Самоучитель

Никита Культин

Программирование в Turbo Pascal 7.0 и Delphi

3-е издание

От алгоритма до работающей программы

Язык программирования Turbo Pascal

Работа с файлами и графикой

Введение в объектно-ориентированное программирование

Работа в среде Delphi

bhv + CD

Никита Культин

Самоучитель
Программирование в Turbo Pascal 7.0 и Delphi 3-е издание

Санкт-Петербург

«БХВ-Петербург»

2007

УДК 681.3.068
ББК 32.973.26-018.1
К90

Культин Н. Б.

К90 Программирование в Turbo Pascal 7.0 и Delphi: 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2007. — 400 с.: ил. + CD-ROM — (Самоучитель)

ISBN 978-5-9775-0109-5

Книга позволяет научиться программированию на языке Pascal в среде Turbo Pascal. Рассмотрен весь процесс создания программы: от разработки алгоритма до получения результата — готовой программы. Приведено описание языка программирования и среды разработки; рассмотрены основные типы данных и алгоритмические структуры. Уделено внимание обработке символьной информации, использованию динамических структур данных, работе с файлами, выводу данных на печать, программированию графики. Описана среда визуального программирования Delphi и показаны основы разработки в ней Windows-приложений.

Книга отличается доступностью изложения материала, большим количеством наглядных примеров и адресована студентам, школьникам старших классов и всем, кто изучает программирование. На прилагаемом компакт-диске находятся приведенные в книге тексты программ.

Для начинающих программистов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Ирина Иноземцева*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 28.07.05.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 32,25.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0109-5

© Культин Н. Б., 2007
© Оформление, издательство "БХВ-Петербург", 2007

Оглавление

Предисловие **1**

ЧАСТЬ I. TURBO PASCAL **3**

Глава 1. Среда программирования Turbo Pascal **5**

Установка 5
Начало работы 7
Первая программа 8
  Набор текста программы 11
  Компиляция 13
  Ошибки времени компиляции 14
Запуск программы 16
Ошибки времени выполнения 17
Создание exe-файла 18
Завершение работы с Turbo Pascal 18
Внесение изменений в программу 19
Запуск программы из операционной системы 20

Глава 2. Введение в программирование **21**

Этапы разработки программы 21
  Определение требований к программе 22
  Разработка алгоритма 22
  Кодирование 22
  Отладка 22
  Тестирование 22
Алгоритм 24
Программа 24
Компиляция 25
Тип данных 26
  Целый тип 26
  Вещественный тип 26

Символьный тип 26
Строковый тип 27
Логический тип 27
Переменная 27
Объявление переменной 27
Константы 28
Числовые константы 28
Строковые и символьные константы 29
Логические константы 29
Именованная константа 29
Инструкция присваивания 30
Выражение 30
Тип выражения 32
Выполнение инструкции присваивания 32
Функции 33
Ввод и вывод 34
Инструкции *WRITE* и *WRITELN* 34
Инструкция *readln* 36
Структура простой программы 37
Запись инструкций программы 37
Стиль программирования 39

Глава 3. Алгоритмические структуры 40

Условие 42
Выбор 44
Инструкция *IF* 44
Инструкция *CASE* 48
Циклы 52
Цикл *FOR* 53
Цикл *REPEAT* 55
Цикл *WHILE* 58

Глава 4. Массивы 61

Объявление массива 61
Доступ к элементу массива 62
Ввод массива 63
Вывод массива 66
Поиск минимального элемента 68
Сортировка массива 70
Сортировка методом прямого выбора 70
Сортировка методом прямого обмена 73

Поиск в массиве....75
Метод перебора....75
Бинарный поиск....77
Многомерные массивы....81
Ошибки при использовании массивов....89

Глава 5. Символы и строки....92

Символы....92
Строки....97
Ввод строк....98
Преобразование строчных букв в прописные....99
Функции манипулирования строками....101
Функция *LENGTH*....101
Процедура *DELETE*....102
Функция *POS*....103
Функция *COPY*....104
Процедура *VAL*....106

Глава 6. Процедуры и функции....109

Функция....109
Стандартные функции....110
Библиотечные функции....111
Функция программиста....112
Процедура....116
Процедура программиста....116
Вызов процедуры....117
Параметр-переменная и параметр-значение....119
Локальные и глобальные переменные....121
Процедура или функция?....123
Структурное программирование....123

Глава 7. Стандартные модули....127

Доступ к библиотечным функциям и процедурам....127
Модуль Crt....128
Управление курсором....128
Управление цветом....130
Очистка экрана....132
Ввод символа с клавиатуры....132

Глава 8. Модуль программиста....137

Структура модуля....137
Подготовка текста модуля....140

Компиляция модуля 140
Использование модуля 140

Глава 9. Файлы 142

Объявление файла 142
Назначение файла 143
Открытие файла 143
Закрытие файла 143
Запись в файл 143
Ошибки доступа к файлу 146
Чтение из файла 148
Чтение строк 152
Конец файла 153
Вывод на печать 155
Пример программы 159
Система проверки знаний 159

Глава 10. Типы данных, определяемые программистом 169

Перечисляемый тип 169
Интервальный тип 172
Запись 173
Объявление записи 173
Доступ к полям записи 174
Инструкция *WITH* 174
Массив записей 175
Ввод и вывод записей в файл 176
Динамические структуры данных 179
Переменные-указатели 180
Динамические переменные 181
Списки 182

Глава 11. Графика 191

Видеосистема компьютера 191
Модуль Graph 192
Инициализация графического режима 192
Экран в графическом режиме 194
Графические примитивы 195
Цвет и вид линий 195
Цвет и стиль закраски области 197
Точка 199
Линия 199

Окружность 200
Эллипс 200
Прямоугольник 201
Круг и сектор....202
Эллипс и эллиптический сектор 203
Вывод текста....203
Инструкции *WRITE* и *WRITELN* 203
Процедуры *OutText* и *OutTextXY* 204
Примеры программ 207
График 207
Анимация 212

Глава 12. Рекурсия....223

Понятие рекурсии....223
Пример программы: поиск пути 225
Пример программы: поиск кратчайшего пути 231

Глава 13. Отладка программы....234

Трассировка программы....236
Точки останова программы....237
Добавление точки останова....237
Изменение характеристик точки останова....238
Удаление точки останова....238
Наблюдение за выводом программы....238
Наблюдение значений переменных....239

Глава 14. Введение в объектно-ориентированное программирование 240

Объектный тип и объект....240
Методы....243
Ограничение доступа к полям объекта 244
Наследование....246
Динамические объекты....249
Полиморфизм и виртуальные методы....250
Модели объектов других языков программирования....256

ЧАСТЬ II. DELPHI 257

Глава 15. Среда программирования Delphi....259

Delphi — что это?....259
Начало работы 260

Первый проект 264
Форма 264
Компоненты 269
Событие и процедура обработки события 277
Редактор кода 281
Справочная система 285
Структура проекта 286
Сохранение проекта 290
Компиляция 291
Запуск программы 294
Ошибки времени выполнения 294
Внесение изменений 296
Окончательная настройка приложения 301
Установка приложения на другой компьютер 304
Модель объекта в Delphi 304
Класс 305
Объект 305
Метод 307
Инкапсуляция и свойства объекта 307
Наследование 310
Директивы *Protected* и *Private* 311
Полиморфизм и виртуальные методы 312
Классы и объекты Delphi 314
Экзаменатор — пример программы 314
Файл теста 315
Форма приложения 318
Отображение иллюстрации 320
Выбор ответа 322
Доступ к файлу теста 323
Текст программы 324
Запуск программы 335

ПРИЛОЖЕНИЯ 337

Приложение 1. Turbo Pascal — краткий справочник 339

Зарезервированные слова и директивы 339
Структура программы 339
Основные типы данных 340
Целые числа 341
Действительные числа 341
Строки 341

Массивы 341
Записи 342
Инструкция *IF* 342
Инструкция *CASE* 343
Циклы 344
Инструкция *FOR* 344
Инструкция *REPEAT* 344
Инструкция *WHILE* 345
Объявление функции 345
Объявление процедуры 345
Процедуры и функции 346
Математические 346
Преобразования 348
Для работы со строками и символами 349
Графического режима 350
Для работы с файлами 358
Прочие 360

Приложение 2. ASCII - кодировка символов 365

Приложение 3. Представление информации в компьютере 367

Десятичные, двоичные и шестнадцатеричные числа 367
Память компьютера 370

Приложение 4. Рекомендуемая литература 372

Приложение 5. Описание CD 373

Предметный указатель 375

Предисловие

В компьютерном мире существует множество языков программирования. Программу, выполняющую одни и те же действия, можно написать на Бэйсике (BASIC), Паскале (Pascal), Си (C). Какой из языков лучше? Ответить на этот вопрос однозначно нельзя. Однако можно с уверенностью утверждать, что Pascal лучше других языков подходит для обучения программированию. И это не удивительно, ведь этот язык был разработан швейцарским ученым Никлаусом Виртом (Niklaus Wirth) специально для обучения программированию.

С момента появления Pascal за короткое время различными фирмами было создано достаточно большое количество компиляторов (компилятор — программа, переводящая инструкции языка программирования на язык команд процессора вычислительной машины). Одной из наиболее удачных стала разработка американской фирмы Borland, в которой были объединены редактор текста и высокоэффективный компилятор. Разработка получила название Turbo Pascal, а язык программирования, используемый в системе Turbo Pascal, стал называться Turbo Pascal. Следует обратить внимание, что Pascal лежит в основе используемого в среде разработки Delphi языка программирования Delphi.

Pascal — не "учебный", не "игрушечный" язык, он используется для разработки сложных, "профессиональных" программ, в том числе, предназначенных для работы в Windows.

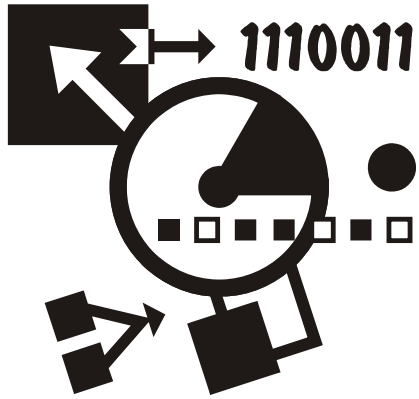
Книга, которую вы держите в руках, представляет собой учебное пособие по программированию на языке Turbo Pascal. Помимо описание языка программирования и среды разработки, в ней кратко изложены общие вопросы программирования. Материал книги несколько шире традиционного курса программирования. В ней изложен ряд тем, которые, как правило, остаются за рамками подобных учебников: обработка символьной информации, использование динамических структур, работа с файлами, отладка, вопросы объектно-ориентированного программирования.

Особое внимание в книге уделено практике составления программ и выработке навыков программирования; рассмотрены типичные ошибки, даны рекомендации по их устранению и предотвращению.

Последовательность изложения материала построена так, чтобы читатель как можно скорее приступил к самостоятельной работе на компьютере.

В качестве дополнительного материала в книге рассматриваются вопросы программирования для Windows в среде визуального программирования Delphi. Приведено описание среды, даны основные определения и термины, на конкретном примере изложена методика создания программ.

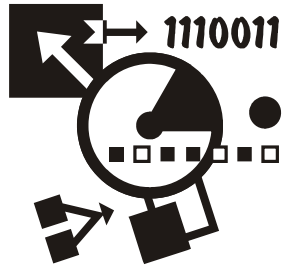
Научиться программировать можно только программируя, решая конкретные задачи. Поэтому, чтобы получить максимальную пользу от книги, вы должны работать с ней активно. Не занимайтесь просто чтением примеров. Вводите их в компьютер. Не бойтесь экспериментировать — вносите изменения в программы. Чем больше вы сделаете самостоятельно, тем большему вы научитесь.



Часть I

TURBO PASCAL

Первая часть книги посвящена программированию в Turbo Pascal. В ней приведено описание среды разработки, языка программирования, рассмотрены основные алгоритмические структуры и структуры данных, операции со строками, массивами, записями и файлами, программирование графики.



Глава 1

Среда программирования Turbo Pascal

Turbo Pascal представляет собой интегрированную среду программирования (разработки компьютерных программ) для операционной системы MS DOS. Термин "интегрированная" обозначает, что среда разработки объединяет в себе несколько элементов, а именно: редактор кода (так программисты называют редактор текста программ), компилятор и отладчик. В качестве языка программирования в Turbo Pascal используется алгоритмический язык Паскаль (Pascal).

Установка

Установка Turbo Pascal на компьютер выполняется путем запуска с установочного CD программы install.exe. В процессе установке на диске C: (по умолчанию Turbo Pascal устанавливается на этот диск) будет создан каталог TP, в подкаталоги которого будут помещены файлы, образующие Turbo Pascal.

По завершении процесса установки, перед тем как запустить Turbo Pascal первый раз, рекомендуется на диске компьютера создать два каталога: первый — для текстов программ, второй — для выполняемых файлов. Назвать эти каталоги можно, например, PAS и EXE&TPU, а разместить — в каталоге TP.

Запустить Turbo Pascal можно, набрав в окне **Запуск программы** (рис. 1.1) C:\TP\BIN\TURBO.EXE. Однако лучше создать на рабочем столе ярлык, обеспечивающий запуск Turbo Pascal.

Чтобы создать ярлык, обеспечивающий запуск Turbo Pascal, надо сделать щелчок правой кнопкой мыши на свободном месте рабочего стола, из появившегося меню выбрать команду **Создать ▸ Ярлык** и в появившемся окне **Создание ярлыка** указать имя файла, обеспечивающего запуск среды разработки (рис. 1.2). В следующем окне, которое становится доступным в результате щелчка на кнопке **Далее**, надо указать подпись ярлыка (рис. 1.3).

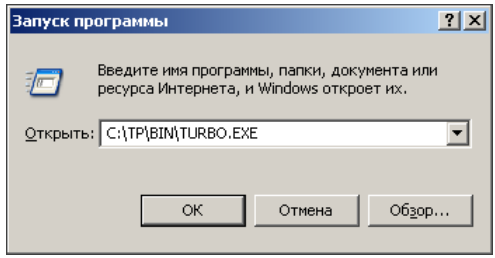


Рис. 1.1. Запуск Turbo Pascal

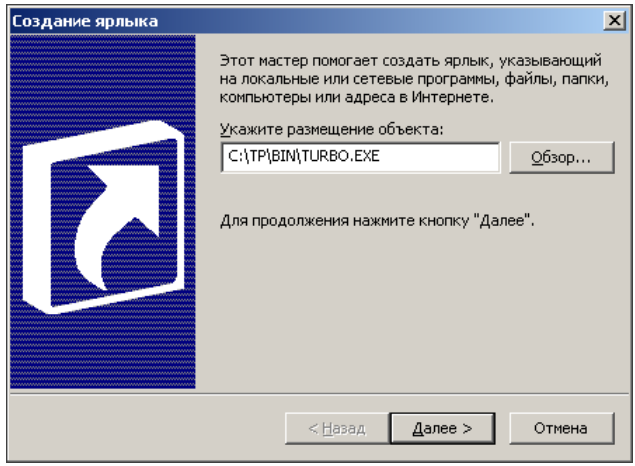


Рис. 1.2. Создание ярлыка Turbo Pascal (шаг 1)

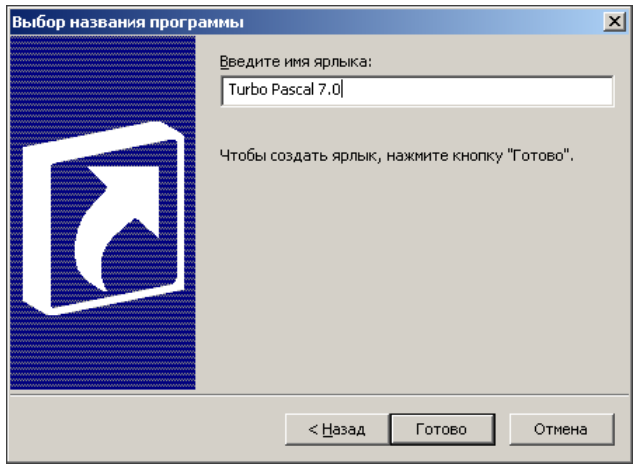


Рис. 1.3. Создание ярлыка Turbo Pascal (шаг 2)

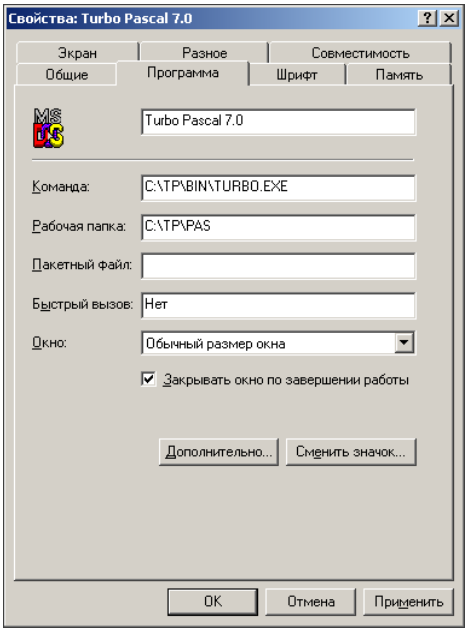


Рис. 1.4. Настройка рабочей папки

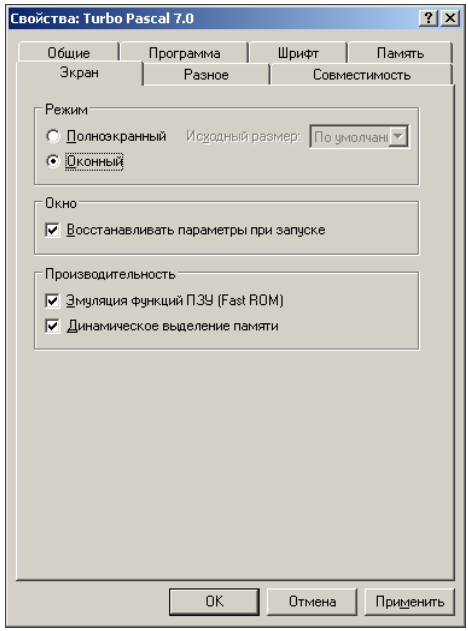


Рис. 1.5. Настройка режима окна

После того как ярлык будет создан, рекомендуется выполнить его настройку — задать рабочий, то есть используемый по умолчанию, каталог и режим экрана. Настройка ярлыка выполняется в окне **Свойства**, которое становится доступным в результате выбора в контекстном меню ярлыка команды **Свойства**. Имя рабочего каталога надо ввести в поле **Рабочая папка** вкладки **Программа** (рис. 1.4), а режим экрана (оконный) — задать на вкладке **Экран** (рис. 1.5).

Начало работы

Чтобы начать работу в Turbo Pascal, надо сделать двойной щелчок мышью на находящемся на рабочем столе ярлыке (процесс создания и настройки ярлыка подробно описан в предыдущем параграфе).

Вид окна, в котором работает Turbo Pascal, приведен на рис. 1.6. В верхней части находится строка главного меню, в нижней — строка подсказки. Основную часть окна занимает окно редактора кода (текста программы).

Шрифт, который используется для отображения текста в окне, и, как следствие, размер окна, можно изменить. Для этого надо сделать щелчок правой кнопкой мыши на значке, который находится в заголовке окна, в появившемся

меню выбрать команду **Свойства** и затем на вкладке **Шрифт** задать характеристики шрифта (например, точечный шрифт 10×18). Также можно увеличить количество строк текста, отображаемых в окне редактора (по умолчанию в окне отображается 23 строки текста). Для этого в меню **Options** надо выбрать команду **Environment ▸ Preferences** и в появившемся окне, в группе **Screen sizes** установить переключатель **43/50 lines**.

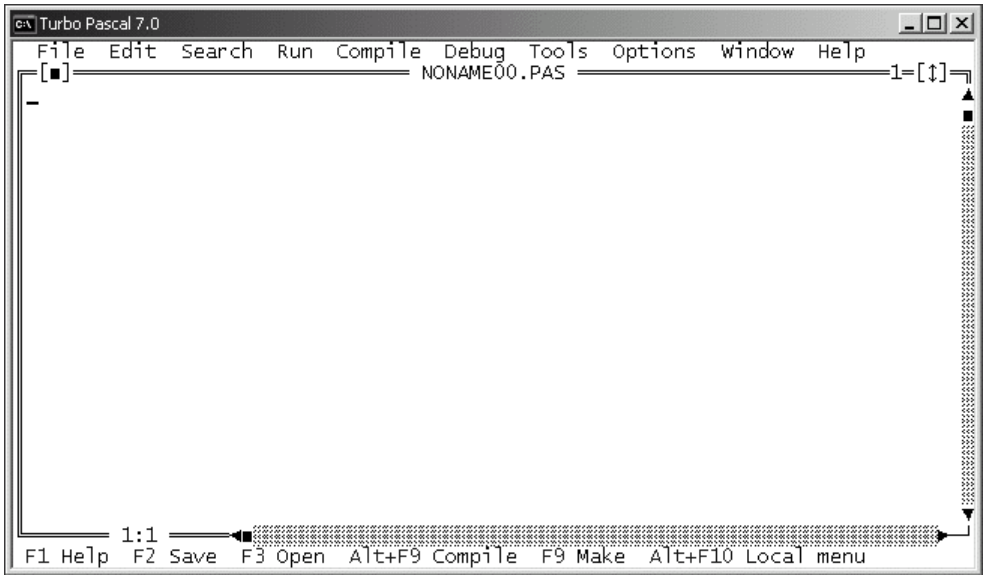


Рис. 1.6. Окно, в котором работает Turbo Pascal

Первая программа

Процесс работы в Turbo Pascal рассмотрим на примере. Создадим программу, при помощи которой можно рассчитать доход по вкладу в банке. Доход можно вычислять по формуле:

Доход = Сумма * (Процентная ставка / 365) * Срок

где: Сумма — сумма вклада; Срок — срок в клада (количество дней); Процентная ставка — процент, начисляемый на сумму вклада, при условии, что срок вклада — год; 365 — количество дней в году. На практике процентная ставка зависит от суммы вклада, чем больше сумма, тем больше процентная ставка. Будем считать, что процентная ставка равна 8%, если сумма вклада меньше 5 тыс. рублей, и 9,5%, если сумма вклада больше указанного значения.

Перед тем как приступить к непосредственной работе в Turbo Pascal на компьютере, рекомендуется разработать (составить) алгоритм решения задачи и написать программу на бумаге. Алгоритм программы (решения поставленной задачи) приведен на рис. 1.7, программа — в листинге 1.1 (текст программы принято называть листингом).

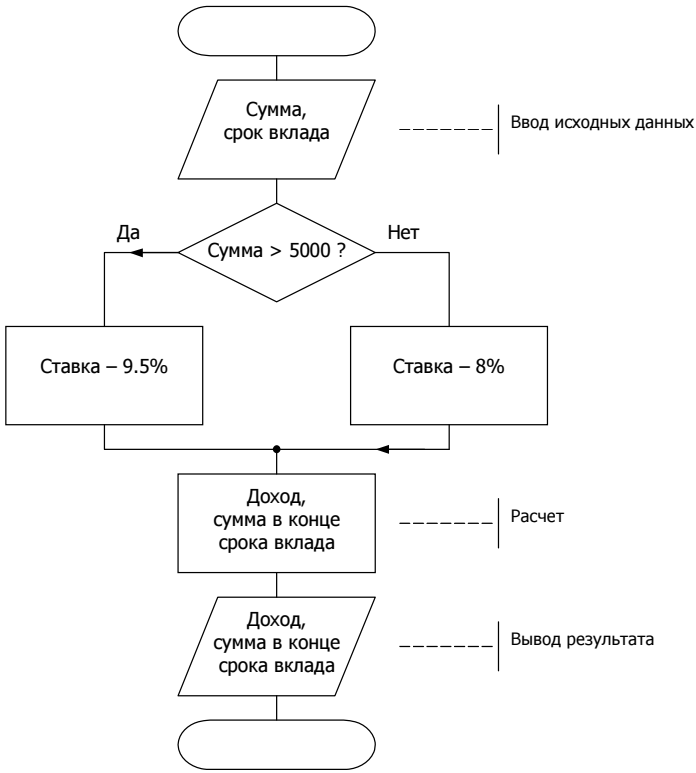


Рис. 1.7. Алгоритм программы вычисления дохода по вкладу

Листинг 1.1. Вычисление дохода по вкладу в банке (p1_1.pas)

```
{ Вычисление дохода по вкладу в банке }
program p1_1;


var
   sum: real;        { сумма вклада }
```

```
   percent: real;    { процентная ставка }
   period: integer; { срок вклада }

   profit: real;     { доход }
   result: real;     { сумма в конце срока вклада }

begin
   writeln('Вычисление дохода по вкладу в банке');
   writeln;

   { ввод исходных данных }
   write('Сумма (руб.) -> ');
   readln(sum);
   write('Срок вклада (дней) -> ');
   readln(period);

   { определить величину процентной ставки }
   if sum > 5000 then
           percent := 0.095 { ставка 9.5%}
        else
           percent := 0.08; { ставка 8%}

   profit := sum * percent/365 * period;
   result  := sum + profit;

   { вывод результата расчета }
   writeln('Сумма в конце срока вклада: ',
            result:6:2, ' руб.');
   writeln('Доход: ', profit:6:2, ' руб.');

   write('Для завершения работы программы нажмите <Enter>');
   readln;

end.
```

Первая строка программы, заключенный в фигурные скобки текст, — это комментарий. Комментарии включают в текст программы, чтобы пояснить назначение программы, переменных, ключевые точки алгоритма. Комментарии облегчают восприятие программы, позволяют понять, как она работает.

Следующая строка — это заголовок программы. В заголовке, за словом `program`, указано имя программы. Далее следует слово `var` (сокращение от `variable` — переменная), отмечающее начало раздела объявления переменных. В разделе объявления переменных перечислены (объявлены) переменные, используемые в программе. После имени переменной, через двоеточие, указан тип данных, для хранения которых предназначена переменная (`real` — дробный тип, `integer` — целый тип). Слово `begin` отмечает начало раздела выполняемых инструкций программы. Первая инструкция программы, инструкция `writeln('Вычисление дохода по вкладу в банке')`, выводит на экран текст **Вычисление дохода по вкладу в банке** и переводит курсор в начало следующей строки. Далее следуют инструкции, обеспечивающие ввод исходных данных с клавиатуры: инструкция `write('Сумма (руб.) ->')` выводит подсказку, а инструкция `readln(sum)` записывает данные, которые пользователь набрал на клавиатуре, в переменную `sum`. Аналогичным образом обеспечивается ввод значения переменной `period`. Следующие инструкции реализуют расчет. Сначала при помощи инструкции `if` определяется величина процентной ставки (значение переменной `percent`). Если значение переменной `sum` больше пяти тысяч, то переменной `percent` присваивается значение 0.095 (процентная ставка 9,5%), в противном случае, то есть если значение `sum` меньше или равно 5000, переменной `percent` присваивается значение 0.08 (процентная ставка 8%). Вычисление дохода (значения переменной `profit`) обеспечивает инструкция `profit := sum * (percent / 365) * period`, суммы в конце срока вклада — инструкция `result := sum + profit`. Далее следуют инструкции, которые выводят на экран результат расчета. Инструкция `writeln('Сумма в конце срока вклада: ', result:6:2, ' руб.')` выводит текст **Сумма в конце срока вклада:**, значение переменной `result` и слово **руб**. Инструкция `writeln('Доход: ', profit:6:2, ' руб.')` выводит значение переменной `profit`. Значения переменных `result` и `profit` отображаются с двумя цифрами после десятичной точки. В конец программы добавлена инструкция `readln`, которая приостанавливает работу программы до тех пор, пока пользователь не нажмет клавишу <Enter>. Так как инструкция `readln` является последней инструкцией программы, то после того, как пользователь нажмет <Enter>, программа завершит работу и окно, в котором работала программа, закроется (исчезнет с экрана).

Набор текста программы

По умолчанию при запуске среды разработки автоматически открывается окно редактора кода (текста программы). Поэтому сразу после запуска Turbo Pascal можно набирать текст программы.

Программу в окне редактора кода набирают обычным образом.

Примечание

Переключение на русский алфавит осуществляется путем нажатия правой клавиши <Shift> (при нажатой клавише <Ctrl>), на латинский — левой клавиши <Shift>. Следует обратить внимание, что комбинация <Ctrl> + "правый" <Shift> для переключения на русский алфавит работает только в том случае, если в настройках операционной системы в качестве языка ввода по умолчанию задан русский язык.

Редактор кода автоматически выделяет цветом ключевые слова языка программирования (`program`, `var`, `begin`, `end` и другие) и комментарии, что облегчает восприятие структуры программы.

В процессе набора текста необходимо соблюдать правила хорошего стиля программирования — записывать инструкции с отступами и добавлять в текст комментарии. В качестве примера на рис. 1.8 приведено окно редактора кода, в котором находится текст программы вычисления дохода по вкладу.

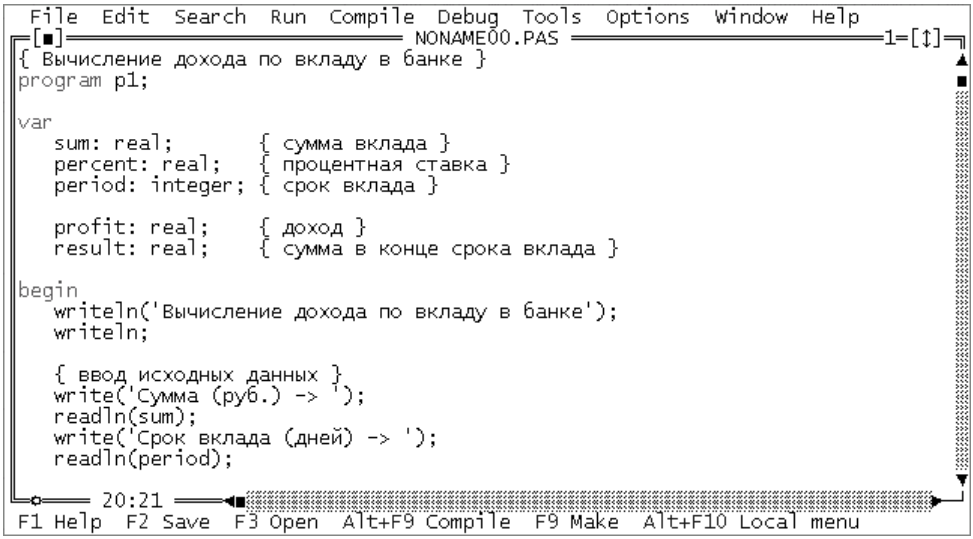


Рис. 1.8. Текст программы в окне редактора кода

После того как текст программы будет набран, его надо сохранить на диске. Для этого в меню **File** надо выбрать команду **Save**, в поле **Save file as** появившегося окна (рис. 1.9) ввести имя файла и сделать щелчок на кнопке **OK**. Следует обратить внимание, что в нижней части окна отображается имя каталога, в который будет помещен файл программы (если программист в поле **Save file as** явно не укажет диск и каталог). Расширение имени файла (PAS) можно не указывать, оно будет добавлено к имени автоматически.

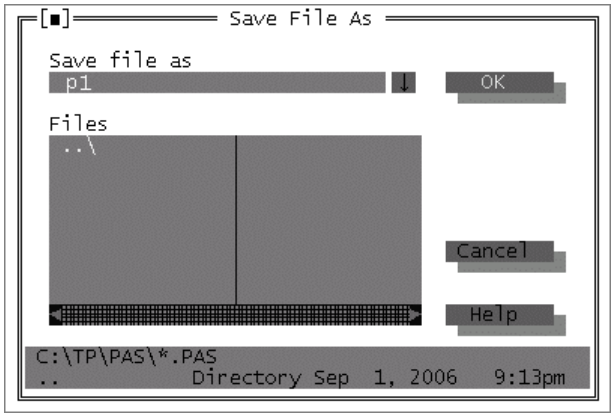


Рис. 1.9. Сохранение программы

Примечание

В Turbo Pascal при записи имен файлов разрешается использовать *только* буквы латинского алфавита и цифры (буквы русского алфавита и пробел использовать нельзя). Кроме того, количество символов в имени файла (без учета точки и расширения PAS) не должно превышать восьми.

Компиляция

Программа, представленная на языке программирования (набранная программистом в окне редактора кода), называется исходной. Это *текст*, понятный человеку. Для того чтобы программа могла быть выполнена процессором, ее необходимо преобразовать в выполняемую программу — последовательность машинных команд. Процесс преобразования исходной программы в выполняемую называется компиляцией.

Чтобы активизировать процесс компиляции, нужно в меню **Compile** выбрать команду **Compile**. В случае, если в программе нет синтаксических ошибок (то есть все инструкции набраны правильно), на экране появляется окно, информирующее об успешном завершении компиляции (рис. 1.10).

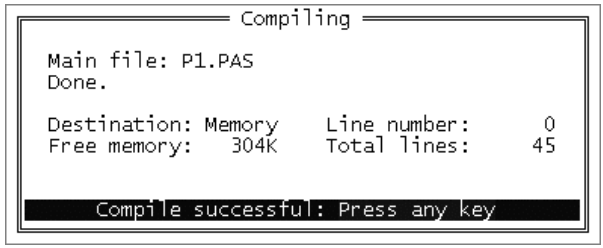


Рис. 1.10. Сообщение об успешном завершении компиляции

Turbo Pascal поддерживает два режима компиляции: "в память" (Memory) и "на диск" (Disk). В режиме Memory создаваемая компилятором выполняемая программа записывается в оперативную память компьютера, в режиме Disk — в exe-файл. Режим компиляции отображается в окне **Compiling** (рис. 1.10) и в строке **Destination** меню **Compile**. По умолчанию компиляция выполняется в память, поэтому запустить откомпилированную программу можно только из среды разработки. Чтобы программу можно было запустить из операционной системы, надо выполнить ее компиляцию на диск (в режиме Disk). Для этого необходимо изменить режим компиляции с Memory на Disk — в меню **Compile** выбрать команду **Destination** (рис. 1.11). Следует обратить внимание, что при компиляции "на диск" exe-файл будет помещен в каталог, имя которого указано в поле **EXE&TPU directory** окна **Directories** (это окно становится доступным в результате выбора в меню **Options** команды **Directories**) или, если имя каталога в этом поле не задано, в каталог, в котором находится компилируемый PAS-файл.

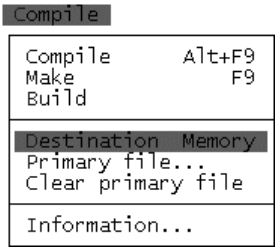


Рис. 1.11. Чтобы изменить режим компиляции, надо в меню **Compile** выбрать команду **Destination**

Ошибки времени компиляции

Очевидно, что в программе могут быть ошибки. Различают синтаксические и алгоритмические ошибки. Синтаксические ошибки — это ошибки записи инструкций программы. Алгоритмические ошибки — это ошибки, связанные с нарушением логики работы программы.

В процессе компиляции текст программы проверяется на отсутствие синтаксических ошибок. Компилятор просматривает программу от начала. Обнаружив ошибку, компилятор выводит сообщение об ошибке (код ошибки и пояснение) и устанавливает курсор в ту точку текста программы, в которой, скорее всего, она находится. В качестве примера на рис. 1.12 приведен результат компиляции программы, в которой есть ошибка. В приведенном

примере ошибка заключается в том, что имя переменной записано неверно: в программе объявлена переменная `sum` (см. листинг), а в инструкции `readln` указано `summ`.

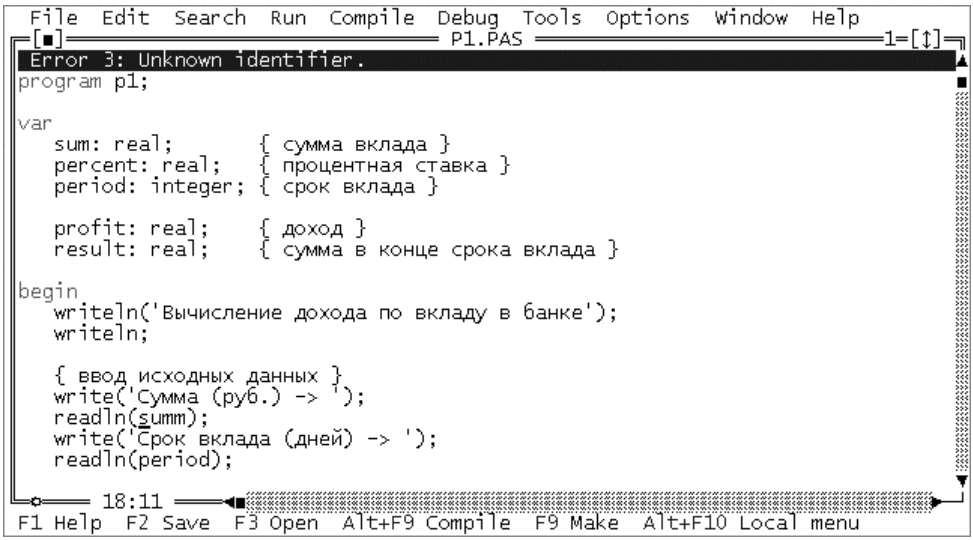


Рис. 1.12. Пример сообщения об ошибке

После анализа причины ошибки и ее устранения (в приведенном примере надо удалить лишнюю букву m) надо снова активизировать процесс компиляции. Таким образом, последовательно исправляя ошибки, обнаруживаемые компилятором, можно устранить все синтаксические ошибки.

В табл. 1.1 приведены сообщения о наиболее типичных ошибках.

Таблица 1.1. *Сообщения компилятора о типичных ошибках*

Сообщение компилятора	**Вероятная причина**
3: Unknown identifier (Неизвестный идентификатор)	Используется переменная, не объявленная в разделе `var` программы; Ошибка записи имени переменной. Например, в разделе `var` объявлена переменная `sum`, а в тексте программы написано `suma`
26: Type mismatch (Несоответствие типов)	В инструкции присваивания тип выражения не соответствует типу переменной, которой присваивается значение выражения

Таблица 1.1 *(окончание)*

Сообщение компилятора	Вероятная причина
85: ";" expected (Ожидается символ "точка с запятой")	Не поставлен символ "точка с запятой" после инструкции
113: Error in statement (Ошибка в выражении)	Неверный синтаксис оператора; например, в инструкции `if` поставлен символ "точка с запятой" после инструкции, следующей за словом `then` (фактически перед `else`)

Запуск программы

Если компиляция программы завершена успешно, то программу, текст которой находится в окне редактора кода, можно запустить. Для этого в меню **Run** надо выбрать команду **Run**. В результате запуска программы становится доступным окно прикладной программы. В это окно программа выводит сообщения, и из этого окна она получает от пользователя данные. На рис. 1.13 в качестве примера приведено окно, в котором работает программа вычисления дохода по вкладу.

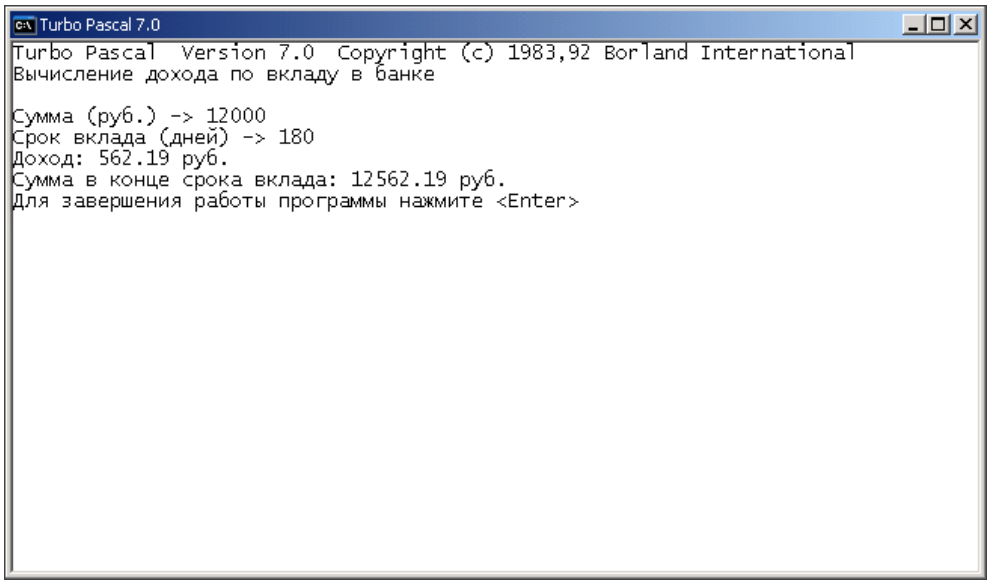


Рис. 1.13. Окно, в котором работает программа, запущенная из Turbo Pascal

По завершении работы программы окно, в котором она работала, автоматически закрывается и вновь становится доступным окно Turbo Pascal (увидеть окно, в котором работала прикладная программа, можно, выбрав в меню **Debug** команду **User Screen** или нажав <Alt> + <F5>).

> ***СОВЕТ***
>
> Чтобы окно, в котором работает программа, не исчезало с экрана сразу после завершения работы программы и, чтобы пользователь смог спокойно оценить результат ее работы, добавьте в конец программы инструкции: `writeln('Программа завершила работу. Нажмите <Enter>'); readln;`

Ошибки времени выполнения

Во время работы программы возможны так называемые ошибки времени выполнения (runtime error). В большинстве случаев причинами ошибок во время работы программы являются неверные данные.

При возникновении ошибки работа программы завершается. Если программа запущена из Turbo Pascal, то в окно редактора кода выводится сообщение об ошибке и курсор устанавливается в ту строку текста программы, в которой находится инструкция, при выполнении которой возникла ошибка. В качестве примера на рис. 1.14 приведено окно Turbo Pascal после возникновения ошибки во время работы программы вычисления дохода по вкладу в результате ввода пользователем неверных данных — строки `10000,00` (1000 рублей 00 копеек). Курсор в окне редактора кода находится перед инструкцией `readln(sum)`. Это показывает, что ошибка произошла во время выполнения именно этой инструкции. Сообщение `Error 106: Invalid numeric format` (неверный формат числа) информирует о том, что причина ошибки — неверные данные (программа ожидает ввода дробного числа, а пользователь ввел строку, которая дробным числом не является).

В табл. 1.2 приведены типичные ошибки времени выполнения программы.

Таблица 1.2. *Типичные ошибки времени выполнения программы*

Сообщение	Ошибка	Вероятная причина
Error 106: Invalid numeric format	Неверный формат числа	Программа ожидает ввода целого числа, а пользователь ввел дробное число При вводе дробного числа в качестве разделителя целой и дробной частей числа пользователь вместо точки ввел запятую
Error 200: Division by zero	Деление на ноль	Второй операнд (делитель) оператора деления равен нулю

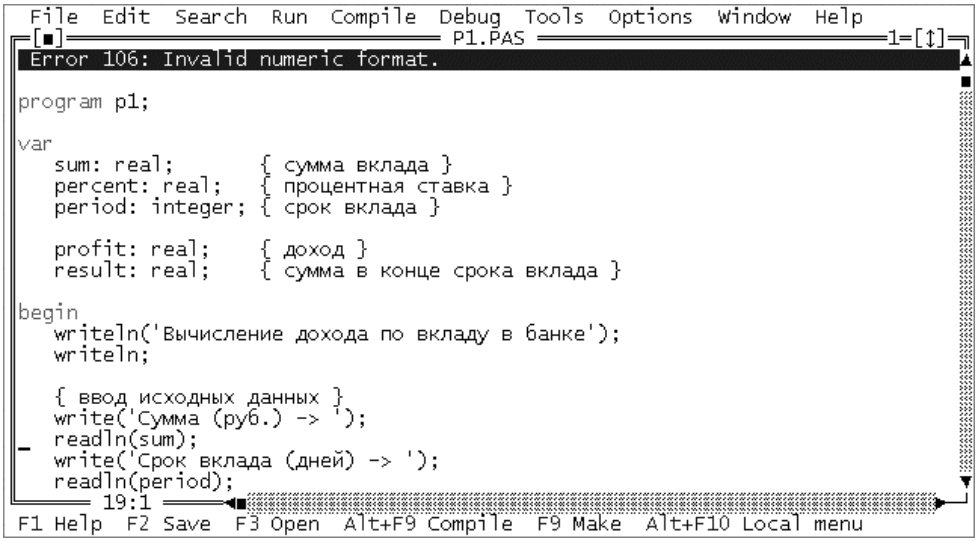


Рис. 1.14. Курсор показывает инструкцию, при выполнении которой произошла ошибка

Создание exe-файла

Чтобы иметь возможность запустить программу из операционной системы, нужно создать исполняемый (exe) файл программы — установить режим компиляции "на диск" (см. *параграф "Компиляция"* этой главы) и выполнить повторную компиляцию программы (в меню **Compile** выбрать команду **Compile**). Созданный компилятором exe-файл будет помещен в каталог, имя которого указано в поле **EXE&TPU Directory** окна **Directories** (это окно становится доступным в результате выбора в меню **Options** команды **Directories**), или, если каталог не задан, в каталог, в котором находится компилируемый pas-файл.

Завершение работы с Turbo Pascal

Чтобы завершить работу с Turbo Pascal, надо в меню **File** выбрать команду **Exit**. Если программа еще не была сохранена на диске или с момента сохранения в нее были внесены изменения, то на экране появится сообщение и вопрос о необходимости сохранить изменения (рис. 1.15). Чтобы сохранить изменения, надо сделать щелчок на кнопке **Yes**. Щелчок на кнопке **Cancel**

отменяет команду завершения работы с Turbo Pascal (в результате вновь становится доступным окно редактора кода).

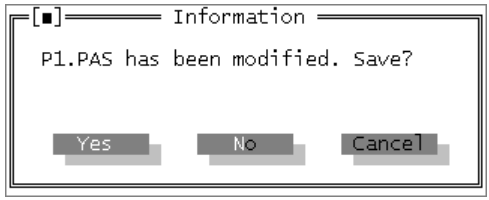


Рис. 1.15. Чтобы сохранить изменения, надо сделать щелчок на кнопке **Yes**

Внесение изменений в программу

Чтобы изменить программу, ее надо загрузить в редактор кода, внести необходимые изменения и выполнить компиляцию (предварительно установив режим компиляции на диск). Загрузка текста программы в редактор кода выполняется следующим образом. Сначала в меню **File** надо выбрать команду **Open**. В появившемся окне **Open a File** следует выбрать программу (PAS-файл программы), которую надо изменить, и щелкнуть на кнопке **Open** (рис. 1.16). Следует обратить внимание, что в списке **Files** окна **Open a File** отображаются имена PAS-файлов, которые находятся в *рабочем* каталоге (имя рабочего каталога отображается в поле **Рабочая папка** вкладки **Программа** окна свойств ярлыка, используемого для запуска Turbo Pascal).

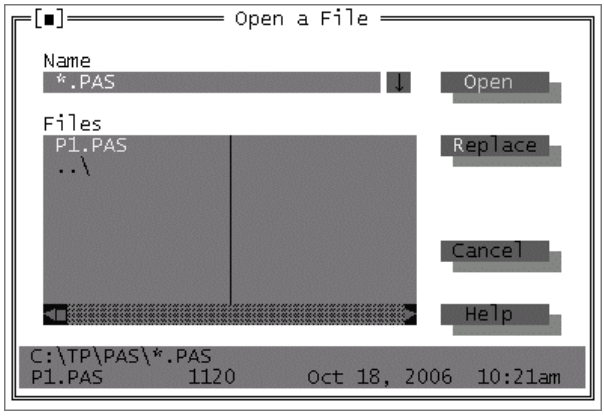


Рис. 1.16. Окно **Open a File**

Запуск программы из операционной системы

Чтобы запустить созданную в Turbo Pascal программу из операционной системы, надо раскрыть окно **Запуск программы** и в поле **Открыть** ввести полное (то есть указать путь) имя exe-файла программы (рис. 1.17).

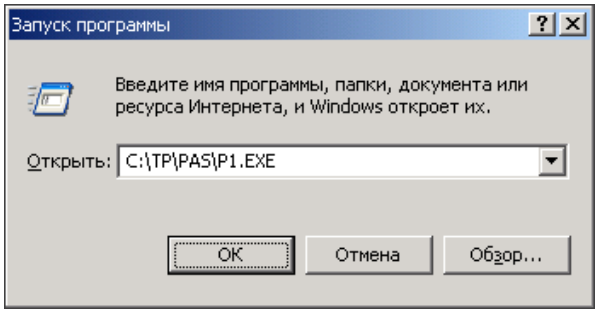
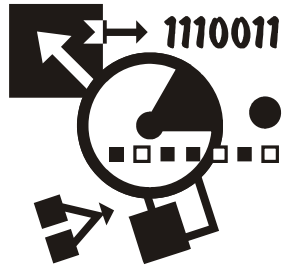


Рис. 1.17. Запуск программы Доход по вкладу из Windows

Задание

1. Установите Turbo Pascal на компьютер.
2. В каталоге C:\TP создайте каталог PAS.
3. На рабочем столе создайте ярлык, обеспечивающий запуск Turbo Pascal, и выполните его настройку: задайте, что рабочим каталогом является каталог C:\TP\PAS.
4. Запустите Turbo Pascal, наберите программу вычисления дохода по вкладу в банке (см. листинг 1.1), сохраните программу в каталоге C:\TP\PAS.
5. Выполните компиляцию программы.
6. Запустите программу вычисления дохода по вкладу, убедитесь, что она работает правильно (введите: сумма — 1000, срок вклада — 365); результат должен быть: доход: 80 руб., сумма в конце срока вклада: 1080 руб.).
7. Завершите работу с Turbo Pascal.
8. * Внимательно изучите листинг программы вычисления дохода по вкладу и "по аналогии" напишите программу пересчета цены из долларов в рубли.



Глава 2

Введение в программирование

Программа, работающая на компьютере, часто отождествляется с самим компьютером, так как человек, использующий программу, "вводит в компьютер" исходные данные с клавиатуры, и "компьютер выдает результат" на экран. На самом деле преобразование исходных данных в результат выполняет процессор.

Процессор выполняет обработку данных в соответствии с программой — последовательностью команд, составленной программистом. Таким образом, чтобы компьютер выполнил некоторую работу, необходимо разработать последовательность команд, обеспечивающую выполнение этой работы, или, как говорят, написать программу.

Этапы разработки программы

Выражение *написать программу* отражает только один из этапов создания компьютерной программы, когда разработчик программы (программист) действительно пишет команды (инструкции) на бумаге или в редакторе текста.

Программирование — это процесс создания программы, который можно представить как последовательность следующих этапов:

- ❒ Определение требований к программе (спецификация)
- ❒ Разработка алгоритма
- ❒ Написание команд (кодирование)
- ❒ Отладка
- ❒ Тестирование

Определение требований к программе

Определение требований к программе — один из важнейших этапов. На этом этапе подробно описывается исходная информация и формулируются требования к результату. Кроме того, описывается поведение программы в особых случаях.

Разработка алгоритма

На этапе разработки алгоритма необходимо определить последовательность действий, которые надо выполнить для достижения результата. Многие задачи можно решить различными способами. В этом случае программист, используя некоторый критерий, должен выбрать лучший. Результатом этапа разработки алгоритма должен быть алгоритм, представленный в виде словесного описания или блок-схемы.

Кодирование

После того как будет разработан алгоритм решения задачи, можно приступить к кодированию — записи алгоритма на языке программирования. Результатом этого этапа является исходная, т. е. представленная на языке программирования, программа.

Отладка

Отладка — процесс проверки работоспособности программы, поиска и устранения ошибок. Ошибки бывают двух типов: синтаксические (ошибки в тексте) и алгоритмические. Синтаксические ошибки — это наиболее легко устраняемые ошибки. Алгоритмические ошибки обнаружить труднее. Этап отладки можно считать завершенным, если программа правильно работает на одном-двух наборах исходных данных, если результат, полученный при использовании программы, совпадает с результатом, полученным методом "ручного" счета.

Тестирование

Этап тестирования особенно важен, если вы предполагаете, что вашей программой будут пользоваться другие. На этом этапе следует проверить, как ведет себя программа на как можно большем количестве входных наборов данных, в том числе и на заведомо неверных. Например, следует проверить, как ведет себя программа вычисления дохода по вкладу, если сумма вклада меньше, равна или больше пяти тысяч.

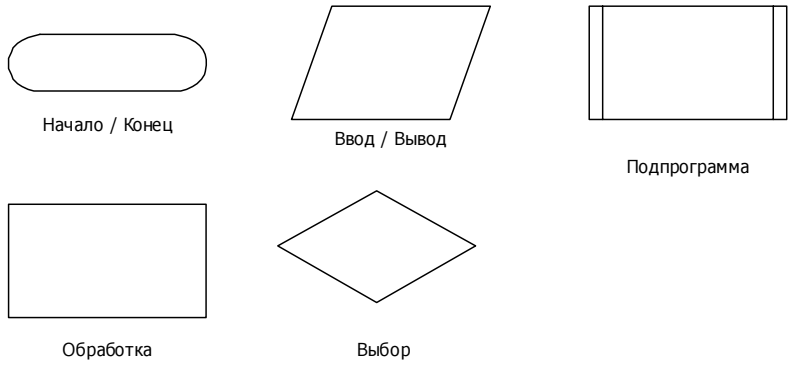


Рис. 2.1. Основные элементы для изображения блок-схем алгоритмов

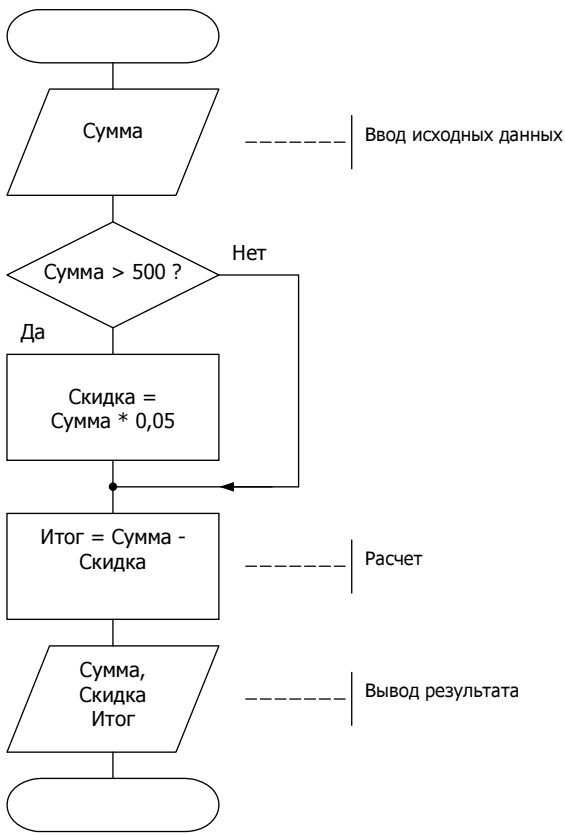


Рис. 2.2. Блок-схема алгоритма вычисления суммы покупки с учетом скидки

Алгоритм

На одном из первых этапов создания программы программист должен определить последовательность действий, которые необходимо выполнить, чтобы решить поставленную задачу, или разработать *алгоритм* — точное предписание, которое определяет процесс перехода от исходных данных к результату.

Существует несколько способов представления алгоритмов. На практике наиболее широко используют графический способ представления алгоритмов — блок-схемы. В блок-схемах для обозначения логически различных элементов программы используются стандартные символы, некоторые из них приведены на рис. 2.1.

Представление алгоритма в виде блок-схемы позволяет наглядно отразить последовательность действий, которые надо выполнить для решения поставленной задачи.

В качестве примера на рис. 2.2 приведен алгоритм вычисления суммы покупки с учетом скидки.

Программа

После разработки алгоритма решения задачи можно перейти к *кодированию* — составлению программы.

Программа — это последовательность инструкций (иногда вместо термина инструкция используют термины "оператор" или "команда"), записанных в соответствии с правилами языка программирования. В качестве примера в листинге 2.1 (текст программы принято называть листингом) приведена программа вычисления суммы покупки с учетом скидки.

Листинг 2.1. Программа вычисления суммы покупки с учетом скидки (p2_1.pas)

```
program p2_1;
var
   sum: real;       { сумма покупки}
   discount: real;  { скидка}
   total: real;     { к оплате }
begin
   writeln('*** Сумма покупки с учетом скидки ***');
   write('Сумма покупки (руб.) ->');
   readln(sum);
```

```
   if sum >= 500
      then discount := sum * 0.05;
   total := sum - discount;

   writeln;
   writeln('------------------------------');
   writeln('Сумма покупки: ',sum:6:2,' руб.');
   writeln('Скидка: ',discount:6:2,' руб.');
   writeln('К оплате: ',total:6:2,' руб.');

   writeln('Для завершения работы программы нажмите <Enter>');
   readln;
end.
```

Компиляция

Программа, представленная на языке программирования, называется исходной. Она состоит из инструкций, понятных человеку, но не понятных процессору. Чтобы процессор смог выполнить работу в соответствии с инструкциями исходной программы, исходная программа должна быть переведена на машинный язык — язык команд процессора. Решение этой задачи обеспечивает специальная программа — компилятор. Схема работы компилятора приведена на рис. 2.3. В процессе работы компилятор сначала проверяет программу на отсутствие синтаксических ошибок, затем, если в программе нет ошибок, создает (генерирует) выполняемую программу — машинный код. Следует отметить, что генерация машинного кода свидетельствует только об отсутствии в тексте программы синтаксических ошибок. Убедиться в правильности работы программы можно только во время ее тестирования — пробных запусков программы и анализа полученных результатов.

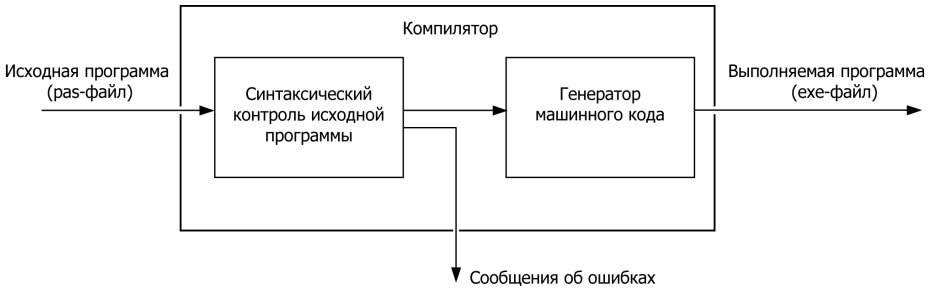


Рис. 2.3. Схема работы компилятора

Тип данных

Программа может оперировать с данными различного типа: целыми и вещественными (дробными) числами, символами, строками символов и логическими величинами.

Целый тип

В языке Pascal определены пять целых типов: `shortint`, `integer`, `longit`, `byte` и `word` (табл. 2.1). На практике наиболее широко используется тип `integer`.

Таблица 2.1. *Целые типы*

Тип	Диапазон	Размер (в байтах)
`shortint`	-128 .. 127	1
`integer`	-32768 .. 32767	2
`longint`	-2147483648 .. 2 147483647	4
`byte`	0 .. 255	1
`word`	0 .. 65535	2

Вещественный тип

В языке Pascal определены четыре вещественных типа: `real`, `single`, `double` и `extended` (табл. 2.2). Самым универсальным является тип `real`.

Таблица 2.2. *Вещественные типы*

Тип	Диапазон	Точность	Размер (в байтах)
`real`	$2.9 \cdot 10^{-39}$ ... $1.7 \cdot 10^{38}$	11-12	6
`single`	$1.5 \cdot 10^{-45}$ ... $3.4 \cdot 10^{38}$	7-8	4
`double`	$5.0 \cdot 10^{-324}$ ... $1.7 \cdot 10^{308}$	15-16	8
`extended`	$3.4 \cdot 10^{-4932}$ ... $1.1 \cdot 10^{4932}$	19-20	10

Символьный тип

Символьный тип (`char`) — это совокупность *символов* — букв русского и латинского алфавитов, цифр и других знаков.

Строковый тип

Строковый тип (`string`) — совокупность всевозможных строк, состоящих не более чем из 255 символов.

Логический тип

Логический (`boolean`) тип — это совокупность двух логических значений: `True` (истина) и `False` (ложь).

Переменная

Переменная — это область памяти компьютера, в которой находятся данные: исходные, промежуточные и результат.

Объявление переменной

Для того чтобы программа могла манипулировать данными, программист должен зарезервировать для этих данных место в оперативной памяти компьютера. Резервирование места для данных (исходных, промежуточных, результата) выполняется путем *объявления* переменных.

Инструкция объявления переменной в общем виде выглядит так:

Переменная`: `*тип*`;`

где:

Переменная — имя переменной, которое используется для доступа к данным;

тип — тип данных, хранение которых обеспечивает переменная.

Примеры:

```
n: integer;
profit: real;
FirstName: string;
found: boolean;
```

Если в программе несколько переменных одного типа, то объявить их можно в одной инструкции, разделив имена переменных запятыми.

Пример:

```
profit, percent, total: real;
```

Объявляя переменную, программист должен придумать для нее имя (идентификатор). В языке Pascal в качестве имени переменной можно использовать последовательность символов, состоящую из букв латинского алфавита

и цифр, а также символов подчеркивания (первым символом должна быть буква).

Желательно, чтобы имя переменной несло смысловую нагрузку, отражало ее назначение. Например, в программе пересчета цены из долларов в рубли переменным можно присвоить имена `usd` (цена в долларах), `kurs` (курс) и `rub` (цена в рублях). Использование несущих смысловую нагрузку имен переменных облегчает восприятие программы, уменьшает вероятность появления ошибок. Следует обратить внимание, что компилятор языка Pascal не различает прописные и строчные буквы, поэтому имена `SUM`, `Sum` и `sum` обозначают одну и ту же переменную.

Константы

В Turbo Pascal существует два вида констант: *обычные* и *именованные*.

Обычная константа — это целое или дробное число, строка символов или отдельный символ, логическое значение.

Константы используются для инициализации переменных (присваивания начального значения) и в качестве операндов выражений.

Числовые константы

В тексте программы числовые константы записываются обычным образом, т. е. так, как, например, при решении математических задач. При записи дробных чисел для разделения целой и дробных частей используется точка. Если константа отрицательная, то непосредственно перед первой цифрой ставится знак "минус".

Ниже приведены примеры числовых констант:

```
123
0.005
-524.03
```

Дробные константы могут изображаться в виде числа с плавающей точкой. Представление в виде числа с плавающей точкой основано на том, что любое число может быть записано в алгебраической форме как произведение числа, меньшего 10, которое называется мантиссой, и степени десятки, именуемой порядком.

В табл. 2.3 приведены примеры чисел, записанных в обычной форме, в алгебраической форме и форме с плавающей точкой.

Таблица 2.3. *Примеры записи дробных чисел*

Число	Алгебраическая форма	Форма с плавающей точкой
1000000	$1 \cdot 10^{6}$	1,0000000000E+06
–123.452	$-1{,}23452 \cdot 10^{2}$	–1,2345200000E+02
0,0056712	$5{,}6712 \cdot 10^{-3}$	5,6712000000E-03

Строковые и символьные константы

Строковые и символьные константы заключаются в кавычки. Ниже приведены примеры строковых и символьной констант:

```
'Turbo Pascal'
'******************'
'Д'
```

Логические константы

Логическое высказывание (выражение) может быть либо истинно, либо ложно. Истине соответствует константа `True`, значению "ложь" — константа `False`.

Именованная константа

Именованная константа — это имя (идентификатор), которое в программе используется вместо самой константы.

Именованная константа, как и переменная, перед использованием должна быть объявлена. В общем виде инструкция объявления именованной константы выглядит следующим образом:

константа = *значение*;

где:

константа — имя константы;

значение — значение константы.

Именованные константы объявляются в программе в разделе объявления констант, который начинается словом `const`. Ниже приведен пример объявления именованных констант (целой, строковой и дробной):

```
const
    HB = 10;
```

```
Title = 'Скорость бега';
NDS = 0.18;
```

После объявления именованной константы вместо самой константы можно использовать ее имя.

В отличие от переменной, при объявлении константы тип явно не указывают. Тип константы определяется ее видом, например:

`10` — константа целого типа;

`0.25` — константа вещественного типа;

`'Borland'` — строковая константа;

`'*'` — символьная константа.

Инструкция присваивания

Инструкция присваивания позволяет изменить значение переменной, в том числе вычислить значение переменной по формуле. В результате выполнения инструкции присваивания значение переменной меняется, ей *присваивается* новое значение.

В общем виде инструкция присваивания выглядит так:

```
Имя := Выражение;
```

где:

Имя — имя переменной, значение которой надо изменить;

двоеточие и следующий за ним знак "равно" — символ "присвоить";

Выражение — выражение, значение которого присваивается переменной, имя которой указано слева от символа операции "присвоить".

Примеры:

```
sum := cena * kol;
rub :=  usd * k;
skidka := sum * 0.05;
percent := 15;
right := right +1;
```

Выражение

Выражение состоит из *операндов* и *операторов*. Операторы (табл. 2.4) обозначают действия, операнды — объекты, над которыми эти действия выполняются. В качестве операндов выражения может выступать константа, переменная функция или другое выражение.

Примеры выражений:

```
sum + profit
sum * 0.75
k - 1
```

Следует обратить внимание, что переменная и константа — это простейшие выражения.

Таблица 2.4. *Операторы*

Оператор	Действие
+	Сложение
-	Вычитание
*	Умножение
/	Деление
div	Деление нацело
mod	Вычисление остатка от деления

Результат выполнения операторов `+`, `-`, `*` и `/` очевиден.

Значением выражения `a div b` является целая часть результата деления `a` на `b`, а выражения `a mod b` — остаток от деления, представленный как целое. Например, значение выражения `125 div 100` равно 1, а выражения `125 mod 100` равно `25`. Следует обратить внимание, что операнды операторов `div` и `mod` должны быть целого типа.

Операторы имеют разный приоритет. Так операторы `*`, `/`, `div`, `mod` имеют более высокий приоритет, чем операторы `+` и `-`. Другими словами, сначала выполняется умножение и деление, затем — сложение и вычитание.

Приоритет операторов влияет на порядок их выполнения. При вычислении значения выражения сначала выполняются операторы с более высоким приоритетом. Если приоритет операторов в выражении одинаковый, то сначала выполняется тот оператор, который находится левее.

Для задания нужного порядка выполнения операций можно использовать скобки. Например:

```
(x2 - x1)/dx
```

Выражение, заключенное в скобки, трактуется как один операнд. Это значит, что операции, стоящие в скобках, будут выполняться в обычном порядке, но раньше, чем операции, находящиеся за скобками. При записи выражений,

содержащих скобки, должна соблюдаться парность скобок, т. е. число открывающих скобок должно быть равно числу закрывающих скобок. Нарушение парности скобок — наиболее распространенная ошибка при записи выражений.

Тип выражения

Тип выражения определяется типом операндов, входящих в выражение, и зависит от операций, выполняемых над ними.

Тип переменной указывается в инструкции ее объявления. Тип константы можно определить по ее виду. Например, константы 0, 1 и -512 целого типа (`integer`), а константы 1.0, 0.0, 0.25 — вещественного типа (`real`).

В табл. 2.5 приведены правила, при помощи которых можно определить тип выражения.

Таблица 2.5. *Типы выражений*

Оператор	Тип выражения
`*`, `+`, `-`	Если хотя бы один из операндов вещественного типа (`real`), то тип выражения — вещественный (`real`); если оба операнда целого типа (`integer`), то тип выражения — целый (`integer`)
`/`	Вне зависимости от типа операндов тип выражения — вещественный (`real`)
`div`, `mod`	Тип выражения — целый (`integer`)

Выполнение инструкции присваивания

Инструкция присваивания выполняется следующим образом: сначала вычисляется значение выражения, указанного справа от символа присваивания, затем вычисленное значение записывается в переменную, имя которой стоит слева от символа присваивания.

Например, в результате выполнения инструкций:

`i := 0;` — значение переменной `i` становится равным нулю;

`a := b + c;` — значением переменной `a` будет число, равное сумме значений переменных `b` и `c`;

`j := j + 1;` — значение переменной `j` увеличивается на единицу.

Инструкция присваивания считается верной, если тип выражения соответствует или может быть приведен к типу переменной. Переменной типа `real`

можно присвоить значение выражения типа `real` или `integer`, а переменной типа `integer` — только типа `integer`. Например, если переменные `k` и `n` целого типа (`integer`), а `s` — вещественного (`real`), то инструкция

```
n := k/10;
```

неправильная, а инструкция

```
s := k;
```

правильная.

Во время перевода исходной программы в выполняемую компилятор проверяет соответствие типов выражений и переменных. Если тип выражения не соответствует или не совместим с типом переменной, то компилятор выдает сообщение об ошибке:

Error 26: Type mismatch (типы не совместимы)

Функции

Функция — это подпрограмма, которая выполняет некоторую работу (вычисление). Например, функция `Sqrt` вычисляет квадратный корень, а функция `Sin` — синус.

Для того чтобы воспользоваться функцией, ее имя надо указать в качестве операнда выражения инструкции присваивания. Например, в результате выполнения инструкции `k := Sqrt(n)` в переменную `k` будет записано значение квадратного корня числа, которое находится в переменной `n`.

В табл. 2.6 приведены некоторые полезные функции.

Таблица 2.6. *Некоторые полезные функции*

Функция	Значение функции
`Abs(n)`	Абсолютное значение n
`Sqrt(n)`	Квадратный корень из n
`Sqr(n)`	Квадрат n
`Sin(n)`	Синус n
`Cos(n)`	Косинус n
`Exp(n)`	Экспонента n
`Ln(n)`	Натуральный логарифм n
`Rardom(n)`	Случайное целое число от 0 до n-1

Ввод и вывод

Обычно исходные данные вводятся с клавиатуры, а результат отображается на экране монитора. Ввод исходных данных обеспечивают инструкции `read` и `readln`, отображение результата — `write` и `writeln`.

Инструкции *WRITE* и *WRITELN*

Инструкции `write` и `writeln` предназначены для вывода на экран монитора сообщений и значений переменных.

Вывод сообщений

В общем виде инструкция вывода сообщения записывается так:

`write(`*Сообщение*`);`

где *Сообщение* — текст (строковая константа), который надо вывести на экран монитора.

Пример:

```
write('Turbo Pascal');
```

Положение текста на экране определяется текущим положением курсора в окне программы. В начале работы программы курсор находится в начале первой строки окна программы, а после вывода сообщения — за последним символом сообщения. Таким образом, следующая инструкция `write` выведет сообщение сразу за текстом, выведенным предыдущей инструкцией `write`. Например, в результате выполнения инструкций:

```
write('Borland ');
write('Turbo ');
write('Pascal');
```

на экране появится строка:

```
Borland Turbo Pascal
```

Чтобы после вывода сообщения курсор переместился в начало следующей строки, вместо инструкции `write` следует использовать инструкцию `writeln`. Например, в результате выполнения инструкций:

```
writeln('Borland');
writeln('Turbo');
writeln('Pascal');
```

на экране появятся три строки текста.

Иногда возникает необходимость разделить строки сообщения. Сделать это можно, вставив инструкцию `writeln` между инструкциями вывода строк сообщения. Например:

```
writeln('Borland');
writeln;
writeln('Turbo Pascal');
```

Вывод значений переменных

Инструкция вывода значения переменной в общем виде выглядит так:

```
write(Переменная);
```

где *Переменная* — имя переменной, значение которой надо вывести на экран монитора.

Пример:

```
write(profit);
```

По умолчанию значения переменных вещественного (`real`) типа отображаются в формате с плавающей точкой. Например, если значение переменной `sum` равно 15275.5, то в результате выполнения инструкции `write(sum)` в окне программы появится строка:

```
1.5275500000E+04
```

т. е. $1.52755 \cdot 10^4$. Чтобы значение вещественной переменной было отображено в привычном для большинства пользователей виде, надо указать *формат* — количество позиций, которое следует использовать для отображения числа и количество цифр дробной части. Формат указывается после имени переменной (через двоеточие) и представляет собой два разделенных двоеточием числа: первое число задает количество позиций, второе — цифр дробной части. Например, если приведенную выше инструкцию записать `write(sum:9:2)`, то на экране появится строка:

```
15275.50
```

Для переменной целого типа (`integer`) можно указать количество позиций, резервируемое для отображения значения переменной, например: `write(period:3)`. Если количество цифр, которые должны появиться в результате вывода на экран значения переменной, меньше указанного в формате, то перед первой цифрой добавляется необходимое количество пробелов.

Несколько следующих одну за другой инструкций вывода сообщений и значений переменных можно объединить в одну, разделив сообщения и имена переменных запятыми. Например, следующие три инструкции:

```
write('Доход: ');
write(profit:6:2);
writeln(' руб.');
```

можно заменить одной:

```
writeln('Доход: ', profit:6:2, ' руб.');
```

Задание

Напишите инструкции, которые выводят на экран четверостишие:

У лукоморья дуб зеленый,
Златая цепь на дубе том.
И днем, и ночью кот ученый
Все ходит по цепи кругом.

А. С. Пушкин

Инструкция *readln*

Инструкция `readln` предназначена для ввода с клавиатуры исходных данных. В общем виде инструкция выглядит так:

```
readln(СписокПеременных);
```

где *СписокПеременных* — разделенные запятыми имена переменных, значение которых надо ввести с клавиатуры во время работы программы.

Примеры:

```
readln(sum);
readln(Cena, Kol);
```

Выполняется инструкция `readln` следующим образом. Сначала она ждет, пока на клавиатуре будут набраны данные и нажата клавиша <Enter>. После нажатия <Enter> введенные данные записываются в указанные переменные. После чего выполняется следующая инструкция программы.

Если в инструкции `readln` указаны несколько переменных, то данные следует вводить в одной строке, разделяя их пробелами.

Чтобы пользователь знал, какие данные ожидает от него программа, перед каждой инструкцией `readln` рекомендуется поместить инструкцию `write`, обеспечивающую вывод подсказки. Например:

```
write('Ширина (см) -> ');
readln(w);
write('Высота (см) -> ');
readln(h);
```

Следует обратить внимание, что тип данных, введенных с клавиатуры, должен соответствовать типу переменной, указанной в инструкции `readln`. Если это не так, то возникает ошибка и программа аварийно завершает работу (инструкции, следующие за `readln`, не выполняются).

Структура простой программы

Программа на языке Pascal — это последовательность инструкций. Начинается программа заголовком, за которым в простейшем случае следуют раздел объявления переменных и раздел выполняемых инструкций.

В общем виде программа выглядит так:

```
program Имя;
var
    { объявления переменных }
begin
    { инструкции }
end.
```

Заголовок программы состоит из слова `program`, за которым следует имя программы.

За заголовком следует раздел объявления переменных, в котором объявляются все переменные, используемые в программе. Раздел объявления переменных начинается словом `var`.

За разделом объявления переменных следует раздел инструкций. Раздел инструкций начинается со слова `begin` и заканчивается словом `end`, за которым следует символ "точка". В разделе инструкций находятся выполняемые инструкции программы.

Запись инструкций программы

Хотя в одной строке можно записать несколько инструкций, тем не менее принято каждую инструкцию писать на отдельной строке.

Некоторые инструкции (`if`, `case`, `repeat`, `while` и др.) записывают в несколько строк и с отступами. Например:

```
if d >= 0
   then
       begin
           x1 := (-b + Sqrt(d)) / (2*a);
           x2 := (-b - Sqrt(d)) / (2*a);
           writeln('Корни уравнения');
           writeln('x1 = ', x1:6:2, '   x2 = ', x2:6:2);
       end
    else
       writeln('Решения нет');
```

Здесь следует обратить внимание, что слова `then` и `else` записаны одно под другим и с отступом относительно слова `if`. Слова `end` записаны строго под

словами `begin`, а инструкции между `begin` и `end` так же записаны одна под другой, с отступом относительно `begin`.

Приведенный выше фрагмент можно записать и так:

```
if d >= 0
then
begin
x1 := (-b + Sqrt(d)) / (2*a);
x2 := (-b - Sqrt(d)) / (2*a);
writeln('Корни уравнения');
writeln('x1 = ',x1:6:2,'  x2 = ', x2:6:2);
end
else writeln('Решения нет');
```

С точки зрения функциональности оба приведенных фрагмента идентичны (делают одно и то же). Однако первый вариант записи более предпочтителен, так как он соответствует структуре алгоритма, реализуемого инструкцией `if`: достаточно одного взгляда, чтобы увидеть инструкции, которые будут выполнены, если условие `d>0` истинно, и инструкцию, которая будет выполнена, если условие ложно.

Длинные инструкции могут быть записаны в несколько строк. Разорвать инструкцию, перенести ее часть на следующую строку можно практически в любом месте. Нельзя разрывать имена переменных, числовые и строковые константы, а также составные операторы, например, оператор присваивания.

Ниже приведен пример записи инструкции в несколько строк:

```
writeln('Сумма вклада: ', sum:6:2, ' руб.  ',
        'Срок вклада (дней): ', period,
        'Процентная ставка (годовых): ', percent:6:2, '%');
```

Для облегчения понимания логики работы программы в текст программы рекомендуется включать комментарии. Комментарий — это текст, заключенный в фигурные скобки. Комментарий обычно располагают на отдельной строке или после инструкции. Ниже приведен пример раздела объявления переменных, в котором для пояснения назначения переменных использованы комментарии:

```
var
    { исходные данные }
    usd: real; { цена в долларах  }
    kurs: real; { курс ЦБ  }

    { результат }
    cena: real; { цена в рублях }
```

Стиль программирования

Работая над программой, программист, особенно начинающий, должен хорошо представлять, что программа, которую он разрабатывает, предназначена, с одной стороны, для пользователя, с другой — для самого программиста. Текст программы нужен прежде всего самому программисту, а также другим людям, с которыми он совместно работает над проектом. Поэтому, для того чтобы работа была эффективной, программа должна быть легко читаемой, ее структура должна соответствовать структуре и алгоритму решаемой задачи. Как этого добиться? Надо следовать правилам хорошего стиля программирования. Стиль программирования — это набор правил, которым следует программист (осознанно или потому, что "так делают другие") в процессе своей работы. Очевидно, что хороший программист должен следовать правилам хорошего стиля.

Хороший стиль программирования предполагает:

- использование комментариев;
- использование несущих смысловую нагрузку имен переменных, процедур и функций;
- использование при записи инструкций отступов;
- использование пустых строк.

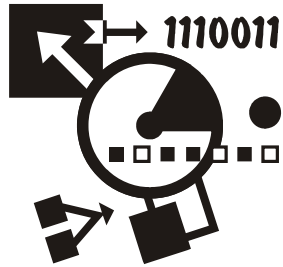
Следование правилам хорошего стиля программирования значительно уменьшает вероятность появления ошибок на этапе набора текста, делает программу легко читаемой, что в свою очередь облегчает процесс отладки и внесения изменений.

Четкого критерия оценки степени соответствия программы хорошему стилю программирования не существует. Вместе с тем, достаточно одного взгляда, чтобы понять, соответствует программа хорошему стилю или нет.

Сводить понятие стиля программирования только к правилам записи текста программы было бы неверно. Стиль, которого придерживается программист, проявляется во время работы программы. Хорошая программа должна быть прежде всего надежной и дружественной к пользователю.

Надежность подразумевает, что программа, не полагаясь на "разумное" поведение пользователя, контролирует исходные данные, проверяет результат выполнения операций, которые по какой-либо причине могут быть не выполнены, например, операций с файлами.

Дружественность предполагает разумное и предсказуемое, с точки зрения пользователя, поведение программы.



Глава 3

Алгоритмические структуры

Для того чтобы решить задачу, надо разработать (составить) алгоритм — правило перехода от исходных данных к результату. Алгоритм любой задачи можно представить как совокупность следующих структур:

- следование
- выбор
- цикл

Следование

Действия, которые надо выполнить, чтобы пересчитать расстояние из верст в километры (расчет по формуле), можно представить так: получить исходные данные (ввод), посчитать (расчет), отобразить результат (вывод). Это пример последовательного алгоритма. В алгоритмической структуре "следование" все действия (шаги алгоритма) выполняются последовательно, одно за другим и всегда в одном и том же порядке (рис. 3.1).

Рис. 3.1. Алгоритмическая структура "следование"

Выбор

На практике редко встречаются задачи, алгоритм решения которых является линейным. Действия, которые необходимо выполнить для достижения результата, как правило, зависят от исходных и промежуточных (полученных во время работы программы) данных. Например, в программе расчета сопротивления электрической цепи, состоящей из двух сопротивлений, формула, которую следует использовать для расчета, выбирается на основе информации о способе соединения сопротивлений.

Алгоритмическая структура, соответствующая ситуации, в которой надо выбрать действие (путь дальнейшего развития алгоритма) в зависимости от некоторого условия, называется выбором. Возможен выбор одного из двух вариантов (рис. 3.2) или одного из нескольких (рис. 3.3). Алгоритмическая структура "выбор" реализуется инструкцией `if`, "множественный выбор" — инструкцией `case`.

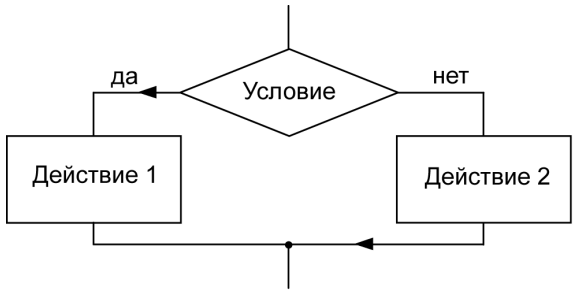


Рис. 3.2. Алгоритмическая структура **Выбор**

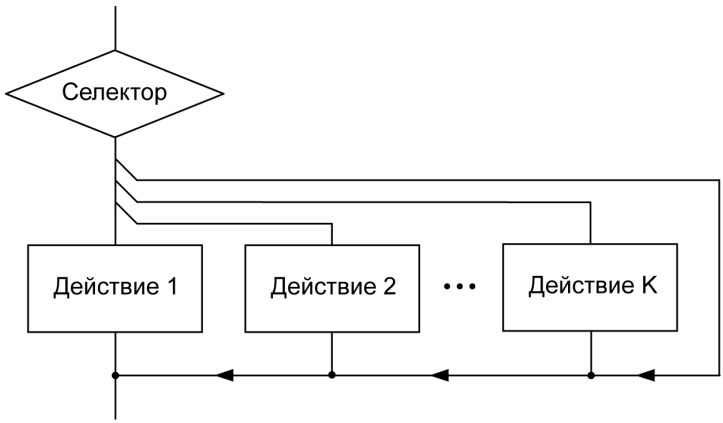


Рис. 3.3. Алгоритмическая структура **Множественный выбор**

Цикл

Часто для достижения результата одну и ту же последовательность действий надо выполнить несколько раз. Например, чтобы вычислить сумму и построить таблицу значений функции, надо вычислить значение функции, вывести на экран значение аргумента и функции, изменить значение аргумента, затем повторить описанные действия. Такой алгоритм решения задачи называется циклом.

Различают циклы с предусловием (рис. 3.4), постусловием (рис. 3.5) и циклы с фиксированным количеством повторений (рис. 3.6). Цикл с предусловием реализуется инструкцией `while`, постусловием — `repeat`. Цикл с фиксированным количеством повторений реализует инструкция `for`.

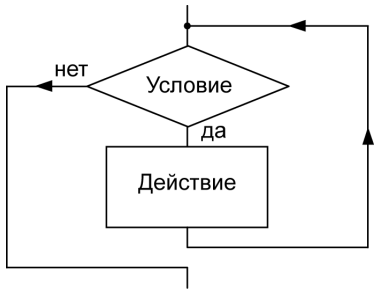


Рис. 3.4. Цикл с предусловием

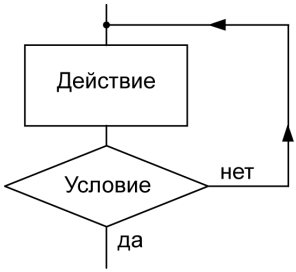


Рис. 3.5. Цикл с постусловием

Рис. 3.6. Цикл с фиксированным количеством повторений

Условие

Условие — это выражение логического типа (`boolean`), которое может принимать одно из двух значений: "истина" (`TRUE`) или "ложь" (`FALSE`).

Различают простое и сложное условия. Простое условие состоит из двух операндов и оператора сравнения. В общем виде простое условие записывается следующим образом:

Оп1 Оператор Оп2

где *Оп1* и *Оп2* — операнды условия, в качестве которых могут выступать переменная, константа, выражение или функция; *Оператор* — оператор сравнения.

В Turbo Pascal есть шесть операторов сравнения, которые перечислены в табл. 3.1.

Таблица 3.1. *Операторы сравнения*

Оператор		**Условие**
=	Равно	TRUE, если *Оп1* равен *Оп2*, иначе FALSE
<>	Не равно	TRUE, если *Оп1* не равен *Оп2*, иначе FALSE
>	Больше	TRUE, если *Оп1* больше *Оп2*, иначе FALSE
<	Меньше	TRUE, если *Оп1* меньше *Оп2*, иначе FALSE
>=	Больше или равно	TRUE, если *Оп1* больше или равен *Оп2*, иначе FALSE
<=	Меньше или равно	TRUE, если *Оп1* меньше или равен *Оп2*, иначе FALSE

Ниже приведены примеры условий.

```
Sum < 1000
i = j
```

В первом примере операндами условия являются переменная и константа. Значение этого условия зависит от значения переменной Sum. Условие будет верным и, следовательно, иметь значение TRUE, если значение переменной Sum меньше, чем 1000. Если значение переменной Sum больше или равно 1000, то значение этого условия будет FALSE.

Во втором примере в качестве операндов используются переменные. Значение этого условия будет TRUE, если значение переменной i равно значению переменной j.

Следует обратить внимание, что операнды операций сравнения должны быть одного типа. Если это не так, то во время компиляции программы при обнаружении в программе неверного условия, компилятор выводит сообщение: Error 26: Type mismatch (несовместимые типы).

Из простых условий (выражений логического типа) при помощи логических операторов and ("логическое И"), or ("логическое ИЛИ") и not (отрицание),

можно строить сложные условия. Результат применения логических операторов `and` и `or` к операндам логического типа `a` и `b` представлен в табл. 3.2.

Таблица 3.2. *Выполнение логических операций and и or*

a	b	a and b	a or b
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE

При записи сложных условий важно учитывать то, что логические операторы имеют более высокий приоритет, чем операторы сравнения, поэтому простые условия следует брать в скобки. Например, пусть условие предоставления скидки сформулировано следующим образом: "Скидка предоставляется, если сумма покупки 500 руб. и день покупки — воскресенье". Если день недели обозначить переменной `Day` целого типа (1 — понедельник, 2 — вторник и т. д.), то условие предоставления скидки можно записать так:

```
(Sum >= 500) and (Day = 7)
```

Если условие предоставления скидки расширить, например, указав, что скидка предоставляется в любой день, если сумма покупки превышает 999 рублей, то условие можно записать:

```
((Sum >= 500) and (Day = 7)) or (Sum > 999)
```

Выбор

Выбор одного из двух или нескольких возможных вариантов можно реализовать при помощи инструкций `if` или `case`.

Инструкция *IF*

Инструкция `if` используется тогда, когда надо выбрать один из двух возможных вариантов действий. В общем виде инструкция `if` записывается так:

```
if Условие
    then
        begin
            { инструкции, которые надо выполнить,
              если условие истинно }
        end
    else
```

```
begin
     { инструкции, которые надо выполнить,
       если условие ложно }
end;
```

Выполняется инструкция `if` следующим образом. Сначала вычисляется значение выражения *Условие*. Затем, если условие выполняется (значение выражения *Условие* равно `TRUE`), выполняются инструкции, следующие за словом `then` (находящиеся между `begin` и `end`). Если условие не выполняется (значение выражения *Условие* равно `FALSE`), то выполняются инструкции, следующие за словом `else` (находящиеся между `begin` и `end`).

Алгоритм, реализуемый инструкцией `if`, представлен на рис. 3.7.

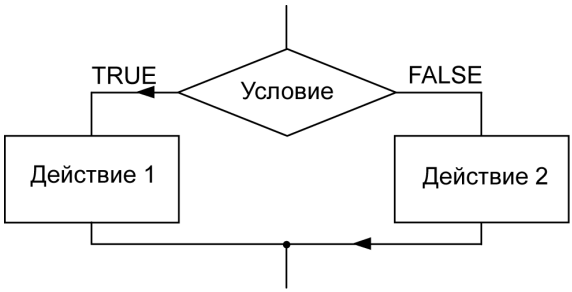


Рис. 3.7. Алгоритм инструкции `if`

В качестве примера рассмотрим следующую задачу. Пусть надо вычислить сопротивление электрической цепи, состоящей из двух резисторов, которые могут быть соединены последовательно или параллельно. Обозначим: `r1` и `r2` — величины сопротивлений первого и второго резисторов; `r` — сопротивление цепи; `t` — способ соединения резисторов (1 —последовательное; 2 — параллельное). Так как формула вычисления сопротивления цепи выбирается в зависимости от способа соединения резисторов (значения переменной `t`), то для вычисления следует использовать инструкцию `if`:

```
if t = 1
    then
        begin
            r := r1 + r2;
        end
    else
        begin
            r := (r1 * r2) / (r1 + r2);
        end;
```

Следует обратить внимание, если между `begin` и `end` находится только одна инструкция, то слова `begin` и `end` можно не писать. То есть приведенную инструкцию можно записать так:

```
if t = 1
    then
        r := r1 + r2
    else
        r := (r1 * r2) / (r1 + r2);
```

Полный текст программы вычисления сопротивления цепи приведен в листинге 3.1.

Листинг 3.1. Вычисление сопротивления электрической цепи (p3_1.pas)

```
{ Вычисление сопротивления электрической цепи }
program p3_1;
var
    r1,r2: real;{ сопротивление резисторов}
    r: real; { сопротивление цепи}
    t: integer; { способ соединения резисторов:
                  1 - последовательное; 2 - параллельное}
begin
    writeln('Вычисление сопротивления цепи');
    writeln('Введите значения сопротивлений (Ом) ');
    write('R1 -> ');
    readln(r1);
    write('R2 -> ');
    readln(r2);
    write('Способ соединения ');
    write('(1 - послед.; 2 - параллельное) -> ');
    readln(t);
    if t = 1
        then
                r := r1 + r2
        else
                r := (r1 * r2) / (r1 + r2);
    writeln('Сопротивление цепи: ', r:6:2, ' Ом');
    writeln;
    write('Для завершения работы программы нажмите <Enter>');
    readln;
end.
```

Если какое-то действие нужно выполнить только в том случае, если условие выполняется, и пропустить ("ничего не делать"), если условие не выполняется, то "веточку" `else` можно не писать. Например, в программе проверки знаний значение счетчика правильных ответов увеличивается только в том случае, если ответ правильный:

```
if n = right { n – номер выбранного ответа, right – правильного}
    then
        k := k + 1; { увеличить счетчик правильных ответов}
```

Часто в программе необходимо реализовать выбор более чем из двух вариантов. Например, известно, что для каждого человека существует оптимальное значение веса, которое может быть вычислено по формуле: Рост(см) –100. Очевидно, что реальный вес может быть меньше оптимального, равняться ему или превышать. Следующая программа запрашивает вес и рост, вычисляет оптимальное значение веса, сравнивает его с реальным и выводит соответствующее сообщение. Алгоритм программы приведен на рис. 3.8, текст — в листинге 3.2.

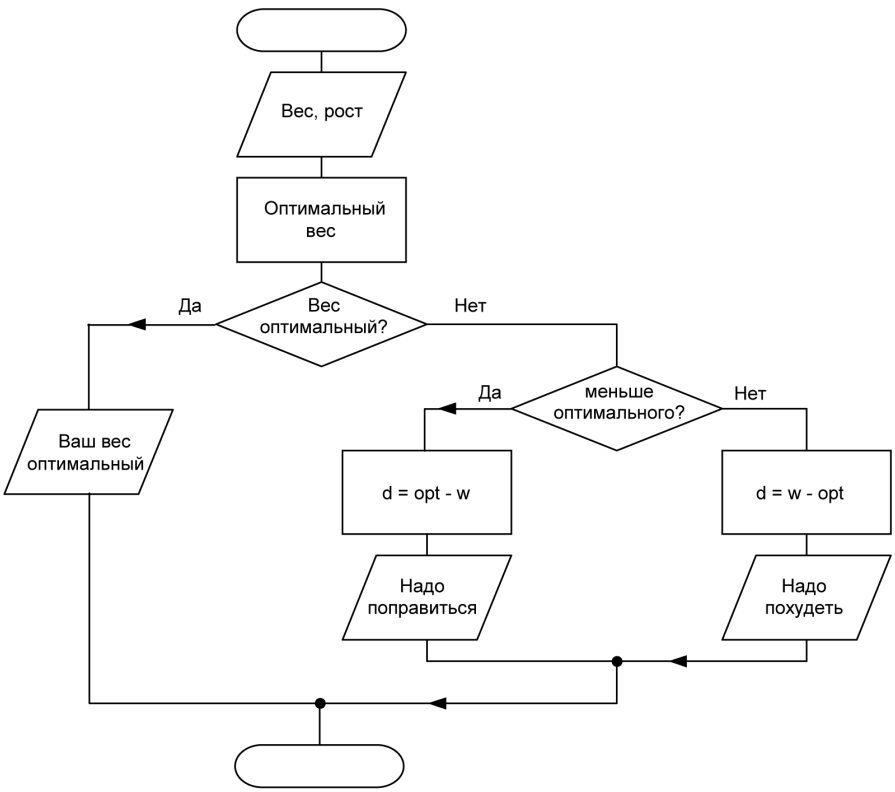


Рис. 3.8. Алгоритм программы **Контроль веса**

Листинг 3.2. Контроль веса (p3_2.pas)

```
{ Контроль веса }
program p3_2;
var
     w:real;   { вес}
     h:real;   { рост }
     opt:real; { оптимальный вес }
     d:real;   { отклонение от оптимального веса }
begin
    writeln('Введите в одной строке ');
    writeln('вес (кг), рост (см) и нажмите <Enter>');
     write('->');
     readln(w,h);
     opt := h - 100;
     if w=opt
         then
              writeln('Ваш вес оптимален!')
         else
            if w < opt
               then begin
                        d := opt - w;
                        writeln('Вам надо поправиться на ',d:5:2,' кг');
                    end
               else begin
                        d:=w - opt;
                        writeln('Вам надо похудеть на ',d:5:2,' кг');
                    end;
    writeln;
    write('Для завершения работы программы нажмите <Enter>');
    readln;
end.
```

Инструкция *CASE*

Существует достаточное количество задач, в которых выбор одного из нескольких возможных вариантов (множественный выбор) осуществляется на основе сравнения значения выражения целого типа (в простейшем случае переменной) с константой (типичный пример — реализация меню).

Инструкция `case` позволяет реализовать множественный выбор и в общем виде записывается так:

```
case Селектор of
    Список1 : begin
                { последовательность инструкций 1 }
              end;
    Список2:  begin
                { последовательность инструкций 2 }
              end;
    СписокN : begin
                { последовательность инструкций N }
              end;
     else
              begin
                { последовательность инструкций, выполняемая }
                { в случае, если значение Селектора не попало }
                { ни в один из списков}
              end;
end;
```

где:

Селектор — выражение целого типа, значение которого определяет последовательность инструкций, которая будет выполнена, если значение выражения *Селектор* будет равно константе, указанной в списке.

Список$_i$ — список констант, с которыми сравнивается значение выражения *Селектор* для выбора действия. Если каждый элемент списка на единицу больше предыдущего, то вместо списка можно указать диапазон. Например, списку 1,2,3,4,5 соответствует диапазон 1..5.

Следует обратить внимание, если между `begin` и `end` находится только одна инструкция, то `begin` и `end` можно не писать.

Примеры:

```
case day of
    1,2,3,4,5: write('Рабочий день');
    6: write('Суббота');
    7: write('Воскресенье');
end;
case day of
```

```
    1..5: write('Рабочий день');
    6: write('Суббота');
    7: write('Воскресенье');
end;
case day of
    6: write('Суббота');
    7: write('Воскресенье');
    else write('Рабочий день');
end;
```

Выполняется инструкция `case` следующим образом. Сначала вычисляется значение выражения, указанного после `case`. Затем полученное значение последовательно сравнивается с константами, указанными в списках *Список*$_1$, *Список*$_2$ и т. д. Если значение выражения *Селектор* равно значению, указанному в списке, то выполняется соответствующая последовательность инструкций, и на этом выполнение инструкции `case` завершается. Если значение выражения *Селектор* не совпадает ни с одной константой, то выполняется последовательность инструкций, следующая за `else`. Синтаксис инструкции `case` позволяет не писать `else` и соответствующую последовательность инструкций. В этом случае, если значение выражения не совпадает ни с одной константой, то выполняется следующая за `case` инструкция программы.

Алгоритм инструкции `case` приведен на рис. 3.9.

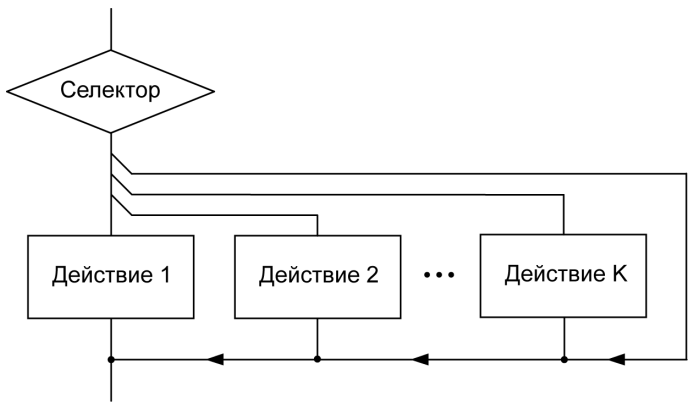


Рис. 3.9. Алгоритм инструкции `case`

Следующая программа, ее алгоритм представлен на рис. 3.10, а текст — в листинге 3.3, демонстрирует использование инструкции `case`, она позволяет рассчитать цену жалюзи.

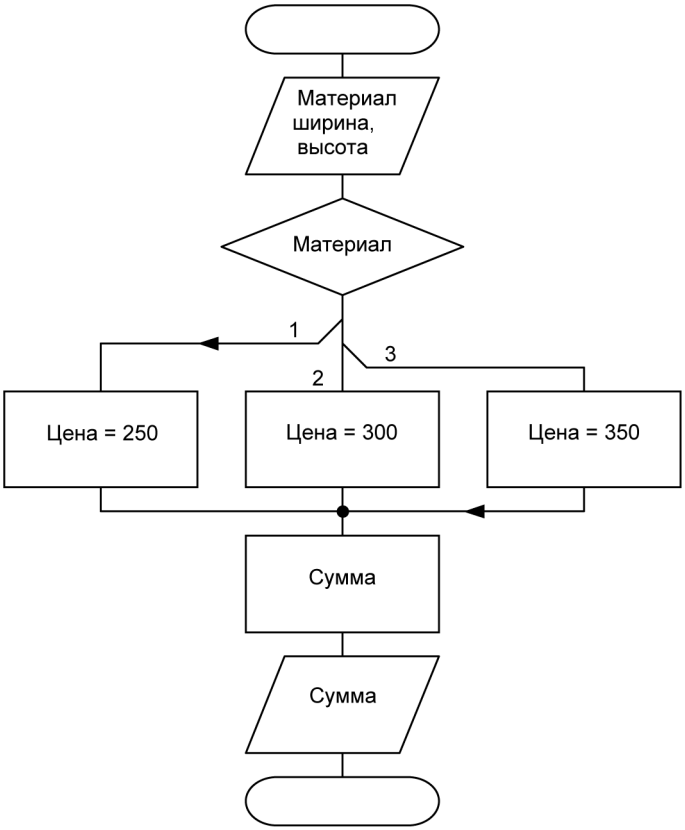


Рис. 3.10. Алгоритм программы **Жалюзи**

Листинг 3.3. Жалюзи (p3_3.pas)

```
program p3_3;
var
    w,h: integer; { высота и ширина}
    m: integer; { материал: 1 – пластик; 2 – текстиль; 3 – алюминий }
    cena: real; { цена за 1 кв.м.}
    st: string; {название материала }
    sum: real; {сумма}
begin
    writeln('Жалюзи');
    write('Ширина (см) ->');
    readln(w);
```

```
    write('Высота (см) ->');
    readln(h);
    write('Материал (1 - пластик; 2 - текстиль; 3 - алюминий) -> ');
    readln(m);
    case m of
        1: begin
               cena := 250;
               st := 'пластик';
           end;
        2: begin
               cena := 300;
               st := 'текстиль';
           end;
        3: begin
               cena := 350;
               st := 'алюминий';
            end;
    end;
    sum := w * h / 10000 * cena;

    writeln;
    writeln('Размер: ', w, 'x', h);
    writeln('Материал: ', st);
    writeln('Сумма: ',sum:6:2, 'руб.');

    writeln('Для завершения работы программы нажмите <Enter>');
    readln;
end.
```

Циклы

Алгоритмы решения многих задач являются циклическими — для достижения результата определенную последовательность действий необходимо выполнить несколько раз. Например, программа проверки знания таблицы умножения должна вывести вопрос (пример), получить ответ испытуемого, сравнить его с правильным, увеличить счетчик правильных ответов (если ответ правильный), затем повторить описанную последовательность действий еще раз, и так до тех пор, пока не будут выведены все вопросы. Другой пример.

Чтобы найти фамилию в списке, надо проверить первую фамилию списка, затем вторую, третью и т. д. до тех пор, пока не будет найдена нужная или не будет достигнут конец списка. Такие повторяющиеся действия называются циклами и реализуются в программе с использованием инструкций циклов.

Повторяющиеся действия могут быть реализованы при помощи инструкций `for`, `while` или `repeat`.

Цикл *FOR*

Цикл `for` используется, если некоторую последовательность действий надо выполнить несколько раз, причем число повторений можно определить заранее (задать во время разработки программы или вычислить во время ее работы).

В общем виде инструкция `for` записывается так:

```
for сч := нз to кз do
   begin
     { инструкции ("тело") цикла }
   end;
```

где:

сч — переменная-счетчик числа повторений инструкций цикла;

нз — выражение целого типа, определяющее начальное значение переменной-счетчика циклов (в простейшем случае — константа);

кз — выражение, определяющее конечное значение переменной-счетчика циклов (в простейшем случае — константа).

Пример:

```
for i:=1 to 10 do
     begin
         writeln('Turbo Pascal');
     end;
```

Инструкции, находящиеся между `begin` и `end`, называют инструкциями цикла. Количество повторений инструкций цикла можно вычислить по формуле: *кз* - *нз* + 1. Следует обратить внимание, если начальное значение счетчика циклов больше конечного, то инструкции цикла ни разу не будут выполнены.

Если в инструкции `for` вместо слова `to` записать `downto`, то после очередного цикла значение счетчика будет не увеличиваться, а уменьшаться. Например, инструкция

```
for j:=10 downto 1 do
     writeln(i);
```

выводит на экран числа от 10 до 1.

Следующая программа демонстрирует использование цикла `for`. Программа, ее алгоритм приведен на рис. 3.11, а текст — в листинге 3.4, вычисляет среднее арифметическое чисел, введенных пользователем.

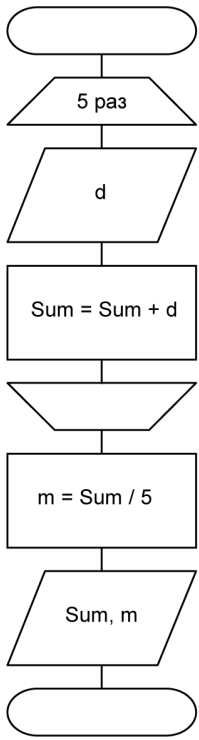


Рис. 3.11. Вычисление среднего арифметического

Листинг 3.4. Среднее арифметическое (p3_4.pas)

```
{ вычисляет среднее арифметическое чисел, введенных с клавиатуры}
program p3_4;
var
    i: integer; { счетчик циклов }
    d: real; {число, введенное пользователем}
    sum: real; { сумма введенных чисел}
    m: real; { среднее арифметическое}
begin
    writeln('Среднее арифметическое');
    write('Введите 5 чисел. ');
```

```
    writeln('После ввода каждого числа нажимайте <Enter>');
    sum := 0;
    for i :=1 to 5 do
        begin
            write(' > '); { символ-подсказка}
            readln(d);
            sum := sum + d; { прибавить введенное число к сумме}
        end;

    m := sum / 5;
    writeln;
    writeln('Сумма: ', sum:6:2);
    writeln('Среднее арифметическое: ', m:6:2 );
    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Цикл *REPEAT*

Цикл `repeat` используется в программе, если надо несколько раз выполнить некоторое действие, но число повторов заранее (во время разработки программы или непосредственно перед началом выполнения цикла) определить нельзя.

В общем виде инструкция `repeat` записывается так:

```
repeat
     { инструкции цикла }
until  условие;
```

где:

условие — условие (выражение логического типа) завершения цикла.

Инструкция `repeat` выполняется следующим образом: сначала выполняются инструкции, которые находятся между `repeat` и `until`. Затем проверяется значение выражения *условие*. Если значение выражения *условие* равно `FALSE`, то инструкции цикла выполняются еще раз. Если значение выражения *условие* равно `TRUE`, то выполнение цикла прекращается. Таким образом, инструкции цикла (находящиеся между `repeat` и `until`) выполняются до тех пор, пока значение выражения условие равно `FALSE`. Алгоритм инструкции `repeat` представлен на рис. 3.12.

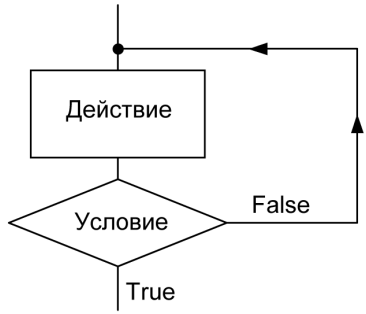


Рис. 3.12. Алгоритм цикла `repeat`

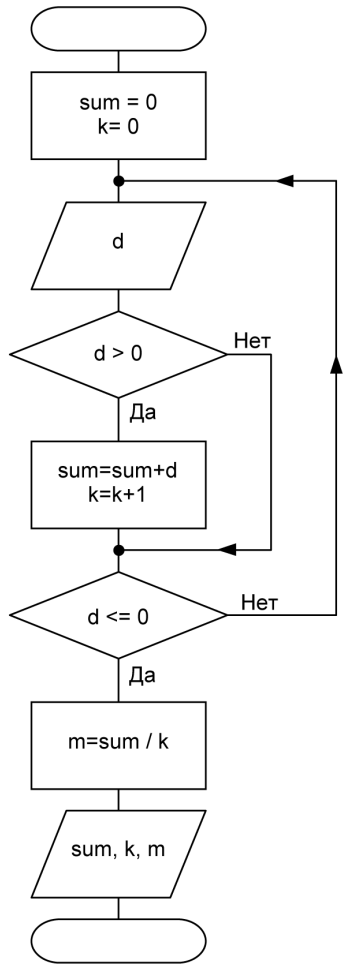


Рис. 3.13. Среднее арифметическое последовательности положительных чисел

Обратите внимание:

- инструкции цикла (между `repeat` и `until`) будут выполнены хотя бы один раз;
- для того чтобы цикл завершился, необходимо, чтобы инструкции, находящиеся между `repeat` и `until`, изменяли значения переменных, входящих в выражение *условие*.

Следующая программа демонстрирует использование цикла `repeat`. Программа, ее алгоритм приведен на рис. 3.13, а текст — в листинге 3.5, вычисляет среднее арифметическое положительных чисел, введенных пользователем с клавиатуры. Количество чисел заранее неизвестно, числа считываются до тех пор, пока пользователь не введет ноль или отрицательное число.

Листинг 3.5. Среднее арифметическое положительных чисел (p3_5.pas)

```
 {  среднее арифметическое положительных чисел, введенных с клавиатуры }
program p3_5;
var
  d: real;      { число, введенное с клавиатуры }
  sum: real; { сумма введенных чисел }
  k: integer; { количество чисел }
  m: real; { среднее арифметическое }
begin
  writeln('Вычисление среднего арифметического положительных целых');
  write('Вводите числа.');
  writeln('После ввода каждого числа нажимайте <Enter>');
  write('Для завершения введите 0');
  sum := 0;
  k := 0;
  repeat
     write('->');
     readln(d);
     if d>0 then
         begin
            sum := sum + d;
            k := k +1;
         end;
  until d <= 0;
  m := sum / k;

  writeln;
```

```
  writeln('Сумма введенных чисел: ',sum:6:2);
  writeln('Количество чисел: ', k);
  writeln('Среднее арифметическое: ', m:6:2);

  writeln;
  write('Для завершения нажмите <Enter>');
  readln;
end.
```

Цикл *WHILE*

Цикл `while`, как и цикл `repeat`, используется в программе, если надо несколько раз выполнить некоторое действие, но число повторов заранее (во время разработки программы или непосредственно перед началом выполнения цикла) определить нельзя.

В общем виде инструкция `while` записывается так:

```
while условие do
   begin
     { последовательность инструкций }
   end;
```

где:

условие — выражение логического типа, определяющее условие выполнения инструкций цикла (находящихся между `begin` и `end`).

Инструкция `while` выполняется следующим образом: сначала проверяется значение выражения *условие*. Если значение выражения *условие* равно `TRUE`, то выполняются инструкции цикла (расположенные между `begin` и `end`) после чего снова проверяется значение выражения *условие* и, если оно равно `TRUE`, инструкции цикла выполняются еще раз. И так до тех пор, пока условие истинно (значение выражения *условие* равно `TRUE`). Алгоритм инструкции `while` представлен на рис. 3.14.

Следует обратить внимание:

- для того чтобы инструкции цикла `while` были выполнены хотя бы один раз, необходимо, чтобы значение выражения *условие* было равно `TRUE`;
- для того чтобы цикл завершился, необходимо, чтобы последовательность инструкций между `begin` и `end` изменяла значения переменных, входящих в выражение *условие*.

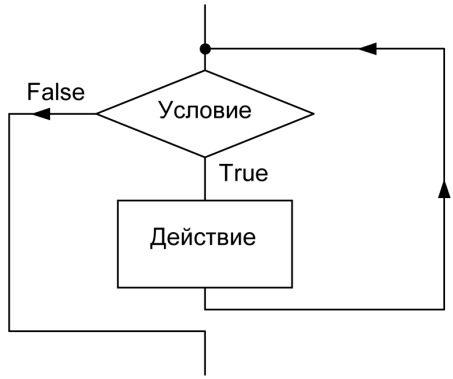


Рис. 3.14. Алгоритм инструкции `while`

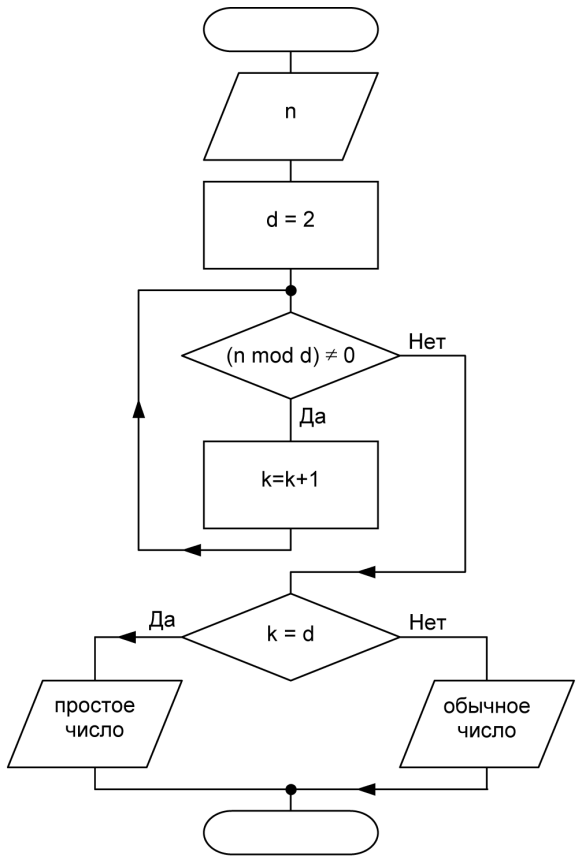
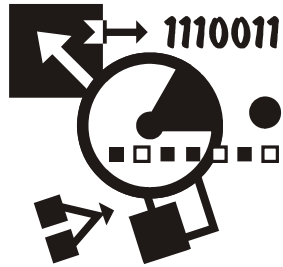


Рис. 3.15. Алгоритм программы Простое число

Следующая программа, ее алгоритм приведен на рис. 3.15, а текст — в листинге 3.6, проверяет, является ли число, введенное пользователем, простым (простым называется число, которое делится только само на себя). Проверка выполняется путем поиска числа, на которое проверяемое число делится без остатка. Проверяемое число делится по модулю на 2, 3, 4 и т. д. до тех пор, пока результат деления (остаток) не равен нулю. Если остаток равен нулю, это значит, что найдено число, на которое исходное число разделилось без остатка. Если это найденное число равно исходному, то число, введенное пользователем, — простое.

Листинг 3.6. Проверка, является ли число простым (p3_6.pas)

```
{ проверяет, является ли число простым }
program p3_6;
var
   n: integer; { проверяемое число (делимое) }
   k: integer; { делитель }


begin
     write('->');
     readln(n);

     k := 2;
     while ( n mod k <> 0 ) do
       begin
            k := k + 1
       end;

     if n = k
        then writeln(n, ' -  простое число')
        else writeln(n, ' -  обычное число');


     write('Для завершения нажмите <Enter>');
     readln;
end.
```



Глава 4

Массивы

Массив — это структура данных, которая представляет собой набор (совокупность) переменных одинакового типа. Различают одномерные и двумерные массивы. Графически массив можно изобразить так, как показано на рис. 4.1. Массивы используют для хранения однородной по своей структуре информации: одномерные — списков, двумерные — таблиц.

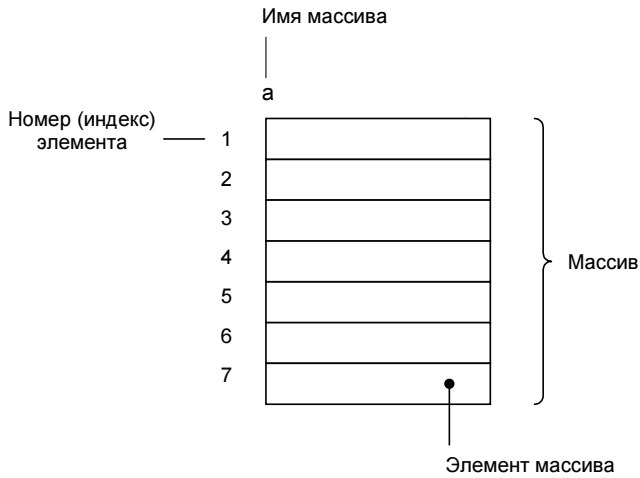


Рис. 4.1. Графическое представление массива

Объявление массива

Массив, как и любая переменная, должен быть объявлен в разделе объявления переменных. В общем виде объявление одномерного массива выглядит так:

Имя:**array**[*нижний_индекс*..*верхний_индекс*] **of** *тип*

где

- *Имя* — имя массива;
- **array** — ключевое слово, обозначающее, что объявляемая переменная является массивом;
- *нижний_индекс* и *верхний_индекс* — целые числа, определяющие диапазон изменения индексов (номеров) элементов массива и, неявно, количество элементов массива;
- *тип* — тип элементов массива.

Примеры объявления массивов:

```
a: array[1..10] of real;
b: array[0..4] of integer;
name: array[1..30] of string[25];
```

При объявлении массива вместо целых констант, задающих значения индексов, удобно использовать именованные константы (объявление именованной константы следует поместить в раздел `const`). Например, приведенное выше объявление массива `a` можно заменить на следующее:

```
const
      N = 10;
var
      a: array[1..N] of real;
```

Доступ к элементу массива

Чтобы получить доступ к элементу массива, надо указать имя массива и номер элемента, заключив его в квадратные скобки. В качестве номера можно использовать выражение целого типа (в простейшем случае — константу или переменную). Например:

```
a[1]
a[i]
a[i+1]
```

Все операции, которые можно выполнять над переменными, можно выполнять и над элементами массива:

```
a[3] := 0;
writeln(a[3]:6:2);
readln(a[3]);
```

Ввод массива

Под вводом массива понимается ввод значений всех элементов массива. Обычно значения элементов массива вводят последовательно, одно за другим, начиная с первого. Ввод удобно реализовать при помощи инструкции `for`. Чтобы пользователь знал, какой элемент он должен ввести, следует организовать систему отображения подсказок. В подсказке обычно указывают индекс элемента массива, значение которого надо ввести.

Следующая программа, ее текст приведен в листинге 4.1, демонстрирует ввод и обработку массива. В процессе ввода массива в строке подсказки отображается номер элемента, который надо ввести. Вид экрана во время работы программы приведен на рис. 4.2.

Листинг 4.1. Ввод массива (p4_1.pas)

```
{ ввод и обработка массива }
program p4_1;
var
   a: array[1..10] of integer; { массив }
   i: integer; { номер элемента }
   sum: integer; { сумма элементов массива}
begin
     writeln('Ввод массива');
     writeln('После ввода каждого числа нажимайте <Enter>');
     for i := 1 to 10 do
         begin
              write(i,'>'); { подсказка - номер элемента}
              readln(a[i]);
         end;

     { вычислить сумму элементов массива }
     for i:=1 to 10 do
         sum := sum + a[i];

     writeln;
     writeln(' Сумма элементов массива: ', sum);
     writeln;
     write('Для завершения нажмите <Enter>');
     readln;
end.
```

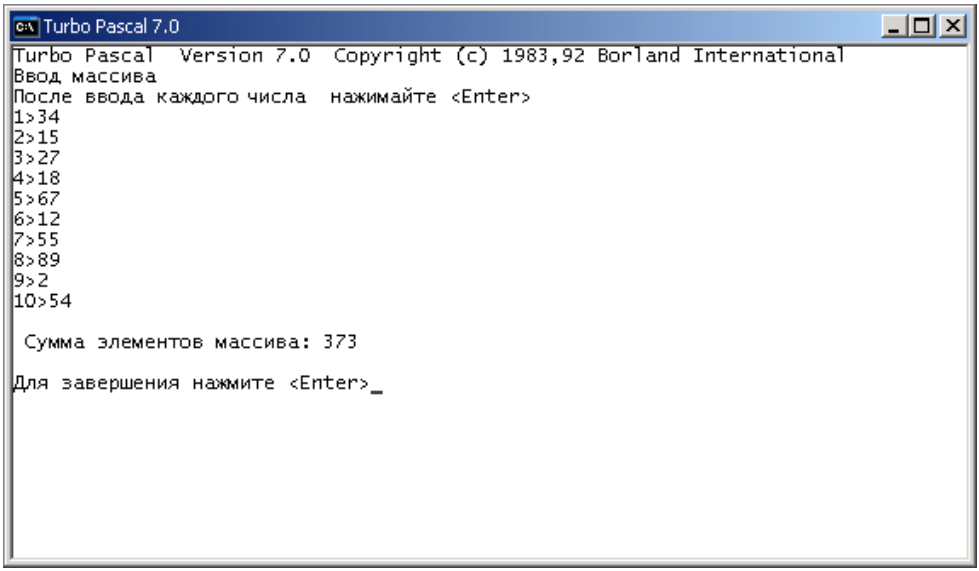


Рис. 4.2. Ввод массива

В приведенной выше программе после набора каждого числа пользователь должен нажимать <Enter>, что не всегда удобно. Следующая программа (листинг 4.2) демонстрирует другой способ ввода массива: значения вводятся в одной строке.

Листинг 4.2. Ввод массива (p4_2.pas)

```
{ ввод и обработка массива }
program p4_2;
const
   N = 5; { размер массива (количество элементов) }
var
   a: array[1..N] of integer; { массив }
   i: integer; { номер элемента }
   sum: integer; { сумма элементов массива}
begin
     writeln('Ввод массива');
     writeln('Введите в одной строке ',N,' чисел и нажмите <Enter>');
     write('>');

     for i := 1 to N - 1 do
```

```
        read(a[i]);
    readln(a[N]);

    { вычислить сумму элементов массива }
    for i:=1 to N do
        sum := sum + a[i];

    writeln;
    writeln('Сумма элементов массива: ', sum);
    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

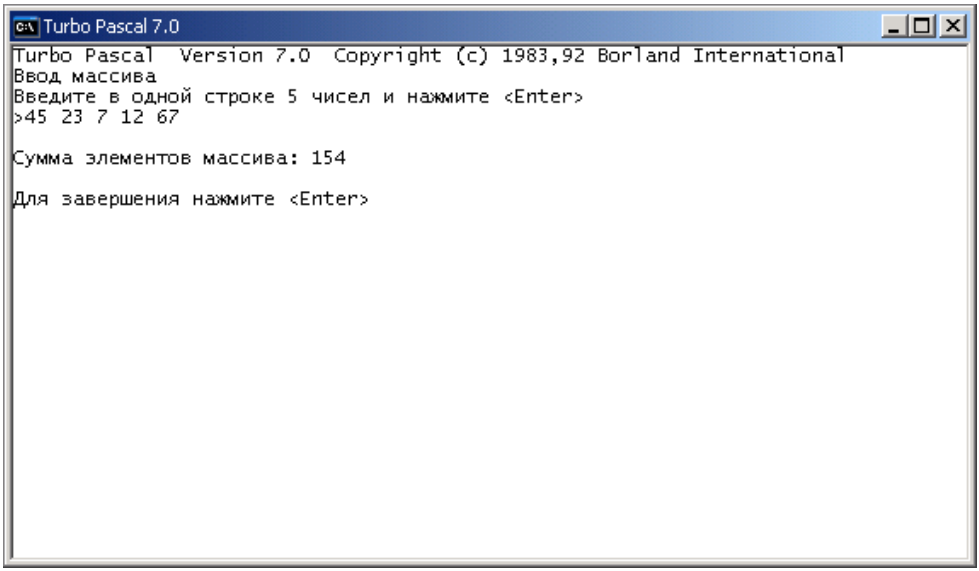


Рис. 4.3. Ввод массива

Инструкции `read` и `readln` читают информацию не с клавиатуры, а из *буфера* клавиатуры. В начале работы программы буфер клавиатуры автоматически очищается. Поэтому инструкции ввода ждут, пока информация там появится (информация появляется в буфере в результате нажатия пользователем клавиши <Enter>). Как только информация появляется в буфере клавиатуры, инструкция чтения забирает оттуда элемент данных (строку символов до первого пробела). Причем инструкция `read` "просто забирает", а `readln` — забирает

и после этого очищает буфер клавиатуры. Таким образом, если в буфере клавиатуры находится несколько элементов данных (например, разделенные пробелами числа), то первая инструкция `read` прочитает первый элемент данных, вторая — второй и т. д.

Работает программа следующим образом. Первый раз (когда значение счетчика циклов равно 1) инструкция `read` ждет, пока пользователь наберет на клавиатуре строку символов и нажмет <Enter>. Как только пользователь введет требуемую информацию (5 чисел в одной строке), инструкция `read` "заберет" первое число из буфера клавиатуры и запишет в первый элемент массива. Следующая инструкция `read` прочитает второй элемент данных и т. д. Последний элемент данных забирает инструкция `readln`, именно `readln`, а не `read`. Сделано это для того, чтобы после прочтения последнего элемента данных очистить буфер клавиатуры. Вид экрана во время работы программы приведен на рис. 4.3.

Вывод массива

Под выводом массива понимается вывод на экран значений всех элементов массива. Проще всего вывести массив можно при помощи инструкции `for`, используя счетчик циклов для доступа к элементам массива. В листинге 4.3 приведена программа, которая выводит каждый элемент массива в отдельной строке. Программа, приведенная в листинге 4.4, выводит элементы массива в одной строке.

Листинг 4.3. Вывод массива в несколько строк (p4_3.pas)

```
{ вывод массива }
program p4_3;
const
   N = 5; { размер массива (количество элементов) }
var
   a: array[1..N] of integer; { массив }
   i: integer; { номер элемента }
   sum: integer; { сумма элементов массива}
begin
     writeln('Ввод массива');
     writeln('Введите в одной строке ',N,' чисел и нажмите <Enter>');
     write('>');

     for i := 1 to N - 1 do
```

```
        read(a[i]);

    readln(a[N]);

    { вывод массива }
    writeln;
    writeln('Введенный массив:');
    for i:=1 to N do
        writeln(i, ' ', a[i]);

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Листинг 4.4. Вывод массива в одну строку (p4_4.pas)

```
{ вывод массива в одну строку}
program p4_4;
const
   N = 5; { размер массива (количество элементов) }
var
   a: array[1..N] of integer; { массив }
   i: integer; { номер элемента }
   sum: integer; { сумма элементов массива}
begin
    writeln('Ввод массива');
    writeln('Введите в одной строке ',N,' чисел и нажмите <Enter>');
    write('>');

    for i := 1 to N - 1 do
        read(a[i]);
    readln(a[N]);

    { вывод массива }
    writeln;
    write('Введенный массив:');
```

```
    for i:=1 to N - 1 do
        write(a[i], ' ');
    writeln(a[N]);

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Поиск минимального элемента

Задача поиска минимального элемента массива заключается в определении номера элемента, значение которого не превышает значение остальных элементов массива. Предполагается, что элементы массива не упорядочены (если элементы упорядочены, т. е. для каждого i-го элемента массива `a` справедливо `a[i-1] ≤ a[i] ≤ a[i+1]`, то задача поиска минимального элемента теряет смысл). Алгоритм поиска минимального элемента можно сформулировать так:

- Предположим, что первый элемент минимальный.
- Будем последовательно сравнивать элементы массива (со второго) с минимальным. Если текущий элемент меньше минимального, запомним его номер, как номер минимального элемента. Перейдем к следующему элементу.

Программа, реализующая описанный алгоритм, приведена в листинге 4.5, пример ее работы — на рис. 4.4.

Листинг 4.5. Поиск минимального элемента (p4_5.pas)

```
{ поиск минимального элемента массива }
program p4_5;
const
   N = 5; { размер массива (количество элементов) }
var
   a: array[1..N] of integer; { массив }
   min: integer; { минимальный элемент (номер) }
   i: integer; { текущий элемент }
begin
    writeln('Поиск минимального элемента массива');
```

```
    write('Введите в одной строке элементы массива (');
    writeln(N,' чисел) и нажмите <Enter>');
    write('>');
    for i:=1 to N-1 do
        read(a[i]);
    readln(a[N]);

    { поиск }
    min := 1;  { предположим, что первый элемент минимальный }

    for i := 2 to N do
        if a[i] < a[min] then min := i;

    write('Минимальный элемент: ', a[min]);
    writeln(' Номер минимального элемента:', min);

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

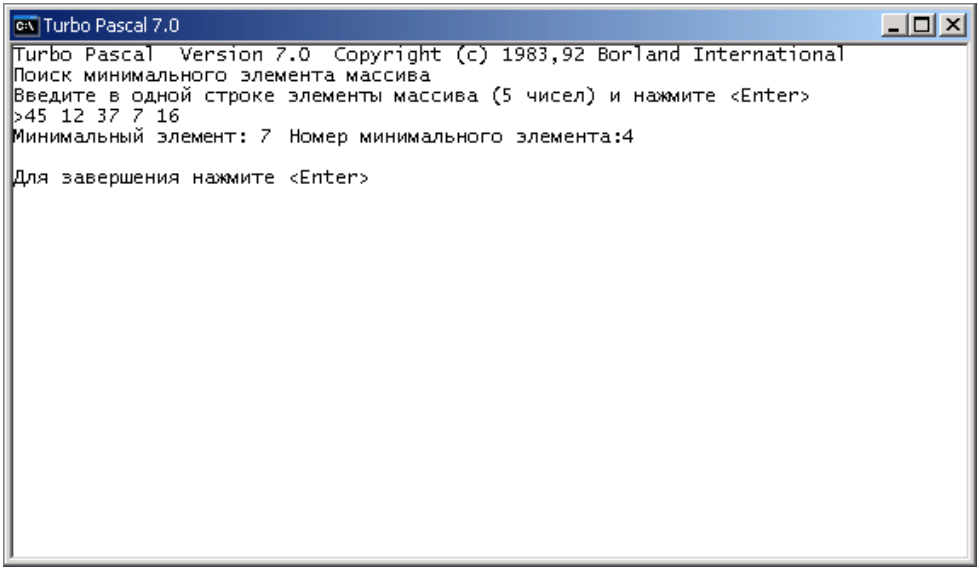


Рис. 4.4. Поиск минимального элемента

Сортировка массива

Под сортировкой массива понимается процесс перестановки элементов с целью упорядочивания их в соответствии с каким-либо критерием.

Различают сортировку по возрастанию и убыванию.

Массив, для элементов которого выполняется условие:

```
a[1] ≤ a[2] ≤ … a[i-1] ≤ a[i] ≤ a[i+1] … ≤ a[k]
```

называется упорядоченным по возрастанию.

Массив, для элементов которого выполняется условие:

```
a[1] ≥ a[2] ≥ … a[i-1] ≥ a[i] ≥ a[i+1] … ≥ a[k]
```

называется упорядоченным по убыванию.

Так как можно сравнивать переменные типов `integer`, `real`, `char` и `string`, то можно сортировать массивы этих типов.

Задача сортировки распространена в информационных системах и используется как предварительный этап задачи поиска, так как поиск в упорядоченном (отсортированном) массиве можно выполнить значительно быстрее, чем в неупорядоченном.

Существует достаточно много методов сортировки массивов. Здесь мы рассмотрим два:

- ❒ метод прямого выбора
- ❒ метод прямого обмена

Сортировка методом прямого выбора

Алгоритм сортировки массива по возрастанию методом прямого выбора может быть представлен так:

1. Просматривая массив от первого элемента, найти минимальный элемент и поместить его на место первого элемента, а первый — на место минимального.
2. Просматривая массив от второго элемента, найти минимальный элемент и поместить его на место второго элемента, а второй — на место минимального.
3. И так далее до предпоследнего элемента.

Программа сортировки массива по возрастанию представлена в листинге 4.6. Для демонстрации процесса сортировки программа выводит массив после

каждого цикла. Окно, в котором работает программа сортировки массива, приведено на рис. 4.5.

Листинг 4.6. Сортировка массива по возрастанию методом прямого выбора (p4_6.pas)

```
{ сортировка массива методом прямого выбора }
program p4_6;
const
     N=5;
var
     a: array[1..N] of integer;
     i:integer;    { номер элемента, от которого ведется поиск
                     минимального эл-та }
     min:integer; { номер минимального элемента в части }
                  { массива от i до верхней границы массива }
     j:integer;    { номер эл-та, сравниваемого с минимальным }
     buf:integer; { буфер, используемый при обмене эл-тов массива }

     k: integer;

begin
     writeln('Сортировка массива методом прямого выбора');
     write('Введите в одной строке ',N,' целых чисел ');
     writeln('и нажмите <Enter>');
     write('>');

     for k := 1 to N-1 do
         read(a[k]);
     readln(a[N]);


     writeln('Сортировка');
     for i:=1 to N-1 do
     begin
          { поиск минимального эл-та
          в части массива от a[i] до a[N]}

          min:=i;
```

```
        for j:=i+1 to N do
            if a[j]<a[min] then
                min:=j;
            { поменяем местами a[min] и a[i] }
            buf:=a[i];
            a[i]:=a[min];
            a[min]:=buf;

            { выведем массив }
            for k:=1 to N-1 do
               write(a[k],' ');
            writeln(a[N]);
    end;

    writeln;
    writeln('Массив отсортирован.');
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

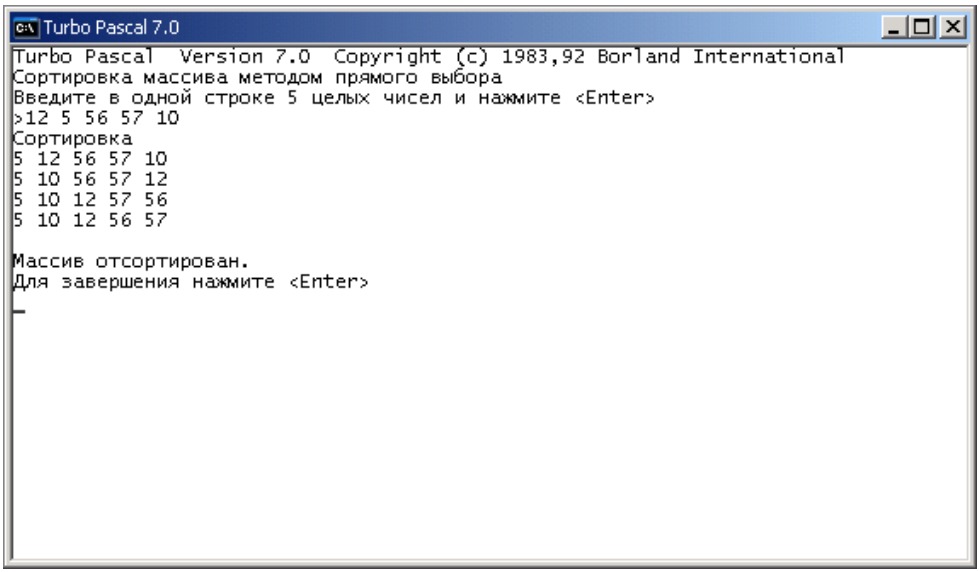


Рис. 4.5. Сортировка массива методом прямого выбора

Сортировка методом прямого обмена

В основе алгоритма лежит идея обмена соседних элементов массива, если следующий элемент меньше предыдущего (при сортировке по возрастанию). Каждый элемент массива, начиная с первого, сравнивается со следующим, и если он больше следующего, то элементы меняются местами. Таким образом, элементы с меньшим значением продвигаются к началу массива ("всплывают"), а элементы с большим значением — к концу массива ("тонут"), поэтому этот метод иногда называют методом "пузырька". Этот процесс повторяется на единицу меньше раз, чем элементов в массиве.

На рис. 4.6 показан пример процесса сортировки массива. Дуги отмечают элементы, которые следует обменять местами на очередном шаге цикла обменов.

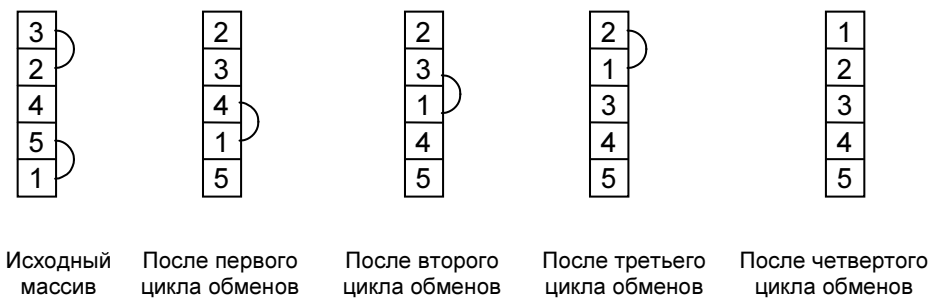


Рис. 4.6. Процесс сортировки массива методом "пузырька"

Программа, реализующая алгоритм сортировки массива по возрастанию методом "пузырька" (прямого обмена), приведена в листинге 4.7. Для демонстрации процесса сортировки состояние массива отображается после выполнения каждого цикла обменов. Пример работы программы показан на рис. 4.7.

Листинг 4.7. Сортировка массива методом "пузырька" (p4_7.pas)

```
{ сортировка массива по возрастанию методом "пузырька" }
program p4_7;
const
     N = 5;
var
        a:array[1 .. N] of integer;
```

```
        i:integer; { счетчик циклов }
        j:integer; { текущий индекс элемента массива }
        buf:integer;
begin
     writeln('Сортировка массива методом "пузырька"');
     write('Введите в одной строке ');
     writeln(N,' чисел и нажмите <Enter>');
     write('>');

     for i:=1 to N-1 do
          read(a[i]);
     readln(a[N]);

     writeln('Сортировка...');
     for i:=1 to N-1 do
       begin
          for j:=1 to N-1 do
             begin
               if a[j] > a[j+1] then
                   begin
                      { обменяем j-й и (j+1)-й элементы }
                      buf := a[j];
                      a[j] := a[j+1];
                      a[j+1] := buf;
                   end;
             end;
          for j:=1 to N do
             write(a[j],' ');
          writeln;
     end;

     writeln;
     writeln('Массив отсортирован');
     write('Для завершения нажмите <Enter>');
     readln;

end.
```

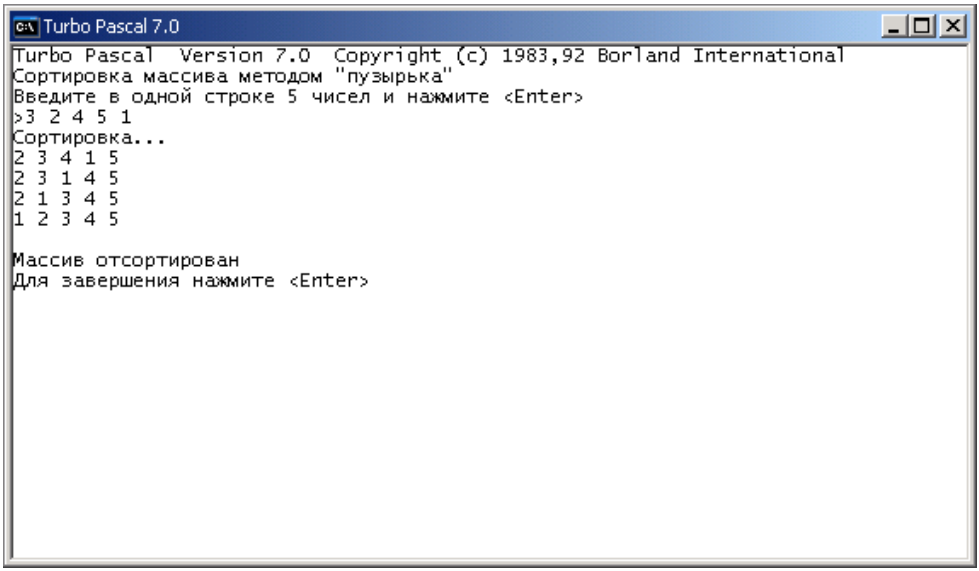


Рис. 4.7. Пример сортировки массива методом "пузырька"

Поиск в массиве

При решении многих задач возникает необходимость установить, содержит массив определенную информацию или нет. Например, проверить, есть ли в массиве фамилий фамилия Петров. Задачи такого типа называются поиском в массиве.

Метод перебора

Для организации поиска в массиве могут быть использованы различные алгоритмы. Наиболее простой — это алгоритм простого перебора. Поиск осуществляется последовательным сравнением элементов массива с образцом до тех пор, пока не будет найден элемент, равный образцу, или не будут проверены все элементы. Алгоритм простого перебора применяется, если элементы массива не упорядочены.

В листинге 4.8 приведен текст программы поиска в массиве целых чисел методом перебора элементов. Перебор элементов массива осуществляет инструкция `repeat`, в теле которой инструкция `if` сравнивает текущий элемент массива с образцом и присваивает переменной `found` значение `TRUE`, если текущий элемент равен образцу. Цикл завершается, если в массиве обнаружен элемент, равный образцу (в этом случае значение переменной `found` равно

TRUE), или если проверены все элементы массива. По завершении цикла по значению переменной found можно определить, успешен поиск или нет. Пример работы программы показан на рис. 4.8.

Листинг 4.8. Поиск в массиве методом перебора элементов (p4_8.pas)

```
program p4_8;
var
     massiv:array[1..10] of integer; { массив целых}
     obrazec:integer; { образец для поиска }
     found:boolean;   { признак совпадения с образцом }
     i:integer;
begin
     { ввод 10 целых чисел }
     writeln('Поиск в массиве');
     write('Введите элементы массива (10 целых чисел в одной строке)');
     writeln ('и нажмите <Enter>');
     write('>');
     for i:=1 to 10 do
          read(massiv[i]);
     { числа введены в массив }
     write('Введите образец для поиска (целое число)-> ');
     readln(obrazec);
     { поиск простым перебором }
     found:=FALSE; { совпадений нет }
     i:=1;         { проверяем с первого элемента массива }
     repeat
          if massiv[i] = obrazec
               then found:=TRUE   { совпадение с образцом }
               else i:=i+1;       { переход к следующему элементу }

     until (i>10) or (found); { завершим, если совпадение
                              { с образцом или проверен }
                              { последний элемент массива }
     if found
          then writeln('Совпадение с элементом номер', i:3,'. ',
                       'Поиск успешен.')
          else writeln('Совпадений с образцом нет.');
end.
```

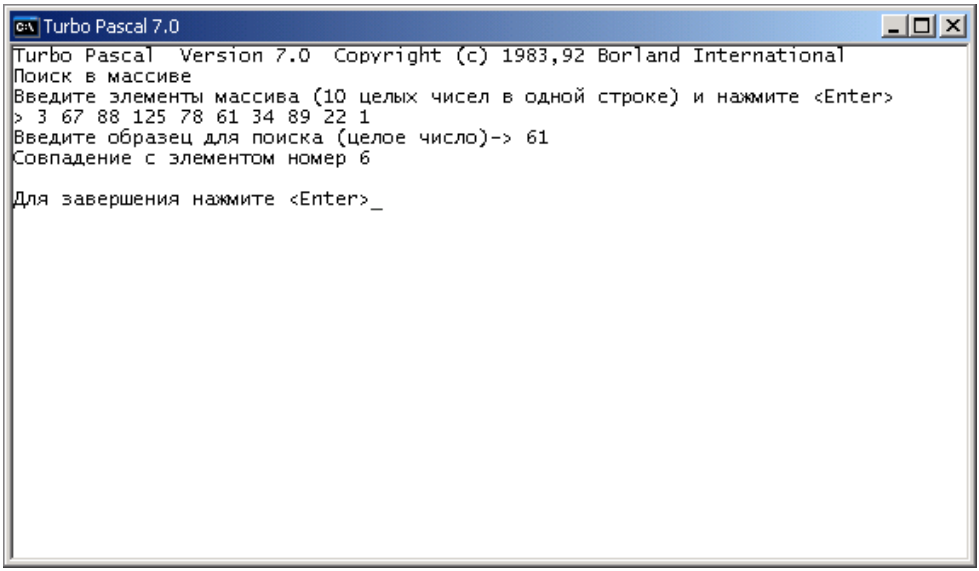


Рис. 4.8. Поиск в массиве методом перебора

Очевидно, чем больше элементов в массиве и чем дальше расположен нужный элемент от начала массива, тем дольше будет программа искать нужный элемент.

Так как операции сравнения применимы как к числам, так и к строкам, то данный алгоритм может использоваться для поиска как в числовых, так и в строковых массивах.

Бинарный поиск

На практике довольно часто проводится поиск в массиве, элементы которого упорядочены по некоторому критерию. Например, массив фамилий, как правило, упорядочен по алфавиту, массив метеорологических данных упорядочен по датам наблюдений.

Для поиска в упорядоченных массивах применяют другие, более эффективные по сравнению с методом простого перебора, алгоритмы, один из которых — метод бинарного поиска.

Суть бинарного поиска заключается в следующем. Выбирается средний (по номеру) элемент упорядоченного массива (элемент с номером `middle`), и образец сравнивается с этим элементом (рис. 4.9).

Если средний элемент равен образцу, то задача решена. Если средний элемент меньше образца (предполагается, что массив упорядочен по возрастанию),

то искомый элемент расположен выше среднего элемента (между элементами с номерами `top` и `middle-1`). Если средний элемент больше образца, то искомый элемент расположен ниже среднего (между элементами с номерами `middle+1` и `bottom`).

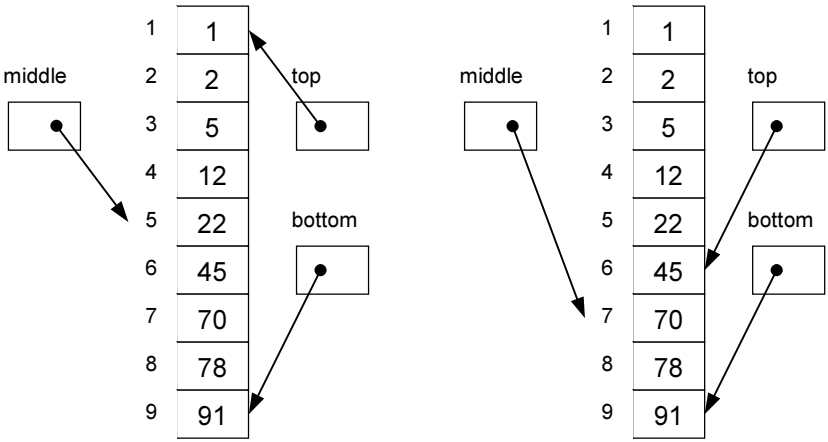


Рис. 4.9. Выбор среднего элемента массива при бинарном поиске

После того как определена часть массива, в которой может находиться искомый элемент, поиск проводят в этой части, выделяя новый средний элемент. Номер среднего элемента вычисляется по формуле `(bottom – top) / 2 + top`. Алгоритм бинарного поиска в упорядоченном массиве представлен на рис. 4.10, программа — в листинге 4.9. В программу добавлены инструкции вывода значений переменных `top`, `botton` и `middle`. Эта информация полезна для понимания сути алгоритма. Пример работы программы приведен на рис. 4.11.

Листинг 4.9. Бинарный поиск в упорядоченном массиве (p4_9.pas)

```
{ бинарный поиск в упорядоченном массиве }
program p4_9;
const
     N = 9;
var
     a: array[1 .. N] of integer; { массив }
     obr:integer; { образец для поиска }

     middle,top,bottom:integer;{ средний, верхний и нижний
                                 элементы массива}
```

```
     found:boolean;{ признак совпадения с образцом }

     k:integer; { счетчик сравнений с образцом }
     i:integer;
begin
     writeln('Бинарный поиск в массиве');
     write('Введите в одной строке ', N, ' целых чисел ');
     writeln('и нажмите <Enter>');

     for i:=1 to N-1 do
          read(a[i]);
     readln(a[N]);

     write('Введите образец для поиска (целое число) ->');
     readln(obr);

     { бинарный поиск }
     top := 1;
     bottom := N;
     found := FALSE;
     k := 0;

     writeln(' top  bottom   middle');
     repeat
          middle := (bottom - top) div 2 + top;
          writeln(top:5, bottom:5, middle:5);
          k := k+1;
          if a[middle] = obr
              then found:=TRUE
              else
                  if obr < a[middle]
                     then bottom := middle -1
                     else top := middle +1;
     until found or (top > bottom);

     if found
          then write('Совпадение с элементом номер ',
                      middle,'. Сравнений: ',k)
```

```
        else writeln('Искомого значения в массиве нет');

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

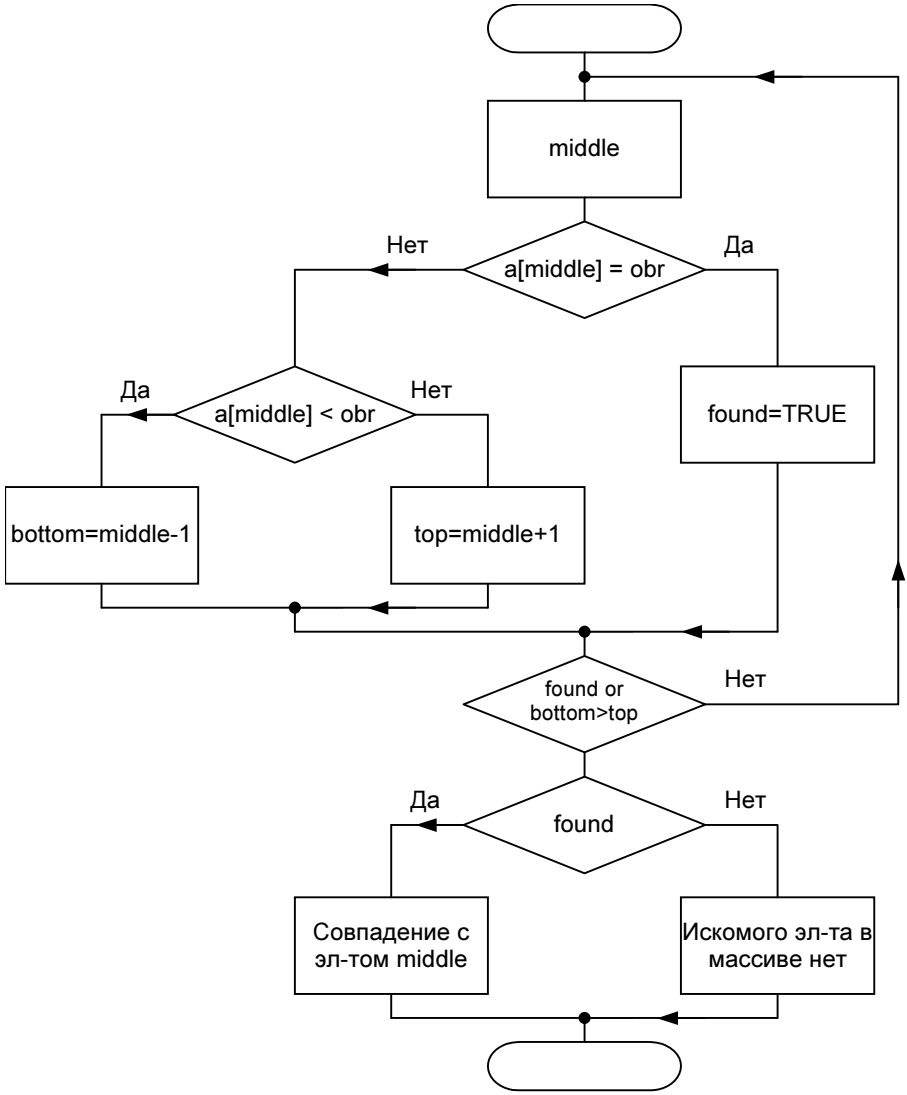


Рис. 4.10. Алгоритм бинарного поиска в упорядоченном по возрастанию массиве

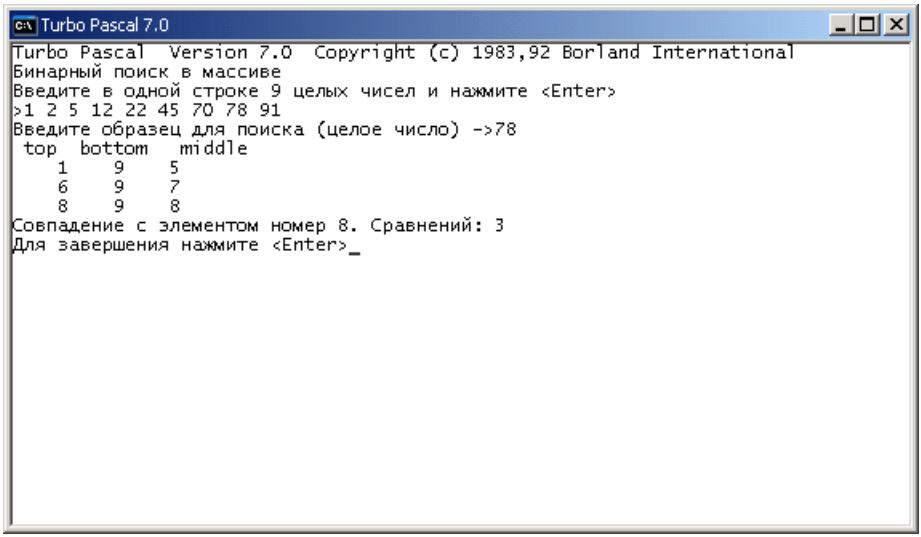


Рис. 4.11. Бинарный поиск в упорядоченном массиве

Многомерные массивы

Исходные данные для решения многих задач часто представляют в табличной форме. Например, результат продаж автомобилей можно представить в виде следующей таблицы:

	I кв.	II кв.	III кв.	IV кв.
Модель_1				
Модель_2				
Модель_3				
Модель_4				
Модель_5				

В программе таблицу можно представить как совокупность массивов. Так, приведенная выше таблица может быть представлена следующим образом:

```
model_1: array[1..4] of integer;
model_2: array[1..4] of integer;
model_3: array[1..4] of integer;
model_4: array[1..4] of integer;
model_5: array[1..4] of integer;
```

Каждый из приведенных массивов хранит информацию о количестве проданных автомобилей одной марки.

Возможно и такое представление таблицы:

```
q_1: array[1...5] of integer;
q_2: array[1...5] of integer;
q_3: array[1...5] of integer;
q_4: array[1...5] of integer;
```

В этом случае каждый массив хранит информацию о количестве проданных автомобилей всех моделей, соответственно, в первом, втором, третьем и четвертом кварталах.

Если данные в таблице одного типа, ее можно представить как двумерный массив.

В общем виде объявление двумерного массива выглядит так:

Имя:**array**[*H1* ..*B1*, *H2* ..*B2*] **of** *Тип*

где: *Имя* — имя массива; **array** — ключевое слово, показывающее, что объявляемый элемент данных является массивом; *H1*, *B1*, *H2*, *B2* — целые константы, определяющие диапазоны изменения индексов и, следовательно, число элементов массива; *Тип* — тип элементов массива.

Приведенную выше таблицу продаж автомобилей можно представить в виде двумерного массива так:

```
nu:array[1..5,1..4] of integer;
```

Количество элементов двумерного массива можно вычислить по формуле:

```
(B1 - H1 + 1) * (B2 - H2 + 1)
```

Таким образом, массив `nu` состоит из 20 элементов типа `integer`.

Чтобы получить доступ к элементу двумерного массива, нужно указать имя массива и индексы элемента, заключив их в квадратные скобки. Первый индекс соответствует номеру строки таблицы, второй — колонки. Так, например, элемент `nu[2,3]` содержит количество автомобилей модели Модель_2, проданных в третьем квартале.

Значения элементов двумерных массивов выводят на экран и вводят с клавиатуры, как правило, по строкам, т. е. сначала все элементы первой строки, затем второй и т. д. Это удобно выполнять при помощи вложенных инструкций `for`. Следующий фрагмент программы выводит на экран массив (значения элементов) по строкам:

```
for i:=1 to 5 do
      begin
```

```
        for j:=1 to 4 do
              writenu[i,j]);
        writeln;
    end;
```

В приведенном фрагменте каждый раз, когда внутренний цикл завершается, внешний цикл увеличивает `i` на единицу, и внутренний цикл выполняется снова. Таким образом все элементы массива `nu` выводятся в следующем порядке: `nu[1,1]`, `nu[1,2]`, ... `nu[1,4]`, `nu[2,1]`, `nu[2,2]`, ... `nu[2,4]` и т. д.

В качестве примера рассмотрим следующую задачу. Пусть есть данные о продажах автомобилей и надо посчитать: общее количество проданных автомобилей, количество автомобилей, проданных в каждом квартале, и количество автомобилей каждой марки, проданных за год (найти сумму элементов массива, сумму элементов по столбцам и по строкам).

Результат обработки таблицы и исходные данные удобно объединить в одну таблицу:

	I кв.	II кв.	III кв.	IV кв.	Всего
Модель_1					
Модель_2					
Модель_3					
Модель_4					
Модель_5					
Всего					

В программе приведенную таблицу можно представить так:

```
nu: array[1..6, 1..5] of integer
```

или, объявив константу `NM` (количество моделей):

```
nu: array[1..NM+1, 1..5] of integer
```

Текст программы, которая решает поставленную задачу, приведен в листинге 4.10, пример ее работы — на рис. 4.12.

Листинг 4.10. Пример работы с двумерным массивом (p4_10.pas)

```
{ пример использования двумерного массива  }
program p4_10;
const
     NM = 5; { количество моделей }
```

```
var
   {  таблица "продажи"}
   nu :array[1 .. NM+1, 1 .. 5] of integer;
   i:integer;
   j:integer;
begin
     writeln('Ввод исходных данных');
     write('Для каждой модели введите в одной строке ');
     writeln('четыре числа и нажмите <Enter>');
     for i:=1 to NM do
         begin
              write('Модель_',i,' >');
              for j:=1 to 3 do
                  read(nu[i,j]);
              readln(nu[i,4]);
         end;

     { вычислим общее количество автомобилей
       каждой марки (суммы в строках ) }
     for i:=1 to NM do
          begin
              nu[i,5]:= 0;
              for j:=1 to 4 do
                   nu[i,5]:= nu[i,5] + nu[i,j];
          end;

     { вычислим количество автомобилей, проданных
       в каждом квартале и общее кол-во (суммы в столбцах) }
     for j:=1 to 5 do
          begin
               nu [NM+1,j]:= 0;
               for i:=1 to NM do
                    nu[NM+1,j] := nu[NM+1,j]+nu[i,j];
          end;

     { вывод итоговой таблицы }
     writeln;
     writeln ('             I     II    III     IV    Всего');
     writeln('-----------------------------------------------');
```

```
    for i:=1 to NM + 1 do
        begin
            if i <= NM
               then write('Модель_',i)
               else begin
                        writeln('---------------------------------');
                        write('Всего:  ');
                    end;
            for j:=1 to 5 do
                write(nu[i,j]:7);
            writeln;
        end;

    writeln;
    writeln('Всего за год: ',nu[NM+1,5]);

    writeln;
    write('Для завершения работы нажмите <Enter>');
    readln;

end.
```

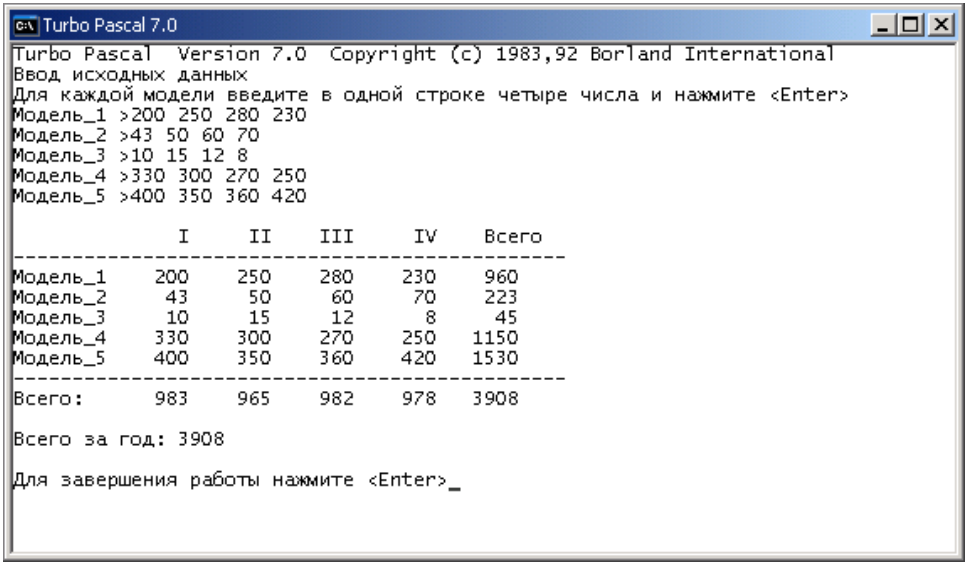


Рис. 4.12. Обработка двумерного массива

Следующая программа, ее текст приведен в листинге 4.11, после вычисления количества продаж автомобилей каждой марки сортирует строки таблицы по содержимому столбца **Всего**. Следует обратить внимание, что исходная таблица в программе представлена двумя массивами. В массиве `model` хранятся названия автомобилей, в массиве `nu` — информация о продажах. Сортировка выполняется методом "пузырька". В качестве буфера при обмене используется последняя строка массива `nu` и дополнительный элемент массива `model` (количество строк в массиве `nu` и количество элементов в массиве `model` на единицу больше, чем количество моделей). Пример работы программы приведен на рис. 4.13.

Листинг 4.11. Сортировка двумерного массива (p4_11.pas)

```
{ сортировка двумерного массива }
program p4_11;
const
     NM = 5; { количество моделей }
var
   { таблица "продажи" }
   model: array[1 ..NM+1] of string; { названия моделей }
   nu :array[1 .. NM+1, 1 .. 5] of integer;

   i:integer;
   j:integer;
   k:integer;

begin
     writeln('Сортировка двумерного массива');
     writeln('Ввод исходных данных');
     for i:=1 to NM do
         begin
              write('Модель>');
              readln(model[i]);   { название модели }
              write('Продажи>');
              for j:=1 to 3 do
                  read(nu[i,j]);
              readln(nu[i,4]);
         end;
```

```
{ вычислим общее количество автомобилей
  каждой марки (суммы в строках) }
for i:=1 to NM do
      begin
          nu[i,5]:= 0;
          for j:=1 to 4 do
               nu[i,5]:= nu[i,5] + nu[i,j];
      end;

{ исходная }
writeln;
writeln (' Модель       I     II    III     IV    Всего');
writeln('-----------------------------------------------');
for i:=1 to NM  do
     begin
         write(model[i]:8);
         for j:=1 to 5 do
               write(nu[i,j]:7);
          writeln;
     end;
writeln('-----------------------------------------------');

{ сортировка строк по содержимому столбца "Всего" }
for i:= 1 to NM -1 do
    for j:=1 to NM-1 do
        if nu[j,5] < nu[j+1,5] then
           begin
                { обменять j-ю и j+1 строки}
                model[NM+1] := model[j];
                model[j] := model[j+1];
                model[j+1] := model[NM+1];
                for k:=1 to 5 do
                    begin
                         nu[NM+1,k] := nu[j,k];
                         nu[j,k] := nu[j+1,k];
                         nu[j+1,k] := nu[NM+1,k];
                    end;
           end;
```

```
    { итоговая таблица }
    writeln;
    writeln (' Модель       I     II    III     IV    Всего');
    writeln('-----------------------------------------------');
    for i:=1 to NM  do
        begin
            write(model[i]:8);
            for j:=1 to 5 do
                  write(nu[i,j]:7);
             writeln;
        end;
    writeln('-----------------------------------------------');

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;

end.
```

```
Turbo Pascal 7.0
Turbo Pascal  Version 7.0  Copyright (c) 1983,92 Borland International
Сортировка двумерного массива
Ввод исходных данных
Модель>a
Продажи>200 250 280 230
Модель>b
Продажи>43 50 60 70
Модель>c
Продажи>10 15 12 8
Модель>d
Продажи>330 300 270 250
Модель>e
Продажи>400 350 360 420

 Модель       I     II    III     IV    Всего
-----------------------------------------------
       a    220    250    280    230    980
       b     43     50     60     70    223
       c     10     15     12      8     45
       d    330    300    270    250   1150
       e    400    350    360    420   1530
-----------------------------------------------

 Модель       I     II    III     IV    Всего
-----------------------------------------------
       e    400    350    360    420   1530
       d    330    300    270    250   1150
       a    220    250    280    230    980
       b     43     50     60     70    223
       c     10     15     12      8     45
-----------------------------------------------

Для завершения нажмите <Enter>_
```

Рис. 4.13. Сортировка двумерного массива

Ошибки при использовании массивов

При работе с массивами наиболее распространенной ошибкой является попытка доступа к несуществующему элементу, выход значения индексного выражения за допустимые границы, указанные при объявлении массива.

Если в качестве индекса указана константа и ее значение выходит за допустимые границы, то такая ошибка обнаруживается на этапе компиляции. Например, если в программе объявлен массив

```
day:array[0..6] of string;
```

то компилятор отметит инструкцию

```
day[7]:='Воскресенье';
```

как ошибочную и выведет сообщение `Constant out of range` (константа за границей допустимого значения).

Если при обращении к элементу массива в качестве индекса используется переменная или выражение, то возможно возникновение ошибки во время выполнения программы (runtime error). Например, в представленной в листинге 4.12 программе синтаксических ошибок нет.

Листинг 4.12. Программа, во время работы которой возникает ошибка (p4_12.pas)

```
{  сортировка массива методом "пузырька" (в программе есть ошибка!)
   демонстрирует возникновение ошибки во время выполнения
   программы  (runtime error) }
program p4_12;
var
   a: array[1 .. 5] of integer;
   b: integer;
   i,j: integer;

begin
   writeln('Введите в одной строке 5 чисел и нажмите <Enter>');
   write('>');

   for i:=1 to 4 do
       read(a[i]);
   readln(a[5]);

   for j := 1 to 5 do
   for i:= 1 to 5 do
```

```
        if a[i] > a[i+1] then
           begin
                b:= a[i];
                a[i] := a[i+1];
                a[i+1] := b;
           end;

    for i :=1 to 5 do
        write(a[i], ' ');
    writeln;

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;

end.
```

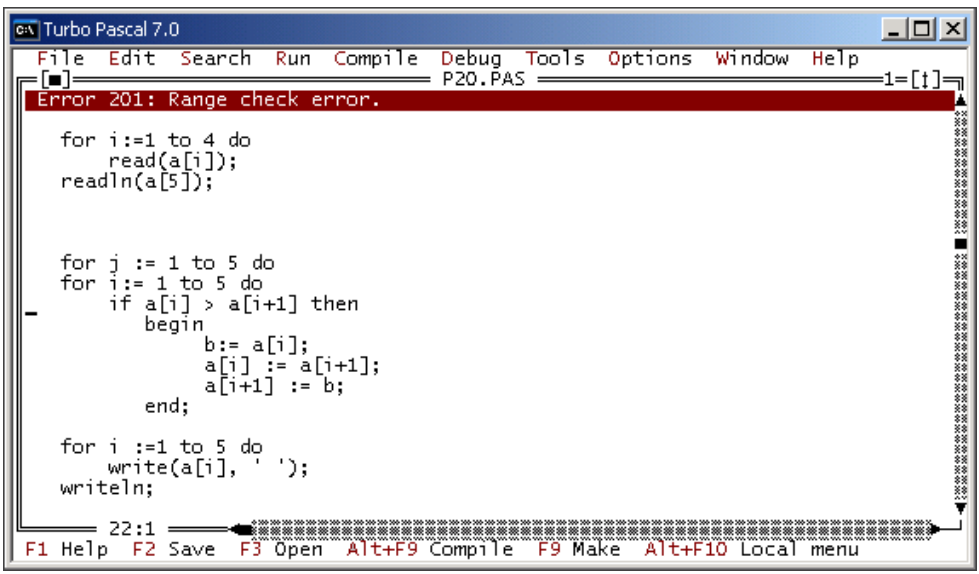


Рис. 4.14. Сообщение об ошибке (программа запущена из среды разработки)

Однако во время работы программы, после ввода данных и нажатия клавиши <Enter> возникает ошибка: в цикле `for`, если `i` равно 5, делается попытка обратиться к шестому элементу массива `a`. При возникновении ошибки, если программа запущена из среды разработки, сообщение (рис. 4.14) отображается

в окне редактора кода (курсор указывает команду, при выполнении которой произошла ошибка). Если программа запущена из операционной системы, то — в окне командной строки (рис. 4.15).

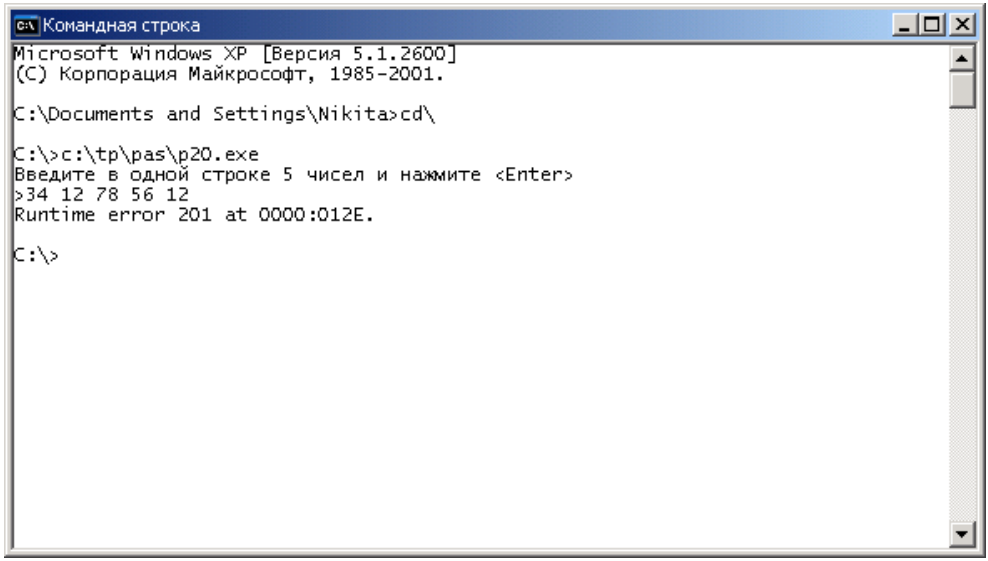
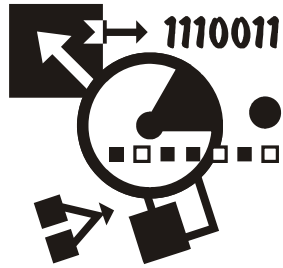


Рис. 4.15. Сообщение об ошибке
(программа запущена из окна командной строки операционной системы)

Следует обратить внимание, что по умолчанию выход индекса за границу допустимого значения не контролируется. Чтобы компилятор добавил в программу код, обеспечивающий контроль значений индексов, следует в меню **Option** выбрать команду **Compiler** и в появившемся окне установить в выбранное состояние переключатель **Range checking**.



Глава 5

Символы и строки

Помимо числовой информации компьютер может обрабатывать символьную. Символьная информация может быть представлена как отдельными символами, так и строками.

Символы

Для хранения отдельных символов используются переменные типа `char`. Значением переменной типа `char` может быть буква, цифра, знак препинания или другой символ.

Переменная символьного типа должна быть объявлена в разделе объявления переменных так:

Имя`:char;`

где `char` — ключевое слово обозначения символьного типа.

Примеры:

```
otv:char;
ch:char;
```

Переменная типа `char` может получить значение в результате выполнения инструкции присваивания или ввода (`read`, `readln`). Если переменная типа `char` получает значение в результате выполнения операции присваивания, то справа от знака `:=` должно стоять выражение типа `char`, например, переменная типа `char` или символьная константа.

Символьная константа — это символ, заключенный в кавычки. Например: 'Y', 'д', '9'. В приведенном примере следует обратить внимание на последнюю константу. Это буква "девять", а не число девять.

В памяти компьютера переменная типа `char` занимает один байт (это значит, что существует 255 различных символов). Каждому символу поставлено

в соответствие число (код), причем код символа '0' меньше кода символа '1', символа 'A' — меньше, чем код символа 'B', а символа 'B', в свою очередь, меньше кода символа 'C'.

Таким образом можно записать:

'0'<'1'<...<'9'<...<'A'<'B'<...<'Z'<'a'<'b'<...<'z'.

Коды символов букв русского алфавита больше кодов букв латинского алфавита, и при этом справедливо следующее:

'А'<'Б'<'В'<...<'Ю'<'Я'<'а'<'б'<'в'<...<'э'<'ю'<'я'.

Как было сказано, символов всего 255. Однако не все символы есть на клавиатуре. Например, на клавиатуре нет символов, с помощью которых рисуются рамки. Если в программе нужно вывести на экран символ, которого нет на клавиатуре, то можно воспользоваться функцией `Chr`, значением которой является символ, код которого указан в качестве параметра. Например, значение Chr(5) — это символ ♣.

Следующая программа, ее текст приведен на листинге 5.1, выводит на экран таблицу кодировки символов (эта кодировка называется ASCII). Результат работы программы приведен на рис. 5.1. Следует обратить внимание, что символы с кодами от 7 до 10 и 13 являются специальными (табл. 5.1) и поэтому при выводе они заменены на пробелы.

Таблица 5.1. *Специальные символы*

Код	Символ	Действие
7	Bell ("звонок")	Звук из динамика
8	Backspace ("забой")	Удаляет символ, который находится перед курсором
9	TAB ("табуляция")	Переводит курсор в следующую позицию строки, номер которой кратен восьми
10	LF ("перевод строки")	Переводит курсор в следующую строку
13	CR ("возврат каретки")	Переводит курсор в начало строки

Листинг 5.1. Таблица символов (p5_1.pas)

```
{ таблица кодировки символов }
program p5_1;
var
     ch:char;       { символ }
```

```
     code:integer; { код символа }
     i: integer;   { номер строки }
     j: integer;   { номер столбца }
     k:integer;    { 0 - первая половина таблицы символов; 1 - вторая }

begin
   writeln('ASCII - таблица символов');
   code:=0;
   for k:=0 to 1 do
   begin
     for i:=0 to 15 do { шестнадцать строк }
          begin
               code:= i + 128*k;
               { k = 0 - первая половина таблицы (0-127),
                 k = 1 - вторая половина таблицы (128-255) }
               for j:=1 to 8 do { восемь колонок }
                    begin
                         if(( code < 7) or ( code > 10)) and (code <> 13)
                              then
                                   write(code:6,'-',' ',
                                       Chr(code):1)
                              else { символы CR,LF,TAB не отображаются }
                                   write(code:6,'-  ');
                         code := code + 16;
                    end;
               writeln;      { переход к новой строке экрана }
          end;

      if k = 0 then
      begin
         writeln;
         write('Для продолжения нажмите <Enter>');
         readln;
      end;
   end;

   writeln;
   write('Для завершения нажмите <Enter>');
   readln;
end.
```

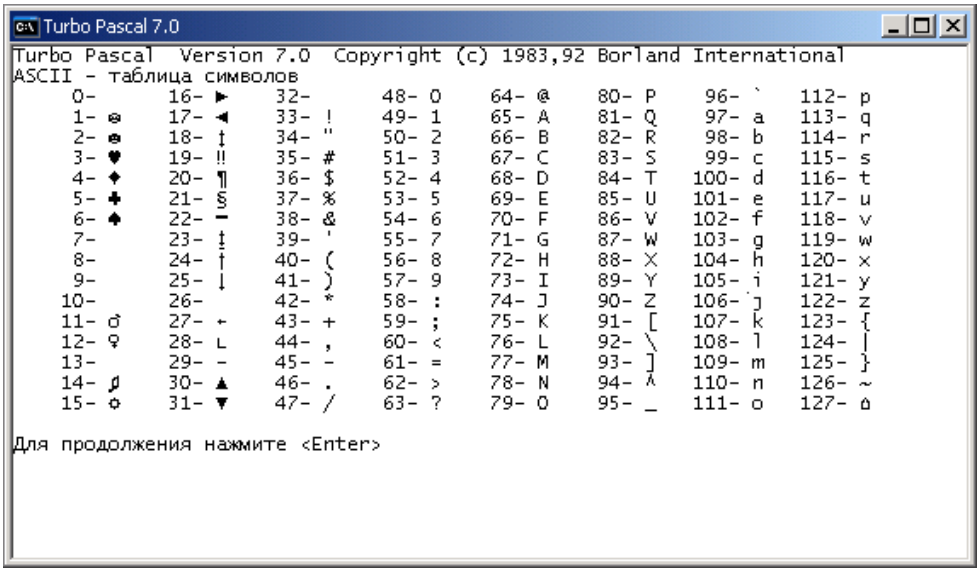


Рис. 5.1. Первая половина ASCII таблицы кодировки символов

Следующая программа, ее текст приведен в листинге 5.2, использует функцию `Chr` для оформления выводимого текста (рис. 5.2).

Листинг 5.2. Использование функции `Chr` (p5_2.pas)

```
program p5_2;
var
   usd:real;  { цена в долларах }
   k: real;   { курс }
   rub: real; { цена в рублях }
   i: integer;
begin
     writeln;
     writeln('Пересчет цены из долларов в рубли');
     writeln('Введите цену, курс и нажмите <Enter>');

     { линия}
     for i := 1 to 45 do
         write(chr(205));
```

```
    writeln;

    write('Цена($)',chr(16));
    readln(usd);
    write('Курс(руб/$)',chr(16));
    readln(k);

    { линия}
    for i := 1 to 45 do
        write(chr(205));
    writeln;

    rub := usd * k;

    writeln('Цена:',rub:9:2,' руб.');
    writeln;

    write('Для завершения нажмите <Enter>');
    readln;
end.
```

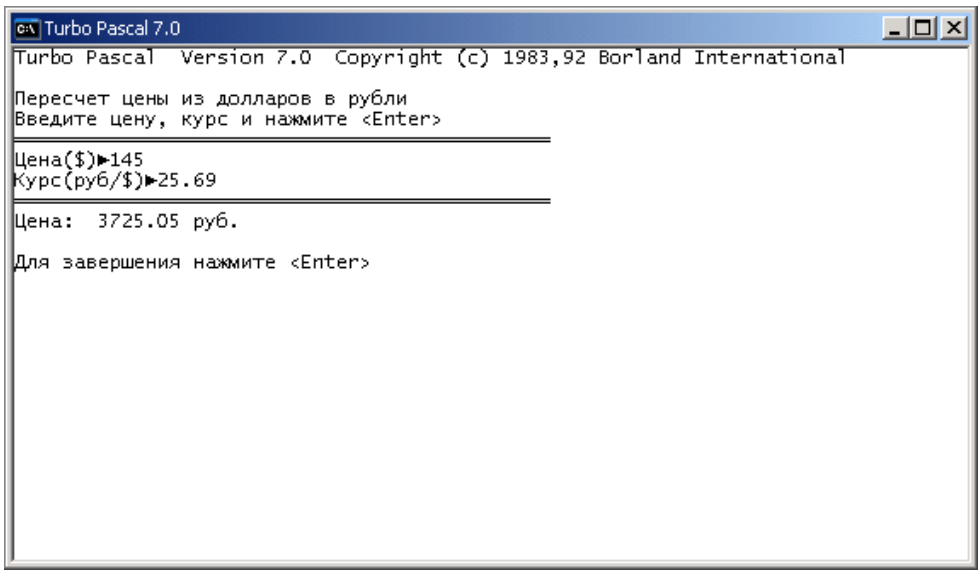


Рис. 5.2. Использование функции `Chr`

Строки

Строка — это последовательность символов. Для хранения строк используются переменные типа `string`.

Объявление строковой переменной в общем виде выглядит так:

Имя: string;

или

Имя: string[*длина*];

где *Имя* — имя переменной; `string` — ключевое слово обозначения строкового типа; *длина* — константа целого типа, которая задает максимально возможную длину строки (количество символов).

Примеры объявления переменных типа `string`:

```
name: string[30];
buff: string;
```

Если в объявлении переменной длина не указана, то по умолчанию длина строки устанавливается равной 255 символам, т. е. объявления

```
stroka: string[255]
stroka: string
```

эквивалентны.

Следует обратить внимание, что параметр *длина*, указанный в объявлении строковой переменной, задает не длину строки как таковую, а количество байт памяти, отводимой для переменной и, следовательно, длину строки, которую можно записать в переменную. Это надо учитывать при объявлении строковых переменных. Рекомендуется в качестве параметра *длина* указать значение, равное максимально возможной длине строки, которая может быть записана в переменную.

Переменная типа `string` может получить значение в результате выполнения инструкции присваивания или ввода значения с клавиатуры. Если переменной присваивается значение, то тип этого выражения должен быть `string`.

Строковая константа — это последовательность символов, заключенная в кавычки. Например: 'Turbo Pascal', 'Санкт-Петербург'.

Переменную типа `string` можно сравнить с другой строковой переменной или константой, используя операторы =, <, >, <=, >=, <>. Строки сравниваются посимвольно от первого символа. Если все символы сравниваемых строк одинаковые, то такие строки считаются равными. Если в одинаковых позициях строк находятся разные символы, большей считается та строка, у которой

в этой позиции находится символ с большим кодом. Ниже приведены примеры сравнения строк.

st 1	st2	Результат сравнения
Иванов	Алексеев	st1 > st2
Иванов	Иванов	st1 = st2
Иванов	Иванов	st1 > st2
Иванов	Петров	st1<st2
Иванов	Иванова	st1 < st2

Для переменных типа `string` определена операция сложения. Результатом ее является новая строка. Например, в результате выполнения инструкций

```
first_name := 'Иванов';
last_name := 'Иван';
ful_name := first_name + ' ' + last_name;
```

значение переменной `ful_name` будет равно `'Иванов Иван'`.

Ввод строк

Для ввода строк с клавиатуры следует использовать инструкцию `readln`. Необходимо обратить внимание, если в объявлении переменной строкового типа указана ее длина, то в переменную будет записано указанное или меньшее количество символов. Если длина не указана (это значит, что значением переменной может быть строка длиной до 255 символов), то в переменную будет записана вся строка, набранная на клавиатуре.

Процесс ввода строк демонстрирует следующий пример. Пусть надо ввести с клавиатуры имя и фамилию в переменные `FirstName` и `LastName`, которые объявлены как строки длиной 15 символов. Если инструкцию ввода записать `readln(FirstName, LastName)`, то пользователь будет должен набрать имя, дополнить его пробелами до 15 символов и затем набрать фамилию, что неудобно. Поэтому, чтобы не заставлять пользователя считать введенные символы, приведенную выше инструкцию `readln` следует заменить двумя: `readln(FirstName)` и `readln(LastName)`. Если по условиям задачи информация должна вводиться в одной строке, то ее сначала следует ввести в промежуточную переменную (буфер), а затем — разделить на отдельные строки (см. пример в описании функции `copy`).

Преобразование строчных букв в прописные

В информационных системах при хранении информации используют прописные буквы. Делается это для того, чтобы избежать путаницы. Например, с точки зрения здравого смысла строки `Петров` и `ПЕТРОВ` — это записанная по-разному фамилия одного и того же человека. Формально же — это две разные строки и, следовательно, это фамилии разных людей. Можно требовать, чтобы оператор вводил информацию прописными буквами. Но можно поступить иначе — возложить на программу задачу преобразования вводимой информации к требуемому виду.

Преобразование строчных букв в прописные основано на том, что код строчной буквы латинского алфавита на 32 больше кода прописной. Также на 32 код прописных букв русского алфавита от "А" до "П" больше кода соответствующих строчных букв. А для букв от "Р" до "Я" разница равна 80. Таким образом, чтобы преобразовать прописную букву в строчную, надо код буквы уменьшить на 32 (латинские и русские от "А" до "П") или 80 (от "Р" до "Я"). Следует обратить внимание, что в Turbo Pascal есть функция `UpCase`, значением которой является строчной символ, соответствующий прописному, указанному в качестве параметра. Однако функция `UpCase` работает только с буквами латинского алфавита.

В листинге 5.3 приведена программа, которая преобразует строчные символы введенной строки в прописные и затем — прописные в строчные. В программе используется функция `Ord`, значением которой является код символа, указанного в качестве ее параметра. Пример работы программы приведен на рис. 5.3.

Листинг 5.3. Преобразование строчных букв в прописные (p5_3.pas)

```
{ преобразование строчных букв в прописные и обратно }
program p5_3;
var
     st : string[80];
     n:integer;       { длина строки }
     i:integer;
begin
     writeln('Введите фамилию и нажмите <Enter>');
     write('-> ');
     readln(st);

     n := length(st);

     { преобразование строчных букв в прописные }
```

```
    for i:=1 to n do
              case st[i] of
                   'a'..'п':st[i]:=chr(ord(st[i])-32);
                   'р'..'я':st[i]:=chr(ord(st[i])-80);
                   else st[i] := UpCase(st[i]);
              end;

    { вывод преобразованной строки }
    writeln;
    writeln(st);

    { преобразование прописных букв (кроме первой) в строчные }
    for i:=2 to n do
        case st[i] of
              'A'..'Z','А'..'П':st[i]:=chr(ord(st[i])+32);
              'Р'..'Я':st[i]:=chr(ord(st[i])+80);
              end;

    writeln;
    writeln(st);

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;

end.
```

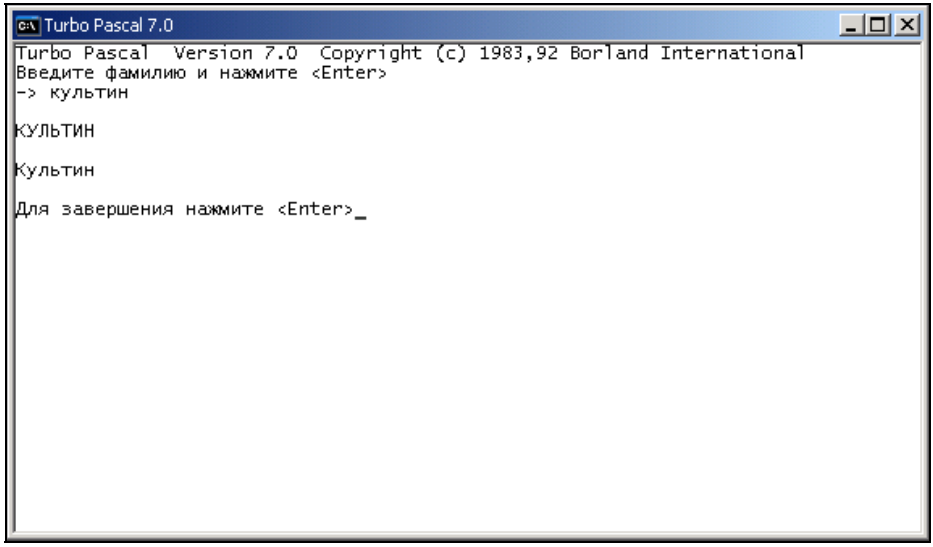


Рис. 5.3. Преобразование строчных букв в прописные

Функции манипулирования строками

В Turbo Pascal есть несколько функций и процедур, позволяющих выполнять операции над строками.

Функция *LENGTH*

Функция `length` позволяет определить длину строки. В качестве параметра надо указать строку (переменную строкового типа), длину которой надо определить. Значение функции (целое число) — количество символов строки.

Следующая программа, ее текст приведен в листинге 5.4, использует функцию `length` для завершения цикла ввода информации в массив.

Листинг 5.4. Ввод символьного массива (p5_4.pas)

```
{ ввод символьного массива}
program p5_4;
const
     N=35;
var
     student:array[1..N] of string[25];
     name:string[30];
     i:integer;
begin
     writeln('Ввод массива строк');
     writeln('После ввода каждой фамилии нажимайте <Enter>');
     writeln('Для завершения просто нажмите <Enter>');
     i:=1;
     repeat
          write('>');
          readln(name);
          if length(name)<>0
               then begin
                    student[i] := name;
                    i:=i+1;
               end;
     until (length(name) = 0) or (i = N);

     { вывод введенного списка }
     writeln;
```

```
    i:=1;
  while (length(student[i])<>0) and (i<=N)  do
        begin
            writeln(i,'. ',student[i]);
            i:=i+1;
       end;

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Программа вводит фамилии и записывает их в массив. Цикл ввода завершается, если введены 35 фамилий или если будет введена строка нулевой длины. Пример работы программы приведен на рис. 5.4.

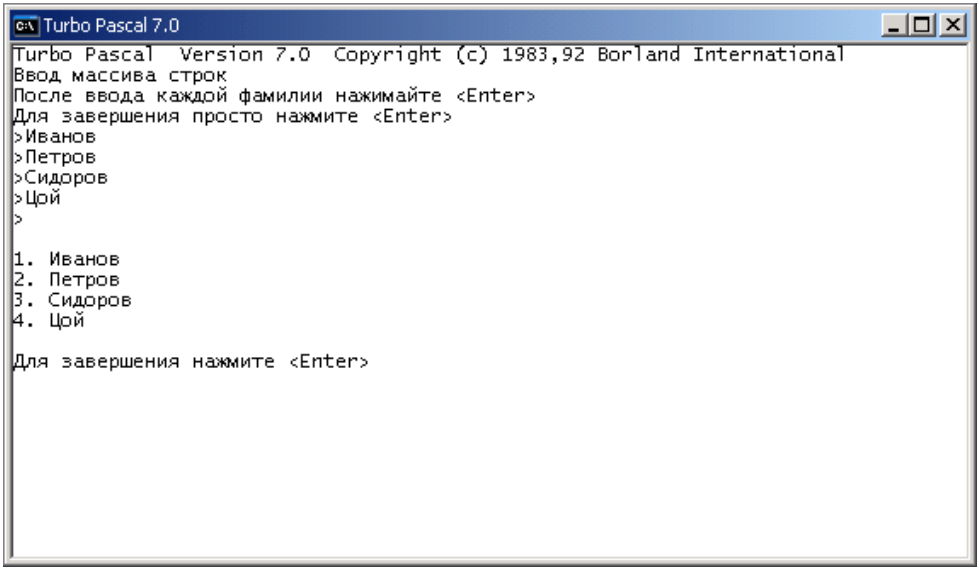


Рис. 5.4. Ввод строк в массив

Процедура *DELETE*

Процедура `delete` позволяет удалить часть строки. В общем виде обращение к процедуре выглядит так:

`delete(`*Строка*`,`*p*`,`*n*`);`

где *Строка* — переменная строкового типа; *p* — номер символа, с которого начинается удаляемая подстрока; *n* — длина удаляемой подстроки.

Например, в результате выполнения инструкций

```
s:='Санкт-Петербург';
delete(s,1,6);
```

значением переменной s будет строка 'Петербург'.

Функция *POS*

Функция pos позволяет определить положение подстроки в строке. В общем виде обращение к функции выглядит так:

```
p := pos(Подстрока,Строка);
```

где *Подстрока* — строковая константа или переменная, которую надо найти в строковой константе или переменной *Строка*.

Например, в результате выполнения инструкции

```
p:=pos('Пе','Санкт-Петербург');
```

значение переменной p будет равно 7.

Если в строке нет искомой подстроки, то функция pos возвращает нуль.

Следующая программа, ее текст приведен в листинге 5.5, использует функцию pos и процедуру delete для удаления пробелов в начале введенной с клавиатуры строки.

Листинг 5.5. Удаление начальных пробелов строки (p5_5.pas)

```
{ Удаление начальных пробелов строки }
program p5_5;
var
     st:string[30];
begin
     write('Введите строку:');
     readln(st);
     while (pos(' ',st) = 1) and (length(st)>0) do
          delete(st,1,1);

     write('Строка без начальных пробелов:',st);

     writeln;
```

```
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Пробелы удаляются в цикле `while` до тех пор, пока функция `pos` обнаруживает пробел в начале строки (значение `pos` при этом равно единице). Необходимость проверки условия `length(st)>0` объясняется тем, что введенная с клавиатуры строка будет состоять только из пробелов. Пример работы программы приведен на рис. 5.5.

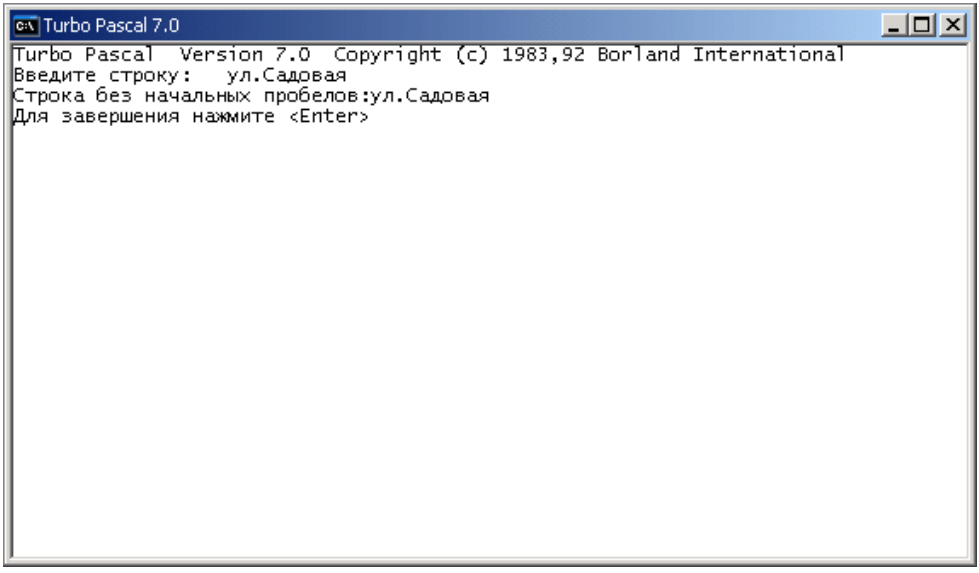


Рис. 5.5. Удаление пробелов в начале строки

Функция *COPY*

Функция `copy` позволяет выделить фрагмент строки (подстроку). В общем виде обращение к функции `copy` выглядит так:

```
st := copy(Строка,p,n);
```

где *st* — переменная, в которую надо записать подстроку; *Строка* — переменная строкового типа, содержащая строку, фрагмент которой надо получить; *p* — номер первого символа в строке *Строка*, с которого начинается выделяемая подстрока; *n* — длина выделяемой подстроки.

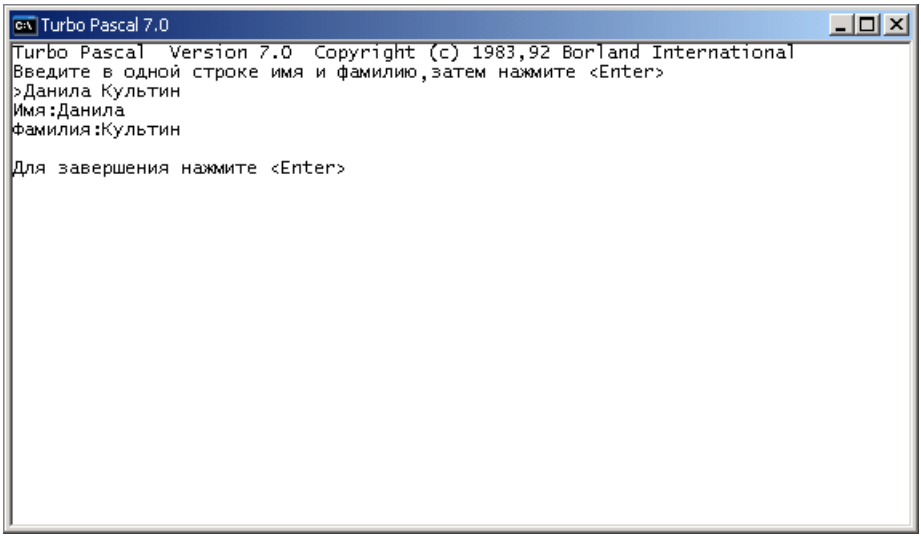


Рис. 5.6. Разделение строки на две

Следующая программа, ее текст приведен в листинге 5.6, демонстрирует использование функций `copy`, `pos` и `length`. Программа показывает, как разделить одну строку на две (выделить из строки подстроку). Пример работы программы приведен на рис. 5.6.

Листинг 5.6. Выделение подстроки (p5_6.pas)

```
program p5_6;
var
   name: string[30];      { полное имя - имя, фамилия }
   f_name: string[30];   { имя }
   l_name: string[30];   { фамилия }

   p: integer;
   l: integer;
begin
    write('Введите в одной строке имя и фамилию, ,');
    writeln('затем нажмите <Enter>');
    write('>');
    readln(name);

    { предполагается, что между именем и фамилией один пробел }
```

```
    p := pos(' ', name);
    f_name := copy(name,1,p-1);
    l := length(name);
    l_name := copy(name,p+1,l);

    writeln('Имя:',f_name);
    writeln('Фамилия:',l_name);

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Процедура *VAL*

Процедура `val` позволяет преобразовать строку (изображение числа) в число. В общем виде инструкция вызова процедуры выглядит так:

```
val(Строка,Имя,Код);
```

где *Строка* — строковая переменная, содержащая изображение числа; *Имя* — переменная целого или дробного типа, в которую надо записать значение, изображение которого находится в переменной *Строка*; *Код* — переменная целого типа, в которую записывается код ошибки. Если строка является правильным изображением числа, то значение *Код* равно нулю. Если строка не может быть преобразована в число из-за того, что строка не является изображением числа, то в переменную *Код* записывается номер неверного символа. Например, после выполнения инструкции

```
val('1,25',n,code)
```

значение переменной `code` будет равно `2`, так как при записи дробных чисел должна использоваться точка, а не запятая.

Процедура `val` полезна при организации ввода с клавиатуры. Например, наиболее распространенной ошибкой при вводе дробных чисел является использование запятой вместо точки. Чтобы избежать ситуации, когда в результате ввода неверных данных программа аварийно завершает работу, можно ввести данные в строковую переменную и затем при помощи процедуры `val` преобразовать введенную строку в число. Если в процессе преобразования возникнет ошибка, то можно попытаться ее устранить — заменить ошибочный символ на правильный. Приведенная в листинге 5.7 программа показывает, как это можно сделать. Следует обратить внимание на то, как выполняется замена символа в строке. Символьная строка — это фактически

массив символов (см. объявление строки). Поэтому доступ к нужному символу можно получить точно так же, как и к элементу массива. Пример работы программы приведен на рис. 5.7.

Листинг 5.7. Преобразование строки в число (p5_7.pas)

```
{ преобразование строки в число }
program p5_7;
var
   st: string[30];  { строка}
   r: real;         { значение  }
   k: integer;      { код ошибки }

   p: integer; { позиция запятой }

begin
     write('Введите дробное число >');
     readln(st);

     val(st,r,k);
     if k <> 0 then
        begin
             { попытаемся исправить ошибку
               вероятно, вместо точки
               пользователь ввел запятую }
            p := pos(',',st);
            if p <> 0 then
               begin
                    st[p] := '.';
                    { повторная попытка преобразования }
                    val(st,r,k);
               end;

        end;

     if k = 0 then
         writeln('Введенное число:',r:6:2)
       else
         writeln('Введенная строка не может быть преобразована в число');
```

```
    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

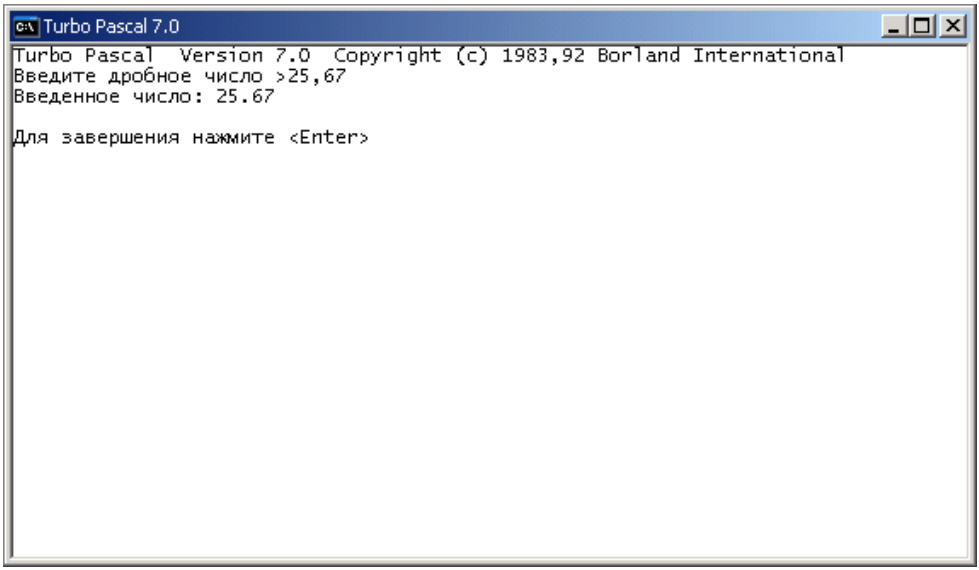
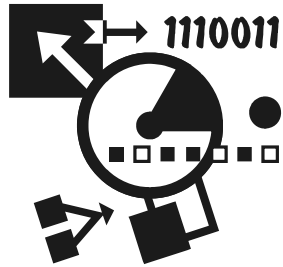


Рис. 5.7. Преобразование строки в число



Глава 6

Процедуры и функции

Большую программу можно разделить по функциональному признаку на части и реализовать каждую часть как *подпрограмму*. Например, программу работы с базой данных можно разделить на следующие подпрограммы: загрузка базы данных, меню, поиск информации, добавление информации, сохранение. Разбивка большой программы на подпрограммы облегчает процесс разработки программы, делает ее структуру более понятной, позволяет работать над одной программой нескольким программистам (каждый разрабатывает свою часть общей программы).

Подпрограмма — это небольшая программа, которая обеспечивает решение некоторой подзадачи (части большой задачи). В языке Pascal существует два типа подпрограмм — функции и процедуры.

Функция

Функция — это подпрограмма. Для того чтобы функция была выполнена, ее надо *вызвать* — указать имя функции в качестве операнда выражения. Например, стандартная функция `sqrt` вычисляет квадратный корень числа, указанного в качестве ее параметра. Значение функции связано с ее именем. Поэтому для того, чтобы вычислить значение квадратного корня из числа, находящегося в переменной `x`, следует записать: `y:=sqrt(x)`.

В общем виде инструкция обращения к функции выглядит так:

Переменная`:=`*Функция*`(`*Параметры*`);`

где *Переменная* — переменная, которой надо присвоить значение, вычисленное функцией; *Функция* — имя функции; *Параметры* — список параметров, которые используются для вычисления значения функции. В качестве параметров обычно используют переменные или выражения.

Примеры использования (вызова) стандартных функций:

```
y:=Abs(x);
n:=Trunc(n/100);
x:= Round(d);
```

Следует обратить внимание на то, что:

- тип переменной, которой присваивается значение функции, должен соответствовать типу функции (типу значения, которое возвращает функция);
- тип и количество параметров для каждой функции строго определены.

Различают следующие типы функций:

- Стандартные
- Библиотечные
- Программиста

Стандартные функции

Стандартными называют функции, которые доступны "по умолчанию". Некоторые из стандартных функций Turbo Pascal приведены в табл. 6.1.

Таблица 6.1. *Некоторые стандартные функции*

Функция	Тип функции	Значение
`Abs(x)`	Тот же, что и тип аргумента	Абсолютное значение x
`Sqrt(x)`	`real`	Квадратный корень x
`Sqr(x)`	Тот же, что и тип аргумента	Квадрат x
`Sin(α)`	`real`	Синус угла α (величина угла должна быть указана в радианах)
`Cos(α)`	`real`	Косинус угла α (величина угла должна быть указана в радианах)
`ArcTan(x)`	`real`	Арктангенс x
`Exp(x)`	`real`	Экспонента x
`Ln(x)`	`real`	Натуральный логарифм x
`Round(x)`	`integer`	Ближайшее к x целое
`Trunc(x)`	`integer`	Целая часть x
`Frac(x)`	`integer`	Дробная часть вещественного x, представленная как целое число
`Int(x)`	`integer`	Целая часть вещественного x, представленная как целое число

Библиотечные функции

Библиотечной называют функцию или процедуру, которая становится доступной в результате подключения к программе библиотеки (модуля), в которой эта функция находится. Например, процедура `TextColor` модуля Crt позволяет установить цвет текста, выводимого на экран инструкцией `write` (`writeln`). Чтобы библиотечная функция стала доступной, модуль, в котором она находится, надо подключить — добавить в текст программы инструкцию `uses`, указав в качестве параметра имя модуля. В качестве примера в листинге 6.1 приведена программа, которая использует процедуры модуля `Crt`. Процедура `ClrScr` очищает экран (закрашивает цветом, заданным процедурой `TextBackground`).

Листинг 6.1. Использование процедур модуля Crt (p6_1.pas)

```
program p6_1;

uses Crt; { сделать доступным модуль Crt}

var
   fnt: real; { вес в фунтах }
   kg: real;  { вес в граммах }

begin
     TextBackground(Blue);  { фон - синий  }
     TextColor(Lightgray);  { текст - серый }

     ClrScr; { очистить экран }

     writeln('Пересчет веса из фунтов в килограммы');
     writeln('Введите вес в фунтах и нажмите <Enter>');
     write('>');
     readln(fnt);
     kg := fnt *0.409;
     writeln;
     writeln(fnt:6:2,' ф. - ',kg:6:2,' кг.');

     write('Для завершения нажмите <Enter>');
     readln;
end.
```

Следует обратить внимание, что модуль `Crt`, который входит в Turbo Pascal 7.0, не работает на компьютерах с процессором Pentium (Celeron): при попытке запустить программу, в которой есть директива `uses Crt`, на экране появляется сообщение **Runtime error 200: Division by Zero**. Чтобы устранить эту ситуацию, следует использовать Turbo Pascal 7.1.

Функция программиста

Turbo Pascal позволяет программисту объявить свою собственную функцию и в дальнейшем использовать ее точно так же, как и стандартные функции. Например, можно определить (объявить) функцию вычисления факториала, назвав ее `Factorial`. Затем в том месте программы, где нужно вычислить факториал, вместо последовательности инструкций, делающих это, написать `k:= Factorial(n);`.

Объявление функции

Объявление функции в общем виде выглядит так:

```
function Имя(Параметры):Тип;
var
   { здесь объявления внутренних переменных функции}
begin
   { здесь инструкции функции}
   Имя := Выражение;
end;
```

где:

`function` — зарезервированное слово языка Turbo Pascal, обозначающее, что далее следуют инструкции, реализующие функцию;

Имя — имя функции;

Параметры — список переменных, которые используются для передачи в функцию информации, необходимой для вычисления значения функции;

Тип — тип значения функции.

Следует обратить внимание, что функция завершается инструкцией, которая присваивает значение идентификатору *Имя*. Именно эта инструкция определяет значение функции.

В качестве примера в листинге 6.2 приведена функция `Factorial`, которая вычисляет факториал числа, указанного в качестве ее параметра.

Листинг 6.2. Функция вычисления факториала (p6_2.pas)

```
function Factorial(n:integer):longint;
   var
     f:longint; { факториал числа n }
     i:integer;
   begin
     f:=1;
     for i:=2 to n do
        f := f*i;
     Factorial := f;
   end;
```

У функции `Factorial` один параметр — переменная `n` типа `integer`. Параметр задает число, факториал которого надо вычислить. Конкретное значение `n` получит при вызове функции. Возвращает функция вычисленное значение факториала — число типа `longint`.

Другой пример. В Turbo Pascal нет стандартной функции вычисления кубического корня. В листинге 6.3 приведено ее возможное определение.

Листинг 6.3. Функция программиста для вычисления кубического корня (p6_3.pas)

```
function cubrt(x:real):real;
var
     pr:real;     { приближенное значение кубического корня}
begin
     pr:=sqrt(x); { первое приближение }
     while abs(pr-x/(pr*pr))>0.001 do
     begin
          {новое приближение - среднее арифметическое }
          {удвоенного приближения на прошлом шаге и текущего}
          pr:=(2*pr+x/(pr*pr))/3;
     end;
     cubrt:=pr;
end;
```

Функция `cubrt` реализует алгоритм "ручного" метода приближенного вычисления кубического корня, при котором в качестве первого приближения, "на глаз", выбирается наиболее подходящее число (в программе это значение

квадратного корня числа, кубический корень которого надо вычислить). Затем число, из которого надо извлечь корень, делится на выбранное приближение, и полученное таким образом значение еще раз делится на выбранное приближение. Если полученное число отличается от выбранного приближения на величину, не большую, чем допустимое значение погрешности, то выбранное приближение принимается за значение корня. Если погрешность превышает допустимое значение, то вычисляется новое приближение как среднее арифметическое удвоенного предыдущего приближения и нового вычисленного приближения, и процесс вычисления повторяется.

Использование функции

Для того чтобы функцию, созданную программистом, можно было использовать, ее (в простейшем случае) следует поместить в текст программы, перед разделом объявления переменных.

Инструкции функции будут выполнены, если в каком-либо из выражений программы в качестве операнда будет указано имя этой функции (переход от инструкций основной процедуры программы к инструкциям функции называется *вызовом функции* или *обращением к функции*). Если в объявлении функции указаны параметры (эти параметры называются формальными), то после имени функции также должны быть указаны параметры (эти параметры называются фактическими), причем количество и тип фактических параметров должно соответствовать количеству и типу формальных параметров. В качестве фактических параметров обычно используют переменные или константы, реже — выражения.

Следующая программа, ее текст приведен в листинге 6.4, демонстрирует использование функции `Factorial`. Она выводит таблицу факториалов (рис. 6.1). Следует обратить внимание, что значение факториала числа 13 больше максимального значения типа `longint`, поэтому при попытке вычислить значение факториала 13 возникает ошибка Arithmetic owerflow (переполнение).

Листинг 6.4. Пример использования функции программиста (p6_4.pas)

```
program p6_4;
{ подпрограмма (функция) }
function Factorial(k:integer):longint;
    var
       f: longint;
       i: integer;
    begin
        f:=1;
        for i:=2 to K do
```

```
            f:=f*i;
        Factorial := f;
    end;

{ основная программа }
var
    n: integer; { верхняя граница диапазона }
    i: integer; { число, факториал которого надо вычислить }
    f: longint; { факториал }

begin
    writeln('Введите целое число от 1 до 12');
    write('>');
    readln(n);

    for i:=1 to n do
        begin
            f := Factorial(i);
            writeln(i:2,' ',f);
        end;

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

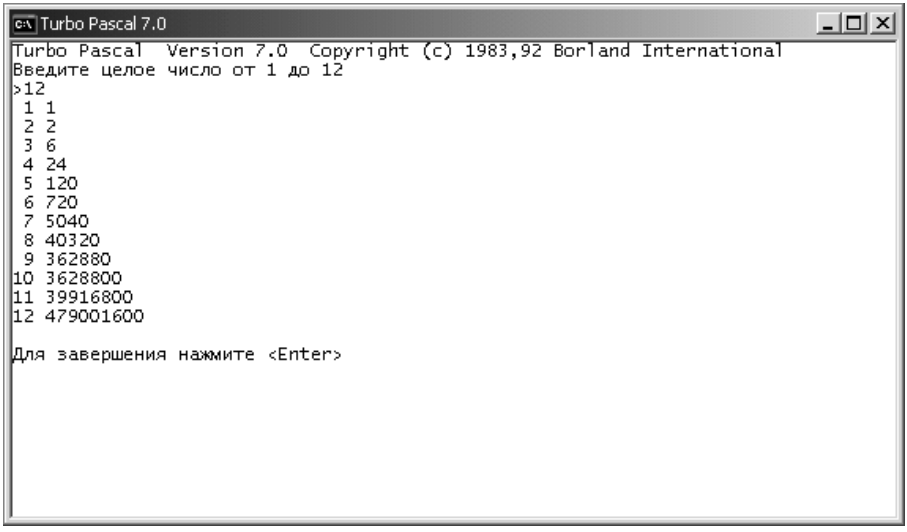


Рис. 6.1. Таблица факториалов

Процедура

Процедура — это разновидность подпрограммы, т. е. имеющая имя последовательность инструкций, предназначенная для решения некоторой задачи.

Для того чтобы процедура была выполнена, ее надо вызвать — указать имя в тексте программы. Например, инструкция вызова стандартной процедуры `Randomize` (инициализации генератора случайных чисел) выглядит так:

```
Randomize;
```

Если у процедуры есть параметры, то их надо указать после имени процедуры. Например, инструкция вызова стандартной процедуры преобразования строки в число выглядит так:

```
Val(s,v,k);
```

Процедура программиста

Программист может создать свою собственную процедуру и использовать ее в программе точно так же, как и другие стандартные процедуры.

Объявление процедуры

В общем виде объявление процедуры выглядит так:

```
procedure Имя(Параметры);
var
    { здесь объявления внутренних переменных процедуры}

begin

{ здесь инструкции процедуры }

end;
```

где:

`procedure` — зарезервированное слово языка Turbo Pascal, обозначающее, что далее следуют инструкции, реализующие процедуру;

Имя — имя функции;

Параметры — список переменных, которые используются для передачи в процедуру информации, необходимой для вычисления значения функции, а также для передачи информации из процедуры в вызвавшую ее программу.

В листинге 6.5 приведена процедура `Line`, которая рисует на экране линию (строку символов). У процедуры два параметра: `n` задает длину линии (количество символов), `c` — символ, которым рисуется строка.

Листинг 6.5. Процедура Line (p6_5.pas)

```
procedure Line(n:integer;c:char);
     var
          i:integer;
     begin
          for i:=1 to n do write(c);
          writeln;
     end;
```

Параметры процедуры могут использоваться не только для передачи информации в процедуру, но и для возврата результата. Если параметр используется для возврата результата из процедуры, перед его именем следует указать слово `var`. Здесь следует обратить внимание, что в этом случае изменение значения переменной-параметра внутри процедуры приводит к изменению значения переменной основной программы, указанной в качестве формального параметра в инструкции вызова процедуры.

В листинге 6.6 приведена процедура `Profit`, которая вычисляет доход по вкладу. Параметр `pr` используется для возврата вычисленного значения дохода.

Листинг 6.6. Процедура Profit (p6_6.pas)

```
procedure Profit(sum:real; percent:real; period:integer; var pr:real);
begin
     pr :=  sum * percent/365 *period;
end;
```

Вызов процедуры

Для того чтобы процедура была выполнена, ее надо вызвать. Инструкция вызова процедуры в общем виде выглядит так:

Имя(*Параметры*);

где:

Имя — имя вызываемой процедуры;

Параметры — разделенные запятыми фактические параметры. Количество и тип фактических параметров должны соответствовать количеству и типу формальных параметров, указанных в объявлении процедуры. В качестве формального параметра можно указать выражение (в простейшем случае

константу или переменную) или, если перед именем параметра стоит слово var, переменную. Например, инструкции вызова приведенной выше процедуры Line могут быть такими:

```
Line(30,'-');
Line(n+2,'*');
Line(n,ch);
```

Следующая программа, ее текст приведен в листинге 6.7, выводит таблицу степеней двойки. Для оформления таблицы используется процедура Line. Результат работы программы приведен на рис. 6.2.

Листинг 6.7. Пример использования процедуры программиста (p6_7.pas)

```
{ Пример использования процедуры программиста }
program p6_7;

{ процедура }
procedure Line(n: integer; c: char);
     var
          i:integer;
     begin
          for i:=1 to n do write(c);
          writeln;
     end;

{ основная программа }
var
   n: integer; { показатель степени }
   k: integer; { степень двойки }

   i: integer;

begin
    writeln('Таблица степеней двойки');
    Line(20,'=');
    writeln('Показатель  Степень');
    Line(20,'=');

    k := 1;
    for i:=1 to 10 do
```

```
        begin
            k := k * 2;
            writeln(i:6,'  ', k:8);
        end;
    Line(20,'-');

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

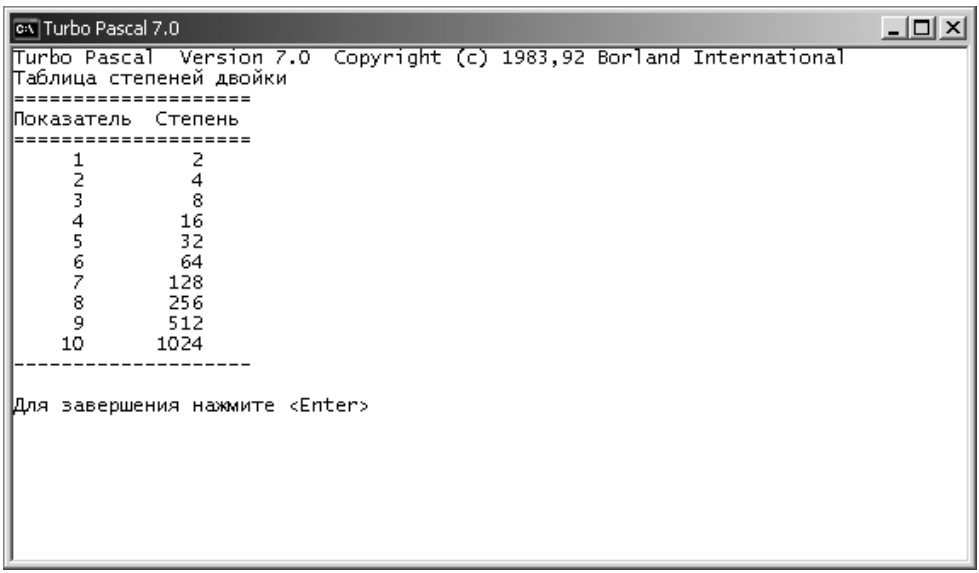


Рис. 6.2. Таблица степеней двойки (линии нарисованы процедурой Line)

Для рисования двойной и одинарной линий использована одна и та же процедура `Line`, но вызывается она с разными фактическими параметрами. Параметры обеспечивают универсальность процедуры. Процедура `Line` может рисовать линии любой длины и из любых символов.

Параметр-переменная и параметр-значение

Параметры, указанные в заголовке описания (объявления) процедуры или функции, называются *формальными*. Параметры, указанные в инструкции вызова процедуры или функции, называются *фактическими*. Количество

и тип фактических параметров должны соответствовать количеству и типу формальных параметров (если соответствие нарушено, компилятор выводит сообщение о ошибке).

В общем случае в качестве фактического параметра может выступать константа, переменная или выражение соответствующего типа. Однако если в описании процедуры или функции перед именем формального параметра указано слово `var`, то в качестве фактического параметра можно использовать *только* переменную соответствующего типа. Использование переменной в качестве фактического параметра позволяет процедуре (функции) изменить значение переменной основной программы, указанной в качестве фактического параметра.

В следующей программе, ее текст приведен в листинге 6.8, для вычисления объема полого цилиндра используется процедура `SCircle`. Параметр `r` (радиус) используется для передачи информации в процедуру, параметр `s` — (площадь круга) для возврата вычисленного значения.

Листинг 6.8. Пример использования процедуры программиста (p6_8.pas)

```
{ пример использования процедуры программиста }
program p6_8;

{  подпрограмма (процедура) }
procedure SCircle(r: real; var s: real);
begin
     s := PI * r*r;
end;

{ основная программа }
var
   r1: real; { радиус цилиндра }
   r2: real; { радиус полости }
   h: real; { высота цилиндра }

   s1: real; { площадь круга в основании цилиндра }
   s2: real; { площадь полости в основании }

   v: real; { объем полого цилиндра }

begin
   writeln('** Объем полого цилиндра **');
```

```
   write('Радиус основания цилиндра (см) ->');
   readln(r1);
   write('Радиус полости (см)->');
   readln(r2);
   write('Высота цилиндра (см) ->');
   readln(h);

   if r1 > r2  then
      begin
           SCircle(r1,s1);
           Scircle(r2,s2);
           v := (s1-s2)*h;
           writeln('Объем цилиндра: ',v:9:2,' см.куб.');
      end
   else
   writeln('Ошибка: радиус полости должен быть меньше радиуса
основания');

   write('Для завершения нажмите <Enter>');
   readln;
end.
```

Типичной ошибкой при создании процедур является отсутствие слова `var` перед именем переменной, используемой для возврата результата из процедуры. Если слово `var` не указано, то изменение значения переменной внутри процедуры не оказывает никакого влияния на значение переменной главной процедуры, указанной в качестве параметра в инструкции вызова процедуры (уберите слово `var` перед параметром `s` процедуры `SCircle` в приведенной выше программе, запустите программу и посмотрите, как она будет работать).

Локальные и глобальные переменные

Переменные, объявленные внутри процедуры или функции, называются *локальными*. Локальные переменные доступны только инструкциям той подпрограммы (процедуры или функции), в которой они объявлены. Следует обратить внимание, что переменные с одинаковыми именами, но объявленные в разных подпрограммах, — это разные переменные. Также необходимо отметить, что значение локальной переменной после завершения подпрограммы теряется.

Объявления переменных основной программы (раздел `var`) можно поместить до и после объявления процедур и функций (листинг 6.9). В первом случае переменные будут доступны внутри всех процедур и функций. Такие переменные называются *глобальными*, во втором — только инструкциям основной программы.

Листинг 6.9. Структура программы (локальные и глобальные переменные)

```
program p;

var
 {здесь объявления глобальных переменных программы}

procedure pr1(Параметры);
    var
        { здесь объявления локальных переменных процедуры pr1 }
    begin
        { последовательность инструкций}
    end;

function f1 (Параметры) : Тип;
    var
         { здесь объявления локальных переменных функции f1 }
    begin
        { последовательность инструкций}
    end;

{ основная программа }
var
     {здесь объявления глобальных переменных, доступных
      только инструкциям основной программы}
begin
        { последовательность инструкций}
end.
```

Глобальные переменные обычно используют для хранения информации, которая необходима многим подпрограммам. Следует обратить внимание, что процедура или функция может получить доступ к глобальной переменной только в том случае, если внутри этой процедуры не объявлена переменная с таким же именем.

Процедура или функция?

Процедура и функция — это два способа оформления подпрограммы или фрагмента программы, предназначенного для решения части общей задачи. Одну и ту же подпрограмму можно оформить как процедуру или как функцию.

Ниже в листинге 6.10 и листинге 6.11 приведены функция и процедура, которые вычисляют доход по вкладу.

Листинг 6.10. Функция Profit (p6_10.pas)

```
function Profit(sum:real; percent:real; period:integer) : real;
    begin
        profit :=  sum * percent/365 *period;
    end;
```

Листинг 6.11. Процедура Profit (p6_11.pas)

```
procedure Profit(sum:real; percent:real; period:integer; var pr:real);
    begin
        pr :=  sum * percent/365 *period;
    end;
```

Очевидно, что оформление подпрограммы в виде функции в данном случае более предпочтительно, так как целью вычисления является получение только одного значения. Кроме того, инструкция обращения к функции

```
p := Profit(s,pr,per);
```

интуитивно более понятна, чем инструкция

```
Profit(s,pr,per,p);
```

вызова процедуры.

Таким образом, при выборе способа оформления подпрограммы следует придерживаться следующего правила: если подпрограмма должна изменить значение только одной переменной основной программы, то ее следует оформить как функцию, в остальных случаях подпрограмму следует оформлять как процедуру.

Структурное программирование

Структурное программирование — метод разработки программ, в основе которого лежит идея разделения общей задачи на подзадачи, а также использование ограниченного набора алгоритмических структур при составлении алгоритма.

Разделение задачи на подзадачи позволяет получить программу, структура которой соответствует структуре решаемой задачи. При разработке структуры программы (алгоритма верхнего уровня) используется технология, получившая название "сверху вниз". Суть ее состоит в том, что сначала создается программа, реализующая общий алгоритм, а решение частных задач откладывается на более поздний срок (процедуры, предназначенные для решения отложенных задач, заменяются процедурами-заглушками). Затем реализуются части программы, обеспечивающие решение частных задач.

Метод структурного программирования предполагает использование ограниченного набора алгоритмических структур при разработке (составлении) алгоритма решения задачи. Алгоритмическими структурами, которые рекомендуется использовать при составлении алгоритмов, являются рассмотренные в *главе 3* структуры: следование, выбор (ветвление), цикл с фиксированным числом повторений, циклы с пред- и постусловием. Такой подход позволяет наиболее просто перевести алгоритм решения задачи на язык программирования.

Применение метода структурного программирования (технологии "сверху вниз") рассмотрим на примере. Пусть надо разработать программу работы с базой данных. Задача работы с базой данных может быть разделена на три подзадачи: отображение меню; добавление информации в базу данных и получение (поиск) нужной информации. Задачу добавления информации можно разделить на три подзадачи: получение данных от пользователя; проверка правильности данных; непосредственное добавление информации в базу данных. Аналогичным образом можно разделить задачу получения информации на подзадачи: получение запроса от пользователя; поиск в базе данных; отображение результата поиска. Алгоритм верхнего уровня программы работы с базой данных можно изобразить так, как показано на рис. 6.3. Можно заметить, что он представляет собой совокупность структур Следование и Множественный выбор и цикл с постусловием.

Задачу отображения меню, а также задачу добавления информации в базу данных и задачу поиска информации следует реализовать как подпрограммы (отображение меню — функция, работа с базой данных — процедура). В этом случае основная программа будет выглядеть так, как показано в листинге 6.12. Следует обратить внимание, что процедуры добавления и поиска информации реализованы как заглушки.

Приведенная программа — это ядро будущей программы, выполняющей все поставленные задачи. Она может быть оттранслирована и выполнена. И хотя, на первый взгляд, она не выполняет никакой полезной работы, с ее помощью может быть проверена логика работы разрабатываемой программы.

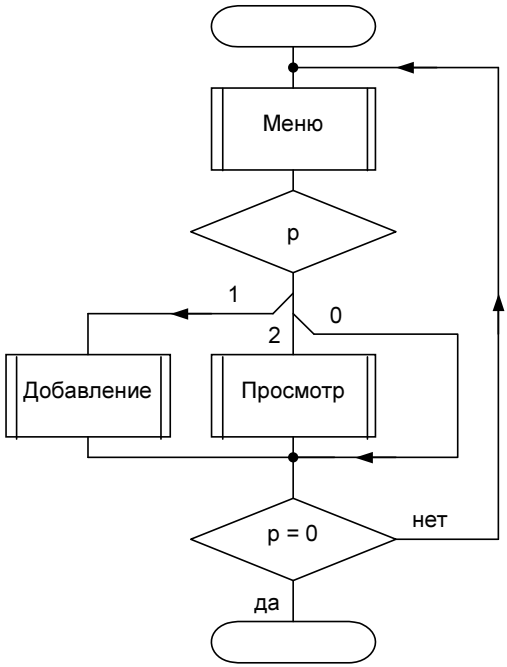
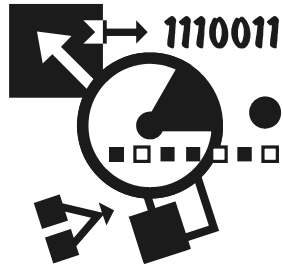


Рис. 6.3. Алгоритм программы работы с базой данных

Листинг 6.12. Программа работы с базой данных (db.pas)

```
program db;

uses Crt;

function Menu: integer;
    var
        p: integer;
    begin
        ClrScr;
        writeln;
        writeln('База данных');
        writeln;
        writeln('1 - Добавление информации');
        writeln('2 - Просмотр информации');
        writeln('0 - Выход');
```

```
        writeln;
        write('Ваш выбор > ');
        readln(p);
        Menu := p;
    end;

{ добавление информации }
procedure AddDB;
    begin
        writeln('Операция добавления еще не реализована');
        writeln('Для продолжения нажмите <Enter>');
        readln;
    end;

{ просмотр информации }
procedure ShowDb;
    begin
        writeln('Операция просмотра еще не реализована');
        writeln('Для продолжения нажмите <Enter>');
        readln;
    end;

{ главная программа }
var
    p: integer;  { пункт меню, выбранный пользователем }
begin
    TextColor(LIGHTGRAY);
    TextBackground(BLUE);
    repeat
        p := Menu;     { отобразить меню }
        case p of
           1: AddDB;  { добавление информации в БД }
           2: ShowDb; { отображение информации }
        end;
    until p = 0;
end.
```



Глава 7

Стандартные модули

Помимо встроенных процедур и функций, Turbo Pascal предоставляет программисту ряд функций и процедур различного назначения, которые по функциональному признаку объединены в библиотеки (модули). Модуль — это совокупность функций, процедур, констант и типов, предназначенных для решения задач определенного класса. Например, модуль `Crt` содержит функции и процедуры управления экраном в текстовом режиме, модуль `Graph` — процедуры и функции отображения графики.

Доступ к библиотечным функциям и процедурам

По умолчанию библиотечные процедуры недоступны. Поэтому, например, если в программу поместить инструкцию вызова процедуры очистки экрана (`ClrScr`), то компилятор выведет сообщение об ошибке в программе: "неизвестный идентификатор" (Unknown identifier).

Чтобы библиотечная процедура или функция стала доступной, необходимо "подключить" модуль, в котором находится эта процедура или функция. Делается это при помощи инструкции `uses`, которая в общем виде выглядит так:

`uses` *Модуль*$_1$`,`*Модуль*$_2$`, ... ,`*Модуль*$_k$`;`

где `uses` — слово языка программирования, обозначающее, что далее следуют имена подключаемых модулей; *Модуль*$_i$ — имя подключаемого модуля.

Следующая программа, ее текст приведен в листинге 7.1, демонстрирует подключение модуля `Crt`.

Листинг 7.1. Подключение модуля Crt (p7_1.pas)

```
program p7_1;

uses Crt; { "подключить" модуль Crt }

var
    d: real;  { длина в дюймах }
    sm: real; { длина в сантиметрах }

begin
    ClrScr; { очистить экран }

    write('Введите длину в дюймах ->');
    readln(d);
    sm := d * 2.54;
    writeln(d:4:2,' дм = ',sm:4:2,' см');
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Модуль Crt

Модуль `Crt` содержит функции, процедуры и константы, полезные при выводе информации на экран. Например, при помощи процедуры `TextColor` можно задать цвет символов текста, выводимого на экран.

Управление курсором

Положение курсора на экране характеризуется номером строки и номером позиции (знакоместа) в строке, которые можно рассматривать как координаты курсора. В стандартном режиме на экране можно отобразить 25 строк текста длиной до 80 символов. Таким образом, горизонтальная координата курсора (позиция в строке) может меняться от 1 до 80, а вертикальная (номер строки) — от 1 до 25. Строки экрана (рис. 7.1) нумеруются сверху вниз, позиции в строке — слева направо. Таким образом, левая верхняя точка экрана имеет координаты (1,1), правая нижняя — (80,25).

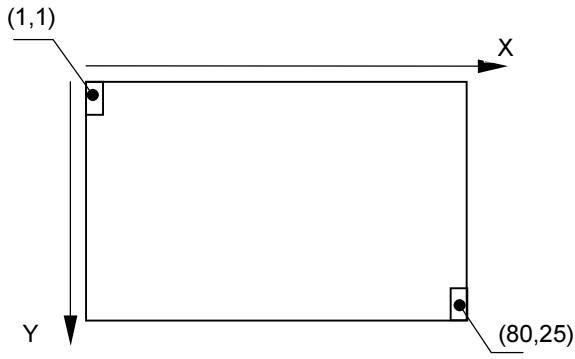


Рис. 7.1. Система координат монитора

Инструкции `write` и `writeln` выводят текст с той позиции, в которой в данный момент находится курсор. Если текст нужно вывести, начиная с определенной позиции экрана, то перед выполнением инструкции вывода курсор надо переместить в эту позицию. Сделать это можно с помощью процедуры `GoToXY`, инструкция вызова которой в общем виде выглядит так:

```
GoToXY(КоординатаX,КоординатаY);
```

где: *КоординатаX*, *КоординатаY* — выражения целого типа, значения которых задают новое положение курсора.

Например, инструкция

```
GoToXY(40,13);
```

перемещает курсор в центр экрана.

Следующая программа, ее текст приведен в листинге 7.2, показывает, как переместить курсор в центр экрана и вывести текст в центр экрана.

Листинг 7.2. Вывод текста в центр экрана (p7_2.pas)

```
program p7_2;
uses Crt;
var
   st: string;
   x,y: integer; { координаты курсора }
begin
    st := 'Turbo Pascal';
    ClrScr;
```

```
    x := 40 - Round( Length(st) / 2);
    y := 12;
    GotoXY(x,y);
    write(st);

    readln;
end.
```

Управление цветом

Процедура `TextColor` позволяет задать цвет символов, выводимых инструкциями `write` и `writeln`, а процедура `TextBackground` — цвет фона, на котором эти символы отображаются.

Чтобы установить цвет символов, надо вызвать процедуру `TextColor`, указав в качестве параметра код цвета (табл. 7.1). Например, инструкция `TextColor(Green)` устанавливает зеленый цвет текста.

Таблица 7.1. *Кодировка цвета*

Цвет	Код	
	десятичная константа	именованная константа
Черный	0	Black
Синий	1	Blue
Зеленый	2	Green
Бирюзовый	3	Cyan
Красный	4	Red
Сиреневый	5	Magenta
Коричневый	6	Brown
Светло-серый	7	LightGray
Темно-серый	8	DarkGray
Голубой	9	LightBlue
Светло-зеленый	10	LightGreen
Светло-бирюзовый	11	LightCyan
Алый	12	LightRed
Светло-сиреневый	13	LightMagenta
Желтый	14	Yellow
Белый	15	White

Аналогичным образом устанавливается цвет фона. Следует обратить внимание, что значение параметра процедуры `TextBackground` не должно быть больше семи.

Иногда необходима информация о текущем цвете символов и фона, например, для того, чтобы программа (или процедура), перед завершением своей работы, могла восстановить цвета, которые она изменила. Информация о текущем цвете символов и фона хранится в глобальной переменной `TextAttr` (младшие четыре бита хранят код цвета символов, следующие три — цвет фона). Если перед тем как изменить цвет символов или фона, сохранить значение переменной `TextAttr`, то в любой момент можно будет восстановить исходные цвета. Следующая программа (листинг 7.3) показывает, как это можно сделать.

Листинг 7.3. Изменение атрибутов текста (p7_3.pas)

```
Program p7_3;
uses Crt;
var
     ta: byte; { атрибуты текста (цвет фона и символов) }
begin
     { сохраним цвет символов и фона }
     ta := TextAttr;

     { установим свои цвета }
     TextBackGround(Blue);
     TextColor(LightGray);

     ClrScr; {очистить экран (закрасить цветом TextBackground) }

     { инструкции программы }

     { восстановим цвет символов и фона }
     TextAttr := ta;
end.
```

Переменную `TextAttr` можно также использовать и для быстрого изменения цвета символов и фона. Для этого ей надо присвоить значение, в качестве которого удобно использовать двухразрядную шестнадцатеричную константу (старший разряд определяет цвет фона, младший — цвет символов). Например,

чтобы установить синий фон и желтый цвет символов, в переменную TextAttr надо записать шестнадцатеричное число 1E. Так как в изображении шестнадцатеричных чисел используются буквы латинского алфавита от A до F, то для того чтобы компилятор различал шестнадцатеричные числа (константы) и имена переменных, перед шестнадцатеричным числом ставится символ $. Так, инструкция

```
TextAttr:=$1E;
```

задает синий фон (номер цвета 1) и желтый цвет символов (номер цвета 14 в десятичной форме, и E — в шестнадцатеричной).

Если к сообщению надо привлечь особое внимание, то можно установить его атрибуты так, что оно будет мигать. Чтобы это сделать, надо к константе, определяющей атрибуты, прибавить 128. Следующая последовательность инструкций выводит мигающее предупреждающее сообщение.

```
TextAttr:=$1E+128;
write('Ошибка данных');
```

Очистка экрана

Процедура ClrScr очищает экран — закрашивает его текущим цветом фона (заданным процедурой TextBackground) и устанавливает курсор в начало первой строки.

Например, в результате выполнения инструкций

```
TextBackground(Blue);
ClrScr;
```

экран закрашивается синим цветом.

Ввод символа с клавиатуры

Функция ReadKey (читать клавишу) возвращает символ (тип char), соответствующий нажатой клавише. Если ни одна клавиша не нажата, то функция ждет до тех пор, пока какая-либо клавиша будет нажата. При использовании функции ReadKey символ нажатой клавиши на экране не отображается.

Используя функцию ReadKey, можно обрабатывать нажатия не только алфавитно-цифровых клавиш, но и служебных: функциональных (<F1>—<F12>), клавиш перемещения курсора, листания текста страницами (PageUp, PageDown) и других. При нажатии служебной клавиши функция ReadKey возвращает 0. Для того чтобы получить номер нажатой служебной клавиши, нужно еще раз вызвать ReadKey.

В листинге 7.4 приведена программа, которая позволяет определить код нажатой клавиши. Программа завершает работу при нажатии клавиши <Esc>.

Листинг 7.4. Отображение кода нажатой клавиши (p7_4.pas)

```
program p7_4;
uses crt;
var
     ch:char;     { символ }
     code: byte; { код символа }
begin
       writeln('Нажмите любую клавишу');
       writeln('Для завершения нажмите <Esc>');
       repeat
          ch:=ReadKey;
          if ch <> chr(0) then
                  write(ch)
          else { нажата служебная клавиша }
               begin
                    writeln('Служебная клавиша ');
                    ch:=ReadKey;
               end;
          code := ord(ch);
          writeln(code:6);
     until ord(ch)=27;      { пока не нажата клавиша Esc }
end.
```

Следующая программа, ее текст приведен в листинге 7.5, использует функцию `ReadKey` для организации меню. Программа выводит список задач (пунктов меню), выделяя название первой задачи цветом. При нажатии клавиши "стрелка вниз" выделяется следующий пункт меню, при нажатии клавиши "стрелка вверх" — предыдущий. Нажатие клавиши <Enter> активизирует процедуру, связанную с выбранным пунктом (в данном случае — процедуру-заглушку).

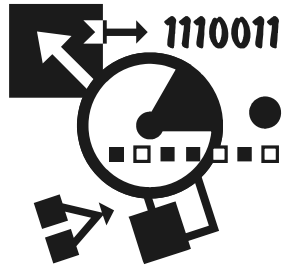
Листинг 7.5. Использование модуля Crt для организации меню (p7_5.pas)

```
program p7_5;
uses Crt;
const
    SEL=$70;    { цвет выбранного пункта }
```

```
    NORM=$17; { цвет невыделенного пункта }
    N=3;        { количество команд в меню}

    menu:array[1..N] of string[12] =
     ('Задача 1','Задача 2','Выход');{ названия пунктов меню }

Procedure pr1;
begin
     ClrScr;
     writeln('Процедура pr1');
     writeln('Нажмите <Enter> для продолжения');
     readln;
end;

Procedure pr2;
begin
     ClrScr;
     writeln('Процедура pr2');
     writeln('Нажмите <Enter> для продолжения');
     readln;
end;

Procedure ShowMenu(x,y,p: integer); { вывод меню на экран }
var i:integer;
begin
     ClrScr;
     for i:=1 to N do begin
          GoToXY(x,y+i-1);
          write(menu[i]);
     end;
     TextAttr:=SEL;
     GoToXY(x,y+p-1);
     write(menu[p]); { выделим строку меню }
     TextAttr:=NORM;
end;

{ программа }
var

     p:integer;    { номер выделенного пункта }
```

```
    ch:char;       { введенный символ }
    x,y:integer;   { координаты первой строки меню }



begin
    menu[1]:=' Действие 1 ';
    menu[2]:=' Действие 2 ';
    menu[3]:=' Выход      ';
    p:=1;
    x:=5;
    y:=5;
    TextAttr:=NORM;

    ShowMenu(x,y,p);
    repeat
        ch:=ReadKey;
        if ch=chr(0) then begin
            ch:=ReadKey;
            case ch of
                chr(80): { стрелка вниз }
                    if p<N then begin
                        GoToXY(x,y+p-1);
                        write(menu[p]);
                        p:=p+1;
                        TextAttr:=SEl;
                        GoToXY(x,y+p-1);
                        write(menu[p]);
                        TextAttr:=NORM;
                    end;
                chr(72):{ стрелка вверх }
                    if p>1 then begin
                        GoToXY(x,y+p-1);
                        write(menu[p]);
                        p:=p-1;
                        TextAttr:=SEl;
                        GoToXY(x,y+p-1);
                        write(menu[p]);
```

```
                              TextAttr:=NORM;
                         end;
               end;
          end
          else
               if ch=chr(13) then begin { нажата клавиша <Enter> }
                    case p of
                         1:pr1;  { вызов процедуры pr1 }
                         2:pr2;  { вызов процедуры pr2 }
                         3:ch:=chr(27);{ выход }
                    end;
                    ShowMenu(x,y,p);
               end;
     until ch=chr(27); { 27 - код <Esc> }
end.
```



Глава 8

Модуль программиста

Программист может создать свой собственный модуль, поместить в него функции, процедуры, объявления констант и затем использовать этот модуль точно так же, как и стандартные модули Turbo Pascal.

Структура модуля

В листинге 8.1 приведена структура модуля. Модуль состоит из раздела интерфейса, раздела реализации и раздела инициализации.

Листинг 8.1. Структура модуля

```
unit ИмяМодуля;

interface
   { объявления типов, констант, переменных, процедур
     и функций, которые могут использоваться в программах,
     использующих данный модуль }

implementation
   { объявления типов, констант, переменных, которые
     используются процедурами и функциями модуля }

   { инструкции реализации процедур и функций модуля }

begin
   { инструкции инициализации переменных модуля }
end.
```

Начинается модуль заголовком, который состоит из слова **unit** и имени модуля.

Слово **interface** отмечает начало раздела интерфейса. В этот раздел помещают объявления типов, констант, переменных, процедур и функций, которые будут доступны программам, использующим данный модуль.

Раздел реализации начинается словом **implementation**. В этот раздел помещают инструкции, реализующие процедуры и функции модуля, объявления локальных констант, типов и переменных.

Раздел инициализации начинается словом **begin**. В этот раздел помещают инструкции инициализации переменных модуля.

В листинге 8.2 приведен модуль, который содержит функции, полезные при работе со строками.

Листинг 8.2. Модуль программиста (p8_2.pas)

```
unit p8_2;

interface

  function LTrim(st:string):string; { удаляет пробелы в начале строки }
  function RTrim(st:string):string; { удаляет пробелы в конце строки }
  function Trim(st:string):string; _{ удаляет пробелы в начале и в конце
строки }
  function Upper(st:string):string; { преобразует к верхнему регистру }

implementation

  function LTrim(st:string):string;
      begin
        while (pos(' ',st) = 1) and (length(st)>0) do
           delete(st,1,1);
        LTrim:=st;
      end;

  function RTrim(st:string):string;
      var
         i: integer;
      begin

        i := Length(st);
        while (st[i] = ' ') and (i > 0) do
              begin
                   Delete(st,i,1);
```

```
                i := Length(st);
            end;
        RTrim:=st;
    end;

function Trim(st:string):string;
    var
       i: integer;
    begin
      {  удалить пробелы в начале строки }
      while (pos(' ',st) = 1) and (length(st)>0) do
         delete(st,1,1);

      { удалить пробелы в конце строки }
      i := Length(st);
      while (st[i] = ' ') and (i > 0) do
            begin
                Delete(st,i,1);
                i := Length(st);
            end;

      Trim:=st;
    end;

function Upper(st:string):string;
    var
       buf:string;
       n: integer; { длина строки }
       i:integer;
  begin
       buf:= st;
       n:=Length(buf);

       for i:=1 to n do
          begin
             case buf[i] of
                'a'..'z','а'..'п' : buf[i]:= Chr(Ord(buf[i])-32);
                'р'..'я' :          buf[i]:= Chr(Ord(buf[i])-80);
        end;
          end;
       Upper:=buf;
    end;

end.
```

Функция `LTrim` удаляет пробелы, которые находятся в начале строки. Ее значением является строка, полученная в качестве параметра, но без ведущих пробелов. Функция `RTrim` удаляет пробелы, которые находятся в конце строки, а функция `Trim` — в начале и в конце. Функция `Upper` преобразует строчные символы строки в прописные, причем она правильно работает как с символами латинского алфавита, так и русского.

Подготовка текста модуля

Процесс создания модуля ничем не отличается от процесса создания программы. Текст модуля надо набрать в окне редактора текста и сохранить в файле с расширением pas.

Компиляция модуля

Компилируется модуль точно так же, как и обычная программа, т. е. выбором в меню **Compile** команды **Compile**. Однако в результате компиляции модуля будет создан не файл программы (файл с расширением exe), а *модуль* — файл с расширением tpu (tpu — сокращение от Turbo Pascal Unit). Созданный компилятором модуль будет помещен в каталог, имя которого указано в поле **EXE&TPU** диалогового окна **Directories**, которое появляется при выборе в меню **Options** команды **Directories**.

Использование модуля

Чтобы программа могла использовать модуль (функции и процедуры, которые находятся в модуле), модуль надо подключить — указать имя модуля в директиве `uses`. Например, для того чтобы программа могла использовать функции, которые находятся в модуле `nk_unit`, в ее начало надо поместить инструкцию `uses nk_unit`.

В качестве примера в листинге 8.3 приведена программа, которая демонстрирует использование модуля программиста. Результат работы программы (функций `Upper`, `LTrim`, `RTrim` и `Trim`) приведен на рис. 8.1. (Для того чтобы можно было видеть, что программа действительно удаляет пробелы, выводимые строки заключены в двоеточия.)

Листинг 8.3. Использование модуля программиста (p8_3.pas)

```
Program p8_3;
    uses nk_unit;

    var
       st: string;

    begin

       write('Введите строку>>');
       readln(st);

       writeln;
       writeln(':', Upper(st), ':');
       writeln(':', LTrim(st), ':');
       writeln(':', RTrim(st), ':');
       writeln(':', Trim(st), ':');

       writeln;
       write('Для завершения нажмите <Enter>');
       readln;
    end.
```

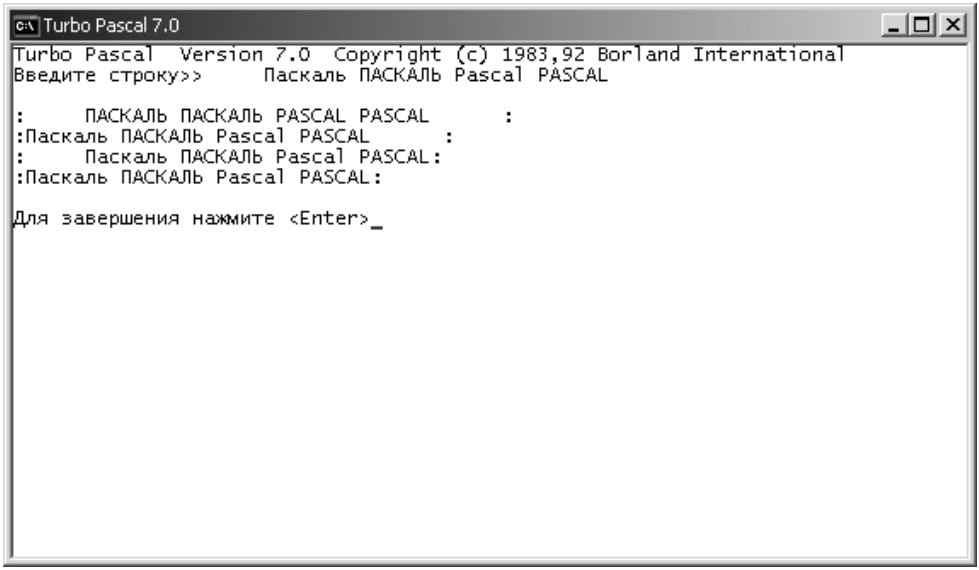
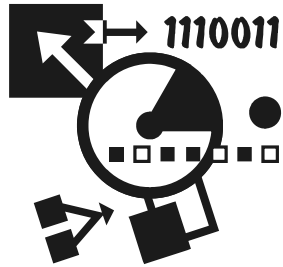


Рис. 8.1. Обработка строки функциями `Upper`, `LTrim`, `RTrim` и `Trim`



Глава 9

Файлы

До этого момента наши программы вводили исходные данные с клавиатуры и выводили результат своей работы на экран, что не всегда удобно. В этой главе рассказывается, как вывести данные в файл, а также о том, как прочитать данные из файла.

Объявление файла

Файл — это структура данных, представляющая собой последовательность элементов одного типа. Количество элементов файла практически не ограничено.

Различают текстовые и двоичные файлы. Текстовый файл — это совокупность символов, его можно просмотреть, загрузив в редактор текста. Увидеть, что находится в двоичном файле, можно только при помощи специальной программы, предназначенной для просмотра файлов соответствующего типа.

Как и любая переменная программы, файл должен быть объявлен в разделе объявления переменных. В объявлении файла указывается тип элементов файла. В общем виде объявление файла выглядит так:

Имя: **file of** *ТипЭлементов*

Примеры**:**

```
f: file of char;       { файл символов }
f: file of real;       { файл вещественных чисел }
f: file of integer;    { файл целых чисел }
```

Файл, компонентами которого являются символы (`char`), называется *текстовым*. Описание **`file of`** `char` можно заменить на `text`, т. е. объявление `f:text` эквивалентно объявлению `f:` **`file of`** `char`.

Назначение файла

Объявление файла (точнее, файловой переменной) задает только тип компонентов файла. Чтобы программа получила доступ к файлу, необходимо указать имя файла. Имя файла задается путем вызова процедуры `assign`, инструкция вызова в общем виде выглядит так:

```
assign( Файл, Имя_Файла);
```

где *Файл* — файловая переменная; *Имя_Файла* — имя файла, к которому надо получить доступ.

В инструкции вызова процедуры `assign` рекомендуется указывать полное имя файла (диск, каталог, имя и расширение файла). Необходимо обратить внимание, имя файла должно быть указано в соответствии с правилами записи имен файлов в MS-DOS (можно использовать только буквы латинского алфавита и цифры; количество символов в имени каталога, подкаталога или файла не должно быть больше восьми).

Ниже приведены примеры вызова процедуры `assign`.

```
assign(f,'a:\test.txt');
assign(f,'c:\test\spb.txt');
```

Открытие файла

Чтобы программа могла прочитать данные из файла или записать данные в файл, файл необходимо *открыть*. В языке Pascal есть несколько инструкций, обеспечивающих открытие файла. Инструкция `reset` открывает файл для чтения данных, `append` — для записи в режиме добавления, `rewrite` — для перезаписи.

Закрытие файла

Перед завершением работы программа должна закрыть все открытые файлы. Делается это вызовом процедуры `close`. В качестве параметра процедуры `close` следует указать файловую переменную, связанную с файлом, который надо закрыть. Пример использования процедуры:

```
close(f);
```

Запись в файл

Для того чтобы программа могла записать данные в файл, файл надо *открыть* для записи.

Если программа, формирующая выходной файл, уже работала, то возможно, что файл с результатами работы программы уже есть на диске. Поэтому программист должен решить, что делать с существующим файлом (заменить старые данные новыми или добавить новые данные к старым) и выбрать соответствующий способ работы с файлом (вариант открытия файла).

Возможны следующие варианты открытия файла для записи:

- перезапись (запись нового файла поверх существующего или, если файл не существует, создание нового файла);
- добавление данных в существующий файл.

Чтобы открыть файл в режиме перезаписи, необходимо вызвать процедуру `rewrite`, указав в качестве параметра файловую переменную.

Чтобы открыть файл в режиме добавления информации, необходимо вызвать процедуру `append`, указав в качестве параметра файловую переменную.

Непосредственно запись (вывод) информации в файл выполняет инструкция `writeln`, у которой в качестве первого параметра указана файловая переменная. Например, если переменная `f` является файловой (объявлена в разделе `var` как `text` или `file of char`), то в результате выполнения инструкции

```
write(f, x);
```

в файл, связанный с переменной `f`, будет записано значение переменной `x`.

Следующая программа, ее текст приведен в листинге 9.1, демонстрирует процесс записи информации в файл. Программа открывает файл в режиме перезаписи (если файла на диске нет, файл будет создан) и записывает в него 5 строк, введенных пользователем. Следует обратить внимание, что сформированный DOS-программой (программа, созданная в Turbo Pascal, таковой и является) файл является DOS-файлом. Поэтому увидеть его содержимое при помощи Блокнота нельзя, вместо букв русского алфавита отображается абракадабра. Это объясняется тем, что буквы русского алфавита в DOS и Windows кодируются разными числами. Также необходимо обратить внимание на то, что при запуске программы из Turbo Pascal файл test.txt будет создан в том каталоге, в котором находится pas-файл, а при запуске из операционной системы — в том, в котором находится exe-файл.

Листинг 9.1. Создание / замена файла (p9_1_.pas)

```
program p9_1;
var
   f: text;     { текстовый файл }
   st: string;
```

```
   i: integer;
begin
   assign(f,'test.txt');
   rewrite(f);   { открыть файл в режиме перезаписи }

   write('Введите 5 имен. ');
   writeln('После ввода каждого имени нажимайте <Enter>');

   for i:=1 to 5 do
       begin
            write(i,'>');
            readln(st);      { ввод строки }
            writeln(f,st);  { запись введенной строки в файл }
       end;
   close(f); { закрыть файл }
   writeln('Данные записаны в файл');
   write('Для завершения нажмите <Enter>');
   readln;
end.
```

В листинге 9.2 приведена программа, которая демонстрирует процесс добавления информации в файл. Программа открывает файл test.txt и добавляет в его конец три строки, введенные пользователем. Следует обратить внимание, если файл не будет доступен (например, из-за того, что неправильно указан путь), то попытка открыть его приведет к возникновению ошибки и аварийному завершению работы программы (при запуске программы из Turbo Pascal в верхней части окна редактора кода появится сообщение Error 2: File not found (файл не найден) и курсор будет установлен в строку, в которой находится инструкция `append`).

Листинг 9.2. Добавление данных в существующий файл (p9_2.pas)

```
program p9_2;
var
   f: text;    { текстовый файл }
   st: string;
   i: integer;
begin
   assign(f,'test.txt');
```

```
    append(f);    { открыть для добавления }

    writeln('Введите 3 имени. ');
    writeln('После ввода каждого имени нажимайте <Enter>');

    for i:=1 to 3 do
        begin
             write(i,'>');
             readln(st);      { ввод строки }
             writeln(f,st);   { запись введенной строки в файл }

        end;
    close(f); { закрыть файл }
    writeln('Данные записаны в файл');
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Ошибки доступа к файлу

Попытка открыть файл может завершиться неудачей и, как следствие, привести к возникновению ошибки времени выполнения и аварийному завершению работы программы. Причин неудачи открытия файла может быть несколько. Например, может быть неправильно указано имя файла или путь к нему. Другая вероятная причина — попытка открыть для добавления несуществующий файл.

Выяснить, завершилась ли успехом процедура открытия файла, можно, проверив значение функции `IOResult` (Input-Output Result — результат ввода/вывода). Значение функции `IOResult` равно нулю, если операция ввода/вывода завершилась успешно, в противном случае значением функции является код ошибки (не ноль). Чтобы программа могла проверить результат выполнения операции ввода/вывода, ей нужно разрешить это сделать — поместить перед и после инструкции, при выполнении которой может произойти ошибка, соответственно строки `{$I-}` и `{$I+}`. Эти строки являются *директивами компилятору*. Директива `{$I-}` сообщает компилятору, что программа берет на себя обработку ошибок, которые могут возникнуть при выполнении операций ввода (I — сокращение от Input). Директива `{$I+}` отменяет действие директивы `{$I-}`.

В качестве примера в листинге 9.3 приведена программа, в которую добавлены инструкции, обеспечивающие обработку возможной ошибки открытия файла. Программа дописывает в файл test.txt введенную пользователем информацию или, если файла на диске нет, создает его.

Листинг 9.3. Обработка ошибки открытия файла (p9_3.pas)

```
{ Программа добавляет информацию в файл.
  Если файла нет, то создает его. }
program p9_3;
var
   f: text;     { текстовый файл }
   st: string;
   i: integer;
begin
   assign(f,'test_.txt');
   { при открытии файла для добавления возможна ошибка }
   {$I-}
   append(f);   {  открыть для добавления }
   {$I+}
   if IOResult <> 0 then
      begin
           writeln('Файл не найден');

           { файл test.txt недоступен, скорее всего файла нет.

             Сделаем попытку создать файл }
           {$I-}
           rewrite(f);
           {$I+}
           if IOResult = 0
           then
              writeln('Файл создан')
           else
              begin
                   writeln('Невозможно создать файл');
                   halt(1);  { завершение работы программы }
              end;
```

```
    end;

  writeln('Введите 3 имени. ');
  writeln('После ввода каждого имени нажимайте <Enter>');

  for i:=1 to 3 do
      begin
           write(i,'>');
           readln(st);      { ввод строки }
           writeln(f,st);   { запись введенной строки в файл }
      end;

  close(f); { закрыть файл }
  writeln('Данные записаны в файл');
  write('Для завершения нажмите <Enter>');
  readln;
end.
```

Чтение из файла

Чтобы программа могла прочитать (ввести) данные из файла, файл необходимо открыть для чтения.

Открытие файла для чтения выполняется вызовом процедуры `reset`. В качестве параметра процедуры необходимо указать файловую переменную. Перед вызовом процедуры `reset` файловой переменной следует присвоить значение — имя файла, который надо открыть. Делается это вызовом процедуры `assign`.

Следующие инструкции открывают файл data.txt для ввода:

```
assign(f,'data.txt');
reset(f);
```

Если файл, который надо открыть, недоступен (неправильно указано имя или путь), то в результате выполнения инструкции `reset` возникает ошибка и программа аварийно завершает работу.

Как и при открытии файла чтения, программа может взять на себя задачу обработки ошибки открытия файла путем проверки значения функции `IOResult`. Следующая программа, ее текст приведен на листинге 9.4, использует значение функции `IOResult` для результата операции открытия файла.

Если попытка открыть файл вызывает ошибку, программа выводит соответствующее сообщение.

Листинг 9.4. Обработка ошибки открытия файла (p9_4.pas)

```
{ обработка ошибки открытия файла }
program p9_4;
var
     f:text;             { текстовый файл}
     fname:string[80];   { имя файла}
begin
     fname:= 'test.txt';
     assign(f,fname);

     {$I-}
     reset(f);    { открыть файл для чтения }
     {$I+}
     if IOResult <> 0 then
         begin
               writeln('ошибка доступа к файлу: файл ',fname,
                       ' не найден');
               write('Для завершения нажмите <Enter>');
               readln;
               halt(1); { завершение работы программы }
          end;

     writeln('Файл открыт для чтения');

     { здесь инструкции программы }

     close(f);
     write('Для завершения нажмите <Enter>');
     readln;

end.
```

Непосредственное чтение данных из файла выполняется при помощи инструкций `read` и `readln`, которые в общем виде записываются так:

```
read(ФайловаяПеременная, СписокПеременных);
readln(ФайловаяПеременная, СписокПеременных);
```

где *ФайловаяПеременная* — переменная типа text; *СписокПеременных* — имена переменных, разделенные запятыми.

Следующая программа (листинг 9.5) демонстрирует чтение данных из файла. Она считывает из файла test.txt первую строку и выводит ее на экран.

Листинг 9.5. Чтение из файла строки символов (p9_5.pas)

```
program p9_5;
var
     f:text;          { текстовый файл}
     fn:string[80];   { имя файла}

     st: string;
begin
     fn:= 'test.txt';
     assign(f,fn);

     {$I-}
     reset(f);    { открыть файл для чтения }
     {$I+}
     if IOResult <> 0 then
         begin
               writeln('ошибка доступа к файлу: файл ',fn, ' не найден');
               write('Для завершения нажмите <Enter>');
               readln;
               halt(1); { завершение работы программы }
          end;

     writeln('Файл открыт для чтения');

     readln(f, st); { прочитать строку и записать ее в st }
     writeln('прочитанная строка:', st);

     close(f);
     write('Для завершения нажмите <Enter>');
     readln;
end.
```

В текстовом файле могут находиться не только символьные, но и числовые данные (листинг 9.6). Текстовые данные считываются из файла и записываются в переменные, указанные в инструкции чтения, без какой-либо обработки. При чтении чисел действие, выполняемое инструкциями `read` и `readln`, состоит из двух: сначала из файла считывается изображение числа (строка символов до появления пробела или конца строки), затем прочитанные символы преобразуются в значение соответствующего типа, которое и записывается в переменную, указанную в инструкции `read` или `readln`. Следующая программа, ее текст приведен в листинге 9.7, демонстрирует чтение данных из текстового файла sales.txt. Следует обратить внимание на тип переменной `volume`. Вместо универсального типа `integer` указан тип `longint`. Сделано это потому, что значения, загружаемые из файла, превышают максимальное значение типа `integer` (32767).

Листинг 9.6. Файл данных (sales.txt)

```
Ford Focus
73468
Renault Logan
49323
Mitsubishi Lancer
46969
Daewoo Nexia
43415
Chevrolet Niva
41155
```

Листинг 9.7. Чтение данных из файла (p9_7.pas)

```
program p9_7;
var
     f:text;              { текстовый файл}
     fn:string;           { имя файла}

     model: string;       { модель }
     volume: longint;     { количество }

begin
     fn:= 'sales.txt';
```

```
    assign(f,fn);

    {$I-}
    reset(f);   { открыть файл для чтения }
    {$I+}
    if IOResult <> 0 then
        begin
              writeln('ошибка доступа к файлу: файл ',fn, ' не найден');
              write('Для завершения нажмите <Enter>');
              readln;
              halt(1); { завершение работы программы }
         end;

    writeln('Файл открыт для чтения');

    readln(f, model);  { название модели автомобиля }
    readln(f, volume); { продано }

    writeln('Модель: ', model);
    writeln('Продано: ', volume);

    close(f);
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Чтение строк

В программе строковая переменная может быть объявлена с указанием длины или без (например, `st:string[10]` или `st:string`).

При чтении из файла значения строковой переменной, длина которой задана явно, из файла читается столько символов, сколько указано в объявлении, но не больше, чем в оставшейся непрочитанной части текущей строки. При чтении значения строковой переменной, длина которой в объявлении явно не задана, в переменную записываются все символы текущей строки или символы, оставшиеся после чтения предыдущего элемента данных.

Если количество символов, которое надо прочитать (указанное в объявлении переменной) не соответствует количеству символов в файле, то существует

вероятность ошибки. Например, если в программе объявлены переменные `first_name: string[10]` и `last_name: string[15]`, то в результате чтения данных из файла, в котором имя и фамилия размещены в одной строке (например, Никита Культин), значение переменной `first_name` будет `Никита Кул`, а переменной `last_name` — `ьтин`.

Чтобы избежать подобной ситуации, рекомендуется каждый элемент данных располагать в отдельной строке.

Конец файла

Пусть на диске есть файл, содержащий информацию продажах автомобилей разных марок (см. листинг 9.6), и нужно вычислить общее количество проданных автомобилей. Алгоритм решения задачи можно представить так:

- прочитать строку (название автомобиля);
- прочитать количество проданных автомобилей (следующая строка) и полученное значение добавить к сумме;
- повторять первый и второй шаги до тех пор, пока не будет обработан весь файл данных.

Но как определить, что обработан весь файл данных? Сделать это можно, проверив после чтения очередного элемента данных значение функции `EOF` (End Of File — конец файла). Функция `EOF`, в качестве параметра которой надо указать файловую переменную, возвращает `TRUE`, если чтение из файла невозможно (достигнут конец файла). Если в файле есть непрочитанные данные, значение функции `EOF` равно `FALSE`.

В листинге 9.8 приведена программа, которая демонстрирует использование функции `EOF`. Программа считывает информацию о количестве проданных автомобилей разных марок и вычисляет общую сумму.

Листинг 9.8. Использование функции EOF (p9_8.pas)

```
program p9_8;
var
     f:text;              { файл данных}
     fn:string;           { имя файла}

     model: string;       { модель }
     volume: longint;     { количество }
```

```
    total: longint;    { общее количество }

begin
    fn:= 'sales.txt';
    assign(f,fn);

    {$I-}
    reset(f);   { открыть файл для чтения }
    {$I+}
    if IOResult <> 0 then
        begin
            writeln('ошибка доступа к файлу: файл ',fn, ' не найден');

            write('Для завершения нажмите <Enter>');
            readln;
            halt(1); { завершение работы программы }
         end;

    writeln('Продажи');

    total := 0;
    while not EOF(f) do
        begin
            readln(f, model);  { название модели автомобиля }
            readln(f, volume); { продано }

            writeln(model, ': ', volume);

            total := total + volume;
        end;

    close(f);

    writeln;
    writeln('Всего: ', total);

    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Следует обратить внимание, что в приведенной программе значение функции EOF проверяется перед каждым циклом чтения данных, в том числе и перед первым. Перед первым чтением EOF проверяется для того, чтобы убедиться, что в файле есть данные (возможна ситуация, когда файл существует, но данных не содержит).

Вывод на печать

Программа может вывести результат своей работы на принтер. Вывод на принтер выполняют те же инструкции (write и writeln), что и вывод в файл. Единственное, что надо сделать, для того чтобы переключить вывод на принтер, — это указать в инструкции assign вместо файла принтер (имя prn, сокращение от printer).

Следует обратить внимание, принтер должен быть подключен к LPT-порту (а не USB). Кроме того, большинство современных принтеров (струйные, лазерные) не обеспечивают правильное отображение символов русского алфавита, представленных в кодировке ASCII, которая используется в DOS-программах (программа, созданная в Turbo Pascal, является DOS-программой). Поэтому существует вероятность, что на бумаге вместо букв русского алфавита будет отображаться абракадабра.

Следующая программа, ее текст приведен в листинге 9.9, выводит на печать информацию о проданных автомобилях. Программа считывает данные из файла sales.txt, обрабатывает их и выводит результат на принтер.

Листинг 9.9. Вывод на принтер (p9_9.pas)

```
program p9_9;
var
     f:text;              { файл данных}
     fn:string;           { имя файла}

     model: string;       { модель }
     volume: longint;     { количество }
     total: longint;      { общее количество }

     p: text; { файл результата - принтер }

begin
     fn:= 'sales.txt';
```

```
assign(f,fn);

{$I-}
reset(f);   { открыть файл для чтения }
{$I+}
if IOResult <> 0 then
    begin
          writeln('ошибка доступа к файлу: файл ',fn, ' не найден');
          write('Для завершения нажмите <Enter>');
          readln;
          halt(1); { завершение работы программы }
     end;

assign(p, 'prn'); { при записи данных в файл p,
                    данные будут отправлены на принтер }


rewrite(p);       { открыть файл (принтер) для записи (вывода) }

writeln('Продажи');          { на экран }
writeln(p,' Продажи ');      { на принтер }

total := 0;
while not EOF(f) do
    begin
         readln(f, model);  { название модели автомобиля }
         readln(f, volume); { продано }

         writeln(model, ': ', volume);    {  на экран }
         writeln(p, model, ': ', volume); {  на принтер }

         total := total + volume;
    end;

{ на экран }
writeln;
writeln('Всего: ', total);

{ на принтер }
```

```
    writeln(p, 'Всего: ', total);

     close(f);
     close(p);

     write('Для завершения нажмите <Enter>');
     readln;

end.
```

Если принтер позволяет печатать разными шрифтами, то программа может выбрать нужный шрифт и установить его характеристики. Выбор шрифта и установка его характеристик выполняется посылкой на принтер последовательности управляющих символов (управляющей последовательности). Отличие управляющей последовательности символов от обычной символьной строки в том, что обычные символы отображаются на бумаге, а символы управляющей последовательности — нет. Конкретный вид управляющей последовательности символов зависит от типа принтера. Ниже, в табл. 9.1, приведены управляющие последовательности принтера Hewlett Packard Desk Jet 520. Строка <ESC> означает управляющий символ "Escape" (его код 27).

Таблица 9.1. *Управляющие последовательности принтера HP Desk Jet 520*

Управляющая последовательность	**Устанавливает**
<ESC>(s12H	Плотность печати в строке: 12 символов на дюйм (12 cpi)
<ESC>(s17H	Плотность печати в строке: 17 символов на дюйм (17 cpi)
<ESC>&l6D	Вертикальная плотность: 6 строк на дюйм (1 интервал)
<ESC>&l4D	Вертикальная плотность: 4 строки на дюйм (1,5 интервала)
<ESC>(s0B	Обычный шрифт
<ESC>(s3B	Полужирный шрифт
<ESC>(s2Q	Обычное качество
<ESC>(s1Q	Экономный режим

Следующая программа, ее текст приведен в листинге 9.10, демонстрирует использование управляющих последовательностей для установки характеристик шрифта принтера.

Листинг 9.10. Управление шрифтом принтера (p9_10.pas)

```
program p9_10;
var
     f:text;              { файл данных}
     fn:string;           { имя файла}

     model: string;       { модель }
     volume: longint;     { количество }
     total: longint;      { общее количество }

     p: text; { файл результата - принтер }

begin
     fn:= 'sales.txt';
     assign(f,fn);

     {$I-}
     reset(f);    { открыть файл для чтения }
     {$I+}
     if IOResult <> 0 then
         begin
               writeln('ошибка доступа к файлу: файл ',fn, ' не найден');
               write('Для завершения нажмите <Enter>');
               readln;
               halt(1); { завершение работы программы }
          end;


     assign(p, 'prn'); { при записи данных в файл p,
                         данные будут отправлены на принтер }

     rewrite(p);        { открыть файл (принтер) для записи (вывода) }

     { установить шрифт принтера }
     write(p,chr(27),'(s3B');  { полужирный }


     writeln('Продажи');        { на экран }
     writeln(p,'Продажи');      { на принтер }
```

```
    write(p,chr(27),'(s0B');  { обычный (отменить полужирный) }

    total := 0;
    while not EOF(f) do
        begin
            readln(f, model);  { название модели автомобиля }
            readln(f, volume); { продано }

            writeln(model, ': ', volume);    {  на экран }
            writeln(p, model, ': ', volume); {  на принтер }

            total := total + volume;
        end;

    { на экран }
    writeln;
    writeln('Всего: ', total);

    { на принтер }
    write(p,chr(27),'(s3B');      { полужирный шрифт }
    writeln(p, 'Всего: ', total);
    writeln(p, chr(12));          { выдать лист }

    close(f);
    close(p);

    write('Для завершения нажмите <Enter>');
    readln;

end.
```

Пример программы

Система проверки знаний

Тестирование широко применяется для оценки уровня знаний в учебных заведениях, при приеме на работу, для оценки квалификации персонала учреждений и т. д. Испытуемому предлагается *тест* — последовательность вопросов, на которые он должен ответить. Обычно к каждому вопросу дается несколько вариантов ответа, из которых надо выбрать правильный. Каждому варианту ответа соответствует некоторая оценка. Суммированием оценок

за ответы получается общий балл, на основе которого делается вывод об уровне подготовленности испытуемого (выставляется оценка).

Ниже рассматривается программа, которая позволяет автоматизировать процесс тестирования.

Требования к программе

В результате анализа различных тестов были сформулированы следующие требования к программе:

- Программа должна обеспечить работу с тестом произвольной длины (не должно быть ограничений на количество вопросов в тесте).
- Каждому вопросу может соответствовать до четырех возможных вариантов ответа, только один из которых является правильным.
- Результат тестирования должен быть отнесен к одному из четырех уровней. Например, "отлично", "хорошо", "удовлетворительно" или "плохо".
- Тест должен представлять собой текстовый файл.
- Программа должна быть инвариантна к различным тестам, т. е. изменения в тесте не должны вести за собой требования изменения программы.
- Программа не должна обеспечивать возврат к предыдущему вопросу. Если вопрос предложен, то на него должен быть получен ответ.

Алгоритм программы

Укрупненный алгоритм работы программы может быть представлен такой последовательностью инструкций (которая, конечно, не является программой на языке Pascal):

```
begin
    { открыть файл теста }
     while { пока есть вопросы }
       begin
         { прочитать из файла вопрос и варианты ответа,
           вывести их на экран }
         { получить от испытуемого номер ответа }
         { добавить к общей сумме балл за выбранный вариант ответа }
       end;
     { используя значение общей суммы,
        вывести результат тестирования }
end.
```

Структура данных

Реализация алгоритма программы в значительной мере зависит от представления обрабатываемых программой данных. Для рассматриваемой программы файл теста имеет следующую структуру:

Заголовок (название) теста

Сообщение для уровня 1

Сумма баллов для достижения уровня 1

Сообщение для уровня 2

Сумма баллов для достижения уровня 2

Сообщение для уровня 3

Сумма баллов для достижения уровня 3

Сообщение для уровня 4

Сумма баллов для достижения уровня 4

Вопрос_1

Количество_вариантов_ответа_1 номер_правильного_ответа_1

Вариант ответа

Вариант ответа

Вариант ответа

Вопрос_2

Количество_вариантов_ответа_2 номер_правильного_ответа_2

Вариант ответа

Вариант ответа

Вариант ответа

...

В листинге 9.11 в качестве примера приведен файл теста, целью которого является проверка знания истории Санкт-Петербурга. Следует обратить внимание, заголовок, сообщения, вопросы и варианты ответов находятся каждый в одной строке, что позволяет прочитать заголовок, сообщение, вопрос или вариант ответа одной инструкцией `readln`.

Листинг 9.11. Пример файла теста (spb.tst)

```
История Санкт-Петербурга
Отлично
7
Хорошо
```

```
6
Удовлетворительно
5
Плохо
4
Архитектор Исаакиевского собора:
3 2
Доменико Трезини
Огюст Монферран
Карл Росси
Александровская колонна воздвигнута в 1836 как памятник, посвященный:
2 1
деяниям императора Александра I.
подвигу народа в Отечественной войне 1812 года.
Архитектор Зимнего дворца:
3 1
Бартоломео Растрелли
Карл Росси
Огюст Монферран
Михайловский замок построен по проекту:
3 1
Винченцо Бренна
Старова Ивана Егоровича
Баженова Василия Ивановича
Остров, на котором находится Ботанический сад, называется:
3 3
Заячий
Медицинский
Аптекарский
Невский проспект получил свое название
3 2
по имени реки, на которой стоит Санкт-Петербург.
по имени близко расположенного монастыря, Александро-Невской лавры.
в память о знаменитом полководце - Александре Невском.
Скульптура памятника Петру I "Медный всадник" выполнена
2 1
Фальконе
Клодтом
```

Подготовить файл теста можно при помощи редактора кода Turbo Pascal (после того как тест будет набран, в меню **File** следует выбрать команду **Save** и в появившемся окне задать имя файла теста) или при помощи Word. В последнем случае в момент сохранения файла в списке **Тип файла** следует выбрать **Текст DOS с разбиением на строки**.

Теперь рассмотрим ключевые моменты реализации программы.

Доступ к файлу теста

Обеспечить работу программы с различными файлами тестов (файлы с разными именами) можно, если при запуске программы указать параметр — имя файла текста. В этом случае команда запуска программы тестирования может быть, например, такой:

```
exam spb.tst
```

где `exam` — имя exe-файла программы тестирования (при запуске программы расширение exe можно не указывать); `spb.tst` — имя файла с текстом теста.

Программа может получить параметр, указанный в командной строке, обратившись к функции `ParamStr` (в инструкции вызова функции необходимо указать номер нужного параметра). Также весьма полезна функция `ParamCount`, которая возвращает количество параметров командной строки. Для приведенного выше примера команды запуска программы тестирования значение `ParamCount` равно 1, а `ParamStr(1)` — `spb.tst`.

При запуске программы в режиме отладки (из Turbo Pascal) параметр следует ввести в поле **Parameter** окна **Program Parameters** (рис. 9.1), которое становится доступным в результате выбора в меню **Run** команды **Parameters**.

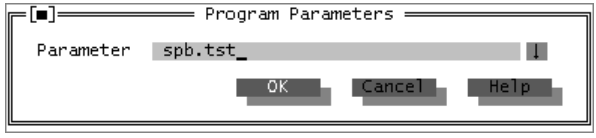


Рис. 9.1. Окно **Program Parameters**

Программа

Текст программы Экзаменатор приведен в листинге 9.12.

Листинг 9.12. Программа проверки знаний (exam.pas)

```
{ программа "Экзаменатор"
  (c) Культин Н.Б., 2007 }
uses Crt;
```

```
{ возвращает номер выбранного ответа.
  функция "реагирует" только на нажатие цифровых
  клавиш от 1 до max, клавиш <Enter>  и <Backspace>. }

function GetAns(max:integer):integer;
const
     { коды клавиш }
     K_ENTER=13;      { Enter }
     K_BACKSPACE=8;   { Backspace}
var
     ch:char;         { символ }
     dec:integer;     { код символа }
     maxdec:integer; { максимально допустимый код символа }
     x,y:integer;     { положение курсора }
     code:integer;
     i:integer; { счетчик введенных цифр }
     n:integer;
begin
     maxdec:= 48 + max; { 48 - код нуля }
     i:=0;
     repeat
          ch:=readkey; { символ нажатой клавиши }
          dec:=ord(ch);
          if(dec> 48) and (dec<=maxdec) and (i=0)
               then begin { нажата допустимая цифра }
                   write(ch);
                   val(ch,n,code);
                   i:=i+1;
               end
          else
               if(dec=K_BACKSPACE)and(i=1)
                    then begin
                         i:=0;
                         { сотрем введенную ранее цифру }
                         x:=whereX;
                         y:=whereY;
                         gotoXY(x-1,y);
                         write(' ');
```

```
                        gotoXY(x-1,y);
                    end;
    until(i=1)and(dec=K_ENTER);
    writeln;
    GetAns:=n;
end;

var
   fname: string;  { имя файла теста }
   f: text;        { файл теста }

   nQues: integer; { количество вопросов теста }
   nRight: integer; { количество правильных ответов }

   { для текущего вопроса }
   nAnsw: integer; { количество вариантов ответа }
   Right: integer; { номер правильного ответа}
   Sel: integer;  { номер выбранного ответа }

   st: string; { строка, читаемая из файла теста }


   i: integer; { счетчик циклов }

   mes: array[1..4] of string;     { сообщение }
   level: array[1..4] of integer; { кол-во правильных ответов,
                                    необходимое для
                                    достижения уровня }

begin
  if ParamCount = 0 then
    begin
      writeln('Не задан файл теста!');
      writeln('Командная строка: exam Файл_Теста');
      halt(1);
    end;

  fname:=ParamStr(1); { имя файла из командной строки }
  Assign(f,fname);
```

```
{$I-}
Reset(f);           { открыть файл для чтения }
{$I+}
if IOResult <> 0 then
   begin
     writeln('Ошибка доступа к файлу: ', fname);
     writeln('Для завершения нажмите <Enter>');

     halt(1); { завершение работы программы }
   end;

TextBackground(Blue);
TextColor(Lightgray);
ClrScr;

readln(f,st);
writeln(st);  {  заголовок }
writeln;

{ читаем критерии оценки - сообщение и количество правильных
  ответов, необходимое для достижения уровня }

for i :=1 to 4 do
   begin
        readln(f,mes[i]);
        readln(f,level[i]);
   end;

write('Сейчас Вам будут предложены вопросы,');
writeln('на которые вы должны ответить.');
writeln('Ваша задача - ввести номер правильного ответа',
        'и нажать <Enter>)');
writeln;
write('Для начала тестирования нажмите <Enter>');
readln;

TextBackground(Blue);
ClrScr;
nQues := 0;
nRight:=0;
```

```
  while not EOF(f) do
   begin
      writeln;
      nQues := nQues + 1;
      readln(f,st);          { читаем вопрос }
      TextColor(White);
      writeln(st);     { выводим вопрос на экран }
      readln(f,nAnsw,Right);  { читаем кол-во вариантов ответа
                              и номер правильного ответа }
      TextColor(LightGray);
      for i:=1 to nAnsw do  { читаем и варианты ответа }
          begin
            readln(f,st);
            writeln(i,'. ',st);
          end;
      writeln;
      write('Ваш выбор ->');
      sel := GetAns(nAnsw);
      if Sel = Right then nRight := nRight + 1;
      writeln;
   end;

   writeln('***Результат тестирования***');
   writeln('Вопросов:', nQues,
           ' Правильных ответов:', nRight);

   { обработка результата тестирования }
   i := 1; { пусть правильных ответов достаточно для
             достижения наивысшего уровня (лучшая оценка) }
   while (nRight  < level[i]) and (i < 4) do
         i := i + 1; { понизим уровень оценки }

   writeln(mes[i]); { оценка }

   writeln;
   write('Для завершения нажмите <Enter>');
   readln;
end.
```

Комментарии к программе

Получение номера варианта ответа от экзаменуемого

Инструкции получения от экзаменуемого номера ответа объединены в функцию `GetAns`, которая получает в качестве параметра количество вариантов ответа для отображаемого в данный момент на экране вопроса. Значение функции `GetAns` — номер выбранного ответа.

Функция `GetAns` реагирует на нажатие цифровых клавиш (от единицы до `n`, где `n` — количество вариантов ответа), клавиши <BackSpace> и клавиши <Enter>, при нажатии которой функция завершает работу.

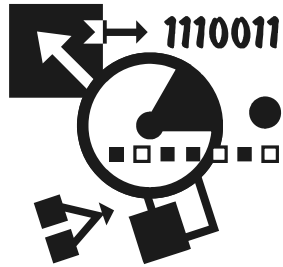
Внутри функции `GetAns` ввод с клавиатуры осуществляется при помощи функции `ReadKey`, которая возвращает символ, соответствующий нажатой клавише, однако на экран его не выводит. Если испытуемый нажимает "правильную" клавишу (клавишу с цифрой от единицы до `n`, где `n` — количество вариантов ответа) и на экране еще нет ни одной цифры, то символ появляется на экране. Таким образом, у экзаменуемого создается впечатление, что программа реагирует только на цифровые клавиши в заданном диапазоне.

Функция `GetAns` позволяет испытуемому изменить номер выбранного ответа. Для этого он должен нажать <BackSpace> и снова нажать клавишу с цифрой. Удаление символа с экрана осуществляется выводом пробела в позицию, где находится символ, который надо стереть. Для определения положения курсора используются функции `WhereY` и `WhereX`, возвращающие текущие координаты курсора.

Обработка результатов теста

Оценка результата тестирования носит интервальный характер. Например, если тест состоит из 10 вопросов, то чтобы получить "отлично", надо правильно ответить на 10 вопросов, "хорошо" — от 8 до 9, "удовлетворительно" — от 6 до 7. Оценка за тест получается сравнением количества правильных ответов и количества ответов, характеризующих уровень. В приведенном примере уровень "отлично" характеризуется числом 10, уровень "хорошо" — числом 8, уровень "удовлетворительно" — числом 6.

В программе при определении оценки количество правильных ответов (`nRight`) сравнивается со значениями, характеризующими уровни оценок (массив `level`). Первоначально предполагается, что набранное количество баллов позволяет получить оценку первого уровня — "отлично". Если предположение неверно, то уровень оценки понижается, и набранное количество баллов сравнивается с оценкой этого уровня, и так до тех пор, пока не будет найден уровень, соответствующий набранному количеству баллов. После того как будет определен уровень, выводится соответствующее сообщение — оценка.



Глава 10

Типы данных, определяемые программистом

До этого момента в своих программах вы использовали стандартные типы данных языка Pascal. Вместе с тем программист может определить свой собственный тип данных и затем использовать его в программе.

Программист может определить перечисляемый, интервальный или составной тип данных.

В программе объявления типов помещают в раздел `type` (раздел объявления типов).

Перечисляемый тип

Тип, для которого можно перечислить значения, образующие этот тип, называется перечисляемым. Все целые типы (`integer`, `longint` и др.), а также тип `char`, являются перечисляемыми типами.

Определить перечисляемый тип — это значит перечислить все значения, которые может принимать переменная, относящаяся к этому типу.

Объявление перечисляемого типа помещается в раздел объявления типов (раздел `type`) и в общем виде выглядит так:

Тип = (*Значение_1*, *Значение_2*, ... *Значение_i*);

где:

Тип — имя определяемого типа данных (следует обратить внимание, согласно принятому в среде программистов соглашению, имя типа должно начинаться с буквы `T`) ;

Значение_i — одно из значений, которое может принимать переменная, относящаяся к объявляемому типу.

Пример:

```
TDayOfWeek =(1,2,3,4,5,6,7);
```

После объявления типа, его можно использовать, например, объявить переменную:

```
ThisDay: TDayOfWeek;
```

При объявлении перечисляемого типа удобно использовать именованные константы. Например, тип `TDayOfWeek` можно объявить так:

```
TDayOfWeek = (MON,TUE,WED,THU,FRI,SAT,SUN);
```

Если именованные константы, используемые при объявлении перечисляемого типа не объявлены в программе явно, то считается, что значение первой константы списка равно нулю, следующей — 1 и т. д.

Переменной перечисляемого типа можно присвоить значение, относящееся к этому типу. Присвоить значение, которое к этому типу не относится, нельзя — компилятор выведет сообщение об ошибке: несоответствие типа значения типу переменной.

Помимо указания значений, которые может принимать переменная, объявление перечисляемого типа задает, как значения, образующие объявляемый тип, соотносятся между собой (если значения именованных констант не указаны явно): меньшим считается то значение, которое в списке объявления находится левее. Так для элементов типа `TDayOfWeek` справедливо:

```
MON < TUE < WED < THU < FRI < SAT < SUN
```

Свойство упорядоченности элементов перечисляемого типа позволяет сравнивать переменные, относящиеся к одному перечисляемому типу, между собой и с константами, принадлежащими этому типу (указанными в объявлении типа). Например:

```
if (Day = SAT) or (Day = SUN) then
     begin
          { действие, если день суббота или воскресенье }
     end;
```

В листинге 10.1 приведена программа, которая демонстрирует использование перечисляемого типа, объявленного пользователем. Следует обратить внимание, что использование несущих смысловую нагрузку именованных констант в объявлении перечисляемого типа существенно облегчает восприятие программы, снижает вероятность допустить ошибку при составлении программы.

Листинг 10.1. Перечисляемый тип, объявленный программистом (p10_1.pas)

```
program p10_1;

type { начало раздела объявления типов }
```

```
    { тип "день недели"}
    TDayOfWeek = (MON,TUE,WED,THU,FRI,SAT,SUN);
var
   sum: real;  { сумма покупки }
   d: integer; { номер дня недели }

   today: TDayOfWeek; { день недели }
   discount: real; { скидка }

begin
   write('День недели (1-понедельник; 2 - вторник и т.д.) -> ');
   readln(d);
   write('Сумма покупки ->');
   readln(sum);

   case d of
     1: today := MON;
     2: today := TUE;
     3: today := WED;
     4: today := THU;
     5: today := FRI;
     6: today := SAT;
     7: today := SUN;
   end;

   if (today = SAT) or (today = SUN) then
       begin
            discount := 0.05 * sum;
            sum := sum - discount;
            writeln('Скидка:', discount:6:2);
            writeln('К оплате:', sum:6:2);
       end
   else
       begin
             writeln('Скидка не предоставляется');
       end;

   write('Для завершения нажмите <Enter>');
   readln;
end.
```

Интервальный тип

Интервальный тип является отрезком или частью какого-то другого типа, называемого *базовым*. При объявлении интервального типа указываются нижняя и верхняя границы интервала, т. е. наименьшее и наибольшее значение, которое может принять переменная этого типа. В общем виде объявление интервального типа выглядит так:

Тип = *НижняяГраница* .. *ВерхняяГраница*;

где *Тип* — имя определяемого типа данных;

НижняяГраница — наименьшее значение, которое может принять переменная объявляемого типа;

ВерхняяГраница — наибольшее значение, которое может принять переменная объявляемого типа;

В качестве базового типа можно использовать перечисляемый тип, например `integer`.

Например, инструкция

```
TIndex = 0 .. 100;
```

записанная в разделе `type`, объявляет интервальный тип `TIndex`.

При объявлении интервального типа удобно использовать именованные константы. Например:

```
const
     LB = 0;  { Low Bound - нижняя граница интервала }
     HB = 10; { High Bound - верхняя граница интервала }
type
     TIndex = LB .. HB;
```

Интервальный тип удобно использовать при объявлении массивов, например, так:

```
const
     LB = 0;  { Low Bound - нижняя граница интервала }
     HB = 10; { High Bound - верхняя граница интервала }
type
     TIndex = LB .. HB;
var
     tabl: array[TIndex] of integer;
     i: TIndex;
```

```
begin
     for i:=LB to HB do
          tabl[i] := 0;
end.
```

Как было сказано раньше, при объявлении интервального типа в качестве базового типа можно использовать любой, в том числе и созданный пользователем, перечисляемый тип. В следующем фрагменте на основе типа `TMonth` (месяц) объявлен интервальный тип `TSummer` (лето).

```
type
     TMonth =(Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
     TSummer = Jun .. Aug;
```

Запись

На практике часто приходится иметь дело с данными, которые состоят из других данных. Например, сведения о сотруднике таковы: имя (`string`), фамилия (`string`), идентификатор подразделения (`integer`), зарплата (`real`). Для представления подобной информации в языке Pascal используется тип "*запись*" (`record`).

Запись — это структура данных, представляющая собой совокупность элементов (компонентов), как правило, разного типа. Компоненты записи называют *полями*.

Объявление записи

Как любой тип, определяемый программистом, тип "запись" должен быть объявлен в разделе `type`. В общем виде объявление типа "запись" выглядит так:

```
Имя = record
          Поле_1: Тип_1;
          Поле_2: Тип_2;
              ...
          Поле_K: Тип_K;
     end;
```

где *Имя* — имя типа; **record** — слово языка Pascal, означающее, что далее следует объявление компонентов (полей) записи; *Поле_i* и *Тип_i* — имя и тип i-го компонента (поля) записи.

Пример объявления записей:

```
TStudent = record
    FirstName: string[15];   { имя }
    LastName: string[20];    { фамилия }
    group: integer;          { номер учебной группы }
end;

TBook = record
    Title: string[40];    { название }
    Author: string[20];   { автор }
    Price: real;          { цена }
end;
```

После объявления типа записи можно объявить запись (переменную-запись), например:

```
student: TStudent;
book: TBook;
```

Доступ к полям записи

Чтобы в программе получить доступ к полю записи, надо указать запись (имя переменной-записи) и имя поля, отделив имя поля от имени переменной-записи точкой. Например, инструкции:

```
writeln(book.Author);
writeln(book.Title);
writeln(book.Price:6:2);
```

выводят на экран значения полей `Author`, `Title` и `Price` записи `book`.

Инструкция *WITH*

Инструкция `with` позволяет использовать в тексте программы имена полей без указания имени переменной-записи. В общем виде инструкция выглядит так:

```
with Запись do
    begin
        { инструкции программы }
    end;
```

где:

with — слово языка Pascal, означающее, что далее, до слова **end**, при обращении к полям переменной-записи *Запись* можно не указывать имя записи; *Запись* — имя переменной-записи.

Например, инструкции:

```
writeln(book.Author);
writeln(book.Title);
writeln(book.Price:6:2);
```

можно записать так:

```
with book do
   begin
      writeln(Author);
      writeln(Title);
      writeln(Price:6:2);
   end;
```

Массив записей

Массив — совокупность данных одинакового типа. Двумерные массивы используют для представления таблиц, если все ячейки таблицы содержат данные одного типа. На практике часто приходится иметь дело с таблицами, столбцы которых содержат информацию разного типа. В программе такую таблицу можно представить как совокупность массивов разного типа. Например, прайс-лист можно представить так:

```
Title: array[1 .. 100] of string[40];   { название }
Author: array[1 .. 100] of string[20];  { автор }
Price: array[1 .. 100] of real;         { цена }
```

Другой способ представить таблицу — объявить массив записей. Например, прайс-лист можно объявить как массив записей так:

```
type
   TBook = record
    Title: string[40];   { название }
    Author: string[20];  { автор }
    Price: real;         { цена }
end;
var
    books: array[1 .. 100] of TBook;  { прайс-лист }
```

Доступ к элементу массива записей осуществляется обычным образом — по номеру нужного элемента, а доступ к полю — по имени. Следующий фрагмент кода демонстрирует работу с массивом записей.

```
type
   TBook = record
    Title: string[40];    { название }
    Author: string[20];   { автор }
    Price: real;          { цена }
end;
var
    books: array[1 ..100] of TBook;  { прайс-лист }
    i: integer;
begin
    for i := 1 to 100 do
       begin
           write(books[i].Title);
           write(books[i].Author);
           writeln(books[i].Price);
       end;
end;
```

Ввод и вывод записей в файл

Записи можно хранить в файле (в этом случае файл записей можно рассматривать как простейшую базу данных). Чтобы сохранить запись в текстовом файле, надо каждое поле записать в файл. Следующая программа, ее текст приведен в листинге 10.2, добавляет в файл books.txt записи — информацию, введенную пользователем с клавиатуры. Каждая запись содержит информацию об одной книге.

Листинг 10.2. Добавление записей в файл (p10_2.pas)

```
program p10_2;
type
    TBook = record
        Title: string[40];  { название }
        Author: string[20]; { автор }
        Price: real;        { цена }
```

```
    end;

var
    f: text;
    book: TBook;
    r: integer;     { выбор пользователя }
begin
    assign(f,'books.txt');
    {$I-}
    append(f); { открыть файл для добавления информации }
    {$I-}
    if IOResult <> 0 then
       begin
            { создать файл }
            {$I-}
            rewrite(f); { открыть файл для перезаписи }
            {$I-}
            if IOResult <> 0 then
                begin
                    writeln('Невозможно создать файл БД');
                    write('Для завершения нажмите <Enter> ');
                    readln;
                    halt(1);

                end;
       end;
    repeat
       with book do
            begin
                write('Автор>');
                readln(Author);
                write('Название>');
                readln(Title);
                write('Цена>');
                readln(Price);

                writeln(f,Author);
                writeln(f,Title);
                writeln(f,Price:6:2);
```

```
            end;
        write('Продолжить ввод данных (1-да,0-нет)?');
        readln(r);
    until r = 0;
    close(f);
    write('Для завершения нажмите <Enter> ');
    readln;
end.
```

После запуска программы файл books.txt может быть, например, таким:

```
Культин Н.Б.
Основы программирования в Turbo Delphi
180.00
Культин Н.Б.
Turbo Pascal в примерах и задачах
 90.00
```

Следующая программа, ее текст приведен в листинге 10.3, демонстрирует чтение записей из файла. Она открывает файл books.txt, который содержит информацию о книгах, и выводит на экран список книг.

Листинг 10.3. Чтение записей из файла (p10_3.pas)

```
program p10_3;
uses Crt;
type
    TBook = record
        Title: string[40];  { название }
        Author: string[20]; { автор }
        Price: real;        { цена }
    end;

var
    f: text;
    book: TBook;
begin
    assign(f,'books.txt');
    {$I-}
    reset(f); { открыть файл чтения информации }
    {$I-}
```

```
    if IOResult <> 0 then
       begin
           writeln('Ошибка доступа к файлу БД');
           write('Для завершения нажмите <Enter>');
           readln;
           halt(1);
       end;

    ClrScr;
    while not EOF(f) do
    begin
       with book do
            begin
                { прочитать запись из файла}
                readln(f,Author);
                readln(f,Title);
                readln(f,Price);
                { вывести информацию на экран }
                write(Author);
                write(' ',Title);
                writeln(' ',Price:6:2);
            end;
    end;
    close(f);
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Динамические структуры данных

До этого момента мы имели дело только со статическими структурами данных. Хотя во время работы программы значения переменных могут меняться, само количество переменных остается неизменным. Это не всегда удобно. Например, если программа предназначена для обработки информации об учениках класса, и для хранения данных используются массивы, то, определяя размер массива, разработчик программы должен ориентироваться на некоторое, предельно возможное количество учеников в классе. При этом если реально учеников в классе меньше предполагаемого числа, то неэффек-

тивно используется память компьютера, а если больше, то программу нельзя будет использовать.

Задачи, обрабатывающие данные, которые по своей природе являются динамическими, удобно решать с использованием *динамических структур*.

Переменные-указатели

Обычно переменная хранит некоторые данные. Помимо таких переменных существуют переменные, которые хранят ссылки на другие переменные. Такие переменные называются *указателями*. Переменная-указатель хранит адрес переменной, что можно изобразить графически (рис. 10.1).

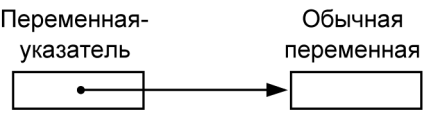


Рис. 10.1. Переменная-указатель

Переменная-указатель, как и любая другая переменная программы, должна быть объявлена. В общем виде объявление переменной-указателя выглядит так:

Имя: ^*Тип*

где:

Имя — имя переменной-указателя;

Тип — тип переменной, на которую может указывать переменная-указатель *Имя*.

Значок "^" показывает, что объявляемая переменная — это переменная-указатель.

Вот примеры объявления переменных-указателей:

```
p1: ^integer;
p2: ^real;
```

Переменная `p1` — это указатель на переменную типа `integer`, `p2` — на переменную типа `real`.

В начале работы программы переменная-указатель "ни на что не указывает", в этом случае говорят, что указатель пустой, и его значение равно **`nil`**.

Указателю можно присвоить значение другого указателя при условии, что они являются указателями на переменные одного типа. Например, если

переменные p1 и p2 — указатели на integer, то в результате выполнения инструкции

```
p2 := p1;
```

переменные p1 и p2 будут указывать на одну и ту же переменную.

Указатель можно использовать для того, чтобы присвоить значение переменной, на которую он указывает. Например, если p указывает на i, то в результате выполнения инструкции

```
p^ := 5;
```

значение переменной i будет равно пяти.

Динамические переменные

Динамической называется переменная, память для которой выделяется во время работы программы. Выделение памяти для динамической переменной осуществляется вызовом процедуры new. У процедуры new один параметр — указатель. Процедура new выделяет память для переменной (создает переменную) и записывает адрес созданной переменной в указатель, указанный в качестве параметра процедуры. Например, если p — указатель на переменную типа real, то в результате выполнения new(p) будет создана переменная типа real, и ее адрес будет записан в переменную-указатель p.

У динамической переменной, созданной процедурой new, нет имени, поэтому обратиться к ней можно только при помощи указателя.

Приведенная в листинге 10.4 программа демонстрирует создание и использование динамических переменных.

Листинг 10.4. Создание, использование и уничтожение динамических переменных (p10_4.pas)

```
program p10_4;
var
     p1,p2,p3: ^integer; { указатели на переменные типа integer }
begin
     { значение указателей p1,p2,p3 равно nil}
     new(p1);      { выделить память для переменной типа integer }
     new(p2);
     new(p3);

     { здесь существуют три переменные и их адреса
```

```
      находятся в указателях p1,p2,p3 }

    writeln('Введите в одной строке два целых числа и нажмите <Enter>');
    write('> ');
    readln(p1^,p2^);
    p3^:=p1^+p2^;
    writeln('Сумма введенных чисел равна ',p3^);
    writeln('Для завершения нажмите <Enter>');
    readln;

    { освободить память, занимаемую переменными, на которые
      указывают p1,p2 и p3 }
    dispose(p1);
    dispose(p2);
    dispose(p3);
end.
```

Списки

Указатели и динамические переменные позволяют создавать сложные динамические структуры данных, такие как списки и деревья.

Список можно изобразить графически (рис. 10.2).

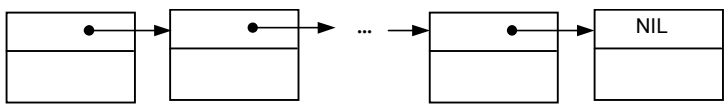


Рис. 10.2. Графическое изображение списка

Каждый элемент списка представляет собой запись, состоящую из двух частей. Первая часть — информационная. Вторая часть обеспечивает связь со следующим, и, возможно, с предыдущим элементом списка. Список, в котором обеспечивается связь только со следующим элементом, называется *односвязным*.

Чтобы программа могла использовать список, надо определить тип компонентов списка и указатель на первый элемент списка. Вот пример объявления списка:

```
type
    pStudent = ^TStudent;  { указатель на переменную типа TStudent}
```

```
TStudent = record
    FirstName: string[20]; { имя }
    LastName: string[20];  { фамилия }
    next: pStudent;        { указатель на следующий элемент списка}
  end;
var
  head: pStudent; { указатель на первый элемент списка }
```

Следует обратить внимание, что при объявлении элемента списка необходимо объявить вспомогательный тип "указатель на элемент списка" (в приведенном примере это тип `TpStudent`) для того, чтобы можно было указать тип переменной, обеспечивающей связь со следующим элементом списка (поле `next`). Переменная `next` это указатель на тип `TStudent`. Однако объявить ее как `^TStudent` нельзя (компилятор выводит сообщение об ошибке `Undefined type in pointer definition` — Неизвестный тип в объявлении указателя), так как во время обработки объявления типа `TStudent` компилятор еще не "знает", что обозначает идентификатор `TStudent`.

Добавлять данные можно в начало, в конец или в нужное место списка. Во всех этих случаях необходимо корректировать указатели. На рис. 10.3 изображен процесс добавления элементов в начало односвязного списка.

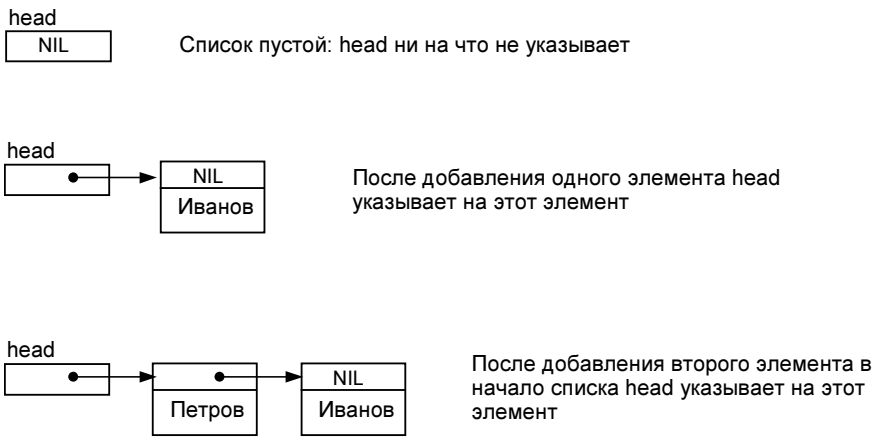


Рис. 10.3. Добавление элементов в список

Следующая программа, ее текст приведен в листинге 10.5, формирует список студентов, добавляя фамилии в начало списка. Данные вводятся с клавиатуры. Если вместо ввода очередной фамилии будет нажата клавиша <Enter>, что равносильно вводу пустой строки, то программа распечатывает введенный список.

Листинг 10.5. Формирование и отображение списка (p10_5.pas)

```
program p10_5;
type
     pStudent = ^TStudent;
     TStudent = record
          name:string [20];
          next:pStudent;
     end;
var
     head: pStudent;  { "голова" - начало списка }
     curr: pStudent;  { текущий элемент списка }
     st: string[20];  { строка, введенная пользователем }

begin
     repeat
          write('Фамилия-> ');
          readln(st);
          if length(st)<>0 then
               begin
                    new(curr);  { выделить память для элемента списка }
                    curr^.name := st;
                    curr^.next := head;
                    head := curr;
               end;
     until length(st)=0;

     writeln;
     writeln('Список:');

     curr := head;  { установить указатель текущего элемента
                      на первый элемент списка }
     while curr <> NIL do
        begin
          writeln(curr^.name );
          curr := curr^.next;    { переместить указатель текущего
                                   элемента на следующий элемент списка}
        end;

     write('Для завершения нажмите <Enter>');
     readln;
end.
```

Приведенная ранее программа добавляет элементы в начало списка. В результате элементы в списке размещаются в том порядке, в котором они были введены. Если нужен упорядоченный список (в таком списке легче найти требуемую информацию), то данные надо вводить по алфавиту, что не всегда удобно, да и возможно. Задачу формирования упорядоченного списка, добавление элемента в нужное место списка, можно возложить на программу.

Чтобы вставить элемент в нужное место списка, надо найти узел, у которого значение ключевого поля больше ключевого поля добавляемого узла, и установить указатель нового узла на этот узел. Затем надо установить указатель узла, после которого вставляется элемент, на новый узел (рис. 10.4).

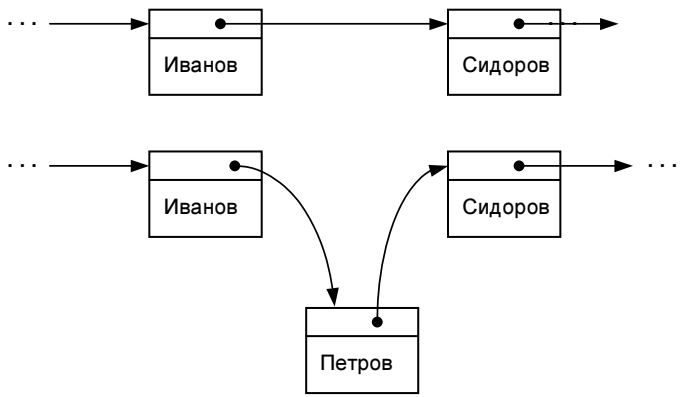


Рис. 10.4. Добавление элемента в упорядоченный список

В листинге 10.6 приведена программа, которая формирует упорядоченный по алфавиту список студентов. Порядок следования записей в списке определяется содержимым поля `name`.

Листинг 10.6. Формирование упорядоченного списка (p10_6.pas)

```
program p10_6;
type
     pStudent = ^TStudent;
     TStudent = record
          name:string [20];
          next:pStudent;
     end;
```

```
var
     head: pStudent;    { указатель на первый элемент списка }
     cur: pStudent;     { текущий элемент (перед которым вставляем) }
     p: pStudent;       { предыдущий элемент (после которого
                          вставляем новый узел }
     name: string[20]; { фамилия, вводимая с клавиатуры }
     node: pStudent;    { новый узел списка }
begin
     repeat
          write('Фамилия-> ');
          readln(name);
          if length (name)<>0 then
            begin
               { создадим новый элемент списка }
               new(node);
               node^.name := name;
               node^.next := NIL;

               { найти место для вставки }
               cur := head;
               p := NIL;
               { Список упорядочен по возрастанию, найдем
                 элемент, который больше чем добавляемый.
                 Перед ним и вставим.}
               while (cur^.name < name) and (cur <> NIL) do
                   begin
                     { введенное значение больше текущего }
                     p := cur;
                     cur := cur^.next; { к следующему узлу }
                   end;
               if p = NIL then
                  begin
                    { узел в начало списка }
                    node^.next:=head;
                    head:=node;
                  end
               else
                  begin
```

```
                node^.next := p^.next;
                p^.next := node;
            end;
        end;
    until length(name)=0; { строка не введена,
                            нажата клавиша < Enter > }

    { распечатаем введенный список }
    writeln;

    writeln('Список:');

    cur := head;
    while cur <> NIL do
       begin
         writeln(cur^.name);
         cur := cur^.next;
       end;

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

Чтобы удалить узел, надо скорректировать указатель узла, который находится перед удаляемым узлом (рис. 10.5).

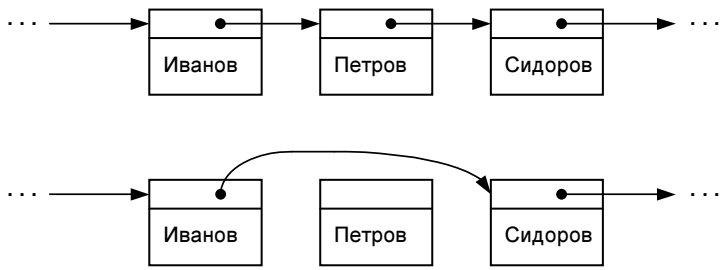


Рис. 10.5. Удаление элемента из списка

Память для узлов списка выделяется динамически. После удаления узла из списка, память, которую занимал узел, не используется. Для того чтобы эта память могла использоваться для размещения других динамических переменных, ее следует освободить. Если этого не сделать, то, возможно, в какой-то момент времени программа не сможет создать очередную динамическую переменную, поскольку для ее создания просто не будет памяти. Освобождение памяти выполняет процедура dispose. У процедуры dispose один параметр — указатель на переменную, память, занимаемая которой, должна быть освобождена.

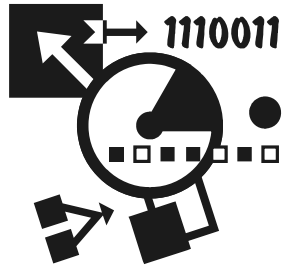
В листинге 10.7 приведена программа, которая демонстрирует удаление узла из списка. Сначала она формирует список, а затем удаляет узлы, содержащие данные о студентах, фамилии которых вводятся с клавиатуры. После исключения узла из списка программа освобождает память, которую занимает исключенный из списка узел.

Листинг 10.7. Удаление узла из списка (p10_7.pas)

```
program p10_7;
type

     TName = string[20];
     p_student = ^TStudent;

     TStudent = record
          name: TName;
          next: p_student;
     end;

{ добавление узла в список }
procedure AddToList(var head: p_student; name: TName);
   var
     curr: p_student;
   begin
     new(curr);
     curr^.name := name;
     curr^.next := head;
     head := curr;
   end;

{ вывод списка }
```

```
procedure PrintList(head: p_student);
   var
       curr: p_student;
   begin
       curr:= head;
       while curr <> NIL do
           begin
              writeln(curr^.name );
              curr := curr^.next;
           end;
   end;

{Удаление узла из списка }
function DelNode(var head: p_student; name:TName):boolean;
   var
     curr: p_student; { текущий элемент }
     p: p_student;    { предыдущий элемент }
     done: boolean;
   begin
     p := NIL;
     curr := head;
     done := FALSE;
     while (not done) and (curr<>NIL) do
         begin
            if curr^.name=name then
              begin
                 if p=NIL
                      then head:=curr^.next
                      else p^.next:=curr^.next;
                  done:=TRUE;
                  dispose(curr);
              end
             else begin
                p:=curr;
                curr:=curr^.next;
             end;
         end;
     DelNode:=done;
   end;
```

```
{ основная процедура }
var
     head: p_student; { начало списка }
     st:TName;

begin
     { ввод списка }

     repeat
          write('Фамилия-> ');
          readln(st);
          if length(st)<>0
               then AddToList(head,st);
     until length(st)=0;

     writeln;
     writeln('Список:');
     PrintList(head);

     writeln('Удаление узлов из списка');
     repeat
          write('Фамилия-> ');
          readln(st);
          if length(st)<>0 then
               if DelNode(head,st)
                    then writeln('Элемент "',st,'" удален из списка')
                    else writeln('Элемента "',st,'" в списке нет');
     until length(st)=0;

     writeln ('Список:');
     PrintList(head);

     write('Для завершения нажмите <Enter>');
     readln;
end.
```



Глава 11

Графика

Turbo Pascal позволяет программисту разрабатывать программы, которые помимо алфавитно-цифровой информации могут выводить на экран графическую информацию (графику), например, диаграммы, графики, схемы или чертежи.

Программа формирует графическое изображение из простых элементов (графических примитивов): прямых линий, окружностей, дуг, прямоугольников и отдельных точек.

Режим работы программы, в котором программа может выводить на экран графику, называется *графическим режимом*.

Видеосистема компьютера

Видеосистему компьютера образуют видеоадаптер и монитор. Видеоадаптер формирует сигналы, необходимые для вывода изображения, а монитор обеспечивает отображение графики. Основной характеристикой адаптера (точнее, режима его работы) является количество точек (по вертикали и горизонтали), из которых формируется изображение (табл. 11.1).

Таблица 11.1. *Режимы отображения графики*

Режим	Разрешение
CGA — Color Graphic Adapter	640×200
EGA — Enhanced Graphic Adapter	640×350
VGA — Video Graphic Array	640×480

Прикладная программа взаимодействует с видеоадаптером не напрямую, а через специальную программу — драйвер. Драйвер получает от прикладной программы универсальные команды, например, "подсветить точку с координатами

x, y", и преобразует их в последовательности элементарных действий, учитывающих особенности видеоадаптера.

Модуль Graph

Все функции и процедуры, а также именованные константы, необходимые для отображения графики, находятся в модуле Graph. Для того чтобы они стали доступны, модуль Graph надо подключить к программе — добавить в ее начало инструкцию `uses Graph`.

Инициализация графического режима

Чтобы программа могла вывести на экран графику, нужно инициализировать графический режим отображения информации. Инициализацию графического режима выполняет процедура `InitGraph`, инструкция вызова которой в общем виде выглядит так:

```
InitGraph(GraphDriver, GraphMode, PathToDriver);
```

Параметр *GraphDriver* (тип `integer`) задает драйвер видеоадаптера, параметр *GraphMode* (тип `integer`) — режим работы видеосистемы, а параметр *PathToDriver* (тип `string`) — место нахождения драйвера.

Например, инструкция инициализации отображения графики с разрешением 640×480 выглядит так:

```
InitGraph(VGA, VGAHi, 'c:\tp\bgi')
```

Следует обратить внимание, VGA и VGAHi — это определенные в модуле Graph именованные константы.

В Turbo Pascal драйверы видеоадаптеров различного типа находятся в каталоге BGI (сокращение от Borland Graphics Interface), в файлах с расширением bgi. Например, драйвер адаптера VGA представляет собой файл egavga.bgi.

Операция инициализации графического режима может завершиться неудачей (например, из-за того, что неправильно указан путь к файлу драйвера). Проверить результат инициализации графического режима можно, проверив значение функции `GraphResult`. Если инициализация выполнена успешно, то значение этой функции равно `grOk` (`grOk` — именованная константа).

Необходимо обратить внимание, что программа, установившая графический режим, перед тем как завершить свою работу, должна восстановить алфавитно-цифровой режим работы видеосистемы компьютера. Делается это вызовом процедуры `CloseGraph`.

В листинге 11.1 приведена программа, которая демонстрирует инициализацию и завершение графического режима. Она активизирует режим VGA

с разрешением 640×480, рисует узор из 100 прямоугольников, выводит сообщение и после нажатия пользователем клавиши <Enter> завершает работу.

Листинг 11.1. Инициализация и завершение графического режима (p11_1.pas)

```
{ Узор из 100 прямоугольников }
program p11_1;
uses Graph;
var
 grDriver:integer;  { драйвер }
 grMode:integer;    { графический режим }
 grPath:string;     { место расположения драйвера }
 ErrCode:integer;   { результат инициализации граф. режима }

 x,y: integer;  { координаты угла прямоугольника}
 c: integer;    { цвет прямоугольника (номер цвета в палитре) }

 i: integer;

begin
 grDriver := detect;     { режим VGA}

 grMode:=VGAHi;       { разрешение 640x480}

 grPath:='c:\tp\bgi'; { драйвер, файл EGAVGA.BGI, находится
                                        в каталоге c:\tp\bgi }

 InitGraph(grDriver, grMode, grPath);
 ErrCode := GraphResult;
 if ErrCode <> grOk then
   begin
     writeln('Ошибка инициализации графического режима.');
     writeln(GraphErrorMsg(ErrCode));
     writeln('Для завершения работы нажмите <Enter>');
     readln;
     Halt(1);
  end;

 { узор из 100 прямоугольников }
 Randomize;
 for i := 1 to 100 do
```

```
    begin
        x := Random(640);

        y := Random(480);
        c := Random(15) + 1;
        SetFillStyle(SolidFill,c); { установить цвет закраски }
        bar(x,y,x+10,y+10);        { нарисовать прямоугольник }
     end;

  OutTextXY(10,470,'Press <Enter>');
  readln;


 CloseGraph;
end.
```

Экран в графическом режиме

Графическая программа рассматривает экран монитора как совокупность отдельных точек — пикселов. Положение пиксела на экране характеризуется горизонтальной (X) и вертикальной (Y) координатами. Левый верхний пиксел имеет координаты (0,0). Координаты пикселов возрастают сверху вниз и слева направо (рис. 11.1). Количество пикселов на экране и, следовательно, значения координат правой нижней точки экрана зависят от режима работы видеосистемы. Например, в стандартном режиме VGA (разрешение 640×480) правый нижний пиксел имеет координаты (639,479).

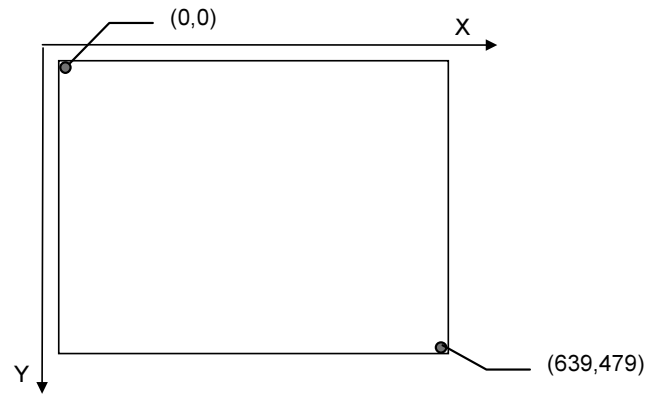


Рис. 11.1. Координаты точек (пикселов) экрана в графическом режиме

Указатель вывода

Когда программа работает в алфавитно-цифровом режиме, на экране монитора находится мигающий курсор. Он показывает знакоместо (позицию), в котором появится очередной символ, выведенный, например, инструкцией `write`. В графическом режиме курсор на экране не отображается. Точку, в которой появится текст в результате выполнения инструкции вывода текста, определяет невидимый *графический курсор*, который также называют *указателем вывода*. В начале работы программы указатель вывода находится в левом верхнем углу экрана, в точке (0,0). Получить координаты указателя вывода программа может, обратившись к функциям `GetX` и `GetY`. Переместить указатель в нужную точку экрана можно, вызвав процедуру `MoveTo`, задав в качестве параметров координаты точки экрана, куда надо переместить указатель. Например, инструкции

```
MoveTo(310,240);
OutText('Turbo Pascal');
```

выводят сообщение примерно в центре экрана.

Переместить указатель относительно его текущего положения можно, вызвав процедуру `MoveRel`, задав, на сколько точек надо переместить указатель. Инструкция вызова процедуры в общем виде выглядит так:

```
MoveRel(dx,dy);
```

Параметры *dx* и *dy* задают количество точек, на которое надо переместить указатель по горизонтали и вертикали относительно его текущего положения. Если значение параметра *dx* (*dy*) положительное, то указатель перемещается вправо (вниз). Если значение параметра *dx* (*dy*) отрицательное, указатель перемещается влево (вверх).

Графические примитивы

Любая картинка, чертеж, схема могут рассматриваться как совокупность *графических примитивов*: точек, линий, прямоугольников, окружностей, дуг и др. Таким образом, чтобы на экране появилась нужная картинка, программа должна нарисовать графические примитивы, составляющие эту картинку. Рисование (вычерчивание) на экране графических примитивов осуществляют соответствующие процедуры.

Цвет и вид линий

Цвет, который используется для вычерчивания линий (границ геометрических фигур, контуров), называется *текущим*. В начале работы программы

текущий цвет — белый. Для смены текущего цвета используется процедура SetColor. Инструкция смены текущего цвета в общем виде выглядит так:

```
SetColor(Цвет)
```

где *Цвет* — код цвета. В качестве параметра *Цвет* (тип integer) рекомендуется использовать одну из определенных в модуле Graph именованных констант (табл. 11.2).

Таблица 11.2. *Кодирование цвета*

Цвет	Константа	Номер цвета
Черный	Black	0
Синий	Blue	1
Зеленый	Green	2
Бирюзовый	Cyan	3
Красный	Red	4
Сиреневый	Magenta	5
Коричневый	Brown	6
Светло-серый	LightGray	7
Серый	DarkGray	8
Голубой	LightBlue	9
Светло-зеленый	LightGreen	10
Светло-бирюзовый	LightCyan	11
Светло-красный (алый)	LightRed	12
Светло-сиреневый	LightMagenta	13
Желтый	Yellow	14
Белый (яркий)	White	15

По умолчанию линия рисуется сплошной, толщиной в один пиксел. Программист может изменить вид (стиль линии). Стиль линии задается вызовом процедуры SetLineStyle, инструкция вызова которой в общем виде выглядит так:

```
SetLineStyle(ТипЛинии, Образец, Толщина);
```

Параметр *ТипЛинии* определяет вид линии. В качестве значения этого параметра рекомендуется использовать одну из именованных констант (табл. 11.3).

Таблица 11.3. *Кодирование вида линий*

Вид линии	Константа
Сплошная (непрерывная)	SolidLn
Пунктирная (с постоянной длиной штрихов)	DottedLn
Штрих-пунктирная линия	CenterLn
Пунктирная (длина штрихов чуть больше, чем у линии типа DottedLn)	DashedLn
Определенный программистом тип линии	UserBitLn

Параметр *Толщина* определяет толщину линии. Линия может быть обычной толщины (один пиксел), в этом случае в качестве параметра указывается именованная константа NormWidth, или утолщенная (константа ThickWidth).

Параметр *Образец* используется в том случае, если процедура SetLineStyle устанавливает тип линии, определяемый программистом (значение параметра *ТипЛинии* равно UserBitLn). Значением параметра *Образец* должна быть четырехразрядная шестнадцатеричная константа, которая кодирует отрезок линии длиной в 16 пикселов. На рис. 11.2 приведены вид линии, пиксельное изображение отрезка этой линии и соответствующая этому пиксельному изображению шестнадцатеричная константа.

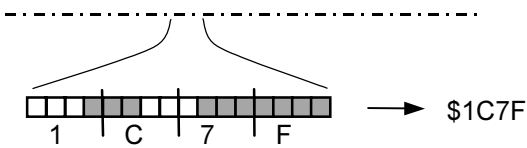


Рис. 11.2. Пример линии, определенной программистом

Цвет и стиль закраски области

Цвет, который используется для закраски внутренних областей геометрических фигур (прямоугольник, окружность) и контуров, называется цветом закраски областей. В начале работы программы цвет закраски областей — белый. Используя процедуру SetFillStyle, можно задать цвет и стиль (способ) закраски областей.

Инструкция вызова процедуры SetFillStyle в общем виде выглядит так:

```
SetFillStyle(Стиль, Цвет);
```

Параметр *Стиль* задает стиль закраски, например, сплошная заливка или вертикальная штриховка. Параметр *Цвет* задает цвет закраски. В качестве параметра *Стиль* рекомендуется использовать одну из именованных констант (табл. 11.4).

Таблица 11.4. *Стили закраски областей*

Стиль закраски	Константа
Без заливки (заливка цветом фона)	EmptyFill
Сплошная заливка текущим цветом	SolidFill
Горизонтальная штриховка	LineFill
Штриховка под углом 45 градусов влево тонкими линиями	LtSlashFill
Штриховка под углом 45 градусов влево	SlashFill
Штриховка под углом 45 градусов вправо тонкими линиями	BkSlashFill
Штриховка под углом 45 градусов вправо	LtBkSlashFill
Штриховка клеткой	HatchFill
Штриховка редкой косой клеткой под углом 45 градусов	XhatchFill
Штриховка частой косой клеткой под углом 45 градусов	InterleaveFill
Заполнение редкими точками	WideDotFill
Заполнение частыми точками	CloseDotFill
Тип заполнения определяется программистом	UserFill

Программист может создать свой стиль заполнения области и использовать его точно так же, как и стандартные стили. Для этого надо сначала сформировать описание битового образа (8×8 пикселов), который будет использоваться для заполнения области. Чтобы получить описание, надо нарисовать битовый образ, каждой строке поставить в соответствие двоичное число (закрашенная клетка — 1, незакрашенная — 0) и затем перевести полученные двоичные числа в шестнадцатеричные (рис. 11.3).

Полученные таким образом восемь чисел являются описанием битового образа.

Чтобы можно было использовать созданный программистом битовый образ для закраски областей, в программе надо объявить переменную типа `FillPatternType` (этот тип объявлен в модуле `Graph` и представляет собой массив, состоящий из восьми элементов типа `byte`) и выполнить ее инициализацию, например, так:

```
my_fill: FillPatternType = ($00,$10,$10,$FE,$38,$28,$44,$00);
```

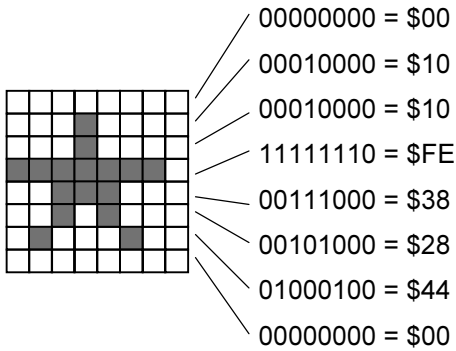


Рис. 11.3. Пример битового образа и его кодировка

После этого надо вызвать процедуру `SetFillStyle` и указать в качестве параметра *Стиль* имя переменной, содержащий описание битового образа.

Точка

Точку рисует процедура `PutPixel`, инструкция вызова которой в общем виде выглядит так:

```
PutPixel(x, y, Цвет)
```

Параметры `x`, `y` задают координаты точки, параметр *Цвет* — цвет.

Например, в результате выполнения инструкции `PutPixel(320,240,Red)` в центре экрана появится красная точка.

Линия

Вычерчивание линии может быть выполнено при помощи процедуры `Line` либо при помощи процедуры `LineTo`.

Процедура `Line` рисует линию между двумя точками экрана, координаты которых указаны в качестве параметров, процедура `LineTo` — из точки, в которой находится указатель вывода — в указанную.

В общем виде инструкция вызова процедуры `Line` выглядит так:

```
Line(x1, y1, x2, y2)
```

где *x1*, *y1* и *x2*, *y2* — координаты концов линии.

Процедура `LineTo` рисует линию из текущей точки (в которой находится указатель вывода) в заданную. Инструкция вызова процедуры `LineTo` в общем виде выглядит так:

```
LineTo(x,y)
```

где `x`, `y` — координаты точки, в которую надо провести линию.

Следует обратить внимание, что после того как линия будет нарисована, указатель вывода перемещается в точку конца линии. Для принудительного перемещения указателя вывода в нужную точку экрана используется процедура `MoveTo`. В качестве параметров процедуры `MoveTo` надо задать координаты точки, в которую необходимо переместить указатель вывода.

Цвет и вид линий, которые рисуют процедуры `Line` и `LineTo`, определяются текущим цветом и стилем рисования линий (по умолчанию — белая сплошная линия). Изменить цвет и стиль линии можно, вызвав соответственно процедуру `SetColor` и `SetLineStyle`. Установленный цвет и стиль рисования линий используются до тех пор, пока они не будут изменены.

Окружность

Окружность рисует процедура `Circle`. Инструкция ее вызова в общем виде выглядит так:

```
Circle(x,y,r);
```

Параметры *x* и *y* задают координаты центра окружности, *r* — радиус.

Цвет линии окружности определяется текущим цветом. Задать требуемый цвет линии окружности можно, вызвав процедуру `SetColor`.

Эллипс

Процедура `Ellipse` рисует эллипс. Инструкция ее вызова в общем виде выглядит так:

```
Ellipse(x, y, α1, α2, Rx, Ry)
```

Параметры *x* и *y* задают координаты центра эллипса, параметры *Rx*, *Ry* — горизонтальный и вертикальный радиусы эллипса, параметры *α1* и *α2* — угловые координаты начальной и конечной точек линии эллипса (рис. 11.4).

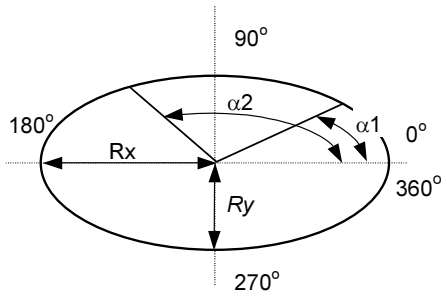


Рис. 11.4. Параметры процедуры `Ellipse`

Эллипс рисуется против часовой стрелки от начальной точки к конечной. Если значения параметров *α1* и *α2* будут равны соответственно 0 и 360, то будет нарисован полный эллипс, в противном случае — дуга.

Например, инструкция `Ellipse(100,100,0,360,20,50)` рисует эллипс, инструкция `Ellipse(100,100,0,180,20,50)` — верхнюю половину эллипса (дугу), а инструкция `Ellipse(100,100,180,0,20,50)` — нижнюю половину.

При равенстве значений параметров *Rx* и *Ry* процедура `Ellipse` рисует окружность. Например, инструкция `Ellipse(100,100,0,90,50,50)` вычерчивает дугу окружности.

Прямоугольник

Прямоугольник можно нарисовать, вызвав процедуру `Rectangle`, `Bar` или `Bar3D`.

Процедура `Rectangle` рисует прямоугольник. Инструкция ее вызова в общем виде выглядит так:

```
Rectangle(x1,y1,x2,y2)
```

где *x1* и *y1* — координаты левого верхнего угла прямоугольника, `x2` и `y2` — координаты правого нижнего угла прямоугольника. Например, инструкция

```
Rectangle(0,0,GetMaxX,GetMaxY);
```

вычерчивает рамку вокруг рабочей области экрана. В приведенном примере `GetMaxX` и `GetMaxY` — это функции, значениями которых являются максимальные значения координат `X` и `Y` для текущего графического режима.

Процедура `Bar` рисует закрашенный прямоугольник. В общем виде инструкция ее вызова выглядит так:

```
Bar(x1,y1,x2,y2);
```

Параметры *x1* и *y1* задают положение левого верхнего угла прямоугольника, а *x2* и *y2* — правого нижнего.

Цвет закраски прямоугольника определяется текущим цветом закраски областей. Задать требуемый цвет прямоугольника и способ его закраски можно, вызвав процедуру `SetFillStyle`. Например, в результате выполнения следующих инструкций

```
SetFillStyle(SolidFill,Green);
Bar(10,50,20,100);
```

будет нарисован зеленый прямоугольник.

Процедура `Bar3D` рисует объемный прямоугольник (параллелепипед). В общем виде инструкция вызова процедуры выглядит так:

```
Bar3D(x1,y1,x2,y2,Глубина,ВерхняяГраница);
```

Параметры *x1* и *y1* задают положение левого верхнего, а *x2* и *y2* — правого нижнего углов передней грани параллелепипеда. Параметр *Глубина* задает расстояние между передней и задней гранями параллелепипеда, а параметр *ВерхняяГраница* (типа `Boolean`) определяет, нужно ли вычерчивать границу верхней грани параллелепипеда.

Например, инструкции

```
SetFillStyle(LineFill, Blue);
Bar3D(20,50,40,100,10,TRUE);
```

рисуют объемный прямоугольник, передняя грань которого заштрихована горизонтальными линиями синего цвета.

Следует обратить внимание, что видимые ребра параллелепипеда рисуются цветом, установленным процедурой `SetColor`.

Круг и сектор

Процедура `PieSlice` рисует круговой сектор. Инструкция ее вызова в общем виде выглядит так:

```
PieSlice(x,y, α1, α2, R);
```

Параметры *x* и *y* задают положение центра сектора, параметр *R* — радиус сектора, параметры *α1* и *α2* — углы прямых, ограничивающих сектор (рис. 11.5). Сектор рисуется против часовой стрелки от линии, положение которой определяет параметр *α1*, к линии, положение которой определяет параметр *α2*.

Цвет и стиль закраски сектора можно задать, вызвав процедуру `SetFillStyle`, цвет границы сектора — `SetColor`.

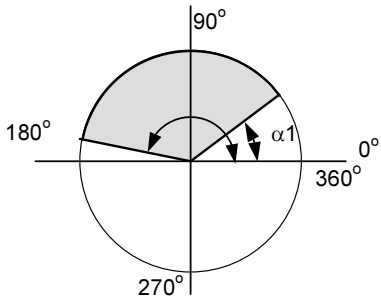


Рис. 11.5. Параметры процедуры PieSlice задают вид сектора

Если значения параметров *α1* и *α2* будут равны соответственно 0 и 360, то процедура `PieSlice` нарисует круг. Однако при этом будет видна линия радиуса, проведенная из центра круга в направлении возрастания значения координаты X. Чтобы ее скрыть, надо установить цвет границы линии контура (`SetColor`) такой же, как и цвет закраски сектора (`SetFillStyle`).

Эллипс и эллиптический сектор

Процедура `Sector` рисует эллиптический сектор. Инструкция ее вызова процедуры в общем виде выглядит так:

```
Sector(x, y, α1, α2, Rx, Ry)
```

Параметры *x* и *y* задают координаты центра эллипса, частью которого является сектор, параметры *Rx*, *Ry* — горизонтальный и вертикальный радиусы этого эллипса, параметры *α1* и *α2* — углы прямых, ограничивающих сектор (рис. 11.6). Сектор рисуется против часовой стрелки от линии, положение которой определяет параметр *α1*, к линии, положение которой определяет параметр *α2*.

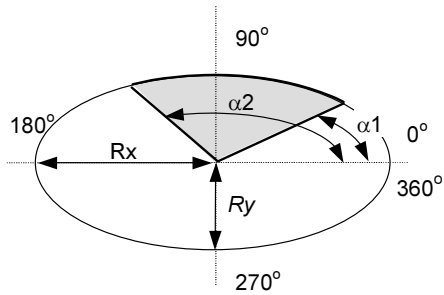


Рис. 11.6. Параметры процедуры Sector задают вид сектора

Если значения параметров *α1* и *α2* будут равны соответственно 0 и 360, то процедура `Sector` нарисует эллипс. Однако при этом будет видна линия радиуса, проведенная из центра эллипса в направлении возрастания значения координаты X. Чтобы ее скрыть, надо установить цвет границы линии контура (`SetColor`) такой же, как и цвет закраски сектора (`SetFillStyle`).

Вывод текста

Инструкции *WRITE* и *WRITELN*

Программа, работающая в графическом режиме, может вывести информацию, в том числе и значения переменных, при помощи инструкций `write` и `writeln`.

Первая инструкция программы, имеется в виду первая инструкция вывода, выводит текст от левого верхнего угла экрана, следующая — от точки, в которой находится последний символ, выведенный предыдущей инструкцией вывода, или, если предыдущий вывод был выполнен инструкцией `writeln`, с новой строки (хотя говорить о строках экрана, когда программа работает в графическом режиме, не совсем корректно).

Процедуры *OutText* и *OutTextXY*

Для вывода текстовой информации в графическом режиме используются процедуры `OutTextXY` и `OutText`.

Процедура `OutText` выводит строку текста, начиная с точки, в которой находится указатель вывода. После выполнения процедуры `OutText` указатель вывода автоматически перемещается в правый верхний угол области, куда был выведен последний символ строки (рис. 11.7).

Следует отметить, что процедура `OutText` не переносит на следующую строку символы, которые не помещаются в текущей строке. Поэтому возможна ситуация, когда на экран будет выведена не вся строка, а только ее часть (символы, не помещающиеся в текущей строке, не отображаются).

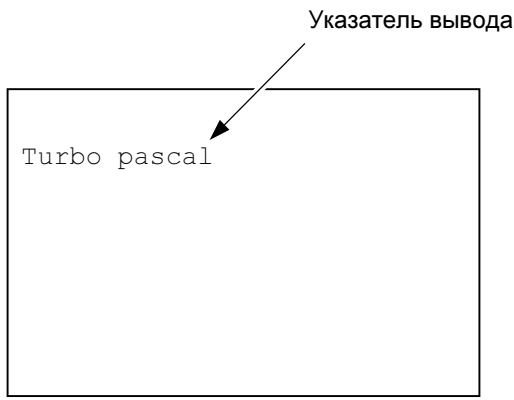


Рис. 11.7. Положение указателя вывода после вывода текста

Процедура `OutTextXY` позволяет вывести строку в нужную область экрана. В общем виде инструкция ее вызова выглядит так:

`OutTextXY(`*`x`*`,`*`y`*`,`*`Строка`*`)`

Параметры `x` и `y` процедуры `OutTextXY` задают координаты левого верхнего угла области вывода текста (рис. 11.8). Следует обратить внимание, что процедура `OutTextXY` не перемещает указатель вывода.

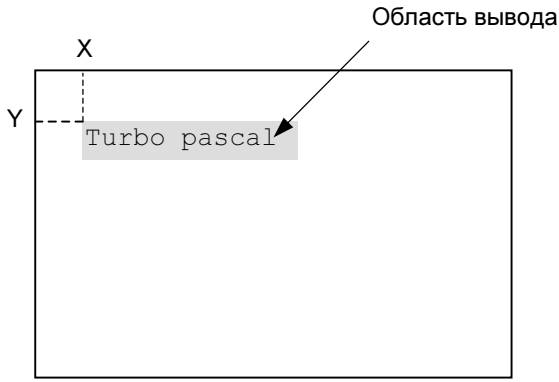


Рис. 11.8. Координаты области вывода текста

Параметр *Строка* процедуры `OutText` и `OutTextXY`, который определяет выводимую на экран информацию, строкового типа. Поэтому, если надо вывести значение числовой переменной, это значение сначала надо преобразовать в строку, а затем эту строку указать в качестве параметра процедуры `OutText (OutTextXY)`. Преобразование численного значения в строку обеспечивает процедура `Str`, инструкция вызова которой в общем виде выглядит так:

```
Str(Значение:Формат, Строка)
```

Параметр *Значение* задает переменную, значение которой надо преобразовать в строку. Необязательный параметр *Формат* задает формат изображения численного значения: для дробных чисел — общее количество цифр и количество цифр дробной части числа; для целых чисел — количество символов. Параметр *Строка* определяет переменную строкового типа, в которую будет записана строка — изображение числа.

Ниже приведен фрагмент кода, который демонстрирует использование процедуры `Str` для вывода на экран значения дробной переменной.

```
Str(d:6:2,st);        { преобразовать численное значение в строку }
OutTextXY(10,32,st); { вывести строку}
```

Иногда возникает необходимость заменить или удалить текст с экрана. Попытка вывести поверх выведенного текста новый или строку пробелов завершается неудачей — старый текст виден сквозь новый. Чтобы удалить текст с экрана, надо при помощи процедуры `Bar` закрасить область, в которой

отображается старый текст, цветом фона. Ниже приведен фрагмент кода, демонстрирующий, как заменить старый текст новым. Для определения размера области, которую надо закрасить, используются функции `TextWidth` и `TextHeight`. Значения этих функций — соответственно ширина и высота области вывода текста.

```
msg := 'To start press <Enter>'; { сообщение }
OutTextXY(10,10,msg);            { вывести сообщение }
readln;

{ задать стиль и цвет закраски области}
SetFillStyle(SolidFill,GetBkColor);
{ закрасить область, в которой находится сообщение}
Bar(10,10,10+TextWidth(msg),10+TextHeight(msg));
```

Программист может задать характеристики шрифта, который будут использовать процедуры `OutText` и `OutTextXY`. Делается это с помощью процедуры `SetTextStyle`, инструкция вызова которой в общем виде выглядит так:

`SetTextStyle(`*`Шрифт`*`, `*`Ориентация`*`, `*`Размер`*`)`

Параметр *`Шрифт`* (именованная константа) задает шрифт, который используется для отображения текста (табл. 11.5). Следует обратить внимание на то, что шрифты TriplexFont, SmallFont, SansSerifFont и GothicFont являются векторными и в них нет букв русского алфавита. Поэтому для вывода сообщений на русском языке можно использовать только стандартный (DefaultFont) шрифт. Файлы шрифтов (chr-файлы) находятся в подкаталоге tp\bgi.

Таблица 11.5. *Шрифты*

Константа	Шрифт
`DefaultFont`	Стандартный (каждый символ отображается в области 8×8 пикселов)
`TriplexFont`	Triplex
`SmallFont`	Мелкий
`SansSerifFont`	SansSerif
`GothicFont`	Готический

Параметр *`Ориентация`* функции `SetTextStyle` задает ориентацию выводимого текста. Если в качестве параметра указать именованную константу `VertDir` (ее численное значение равно единице), то строки выводимого процедурами

OutText и OutTextXY текста будут расположены вертикально, причем текст будет выводиться вверх от точки вывода (рис. 11.9).

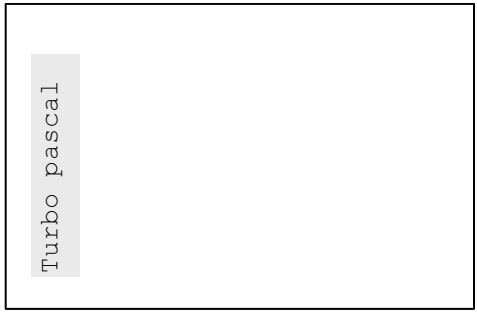


Рис. 11.9. Пример вертикально ориентированного текста

Для возврата в режим обычной ориентации текста используется константа HorizDir (ее численное значение равно нулю).

Параметр *Размер* функции SetTextStyle позволяет установить размер символов выводимого процедурами OutText и OutTextXY текста.

Примеры программ

График

Графики обычно используют для наглядного представления данных, которые меняются во времени (например, изменение температуры воздуха, цены какого-либо товара, курса валюты и т. д.). В листинге 11.2 приведена программа, которая строит график изменения температуры воздуха (рис. 11.10). Данные, отображаемые на графике, загружаются из текстового файла meteo.txt, каждая строка которого содержит одно значение температуры (предполагается, что в файле находится информация за последние 7 дней). Отличительной особенностью программы является то, что она строит график таким образом, чтобы он занимал всю область экрана, выделенную для графика.

Сначала программа загружает данные из файла в массив и определяет диапазон изменения значений данных (минимальное и максимальное значения), которые надо отобразить на графике. Затем, для того чтобы график занимал всю область, предназначенную для него, по формуле h / (max - min) вычисляется коэффициент масштабирования. В приведенной формуле: h — высота области отображения графика; max и min — максимальное и мини-

мальное значения ряда данных. Например, если диапазон изменения температуры от –5 до + 12 и для отображения графика используется область высотой 300 пикселей, то значение коэффициента масштабирования равно 17.65. Коэффициент масштабирования используется для вычисления координат точек графика: $y_i = y_0 + h - k_y * (D_i - D_{min})$, где: y_0 — координата Y левого верхнего угла области построения графика; h — высота области построения графика; k_y — коэффициент масштабирования; D_{min} — минимальное значение ряда данных, D_i — значение, которое изображает точка графика. Если, например, $y_0 = 50$, а $h = 300$, то координаты точек, изображающих значения –5, 0 и +12, равны, соответственно, 350, 261 и 50. Следует обратить внимание, что значение координаты Y, вычисленное по приведенной формуле, дробного типа. Но аргументы процедур, обеспечивающих вычерчивание элементов графика, должны быть целого типа. Поэтому для преобразования дробного значения в целое следует использовать функцию `Round`, значением которой является целое, полученное округлением аргумента.

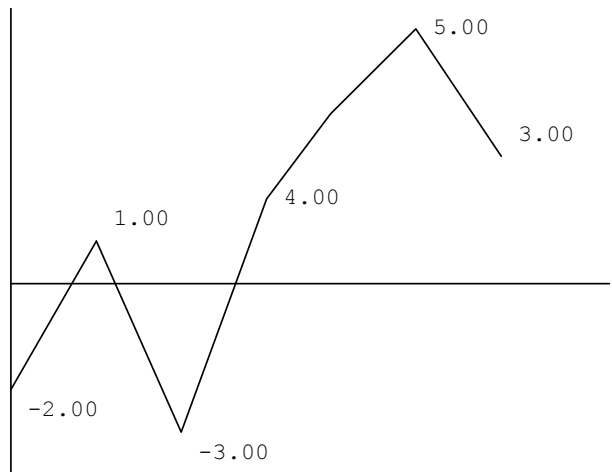


Рис. 11.10. График изменения температуры

Листинг 11.2. График (p11_2.pas)

```
{ График (изменение температуры) }
program p11_2;
uses Graph;

const
  HB=7;
```

```
var
 grDriver:integer;  { драйвер }
 grMode:integer;    { графический режим }
 grPath:string;     { место расположения драйвера }
 ErrCode:integer;   { результат инициализации граф. режима }

 f: text;    { файл данных }
 fn: string; { имя файла }

 d: array[1..HB] of real; { данные }
 min,max: real;           { минимальное и максимальное значения }
 ky: real;                { коэффициент масштабирования }

 { координаты точки }
 x: array[1..HB] of integer;
 y: array[1..HB] of integer;

 dx: integer;

 { область построения }
 x0,y0: integer; { левый верхний угол }
 h,w: integer;   { высота и ширина }

 i: integer;
 st: string;

 yh: integer;

begin
    { загрузка данных из файла }
    fn:= 'meteo.txt';
    assign(f,fn);

    {$I-}
    reset(f);   { открыть файл данных для чтения }
    {$I+}
    if IOResult <> 0 then
        begin
                writeln('ошибка доступа к файлу: файл ',fn, ' не найден');
```

```
            write('Для завершения нажмите <Enter>');
            readln;
            halt(1); { завершение работы программы }
      end;

  i:=0;
  while (not EOF(f)) and (i < HB) do
      begin
           i := i + 1;
           readln(f, d[i]);
      end;
  close(f);

  grDriver := detect;  { режим VGA}
  grMode:=VGAHi;       { разрешение 640x480}
  grPath:='c:\tp\bgi'; { драйвер, файл EGAVGA.BGI, находится
                                        в каталоге c:\tp\bgi }

  InitGraph(grDriver, grMode, grPath);
  ErrCode := GraphResult;
  if ErrCode <> grOk then
      begin
          writeln('Ошибка инициализации графического режима.');
          writeln(GraphErrorMsg(ErrCode));
          writeln('Для завершения работы нажмите <Enter>');
          readln;
          Halt(1);
      end;

  min := d[1];
  max := d[1];
  for i := 2 to HB do
  begin
     if d[i] < min then min := d[i];
     if d[i] > max then max := d[i];
  end;

x0 := 50;
y0 := 50;
```

```
  h := 200;
  w := 300;

  dx := Round(w / (HB - 1));

  ky := h / (max - min);

  x[1] := x0;
  y[1] := y0 + h - Round(ky*(d[1]-min));

  for i:=2 to HB do
     begin
          x[i] := x[i-1] + dx;
          y[i] := y0 + h - Round(ky*(d[i]-min));
     end;

  MoveTo(x[1],y[1]);
  for i:=2 to HB do
     begin
          LineTo(x[i],y[i]);
     end;

  for i:=1 to HB do
     begin
          Str(d[i]:5:2,st);
          if d[i] > 0
             then OutTextXY(x[i]+5,y[i]-5,st)
             else OutTextXY(x[i]+5,y[i]+5,st);
     end;

  yh := y0 + h - Round(ky*(0 - min));
  Line(x0,yh,x0+w,yh);

  Line(x0,y0,x0,y0+h);

  OutTextXY(10,470,'Press <Enter>');
  readln;

 CloseGraph;
end.
```

Анимация

Анимация (дословно оживление) — это технология создания "живых" изображений. Анимированное изображение, или анимация, в отличие от обычной картинки, элементы которой статичны, представляет собой изображение, элементы которого ведут себя подобно объектам реального мира.

Существует два подхода к реализации анимации. Первый (классический) предполагает наличие заранее подготовленной серии картинок (кадров), на которых изображены фазы движения объекта, последовательное отображение которых и создает эффект анимации. Этот подход используют создатели мультфильмов. Второй подход, который используют разработчики компьютерных игр, предполагает создание кадров анимации "на лету", во время работы программы. При реализации этого подхода изображение фаз движения объектов формируется из графических примитивов непосредственно на "экране", в качестве которого обычно используется фоновый рисунок.

Движение — один из основных анимационных эффектов. Реализовать его достаточно просто: сначала нужно нарисовать объект (вывести изображение объекта), затем через некоторое время стереть и снова вывести, но уже на некотором расстоянии от его первоначального положения. Подбором времени между выводом и удалением изображения, а также расстояния между новым и предыдущим положением объекта, можно добиться того, что у наблюдателя (зрителя) будет складываться впечатление, что объект равномерно движется.

Следующая программа, ее текст приведен в листинге 11.3, показывает, как можно реализовать движение объекта по экрану. Объект (окружность) движется от левой границы экрана к правой. Окружность рисует процедура `Circle`. Стирание окружности выполняется перерисовкой окружности цветом фона. Задержку между вычерчиванием и стиранием окружности обеспечивает процедура `Delay`, которой в качестве параметра передается величина (в миллисекундах) необходимой задержки.

Листинг 11.3. Движение простого объекта (p11_3.pas)

```
{ Анимация (движение простого объекта)  }
program p11_3;
uses Graph, Crt;
var
 x,y,r:integer;      { координаты центра и радиус окружности }
 dx:integer;         { шаг перемещения окружности }
```

```
 dt:integer;         { задержка перерисовки на новом месте }

 grDriver:integer;   { драйвер }
 grMode:integer;     { графический режим }
 grPath:string;      { место расположения драйвера }
 ErrCode:integer;    { результат инициализации граф. режима }

 msg: string;
begin
 grDriver := VGA;      { режим VGA}
 grMode:=VGAHi;        { разрешение 640x480}
 grPath:='c:\tp\bgi'; { драйвер, файл EGAVGA.BGI, находится
                        в каталоге d:\tp\bgi }

 InitGraph(grDriver, grMode,grPath);
 ErrCode := GraphResult;
 if ErrCode <> grOk then
   begin
     writeln('Ошибка инициализации графического режима.');
     writeln('Для завершения нажмите <Enter>');
     readln;
     Halt(1);
  end;

 x:=0;     { начальное положение окружности }
 y:=100;
 r:=5;     { радиус окружности }
 dx:=2;    { шаг перемещения   }
 dt:=800; { задержка }


 msg := 'To start press <Enter>';
 SetColor(Lightgray);
 OutTextXY(10,10,msg);
 readln;

 { стереть сообщение }
 SetFillStyle(SolidFill,Black);
```

```
 Bar(10,10,10+TextWidth(msg),10+TextHeight(msg));

 while x < 639 do
    begin
       { нарисовать окружность }
       SetColor(Yellow);
       Circle(x,y,r);

       { задержка }
       Delay(dt);

       { стереть окружность }
       SetColor(0);
       Circle(x,y,r);

       { изменить положение центра окружности }
       x:=x+dx;
    end;

 SetColor(Lightgray);
 OutTextXY(10,10,'To exit press <Enter>');

 readln;
 CloseGraph;
end.
```

При программировании движения сложных объектов, состоящих из множества элементов, используется метод "базовой точки". Суть метода заключается в следующем. Выбирается некоторая точка изображения объекта, которая принимается за базовую. Координаты остальных точек отсчитываются от базовой точки. Следует обратить внимание, если координаты точек относительно базовой отсчитывать в относительных единицах, то появляется возможность масштабирования изображения.

На рис. 11.11 приведено изображение яхты. Базовой точкой является точка (x_o,y_o). Координаты остальных точек отсчитываются именно от нее.

В листинге 11.4 приведен текст программы, которая выводит на экран изображение плывущей яхты.

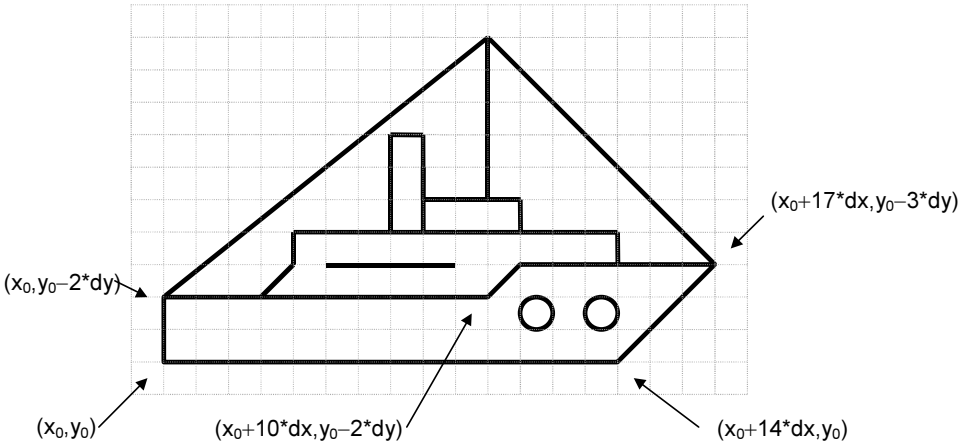


Рис. 11.11. Пример определения координат изображения относительно базовой точки

Листинг 11.4. Движение объекта (p11_4.pas)

```
{ Движение объекта }
program p11_4;
uses Graph,
     Crt;

{ рисует яхту }
Procedure Ship(x,y:integer;     { координаты базовой точки }
                  color:word); { цвет  }
const
  dx=3;
  dy=3;
var
  OldColor:word;
begin
  OldColor:= GetColor; { сохранить текущий цвет }
  SetColor(color);     { установить новый цвет }

  {  корпус }
  MoveTo(x,y);
  LineTo(x,y-2*dy);
  LineTo(x+10*dx,y-2*dy);
  LineTo(x+11*dx,y-3*dy);
```

```
  LineTo(x+17*dx,y-3*dy);
  LineTo(x+14*dx,y);
  LineTo(x,y);
  { надстройка }
  MoveTo(x+3*dx,y-2*dy);
  LineTo(x+4*dx,y-3*dy);
  LineTo(x+4*dx,y-4*dy);
  LineTo(x+13*dx,y-4*dy);
  LineTo(x+13*dx,y-3*dy);
  Line(x+5*dx,y-3*dy,x+9*dx,y-3*dy);
  { капитанский мостик }
  Rectangle(x+8*dx,y-4*dy,x+11*dx,y-5*dy);
  { труба}
  Rectangle(x+7*dx,y-4*dy,x+8*dx,y-7*dy);
  { иллюминаторы }
  Circle(x+12*dx,y-2*dy,Trunc(dx/2));
  Circle(x+14*dx,y-2*dy,Trunc(dx/2));
  { мачта }
  Line(x+10*dx,y-5*dy,x+10*dx,y-10*dy);
  { оснастка }
  MoveTo(x+17*dx,y-3*dy);
  LineTo(x+10*dx,y-10*dy);
  LineTo(x,y-2*dy);
  SetColor(OldColor); { восстановить текущий цвет }
end;

var
 grDriver:integer;  { драйвер }
 grMode:integer;    { графический режим }
 grPath:string;     { место расположения драйвера }
 ErrCode:integer;   { результат инициализации граф. режима }

 x,y:integer;  { координаты кораблика }
 color:word;   { цвет кораблика }
 bkcolor:word; { цвет фона экрана }

 msg: string;

begin
 grDriver := VGA;      { режим VGA}
 grMode:=VGAHi;        { разрешение 640x480}
```

```
 grPath:='c:\tp\bgi'; { драйвер, файл EGAVGA.BGI, находится
                        в каталоге d:\bp\bgi }

 InitGraph(grDriver, grMode,grPath);
 ErrCode := GraphResult;
 if ErrCode <> grOk then
   begin
     writeln('Ошибка инициализации графического режима.');
     writeln('Для завершения нажмите <Enter>');
     readln;
     Halt(1);
  end;

 x:=10;
 y:=200;
 color:=LightGray;
 SetBkColor(Blue);
 bkcolor:=GetBkColor;

msg := 'To start press <Enter>';
 SetColor(Lightgray);
 OutTextXY(10,10,msg);
 readln;

 { стереть сообщение }
 SetFillStyle(SolidFill,GetBkColor);
 Bar(10,10,10+TextWidth(msg),10+TextHeight(msg));

 repeat
  Ship(x,y,color);      { нарисовать яхту }
  Delay(2000);
  Ship(x,y,bkcolor);    { стереть яхту }
  x:=x+2;
 until (x > 640);

 OutTextXY(10,10,'To exit press <Enter>');
 readln;
 CloseGraph;
end.
```

Вывод и стирание изображения кораблика выполняет процедура `Ship`, которая получает в качестве параметров координаты базовой точки и цвет, которым надо нарисовать яхту. Если цвет отличается от цвета фона экрана, то процедура рисует яхту, а если совпадает — то стирает. В процедуре `Ship` объявлены константы `dx` и `dy`, определяющие шаг (в пикселах), используемый при вычислении координат точек изображения. Изменив значения этих констант, можно задать требуемый размер изображения.

Выполняемый файл графической программы

Для того чтобы отлаженная программа могла быть запущена не только из среды разработки, но и из операционной системы, ее надо откомпилировать на диск. Однако здесь следует обратить внимание на то, что графическая программа использует драйвер, местоположение которого указывается в тексте программы. Если по какой-либо причине программа не сможет найти драйвер в указанном каталоге или сам каталог (такое может быть, например, если программа будет перенесена на другой компьютер), то на экран будет выведено сообщение об ошибке и программа завершит свою работу. Предупредить эту ситуацию можно несколькими способами. Наиболее простой из них — поместить копию драйвера в тот каталог, где находится exe-файл программы. Другой способ — встроить драйвер в выполняемый файл.

Чтобы встроить драйвер в выполняемую программу, надо сначала преобразовать файл драйвера в OBJ-файл, затем — поместить в текст программы инструкции, обеспечивающие встраивание и регистрацию драйвера.

Преобразование файла драйвера (BGI-файла) в файл OBJ-формата можно выполнить при помощи входящей в состав Turbo Pascal утилиты binobj. Файл binobj.exe находится в подкаталоге BIN каталога, в который установлен Turbo Pascal. Например, если Turbo Pascal установлен на диск C:, то утилита binobj должна находиться в каталоге C:\TP\BIN. Команда запуска утилиты набирается в окне командной строки, для доступа к которому в меню **File** надо выбрать команду **Dos shell**. В результате этих действий окно Turbo Pascal будет закрыто и станет доступным окно командной строки (рис. 11.12).

В строке DOS надо набрать команду запуска утилиты binobj и параметры. Команда запуска — это имя выполняемого файла. Параметры определяют имя исходного (преобразуемого) файла, имя создаваемого OBJ-файла и имя процедуры-драйвера. Например, команда преобразования EGAVGA драйвера выглядит так:

```
c:\tp\bin\binobj.exe c:\tp\bgi\egavga.bgi c:\tp\obj\egavga.obj EGAVGADriverProc
```

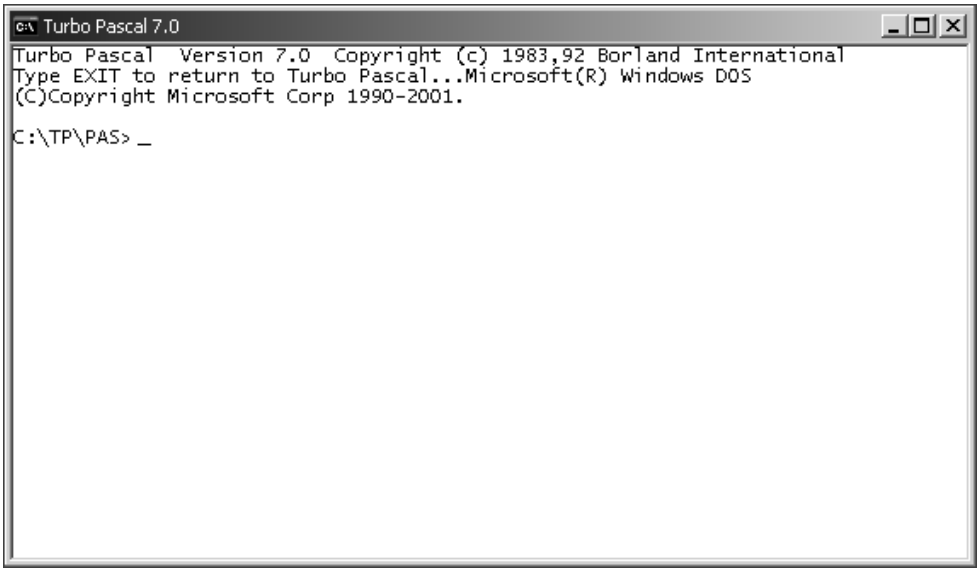


Рис. 11.12. Вид экрана после перехода в режим командной строки DOS

В приведенном примере `c:\tp\bin\` — это имя каталога, в котором находится утилита `binobj.exe`, `c:\tp\bgi\` — путь к BGI-файлу драйвера, `c:\tp\obj` — имя каталога, в который будет помещен obj-файл, а `egavga.obj` — имя создаваемого OBJ-файла. Последний параметр, `EGAVGADriver`, задает имя процедуры-драйвера. Это имя должно использоваться программой, в которую встраивается драйвер, для регистрации драйвера.

Рисунок 11.13 демонстрирует процесс преобразования драйвера. Следует обратить внимание, что во время набора команды текущим каталогом является каталог `c:\tp` (его имя указано в строке приглашения). Поэтому имя каталога в команде не указано (выход из каталога `pas`, переход на один уровень вверх, обеспечивает команда `cd . .`).

Для выхода из режима командной строки и продолжения работы в среде Turbo Pascal в строке приглашения надо набрать `exit` и нажать <Enter>.

После создания OBJ-файла драйвера в текст программы нужно добавить директиву включения в выполняемый файл OBJ-файла, инструкцию объявления драйвера и инструкцию регистрации драйвера. В качестве примера в листинге 11.5 приведена простая программа, при компиляции которой на диск драйвер будет включен в exe- файл.

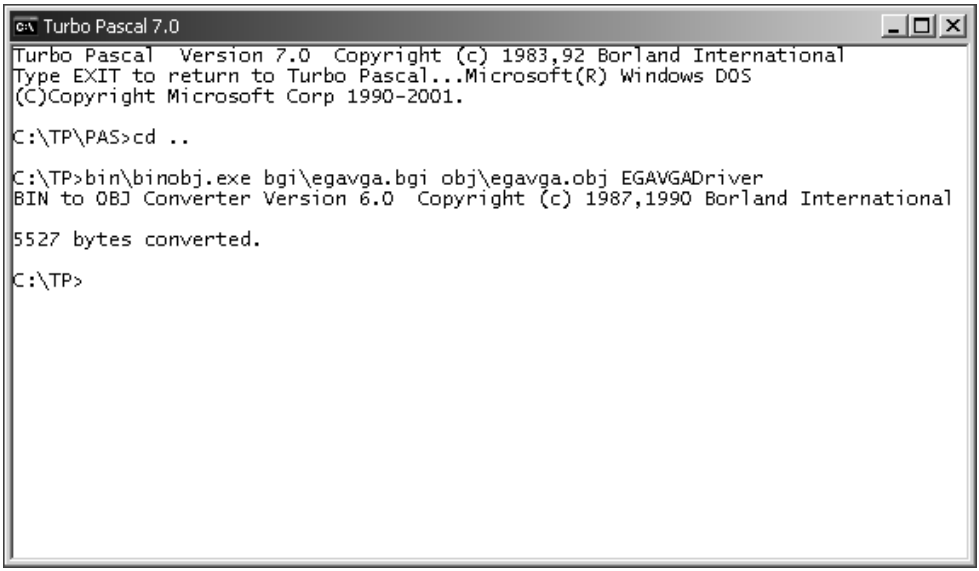


Рис. 11.13. Вид экрана после создания OBJ-файла драйвера

Листинг 11.5. Включение драйвера в exe-файл (p11_5.pas)

```
{ включение драйвера в exe-файл }
program p11_5;

uses Graph;

var
  GraphDriver, GraphMode, Error : integer;

procedure EGAVGADriver; external;

{$L EGAVGA.OBJ}

var
   x,y,r,c: integer; { координаты, радиус и цвет окружности }
   i: integer;
begin

  if RegisterBGIdriver(@EGAVGADriver) < 0 then
```

```
  begin
    writeln('EGA/VGA: ',GraphErrorMsg(GraphResult));
    halt(1);
  end;

  GraphDriver := Detect;
  InitGraph(GraphDriver, GraphMode, '');
  if GraphResult <> grOk then
  begin
    Writeln('Ошибка инициализации графического режима: ',
             GraphErrorMsg(GraphDriver));
    Halt(1);
  end;

  Randomize;
  for i := 1 to 100 do
     begin
         x := Random(640);
         y := Random(480);
         c := Random(15) + 1;
         SetColor(c);          { установить цвет линии }
         Circle(x,y,5);        { нарисовать окружность }

      end;

  OutTextXY(10,460,'To exit press <Enter>');
  Readln;
  CloseGraph;
end.
```

Директива `{$L EGAVGA.OBJ}` указывает компилятору, что в exe-файл надо включить файл EGAVGA.OBJ, т. е. драйвер. Если в директиве включения obj-файла путь к включаемому файлу не указан, то компилятор будет искать OBJ-файл в том каталоге, из которого загружен компилируемый pas-файл, и если его там нет, то в каталоге, имя которого указано в поле **Object directories** окна **Directories** (рис. 11.14), которое становится доступным в результате выбора в меню **Options** команды **Directories**.

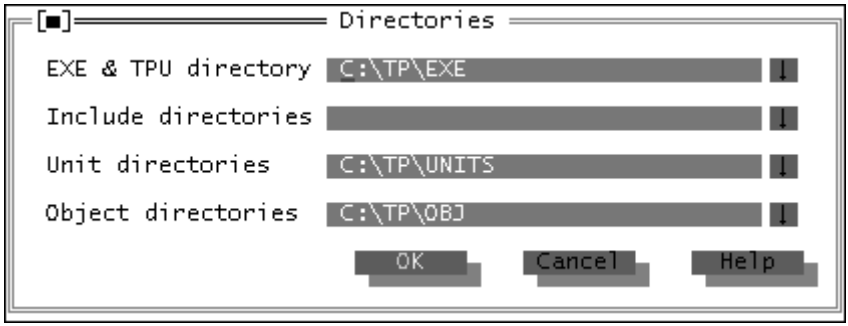


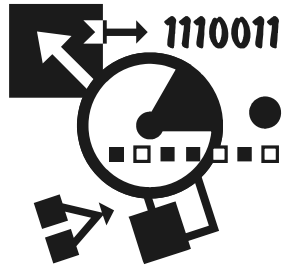
Рис. 11.14. Окно **Directories**

Инструкция

```
procedure EGAVGADriver; external;
```

объявляет процедуру `EGAVGADriverProc` и указывает, что она является внешней (`external`), т. е. инструкции, реализующие процедуру, находятся в другом файле. Следует обратить внимание, что имя этой процедуры должно совпадать с именем, указанным в команде создания OBJ-файла драйвера.

Регистрацию драйвера выполняет функция `RegisterBGIDriver`, которой в качестве параметра передается адрес драйвера (символ `@`, поставленный перед именем функции, означает, что это не имя функции, а адрес ячейки памяти, в которой находится первая инструкция функции).



Глава 12

Рекурсия

Понятие рекурсии

Рекурсивным называется объект, частично состоящий или определяемый с помощью самого себя. Факториал — пример рекурсивного объекта. Факториал числа n определяется как произведение целых чисел от 1 до n и обозначается n!:

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n.$$

Приведенное выражение можно переписать так:

$$n! = n \times ((n-1) \times (n-2) \times ... \times 3 \times 2 \times 1) = n \times (n-1)!$$

То есть *факториал* числа n равен произведению числа n на *факториал* числа (n–1). В свою очередь, факториал числа (n–1) — это произведение числа (n–1) на факториал числа (n–2) и т. д.

Таким образом, если вычисление факториала n реализовать как функцию, то в теле этой функции будет инструкция вызова функции вычисления факториала числа (n–1), т. е. функция будет вызывать сама себя. Такой способ вызова называется *рекурсией*, а функция, которая обращается сама к себе, называется *рекурсивной функцией*.

В листинге 12.1 приведена рекурсивная функция вычисления факториала.

Листинг 12.1. Рекурсивная функция вычисления факториала (p12_1.pas)

```
function factorial(k:integer):longint;
     begin
          if k = 1
               then factorial:=1
               else factorial:= k*factorial(k-1);
     end;
```

Обратите внимание, что функция вызывает сама себя только в том случае, если значение параметра больше единицы. Если значение параметра равно единице, то функция сама себя не вызывает, а возвращает значение, и рекурсивный процесс завершается.

Следующая программа, ее текст приведен в листинге 12.2, используя рекурсивную функцию `factorial`, выводит на экран таблицу факториалов (рис. 12.1).

Листинг 12.2. Использование рекурсивной функции (p12_2.pas)

```
{ таблица факториалов (использование рекурсивной функции) }
program p12_2;

{ Рекурсивная функция "Факториал" }
function factorial(k:integer):longint;
     begin
          if k = 1
               then factorial:=1
               else factorial:= k*factorial(k-1);
     end;

var
     i:integer;      { число, факториал которого надо вычислить}
     f:longint;      { факториал }

begin
     writeln('Таблица факториалов');
     writeln;
     for i:=1 to 12 do
         begin
              f:=factorial(i);
              writeln(i:2,'  ', f);
         end;

     writeln;
     write('Для завершения нажмите <Enter>');
     readln;
end.
```

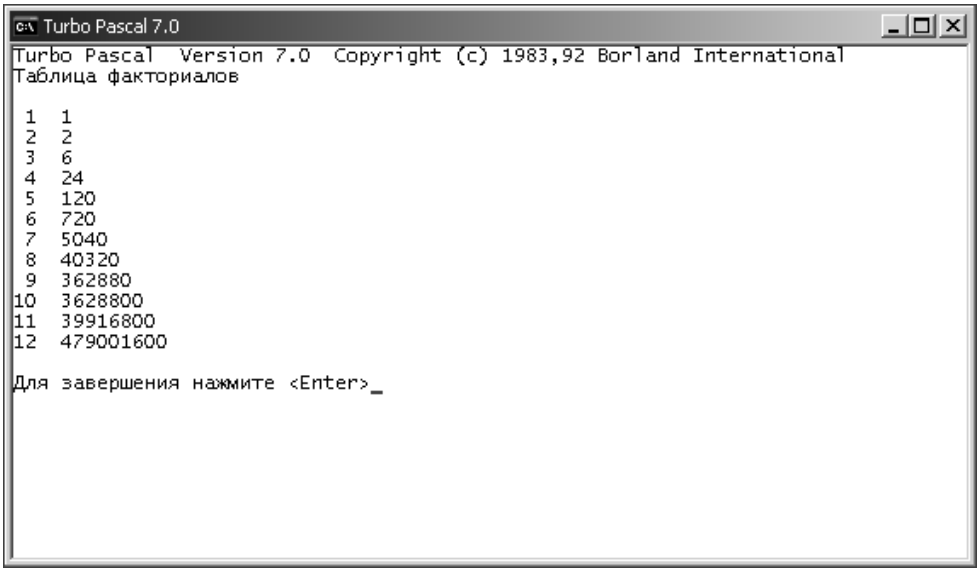


Рис. 12.1. Таблица факториалов

Пример программы: поиск пути

Механизм рекурсии весьма эффективен при программировании различных задач поиска. В качестве примера рассмотрим задачу поиска пути между двумя населенными пунктами. Если города соединены дорогами, то очевидно, что попасть из одного города в другой, проходя по каждой из дорог не более одного раза, можно по различным маршрутам. Задача состоит в том, что надо найти все возможные маршруты.

Карту дорог между городами можно изобразить в виде графа — набора вершин, обозначающих города, и ребер, обозначающих дороги (рис. 12.2).

Процесс поиска маршрута может быть представлен как последовательность отдельных шагов. На каждом шаге с использованием некоторого критерия выбирается точка, в которую можно попасть из текущей, и включается в маршрут. Если очередная выбранная точка совпала с заданной конечной точкой, то маршрут найден. Если не совпала, то делается следующий шаг. Так как текущая точка может быть соединена с несколькими другими, то сначала будем выбирать точку с наименьшим номером (это и есть критерий выбора).

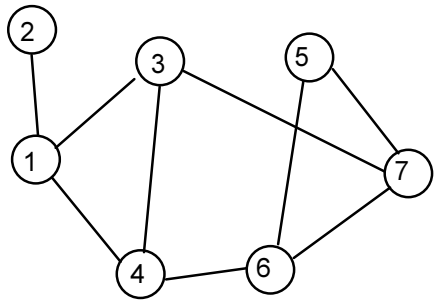


Рис. 12.2. Представление карты дорог между населенными пунктами в виде графа

Процесс решения задачи поиска маршрута рассмотрим на примере. Пусть надо найти все возможные пути из точки 1 в точку 5. Согласно принятому правилу, сначала выбираем точку 2. На следующем шаге делаем попытку найти точку, в которую можно перейти из точки 2, и выясняем, что точка 2 тупиковая, поэтому возвращаемся в точку 1 и делаем шаг в точку 3. Из точки 3 в точку 4, из 4 — в 6 и из точки 6 в точку 5. Один маршрут найден. После этого возвращаемся в точку 6 и проверяем, возможен ли шаг в точку, отличную от 5. Так как это возможно, то делаем шаг в 7 и затем в 5. Найден еще один путь. Таким образом, процесс поиска состоит из шагов вперед и возвратов назад. Поиск завершается, если из узла начала движения уже некуда идти.

Алгоритм поиска имеет рекурсивный характер — чтобы сделать шаг, мы выбираем точку и опять делаем шаг, до тех пор пока не достигаем цели.

Таким образом, задача поиска маршрута может рассматриваться как задача выбора очередной точки (города) и поиска оставшейся части маршрута, т. е. имеет место рекурсия.

В программе граф (карту) можно представить двумерным массивом `map` (карта). Значение элемента массива `map[i,j]` — единица, если города `i` и `j` соединены прямой дорогой, или ноль, если города не соединены дорогой. Для приведенного графа массив `map` можно изобразить в виде матрицы (таблицы) так:

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	1	0	0	0	0	0	0
3	1	0	0	1	0	0	1
4	1	0	1	0	0	1	0
5	0	0	0	0	0	1	1
6	0	0	0	1	1	0	1
7	0	0	1	0	1	1	0

Содержимое ячейки таблицы на пересечении строки i и столбца j соответствует значению `map[i, j]`.

Помимо массива `map` нам потребуется массив `road` (дорога) и массив `incl` (от слова include — включать).

В `road[i]` мы будем записывать номера пройденных городов. В момент достижения конечной точки он будет содержать номера всех пройденных точек, т. е. описание маршрута.

В `incl[i]` будем записывать `TRUE`, если точка с номером i включена в маршрут. Делается это для того, чтобы не включать в маршрут уже пройденную точку (не ходить по кругу).

Так как мы используем рекурсивную процедуру, то надо обратить особое внимание на условие завершения рекурсивного процесса. Процедура должна прекратить вызывать сама себя, если текущая точка совпала с заданной конечной точкой.

На рис. 12.3 приведен алгоритм выбора очередной точки формируемого маршрута.

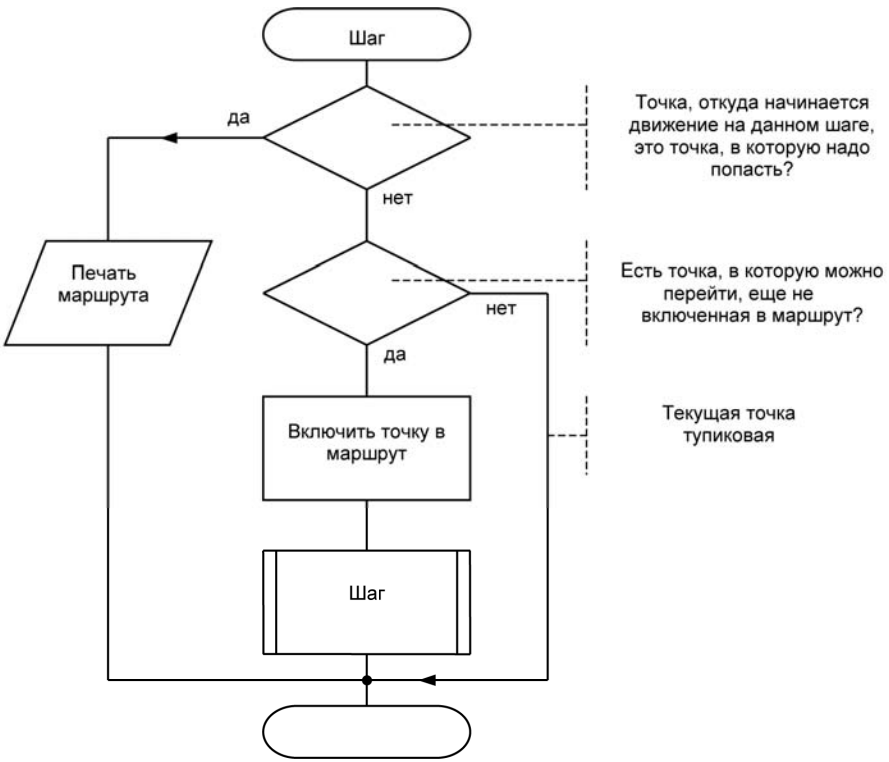


Рис. 12.3. Алгоритм процедуры выбора очередной точки

Программа поиска всех возможных маршрутов между двумя точками графа приведена в листинге 12.3.

Листинг 12.3. Поиск всех маршрутов между двумя точками графа (p12_3.pas)

```
{ Поиск всех маршрутов между двумя точками графа }
program p12_3;

const
     N=7;{ кол-во вершин графа}
var
     map:array[1..N,1..N] of integer; { граф (карта) map[i,j] не 0,
                                        если точки i и j соединены }
     road:array[1..N] of integer;{ маршрут - номера точек карты }
     incl:array[1..N] of boolean;{ incl[i]=TRUE, если точка
                                   с номером i включена в road }

     start,finish:integer; { начальная и конечная точки (откуда и куда) }

     i,j:integer;


procedure step(s,f,p:integer);{ s - точка, из которой делается шаг}
                              { f - точка, куда надо попасть (конечная) }
                              { p - номер искомой точки маршрута}
var
     c:integer;{ Номер точки, в которую делается очередной шаг }
begin
     if s=f then
       begin
          {Точки s и f совпали!}
          write('Маршрут: ');
          for i:=1 to p-1 do write(road[i],' ');
          writeln;
       end
     else
       begin
          { Выбираем очередную точку }
```

```
          for c:=1 to N do
             begin { Проверяем все вершины }
                   if(map[s,c]<>0)and(NOT incl[c])
                   { Точка соединена с текущей и не включена }
                   { в маршрут}
                   then
                      begin
                        road[p]:=c;   { Добавим точку в маршрут }
                        incl[c]:=TRUE;{ и пометим ее  }
                                      { как включенную }
                        step(c,f,p+1);
                        incl[c]:=FALSE;
                        road[p]:=0;
                      end;
               end;
       end;
end; { процедура step }

{ Основная программа }
begin
     { Инициализация массивов }
     for i:=1 to N do road[i]:=0;
     for i:=1 to N do incl[i]:=FALSE;

     for i:=1 to N do
        for j:=1 to N do map[i,j]:=0;

     { ввод карты }
     map[1,2]:=1; map[2,1]:=1;
     map[1,3]:=1; map[3,1]:=1;
     map[1,4]:=1; map[4,1]:=1;
     map[3,4]:=1; map[4,3]:=1;
     map[3,7]:=1; map[7,3]:=1;
     map[4,6]:=1; map[6,4]:=1;
     map[5,6]:=1; map[6,5]:=1;
     map[5,7]:=1; map[7,5]:=1;
```

```
    map[6,7]:=1; map[7,6]:=1;

    writeln('Поиск маршрута');
    write('Начальная точка ->');
    readln(start);
    write('Конечная точка ->');
    readln(finish);
    writeln;

    road[1]:=start;        { внесем точку в маршрут }
    incl[start]:=TRUE;     { и пометим ее как включенную }

    step(start,finish,2); { ищем вторую точку маршрута }

    writeln;
    writeln('Для завершения нажмите <Enter>');
    readln;
end.
```

Пример работы программы поиска маршрутов между двумя населенными пунктами приведен на рис. 12.4.

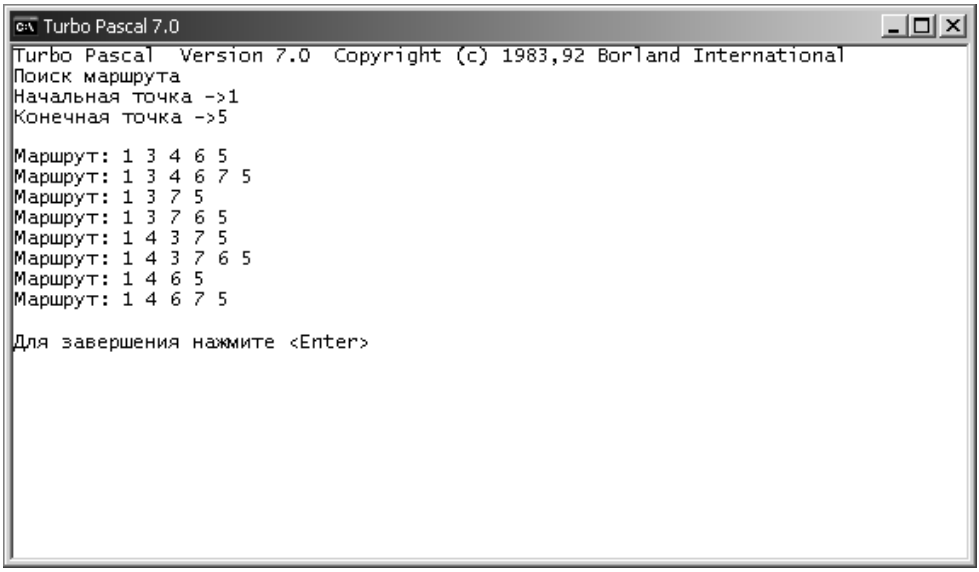


Рис. 12.4. Поиск маршрутов между двумя населенными пунктами (точками графа)

Пример программы: поиск кратчайшего пути

После того как все маршруты найдены, можно выбрать один наилучший. Обычно под наилучшим понимают самый короткий маршрут. Если необходимо найти кратчайший маршрут, то совсем не обязательно искать все маршруты. Нужно во время выбора очередной точки проверить, не превысит ли длина формируемого маршрута длину уже найденного, если эта точка будет включена в маршрут, и если превысит, то эту точку следует пропустить и выбрать следующую. Таким образом, после того как будет найден первый маршрут, программа будет вести поиск только по тем ветвям графа, которые могут улучшить найденное решение, отсекая пути, делающие формируемый маршрут длиннее уже найденного.

Программа поиска кратчайшего маршрута (пути минимальной длины) приведена в листинге 12.4, а пример ее работы — на рис. 12.5.

Листинг 12.4. Поиск кратчайшего маршрута между двумя точками графа (p12_4.pas)

```
{ поиск кратчайшего пути между двумя точками графа }
program p12_4;
const
     N=7;{ количество вершин графа}
var
     map:array[1..N,1..N] of integer;{ Карта: map[i,j] не 0,
                                       если точки i и j соединены }
     road:array[1..N] of integer;{ Маршрут - номера точек карты }
     incl:array[1..N] of boolean;{ incl[i]=TRUE, если точка )
                                  { с номером i включена в road }
     len:integer;{ Длина последнего найденного маршрута }
     c_len:integer;{ Длина текущего маршрута }

     start,finish:integer;{ Начальная и конечная точки }

     i,j:integer;

procedure step(s,f,p:integer);
var
     c:integer;{ Номер точки,в которую делаем очередной шаг }
```

```
begin
     if s=f then
          begin
               len:=c_len;{ сохраним длину найденного маршрута }
               write('Маршрут: ');
               for i:=1 to p-1 do write(road[i],' ');
               writeln(' Длина: ',len);
          end
     else
          { выбираем очередную точку }
          for c:=1 to N do { Проверяем все вершины }
               if(map[s,c]<>0) AND (NOT incl[c]) AND
                     ((len=0) OR (c_len+map[s,c]<len))
                    { Точка соединена с текущейне включена в маршрут}
                    then begin
                         road[p]:=c;{ Добавим вершину в путь }
                         incl[c]:=TRUE;{ Пометим вершину }
                                       { как включенную }
                         c_len:=c_len+map[s,c];
                         step(c,f,p+1);
                         incl[c]:=FALSE;
                         road[p]:=0;
                         c_len:=c_len-map[s,c];
                    end;
end;{ Конец процедуры step }

{ Основная программа }
begin
     { Инициализация массивов }
     for i:=1 to N do road[i]:=0;
     for i:=1 to N do incl[i]:=FALSE;

     for i:=1 to N do
        for j:=1 to N do map[i,j]:=0;

     { ввод карты }
     map[1,2]:=1; map[2,1]:=1;
     map[1,3]:=1; map[3,1]:=1;
     map[1,4]:=1; map[4,1]:=1;
     map[3,4]:=1; map[4,3]:=1;
```

```
    map[3,7]:=1; map[7,3]:=1;
    map[4,6]:=1; map[6,4]:=1;
    map[5,6]:=1; map[6,5]:=1;
    map[5,7]:=1; map[7,5]:=1;
    map[6,7]:=1; map[7,6]:=1;


    writeln('Поиск маршрута');
    write('Начальная точка ->');
    readln(start);
    write('Конечная точка ->');
    readln(finish);
    writeln;

    road[1]:=start;       { внесем точку в маршрут }
    incl[start]:=TRUE;    { и пометим ее как включенную }

    len:=0;
    c_len:=0;
    step(start,finish,2); { ищем вторую точку маршрута }

    writeln;
    write('Для завершения нажмите <Enter>');
    readln;
end.
```

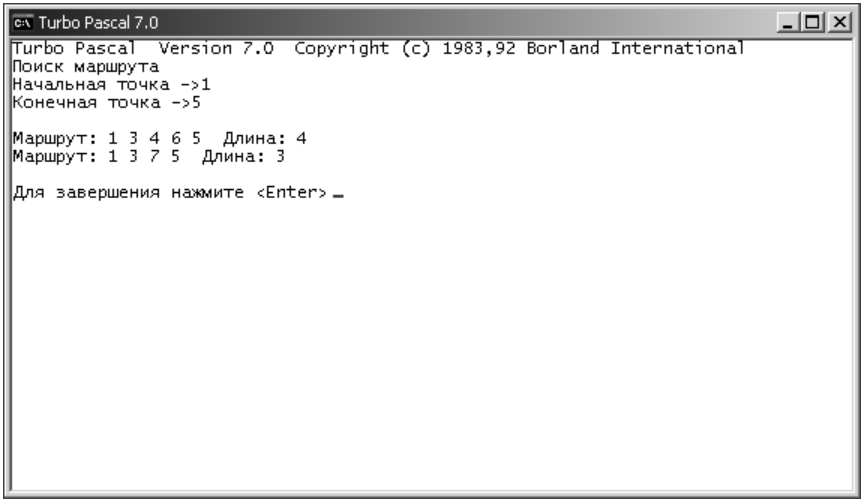
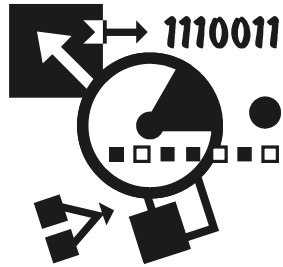


Рис. 12.5. Поиск кратчайшего маршрута
между двумя населенными пунктами (точками графа)



Глава 13

Отладка программы

Под отладкой понимается процесс поиска и устранения ошибок в программе.

Ошибки, которые могут быть в программе, принято делить на три группы:

- синтаксические ошибки
- ошибки времени выполнения
- алгоритмические ошибки

Синтаксические ошибки, их также называют ошибками времени компиляции (Compile-time error), — наиболее легко устранимые. Их обнаруживает компилятор. Программисту остается только внести изменения в текст программы и выполнить повторную компиляцию.

Ошибки времени выполнения (Run-time error) тоже, как правило, легко устранимы. Они обычно проявляются уже при первых запусках программы и во время тестирования. Если ошибка возникает в программе, которая запущена из среды разработки программы, то среда разработки прерывает работу программы и устанавливает курсор в строку с инструкцией, выполнение которой вызвало ошибку. Если программа запущена не из среды разработки, а, например, из окна командной строки, то при возникновении ошибки на экран выводится сообщение вида:

```
Runtime error 106 at 0000:01B5
```

В приведенном примере 106 — это код ошибки, десятичное число, значение которого однозначно определяет ошибку, `0000:01B5` — адрес ячейки памяти компьютера, где находится инструкция, выполнение которой привело к возникновению ошибки. Чтобы выяснить, выполнение какой инструкции вызвало ошибку, надо запустить среду разработки, загрузить исходную программу, в меню **Search** (Найти) выбрать команду **Find error** (Поиск ошибки). Затем в поле **Error address** (Адрес ошибки) появившегося диалогового окна **Find Error**

(рис. 13.1) надо ввести адрес инструкции, вызвавшей ошибку, и щелкнуть на командной кнопке **OK**. В результате этих действий курсор будет установлен на инструкцию программы, выполнение которой вызвало ошибку.

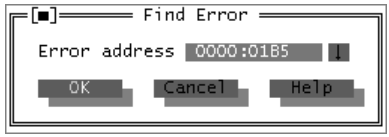


Рис. 13.1. Окно **Find Error**

С алгоритмическими ошибками дело обстоит иначе. Компиляция программы, в которой есть алгоритмическая ошибка, завершается успехом. При пробных запусках программа ведет себя вроде бы как надо, но выдает неправильный результат.

Turbo Pascal предоставляет программисту мощное средство поиска и устранения ошибок в программе — отладчик. Отладчик позволяет выполнять *трассировку* программы, наблюдать значения переменных, контролировать выводимые программой данные.

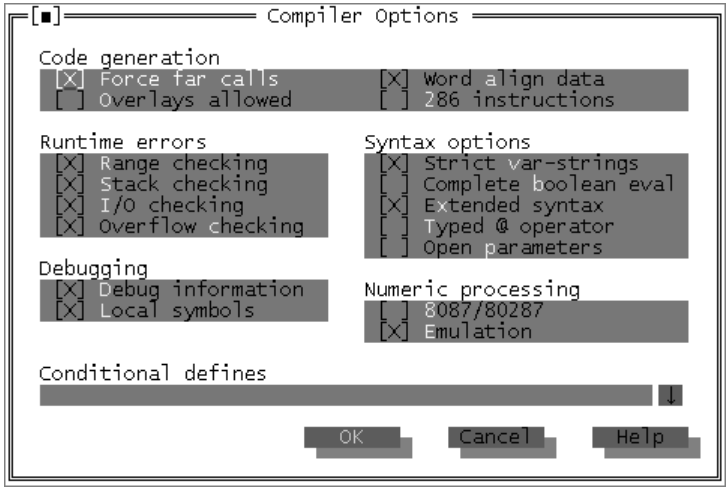


Рис. 13.2. Окно **Compiler Options**

Для того чтобы можно было использовать отладчик, компилятор должен добавить в выполняемую программу необходимую отладчику некоторую информацию. Чтобы компилятор это сделал, нужно в окне **Compiler Options** (рис. 13.2), которое становится доступным в результате выбора в меню **Options** команды **Compiler**, установить во включенное состояние переключатель **Debug information** (Отладочная информация). Также необходимо ус-

тановить во включенное состояние переключатели **Range checking** (контроль диапазонов), **Stack checking** (контроль стека), **I/O checking** (контроль операций ввода-вывода) и **Overflow checking** (контроль переполнения, выход значений за границы допустимого диапазона).

Трассировка программы

В результате запуска программы инструкции программы выполняются одна за другой со скоростью работы процессора компьютера. При этом программист не может определить, какая инструкция выполняется в данный момент, и, следовательно, определить, соответствует ли реальный порядок выполнения инструкций разработанному им алгоритму. В случае неправильной работы программы необходимо видеть реальный порядок выполнения инструкций. Это можно сделать, выполнив трассировку программы. Трассировка — это процесс выполнения программы по шагам (step-by-step), инструкция за инструкцией. Во время трассировки команду выполнить очередную инструкцию программы дает программист.

Turbo Pascal обеспечивает два режима трассировки: без захода в подпрограмму (Step over) и с заходом в подпрограмму (Trace into). Режим трассировки без захода в подпрограмму выполняет трассировку только главной программы, при этом трассировка подпрограмм не выполняется, вся подпрограмма выполняется за один шаг. В режиме трассировки с заходом в подпрограмму выполняется трассировка всей программы, т. е. по шагам выполняется не только главная программа, но и все подпрограммы.

Чтобы начать трассировку, нужно в меню **Run** выбрать команду **Step over** или **Trace into**. В результате этого в окне редактора текста программы будет выделена инструкция, которая будет выполнена первой. Чтобы выполнить выделенную инструкцию, надо в меню **Run** выбрать команду **Step over** или **Trace into**. После выполнения инструкции будет выделена следующая. Таким образом, выбирая нужную команду в меню **Run**, можно выполнить трассировку программы. Активизировать и выполнить трассировку можно при помощи функциональной клавиатуры. Команде **Step over** соответствует клавиша <F8>, команде **Trace into** — <F7>.

В любой момент времени можно завершить трассировку и продолжить выполнение программы в реальном темпе. Для этого надо в меню **Run** выбрать команду **Run**.

Если надо выполнить трассировку части программы, то следует установить курсор на инструкцию программы, с которой надо начать трассировку, и в меню **Run** выбрать команду **Go to cursor** или нажать <F4>. Затем, нажимая <F7> или <F8>, выполнить трассировку нужного фрагмента программы.

Во время трассировки можно наблюдать не только порядок выполнения инструкций программы, но и значения переменных. О том, как это сделать, рассказывается в одном из следующих разделов.

Точки останова программы

При отладке широко используется метод, который называют *методом точек останова*. Суть метода заключается в том, что программист помечает некоторые инструкции программы (ставит точки останова), при достижении которых программа приостанавливает свою работу, и программист может начать трассировку или проконтролировать значения переменных.

Добавление точки останова

Для того чтобы поставить в программу точку останова (breakpoint), надо в меню **Debug** (Отладка) выбрать команду **Add breakpoint** (Добавить точку останова). В появившемся окне (рис. 13.3) будет выведена информация о добавляемой точке останова: в поле **File name** — имя файла программы, куда добавляется точка останова; в поле **Line number** — номер строки текста программы, в которую добавляется точка останова. О назначении полей **Condition** (Условие) и **Pass count** (Число пропусков) будет сказано чуть позже.

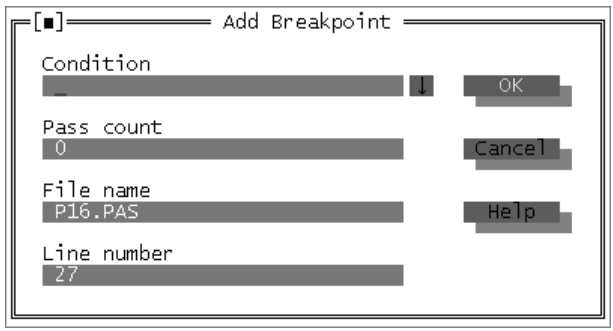


Рис. 13.3. Пример диалогового окна **Add Breakpoint**

Теперь, если нажать командную кнопку **OK**, точка останова будет добавлена в программу, при этом строка, в которой находится точка останова, выделяется цветом.

Для точки останова можно задать условие, при выполнении которого программа приостановит свою работу в данной точке, например, если значение переменной будет равно некоторому значению. Условие (выражение логического

типа) следует ввести в поле **Condition** окна **Add Breakpoint**. Если условие задано, то процесс выполнения программы будет приостановлен, если выражение, находящееся в поле **Condition**, будет истинно (значение равно `TRUE`).

Кроме условия для точки останова можно задать количество пропусков данной точки. Если в поле **Pass count** (Число пропусков) записать отличное от нуля число, то программа приостановит свою работу в этой точке только после того, как инструкция, находящаяся в строке, помеченной точкой останова, будет выполнена указанное количество раз.

Изменение характеристик точки останова

Программист может изменить характеристики точки останова, например условие или количество пропусков. Чтобы это сделать, надо в меню **Debug** выбрать команду **Breakpoints**, затем в появившемся диалоговом окне **Breakpoints** выбрать точку останова, характеристики которой надо изменить, и нажать кнопку **Edit** (Редактировать). В результате этих действий открывается окно **Edit Breakpoint** (Изменить точку останова), в котором можно изменить характеристики точки. После того как будут установлены требуемые значения характеристик точки останова, следует нажать командную кнопку **Modify** (Изменить).

Удаление точки останова

Чтобы удалить точку останова, надо установить курсор в ту строку программы, где находится эта точка, и нажать <Ctrl>+<F8>. Другой способ — в меню **Debug** выбрать команду **Breakpoints**, в появившемся диалоговом **Breakpoints** выбрать нужную точку и нажать командную кнопку **Delete** (Удалить). Используя окно **Breakpoints**, можно быстро удалить все точки останова. Для этого надо нажать командную кнопку **Clear all** (Удалить все).

Наблюдение за выводом программы

Во время трассировки (выполнения программы по шагам) можно следить за выводимой программой информацией двумя способами. Можно просмотреть содержимое окна пользователя (для этого надо нажать <Alt>+<F5> или из меню **Debug** выбрать команду **User screen** (Окно программы)). Однако лучше открыть отдельное окно (окно вывода), в которое программа будет выполнять вывод.

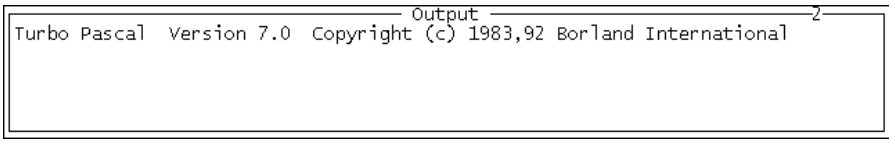


Рис. 13.4. Окно **Output**

Чтобы открыть окно вывода, надо в меню **Debug** выбрать команду **Output** (Окно вывода). Появившееся окно вывода (рис. 13.4), как и другие окна среды разработки, можно перемещать по экрану, также можно менять его размер.

При использовании окна вывода рекомендуется изменить размер окна редактора кода таким образом, чтобы одновременно были видны два окна: окно программы и окно вывода. В этом случае во время выполнения программы по шагам будут видны оба окна. Наиболее просто изменить положение и размер окна можно при помощи мыши. Чтобы изменить размер окна, надо установить указатель мыши на изображение правого нижнего угла, нажать левую кнопку мыши и, удерживая ее нажатой, перемещать мышь. После установки требуемого размера окна надо отпустить кнопку мыши. Аналогичным образом окно перемещается по экрану. При этом курсор следует установить на изображение верхней границы окна.

Наблюдение значений переменных

При выполнении программы по шагам часто бывает полезно знать, чему равно значение той или иной переменной. Отладчик позволяет наблюдать значения переменных программы.

Чтобы во время выполнения программы по шагам иметь возможность видеть значение переменной, надо в меню **Debug** выбрать команду **Add Watch**, в поле **Watch expression** появившегося окна **Add Watch** (рис. 13.5) ввести имя переменной, значение которой надо контролировать, и нажать кнопку **OK**. Затем следует открыть окно **Watches** (в меню **Debug** выбрать команду **Watch**) и изменить его размер так, чтобы одновременно были видны окно редактора кода и окно **Watches**. После этого надо активизировать режим трассировки. Следует обратить внимание, что сразу после добавления в окно **Watches** имени переменной вместо ее текущего значения отображается сообщение Unknown identifier (неизвестный идентификатор). Однако сразу после начала трассировки это сообщение будет заменено значением переменной.

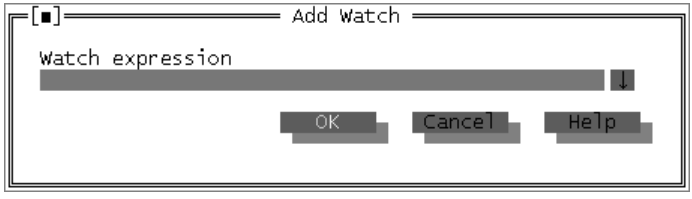
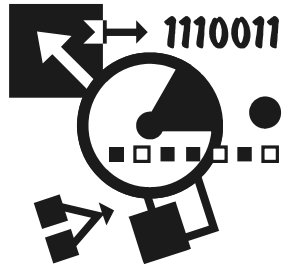


Рис. 13.5. Окно **Add Watch**



Глава 14

Введение в объектно-ориентированное программирование

Исторически сложилось так, что программирование возникло и развивалось как процедурное программирование. Процедурное программирование предполагает, что основой программы является алгоритм, *процедура* обработки данных.

Объектно-ориентированное программирование (ООП) — это методика разработки программ, в основе которой лежит понятие объекта как некоторой структуры, соответствующей объекту реального мира. Задача, решаемая с использованием методики ООП, описывается в терминах объектов и операций над ними. Программа при таком подходе представляет собой набор объектов и связей между ними.

Объектный тип и объект

Turbo Pascal, поддерживая концепцию объектно-ориентированного программирования, дает возможность определять *объекты*. Например:

```
type
    TPerson = object
     fName:string[20];
     fAddress:string[40];
    end;
var
   Person: TPerson;
```

В приведенном примере объявлены объектный тип `TPerson` и переменная `Person`. Переменная `Person` — это *объект*, экземпляр объектного типа. Объект `Person` представляет собой структуру, которая состоит из двух полей строкового типа: `fName` и `fAddress`.

Примечание

Согласно принятому в среде программистов соглашению, имена типов должны начинаться с буквы T (от Type — тип), а имена полей — с буквы f (от field — поле).

В реальном мире одно понятие может быть определено через другое. Например, можно дать следующие определения понятий "студент" и "профессор". Студент — это человек, который учится в институте или в университете. Профессор — это человек, который обучает в институте или в университете. В приведенных определениях понятия "студент" и "профессор" определены через понятие "человек". В программе моделью реального студента, объекта "студент", может быть набор переменных, характеризующих студента, с одной стороны, как человека (имя и адрес), с другой — как учащегося. Например, характеристикой студента как учащегося может быть номер учебной группы. Аналогичным образом может быть представлена модель "профессор". Ниже приведено объявление объектных типов `TStudent` и `TProfessor`, учитывающих, что Turbo Pascal дает возможность определить один объектный тип как расширение другого объектного типа.

```
type
    TPerson = object
     fName:string[20];     { имя   }
     fAddress:string[40]; { адрес }
    end;

    TStudent = object (TPerson)
     fGroup:integer; { номер учебной группы }
    end;

    TProfessor = object (TPerson)
     fKafedra:string[40];      { название кафедры }
    end;
```

В приведенном примере тип `TPerson` является родительским типом для типов `TStudent` и `TProfessor`. В свою очередь типы `TStudent` и `TProfessor` являются дочерними типами для типа `TPerson`, они наследуют поля своего родительского типа. При объявлении дочернего типа родительский тип указывается в скобках, после слова `object`.

В принципе, приведенные выше определения типов `TStudent` и `TProfessor` могут быть записаны так:

```
type
    TStudent = object
```

```
    fName:string[20];      { имя   }
    fAddress:string[40];   { адрес }
    fGroup:integer;        { номер учебной группы }
  end;

  TProfessor = object
    fName:string[20];      { имя   }
    fAddress:string[40];   { адрес }
    fKafedra:string[40];   { название кафедры }
  end;
```

Однако второй вариант объявления менее предпочтителен. Например, если в программе надо будет добавить информацию о дате рождения студентов и преподавателей, то в первом случае достаточно будет внести изменения только в объявление типа `TPerson`, например, так:

```
TPerson = object
      fName:string[20];     { имя   }
      fAddress:string[40]; { адрес }
      { число, месяц и год рождения }
      fDay:byte;
      fMonth:byte;
      fYear:integer;
  end;
```

Во втором случае придется вносить изменения в объявления обоих типов: `TProfessor` и `TStudent`.

Доступ к данным (полям) объекта может осуществляться точно так же, как и к полям записи, в том числе и с использованием **`with`**. В листинге 14.1 приведена простая программа, которая демонстрирует варианты доступа к полям объекта.

Листинг 14.1. Объявление и работа с объектами (p14_1.pas)

```
program p14_1;
type
  TPerson = object
    fName:string[20];
    fAddress:string[40];
  end;
  TStudent = object (TPerson)
    fGroup:integer;
```

```
   end;
var
   Student:TStudent;
begin
   writeln('Введите фамилию, адрес, номер группы и нажмите <Enter>');
   write('Фамилия ->');
   readln(Student.fName);
   write('Адрес ->');
   readln(Student.fAddress);
   write('Номер группы ->');
   readln(Student.fGroup);
   with Student do
      begin
        writeln('Фамилия:',fName);
        writeln('Адрес:  ',fAddress);
        writeln('Группа: ',fGroup);
      end;
end.
```

Следует обратить внимание на то, что поля `fName` и `fAddress`, объявленные в объектном типе `TPerson`, являются также полями объекта типа `TStudent`, так как тип `TStudent` порожден от типа `TPerson`.

Методы

Метод — это процедура или функция, которая выполняет некоторое действие над объектом. Объявления методов помещают в объявление объектного типа. Включая объявление методов в объявление объектного типа, программист явно указывает действия, которые могут быть выполнены над объектом.

Вот пример объявления объектного типа, в состав которого включены три метода:

```
type
   TPerson = object
      fName:string;
      fAddress:string;
      procedure Init(aName,aAddress:string);
      procedure Show;
      function GetAddress:string;
end;
```

Метод-процедура `Init` обеспечивает инициализацию объекта, метод-процедура `Show` — отображение характеристик (содержимого полей) объекта, метод-функция `GetAddress` — доступ к полю `fAddress`.

В программе методы определяются точно так же, как обычные процедуры и функции, за исключением того, что имя процедуры или функции, являющейся методом, должно состоять из двух частей: имени объектного типа, которому принадлежит метод, и имени метода. Имя метода отделяется от имени объектного типа точкой. Ниже приведены определения методов объектного типа `TPerson`.

```
procedure TPerson.Init(aName,aAddress:string);
begin
     fName := aName;
     fAddress := aAddress;
end;
procedure TPerson.Show;
begin
     writeln('Имя:', fName);
     writeln('Адрес:', fAddress);
end;
function TPerson.GetAddress:string;
begin
     GetAddress := fAddress;
end;
```

Чтобы выполнить действие над объектом, следует указать объект и метод, который обеспечивает выполнение этого действия. Инструкция применения метода к объекту в общем виде выглядит так:

Объект.*Метод*;

Например, в результате выполнения инструкции

```
Student.Show;
```

к объекту `Student` будет применен метод `Show`, в результате чего на экран будет выведено содержимое полей объекта `Student`.

Ограничение доступа к полям объекта

Доступ к полям объекта с использованием имени поля возможен, однако в соответствии с принципами ООП вместо прямого доступа к полям объекта следует использовать методы.

Ниже приведено объявление типа TPerson. Метод SetAddress позволяет изменить значение поля fAddress, методы GetName и GetAddress — получить значения соответствующих полей.

```
type
   TPerson = object
      fName:string;
      fAddress:string;
      procedure Init(n,a: string);
      function GetName: string;
      function GetAddress: string;
      procedure SetAddress(NewAddress:string);
end;
```

Для защиты полей объекта от прямого доступа используют директиву **private**.

> ***Примечание***
>
> Слову Private в русском языке соответствует несколько значений: частный, личный, тайный, уединенный. Применительно к рассматриваемому вопросу наиболее точно смысл директивы private передает слово "частный", в смысле "закрытый от других".

Директиву **private** помещают перед объявлением полей, доступ к которым надо запретить. Объявления частных полей метода помещают после объявления обычных (доступных) полей и методов типа. Например:

```
type
   TPerson = object
      procedure Init(n,a:string);
      function GetName:string;
      function GetAddress:string;
      procedure SetAddress(NewAddress:string);
    private
      fName: string;
      fAddress: string;
end;
```

Директива **private** также используется для ограничения использования некоторых методов объекта. Поля и методы, объявленные как **private**, доступны только в том модуле, в котором объявлен тип, которому они принадлежат. В модуле, который использует этот тип, поля и методы, объявленные как **private**, недоступны. Таким образом, чтобы запретить доступ к полям объекта и использование некоторых его методов, нужно поместить объявление объектного типа в отдельный модуль.

Наследование

Объект-потомок наследует не только поля объекта-родителя, но и методы.

Пусть имеет место следующее определение:

```
type
    TPerson = object
       fName: string;      { имя }
       fAddress: string;   { адрес }
       procedure Show;     { выводит имя и адрес }
    end;

    TStudent = object (TPerson)
       fGroup: integer;    { номер учебной группы }
    end;

procedure TPerson.Show;
    begin
       writeln('Имя:', fName);
       writeln('Адрес:', fAddress);
    end;

var
    Student: TStudent;
```

Хотя для типа `TStudent` метод `Show` не определен явно, инструкция

```
Student.Show;
```

является верной, так как тип `TStudent` порожден от типа `TPerson` и, следовательно, наследует метод `Show` своего родительского типа.

В результате выполнения инструкции `Student.Show` к объекту `Student` будет применен метод `TPerson.Show` (метод его родительского типа). При этом на экран будут выведены имя и адрес студента. Для того чтобы на экран была выведена вся информация о студенте, следует в объявлении типа `TStudent` объявить метод, который будет это делать. В принципе, этому методу можно присвоить любое имя. Однако Turbo Pascal позволяет объявить в дочернем типе метод, имя которого будет совпадать с именем одного из методов родительского типа. Использование имени метода родительского типа в объявлении дочернего типа называется *переопределением метода*.

Чтобы переопределить метод родительского типа, надо в объявлении дочернего типа объявить метод, имя которого будет совпадать с именем переопределяемого метода родительского типа. Например:

```
TStudent = object (TPerson)
       fGroup: integer;   { номер учебной группы }
       procedure Show;
    end;
procedure TStudent.Show;
   begin
      writeln('Имя:', fName);
      writeln('Адрес:', fAddress);
      writeln('Группа:', fGroup);
   end;
```

Если переопределяемый метод, метод родительского типа, имеет параметры, замещающий метод не обязан иметь такое же количество и такие же типы параметров, что и у соответствующего родительского метода.

Замещая метод родительского типа, программист должен иметь в виду, что замещаемый метод может выполнять какую-то важную работу, которую после замещения метода родительского класса должен будет выполнить соответствующий замещающий метод. Например, ниже приведено описание типа `TPerson`, в состав которого включен метод `Init`, обеспечивающий инициализацию полей объекта. В объектный тип `TStudent`, порожденный от `TPerson`, очевидно, также должен быть включен метод, который обеспечивает инициализацию полей объекта. Объявление типов `TPerson`, `TStudent` и методов `Init` может быть таким:

```
type
   TPerson = object
      fName: string;
      fAddress: string;
      procedure Init(aName, aAddress:string);
   end;

   TStudent = object(TPerson)
      fGroup: integer;
      procedure Init(aName, aAddress:string; aGroup:integer);
   end;

   procedure TStudent.Init(aName, aAddress:string; aGroup:integer);
```

```
  begin
   fName := aName;
   fAaddress := aAddress;
   fGroup := aGroup;
  end;
```

В приведенном примере метод `TStudent.Init` напрямую использует поля своего родительского типа. Однако такое решение не всегда возможно и эффективно. Например, если родительский тип объявлен в другом модуле, и его поля находятся в защищенной секции (**`private`**), то прямой доступ к полям родительского типа закрыт. Кроме того, иерархия объектов программы может иметь и, как правило, имеет несколько уровней, т. е. у дочернего объекта несколько родителей (хотя надо еще раз отметить, что у любого объектного типа только один непосредственный родитель). В этом случае программисту приходится заботиться об инициализации и правильном использовании всех полей родительских типов. Очевидно, что прямое использование полей родительских типов является не лучшим решением. Возникает желание использовать метод инициализации родительского типа. Это можно сделать непосредственным вызовом метода родительского типа. Инструкция вызова перегруженного метода родительского типа из метода дочернего типа в общем виде выглядит так:

Тип`.`*Метод*`;`

где *Тип* — имя объектного типа, для которого вызывается переопределенный в дочернем типе метод.

Ниже приведен вариант реализации метода `TStudent.Init`, соответствующий концепции ООП.

```
procedure TStudent.Init(aName,aAddress: string; aGroup: integer);
  begin
   TPerson.Init(aName, aAddress); { вызов метода родительского типа }
   fGroup := aGroup;
  end;
```

Следует обратить внимание, что в инструкции вызова метода родительского типа, переопределенного в дочернем, нельзя указать только имя вызываемого метода, так как имя переопределенного метода присутствует в объявлении дочернего типа, и если указать только имя метода, то будет вызван метод дочернего, а не родительского типа. Поэтому в инструкции вызова метода перед именем метода помещают квалификатор, уточняющий, метод какого объектного типа вызывается.

Динамические объекты

Для хранения объектов можно использовать динамическую память. В этом случае в программе объявляется не переменная объектного типа, а указатель на объектный тип. Например:

```
type
   TPerson = object
     fName: string;
     fAddress: string;
     procedure Init(aName, aAddress:string);
     procedure Show;
   end;
var
   Person: ^TPerson;
```

Память для объекта выделяется вызовом процедуры `New`, которой в качестве параметра передается имя переменной-указателя на объект.

После выделения памяти доступ к объекту осуществляется обычным образом.

В качестве примера в листинге 14.2 приведена программа, которая демонстрирует работу с динамическим объектом.

Листинг 14.2. Динамический объект (p14_2.pas)

```
program p14_2;

type
    TPerson = object
       fName: string;      { имя }
       fAddress: string;   { адрес }
       procedure Init(aName,aAddress: string);
       procedure Show;
    end;

procedure TPerson.Init(aName,aAddress: string);
begin
     fName := aName;
     fAddress := aAddress;
```

```
end;

procedure TPerson.Show;
begin
     writeln('Имя: ', fName);
     writeln('Адрес: ', fAddress);
end;

var
   person: ^TPerson;  { указатель на объект }

begin
   New(person); { выделить память для объекта}

   person^.Init('Вася Васильчиков','ул. Зверинская');
   person^.Show;

   write('Нажмите <Enter>');
   readln;

   Dispose(person); { освободить память, занимаемую объектом}
end.
```

Полиморфизм и виртуальные методы

Если несколько объектных типов связаны между собой отношениями родитель-потомок, то указателю на базовый тип может быть присвоено значение указателя на любой из его дочерних типов.

Например, пусть определены типы `TStud` и `TProf`, а также тип `TPerson`, который является для типов `TStud` и `TProf` базовым:

```
type
   TPerson = object
      fName: string;
      fAddress: string;
      procedure print;
   end;
   TStud = object(TPerson)
```

```
    fGroup:integer;
    procedure print;      {  выводит информацию о студенте }
  end;
  TProf = object(TPerson)
    fKafedra: string;
    procedure print;      { выводит информацию о преподавателе }
  end;
```

Если в программе определены указатели на каждый из этих типов:

```
p: ^TPerson;
ps: ^TStud;
pp: ^TProf;
```

то указателю на базовый тип можно присвоить значение указателя на дочерний:

```
p := ps;
p := pp;
```

Описанное свойство указателей на объектные типы позволяет организовать список студентов и преподавателей как массив указателей на объектный тип `TPerson`, например, так:

```
spisok: array[1..N] of ^TPerson;
```

Тогда инструкция

```
for i:=1 to N do
  spisok[i]^.print;
```

выведет (если метод `print` объявлен как виртуальный, см. ниже) на экран список преподавателей и студентов. Однако здесь следует обратить внимание на то, что во время разработки программы нельзя знать, на объект какого типа будет указывать конкретный элемент массива `spisok` во время работы программы. Элемент массива может быть указателем на `TStud` или на `TProf`. Поэтому нельзя заранее указать, какой из методов `print` будет вызван — `TStud.print` или `TProf.print`. Решение о выборе метода принимается во время выполнения программы.

Изложенный выше пример вывода списка студентов и преподавателей демонстрирует концепцию *полиморфизма*, которая состоит в том, что при применении метода к объекту используется именно тот метод, который соответствует типу объекта.

Концепция полиморфизма реализуется при помощи *виртуальных* методов.

> ***Примечание***
>
> Термин "виртуальный" широко используется в вычислительной технике и, в последнее время, в повседневной жизни. Например, можно слышать "виртуаль-

ный мир", "виртуальная реальность". Но что означает само понятие "виртуальный"? Ниже приведено объяснение, взятое из логического словаря-справочника (Кондаков Н. И., Логический словарь-справочник, М.: Наука, 1975, 720 с.).

ВИРТУАЛЬНЫЙ (лат. virtualis — возможный) — такой возможный объект, который нами еще не воспринимается как что-то вполне определенное, но способный при наличии известных условий возникнуть, проявиться; иногда виртуальным называют и такой объект, который просто "способен к действию".

Виртуальный метод объявляется в базовом объектном типе и в порожденных от базового типах. Метод считается виртуальным, если после его объявления указано слово **virtual**.

В листинге 14.3 приведена программа, которая иллюстрирует понятие "полиморфизм". Программа сначала формирует список, состоящий из объектов типа `TStud` и `TProf`, затем, применяя метод `print` к элементам массива, выводит этот список на экран.

Листинг 14.3. Демонстрация полиморфизма и виртуальных методов (p14_3.pas)

```
program p14_3;

const
   HB = 10;  { длина списка }

type
   TPerson = object
      FName: string;      { имя    }
      FAddress: string;   { адрес }
      constructor init(aName,aAddress:string); { конструктор объекта }
      destructor  Done; virtual;               { деструктор объекта  }
      procedure print; virtual;
   end;

   TStud = object(TPerson)
      fGroup: integer;        { номер группа }
      constructor init(aName,aAddress:string;aGroup:integer);
      destructor  Done; virtual;
      procedure print; virtual;
   end;

   TProf = object(TPerson)
```

```
      FKafedra: string[30];  { кафедра }
      constructor init(aName,aAddress,aKafedra:string);
      destructor  Done; virtual;
      procedure print; virtual;
   end;

   { Указатели на типы TStud и TProf }
   TpStud = ^TStud;
   TpProf = ^TProf;

constructor TPerson.init(aName,aAddress:string);
   begin
     fName := aName;
     fAddress := aAddress;
   end;

destructor TPerson.Done;
begin
end;

procedure TPerson.print;
   begin
      writeln(fName);
      writeln(fAddress);
   end;

constructor TStud.init(aName,aAddress:string;aGroup:integer);
   begin
     TPerson.init(aName,aAddress);
     fGroup:=aGroup;
   end;

destructor TStud.Done;
begin
   inherited Done; { вызов деструктора родительского типа }
end;

procedure TStud.print;
```

```
   begin
      TPerson.print;
      writeln('гр. ',fGroup);
   end;

constructor TProf.init(aName,aAddress,aKafedra:string);
   begin
     TPerson.init(aName,aAddress);
     fKafedra := aKafedra;
   end;

destructor TProf.Done;
begin
   inherited Done;
end;

procedure TProf.print;
   begin
      TPerson.print;
      writeln('каф. ',fKafedra);
   end;

{ программа }
var
   list: array[1..HB] of ^TPerson; { список студентов и преподавателей }
   i:integer;

begin
   { инициализация списка }
   for i:=1 to HB do
       list[i] := NIL;

   { создать три объекта и поместить в список }
   list[1] := new(TpStud,init('Иванов','Лесной, д.29, к.203',1813));
   list[2] := new(TpStud,init('Петров','Морская, д.5, кв.9',1813));
   list[3] := new(TpProf,init('Некрасов','Большой, д.12, кв.3','АиВТ'));

   { вывести список }
```

```
   for i:=1 to HB do
      if list[i] <> NIL
         then list[i]^.print;

   { уничтожить объекты }
   for i:=1 to HB do
      if list[i] <> NIL then
            begin
              Dispose(list[i],Done);
              list[i]:=NIL;
            end;

   write('Для завершения нажмите <Entr>');
   Readln;

end.
```

В приведенной программе в объявлении объектных типов следует обратить внимание на слово **constructor**, заменившее **procedure**. Конструктор — это особый метод, который выполняет некоторую работу, обеспечивающую поддержку механизма виртуальных методов.

> ***Примечание***
>
> Конструктор должен вызываться раньше, чем любой другой метод объекта, поэтому на конструктор, как правило, возлагают задачу инициализации объекта.

Для создания объектов используется расширенный синтаксис вызова функции `new`, который позволяет выделить память для объекта и инициализировать объект с помощью вызова конструктора. В общем виде инструкция обращения к функции `new` с использованием расширенного синтаксиса выглядит так:

```
Указатель:=new(ТипОбъекта, КонструкторОбъекта);
```

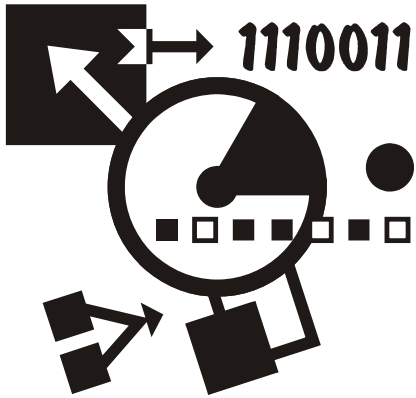
В приведенной программе объекты являются динамическими. Если в программе объект больше не нужен, то лучше освободить память, которую он использует. В простейшем случае это можно сделать при помощи процедуры `Dispose`. Например, если `p` — указатель на размещенный в динамической памяти объект типа `TPerson`, то инструкция `Dispose(p)` освобождает занимаемую этим объектом память. В результате освобождения памяти, занимаемой объектом, объект перестает существовать, поэтому действия по освобождению памяти, занимаемой объектом, часто называют уничтожением объекта.

Объект, как правило, представляет собой сложную динамическую структуру, в состав которой могут входить указатели на другие динамические структуры или объекты, занимаемая которыми память также должна освобождаться при уничтожении объекта. Поэтому для освобождения памяти, занимаемой динамическим объектом, рекомендуется использовать метод `Done` (`Done` — это принятое в Turbo Pascal имя метода, обеспечивающего освобождение памяти, занимаемой динамическим объектом). Метод `Done` оформляется как деструктор (**`destructor`**) — особая процедура, которая выполняет некоторую специфическую работу, в том числе обеспечивает объединение освобождающихся, возможно, небольших участков памяти в более крупные блоки.

В программе для освобождения памяти, занимаемой объектом, деструктор следует вызывать с использованием расширенного синтаксиса процедуры `Dispose`. Например, если `p` — указатель на динамический объект типа `TPerson`, а `Done` — деструктор этого объектного типа, то инструкция `Dispose(p,Done);` уничтожает объект, на который указывает `p`.

Модели объектов других языков программирования

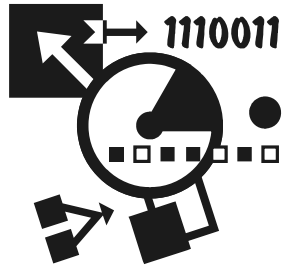
В описании многих языков программирования (C++, Object Pascal, Visual Basic) указывается, что язык поддерживает концепцию ООП. Однако здесь следует понимать, что модели объектов, принятые в разных языках программирования, могут существенно различаться. Например, объектная модель языка программирования Object Pascal (используется в среде разработки программ Delphi), которая является преемницей объектной модели Turbo Pascal, предполагает использование только динамических объектов. В эту модель также введено понятие "свойство" как метода особого вида, обеспечивающего доступ к полям объекта.



Часть II

DELPHI

Вторая часть книги посвящена основам программирования в Delphi. В ней приведено краткое описание среды Delphi, на примере программы вычисления дохода по вкладу демонстрируется технология визуального проектирования и событийного программирования.



Глава 15

Среда программирования Delphi

Delphi — что это?

Среди пользователей персональных компьютеров в настоящее время наиболее популярно семейство операционных систем Windows, и естественно, что тот, кто собирается программировать, стремится писать программы, которые будут работать в этих системах.

Несколько лет назад рядовому программисту оставалось только мечтать о создании собственных программ, работающих в среде Windows, т. к. единственным средством разработки был Borland C++ for Windows, явно ориентированный на профессионалов, обладающих серьезными знаниями и опытом.

Бурное развитие вычислительной техники, потребность в эффективных средствах разработки программного обеспечения привели к появлению систем программирования, ориентированных на так называемую "быструю разработку", среди которых можно выделить Borland Delphi и Microsoft Visual Basic. В основе систем быстрой разработки (RAD-систем, Rapid Application Development — среда быстрой разработки приложений) лежит технология визуального проектирования и событийного программирования, суть которой заключается в том, что среда разработки берет на себя большую часть рутинной работы, оставляя программисту работу по конструированию диалоговых окон и созданию процедур обработки событий. Производительность программиста при использовании RAD-систем — фантастическая!

Delphi — это среда разработки компьютерных программ, в которой в качестве языка программирования используется язык Delphi. В основе идеологии Delphi лежит технология визуального проектирования и событийного программирования. Следует обратить внимание на то, что хотя среда Delphi является объектно-ориентированной, тем не менее при решении прикладных задач она не требует от программиста фундаментальных знаний в области

объектно-ориентированного программирования (достаточно понимания концепции), что позволяет использовать ее "обычным" программистам.

Среди программистов наиболее популярна седьмая версия пакета Delphi — Borland Delphi 7 Studio. Она позволяет создавать самые различные программы: от простейших однооконных приложений до программ работы с распределенными базами. Необходимо отметить, что Delphi 7 — это не последняя версия пакета. В настоящее время доступна широкая линейка продуктов Delphi (Borland практически каждый год выпускает новую версию пакета). Принципиальных отличий между Delphi 7 и последующими версиями пакета с точки зрения *технологии* создания программ нет; изучив Delphi 7, вы сможете в кратчайший срок "перейти" на другую версию.

Delphi 7 может работать в среде операционных систем от Microsoft Windows 2000 до Windows Vista. Особых требований, по современным меркам, к ресурсам компьютера пакет не предъявляет: процессор должен быть типа Pentium, Celeron или AMD с тактовой частотой не ниже 166 МГц (рекомендуется 400 МГц), оперативной памяти — 128 Мбайт (рекомендуется 256 Мбайт), достаточное количество свободного места на жестком диске (для полной установки версии Enterprise необходимо приблизительно 475 Мбайт).

Начало работы

Запускается Delphi обычным образом, т. е. выбором в меню **Borland Delphi 7** команды **Delphi 7** (рис. 15.1).

Запуск Delphi активизирует процесс создания нового *приложения* (приложение — это *прикладная* программа, т. е. программа, обеспечивающая решение некоторой *прикладной* задачи).

Вид экрана после запуска Delphi несколько необычен, на экране отображается не одно окно, а пять (рис. 15.2):

- главное окно;
- окно стартовой формы — **Form 1**;
- окно редактора свойств объектов — **Object Inspector**;
- окно списка объектов — **Object TreeView**;
- окно редактора кода — **Unit1.pas**.

Следует обратить внимание, что окно редактора находится за окном кода и почти полностью закрыто окном стартовой формы.

В главном окне (рис. 15.3) отображаются: меню команд, панели инструментов и палитра компонентов.

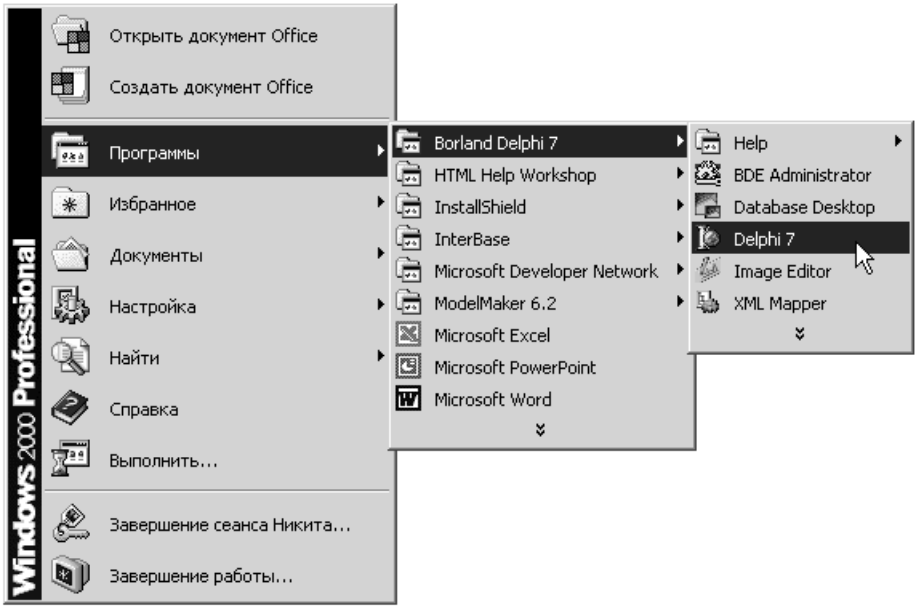


Рис. 15.1. Запуск Delphi

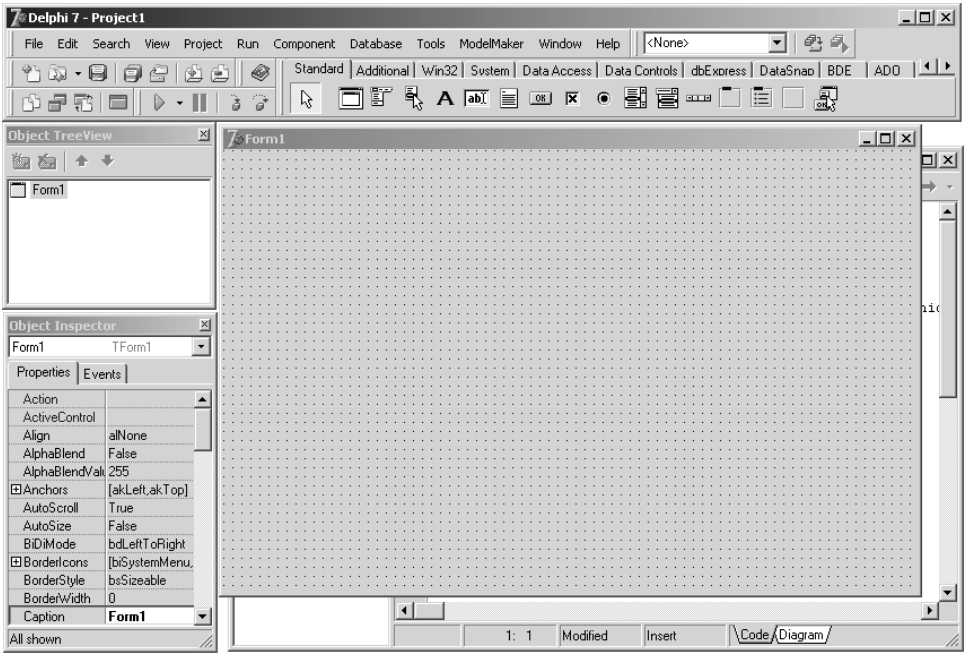


Рис. 15.2. Вид экрана после запуска Delphi

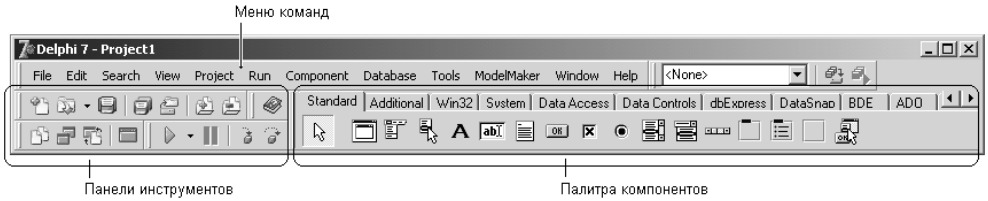


Рис. 15.3. Главное окно

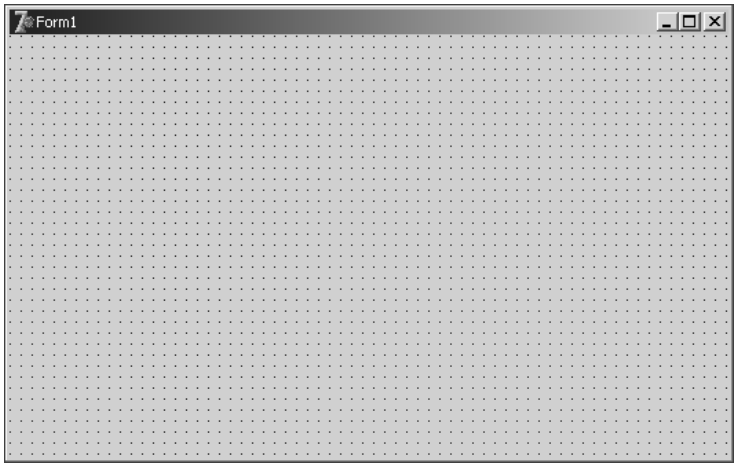


Рис. 15.4. Стартовая форма

Окно стартовой формы (рис. 15.4) представляет собой заготовку главного окна разрабатываемого *приложения*.

> ***Примечание***
>
> Программное обеспечение принято делить на системное и прикладное. Системное программное обеспечение — это все то, что составляет операционную систему. Остальные программы принято считать прикладными. Для краткости прикладные программы называют приложениями.

Окно **Object Inspector** (рис. 15.5) — окно редактора свойств объектов предназначено для редактирования значений свойств объектов. В терминологии визуального проектирования *объекты* — это диалоговые окна и элементы управления (поля ввода и вывода, командные кнопки, переключатели и др.). *Свойства объекта* — это характеристики, определяющие вид, положение и поведение объекта. Например, свойства `Width` и `Height` задают размер (ширину и высоту) формы, свойства `Top` и `Left` — положение формы на экране, свойство `Caption` — текст заголовка.

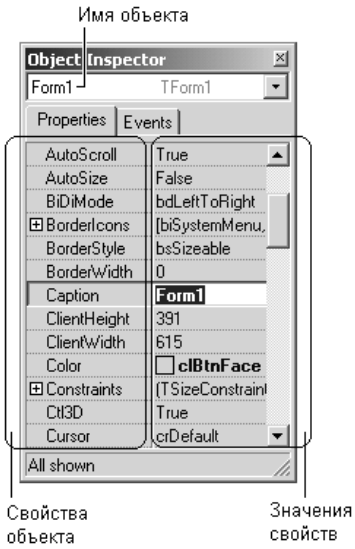


Рис. 15.5. На вкладке **Properties** окна **Object Inspector** перечислены свойства объекта и указаны их значения

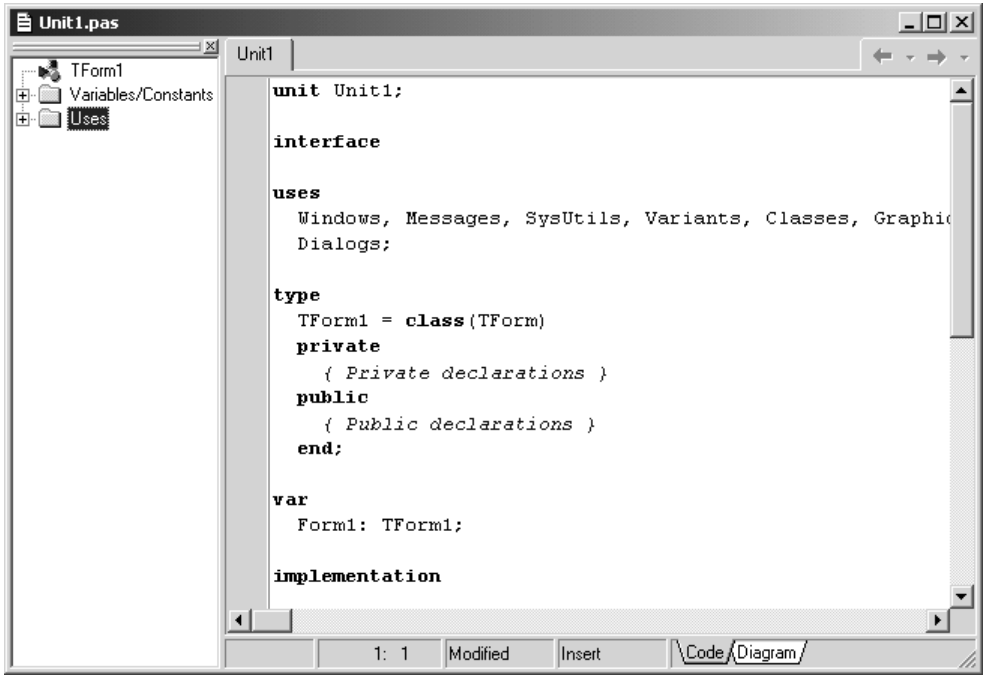


Рис. 15.6. Окно редактора кода

В окне редактора кода (рис. 15.6), которое можно увидеть, отодвинув в сторону окно формы, следует набирать текст программы. В начале работы над новым проектом это окно редактора кода содержит сформированный Delphi шаблон программы.

Первый проект

Для демонстрации процесса программирования в Delphi, технологии визуального проектирования и событийного программирования, разработаем приложение, используя которое можно рассчитать доход по вкладу. Вид окна программы во время ее работы приведен на рис. 15.7.

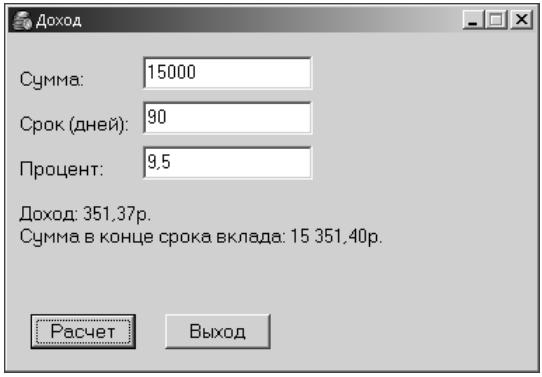


Рис. 15.7. Окно программы **Доход**

Для начала работы над новой программой запустите Delphi. Если вы уже работаете в среде разработки и у вас загружен другой проект, в меню **File** выберите команду **New ▸ Application**.

Форма

Работа над *новым проектом*, так в Delphi называется разрабатываемое приложение, начинается с создания стартовой формы. Так на этапе разработки программы называют диалоговые окна.

Стартовая форма создается путем изменения значений свойств формы **Form1** и добавления к форме необходимых компонентов (полей ввода данных, отображения текста, командных кнопок).

Свойства формы (табл. 15.1) определяют ее размер, вид границы, положение на экране, текст заголовка. Чтобы изменить вид формы, надо изменить значение соответствующего свойства. Например, чтобы изменить текст в заголовке, надо изменить значение свойства `Caption`.

Таблица 15.1. *Свойства формы (объекта TForm)*

Свойство	Описание
Name	Имя формы. В программе имя формы используется для управления формой и доступа к компонентам формы
Caption	Текст заголовка
Width	Ширина формы
Height	Высота формы
Top	Расстояние от верхней границы формы до верхней границы экрана
Left	Расстояние от левой границы формы до левой границы экрана
BorderStyle	Вид границы. Граница может быть обычной (bsSizeable), тонкой (bsSingle) или отсутствовать (bsNone). Если у окна обычная граница, то во время работы программы пользователь может при помощи мыши изменить размер окна. Изменить размер окна с тонкой границей нельзя. Если граница отсутствует, то на экран во время работы программы будет выведено окно без заголовка. Положение и размер такого окна во время работы программы изменить нельзя
BorderIcons	Кнопки управления окном. Значение свойства определяет, какие кнопки управления окном будут доступны пользователю во время работы программы. Значение свойства задается путем присвоения значений уточняющим свойствам biSystemMenu, biMinimaze, biMaximaze и biHelp. Свойство biSystemMenu определяет доступность кнопки **Свернуть** и кнопки системного меню, biMinimaze — кнопки **Свернуть**, biMaximaze — кнопки **Развернуть**, biHelp — кнопки вывода справочной информации
Icon	Значок в заголовке диалогового окна, обозначающий кнопку вывода системного меню
Color	Цвет фона. Цвет можно задать, указав название цвета или привязку к текущей цветовой схеме операционной системы. Во втором случае цвет определяется текущей цветовой схемой, выбранным компонентом привязки и меняется при изменении цветовой схемы операционной системы
Font	Шрифт. Шрифт, используемый "по умолчанию" компонентами, находящимися на поверхности формы. Изменение свойства Font формы приводит к автоматическому изменению свойства Font компонента, располагающегося на поверхности формы. То есть компоненты наследуют свойство Font от формы (имеется возможность запретить наследование)

Для изменения значений свойств формы и ее компонентов используется окно **Object Inspector**. В верхней части окна **Object Inspector** указано имя объекта, значения свойств которого отображается в данный момент. В левой колонке вкладки **Properties** (Свойства) перечислены свойства объекта, а в правой — указаны их значения.

При создании формы в первую очередь следует изменить значение свойства `Caption` (Заголовок). В нашем примере надо заменить текст `Form1` на "Доход". Чтобы это сделать, нужно в окне **Object Inspector** щелкнуть мышью в строке `Caption`, в результате чего будет выделено текущее значение свойства, в поле редактирования появится курсор, и можно будет ввести текст "Доход" (рис. 15.8).

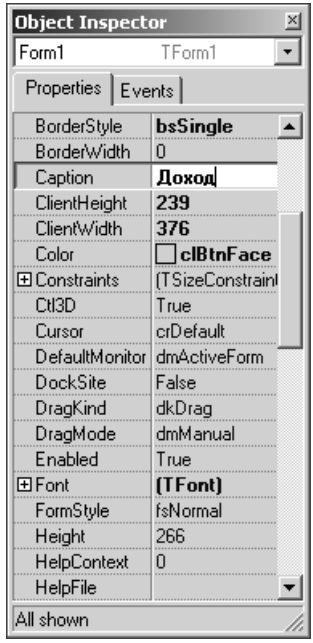


Рис. 15.8. Установка значения свойства путем ввода значения

Аналогичным образом можно установить значения свойств `Height` и `Width`, которые определяют высоту и ширину формы. Размер формы и ее положение на экране, а также размер других элементов управления и их положение на поверхности формы задают в пикселах, т. е. точках экрана. Свойствам `Height` и `Width` надо присвоить значения 266 и 384, соответственно.

Форма — это обычное окно. Поэтому его размер можно изменить точно так же, как любого другого окна, т. е. захватом и перемещением (с помощью

мыши) границы. По окончании перемещения границ автоматически изменятся значения свойств `Height` и `Width`. Они будут соответствовать установленному размеру формы.

Положение окна программы на экране после запуска программы соответствует положению формы во время ее разработки, которое определяется значением свойств `Top` (отступ от верхней границы экрана) и `Left` (отступ от левой границы экрана). Значения этих свойств также можно задать путем перемещения окна формы при помощи мыши.

При выборе некоторых свойств, например, `BorderStyle`, справа от текущего значения свойства появляется значок раскрывающегося списка. Очевидно, что значение таких свойств можно задать путем выбора из списка (рис. 15.9).

Некоторые свойства являются сложными, т. е. их значение определяется совокупностью значений других (уточняющих) свойств. Перед именами сложных свойств стоит значок "+", при щелчке на котором раскрывается список уточняющих свойств (рис. 15.10). Например, свойство `BorderIcons` определяет, какие кнопки управления окном будут доступны во время работы программы. Так, если свойству `biMaximize` присвоить значение `false`, то во время работы программы кнопки **Развернуть** в заголовке окна не будет.

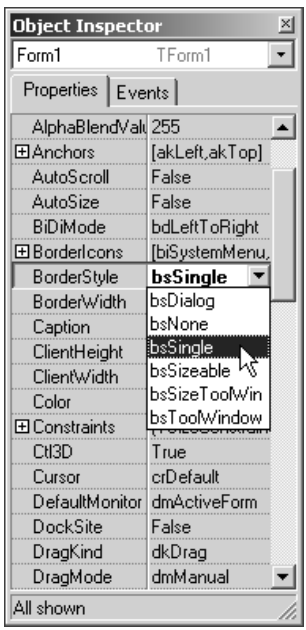


Рис. 15.9. Установка значения свойства путем выбора из списка

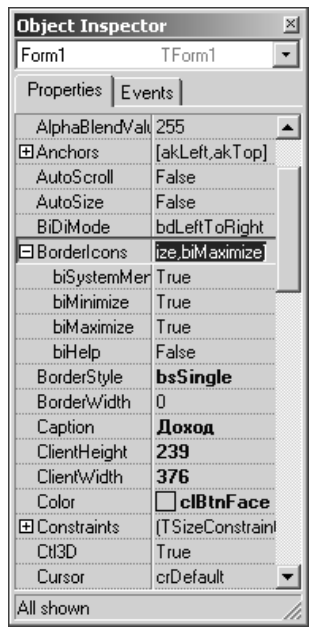


Рис. 15.10. Раскрытый список вложенных свойств сложного свойства `BorderIcons`

Рядом со значениями некоторых свойств отображается командная кнопка с тремя точками. Это значит, что для задания значения свойства можно воспользоваться дополнительным диалоговым окном. Например, значение сложного свойства `Font` можно задать путем непосредственного ввода значений уточняющих свойств, а можно воспользоваться стандартным диалоговым окном выбора шрифта.

В табл. 15.2 перечислены свойства формы разрабатываемой программы, которые следует изменить. Остальные свойства оставлены без изменения и в таблице не приведены.

Таблица 15.2. *Значения свойств стартовой формы*

Свойство	Значение
`Caption`	Доход
`Height`	266
`Width`	384
`BorderStyle`	bsSingle
`BorderIcons.biMinimize`	False
`BorderIcons.biMaximize`	False
`Font.Size`	10

В приведенной таблице в именах некоторых свойств есть точка. Это значит, что надо задать значение уточняющего свойства. После того как будут установлены значения свойств главной формы, она должна иметь вид, приведенный на рис. 15.11.

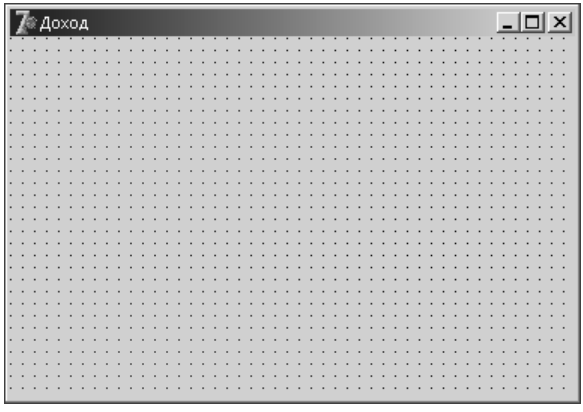


Рис. 15.11. Так выглядит форма после установки значений ее свойств

Компоненты

Программа вычисления дохода должна получить от пользователя исходные данные — сумму, срок вклада и значение процентной ставки. Для ввода данных с клавиатуры следует использовать поле редактирования — компонент `Edit`.

Компонент `Edit` (и другие наиболее часто используемые компоненты) находятся на вкладке **Standard** (рис. 15.12).

Для того чтобы добавить на форму компонент, надо раскрыть вкладку палитры компонентов, на которой компонент находится, сделать щелчок левой кнопкой мыши на значке компонента, установить курсор в ту точку формы, в которой должен быть левый верхний угол компонента, и еще раз щелкнуть левой кнопкой мыши.

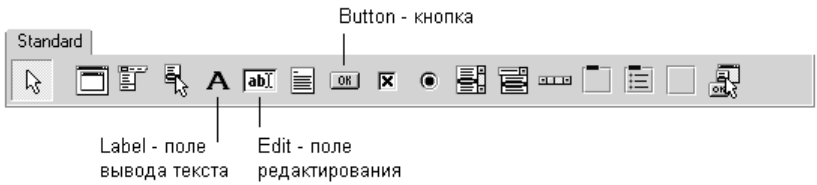


Рис. 15.12. Вкладка **Standard** содержит наиболее часто используемые компоненты

Размер компонента можно задать в процессе его добавления к форме. Для этого надо после выбора компонента из палитры поместить курсор мыши в ту точку формы, где должен находиться левый верхний угол компонента, нажать левую кнопку мыши и, удерживая ее нажатой, переместить курсор в точку, где должен находиться правый нижний угол компонента, затем отпустить кнопку мыши. На форме появится компонент нужного размера.

Каждому добавленному на форму компоненту Delphi автоматически присваивает имя, которое формируется из названия компонента и порядкового номера. Например, если к форме добавить два компонента `Edit`, то их имена будут `Edit1` и `Edit2`. Программист, путем изменения значения свойства `Name`, может изменить имя компонента. В простых программах имена компонентов, как правило, не изменяют.

На рис. 15.13 приведен вид формы после добавления трех компонентов `Edit` (полей редактирования), предназначенных для ввода исходных данных. Один из компонентов выделен. Свойства выделенного компонента отображаются в окне **Object Inspector**. Чтобы увидеть свойства другого компонента, надо

щелкнуть левой кнопкой мыши на изображении нужного компонента. Можно также выбрать имя компонента в окне **Object TreeView** или из находящегося в верхней части окна **Object Inspector** раскрывающегося списка объектов.

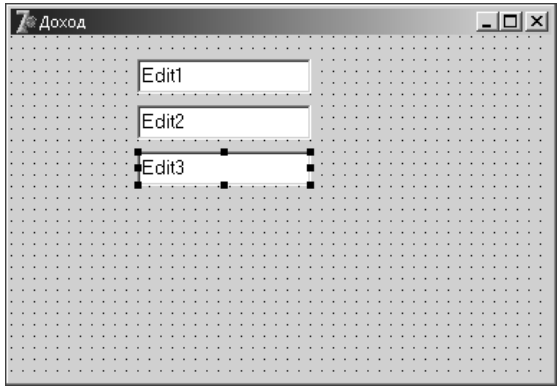


Рис. 15.13. Форма после добавления компонентов `Edit`

В табл. 15.3 перечислены основные свойства компонента `Edit` — поля ввода-редактирования.

Таблица 15.3. *Свойства компонента* `Edit` *(поле ввода-редактирования)*

Свойство	Описание
`Name`	Имя компонента. Используется в программе для доступа к компоненту и его свойствам, в частности — для доступа к тексту, введенному в поле редактирования
`Text`	Текст, находящийся в поле ввода и редактирования
`Left`	Расстояние от левой границы компонента до левой границы формы
`Top`	Расстояние от верхней границы компонента до верхней границы формы
`Height`	Высота поля
`Width`	Ширина поля
`Font`	Шрифт, используемый для отображения вводимого текста
`ParentFont`	Признак наследования компонентом характеристик шрифта формы, на которой находится компонент. Если значение свойства равно `True`, то при изменении свойства `Font` формы автоматически меняется значение свойства `Font` компонента

Изменить размер и положение компонента можно и при помощи мыши.

Для того чтобы изменить положение компонента, необходимо установить курсор мыши на изображение компонента, нажать левую кнопку мыши и, удерживая кнопку нажатой, переместить контур компонента в нужную точку формы, затем отпустить кнопку мыши. Во время перемещения компонента (рис. 15.14) отображаются текущие значения координат левого верхнего угла компонента (значения свойств `Left` и `Top`).

Для того чтобы изменить размер компонента, необходимо его выделить, установить указатель мыши на один из маркеров, помечающих границу компонента, нажать левую кнопку мыши и, удерживая ее нажатой, изменить положение границы компонента. Затем отпустить кнопку мыши. Во время изменения размера компонента отображаются текущие значения свойств `Height` и `Width` (рис. 15.15).

Свойства компонента так же, как и свойства формы, можно изменить при помощи **Object Inspector**. Для того чтобы свойства требуемого компонента были выведены в окне **Object Inspector**, нужно выделить этот компонент (щелкнуть мышью на его изображении). Можно также выбрать компонент из находящегося в верхней части окна **Object Inspector** раскрывающегося списка объектов (рис. 15.16) или из списка в окне **Object TreeView** (рис. 15.17).

В табл. 15.4 приведены значения свойств компонентов `Edit1` – `Edit3`. Компонент `Edit1` предназначен для ввода суммы вклада, `Edit2` — срока вклада, `Edit3` — процентной ставки. Обратите внимание на то, что значением свойства `Text` всех компонентов является пустая строка.

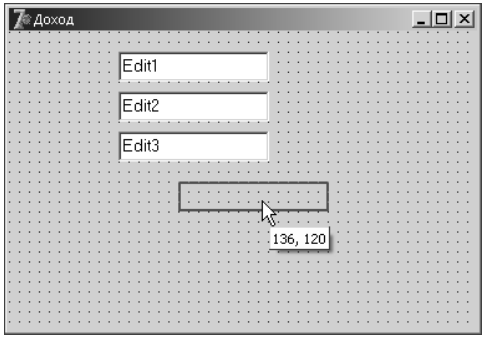


Рис. 15.14. Отображение текущих значений свойств `Left` и `Top` при изменении положения компонента

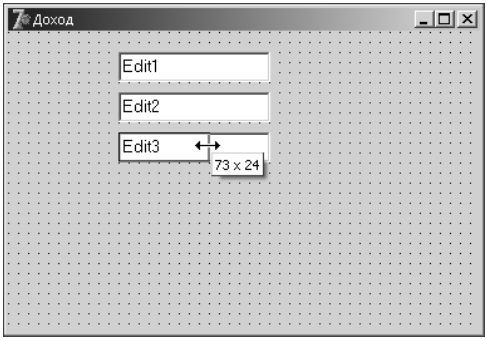


Рис. 15.15. Отображение текущих значений свойств `Height` и `Width` при изменении размера компонента

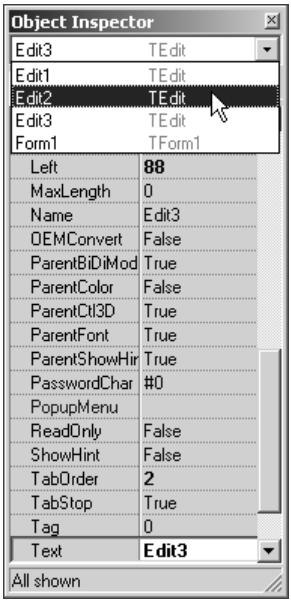


Рис. 15.16. Выбор компонента из списка в окне **Object Inspector**

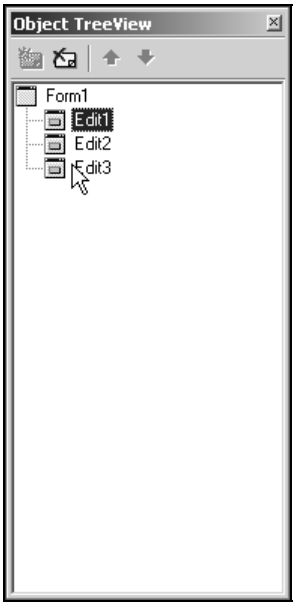


Рис. 15.17. Выбор компонента в окне **Object TreeView**

Таблица 15.4. *Значения свойств компонентов* `Edit`

Компонент	Свойство	Значение
Edit1	`Top`	16
	`Left`	96
	`Height`	24
	`Width`	121
	`Text`	
Edit2	`Top`	48
	`Left`	96
	`Height`	24
	`Width`	121
	`Text`	
Edit3	`Top`	80
	`Left`	96
	`Height`	24
	`Width`	121
	`Text`	

В окне программы должна отображаться информация о назначении полей ввода. Отображение текста на форме обеспечивает компонент `Label`. Значок компонента `Label` находится на вкладке **Standard** (рис. 15.18). Свойства компонента `Label` перечислены в табл. 15.5.

Рис. 15.18. Компонент `Label` — поле вывода текста

Таблица 15.5. *Свойства компонента* `Label` *(поле отображения текста)*

Свойство	Описание
`Name`	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
`Caption`	Отображаемый текст
`Font`	Шрифт, используемый для отображения текста
`ParentFont`	Признак наследования компонентом характеристик шрифта формы, на которой находится компонент. Если значение свойства равно `True`, текст выводится шрифтом, установленным для формы
`AutoSize`	Признак того, что размер поля определяется его содержимым
`Left`	Расстояние от левой границы поля вывода до левой границы формы
`Top`	Расстояние от верхней границы поля вывода до верхней границы формы
`Height`	Высота поля вывода
`Width`	Ширина поля вывода
`WordWrap`	Признак того, что слова, которые не помещаются в текущей строке, автоматически переносятся на следующую строку

Следует обратить внимание на свойства `AutoSize` и `WordWrap`. Эти свойства нужно использовать, если поле вывода должно содержать несколько строк текста. После добавления к форме компонента `Label` значение свойства `AutoSize` равно `True`, т. е. размер поля определяется автоматически в процессе изменения значения свойства `Caption`. Если вы хотите, чтобы находящийся в поле вывода текст занимал несколько строк, то надо сразу после добавления к форме компонента `Label` присвоить свойству `AutoSize` значение `False`,

свойству WordWrap — значение True. Затем изменением значений свойств Width и Height нужно задать требуемый размер поля. Только после этого можно ввести в свойство Caption текст, который должен быть выведен в поле.

В форму разрабатываемого приложения надо добавить четыре компонента Label. Первые три компонента предназначены для вывода информации о назначении полей ввода, четвертое поле — для отображения результата расчета.

Добавляется компонент Label в форму точно так же, как и поле редактирования.

После того как компоненты будут добавлены в форму и настроены (табл. 15.6), форма программы должна выглядеть так, как показано на рис. 15.19.

Таблица 15.6. *Значения свойств компонентов* Label

Компонент	Свойство	Значение
Label1	Top	24
	Left	8
	Caption	Сумма:
Label2	Top	56
	Left	8
	Caption	Срок (дней):
Label3	Top	88
	Left	8
	Caption	Процент:
Label4	AutoSize	False
	WordWrap	True
	Top	120
	Left	8
	Height	57
	Width	209
	Caption	

Обратите внимание, что значение свойства Caption в окне **Object Inspector** вводится как одна строка. Расположение текста в поле компонента Label определяется размером компонента, значением свойств AutoSize и WordWrap,

а также зависит от характеристик используемого для вывода текста шрифта. Свойства компонентов следует устанавливать (изменять) в том порядке, в котором они приведены в таблице.

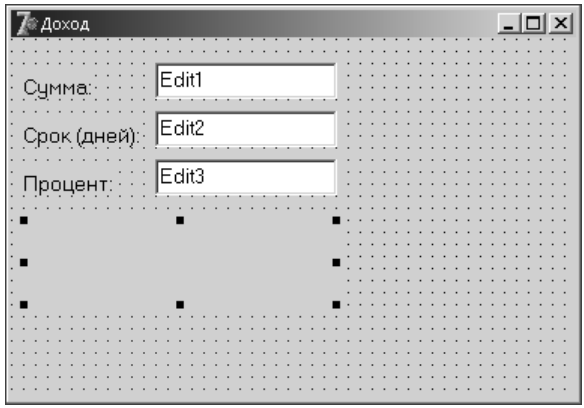


Рис. 15.19. Вид формы после настройки компонентов Label

Следующее, что надо сделать на этапе создания формы, — добавить в форму две командные кнопки: **Расчет** и **Выход**. Назначение этих кнопок очевидно.

Командная кнопка, компонент `Button`, добавляется в форму точно так же, как и другие компоненты. Значок компонента `Button` находится на вкладке **Standard** (рис. 15.20). Свойства компонента приведены в табл. 15.7.

Рис. 15.20. Командная кнопка — компонент `Button`

Таблица 15.7. *Свойства компонента* `Button` *(командная кнопка)*

Свойство	Описание
`Name`	Имя компонента. Используется в программе для доступа к компоненту и его свойствам
`Caption`	Текст на кнопке
`Enabled`	Признак доступности кнопки. Кнопка доступна, если значение свойства равно `True`, и не доступна, если значение свойства равно `False`

Таблица 15.7 *(окончание)*

Свойство	Описание
`Left`	Расстояние от левой границы кнопки до левой границы формы
`Top`	Расстояние от верхней границы кнопки до верхней границы формы
`Height`	Высота кнопки
`Width`	Ширина кнопки

Таблица 15.8. *Значения свойств компонентов* `Button`

Компонент	Свойство	Значение
Button1	`Caption`	Расчет
	`Top`	200
	`Left`	16
	`Height`	25
	`Width`	75
Button2	`Caption`	Выход
	`Top`	200
	`Left`	112
	`Height`	25
	`Width`	75

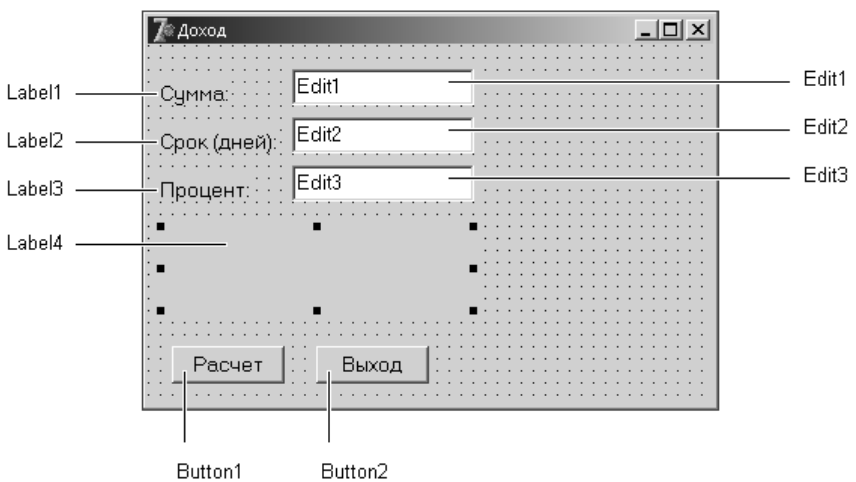


Рис. 15.21. Форма программы **Доход**

После добавления к форме двух командных кнопок нужно установить значения их свойств (табл. 15.8).

Окончательный вид формы приложения приведен на рис. 15.21.

Завершив работу по созданию формы приложения, можно приступить к созданию процедур обработки событий.

Событие и процедура обработки события

Вид формы подсказывает, как работает приложение. Очевидно, что пользователь должен ввести в поля редактирования исходные данные и щелкнуть мышью на кнопке **Расчет**. Щелчок на изображении командной кнопки — это пример того, что называется *событием*.

Событие (`Event`) — это то, что происходит во время работы программы. У каждого события есть имя. Например, щелчок кнопкой мыши — это событие `Click`, нажатие клавиши — событие `KeyPress`.

В табл. 15.9 приведены некоторые события.

Таблица 15.9. *События*

Событие	Происходит
`Click`	При щелчке кнопкой мыши
`DblClick`	При двойном щелчке кнопкой мыши
`KeyPress`	При нажатии клавиши клавиатуры
`KeyDown`	При нажатии клавиши клавиатуры. События `KeyDown` и `KeyPress` — это чередующиеся, повторяющиеся события, которые происходят до тех пор, пока не будет отпущена удерживаемая клавиша (в этот момент происходит событие `KeyUp`)
`KeyUp`	При отпускании нажатой клавиши клавиатуры
`MouseDown`	При нажатии кнопки мыши
`MouseUp`	При отпускании кнопки мыши
`MouseMove`	При перемещении мыши
`Create`	При создании объекта (формы, элемента управления). Процедура обработки этого события обычно используется для инициализации переменных, выполнения подготовительных действий

Таблица 15.9 *(окончание)*

Событие	Происходит
`Paint`	При появлении окна на экране в начале работы программы, при появлении окна, которое было свернуто, при появлении части окна, которая была закрыта другим окном, и в других случаях
`Enter`	При получении элементом управления фокуса
`Exit`	При потере элементом управления фокуса

Реакцией на событие должно быть какое-либо действие. В Delphi реакция на событие реализуется как *процедура обработки события*. Таким образом, для того чтобы программа выполняла некоторую работу в ответ на действия пользователя, программист должен написать процедуру обработки соответствующего события. Следует обратить внимание на то, что значительную часть обработки событий берет на себя компонент. Поэтому программист должен разрабатывать процедуру обработки события только в том случае, если реакция на событие отличается от стандартной или не определена. Например, если по условию задачи ограничений на символы, вводимые в поле `Edit`, нет, то процедуру обработки события `KeyPress` писать не надо, т. к. во время работы программы будет использована стандартная (скрытая от программиста) процедура обработки этого события.

Методику создания процедур обработки событий рассмотрим на примере процедуры обработки события `Click` для кнопки **Расчет**.

Чтобы приступить к созданию процедуры обработки события, надо выбрать компонент (сделать щелчок на изображении компонента), для которого создается процедура обработки события. Затем в окне **Object Inspector** следует выбрать вкладку **Events** (События).

В левой колонке вкладки **Events** (рис. 15.22) перечислены события, которые может воспринимать компонент (строго говоря, на вкладке **Events** перечислены свойства, значения которых определяют процедуры обработки соответствующих событий). Если для события определена процедура обработки, то в правой колонке вкладки **Events** отображается имя этой процедуры.

Для того чтобы создать процедуру обработки события, нужно сделать двойной щелчок мышью в поле имени процедуры обработки соответствующего события. В результате этого откроется окно редактора кода, в которое будет добавлен шаблон процедуры обработки события, а в окне **Object Inspector** рядом с именем события появится имя процедуры его обработки (рис. 15.23).

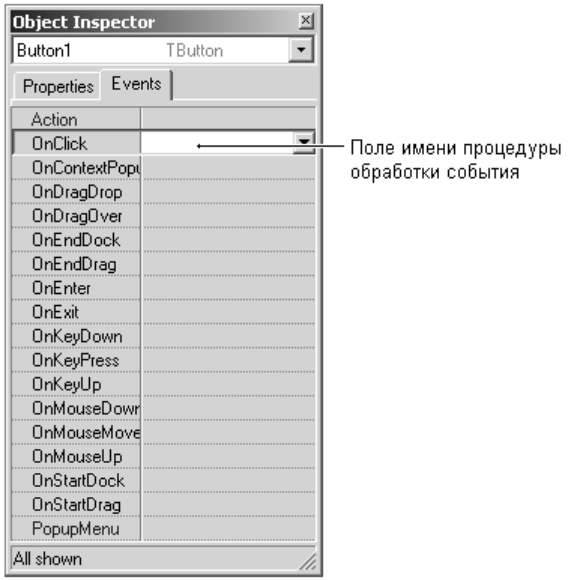


Рис. 15.22. На вкладке **Events** перечислены события, которые может воспринимать компонент (в данном случае — командная кнопка)

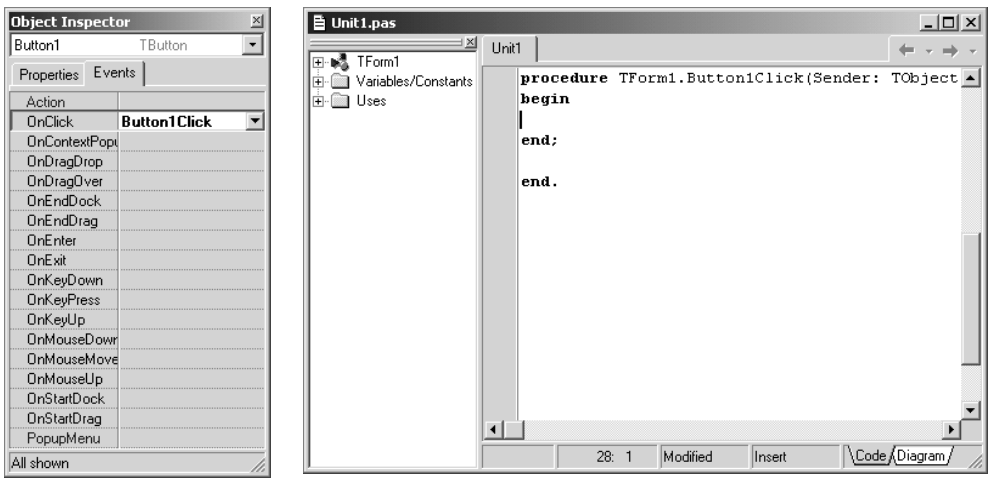


Рис. 15.23. Шаблон процедуры обработки события, сгенерированный Delphi

Delphi присваивает процедуре обработки события имя, которое состоит из двух частей. Первая часть имени идентифицирует форму, вторая — объект (компонент) и событие. В рассматриваемом примере имя формы — `Form1`, имя командной кнопки — `Button1`, а имя события — `Click`.

После того как процедура обработки события будет создана, в окне редактора кода можно набирать инструкции, реализующие процедуру обработки события.

> ***Примечание***
>
> Создать процедуру обработки события `Click` для командной кнопки можно, сделав двойной щелчок левой кнопкой мыши на изображении кнопки.

В листинге 15.1 приведена процедура обработки события `Click` для кнопки **Расчет**. Обратите внимание на то, как представлена программа. Ее общий вид соответствует тому, как она выглядит в окне редактора кода: ключевые слова выделены полужирным, комментарии — курсивом (выделение выполняет редактор кода). Кроме того, инструкции программы набраны с отступами в соответствии с принятыми в среде программистов правилами хорошего стиля.

Листинг 15.1. Обработка события Click на кнопке Расчет

```
// щелчок на кнопке Расчет
procedure TForm1.Button1Click(Sender: TObject);
var
  sum: real;         // сумма вклада
  period: integer;   // срок вклада
  interest: real;    // годовая процентная ставка

  profit: real; // доход
  total: real;  // сумма в конце срока вклада
begin
   // ввод исходных данных
   sum := StrToFloat(Edit1.Text);
   period := StrToInt(Edit2.Text);
   interest := StrToFloat(Edit3.Text) / 100;

   // расчет
   profit := sum * (interest / 365) * period;
   total := sum + profit;

   // вывод результата расчета
   Label4.Caption :=
        'Доход: ' + FloatToStr(profit) + #13 +
        'Сумма в конце срока вклада: ' + FloatToStr(total);
end;
```

Процедура Button1Click выполняет расчет и выводит результат расчета в поле Label4. Исходные данные вводятся из полей редактирования Edit1, Edit2 и Edit2 путем обращения к свойству Text. Во время работы программы свойство Text содержит строку символов, которую ввел пользователь. Для преобразования строк символов в числа используются функции StrToInt и StrToFloat. Функция StrToInt преобразует строку символов в целое число, StrToFloat — в дробное. После того как исходные данные будут помещены в переменные sum, period и interest, выполняется расчет дохода по формуле profit = sum * (interest / 365) * period, где: profit — доход; sum — сумма вклада; interest — годовая процентная ставка; 365 — количество дней в году; period — срок вклада (дней). Следует обратить внимание, что в приведенной формуле значение процентной ставки должно быть меньше единицы. Вычисленные значения дохода и сумма в конце срока вклада выводится в поле Label4 путем присваивания значения свойству Caption. Так как свойство Caption строкового типа, то численные значения преобразуются в строковые. Для преобразования используется функция FloatToStr.

В результате нажатия кнопки **Выход** программа должна завершить работу. Чтобы это произошло, надо закрыть окно программы. Делается это при помощи метода Close. Процедура обработки события Click для кнопки **Выход** приведена в листинге 15.2.

Листинг 15.2. Обработка события Click на кнопке Выход

```
// щелчок на кнопке Выход
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form1.Close;
end;
```

Редактор кода

Редактор кода выделяет ключевые слова языка программирования (procedure, var, begin, end, if и др.) полужирным шрифтом, что делает текст программы более выразительным и облегчает восприятие структуры программы.

Помимо ключевых слов, редактор кода выделяет курсивом комментарии.

В процессе разработки программы часто возникает необходимость переключения между окном редактора кода и окном формы. Сделать это можно при помощи командной кнопки **Toglle Unit/Form**, находящейся на панели инструментов **View** (рис. 15.24), или нажав клавишу <F12>. На этой же панели инструментов находятся командные кнопки **View Unit** и **View Form**, исполь-

зуя которые можно выбрать нужный модуль или форму в случае, если проект состоит из нескольких модулей или форм.

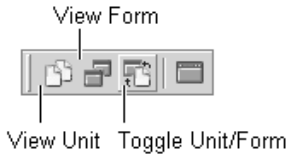


Рис. 15.24. Панель инструментов **View**

Система подсказок

В процессе набора текста программы редактор кода выводит справочную информацию о параметрах процедур и функций, о свойствах и методах объектов.

Например, если в окне редактора кода набрать `MessageDlg` (имя функции, которая выводит на экран окно сообщения) и открывающую скобку, то на экране появится окно подсказки, в котором будут перечислены параметры функции `MessageDlg` с указанием их типа (рис. 15.25). Один из параметров выделен полужирным шрифтом. Так редактор подсказывает программисту, какой параметр он должен вводить. После набора параметра и запятой в окне подсказки будет выделен следующий параметр. И так до тех пор, пока не будут указаны все параметры.

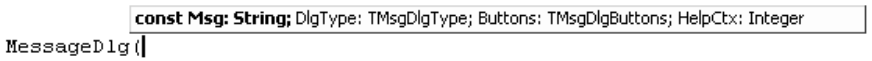


Рис. 15.25. Пример подсказки

Для объектов редактор кода выводит список свойств и методов. Как только программист наберет имя объекта (компонента) и точку, так сразу на экране появляется список свойств и методов этого объекта (рис. 15.26). Перейти к нужному элементу списка можно при помощи клавиш перемещения курсора или набрав на клавиатуре несколько первых букв имени нужного свойства или метода. После того как будет выбран нужный элемент списка и нажата клавиша <Enter>, выбранное свойство или метод будут вставлены в текст программы.

Система подсказок существенно облегчает процесс подготовки текста программы, избавляет от рутины. Кроме того, если во время набора программы подсказка не появилась, то это значит, что программист допустил ошибку, скорее всего, неверно набрал имя процедуры или функции.

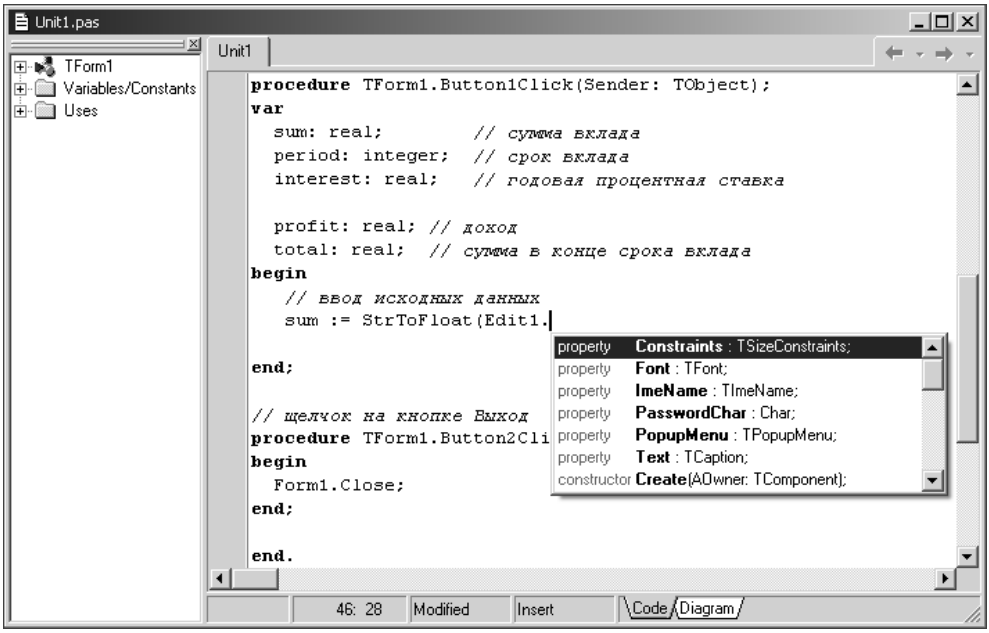


Рис. 15.26. Редактор кода автоматически выводит список свойств и методов объекта (компонента)

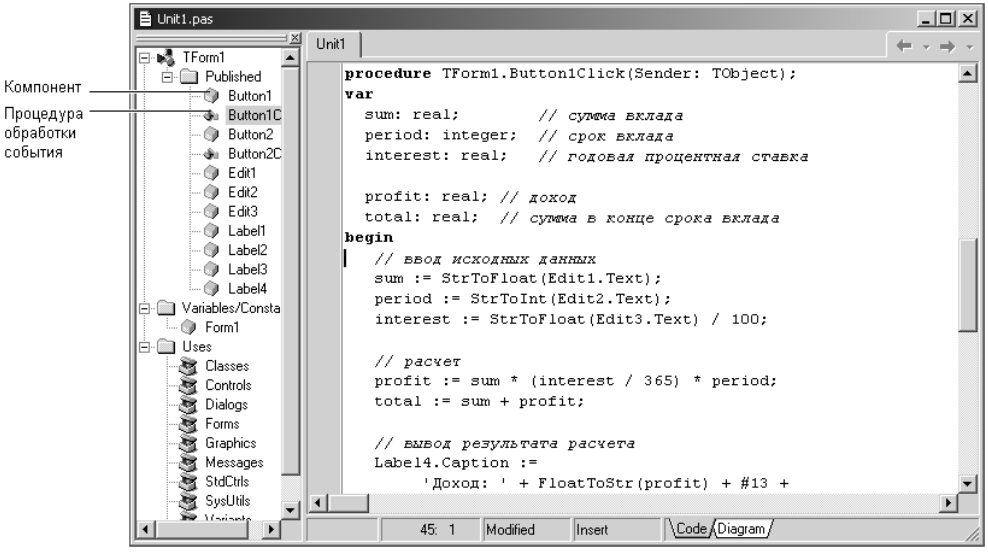


Рис. 15.27. Окно **Code Explorer** облегчает навигацию по тексту программы

Навигатор кода

Окно редактора кода разделено на две части (рис. 15.27). В правой части находится текст программы. Левая часть, которая называется навигатором кода (**Code Explorer**), облегчает навигацию по тексту (коду) программы. В иерархическом списке, структура которого зависит от проекта, над которым идет работа, перечислены формы проекта, их компоненты, процедуры обработки событий, функции, процедуры, глобальные переменные и константы. Выбрав соответствующий элемент списка, можно быстро перейти к нужному фрагменту кода.

Если окно навигатора кода не отображается, то для того чтобы оно появилось на экране, нужно из меню **View** выбрать команду **Code Explorer**.

Шаблоны кода

В процессе набора текста удобно использовать *шаблоны кода* (Code Templates). Шаблон кода — это инструкция языка программирования, записанная в общем виде. Например, шаблон для инструкции `case` выглядит так:

```
case  of
    : ;
    : ;
  else ;
  end;
```

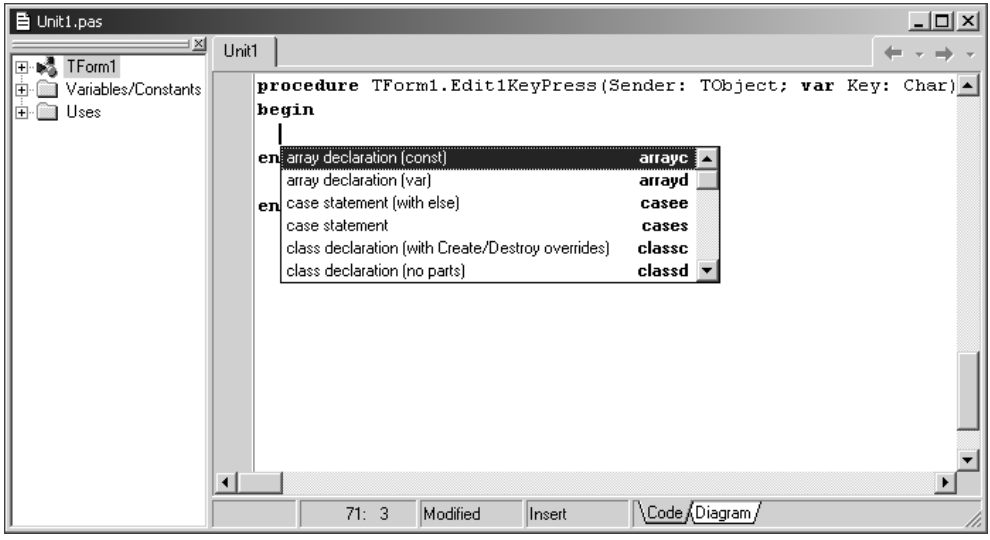


Рис. 15.28. Список шаблонов кода отображается в результате нажатия <Ctrl>+<J>

Редактор кода предоставляет программисту большой набор шаблонов: объявления массивов, классов, функций, процедур; инструкций выбора (`if`, `case`), циклов (`for`, `while`). Для некоторых инструкций, например, `if` и `while`, есть несколько вариантов шаблонов.

Для того чтобы в процессе набора текста программы воспользоваться шаблоном кода и вставить его в текст программы, нужно нажать комбинацию клавиш <Ctrl>+<J> и в появившемся списке выбрать нужный шаблон (рис. 15.28). Шаблон можно выбрать обычным образом, прокручивая список, или вводом первых букв имени шаблона (имена шаблонов в списке выделены полужирным шрифтом). Выбрав в списке шаблон, нужно нажать <Enter>, шаблон будет вставлен в текст программы.

Справочная система

В процессе набора программы можно получить справку, например, о конструкции языка или функции. Для этого нужно в окне редактора кода набрать слово (инструкцию языка программирования, имя процедуры или функции и т. д.), о котором надо получить справку, и нажать клавишу <F1>.

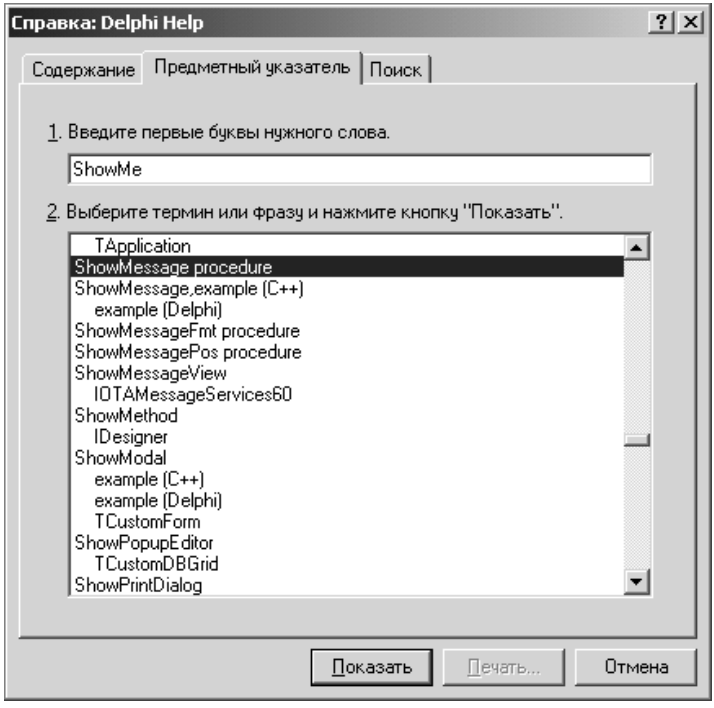


Рис. 15.29. Поиск справочной информации по ключевому слову

Справочную информацию можно получить, выбрав из меню **Help** команду **Delphi Help**. В этом случае на экране появится окно справочной системы (рис. 15.29). В этом окне на вкладке **Предметный указатель** нужно ввести ключевое слово, определяющее тему, по которой нужна справка. Как правило, в качестве ключевого слова используют первые несколько букв имени функции, процедуры, свойства или метода.

Структура проекта

Программу (приложение), над которой работает программист, принято называть проектом. Проект представляет собой набор программных единиц — модулей. Один из модулей — главный, содержит инструкции, с которых начинается выполнение программы. Главный модуль формирует Delphi.

Главный модуль представляет собой файл с расширением dpr. Для того чтобы увидеть текст главного модуля приложения, нужно из меню **Project** выбрать команду **View Source**.

В листинге 15.3 приведен главный модуль программы вычисления дохода по вкладу.

Листинг 15.3. Главный модуль приложения Доход

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Начинается главный модуль словом `program`, за которым следует имя программы, которое совпадает с именем проекта. Имя проекта задается в момент сохранения проекта, и оно определяет имя создаваемого компилятором ис-

полняемого файла программы (exe-файла). Далее за словом `uses` следуют имена используемых модулей: библиотечного модуля `Forms` и модуля формы.

Строка `{$R*.RES}`, которая похожа на комментарий, — это директива компилятору подключить файл *ресурсов*. Файл ресурсов содержит *ресурсы* приложения: пиктограммы, курсоры, битовые образы и др. Звездочка показывает, что имя файла ресурсов такое же, как и у файла проекта, но с расширением res.

Файл ресурсов не является текстовым файлом, поэтому просмотреть его с помощью редактора текста нельзя. Для работы с ресурсами можно использовать утилиту Image Editor, доступ к которой можно получить выбором в меню **Tools** команды **Image Editor**.

Исполняемая часть главного модуля находится между инструкциями `begin` и `end`. Инструкции исполняемой части обеспечивают инициализацию приложения, создание стартовой формы (окна) и вывод на экран стартового окна.

Помимо главного модуля, в каждом проекте есть, как минимум, один модуль формы, который содержит описание стартовой формы и поддерживающих ее работу процедур. Модуль стартовой формы программы вычисления дохода по вкладу приведен в листинге 15.4.

Листинг 15.4. Модуль стартовой формы программы Доход

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Button1: TButton;
```

```
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// щелчок на кнопке Расчет
procedure TForm1.Button1Click(Sender: TObject);
var
  sum: real;        // сумма вклада
  period: integer;  // срок вклада
  interest: real;   // годовая процентная ставка

  profit: real; // доход
  total: real;  // сумма в конце срока вклада
begin
   // ввод исходных данных
   sum := StrToFloat(Edit1.Text);
   period := StrToInt(Edit2.Text);
   interest := StrToFloat(Edit3.Text) / 100;

   // расчет
   profit := sum * (interest / 365) * period;
   total := sum + profit;

   // вывод результата расчета
   Label4.Caption :=
        'Доход: ' + FloatToStr(profit) + #13 +
```

```
        'Сумма в конце срока вклада: ' + FloatToStr(total);
end;

// щелчок на кнопке Выход
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form1.Close;
end;

end.
```

Начинается модуль словом `unit`, за которым следует имя модуля. Именно это имя упоминается в списке используемых модулей в инструкции `uses` главного модуля приложения, текст которого приведен в листинге 15.3.

Модуль состоит из следующих разделов:

- интерфейса;
- реализации;
- инициализации.

Раздел интерфейса (начинается словом `interface`) сообщает компилятору, какая часть модуля является доступной для других модулей программы. В этом разделе перечислены (после слова `uses`) библиотечные модули, используемые данным модулем. Также здесь находится сформированное Delphi описание типа формы, которое следует за словом `type`.

Раздел реализации открывается словом `implementation` и содержит объявления локальных переменных, процедур и функций, поддерживающих работу формы.

Начинается раздел реализации директивой `{$R *.DFM}`, указывающей компилятору, что в процессе генерации выполняемого файла надо использовать описание формы. Описание формы находится в файле с расширением dfm, имя которого совпадает с именем модуля. Файл описания формы генерирует Delphi на основе внешнего вида формы.

За директивой `{$R *.DFM}` следуют процедуры обработки событий. Сюда же программист может поместить другие процедуры и функции.

Раздел инициализации позволяет выполнить инициализацию переменных модуля. Инструкции раздела инициализации располагаются после раздела реализации (описания всех процедур и функций) между `begin` и `end`. Если раздел инициализации не содержит инструкций (как в приведенном примере), то слово `begin` не указывается.

Следует отметить, что значительное количество инструкций модуля формирует Delphi. Например, Delphi, анализируя действия программиста по созданию формы, генерирует описание класса формы (после слова `type`). В приведенном примере инструкции, набранные программистом, выделены фоном. Очевидно, что Delphi выполнила значительную часть работы по составлению текста программы.

Сохранение проекта

Проект — это набор файлов, используя которые компилятор создает исполняемый файл программы (exe-файл). В простейшем случае проект состоит из файла описания проекта (dof-файл), файла главного модуля (dpr-файл), файла ресурсов (res-файл), файла описания формы (dfm-файл), файла модуля формы, в котором находится основной код приложения, в том числе процедуры обработки (pas-файл), файл конфигурации (cfg-файл).

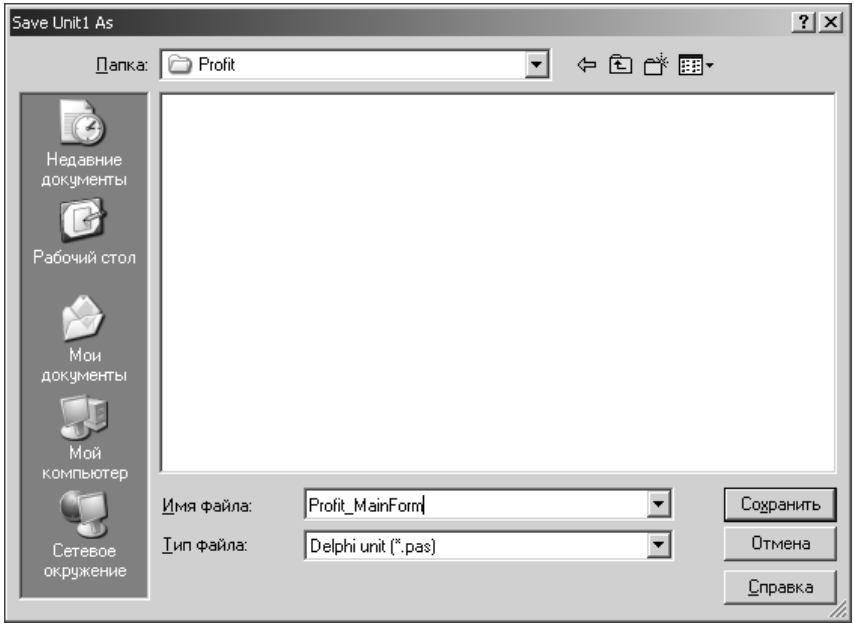


Рис. 15.30. Сохранение модуля формы

Чтобы сохранить проект, нужно в меню **File** выбрать команду **Save Project As**. Если проект еще ни разу не был сохранен, то Delphi сначала предложит сохранить модуль формы (содержимое окна редактора кода), поэтому на экране появится окно **Save Unit1 As**. В этом окне надо выбрать папку, предна-

значенную для проектов (по умолчанию проекты сохраняются в папке c:\Program Files\Borland\Delphi 7\Projects), щелчком на кнопке **Создать новую папку** создать папку для сохраняемого проекта, открыть созданную папку и в поле **Имя файла** ввести имя сохраняемого модуля (рис. 15.30). После нажатия кнопки **Сохранить** появляется окно **Save Project As**, в котором необходимо ввести имя проекта (рис. 15.31).

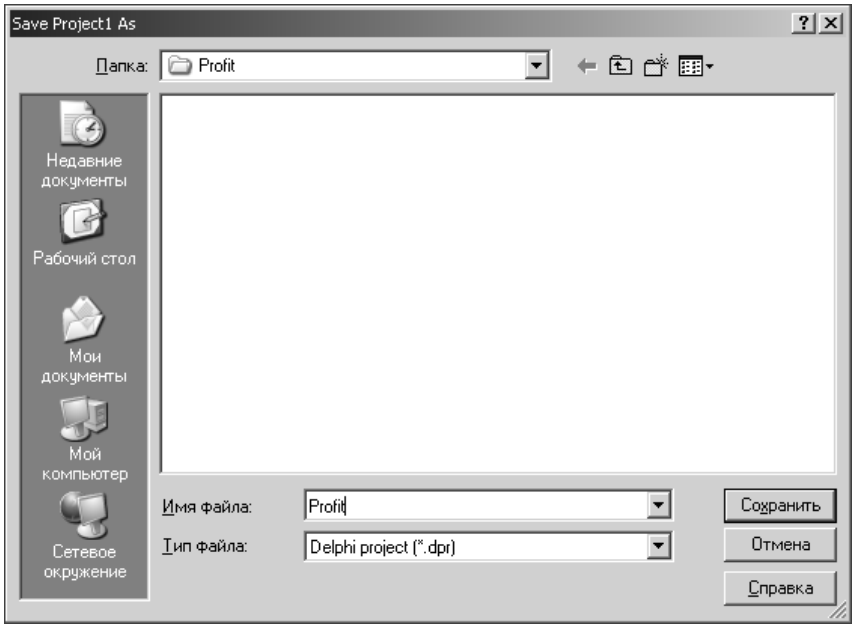


Рис. 15.31. Сохранение проекта

Обратите внимание на то, что имена файлов модуля (pas-файл) и проекта (dpr-файл) должны быть разными. Имя генерируемого компилятором исполняемого файла совпадает с именем проекта. Поэтому файлу проекта следует присвоить такое имя, которое, по вашему мнению, должен иметь исполняемый файл программы, а файлу модуля — какое-либо другое имя, например, полученное путем добавления к имени файла проекта порядкового номера модуля.

Компиляция

После сохранения проекта можно в меню **Project** выбрать команду **Compile** и выполнить компиляцию. Процесс и результат компиляции отражаются в диалоговом окне **Compiling** (рис. 15.32). В это окно компилятор выводит

информацию о количестве ошибок (`Errors`), предупреждений (`Warnings`) и подсказок (`Hints`). Сами сообщения об ошибках, предупреждения и подсказки отображаются в нижней части окна редактора кода (рис. 15.33).

Примечание

Если во время компиляции окно **Compiling** на экране не отображается, то выберите в меню **Tools** команду **Environment options** и на вкладке **Preferences** установите во включенное состояние переключатель **Show compiler progress**.

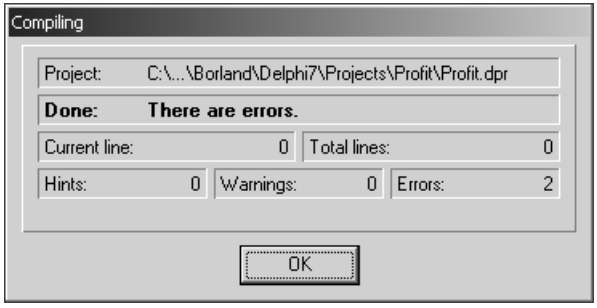


Рис. 15.32. Результат компиляции

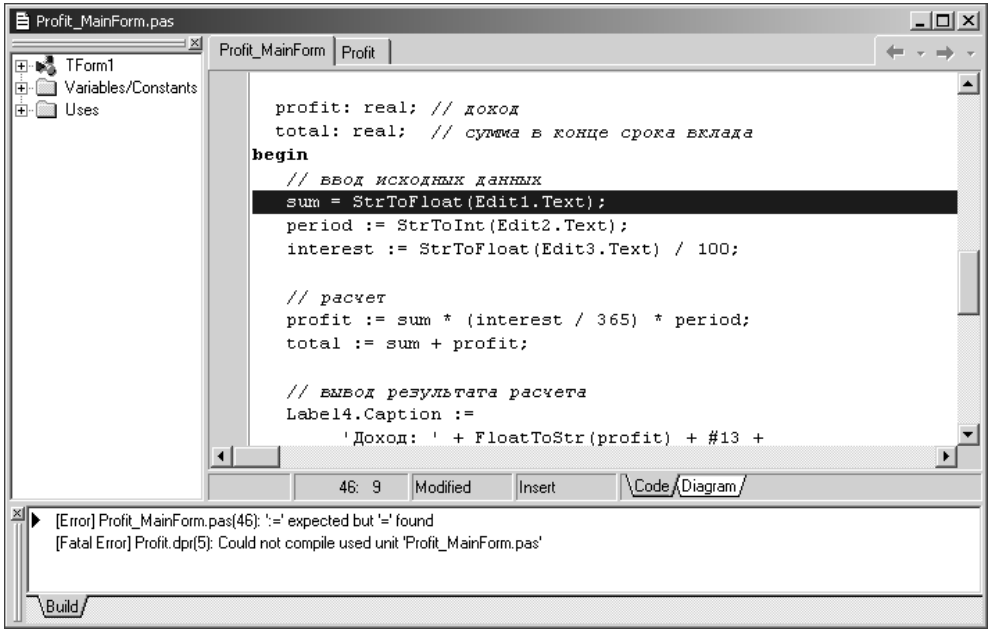


Рис. 15.33. Сообщения компилятора об обнаруженных ошибках

Ошибки

Компилятор генерирует исполняемую программу лишь в том случае, если исходный текст не содержит синтаксических ошибок. В большинстве случаев в только что набранной программе есть ошибки. Программист должен их устранить.

Чтобы перейти к фрагменту кода, который содержит ошибку, надо сделать двойной щелчок левой кнопкой мыши в строке сообщения об ошибке.

В табл. 15.10 перечислены наиболее типичные ошибки и соответствующие им сообщения компилятора.

Таблица 15.10. *Сообщения компилятора об ошибках*

Сообщение	Вероятная причина
`Undeclared identifier` (Необъявленный идентификатор)	• Используется переменная, не объявленная в разделе `var` • Ошибка при написании имени переменной. Например, объявлена переменная `Sum`, а в тексте программы написано: `Suma`
`Unterminated string` (Незавершенная строка)	При записи строковой константы, например, сообщения, не поставлена завершающая кавычка
`Incompatible types ...and ...` (Несовместимые типы)	В инструкции присваивания тип выражения не соответствует или не может быть приведен к типу переменной, получающей значение выражения
`Missing operator or semicolon` (Отсутствует оператор или точка с запятой)	После инструкции не поставлена точка с запятой

Если компилятор обнаружил достаточно много ошибок, то просмотрите все сообщения, устраните сначала наиболее очевидные ошибки и выполните повторную компиляцию. Вполне вероятно, что после этого количество ошибок значительно уменьшится. Это объясняется особенностями синтаксиса языка, когда одна незначительная ошибка может "тащить" за собой довольно большое количество других.

Если в программе нет синтаксических ошибок, компилятор создает исполняемый файл программы. Имя исполняемого файла такое же, как и у файла проекта, а расширение — exe. Delphi помещает исполняемый файл в тот же каталог, где находится файл проекта.

Предупреждения и подсказки

При обнаружении в программе неточностей, которые не являются ошибками, компилятор выводит подсказки (`Hints`) и предупреждения (`Warnings`).

Например, наиболее часто выводимой подсказкой является сообщение об объявленной, но не используемой переменной:

```
Variable ... is declared but never used in ...
```

Действительно, зачем объявлять переменную и не использовать ее?

В табл. 15.11 приведены предупреждения, наиболее часто выводимые компилятором.

Таблица 15.11. *Предупреждения компилятора*

Предупреждение	Вероятная причина
`Variable... is declared but never used in ...`	Переменная не используется
`Variable ... might not have been initialized.` (вероятно, используется неинициализированная переменная)	В программе нет инструкции, которая присваивает переменной начальное значение

Запуск программы

Пробный запуск программы можно выполнить непосредственно из Delphi. Для этого нужно в меню **Run** выбрать команду **Run** или сделать щелчок на соответствующей кнопке панели инструментов **Debug** (рис. 15.34).

Рис. 15.34. Запуск программы из среды разработки

Ошибки времени выполнения

Во время работы приложения могут возникать ошибки, которые называют *исключениями* (`exceptions`). В большинстве случаев причинами исключений являются неверные исходные данные. Например, если в поле **Процент** окна программы вычисления дохода по вкладу ввести `9.5`, т. е. для отделения дробной части числа от целой использовать точку, то в результате нажатия кнопки **Расчет** на экране появится окно с сообщением об ошибке (рис. 15.35).

Рис. 15.35. Пример ошибки времени выполнения (программа запущена из Windows)

Причина возникновения ошибки заключается в следующем. При вводе данных в поле редактирования пользователь может (если не предпринять никаких дополнительных усилий) отделить дробную часть числа от целой точкой или запятой. Какой из этих двух символов является допустимым, зависит от настройки Windows. Если в настройке Windows указано, что разделитель целой и дробной частей числа — запятая (для России — это стандартная установка), а пользователь во время работы программы введет в поле редактирования, например, строку `9.5`, то при выполнении инструкции

```
interest := StrToFloat(Edit3.Text)
```

возникнет исключение, т. к. при стандартной для России настройке Windows содержимое поля `Edit2` и, следовательно, аргумент функции `StrToFloat` не является изображением дробного числа.

Если программа запущена из среды разработки, то при возникновении исключения выполнение программы приостанавливается, и на экране появляется окно с сообщением об исключении и его типе. В качестве примера на рис. 15.36 приведено окно с сообщением о том, что введенная пользователем строка не является дробным числом.

После нажатия кнопки **OK** программист может продолжить выполнение программы (для этого надо в меню **Run** выбрать команду **Step Over**) или прервать выполнение программы. В последнем случае нужно из меню **Run** выбрать команду **Program Reset**.

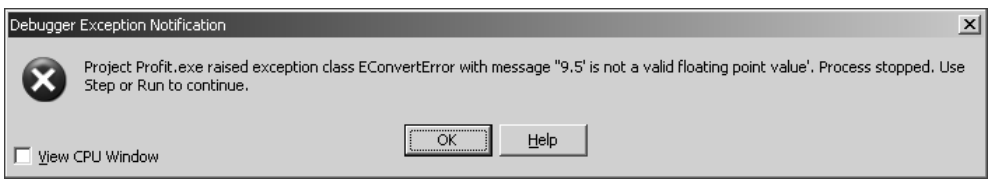


Рис. 15.36. Пример сообщения о возникновении исключения (программа запущена из Delphi)

При разработке программы программист должен постараться предусмотреть все возможные действия пользователя, которые могут привести к возникновению ошибок времени выполнения (исключения), и обеспечить способы защиты от них.

В листинге 15.5 приведена версия программы **Доход**, в которой реализована защита от некоторых некорректных действий пользователя, в частности, программа позволяет вводить в поля редактирования только числа.

Внесение изменений

После нескольких запусков программы **Доход** возникает желание внести изменения в программу. Например, было бы неплохо, чтобы после ввода суммы вклада и нажатия клавиши <Enter> курсор переходил в поле **Срок**, а также чтобы в поля редактирования можно было ввести только правильные данные (в поля **Сумма** и **Процент** — дробные числа, а в поле **Срок** — целое).

Чтобы внести изменения в программу, нужно запустить Delphi и открыть соответствующий проект. Сделать это можно обычным способом, выбрав в меню **File** команду **Open Project**. Можно также воспользоваться командой **File ▸ Reopen**, в результате выбора которой открывается список проектов, над которыми в последнее время работал программист.

В листинге 15.5 приведена программа **Доход**, в которую добавлены процедуры обработки событий `KeyPress` для компонентов `Edit1`, `Edit2` и `Edit3`. Эти процедуры не позволяют пользователю ввести в поля редактирования неверные символы. Каждая из этих процедур проверяет символ, соответствующий нажатой клавише (символ передается в процедуру обработки через параметр `Key`), и, если символ неверный, заменяет его "нулевым" символом. В результате символ в поле редактирования не появляется, а у пользователя складывается впечатление, что программа не реагирует на нажатие клавиши. Следует обратить внимание на то, как в программе реализована обработка десятичного разделителя. Если пользователь нажал точку вместо запятой, то точка заменяется на запятую. Также следует обратить внимание на то, что для вывода результата вместо функции `FloatToStr` используется функция `FloatToStrF` (см. процедуру обработки события `Click` кнопки `Button1`). Эта функция позволяет задать формат формируемой строки. В рассматриваемом примере функция возвращает численное значение в денежном (финансовом) формате — дописывает после числа обозначение денежной единицы. Кроме того, в программу добавлена процедура обработки события `Change` для полей редактирования (событие возникает при изменении содержимого поля редактирования в результате добавления или удаления символа). Эта процедура обрабатывает событие `Change` *всех* компонентов `Edit`. Она проверяет содержимое полей редактирования и делает доступной кнопку **Расчет** только

в том случае, если во всех полях есть данные (во время создания формы свойству `Enabled` кнопки `Button1` следует присвоить значение `False`). Создается процедура так. Сначала обычным образом надо создать процедуру для поля редактирования `Edit1`, назвав ее `EditChange`. Затем надо выбрать компонент `Edit2`, на вкладке **Events** в поле значения свойства `OnChange` раскрыть список и выбрать процедуру обработки события (рис. 15.37). Аналогичным образом надо задать процедуру обработки события `Change` для компонента `Edit3`. В принципе, для каждого из компонентов `Edit` можно создать свою процедуру обработки события `Change`, но тогда все три процедуры будут содержать один и тот же код.

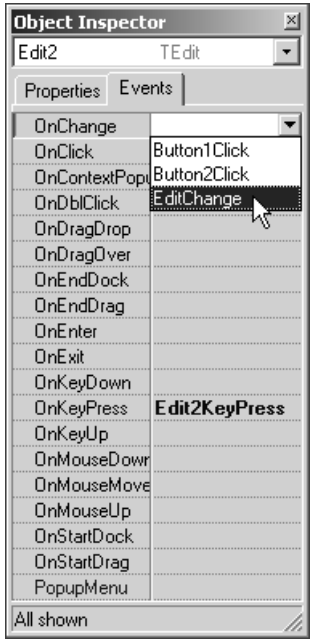


Рис. 15.37. Назначение существующей процедуры обработки события другому компоненту

Листинг 15.5. Модуль стартовой формы программы Доход после внесения изменений

```
unit Profit_MainForm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
```

```
type
  TForm1 = class(TForm)
    Edit1: TEdit;
    Edit2: TEdit;
    Edit3: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Button1: TButton;
    Button2: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Edit1KeyPress(Sender: TObject; var Key: Char);
    procedure Edit2KeyPress(Sender: TObject; var Key: Char);
    procedure Edit3KeyPress(Sender: TObject; var Key: Char);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

// щелчок на кнопке Расчет
procedure TForm1.Button1Click(Sender: TObject);
var
  sum: real;         // сумма вклада
  period: integer;   // срок вклада
  interest: real;    // годовая процентная ставка

  profit: real; // доход
  total: real;  // сумма в конце срока вклада
begin
   // ввод исходных данных
   sum := StrToFloat(Edit1.Text);
```

```
   period := StrToInt(Edit2.Text);
   interest := StrToFloat(Edit3.Text) / 100;

   // расчет
   profit := sum * (interest / 365) * period;
   total := sum + profit;

   // вывод результата расчета
   Label4.Caption :=
        'Доход: ' + FloatToStrF(profit,ffCurrency,6,2) + #13 +
        'Сумма в конце срока вклада: ' +
         FloatToStrF(total,ffCurrency,6,2);
end;

// щелчок на кнопке Выход
procedure TForm1.Button2Click(Sender: TObject);
begin
  Form1.Close;
end;

// нажатие клавиши в поле Сумма
procedure TForm1.Edit1KeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0' .. '9': ; // <Backspace> и цифры
    '.',',','б','ю':  // точка  и запятая
        begin
           Key := ',';
           if pos(',',Edit1.Text) <> 0
               then Key := #0;
        end;
    #13:                // клавиша <Enter>
        Edit2.SetFocus;  // переместить курсор в поле Срок (Edit2)

  else // остальные клавиши (символы) запрещены
      Key := #0;
  end;

end;

// нажатие клавиши в поле Срок
procedure TForm1.Edit2KeyPress(Sender: TObject; var Key: Char);
```

```
begin
  case Key of
    #8, '0' .. '9': ; // <Backspace> и цифры

    #13:                  // клавиша <Enter>
        Edit3.SetFocus;  // переместить курсор в поле Процент (Edit3)

  else // остальные клавиши (символы) запрещены
      Key := #0;
  end;
end;

// нажатие клавиши в поле Процент
procedure TForm1.Edit3KeyPress(Sender: TObject; var Key: Char);
begin
  case Key of
    #8, '0' .. '9': ; // <Backspace> и цифры
    '.',',','б','ю':  // точка  и запятая
        begin
           Key := ',';
           if pos(',',Edit3.Text) <> 0
               then Key := #0;
        end;
    #13:                    // клавиша <Enter>
        Button1.SetFocus;  // переместить фокус на кнопку Расчет

  else // остальные клавиши (символы) запрещены
      Key := #0;
  end;
end;

// изменилось содержимое поля редактирования
// эта процедура обрабатывает событие Change
// компонентов Edit1, Edit2 и Edit3
procedure TForm1.EditChange(Sender: TObject);
begin
  Label4.Caption := '';
  if (Length(Edit1.Text) = 0) or
     (Length(Edit2.Text) = 0) or
     (Length(Edit3.Text) = 0)
  then
    Button1.Enabled := False
```

```
  else
    Button1.Enabled := True;
end;
end.
```

После внесения изменений проект следует сохранить. Для этого нужно из меню **File** выбрать команду **Save all**.

Окончательная настройка приложения

После того как программа отлажена, необходимо выполнить ее окончательную настройку — задать название приложения и значок, который будет изображать исполняемый файл программы в папке или на рабочем столе, а также программу на панели задач во время ее работы.

Настройка приложения выполняется на вкладке **Application** окна **Project Options** (рис. 15.38), которое появляется в результате выбора в меню **Project** команды **Options**. Название приложения надо ввести в поле **Title**.

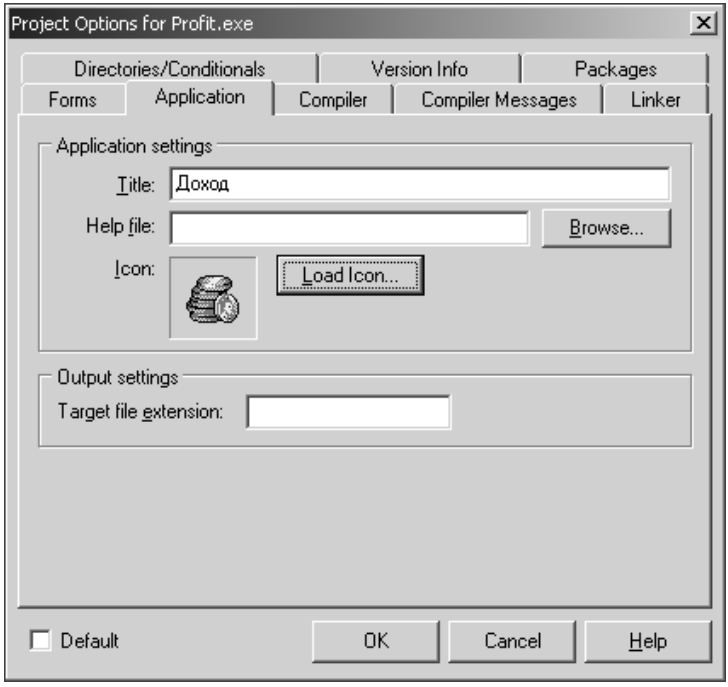


Рис. 15.38. На вкладке **Application** надо задать название приложения и значок

Чтобы назначить приложению значок, нужно сделать щелчок на кнопке **Load Icon** и, используя стандартное окно просмотра папок, найти подходящий значок (значки находятся в файлах с расширением ico). Если подходящего значка нет, то его можно создать при помощи утилиты Image Editor.

Создание значка для приложения

В состав Delphi входит утилита **Image Editor**, при помощи которой программист может создать для своего приложения уникальный значок. Запускается **Image Editor** выбором соответствующей команды в меню **Tools**.

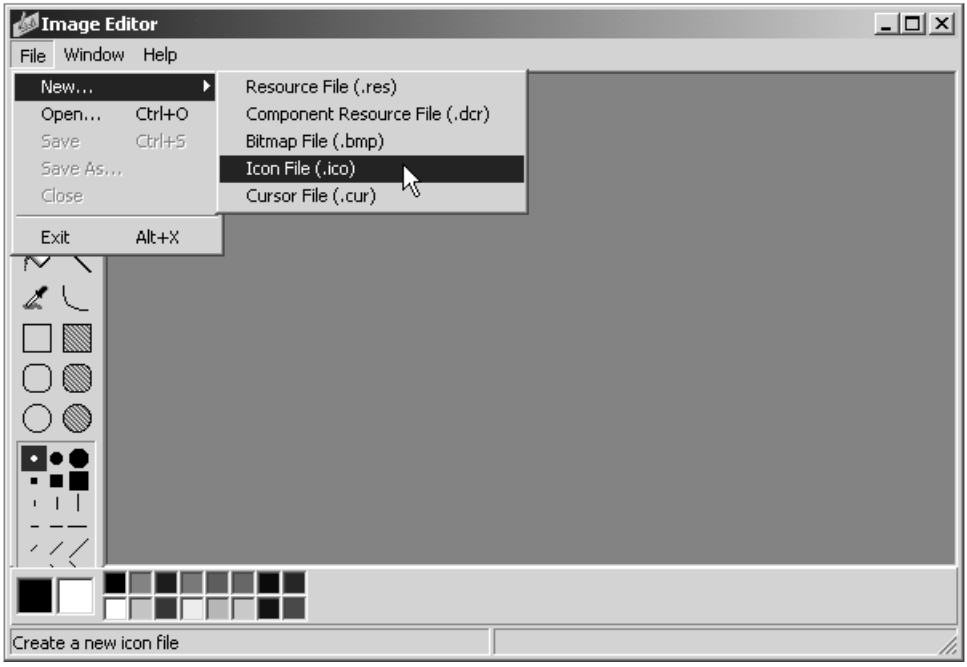


Рис. 15.39. Начало работы над новым значком

Чтобы начать работу над новым значком, нужно в меню **File** выбрать команду **New ▸ Icon File** (рис. 15.39), в появившемся окне **Icon Properties** (рис. 15.40) выбрать характеристики значка (`Size` — 32×32; `Colors` — 16) и сделать щелчок на кнопке **OK**. В результате откроется окно графического редактора (рис. 15.41), в котором можно нарисовать значок.

Процесс рисования в Image Editor практически ничем не отличается от процесса создания картинки в обычном графическом редакторе. Однако есть одна тонкость. Первоначально поле изображения закрашено "прозрачным" (`transparent`) цветом. Если значок нарисовать на этом фоне, то при его вы-

воде части поля изображения, закрашенные "прозрачным" цветом, примут цвет фона, на котором будет находиться значок.

В процессе рисования картинки можно удалить (стереть) ошибочно нарисованные элементы, закрасив их прозрачным цветом (рис. 15.42).

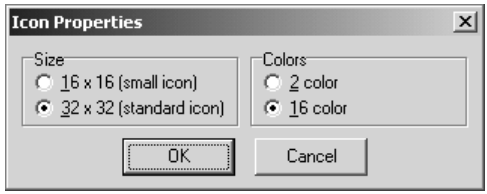


Рис. 15.40. Выбор характеристик значка

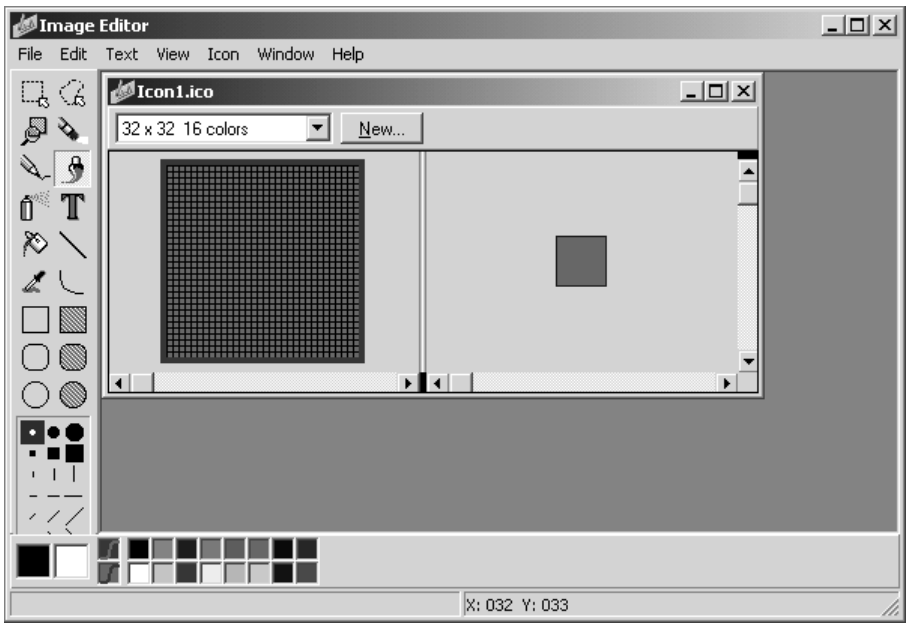


Рис. 15.41. Окно рисования значка

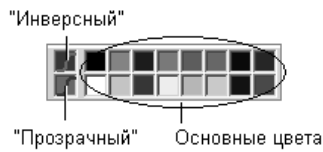


Рис. 15.42. Палитра

Кроме "прозрачного" цвета, в палитре есть "инверсный" цвет. Нарисованные этим цветом части рисунка при выводе на экран окрашиваются инверсным относительно цвета фона цветом.

Чтобы сохранить нарисованный значок, надо в меню **File** выбрать команду **Save**.

Установка приложения на другой компьютер

Программу (exe-файл), созданную в Delphi, можно перенести на другой компьютер. Вместе с тем следует понимать, что приложение, созданное в Delphi, будет работать на другом компьютере только в том случае, если на этом компьютере есть динамические библиотеки, необходимые для его работы (на компьютер программиста эти библиотеки устанавливаются вместе с Delphi). Таким образом, помимо exe-файла, на компьютер пользователя надо перенести динамические библиотеки, используемые программой. В простейшем случае, если программа использует стандартные компоненты, на компьютер пользователя надо перенести библиотеку визуальных компонентов (Visual Components Library) и библиотеку времени выполнения (Runtime Library). На компьютере программиста эти библиотеки (файлы vcl70.bpl и rtl70.bpl) находятся в каталоге c:\Windows\System32.

На компьютере пользователя библиотеки, используемые программой, должны находиться в том же каталоге, что и exe-файл, или, если эти библиотеки используются несколькими программами, то в каталоге System32.

Весь необходимый для работы программы код можно включить в EXE-файл (в этом случае динамические библиотеки на компьютер переносить не надо). Для этого, перед тем, как выполнить компиляцию, надо в меню **Project** выбрать команду **Options**, и на вкладке **Packages** появившегося окна **Project options** сбросить флажок **Build with runtime packages**.

Модель объекта в Delphi

Чтобы разрабатывать приложения в Delphi на базе предоставляемых средой разработки компонентов, не обязательно обладать глубокими знаниями в области ООП. Однако для более глубокого понимания того, как программа взаимодействует с компонентами, что и почему Delphi добавляет в текст программы, материал данной главы весьма полезен.

В Delphi 7 в качестве языка программирования используется объектно-ориентированный язык Object Pascal, который хотя и является развитием Turbo Pascal, имеет несколько весьма существенных отличий от своего

предшественника. В частности, в Delphi используется иная, чем в Turbo Pascal, модель объекта.

Класс

Классический язык Pascal позволяет программисту определять свои собственные объекты (object). Object Pascal, поддерживая концепцию объектно-ориентированного программирования, дает возможность определять *классы*. Класс — это тип, включающий в себя, помимо описания типов данных, описание процедур и функций, которые могут быть выполнены над представителем (экземпляром) класса — объектом.

Вот пример объявления простого класса:

```
TPerson = class
  private
    fname: string[15];
    faddress: string[35];
  public
    procedure show;
end;
```

Данные класса называются полями, процедуры и функции — методами. В приведенном примере `TPerson` — это имя класса, `fname` и `faddress` — имена полей, `show` — имя метода.

В программе описание (объявление) класса помещают в раздел описания типов (type).

Объект

В программе представители класса — объекты объявляются в разделе `var`. Например, так:

```
var
  student: TPerson;
  professor: TPerson;
```

Следует обратить особое внимание на то, что в Object Pascal объект — это динамическая структура. Переменная-объект содержит не данные, а ссылку на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных.

Выделение памяти осуществляется при помощи специального метода класса — конструктора, которому обычно присваивают имя `create` (создать).

Чтобы подчеркнуть особую роль и поведение конструктора, в описании класса вместо слова `procedure` используется слово `constructor`. Ниже приведено описание класса `TPerson`, в состав которого введен конструктор:

```
TPerson = class
  private
     fname:string[15];
     faddress:string[35];
     constructor create;  // конструктор
  public
    procedure show;       // метод
end;
```

Выделение памяти для данных объекта происходит присваиванием значения результата применения метода-конструктора к типу (классу) объекта. Например, после выполнения инструкции

```
professor := TPerson.create;
```

выделяется необходимая память для данных объекта `professor`.

Помимо выделения памяти, конструктор, как правило, решает задачу присваивания полям объекта начальных значений, т. е. осуществляет инициализацию объекта. Ниже приведен пример реализации конструктора для объекта `TPerson`.

```
constructor TPerson.create;
begin
   fname:='';
   faddress:='';
end;
```

Реализация конструктора несколько необычна. Во-первых, в теле конструктора нет привычных инструкций `New`, обеспечивающих выделение динамической памяти (всю необходимую работу по выделению памяти выполняет код, формируемый компилятором). Во-вторых, формально, конструктор не возвращает значения, хотя в программе обращение к конструктору осуществляется как к методу-функции.

После объявления и инициализации объект можно использовать, например, установить значение поля объекта. Доступ к полю объекта осуществляется указанием имени объекта и имени поля, которые отделяются друг от друга точкой. Хотя объект является ссылкой, однако правило доступа к данным с помощью ссылки, согласно которому после имени переменной, являющейся ссылкой, надо ставить значок ^, на объекты не распространяется. Например, для доступа к полю `fname` объекта `professor` вместо

```
professor^.fname
```

надо писать

```
professor.fname
```

Очевидно, что такой способ доступа к полям объекта более нагляден.

Если в программе некоторый объект больше не используется, то можно освободить память, занимаемую полями этого объекта. Для выполнения этого действия используют метод-деструктор `free`. Например, чтобы освободить память занимаемую полями объекта `professor`, достаточно записать:

```
professor.free;
```

Метод

Методы класса (процедуры и функции, объявление которых включено в описание класса) выполняют действия над объектами класса. Чтобы метод был выполнен, надо указать имя объекта и имя метода, отделив одно имя от другого точкой. Например, инструкция

```
professor.show;
```

вызывает применение метода `show` к объекту `professor`. Фактически инструкция применения метода к объекту — это специфический способ записи инструкции вызова процедуры.

В программе методы класса определяются точно так же, как обычные процедуры и функции, за исключением того, что имя процедуры или функции, являющейся методом, состоит из двух частей: имени класса, к которому принадлежит метод, и имени метода. Имя класса от имени метода отделяется точкой.

Ниже приведен пример определения метода `show` приведенного выше класса `TPerson`.

```
// метод show класса TPerson
procedure TPerson.show;
   begin
      ShowMessage('Имя:' + fname + #13 + 'Адрес:' + faddress);
   end;
```

Следует обратить внимание на то, что в инструкциях метода доступ к полям объекта осуществляется без указания имени объекта.

Инкапсуляция и свойства объекта

Под *инкапсуляцией* понимается скрытие полей объекта с целью обеспечения доступа к ним только посредством методов класса.

В Object Pascal ограничение доступа к полям объекта реализуется при помощи свойств объекта. Свойство объекта характеризуется полем, хранящим значение свойства, и двумя методами, обеспечивающими доступ к полю свойства. Метод установки значения свойства называется *методом записи свойства* (`write`), метод получения значения свойства называется *методом чтения свойства* (`read`).

В описании класса перед именем свойства записывают слово `property` (свойство). После имени свойства указывается его тип, затем имена методов, обеспечивающих доступ к значению свойства. После слова `read` указывается имя метода, обеспечивающего чтение свойства, после слова `write` — записи свойства имя метода. Ниже приведен пример описания класса `TPerson`, содержащего два свойства: `Name` и `Address`.

```
type
   TName = string[15];
   TAddress = string[35];
   TPerson = class
      private
        fName:TName;          // имя
        fAddress:TAddress;    // адрес
        Constructor Create(Name:TName);
        Procedure Show;
        Function GetName:TName;
        Function GetAddress:TAddress;
        Procedure SetAddress(NewAddress:TAddress);
      public
        Property Name: TName
           read GetName;
        Property Address: TAddress
           read GetAddress
           write SetAddress;
   end;
```

В программе для установки значения свойства не обязательно записывать инструкцию применения к объекту метода установки значения свойства, можно записать обычную инструкцию присваивания значения свойству. Например, чтобы присвоить значение свойству `Address` объекта `student`, достаточно записать

```
student.Address:= 'С.Петербург, ул.Садовая 21, кв.3';
```

Компилятор перетранслирует приведенную инструкцию присваивания значения свойству в инструкцию вызова метода

```
student.SetAddress('С.Петербург, ул.Садовая 21, кв.3');
```

Внешне использование свойств в программе ничем не отличается от использования полей объекта. Вместе с тем между свойством и полем объекта существует принципиальное отличие: при присвоении и чтении значения свойства автоматически вызывается процедура, которая выполняет некоторую работу.

В программе на методы свойства можно возложить некоторые дополнительные задачи. Например, с помощью метода можно проверить корректность присваиваемых свойству значений, установить значения других, логически связанных со свойством, полей, вызвать вспомогательную процедуру.

Оформление данных объекта как свойства позволяет ограничить доступ к полям, хранящим значения свойств объекта, например, можно разрешить только чтение. Чтобы инструкции программы не могли изменить значение свойства, в описании свойства надо указать только имя метода чтения. Попытка присвоить значение свойству, предназначенному только для чтения, вызывает ошибку времени компиляции. В приведенном выше описании класса `TPerson` свойство `Name` доступно только для чтения, а свойство `Address` — для чтения и записи.

Установить значение свойства, защищенного от записи, можно во время инициализации объекта. Ниже приведены методы класса `TPerson`, обеспечивающие создание объекта класса `TPerson` и доступ к его свойствам.

```
// конструктор объекта TPerson
Constructor TPerson.Create(Name:TName);
begin
     fName := Name;
end;

// метод получения значения свойства Name
Function TPerson.GetName;
begin
     Result := fName;
end;
// метод получения значения свойства Address
function TPerson.GetAddress;
```

```
begin
     Result := fAddress;
end;

// метод изменения значения свойства Address
Procedure TPerson.SetAddress(NewAddress:TAddress);
begin
     if fAddress =''
        then fAddress:=NewAddress;
end;
```

Приведенный конструктор объекта `TPerson` создает объект и устанавливает значение поля `fName`, определяющего значение свойства `Name`.

Инструкции программы, обеспечивающие создание объекта класса `TPerson` и установку его свойства, могут быть, например, такими:

```
student := TPerson.create('Иванов');
student.address := 'ул. Садовая, д.3, кв.25';
```

Наследование

Концепция объектно-ориентированного программирования предполагает возможность определять новые классы посредством добавления полей, свойств и методов к уже существующим классам. Такой механизм получения новых классов называется *порождением*. При этом новый, порожденный, класс (потомок) наследует свойства и методы своего базового, родительского класса.

В объявлении класса-потомка указывается класс родителя. Например, класс `TEmployee` (сотрудник) может быть порожден от рассмотренного выше класса `TPerson` путем добавления поля `Department` (отдел). Объявление класса `TEmployee` в этом случае может выглядеть так:

```
TEmployee = class(TPerson)
      fDepartment:integer; // номер отдела
      constructor Create(Name:TName;Dep:integer);
end;
```

Заключенное в скобки имя класса `TPerson` показывает, что класс `TEmployee` является производным от класса `TPerson`. В свою очередь, класс `TPerson` является базовым для класса `TEmployee`.

Класс TEmployee имеет свой собственный конструктор, который обеспечивает инициализацию класса родителя и своих полей. Вот пример реализации конструктора класса TEmployee:

```
constructor TEmployee.Create(Name:Tname;Dep:integer);
begin
     inherited Create(Name);
     FDepartment:=Dep;
end;
```

В приведенном примере директивой inherited вызывается конструктор родительского класса, затем присваивается значение полю класса потомка.

После создания объекта производного класса в программе можно использовать поля и методы родительского класса. Ниже приведен фрагмент программы, демонстрирующей эту возможность:

```
engineer := TEmployee.create('Сидоров',413);
engineer.address := 'ул.Блохина, д.8, кв.10';
```

Первая инструкция создает объект типа TEmployee. Вторая устанавливает значение свойства, которое относится к родительскому классу.

Директивы *Protected* и *Private*

Помимо объявлений элементов класса (полей, методов, свойств), описание класса, как правило, содержит директивы Protected (защищенный) и Private (закрытый), которые устанавливают степень видимости элементов класса в программе.

Элементы класса, объявленные в секции Protected, доступны только в порожденных от него классах. Область видимости элементов класса этой секции не ограничивается модулем, в котором находится описание класса. Обычно в секцию Protected помещают описание методов класса.

Элементы класса, объявленные в секции Private, видимы внутри модуля. Эти элементы не доступны за пределами модуля, даже в производных классах. Обычно в секцию Private помещают описание полей класса, а методы, обеспечивающие доступ к этим полям, помещают в секцию Protected.

Ниже приведено описание класса TPerson, в которое включены директивы управления доступом:

```
TPerson = class
   private
      fName:TName;       { значение св-ва Name }
```

```
    fAddress:TAddress;   { значение св-ва Address}
  protected
    Constructor Create(Name:TName);
    Function GetName:TName;
    Function GetAddress:TAddress;
    Procedure SetAddress(NewAddress:TAddress);
    Property Name:TName
          read GetName;
    Property Address:TAddress
          read GetAddress
          write SetAddress;
  end;
```

Полиморфизм и виртуальные методы

Полиморфизм — это возможность использовать одинаковые имена для методов, входящих в различные классы. Концепция полиморфизма обеспечивает при применении метода к объекту использование именно того метода, который соответствует классу объекта.

Пусть определены три класса, один из них является базовым для двух других.

```
type
  // базовый класс
  TPerson = class
    fName:string;{ имя }
    constructor Create(name:string);
    function info: string; virtual;
  end;

  // производный от базового TPerson
  TStud = class(TPerson)
    fgr:integer; { номер группы }
    constructor Create(name:string;gr:integer);
    function info: string; override;
  end;

  // производный от базового TPerson
  TProf = class(TPerson)
    fdep:string; { название кафедры }
```

```
    constructor Create(name:string;dep:string);
    function info: string; override;
  end;
```

В каждом из этих классов определен метод `info`. В базовом классе при помощи директивы `virtual` метод `info` объявлен виртуальным. Объявление метода виртуальным дает возможность дочернему классу произвести замену виртуального метода своим собственным. В каждом дочернем классе определен свой метод `info`, который замещает соответствующий метод родительского класса (метод порожденного класса, замещающий виртуальный метод родительского класса, помечается директивой `override`). Ниже приведено определение метода `info` для каждого класса.

```
function TPerson.info:string;
begin
   result:='';
end;
function TStud.info:string;
begin
   result:=fname+' гр.'+IntTostr(fgr);
end;
function TProf.info:string;
begin
   result:=fname+' каф.'+fdep;
end;
```

В программе список людей можно представить массивом объектов класса `TPerson`.

```
list: array[1..SZL] of TPerson;
```

Объявить подобным образом список можно потому, что Object Pascal позволяет указателю на родительский класс присвоить значение указателя на дочерний класс. Поэтому элементами массива `list` могут быть как объекты класса `TStud`, так и объекты класса `TProf`.

Вывод списка можно осуществить применением метода `info` к элементам массива, например, так:

```
st:='';
for i:=1 to SZL do // SZL — размер массива-списка
   if list[i] <> NIL
       then st := st + info.list[i]+#13;
ShowMessage(st);
```

Во время работы программы каждый элемент массива может содержать как объект типа `TStud`, так и объект типа `TProf`. Концепция полиморфизма обеспечивает применение объекту именно того метода, который соответствует типу объекта.

Классы и объекты Delphi

Для реализации интерфейса Delphi использует библиотеку классов (она называется VCL — Visual Components Library), которая содержит большое количество разнообразных классов, поддерживающих форму и различные компоненты (командные кнопки, поля редактирования и т. д.).

Во время проектирования формы приложения Delphi автоматически добавляет в текст программы необходимые объекты. Если сразу после запуска Delphi просмотреть содержимое окна редактора кода, то там можно обнаружить следующие строки:

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1
```

Это описание класса исходной (пустой) формы приложения и объявление объекта — формы приложения.

Когда программист, добавляя необходимые компоненты, создает форму, Delphi формирует описание класса формы. Когда программист создает функцию обработки события формы или ее компонента, Delphi добавляет объявление метода в описание класса формы приложения.

Помимо классов визуальных компонентов, в библиотеку классов входит класс обработки исключительных ситуаций (ошибок), класс общего управления приложением и другие. Рассмотрение этих классов в задачу данной книги не входит.

Экзаменатор – пример программы

В первой части книги, посвященной программированию в Turbo Pascal, в качестве примера была рассмотрена программа контроля знаний. В этой главе

рассматривается подобная программа (рис. 15.43), но в отличие от предыдущей, она, во-первых, работает в Windows, во-вторых, позволяет к вопросу добавить иллюстрацию, что делает программу более привлекательной, расширяет диапазон ее возможного применения.

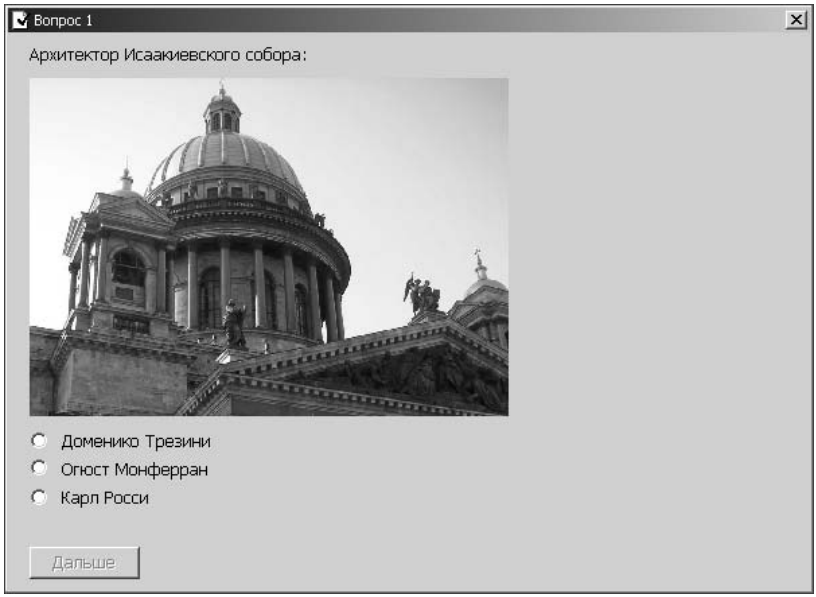


Рис. 15.43. Окно программы тестирования.
Задача испытуемого — выбрать правильный ответ

Файл теста

Универсальность программы тестирования обеспечивается за счет того, что вопросы теста находятся в файле, имя которого указывается в команде запуска программы. Помимо вопросов (здесь и далее, вопрос — это вопрос и несколько вариантов ответа), в файле теста находится информация, необходимая для выставления оценки.

Файл теста состоит из:

- заголовка;
- раздела оценок;
- раздела вопросов.

Заголовок состоит из двух абзацев: первый абзац — название теста (отображается в заголовке окна программы тестирования), второй — информация о тесте.

Вот пример заголовка:

```
История Санкт-Петербурга
Сейчас Вам будут предложены вопросы о памятниках и архитектурных
сооружениях Санкт-Петербурга. Вы должны из предложенных нескольких
вариантов ответа выбрать правильный.
```

За заголовком следует раздел оценок, в котором указывается количество баллов (количество правильных ответов), необходимое для достижения уровня, и сообщение, информирующее испытуемого о достижении уровня. В простейшем случае сообщение — это оценка. Для каждого уровня указывается количество правильных ответов и в следующей строке сообщение. Вот пример раздела оценок:

```
100
Отлично
85
Хорошо
60
Удовлетворительно
50
Плохо
```

За разделом оценок следует раздел вопросов теста.

Каждый вопрос начинается текстом вопроса. В следующей строке указывается количество альтернативных ответов, номер правильного ответа и признак наличия иллюстрации. Если вопрос сопровождается иллюстрацией, то значение признака должно быть равно единице, если нет — то нулю. В следующей строке, если значение признака наличия иллюстрации равно единице, указывается имя файла иллюстрации. Далее следуют альтернативные ответы, каждый из которых должен представлять собой абзац текста.

Вот пример вопроса:

```
Архитектор Зимнего дворца
3 1 1
herm.jpg
Бартоломео Растрелли
Карл Росси
Огюст Монферран
```

В приведенном примере к вопросу даны три варианта ответа, правильным является первый вариант ответа (архитектор Зимнего дворца — Бартоломео Растрелли). К вопросу есть иллюстрация (третье число во второй строке — единица), которая находится в файле herm.jpg.

В листинге 15.6 в качестве примера приведен текст файла вопросов для проверки знания истории памятников и архитектурных сооружений Санкт-Петербурга.

Листинг 15.6. Файл вопросов

```
История Санкт-Петербурга
Сейчас Вам будут предложены вопросы о знаменитых памятниках
и архитектурных сооружениях Санкт-Петербурга. Вы должны из предложенных
нескольких вариантов ответа выбрать правильный.
7
Вы прекрасно знаете историю Санкт-Петербурга!
6
Вы много знаете о Санкт-Петербурге, но на некоторые вопросы ответили
неверно.
5
Вы недостаточно хорошо знаете историю Санкт-Петербурга.
4
Вы, вероятно, только начали знакомиться с историей Санкт-Петербурга?
Архитектор Исаакиевского собора:
3 2 1
is.jpg
Доменико Трезини
Огюст Монферран
Карл Росси
Александровская колонна воздвигнута в 1836 году по проекту
Огюста Монферрана как памятник, посвященный:
2 1 0
деяниям императора Александра I.
подвигу народа в Отечественной войне 1812 года.
Архитектор Зимнего дворца
3 1 1
herm.jpg
Бартоломео Растрелли
Карл Росси
Огюст Монферран
Михайловский (Инженерный) замок — жемчужина архитектуры Петербурга
построен по проекту
3 1 0
Винченцо Бренна
Старова Ивана Егоровича
Баженова Василия Ивановича
```

```
Остров, на котором находится Ботанический сад, основанный императором
Петром I, называется:
3 3 1
bot.jpg
Заячий
Медицинский
Аптекарский
Невский проспект получил свое название
3 2 0
по имени реки, на которой стоит Санкт-Петербург.
по имени близко расположенного монастыря, Александро-Невской лавры.
в память о знаменитом полководце — Александре Невском.
Скульптура памятника Петру I "Медный всадник" выполнена
2 1 0
Фальконе
Клодтом
```

Файл теста можно подготовить в текстовом редакторе, который сохраняет "чистый" (без символов форматирования) текст, например, в Блокноте. В момент записи текста на диск вместо стандартного расширения txt следует указать какое-либо другое, например, exm (от Examiner — экзаменатор). Такое решение позволит настроить операционную систему так, что тест будет запускаться автоматически, в результате щелчка на имени файла теста.

Форма приложения

Форма программы тестирования приведена на рис. 15.44.

Вопрос отображается в поле `Label5`, варианты ответа — в полях `Label1`—`Label4`. Переключатели `RadioButton1`—`RadioButton4` обеспечивают выбор ответа, а `RadioButton5` (во время работы программы он не отображается) — сброс переключателей выбора ответа. Кнопка `Button1` (она доступна только в том случае, если ответ выбран) обеспечивает переход к следующему вопросу. Следует обратить внимание, что текст на кнопке `Button1` во время работы программы меняется. В начале и в конце работы программы на кнопке находится текст **Ok**, а в процессе тестирования — **Дальше**. Если вопрос сопровождается иллюстрацией, то она отображается в поле компонента `Image1`.

В табл. 15.12 приведены значения свойств формы, в табл. 15.13 — значения свойств компонентов.

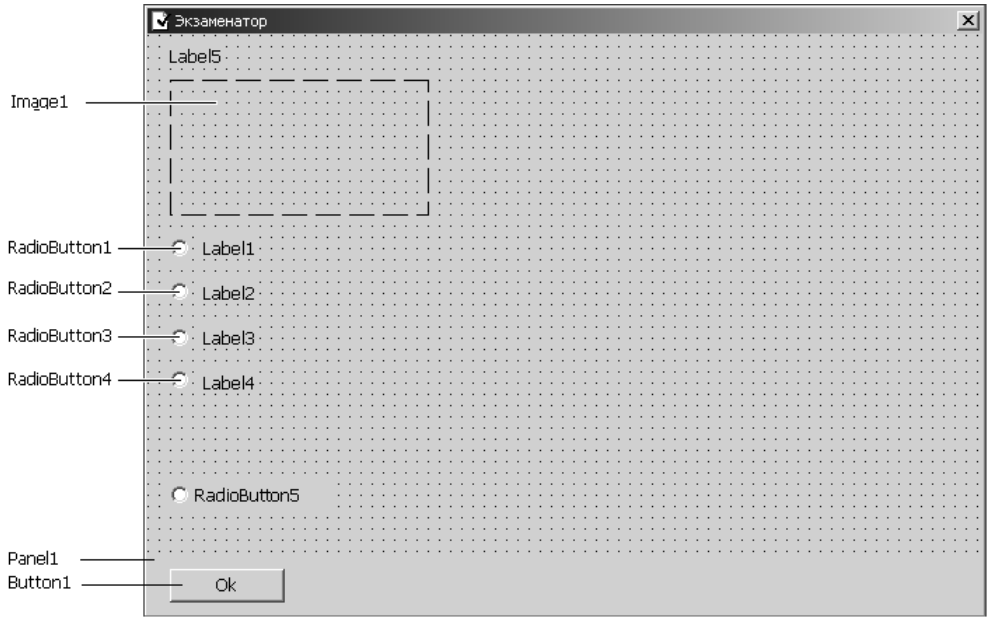


Рис. 15.44. Форма программы Экзаменатор

Таблица 15.12. *Значения свойств формы*

Свойство	Значение
BorderIcons.biMinimize	False
BorderIcons.biMaximize	False
BorderStyle	bsSingle

Таблица 15.13. *Значения свойств компонентов*

Компонент	Свойство	Значение
Label1	WordWrap	True
	AutoSize	True
Label2	WordWrap	True
	AutoSize	True
Label3	WordWrap	True
	AutoSize	True

Таблица 15.13 *(окончание)*

Компонент	Свойство	Значение
Label4	WordWrap	True
	AutoSize	True
Label5	WordWrap	True
	AutoSize	True
Image1	Proportional	True
Panel1	Align	alBottom
	BevelOuter	bvNone
	Height	41
RadioButton5	Visible	False

Следует обратить внимание, что значения свойств, определяющих положение компонентов, предназначенных для отображения вопроса, альтернативных ответов и иллюстрации, а также выбора ответа, вычисляются во время работы программы, после того как будет прочитан очередной вопрос, поэтому во время создания формы их можно поместить в любую точку формы. Положение компонента `Label1` отсчитывается от нижней границы компонента `Label5` или, если вопрос сопровождается иллюстрацией, от нижней границы компонента `Image1`. Положение компонента `Label2` отсчитывается от нижней границы компонента `Label1`. Аналогичным образом вычисляется положение компонентов `Label3` и `Label4`.

Отображение иллюстрации

Отображение иллюстраций в окне программы тестирования обеспечивает компонент `Image`. Значок компонента находится на вкладке **Additional** палитры компонентов (рис. 15.45). Свойства компонента `Image` приведены в табл. 15.14.

Рис. 15.45. Значок компонента Image

Таблица 15.14. *Свойства компонента* `Image`

Свойство	Комментарий
`Name`	Имя компонента
`Picture`	Свойство, являющееся объектом типа `Tbitmap`. Определяет выводимую картинку
`Left`	Расстояние от левого края формы до левой границы области картинки
`Top`	Расстояние от верхней границы формы до верхней границы области картинки
`Height`	Высота картинки
`Width`	Ширина картинки
`Proportional`	Признак автоматического изменения размера картинки таким образом, чтобы она без искажения занимала максимально возможную часть области отображения иллюстрации
`AutoSize`	Признак автоматического изменения высоты и ширины области вывода картинки (размера компонента) в зависимости от реального размера картинки: если значение свойства `AutoSize` равно `True`, то при установке значения свойства `Picture` автоматически меняется размер области таким образом, чтобы была видна вся картинка. Если значение свойства `AutoSize` равно `False`, а размер картинки превышает размер области, то отображается только часть картинки
`Stretch`	Признак автоматического сжатия или растяжения картинки таким образом, чтобы она была видна полностью в области с заданными свойствами `Width` и `Height` размерами

Картинку, отображаемую в области компонента `Image`, можно задать во время создания формы или во время работы программы. Во время создания формы картинка задается установкой значения свойства `Picture`. Во время работы программы — применением метода `LoadFromFile`. Например, для разрабатываемого приложения инструкция вывода иллюстрации, находящейся в файле is.jpg (изображение Исаакиевского собора), может быть такой:

```
Image1.Picture.LoadFromFile('is.jpg')
```

Следует обратить внимание, для того чтобы программа могла отобразить jpg-иллюстрацию, в директиву `uses` модуля формы надо добавить ссылку на модуль jpeg.

Очевидно, что размер области формы, которая может использоваться для вывода иллюстрации, зависит от длины (количества слов) вопроса, длины

и количества альтернативных ответов. Чем длиннее вопрос и ответы, тем больше места в поле формы они занимают, и тем меньше места остается для иллюстрации. В рассматриваемой программе положение и размер области отображения иллюстрации (компонента `Image1`) вычисляются после загрузки в компоненты `Label` вопроса и альтернативных ответов.

Выбор ответа

Выбор правильного ответа осуществляется при помощи щелчка на одном из компонентов `RadioButton`. Компонент `RadioButton` (его значок находится на вкладке **Standard**, рис. 15.46) представляет собой переключатель (радиокнопку), состояние которого зависит от состояния других радиокнопок, находящихся на форме. Так как только одна из радиокнопок, находящихся на форме, может быть выбрана, то щелчок на невыбранном переключателе автоматически сбрасывает выбранный переключатель и переводит в выбранное состояние переключатель, на котором сделан щелчок. Свойства компонента `RadioButton` приведены в табл. 15.15.

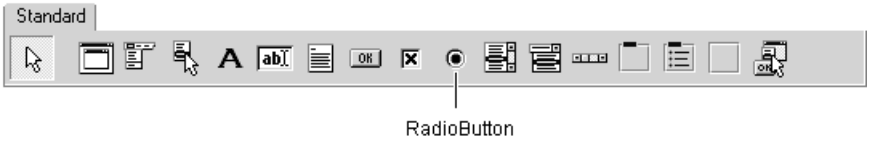


Рис. 15.46. Значок компонента RadioButton

Таблица 15.15. *Свойства компонента* `RadioButton`

Свойство	Описание
`Name`	Имя (идентификатор) компонента
`Caption`	Текст, который отображается справа от кнопки
`Checked`	Состояние. Определяет вид кнопки: `True` — кнопка выбрана, `False` — кнопка не выбрана (выбрана другая кнопка группы)
`Left`	Расстояние от левой границы флажка до левой границы формы
`Top`	Расстояние от верхней границы флажка до верхней границы формы
`Height`	Высота поля вывода поясняющего текста
`Width`	Ширина поля вывода поясняющего текста
`Font`	Шрифт, используемый для отображения поясняющего текста

В программе тестирования переключатели RadioButton1 — RadioButton4 используются для выбора ответа, переключатель RadioButton5 — для сброса переключателей выбора ответа перед выводом очередного вопроса.

Доступ к файлу теста

Для того чтобы программа Экзаменатор была действительно универсальной, у пользователя должна быть возможность задать имя файла теста.

Обеспечить возможность настройки программы на работу с конкретным файлом можно несколькими способами. Например, программа может считать имя файла теста из файла конфигурации (INI-файла) или получить его из командной строки.

Командная строка — это строка, которую пользователь должен набрать в окне **Запуск программы**, для того чтобы запустить программу. В простейшем случае командная строка — это имя EXE-файла. В командной строке после имени EXE-файла можно указать дополнительную информацию, которую надо передать программе, например, имя файла, с которым она должна работать.

Программа может получить параметр, указанный в командной строке, как значение функции ParamStr(n), где n — порядковый номер параметра. Количество параметров командной строки находится в глобальной переменной ParamCount. Следует обратить внимание, что значение ParamStr(0) — это полное имя EXE-файла программы.

Ниже приведен фрагмент кода, который демонстрирует использование функции ParamStr:

```
if ParamCount = 0
     then begin
             ShowMessage('Ошибка! Не задан файл вопросов теста.');
             Exit;
     end;
fn := ParamStr(1); // имя файла – параметр команды запуска программы
```

Как было сказано раньше, если программе нужны параметры, то они указываются в команде ее запуска после имени EXE-файла. При запуске программы из среды разработки, если программе необходимы параметры, их надо указать в поле **Parameters** окна **Run Parameters** (рис. 15.47), которое становится доступным в результате выбора в меню **Run** команды **Parameters**.

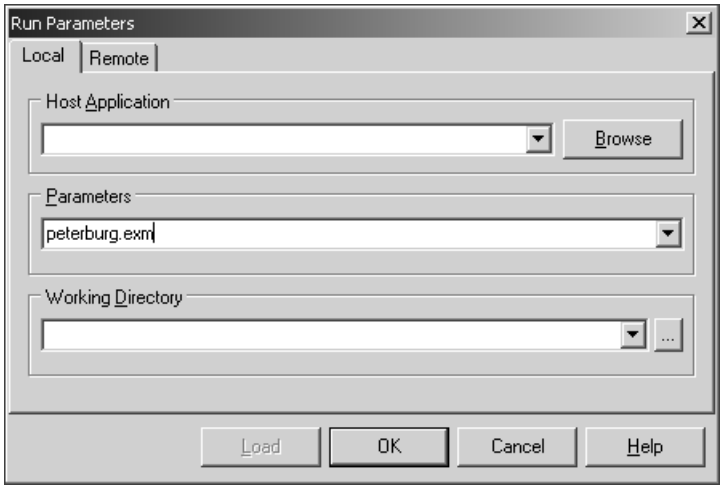


Рис. 15.47. Параметр командной строки надо ввести в поле **Parameters**

Текст программы

После настройки формы и компонентов можно приступить к набору текста программы (листинг 15.7). Сначала в раздел `private` объявления формы надо поместить объявления вспомогательных процедур и функций, а в раздел `var` секции `implementation` — объявления переменных. После этого следует набрать процедуры и функции, а затем создать процедуры обработки событий. Следует обратить внимание, что событие `Click` на компонентах `RadioButton1`—`RadioButton4` обрабатывает одна процедура.

Листинг 15.7. Модуль главной формы программы Экзаменатор

```
{
  Универсальная программа тестирования.
  (c) Культин Н.Б.

  Тест загружается из файла, имя которого указано
  в команде запуска программы.
}

unit ExamMainForm;

interface

uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
```

```
  Forms, Dialogs, StdCtrls, ExtCtrls,

  jpeg; // обеспечивает отображение JPG-иллюстраций

type
  TForm1 = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    RadioButton1: TRadioButton;
    RadioButton2: TRadioButton;
    RadioButton3: TRadioButton;
    RadioButton4: TRadioButton;
    RadioButton5: TRadioButton;
    Image1: TImage;
    Button1: TButton;
    Panel1: TPanel;

    procedure FormActivate(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure RadioButtonClick(Sender: TObject);

  private
    // Эти объявления вставлены сюда вручную
    procedure Info;
    function VoprosToScr: boolean;
    procedure ShowPicture; // выводит иллюстрацию
    procedure ResetForm;  // "очистка" формы перед выводом вопроса
    procedure Itog;        // результат тестирования
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.DFM}
```

```
var
  f:TextFile;
  fn:string; // имя файла вопросов

  level:array[1..4] of integer; // сумма, соответствующая уровню
  mes:array[1..4] of string;    // сообщение, соответствующее уровню

  vopr: record
    text: string;  // вопрос
    src: string;   // иллюстрация
    otv: array[1..4] of string; // варианты ответа
    notv: integer;  // количество вариантов ответа
    right: integer; // номер правильного ответа
  end;

  cv: integer = 0; // номер вопроса
  otv : integer;   // номер выбранного ответа
  sum: integer;

// информации о тесте
procedure Tform1.Info;
var
   s: string;
begin
   readln(f,s);
   Form1.Caption := s;
   readln(f,s);
   Label5.caption:=s;
  end;

// прочитать из файла информацию об оценках
Procedure GetLevel;
var
   i:integer;
begin
     for i:=1 to 4 do
     begin
         readln(f,level[i]); // оценка
```

```
        readln(f,mes[i]);  // сообщение

    end;
end;

// отображение иллюстрации
Procedure TForm1.ShowPicture;
var
  w,h: integer; // максимально возможные размеры картинки
begin
  // вычислить допустимые размеры картинки
  w:=ClientWidth-10;
  h:=ClientHeight
       - Panel1.Height -10
       - Label5.Top
       - Label5.Height - 10;

  // вопросы
  if Label1.Caption <> ''
     then h:=h-Label1.Height-10;
  if Label2.Caption <> ''
     then h:=h-Label2.Height-10;
  if Label3.Caption <> ''
     then h:=h-Label3.Height-10;
  if Label4.Caption <> ''
     then h:=h-Label4.Height-10;

  // если размер картинки меньше w на h,
  // то она не масштабируется
  Image1.Top:=Form1.Label5.Top+Label5.Height+10;
  if Image1.Picture.Height > h
       then Image1.Height:=h
       else Image1.Height:= Image1.Picture.Height;
  if Image1.Picture.Width > w
       then Image1.Width:=w
          else Image1.Width:=Image1.Picture.Width;

  Image1.Visible := True;
end;
```

```
// вывести вопрос
function TForm1.VoprosToScr: boolean;
  var
   i: integer;
   p: integer;
  begin

     if EOF(f) then
     begin
       // достигнут конец файла
       VoprosToScr := False;
       exit;
     end;

     readln(f, vopr.text); // прочитать вопрос

     if  vopr.text = '*' then
     begin
       // признак конца теста
       VoprosToScr := False;
       exit;
     end;

     cv := cv + 1;
     caption:='Вопрос ' + IntToStr(cv);
     Label5.caption:= vopr.text;  // вывести вопрос

     // прочитать информацию об ответе:
     // количество вариантов, номер правильного ответа и признак наличия
     // иллюстрации
     readln(f,vopr.notv,vopr.right, p); // p — признак иллюстрации

     if p <> 0 then // есть иллюстрация
     begin
        readln(f,vopr.src);  // имя файла иллюстрации
        Image1.Tag:=1;
          try
            Image1.Picture.LoadFromFile(vopr.src);
            except
```

```
            on E:EFOpenError do
               Image1.Tag:=0;
        end
    end
    else Image1.Tag := 0; // нет иллюстрации

    // читаем варианты ответа
    for i:= 1 to vopr.notv do
       readln(f,vopr.otv[i]);

    for i:= 1 to vopr.notv do
       case i of
           1: Label1.caption:= vopr.otv[i];
           2: Label2.caption:= vopr.otv[i];
           3: Label3.caption:= vopr.otv[i];
           4: Label4.caption:= vopr.otv[i];
       end;

    // здесь прочитана иллюстрация и альтернативные ответы

    // текст вопроса уже выведен
    if Image1.Tag =1  // есть иллюстрация к вопросу
       then ShowPicture;

    // вывод альтернативных ответов
    if Label1.Caption <> ''
    then begin
       if Image1.Tag =1
           then Label1.top:=Image1.Top+Image1.Height+10
           else Label1.top:=Label5.Top+Label5.Height+10;
       RadioButton1.top:=Label1.top;
       Label1.visible:=TRUE;
       RadioButton1.Visible:=TRUE;
    end;

    if Label2.Caption <> ''
    then begin
       Label2.top:=Label1.top+ Label1.height+5;
       RadioButton2.top:=Label2.top;
       Label2.visible:=TRUE;
```

```
      RadioButton2.Visible:=TRUE;
    end;

    if Label3.Caption <> ''
    then begin
      Label3.top:=Label2.top+ Label2.height+5;
      RadioButton3.top:=Label3.top;
      Label3.visible:=TRUE;
      RadioButton3.Visible:=TRUE;
    end;

    if Label4.Caption <> ''
    then begin
      Label4.top:=Label3.top+ Label3.height+5;
      RadioButton4.top:=Label4.top;
      Label4.visible:=TRUE;
      RadioButton4.Visible:=TRUE;
    end;

    Label6.Caption := '';
    VoprosToScr := True;
  end;

Procedure TForm1.ResetForm;
  begin
    // сделать невидимыми все метки и переключатели
    Label1.Visible:=FALSE;
    Label1.caption:='';
    Label1.width:=ClientWidth-Label1.left-5;
    RadioButton1.Visible:=FALSE;

    Label2.Visible:=FALSE;
    Label2.caption:='';
    Label2.width:=ClientWidth-Label2.left-5;
    RadioButton2.Visible:=FALSE;

    Label3.Visible:=FALSE;
    Label3.caption:='';
    Label3.width:=ClientWidth-Label3.left-5;
    RadioButton3.Visible:=FALSE;

    Label4.Visible:=FALSE;
```

```
    Label4.caption:='';
    Label4.width:=ClientWidth-Label4.left-5;
    RadioButton4.Visible:=FALSE;

    Label5.width:=ClientWidth-Label5.left-5;

    Image1.Visible:=FALSE;
  end;

// определение достигнутого уровня
procedure TForm1.Itog;
  var
   i:integer;
   buf:string;
  begin
   buf:='';
   buf:='Результаты тестирования'+ #13 +
        'Всего вопросов: '+ IntToStr(cv) + #13 +
        'Правильных ответов: ' + IntToStr(sum);
   i:=1;
   while (sum < level[i]) and (i<4) do
         i:=i+1;
   buf:=buf+ #13+ mes[i];
   Label5.Top:= 20;
   Label5.caption:=buf;
  end;

procedure TForm1.FormActivate(Sender: TObject);
  begin
     if ParamCount = 0
     then begin
             Label5.caption:= 'Не задан файл вопросов теста.';
             Button1.caption:='Ok';
             Button1.tag:=2;
             Button1.Enabled:=TRUE
          end
     else begin
          fn := ParamStr(1);
          assignfile(f,fn);
          try
```

```
            reset(f);
          except
            on EFOpenError do
               begin
                    ShowMessage('Файл теста '+fn+' не найден.');
                    Button1.caption:='Ok';
                    Button1.tag:=2;
                    Button1.Enabled:=TRUE;
                    exit;
               end;
             end;

          // ключ /t активизирует режим отладки/обучения
          if ParamStr(2) = '/t' then
              Label6.Visible := True;

          ResetForm;
          Info;       // прочитать и вывести информацию о тесте
          GetLevel;   // прочитать информацию об уровнях оценок
     end;
end;

// щелчок на кнопке Button1
procedure TForm1.Button1Click(Sender: TObject);
  begin
     case Button1.tag of
       0: begin
            Button1.caption:='Дальше';
            Button1.tag:=1;
            RadioButton5.Checked:=TRUE;
            // вывод первого вопроса
            Button1.Enabled:=False;
            ResetForm;
            VoprosToScr;
          end;
       1: begin // вывод остальных вопросов
            if otv = vopr.right then
               sum := sum + 1;
            RadioButton5.Checked:=TRUE;
            Button1.Enabled:=False;
```

```
                ResetForm;
                if not VoprosToScr then
                      begin
                         closefile(f);
                         Button1.caption:='Ok';
                         Form1.caption:='Результат';
                         Button1.tag:=2;
                         Button1.Enabled:=TRUE;
                         Label6.Visible := False;
                         Itog; // вывести результат
                      end;
          end;
        2: begin // завершение работы
             Form1.Close;
          end;
      end;
   end;

// Процедура обработки события Click
// для компонентов RadioButton1-RadioButton4
procedure TForm1.RadioButtonClick(Sender: TObject);
begin
  if sender = RadioButton1
    then otv:=1
          else  if sender = RadioButton2
                then otv:=2
                else   if sender = RadioButton3
                             then otv:=3
                             else otv:=4;
  Button1.enabled:=TRUE;
  Label6.Caption := 'Выбран ответ: ' + IntToStr(otv) +
           ' Правильный ответ: ' + IntToStr(vopr.right);
end;

end.
```

Работает программа Экзаменатор так. После запуска программы процедура обработки события `FormActivate` проверяет, указан ли при запуске программы параметр — имя файла теста. Если параметр не указан (в этом случае значение функции `ParamCount` равно нулю), в поле `Label5` выводится сооб-

щение об ошибке, и после щелчка на кнопке **Ok** программа завершает работу. Если параметр указан, то имя файла теста (значение функции `ParamStr(1)`) записывается в переменную `fn` и делается попытка открыть файл. Если файл открыт, вызывается процедура `Info`, которая считывает из файла теста информацию о тесте и выводит ее в поле метки `Label5`. Затем вызывается процедура `GetLevel`, которая считывает из файла теста информацию об уровнях оценки и записывает ее в массивы `level` и `mes`. Следует обратить внимание, что в программе реализован режим проверки файла теста: если в командной строке указан второй параметр и этот параметр — строка `/t`, то свойству `Visible` компонента `Label6` присваивается значение `True`. В результате во время работы программы в поле компонента `Label6` после выбора варианта ответа в нижней части окна отображается номер правильного ответа. После того как испытуемый прочтет информацию о тесте и сделает щелчок на кнопке **Ok**, начинается процесс тестирования.

Следует обратить внимание, что кнопка `Button1` используется для завершения работы программы (если не указан файл теста), активизации процесса тестирования, перехода к следующему вопросу (после выбора варианта ответа) и завершения работы программы. Какое из перечисленных действий будет выполнено в результате щелчка на кнопке `Button1`, определяет значение свойства `Tag` этой кнопки (см. процедуру обработки события `Click`). В начале работы программы значение свойства `Tag` кнопки `Button1` равно нулю. Поэтому в результате первого щелчка на кнопке выполняется та часть программы, которая обеспечивает отображение первого вопроса и записывает в свойство `Tag` единицу. В процессе тестирования значение свойства `Tag` равно единице. Поэтому процедура обработки события `Click` увеличивает на единицу счетчик правильных ответов (если выбран правильный ответ) и вызывает функцию `VoprosToScr`, которая считывает из файла очередной вопрос и выводит его на форму. Если текущий вопрос последний (в этом случае значение функции равно `False`), то вызывается процедура `Itog`, которая выводит результат тестирования, и в свойство `Tag` кнопки `Button1` записывается двойка. В результате, после следующего щелчка на кнопке `Button1`, программа завершает работу.

Отображение вопроса выполняет функция `VoprosToScr`. Сначала она делает попытку прочитать из файла очередной вопрос. Если прочитанная строка — "звездочка", то это значит, что достигнут конец теста (как показал опыт, необходимо явно указать признак конца теста). Если попытка прочитать была успешна, из файла теста считывается информация о количестве вариантов ответа, номер правильного ответа и признак наличия иллюстрации. Далее считывается имя файла иллюстрации (если признак наличия иллюстрации равен единице) и варианты ответа. Следует обратить внимание, что хотя

вопросы считываются непосредственно в свойства `Caption` компонентов `Label`, на форме они сразу после прочтения не отображаются (значение свойства `Visible` компонентов `Label1` перед чтением очередного вопроса устанавливается равным `False`). После этого, если к вопросу есть иллюстрация, вызывается процедура `ShowPicture`, которая выводит иллюстрации. Затем выводятся варианты ответа. Необходимо обратить внимание, что положение области отображения первого ответа (компонента `Label1`) отсчитывается от нижней границы области отображения иллюстрации (компонента `Image1`) или от нижней границы области отображения вопроса (компонента `Label5`). Процедура `ShowPicture` выводит иллюстрацию, но сначала на основе информации о размере компонентов `Label1`—`Label5` (значение свойства `AutoSize` равно `True`, поэтому размер компонента определяется текстом, который загружен в свойство `Caption`), она вычисляет размер области, которую можно использовать для отображения иллюстрации.

Процедура `ResetForm` путем выбора невидимого переключателя `RadioButton5` сбрасывает переключатели выбора ответа и делает недоступной кнопку **Дальше** (`Button1`).

Обработку события `Click` на переключателях `RadioButton1`—`RadioButton4` выполняет одна, общая для всех компонентов, процедура `TForm1.RadioButtonClick`. Параметр `Sender` этой процедуры позволяет определить, на каком объекте произошло событие. Процедура фиксирует в переменной `otv` номер переключателя и соответственно номер выбранного ответа. Также она делает доступной кнопку перехода к следующему вопросу.

Процедура `Itog`, сравнивая набранную сумму баллов `summa` со значением элементов массива `level`, определяет, какого уровня достиг испытуемый, и выводит соответствующее сообщение с присвоением значения свойству `Label5.Caption`.

Запуск программы

Программа Экзаменатор получает имя файла теста из командной строки. Поэтому, чтобы "запустить тест", пользователь должен в окне **Запуск программы** набрать имя программы тестирования и указать файл теста. Это не совсем удобно. Чтобы облегчить жизнь пользователю, можно настроить операционную систему так, что программа тестирования будет запускаться в результате щелчка на значке файла теста. Настройка выполняется следующим образом. Сначала надо раскрыть папку, в которой находится файл теста (EXM-файл), и сделать двойной щелчок на значке файла. Так как расширение exm не является стандартным, то система не знает, какую программу надо запустить, чтобы открыть EXM-файл. Поэтому она предложит указать

программу, с помощью которой надо открыть файл — на экране появится окно **Выбор программы**. В этом окне нужно сделать щелчок на кнопке **Обзор**, раскрыть папку, в которой находится программа Экзаменатор, и выбрать EXE-файл. После этого надо установить флажок **Использовать ее для всех файлов такого типа**. Вид окна **Выбор программы** после выполнения всех перечисленных действий приведен на рис. 15.48.

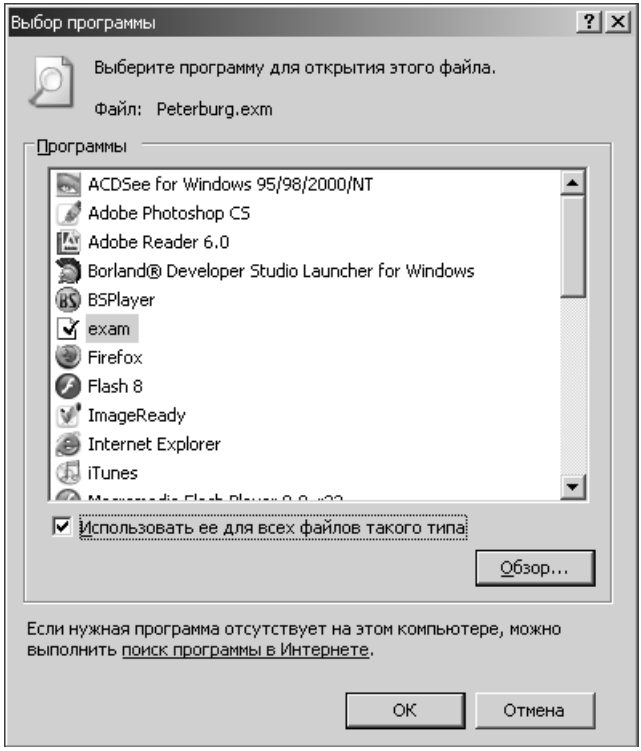
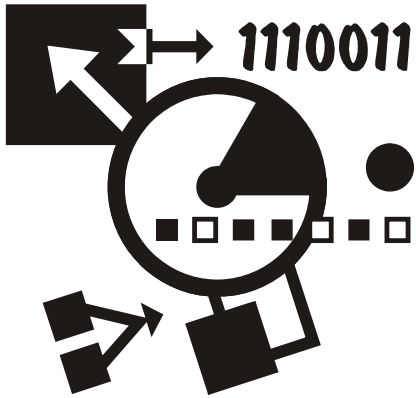
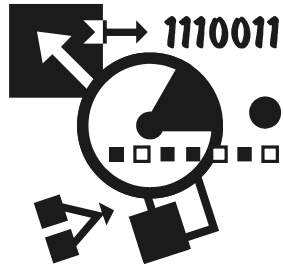


Рис. 15.48. Теперь файлы с расширением exm будет открывать Экзаменатор

В результате описанных действий в реестр операционной системы будет внесена информация о том, что файлы с расширением exm надо открывать с помощью программы Экзаменатор (имя файла, на котором сделан щелчок передается программе как параметр). Следует обратить внимание, что задачу настройки операционной системы можно возложить на программу, обеспечивающую ее установку.



ПРИЛОЖЕНИЯ



Приложение 1

Turbo Pascal — краткий справочник

Зарезервированные слова и директивы

Зарезервированные слова Turbo Pascal

and	file	not	then
array	for	object	to
asm	function	of	type
begin	goto	or	unit
case	if	packed	until
const	implementation	procedure	uses
constructor	in	program	var
destructor	inherited	record	while
div	inline	repeat	with
do	intenface	set	xor
downto	label	shl	
else	mod	shr	
end	nil	string	

Директивы Turbo Pascal

absolute	far	near	virtual
assembler	forward	private	
external	interrupt	public	

Структура программы

Программа на языке Pascal состоит из следующих разделов:

- ❒ Заголовок;
- ❒ Объявление используемых модулей;

- ❒ Объявление меток;
- ❒ Объявление констант;
- ❒ Объявление типов;
- ❒ Объявление процедур и функций;
- ❒ Объявление переменных;
- ❒ Инструкции.

Структура программы в общем виде:

```
program ИмяПрограммы;
uses
   {объявление используемых модулей}
label
   { объявление меток }
const
   { объявление констант }
type
   { объявление типов }

{ объявление процедур и функций программиста }
var
   { объявление переменных }
begin
   { инструкции программы}
end.
```

Основные типы данных

- ❒ целые числа (`integer` и др.);
- ❒ действительные числа (`real` и др.);
- ❒ символы (`char`);
- ❒ строки (`string`);
- ❒ логический (`boolean`).

Целые числа и числа с плавающей точкой могут быть представлены в различных форматах.

Целые числа

Формат	Диапазон значений
`shortint`	-128...127
`integer`	-32 768..32 767
`longint`	-2 147 483 648...2 147 483 647
`byte`	0...255
`word`	0...65 535

Действительные числа

Формат	Диапазон значений	Количество значащих цифр
`real`	$2.9 \cdot 10^{-39}$ ... $1.7 \cdot 10^{38}$	11—12
`single`	$1.5 \cdot 10^{-45}$ ... $3.4 \cdot 10^{38}$	7—8
`double`	$5.0 \cdot 10^{-324}$ ... $1.7 \cdot 10^{308}$	15—16
`extended`	$3.4 \cdot 10^{-4932}$ ... $1.1 \cdot 10^{4932}$	19—20

Строки

Строку можно объявить двумя способами.

Вариант 1

```
Имя: string;
```

Объявлена переменная-строка длиной 256 символов.

Вариант 2

```
Имя: string[Длина];
```

Объявлена переменная-строка указанной длины.

Массивы

Объявление одномерного массива:

```
Имя:array[НижнийИндекс .. ВерхнийИндекс] of ТипЭлементов;
```

Объявление двумерного массива:

```
Имя:array[НижнийИндекс1 .. ВерхнийИндекс1,
          НижнийИндекс2 .. ВерхнийИндекс2 ] of ТипЭлементов;
```

Записи

Запись можно объявить двумя способами.

Вариант 1

```
      Имя: record
             Поле1:Тип1;
             Поле2:Тип2;
               ...
             Полеi:Типi
      end;
```

Вариант 2

```
type
   Тип = record
             Поле1:Тип1;
             Поле2:Тип2;
               ...
             Полеi:Типi;
   end;
var
      Имя: Тип;
```

Сначала объявляется тип записи, затем — переменная-запись.

Инструкция *IF*

Вариант 1

```
if Условие
   then
      begin
        { Эти инструкции выполняются, }
        { если Условие истинно  }
      end
   else
      begin
       { Эти инструкции выполняются, }
       { если Условие ложно  }
      end;
```

Вариант 2

```
if Условие
   then
      begin
        { Эти инструкции выполняются, }
        { если Условие истинно }
      end;
```

Инструкция *CASE*

Вариант 1

```
case Выражение of
      Список_констант1: begin
                          { Инструкции 1 }
                        end;
      Список_констант2: begin
                          { Инструкции 2 }
                        end;
                        ...
      Список_константJ: begin
                          { Инструкции J }
                        end;
end;
```

Вариант 2

```
case Выражение of
      Список_констант1: begin
                          { Инструкции 1}
                        end;
      Список_констант2: begin
                          { Инструкции 2}
                        end;
                        ...
      Список_константJ: begin
                          { Инструкции J}
                        end
                else    begin
                          { Инструкции }
                        end;
end;
```

Инструкции между **begin** и **end** выполняются, если значение выражения, указанного после **case**, совпадает с константой из соответствующего списка. Если это не так, то выполняются инструкции между **begin** и **end**, расположенными после **else**.

Циклы

Инструкция *FOR*

Вариант 1 (с увеличением счетчика циклов)

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

Инструкции между **begin** и **end** выполняются

[(*КонечноеЗначение* - *НачальноеЗначение*) + 1] раз.

Если *НачальноеЗначение* больше, чем *КонечноеЗначение*, то инструкции между **begin** и **end** не выполняются.

Вариант 2 (с уменьшением счетчика циклов)

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

Инструкции между **begin** и **end** выполняются

[(НачальноеЗначение — КонечноеЗначение) + 1] раз.

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

Инструкция *REPEAT*

```
repeat
    begin
      { Инструкции }
    end
until Условие;
```

Сначала выполняются инструкции, которые находятся между **begin** и **end**, затем проверяется значение выражения *Условие*. Если условие не выполняется

(значение выражения *Условие* равно `False`), то снова выполняются инструкции цикла. И так до тех пор, пока условие не станет истинным.

Инструкция *WHILE*

```
while Условие do
   begin
      { Инструкции }
   end;
```

Сначала проверяется значение выражения *Условие*. Если оно истинно, то выполняются инструкции, находящиеся между **begin** и **end**. И так до тех пор, пока условие не станет ложным.

Объявление функции

```
function Имя(var Параметр1:Тип1; … var Параметрi:Типi):Тип;
const
      { Описание констант }
var
      { Описание локальных переменных }
begin
      { Инструкции функции }

      Имя := Значение;
end;
```

Объявление процедуры

```
procedure Имя(var Параметр1:Тип1; … Параметрi:Типi);
const
      { Описание констант }
var
      { Описание локальных переменных}
begin
      { Инструкции процедуры  }
end;
```

Процедуры и функции

При описании функций и процедур приняты следующие обозначения:

- Параметры выделены курсивом. В качестве параметров могут использоваться константы, переменные или выражения соответствующих типов. Если параметром обязательно должна быть переменная основной программы, то перед ним поставлено слово **var**. После параметра указывается его тип.
- Необязательные параметры указаны в квадратных скобках.
- После списка параметров функции через двоеточие указан тип результата, возвращаемого функцией.

Математические

Abs

Объявление:

```
function Abs(X);
```

Действие:

Возвращает абсолютное значение аргумента, в качестве которого можно использовать выражение целого или вещественного типа.

Arctan

Объявление:

```
function Arctan(X: real): real;
```

Действие:

Возвращает арктангенс аргумента — угла, величина которого выражена в радианах.

Cos

Объявление:

```
function Cos(X: real): real;
```

Действие:

Возвращает косинус аргумента — угла, величина которого выражена в радианах.

Exp

Объявление:

```
function Exp(X: real): real;
```

Действие:

Возвращает значение, равное экспоненте аргумента.

Ln

Объявление:

```
function Ln(X: real): real;
```

Возвращает значение, равное натуральному логарифму аргумента.

Sin

Объявление:

```
function Sin(X: real): real;
```

Действие:

Возвращает синус аргумента — угла, величина которого выражена в радианах.

Sqr

Объявление:

```
function Sqr(X);
```

Действие:

Возвращает квадрат аргумента, в качестве которого можно использовать выражение целого или вещественного типа.

Sqrt

Объявление:

```
function Sqrt(X: real): real;
```

Действие:

Возвращает значение, равное квадратному корню из аргумента.

Random

Объявление:

```
function Random[(Диапазон: word)];
```

Действие:

Если параметр *Диапазон* не указан, то возвращает случайное X, которое удовлетворяет условию 0 <=X< 1. Если параметр *Диапазон* указан, то функ-

ция возвращает случайное число типа `word`, удовлетворяющее условию 0 <=X< *Диапазон*.

Перед первым обращением к функции `Random` необходимо вызовом процедуры `Randomize` инициализировать программный генератор случайных чисел.

Randomize

Объявление:

```
procedure Randomize;
```

Действие:

Инициализирует программный генератор случайных чисел.

Преобразования

Int

Объявление:

```
function Int(X: real): real;
```

Действие:

Возвращает целую часть аргумента как значение вещественного типа. Дробная часть аргумента при преобразовании не учитывается, т. е. функция не производит округление.

Round

Объявление:

```
function Round(X: real): longint;
```

Возвращает округленное к ближайшему целому значение аргумента.

Str

Объявление:

```
procedure Str(X [:КоличествоСимволов [:ДробнаяЧасть ]];
              var Строка: string);
```

Действие:

Выполняет преобразование числового выражения в его строковое представление. *КоличествоСимволов* и *ДробнаяЧасть* — необязательные выражения целого типа, которые задают общее количество символов и количество символов дробной части в изображении числа.

Trunc

Объявление:

```
function Trunc(X: real): longint;
```

Возвращает целую часть аргумента как значение целого типа. Дробная часть аргумента при преобразовании не учитывается, т. е. функция не производит округление.

Val

Объявление:

```
procedure Val(Строка: string; var Переменная;
              var Ошибка: integer);
```

Действие:

Выполняет преобразование строки, изображающей целое или вещественное число, в число. Полученное значение присваивается переменной, указанной при вызове процедуры. Если преобразование не может быть выполнено, то в переменную *Ошибка* записывается номер символа строки, который явился причиной неудачи преобразования. Если преобразование выполнено успешно, то значение *Ошибка* равно нулю.

Для работы со строками и символами

Chr

Объявление:

```
function Chr(КодСимвола: byte): char;
```

Действие:

Возвращает символ с указанным кодом.

Concat

Объявление:

```
function Concat(Строка1 [, Строка2, ..., СтрокаN]: string): string;
```

Действие:

Возвращает строку, являющуюся объединением строк, указанных при вызове функции.

Copy

Объявление:

```
function Copy(Строка: string; НомерСимвола: integer;
              Длина: integer): string;
```

Возвращает подстроку. Параметр *Строка* задает строку, параметры *НомерСимвола* и *Длина* — начало и длину подстроки.

Delete

Объявление:

```
procedure Delete(var Строка: string; НомерСимвола: integer;
                     Длина: integer);
```

Действие:

Удаляет из строки подстроку. Параметр *Строка* задает строку, параметры *НомерСимвола* и *Длина* — начало и длину удаляемой подстроки.

Length

Объявление:

```
function Length(Строка: string): integer;
```

Действие:

Возвращает значение, равное количеству символов строки-аргумента.

Pos

Объявление:

```
function Pos(Строка: string; Подстрока: string): byte;
```

Действие:

Возвращает позицию (номер символа) подстроки в строке.

Графического режима

Bar

Объявление:

```
procedure Bar(x1,y1,x2,y2: integer);
```

Действие:

Вычерчивает закрашенный прямоугольник. Параметры *x1* и *y1* задают положение левого верхнего угла прямоугольника, *x2* и *y2* — правого нижнего. Используемый стиль и цвет заливки задается процедурой `SetFillStyle`.

Bar3D

Объявление:

```
procedure Bar3D(x1,y1,x2,y2: integer; Глубина: word;
                Граница: boolean);
```

Действие:

Вычерчивает параллелепипед. Параметры *x1* и *y1* задают положение левого верхнего, а *x2* и *y2* — правого нижнего угла ближней грани параллелепипеда. Параметр *Глубина* задает расстояние между передней и задней гранями, параметр *Граница* определяет, нужно ли вычерчивать верхнюю границу задней грани параллелепипеда. Ближняя грань закрашивается в соответствии с текущим, установленным процедурой `SetFillStyle`, стилем.

Circle

Объявление:

```
procedure Circle(x,y: integer; r: word);
```

Действие:

Вычерчивает окружность радиуса *r* с центром в точке с координатами *(x,y)*. Окружность вычерчивается линией со стилем, установленным процедурой `SetLineStyle`.

DetectGraph

Объявление:

```
procedure DetectGraph(var Драйвер,Режим: integer);
```

Действие:

Проверяет графический адаптер и определяет, какие графический драйвер и режим используются.

Ellipse

Объявление:

```
procedure Ellipse(x,y: integer; УголНачала,УголКонца: word;
                                РадиусX,РадиусY: word);
```

Действие:

Вычерчивает эллипс или дугу эллипса с центром в точке с координатами `(x,y)`. Параметры *УголНачала* и *УголКонца* задают круговые координаты начальной и конечной точек линии эллипса, которая вычерчивается против часовой стрелки от начальной к конечной точке. Угловые координаты задаются

в градусах. Значение угловой координаты возрастает против часовой стрелки. Параметры *РадиусX* и *РадиусY* задают горизонтальный и вертикальный радиусы эллипса. Линия эллипса вычерчивается с установленным процедурой SetLineStyle стилем.

GetX, GetY

Объявление:

```
function GetX: integer;
function GetY: integer;
```

Действие:

Возвращает координату X (Y) указателя вывода.

GraphResult

Объявление:

```
function GraphResult: integer;
```

Действие:

Возвращает результат (код ошибки) последней выполненной графической операции. Если операция выполнена успешно, функция возвращает ноль. Код ошибки выполнения графической операции устанавливают процедуры: Bar, Bar3D, InitGraph, PieSlice, SetFillPattern, SetFillStyle, SetLineStyle, SetTextStyle и др.

InitGraph

Объявление:

```
procedure InitGraph(var Driver: integer; var Mode: integer; Path: string);
```

Действие:

Инициализирует графический режим. Параметр *Driver* определяет драйвер видеосистемы, параметр *Mode* — режим работы видеосистемы, параметр *Path* — каталог, где находится драйвер.

Line

Объявление:

```
procedure Line(x1,y1,x2,y2: integer);
```

Действие:

Вычерчивает линию между двумя точками экрана, координаты которых указаны при вызове процедуры. Вид линии определяется установленным процедурой SetLineStyle стилем (типом) линии.

LineTo

Объявление:

```
procedure Line(x1,y1: integer);
```

Действие:

Вычерчивает линию от текущего положения указателя вывода до точки, координаты которой указаны при вызове процедуры. Вид линии определяется установленным процедурой `SetLineStyle` стилем линии.

MoveRel

Объявление:

```
procedure MoveRel(dx:,dy: integer);
```

Действие:

Перемещает указатель вывода на `dx` и `dy` пикселов. Если значение параметра `dx` (`dy`) положительное, то указатель перемещается вниз (вправо), если отрицательное, то — вверх (влево).

MoveTo

Объявление:

```
procedure MoveTo(x,y: integer);
```

Действие:

Перемещает указатель вывода в точку с координатами `x` и `y`.

OutText

Объявление:

```
procedure OutText(Текст: string);
```

Действие:

Выводит строку символов *Текст* от текущего положения указателя вывода текущим, установленным процедурой `SetTextStyle`, шрифтом.

OutTextXY

Объявление:

```
procedure OutTextXY(x,y: integer; Текст: string);
```

Действие:

Устанавливает указатель вывода в точку с координатами `(x,y)` и выводит строку символов *Текст* текущим, установленным процедурой `SetTextStyle`, шрифтом.

PieSlice

Объявление:

```
procedure PieSlice(x,y: integer;
                   УголНачала,УголКонца,Радиус: word);
```

Действие:

Вычерчивает окружность или дугу окружности радиуса *Радиус* с центром в точке с координатами (x,y). Параметры *УголНачала* и *УголКонца* задают круговые координаты начальной и конечной точек линии окружности, которая вычерчивается против часовой стрелки от начальной к конечной точке. Угловые координаты задаются в градусах. Значение угловой координаты возрастает против часовой стрелки. Нулевому углу соответствует горизонтальный отрезок, проведенный из точки (x,y) в сторону возрастания координаты x. Если *УголНачала*=0, а *УголКонца*=360, то процедура PieSlice вычерчивает окружность. Линия окружности или дуги вычерчивается с установленным процедурой SetLineStyle стилем.

PutPixel

Объявление:

```
procedure PutPixel(x,y: integer; Цвет: word);
```

Действие:

Окрашивает пиксел, точку с координатами (x,y), цветом *Цвет*. В качестве параметра *Цвет* обычно используют именованную константу (см. "SetColor").

Rectangle

Объявление:

```
procedure Rectangle(x1,y1,x2,y2: integer);
```

Действие:

Вычерчивает прямоугольник. Параметры *x1* и *y1* задают положение левого верхнего угла прямоугольника, *x2* и *y2* — правого нижнего. Контур прямоугольника вычерчивается с установленным процедурой SetLineStyle стилем.

Sector

Объявление:

```
procedure Sector(x,y: integer; Угол1,Угол2,РадиусX,РадиусY: word);
```

Действие:

Вычерчивает сектор окружности (*РадиусX* = *РадиусY*) или эллипса (*РадиусX* ≠ *РадиусY*). Параметры *x* и y задают местоположение центра сектора. Парамет-

ры `Угол1` и `Угол2` задают углы прямых, ограничивающих сектор, параметры *`РадиусX`* и *`РадиусY`* задают радиусы круга (эллипса) по осям X и Y, из которого "вырезается" отображаемый сектор. Нулевому углу соответствует горизонтальный отрезок, проведенный из точки `(x,y)` в сторону возрастания координаты `x`. Если *`Угол1`*`=0`, а *`Угол2`*`=360`, то процедура `Sector` вычерчивает полный круг (эллипс). Сектор закрашивается с текущим, установленным процедурой `SetFillStyle`, стилем.

SetColor

Объявление:

```
procedure SetColor(Цвет:word);
```

Действие:

Задает цвет вывода текста (процедуры `OutTextXY` и `OutText`), вычерчивания линий и фигур (процедуры `Line`, `Circle`, `Recktangle` и др.). В качестве параметра *`Цвет`* обычно используют именованные константы (табл. П1.1).

Таблица П1.1. *Кодирование цвета*

Цвет	Константа	Значение константы
Черный	`Black`	0
Синий	`Blue`	1
Зеленый	`Green`	2
Бирюзовый	`Cyan`	3
Красный	`Red`	4
Сиреневый	`Magenta`	5
Коричневый	`Brown`	6
Белый (светло-серый)	`LightGray`	7
Серый	`DarkGray`	8
Голубой	`LightBlue`	9
Светло-зеленый	`LightGreen`	10
Светло-бирюзовый	`LightCyan`	11
Светло-красный (алый)	`LightRed`	12
Светло-сиреневый	`LightMagenta`	13
Желтый	`Yellow`	14
Ярко-белый	`White`	15

SetFillStyle

Объявление:

```
procedure SetFillStyle(Стиль,Цвет: word);
```

Действие:

Устанавливает стиль и цвет заливки (закрашивания), используемый процедурами вывода областей (Bar, Bar3D, Sector и др.). В качестве параметра *Стиль* обычно используют одну из именованных констант, список которых приведен в табл. П1.2. Параметр *Цвет* также задается именованной константой (см. "SetColor").

Таблица П1.2. *Стили заливки областей*

Константа	Стиль заполнения области
EmptyFill	Без заливки (сплошная заливка цветом фона)
SolidFill	Сплошная заливка текущим цветом
LineFill	Горизонтальная штриховка
LtSlashFill	Штриховка под углом 45 градусов влево тонкими линиями
SlashFill	Штриховка под углом 45 градусов влево
BkSlashFill	Штриховка под углом 45 градусов вправо тонкими линиями
LtBkSlashFill	Штриховка под углом 45 градусов вправо
HatchFill	Штриховка клеткой
XhatchFill	Штриховка под углом 45 градусов редкой косой клеткой
InterleaveFill	Штриховка под углом 45 градусов частой косой клеткой
WideDotFill	Заполнение редкими точками
CloseDotFill	Заполнение частыми точками
UserFill	Тип заполнения определяется программистом

SetLineStyle

Объявление:

```
procedure SetLineStyle(ТипЛинии: word; Образец: word;
                       Толщина: word);
```

Устанавливает стиль вычерчиваемых контуров и линий (см. процедуры Line, Circle и др.).

Параметр *ТипЛинии*, в качестве которого обычно используется одна из именованных констант, определяет вид линии (табл. П1.3).

Таблица П1.3. *Кодирование вида линий*

Константа	Тип линии
SolidLn	Сплошная, непрерывная
DottedLn	Пунктирная, с постоянной длиной штрихов
CenterLn	Штрих-пунктирная линия
DashedLn	Пунктирная, длина штрихов чуть больше, чем у линии типа DottedLn
UserBitLn	Определенный программистом тип линии

Параметр *Толщина* определяет толщину линии. Линия может быть обычной толщины (константа NormWidth) или утолщенная (константа ThickWidth).

Параметр *Образец* используется в том случае, если процедура SetLineStyle устанавливает тип линии, определяемый программистом. Значением параметра *Образец* должна быть четырехразрядная шестнадцатеричная константа, кодирующая отрезок линии длиной в 16 пикселов.

SetTexStyle

Объявление:

```
procedure SetTexStyle(Шрифт,Ориентация,Размер: word);
```

Действие:

Устанавливает шрифт, размер и ориентацию текста, выводимого процедурами OutTextXY и OutText. В качестве параметра *Шрифт* можно использовать одну из констант, перечисленных в табл. П1.4.

Таблица П1.4. *Шрифты*

Константа	Значение	Шрифт
DefaultFont	0	Стандартный. Каждый выводимый символ формируется в квадрате размером 8×8 пикселов
TriplexFont	1	Triplex
SmallFont	2	Мелкий
SansSerifFont	3	SansSerif
GothicFont	4	Готический

Параметр *Ориентация* задает ориентацию выводимого процедурами OutText и OutTextXY текста. Текст может быть ориентирован обычным образом (значение параметра *Ориентация* в этом случае равно именованной константе Normal) или вертикально, снизу вверх (в этом случае значение параметра *Ориентация* равно Vertical).

Для работы с файлами

Append

Объявление:

```
procedure Append(var F: text);
```

Действие:

Открывает существующий файл, связанный с файловой переменной *F*, в режиме добавления в конец файла. Попытка открыть несуществующий файл вызывает ошибку времени выполнения программы.

Assign

Объявление:

```
procedure Assign(var F; ИмяФайла: string);
```

Действие:

Связывает файловую переменную *F* с конкретным файлом.

Close

Объявление:

```
procedure Close(var F);
```

Действие:

Закрывает файл, связанный с файловой переменной *F*.

EOF

Объявление:

```
function EOF(var F): boolean;
```

Действие:

Проверяет, не достигнут ли конец файла (end of file) при чтении из файла, связанного с файловой переменной *F*. Если указатель чтения достиг конца файла, то функция EOF возвращает TRUE, в противном случае — FALSE.

Erase

Объявление:

```
procedure Erase(var F);
```

Действие:

Уничтожает файл, имя которого связано с файловой переменной *F*.

IOResult

Объявление:

```
function IOResult: integer;
```

Действие:

Возвращает код результата последней выполненной операции файлового ввода/вывода (в том числе открытия и закрытия файла). Если операция ввода/вывода выполнена успешно, функция возвращает ноль.

Чтобы программа могла использовать функцию `IOResult`, надо перед инструкциями, в результате которых может возникнуть ошибка ввода/вывода, поместить директиву `{$I-}`, а после этих инструкций — директиву `{$I+}`.

Reset

Объявление:

```
procedure Reset(var F [:file; РазмерЗаписи: word]);
```

Действие:

Открывает существующий файл. Файл может быть любого типа. Если элементы файла не относятся к одному из стандартных типов, то параметр *РазмерЗаписи* задает размер записи. При попытке открыть несуществующий файл возникает ошибка времени выполнения.

Rewrite

Объявление:

```
procedure Rewrite(var F [:file; РазмерЗаписи: word]);
```

Действие:

Создает и открывает файл с именем, связанным с файловой переменной *F*. Если файл с таким именем уже существует, то процедура `Rewrite` его уничтожает и создает новый.

Прочие

ClrEol

Объявление:

```
procedure ClrEol;
```

Действие:

Очищает текущую строку (строка, в которой находится курсор) от позиции, в которой находится курсор, до конца строки, закрашивая ее текущим, заданным процедурой `TextBackGround`, цветом.

ClrScr

Объявление:

```
procedure ClrScr;
```

Действие:

Очищает текущее, заданное процедурой `Window`, окно экрана, закрашивая его текущим, заданным процедурой `TextBackGround`, цветом.

Delay

Объявление:

```
procedure Delay(Задержка: word);
```

Действие:

Обеспечивает задержку на указанное при вызове процедуры количество миллисекунд.

Dispose

Объявление:

```
procedure Dispose(var p);
```

Действие:

Освобождает память, занимаемую динамической переменной, на которую указывает указатель *p*.

Eoln

Объявление:

```
function Eoln[(var F: text)]: boolean;
```

Действие:

Если при обращении к функции `Eoln` параметр не указан, то функция проверяет, не является ли очередной читаемый из буфера клавиатуры символ

символом "новая строка". Если является, то возвращает `TRUE`, в противном случае — `FALSE`. Если параметр указан, то функция аналогичным образом проверяет очередной символ текстового файла, связанного с файловой переменной, указанной при вызове функции.

GotoXY

Объявление:

```
procedure GoToXY(x,y: byte);
```

Действие:

Перемещает курсор в точку экрана с координатами `(x,y)`.

Halt

Объявление:

```
procedure Halt[(КодЗавершения: word)];
```

Действие:

Завершает выполнение программы и передает управление операционной системе.

New

Объявление:

```
procedure New(var p);
```

Действие:

Выделяет память для динамической переменной и присваивает указателю `p` адрес выделенной области.

ParamCount

Объявление:

```
function ParamCount: word;
```

Действие:

Возвращает количество параметров командной строки.

ParamStr

Объявление:

```
function ParamStr(N: word): string;
```

Действие:

Возвращает параметр командной строки, номер которого указан при обращении к функции. Значением `ParamStr(0)` является путь к файлу выполняемой программы и его имя, например, c:\tp\exe&tpu\myprog.exe.

ReadKey

Объявление:

```
function ReadKey: char;
```

Действие:

Возвращает символ, соответствующий нажатой клавише. Если нажата служебная клавиша, то возвращает ноль.

TextBackGround

Объявление:

```
procedure TextBackGround(Цвет: byte);
```

Действие:

Задает цвет фона сообщений, выводимых инструкциями `write` и `writeln`.

В качестве параметра *Цвет* может использоваться одна из именованных констант, перечисленных в табл. П1.5.

Таблица П1.5. *Кодирование цвета фона*

Константа	Цвет	Номер цвета
Black	Черный	0
Blue	Синий	1
Green	Зеленый	2
Cyan	Бирюзовый	3
Red	Красный	4
Magenta	Сиреневый	5
Brown	Коричневый	6
LightGray	Белый (светло-серый)	7

TextColor

Объявление:

```
procedure TextColor(Цвет: byte);
```

Действие:

Устанавливает цвет символов сообщений, выводимых инструкциями `write` и `writeln`. В качестве параметра *`Цвет`* может использоваться одна из именованных констант, перечисленных в табл. П1.6.

Таблица П1.6. *Кодирование цвета символов*

Константа	Цвет	Номер цвета
`Black`	Черный	0
`Blue`	Синий	1
`Green`	Зеленый	2
`Cyan`	Бирюзовый	3
`Red`	Красный	4
`Magenta`	Сиреневый	5
`Brown`	Коричневый	6
`LightGray`	Белый (светло-серый)	7
`DarkGray`	Серый	8
`LightBlue`	Голубой	9
`LightGreen`	Светло-зеленый	10
`LightCyan`	Светло-бирюзовый	11
`LightRed`	Светло-красный (алый)	12
`LightMagenta`	Светло-сиреневый	13
`Yellow`	Желтый	14
`White`	Ярко-белый	15

WhereX

Объявление:

```
function WhereX: byte;
```

Действие:

Возвращает координату `X` курсора в текущем, заданном процедурой `Window`, окне.

WhereY

Объявление:

```
function WhereY: byte;
```

Действие:

Возвращает координату `Y` курсора в текущем, заданном процедурой `Window`, окне.

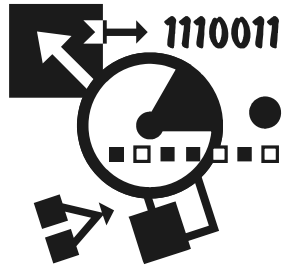
Window

Объявление:

```
procedure Window(x1,y1,x2,y2: byte);
```

Действие:

Определяет окно — область экрана. Параметры *x1*, *y1* задают координаты левого верхнего угла окна относительно экрана, параметры *x2*, *y2* — правого нижнего.



Приложение 2

ASCII - кодировка символов

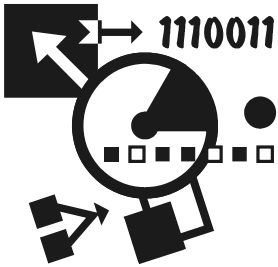
В операционной системе DOS используется кодировка символов, которая называется ASCII. Ниже приведены символы и соответствующие им коды.

- 0	► - 16	- 32	0 - 48	@ - 64	P - 80	` - 96	p - 112
☺ - 1	◄ - 17	! - 33	1 - 49	A - 65	Q - 81	a - 97	q - 113
☻ - 2	↕ - 18	" - 34	2 - 50	B - 66	R - 82	b - 98	r - 114
♥ - 3	‼ - 19	# - 35	3 - 51	C - 67	S - 83	c - 99	s - 115
♦ - 4	¶ - 20	$ - 36	4 - 52	D - 68	T - 84	d - 100	t - 116
♣ - 5	§ - 21	% - 37	5 - 53	E - 69	U - 85	e - 101	u - 117
♠ - 6	▬ - 22	& - 38	6 - 54	F - 70	V - 86	f - 102	v - 118
- 7	↨ - 23	' - 39	7 - 55	G - 71	W - 87	g - 103	w - 119
- 8	↑ - 24	(- 40	8 - 56	H - 72	X - 88	h - 104	x - 120
- 9	↓ - 25	) - 41	9 - 57	I - 73	Y - 89	i - 105	y - 121
- 10	→ - 26	* - 42	: - 58	J - 74	Z - 90	j - 106	z - 122
♂ - 11	← - 27	+ - 43	; - 59	K - 75	[- 91	k - 107	{ - 123
♀ - 12	∟ - 28	, - 44	< - 60	L - 76	\ - 92	l - 108	| - 124
- 13	↔ - 29	- - 45	= - 61	M - 77	] - 93	m - 109	} - 125
♫ - 14	▲ - 30	. - 46	> - 62	N - 78	^ - 94	n - 110	~ - 126
☼ - 15	▼ - 31	/ - 47	? - 63	O - 79	_ - 95	o - 111	⌂ - 127

А - 128	Р - 144	а - 160	▒ - 176	└ - 192	╨ - 208	р - 224	Ё - 240
Б - 129	С - 145	б - 161	▓ - 177	┴ - 193	╤ - 209	с - 225	ё - 241
В - 130	Т - 146	в - 162	│ - 178	┬ - 194	╥ - 210	т - 226	Є - 242
Г - 131	У - 147	г - 163	┤ - 179	├ - 195	╙ - 211	у - 227	є - 243
Д - 132	Ф - 148	д - 164	┐ - 180	─ - 196	╘ - 212	ф - 228	Ї - 244
Е - 133	Х - 149	е - 165	╡ - 181	┼ - 197	╒ - 213	х - 229	ї - 245
Ж - 134	Ц - 150	ж - 166	╢ - 182	╞ - 198	╓ - 214	ц - 230	Ў - 246
З - 135	Ч - 151	з - 167	╖ - 183	╟ - 199	╫ - 215	ч - 231	ў - 247
И - 136	Ш - 152	и - 168	╕ - 184	╚ - 200	╪ - 216	ш - 232	° - 248
Й - 137	Щ - 153	й - 169	╣ - 185	╔ - 201	┘ - 217	щ - 233	∙ - 249
К - 138	ъ - 154	к - 170	║ - 186	╩ - 202	┌ - 218	ъ - 234	· - 250
Л - 139	Ы - 155	л - 171	╗ - 187	╦ - 203	█ - 219	ы - 235	√ - 251
М - 140	ь - 156	м - 172	╝ - 188	╠ - 204	▄ - 220	ь - 236	№ - 252
Н - 141	Э - 157	н - 173	╜ - 189	═ - 205	▌ - 221	э - 237	¤ - 253
О - 142	Ю - 158	о - 174	╛ - 190	╬ - 206	▐ - 222	ю - 238	■ - 254
П - 143	Я - 159	п - 175	┐ - 191	╧ - 207	▀ - 223	я - 239	- 255

Таблица П2.1. *Специальные символы*

Символ	Код	Действие
"Забой" (Backspace)	8	Перемещает курсор на одну позицию влево
"Табуляция" (Tab)	9	Перемещает курсор в следующую позицию, номер которой кратен восьми
"Перевод строки" (LF)	10	Перемещает курсор в следующую строку
"Возврат каретки" (CR)	13	Перемещает курсор в начало текущей строки



Приложение 3

Представление информации в компьютере

Десятичные, двоичные и шестнадцатеричные числа

В повседневной жизни человек имеет дело с десятичными числами. В *десятичной системе* счисления для представления чисел используются цифры от 0 до 9. Значение числа определяется как сумма произведений цифр числа на их весовые коэффициенты, определяемые местами цифр в числе. Весовой коэффициент самой правой цифры равен единице, цифры перед ней — десяти, затем — ста и т. д. Например, число 2705 равно $2\times1000+7\times100+0\times10+5\times1$.

Если места цифр (разряды) пронумеровать справа налево и самой правой позиции присвоить номер "ноль", то можно заметить, что вес i-го разряда равен i-й степени десяти (рис. П3.1).

Для внутреннего представления чисел компьютер использует *двоичную систему* счисления. Двоичные числа записываются при помощи двух цифр — нуля и единицы. Как и десятичная, двоичная система — позиционная. Весовой коэффициент разряда i-го равен двум в i-й степени (рис. П3.2).

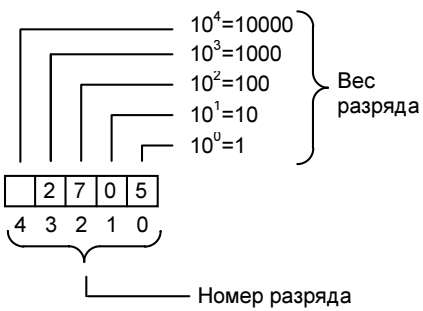
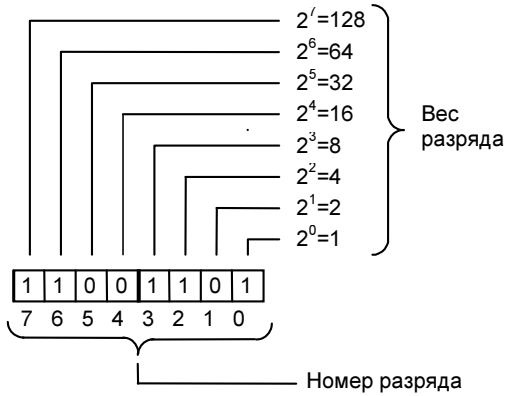


Рис. П3.1. Вес разряда в десятичной системе счисления

Двоичные числа наиболее точно отражают состояние памяти, регистров процессора и внешних устройств компьютера. Вместе с тем, работать с двоичными числами не совсем удобно — слишком много цифр приходится записывать. Поэтому была разработана шестнадцатеричная система счисления и записи чисел, позволяющая компактно записывать двоичные числа и обеспечивающая простой способ перевода двоичного числа в шестнадцатеричное и обратно.



Можно задать одно и то же число так:
$1\times128 + 1\times64 + 0\times32 + 0\times16 + 1\times8 + 1\times4 + 0\times2 + 1\times1 = 205$
или
$(11001101)_2 = (205)_{10}$

Рис. П3.2. Вес разряда в двоичной системе счисления

Используя четыре двоичные цифры, можно записать шестнадцать чисел (максимальное значение четырехразрядного двоичного числа равно пятнадцати).

Шестнадцатеричное число получается из двоичного следующим образом (рис. П3.3).

Цифры двоичного числа делятся на группы по четыре. Каждой группе ставится в соответствие сначала десятичное число, являющееся десятичным эквивалентом четырехзначного двоичного, затем полученное десятичное число записывается шестнадцатеричной цифрой. В табл. П3.1 приведены десятичные числа от нуля до 15 и соответствующие им шестнадцатеричные цифры.

В тексте программы перед первой цифрой шестнадцатеричного числа ставится знак $. Вот примеры шестнадцатеричных чисел: $2A, $FF, $01.

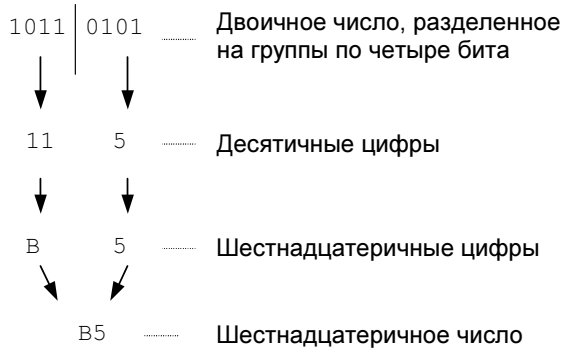


Рис. П3.3. Перевод двоичного числа в шестнадцатеричное

Таблица П3.1. *Десятичные числа и шестнадцатеричные цифры*

Десятичное число	Шестнадцатеричная цифра
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Память компьютера

В памяти компьютера числа хранятся в виде *битов*. Каждый бит может принимать значение одной двоичной цифры. Следовательно, значением бита может быть ноль или единица. Восемь битов объединены в *байт*. Максимальное число, которое можно записать при помощи восьми двоичных цифр, это 1111 1111, что соответствует десятичному числу 255, минимальное — ноль. Поэтому значением байта может быть число от нуля до 255.

Переменные хранятся в памяти. Так как переменные различных типов могут принимать различные значения, то для их хранения нужно разное количество памяти. Память под переменные выделяется целым числом байтов. Например, значением переменной типа `char` может быть любой из 256 символов. Поэтому для хранения переменной этого типа достаточно одного байта. Значением переменной типа `integer` может быть число от –32 768 до 32 767 (65 535 значений), для хранения переменной этого типа требуется два байта. Таким образом, чем больше диапазон значений типа, тем больше байтов нужно для хранения переменной этого типа (см. табл. П3.2).

Таблица П3.2. *Память, занимаемая переменными*

Тип переменной	Диапазон значений	Занимаемая память (байтов)
`char`	Любой символ	1
`string`	Строка до 256 символов	256
`string[n]`	Строка до n символов	1 × n
`byte`	0...255	1
`word`	0...65 535	2
`integer`	-32 768...32 767	2
`longint`	-2 147 483 648...2 147 483 647	4
`real`	$2.9 \cdot 10^{-39}$...$1.7 \cdot 10^{38}$	6
`single`	$1.5 \cdot 10^{-45}$...$3.4 \cdot 10^{38}$	4
`double`	$5.0 \cdot 10^{-324}$...$1.7 \cdot 10^{308}$	8
`extended`	$3.4 \cdot 10^{-4932}$...$1.1 \cdot 10^{4932}$	8

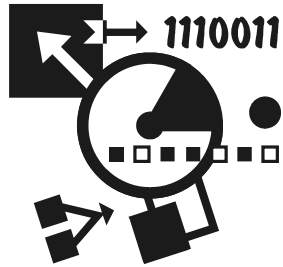
В программе для одного и того же значения можно использовать переменные разных типов (при этом будет использоваться разное количество памяти).

Например, если в программе используется переменная `Day` (число месяца), то для нее можно задать тип `byte`, `integer` или `longint`. В первом случае будет занят один байт памяти, во втором — два, в третьем — четыре. Но реально будет использоваться только один байт, а остальные будут просто заняты. Поэтому, выбирая тип для переменной, следует подбирать наиболее подходящий. Особо следует обратить внимание на описание строковых переменных и массивов.

Выделяя память для строковых переменных, следует помнить, что если не указана предельная длина строки, то переменной выделяется 256 байт. Объявляя переменную, предназначенную, например, для хранения имени человека, следует писать `name:string[30]`, а не `name:string`.

Каждому массиву программы выделяется память, объем которой определяется как типом элементов массива, так и их количеством. Для хранения двумерного массива, например, 20×20, вещественных чисел нужно более трех килобайт памяти ($20 \times 20 \times 8 = 3200$).

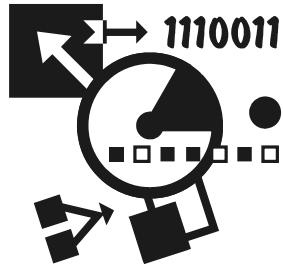
Память компьютера кажется неограниченной, но если ее нерационально использовать, то в некоторый момент ее может не хватить.



Приложение 4

Рекомендуемая литература

1. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. — М.: Мир, 1989. — 360 с., ил.
2. Зелковиц М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения: Пер. с англ. — М.: Мир, 1982. — 386 с., ил.
3. Практическое руководство по программированию: Пер. с англ./ Б. Мик, П. Хит, Н. Рашби и др.; под ред. Б. Мика, П. Хит, Н. Рашби. — М.: Радио и связь, 1986. — 168 с., ил.
4. Фокс Дж. Программное обеспечение и его разработка: пер. с англ. — М.: Мир, 1985. — 368 с., ил.
5. Культин Н. Б. Turbo Pascal в примерах и задачах. — СПб.: БХВ-Петербург, 2000. — 256 с., ил.
6. Культин Н. Б. Delphi в задачах и примерах. — СПб.: БХВ-Петербург, 2003. — 288 с., ил.
7. Культин Н. Б. Основы программирования в Turbo Delphi. — СПб.: БХВ-Петербург, 2007. — 384 с., ил.



Приложение 5

Описание CD

Прилагаемый к книге CD содержит исходные тексты Pascal-программ, а также проекты Delphi, которые приведены в книге в качестве примеров. Pascal-программы находятся в каталоге PAS, проекты Delphi — в каталоге Delphi.

Предметный указатель

A

and 43
append 143
assign 143
AutoSize 273, 321

B

Bar 201
Bar3D 202
boolean 27
Button 275

C

Caption 273, 323
case 49, 343
char 26, 92
Checked 323
Chr 93
Circle 200
Click 277
close 143
CloseGraph 192
ClrScr 132
Code Explorer 284
constructor 255
copy 104
Create 277
Crt 111

D

DblClick 277
Delay 212
delete 102
Delphi 259
destructor 256
dispose 188
Dispose 255
done 256
double 26

E

Edit 269
Ellipse 200
Enter 277
EOF 153
Exit 277
extended 26

F

Font 270, 273, 323
for 53, 344

G

GetX 195
GetY 195
GoToXY 129
GraphResult 192

H, I

Height 270, 273, 321, 323
if 44, 342
InitGraph 192
IOResult 146

K

KeyDown 277
KeyPress 277
KeyUp 277

L

Label 273
Left 270, 273, 321, 323
length 101
Line 199
LineTo 199

M

MouseDown 277
MouseMove 277
MouseUp 277
MoveRel 195
MoveTo 195, 200

N

Name 270, 273, 321, 323
new 181
New 249

O

Object Inspector 262
or 43
Ord 99
OutText 204
OutTextXY 204

P

Paint 277
ParamCount 163
ParamStr 163
ParentFont 270, 273
Picture 321
PieSlice 202
pos 103
private 245
Proportional 321
PutPixel 199

R

read 149
ReadKey 132
readln 36, 98, 149
real 26
Rectangle 201
RegisterBGIDriver 222
repeat 55, 344
reset 143, 148
rewrite 143
Round 208

S

Sector 203
SetColor 196, 200
SetFillStyle 197
SetLineStyle 196, 200
SetTextStyle 206
single 26
Standard 269
Str 205
Stretch 321
string 27

T

Text 270
TextBackground 130
TextColor 130
TextHeight 206
TextWidth 206
Top 270, 273, 321, 323

U

UpCase 99
uses 111, 127

V, W

val 106
while 58, 345
Width 270, 273, 321, 323
with 174
WordWrap 273
write 34, 203
writeln 34, 203

А, Б

Алгоритм 24
Байт 370
Библиотека:
  CRT 128
  Graph 192
  визуальных компонентов 304
  времени выполнения 304
Бинарный поиск 77
Бит 370
Буфер клавиатуры 65

В

Ввод данных 36
Видеоадаптер 191
Виртуальный метод 251, 312
Выбор 41, 44
Вывод данных 34
Выполнение программы 16
Выражение 30
  тип 32
Вычерчивание:
  линии 199
  прямоугольника 201
  эллипса 200

Г

Графический
  драйвер 192
  примитив 195
  режим 192

Д

Деструктор 256, 307
Драйвер 191

З

Запись 173
  ввод 176
  вывод 176
  массив 175
  объявление 173, 342
  поле 173
Запуск:
  Turbo Pascal 5
  Delphi 260

И

Инкапсуляция *307*
Исключение 294
Исполняемый файл 18

К

Класс 305
Командная кнопка 275
Компиляция 13, 25
  в память 14
  на диск 14
Компонент 264
  добавление 269
  изменение размера 271
  имя 269
  перемещение 271
Константа:
  именованная 29
  логическая 29
  объявление 29
  символьная 92
  строковая 29, 97
  тип 30
  числовая 28
Конструктор 255, 305
Координаты:
  курсора 128
  точки экрана в графическом
    режиме 194

Л

Линия:
  вычерчивание 199
  стиль 196

М

Массив:
  ввод 63
  вывод 66
  доступ к элементу 62
  многомерный 81
  объявление 61, 341
  ошибки
    при использовании 89
  поиск элемента 75
  сортировка 70
Метод 243
  виртуальный 251
  деструктор 256
  конструктор 255
  наследование 246
  объявление 244
  переопределение 246
  класса 307
Модуль программиста:
  использование 140
  компиляция 140
  разделы 137

Н

Наследование полей 241

О

Область:
  стиль 197
  стиль программиста 198
  цвет 197
Объект 240, 262, 305
  динамический 249
  объявление 240
  свойства 262
  создание 255
  тип 240
  уничтожение 255
Объявление:
  записи 173
  процедуры 116
  файла 142
  функции 112
ООП 240
Операнд 30
Оператор 30
  логический 43
  приоритет 31
  сравнения 43
Ориентация вывода текста 206
Отладка по шагам 236
Отладчик 235
Ошибка:
  времени выполнения 17, 234, 294
  времени компиляции 14
  открытия файла 146
  синтаксическая 14, 293

П

Параметр:
  командной строки DOS 163
  фактический 119
  формальный 119
Переменная 27
  глобальная 122
  динамическая 181
  контроль значения 239
  локальная 121
  объявление 27
  указатель 180
Пиксел 194
Поле:
  вывода текста 273
  редактирования 269
Полиморфизм 251
Приложение 260
Приоритет операторов 31

Присваивание 30, 32
Программа прикладная 260
Программирование объектно-ориентированное 240
Процедура 345
  библиотечная 127
  программиста 116

Р

Раздел:
  инициализации 137
  интерфейса 137
  реализации 137
Редактор кода 11
Рекурсия 223
Ресурс 287

С

Свойства объекта 262, 308
Символ 92
  код 93
  объявление 92
  специальный 93
Система счисления:
  двоичная 367
  десятичная 367
  шестнадцатеричная 368
Следование 40
Событие 277
  процедура обработки 278
Сообщение об ошибке 15
Список:
  добавление элемента 183
  односвязный 182
  удаление элемента 187
Стиль линии 196
Строка 97
  ввод 98
  выделение подстроки 104
  вычисление длины 101
  объявление 97, 341
  преобразование в число 106
  сложение 98
  сравнение 97
  удаление подстроки 102
Структура программы 37

Т

Текущий цвет 196
Тип данных 340
  вещественный 26
  интервальный 172
  логический 27
  определяемый программистом 169
  перечисляемый 169
  символьный 26, 92
  строковый 27, 97
  целый 26
Точка останова 237
  характеристики 238
Трассировка 236

У

Указатель вывода 195
  перемещение 195
  положение 195
Условие 42

Ф

Файл 142
  закрытие 143
  запись 143
  имя 143
  исполняемый 18
  объявление 142
  открытие 143
  чтение 148
Форма 264
  заголовок 266
Формат 35
Функция 33, 109, 345
  библиотечная 127
  программиста 112
  стандартная 110

Ц

Цвет
  отрисовки в графическом режиме 196
  символов 130
  фона 130
Цикл 42, 52
  с постусловием 55, 344
  с предусловием 58, 345
  с фиксированным количеством повторений 53, 344

Ш

Шаблон кода 284
Шестнадцатеричное число 369
Шрифт 206

Э, Я

Экран:
  в алфавитно-цифровом режиме 128
  в графическом режиме 194
Ярлык, создание 5