 28  
структурные операторы, 21  
сферические координаты, 274

## Т

текстовые файлы, 100  
признак конца файла, 100  
форматы UNIX и MS-DOS, 114  
текстовый файл, 44  
тело программы, 16  
тело цикла, 25  
теория чисел, 379  
тестирование, 29, 37  
типовизированные  
константы, 19  
файлы, 112  
типы данных  
Boolean, 63  
Byte, 63, 82, 99, 165, 211  
Char, 63, 64, 66, 99  
Comp, 64, 90  
Double, 64, 94  
Extended, 19, 37, 47, 64, 90, 91, 113  
FillPatternType, 211  
Integer, 63, 75, 80, 82, 456  
Longint, 35, 63, 80, 82, 90, 126, 157,  
165, 307  
PathStr, 105, 156  
Real, 19, 32, 90

Registers, 166  
SearchRec, 157  
Shortint, 63, 80, 82  
Single, 64  
string, 71  
Text, 44, 100  
Word, 19, 32, 61, 63, 80, 82, 83, 127,  
165, 189, 210, 213, 214, 226, 333, 334  
булев, 26, 87  
вещественные, 90, 98, 99  
группы типов, 62  
диапазоны, 98  
задаваемые программно, 62, 97  
записи, 116  
поля, 116  
с вариантами, 121  
имена, 97  
контроль, 31  
множества, 98, 99  
описание, 97  
определение новых, 41  
отрезки типов, 89, 98  
переменных, 18  
перечисляемые, 26, 89  
порядковые, 99  
предопределенные, 62  
преобразование, 82  
приведение типов, 99, 118  
простые, 62  
вещественные, 62  
порядковые, 62  
символьный, 26, 99  
скалярные, 26, 77, 89  
совместимость, 98  
по присваиванию, 98  
ссылочные, 123  
строковые, 71, 99  
структурные, 62  
массивы, 76  
множества, 80  
указатели, 99  
файлы  
нетипизированные, 113  
текстовые, 100  
типовизированные, 112  
функциональные, 41  
целые, 26, 98, 99  
трансляция, 16  
треугольник Паскаля, 327

## У

указатели, 99, 123  
nil, константа, 126  
базовый тип, 123  
разыменование, 124, 126

управляющие операторы, 15  
 управляющие символы, 66  
 CR (возврат каретки), 113, 114  
 LF (перевод строки), 113, 114  
 прогон страницы, 45  
 уравнения  
 диофантовы, 32  
     линейные, 33  
     нелинейные, 34  
 дифференциальные, 422  
     нелинейные, 307, 405  
     разностные, 314  
 условный оператор, 21, 33, 42, 81  
 вложенный, 21, 23  
     оптимальная  
         последовательность, 23  
 полная версия, 22  
 устройства ввода/вывода, 42

**Ф**

файловые переменные, 44, 100, 101  
 файлы, 42  
     rgp, файл принтера, 456  
     атрибуты, 157  
     закрытие, 101  
     исполняемые, 16, 151  
     командные, 151  
      маршрут, 151  
         абсолютный, 152  
         относительный, 152  
     нетипизированные, 113  
     открытие, 101  
     полное имя файла, 151  
     составное имя файла, 151  
     текстовые, 44  
         признак конца файла, 100  
         форматы UNIX и MS-DOS, 114  
     типовизированные, 112  
     шаблон имени, 151  
 фактические параметры, 57  
 Феттисса алгоритм, 388  
 Фибоначчи числа, 305  
 флаги, 168  
 формальные параметры, 57  
 форматирование текста программы, 17  
 фракталы, 321  
     Abs, 39, 83, 90  
     Addr, 145  
     ArcTan, 90  
     Chr, 66  
     Copy, 75  
     Cos, 90  
     Dec, 35  
     Eof, 45  
     Exp, 90

функции (*продолжение*)  
 FExpand, 105, 156, 160  
 FileSize, 110  
 Frac, 90  
 GetEnv, 161  
 GetMaxX, 203  
 GetMaxY, 203  
 GetPixel, 216  
 GraphResult, 202  
 ImageSize, 213  
 Inc, 35  
 Int, 90  
 IOResult, 102, 106, 158  
 Length, 74  
 Ln, 90  
 Odd, 35, 83  
 Ofs, 145  
 Ord, 66  
 Pos, 75, 111  
 Pred, 89  
 Ptr, 145  
 Random, 225  
 ReadKey, 68, 69, 70  
 Round, 90  
 Seg, 145  
 Sin, 90  
 SizeOf, 82, 118, 129  
 Sqr, 32, 83, 90  
 Sqrt, 90  
 Succ, 89  
 Tan, 90  
 Trunc, 35, 90  
 UpCase, 66  
 WhereY, 369  
     арифметические, 37  
     внутренняя структура, 37  
     гиперболические, 46  
     завершение выполнения, 42  
     имена, 37  
     список параметров, 37  
     тело, 37  
     тип результата, 37  
 функциональный тип данных, 41  
 функция Аккермана, 316

**Х**

характеристики звука  
 амплитуда, 341  
 высота, 341  
 модуляция, 343  
 огибающая, 342  
     атака, 343  
     задержка, 343  
      затухание, 343  
     полное затухание, 343

характеристики звука (*продолжение*)  
участок стабильного  
звучания, 343  
основной тон, 341  
тембр, 341

Хаусхольдера преобразование, 400  
Хьюна метод, 423

## Ц

целые типы данных, 26, 98, 99  
циклы, 25  
бесконечные, 25, 26, 42  
выход из середины, 28  
прерывание итерации, 28  
рекомендации по применению, 27  
с постусловием, 25, 28, 35, 42  
с предусловием, 25, 28, 38, 45  
со счетчиком, 26, 28, 30, 32, 33  
управляющая  
переменная, 26, 27, 32  
цифры римские, 313

## Ч

частные производные, 406  
частотный синтез звука, 345  
оператор, 345  
часы системные, 58  
Чебышева полиномы, 385  
«черепашья» графика, 320  
числа  
простые, 381  
Фибоначчи, 305  
шестнадцатеричные, 144

численное интегрирование, 409  
адаптивное, 409, 412  
квадратурная формула, 409  
коэффициенты, 409  
узлы, 409  
метод  
Гаусса, 409, 417  
Ромберга, 409, 414  
Симпсона, 409

## Ш

шестнадцатеричные  
числа, 84, 144  
шрифты  
векторные, 218  
подключение к исполняемому  
файлу, 221  
растровые, 218

## Э

Эйлера метод, 422  
модифицированный, 423  
экранные координаты, 275, 281  
Эратосфена решето, 381  
Эрмита полиномы, 385  
эффект перспективы, 273  
линия горизонта, 273  
направление  
наблюдения, 273  
точка наблюдения, 273

## Я

якобиева матрица, 406