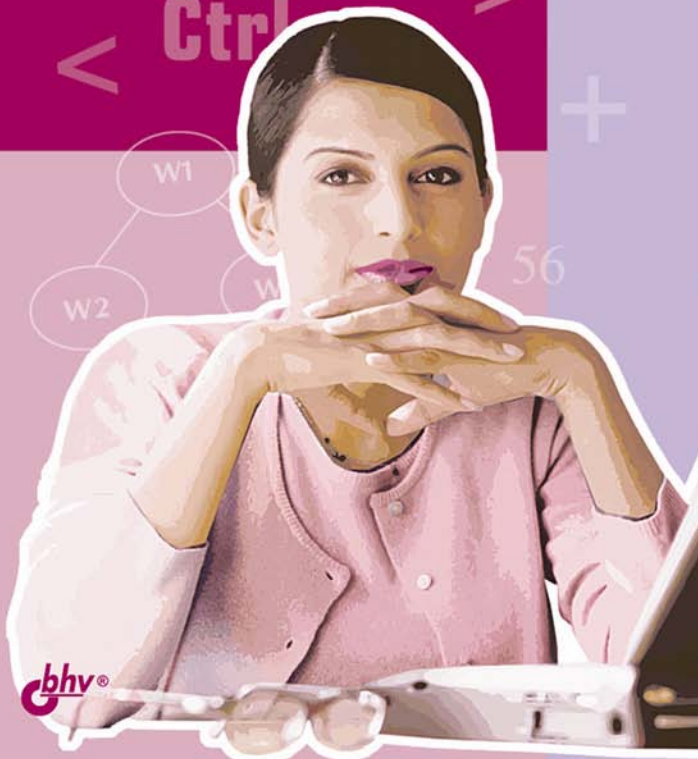
Б. И. Пахомов

C/C++ и Borland C++ Builder

для студента



bhv®

Борис Пахомов

C/C++ и Borland C++ Builder для студента

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06+800.92(075.8)C++
ББК 32.973.26-018.1я73
П12

Пахомов Б. И.

П12 С/С++ и Borland C++ Builder для студента. — СПб.: БХВ-Петербург, 2006. — 448 с.: ил.

ISBN 5-94157-816-4

Книга является руководством для студентов, начинающих изучать среду Borland C++ Builder и языки программирования С/С++. Рассмотрены основные элементы языков С/С++ и примеры создания простейших классов и программ. Изложены принципы визуального проектирования и событийного программирования. На конкретных примерах показаны основные возможности визуальной среды разработки C++ Builder, назначение базовых компонентов и процесс разработки различных типов Windows- и интернет-приложений, в том числе приложений с использованием технологий BDE и MIDAS.

Для студентов, изучающих курс информационных технологий

УДК 681.3.06+800.92(075.8)C++
ББК 32.973.26-018.1я73

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Алия Шаулис*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 17.01.06.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 28.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 5-94157-816-4

Оглавление

Введение

Мы приступаем к изучению среды Borland C++ Builder. Что это за среда? Какую помощь она оказывает в разработке программного обеспечения и чем отличается от других программных продуктов? Можно ли ее установить на вашем компьютере? Попробуем ответить на эти и ряд других вопросов, касающихся применения Borland C++ Builder (в дальнейшем для краткости Builder).

Builder — это среда, в которой можно осуществлять так называемое *визуальное программирование*, т. е. создавать программы, которые во время исполнения взаимодействуют с пользователем благодаря многооконному графическому интерфейсу. Какими преимуществами обладает многооконный графический интерфейс? Если сказать просто, то в момент выполнения программы элементы управления программой могут выглядеть на экране как графические объекты:

- кнопки, на которые можно нажимать мышью, после чего происходят некоторые "привязанные" к этим кнопкам действия;
- поля для ввода-вывода данных;
- списки данных, из которых можно выбирать данные для дальнейших расчетов в программе;
- различные меню, позволяющие выбирать и выполнять определенные действия;
- элементы, контролирующие состояния объектов (типа "включен — выключен", "выбран — не выбран" и т. п.);
- элементы, позволяющие следить за ходом некоторых процессов по времени их выполнения;
- элементы, позволяющие выбирать даты из календаря;
- элементы, обеспечивающие стандартный выбор файлов, шрифтов, цвета, настройки принтеров и т. д.;

- элементы, позволяющие вставлять в вашу программу другие программы-объекты (например, программы Word, Excel, анимационные файлы, диаграммы, "устройства" аудио- и видеозаписи, различные клипы мультимедиа и пр.).

Кроме этого, на экране могут появляться различные изображения, иерархические деревья, которые помогают лучше понять ход выполнения программы и управлять им. Среда Builder позволяет работать как с простыми локальными и удаленными базами данных, так и с многозвенными распределенными базами данных. Кроме того, среда Builder позволяет установить соединение и взаимодействие вашей программы с Интернетом.

В среде Builder разработка программ ведется на основе современного метода — объектно-ориентированного программирования.

На рынке программных продуктов есть много сред для автоматизации программирования (например, Visual Basic, Visual C++, Borland Delphi). По мощности и удобству использования со средой Builder может соперничать только Borland Delphi. Но эта последняя, по мнению автора, уступает среде Builder из-за того, что использует алгоритмический язык Объектный Паскаль, в то время как языком среды Builder является алгоритмический язык C++, более мощный и удобный для программирования. Если изучить сначала C++, а потом изучить Паскаль и попробовать составлять на нем программы (например, в среде Delphi), то после работы на C++ это не принесет никакой радости. Постоянно что-то раздражает: то надо каждый раз при добавлении новых переменных и меток обращаться к разделу объявлений, то не идет простейшее преобразование данных при присвоении, и надо начинать использовать операцию преобразования типов данных. И потом, в Паскале — тьма типов числовых данных, без которых C++ успешно обходится! На C++ написана масса стандартных программ, которые позволяют применять его во многих областях знания и даже на уровне Ассемблера. Мало того, что программа на C++ позволяет включать в свой состав блоки ассемблерных программ, но существуют и стандартные программы на C++, реализующие ассемблерные функции (например, обработку прерываний, работу с секторами дисков и т. п.).

Кроме того, по мнению автора, Паскаль более труден для освоения начинающими. Только имея определенный опыт работы на

C++, начинаешь глубоко понимать многие моменты в языке Паскаль. А школьники, которым его преподают в течение двух лет, а студенты, не имеющие опыта работы с подобными языками... Кажется весьма сомнительной необходимость преподавания этим категориям учащихся языка Паскаль в качестве их первого алгоритмического языка. Такой метод надолго отбивает охоту у людей к изучению алгоритмических языков вообще. С высоты знаний сегодняшних экономических отношений в обществе приходит в голову крамольная мысль, что кому-то не без выгоды удалось протолкнуть свое детище в широкие массы учащейся молодежи.

В основе языка C++ лежит язык C. Основным, *кардинальным* отличием C от C++ является наличие *классов* в последнем. Поэтому изучение C++ начинается с изучения C.

Предлагаемая книга состоит из двух частей. В первой изучается C и его расширение C++, во второй — собственно среда Builder, использующая C++. Для изучения C (C++) применен такой подход: читателю сначала предлагается программа, написанная на C, затем разъясняется, как и почему она работает, а затем объясняются все элементы, входящие в программу (объявления переменных, что такое переменные, типы данных, что это такое и т. п.). То есть: от практики — к теории, а не наоборот. Автор надеется, что такой подход вызовет бóльший интерес у начинающих и не отобьет у них охоту к изучению языка. В то время как при старом, традиционном подходе (от теории — к практике) непривычные термины ("типы данных", "переменные" и т. д.) тут же вызывают непонимание, скуку на лицах и нескрываемую зевоту.

Программы, которые мы будем создавать в среде Builder на языке C без применения собственно элементов Builder, — это так называемые *консольные приложения*. То есть программы, которые запускаются без графического интерфейса в консольном окне. Когда запускается консольное приложение, Windows создает консольное (опорное) окно в текстовом режиме, через которое пользователь взаимодействует с приложением. С этим окном тут же связывается стандартный вывод: все, что будет выводить программа с помощью стандартных программ вывода данных, станет отображаться в этом окне. Консольные приложения обычно

не требуют большого пользовательского ввода и выполняют ограниченный набор функций. Мы станем применять консольные приложения для изучения языка C, чтобы потом работать с этим языком в более сложных, графических приложениях, имеющих многооконный графический интерфейс, через который пользователь может взаимодействовать с приложением.

Чтобы создать консольное приложение, следует выполнить команды главного меню **Builder: File/New** (в версии 5) или **File/New/Other** (в версии 6) и в открывшемся диалоговом окне выбрать значок **Console Application** на вкладке **New**. После этого в другом открывшемся диалоговом окне надо выбрать тип языка (C или C++) для главной функции будущего приложения-программы.

Примечание

Главная функция — это функция, с которой запускается сама программа. В консольном приложении ее имя — `main()`, для неконсольного же приложения (т. е. приложения с использованием элементов собственно среды Builder) имя главной функции `WinMain()`. Но об этом — по мере изучения материала.

В последнем диалоговом окне есть кнопка **VCL**. Этой аббревиатурой обозначают визуальные компоненты и другие классы, применяемые в среде Builder. В консольном приложении использовать эту кнопку следует на этапе изучения классов в C++: например, если надо построить приложение со строковым классом `String` (об этом — в свое время).

Итак, для изучения C (C++) строятся консольные приложения. Для изучения же самой среды Builder требуется строить приложения неконсольные, с многооконными графическими интерфейсами, которые и будут предложены читателю в соответствующих местах книги.

Примечание

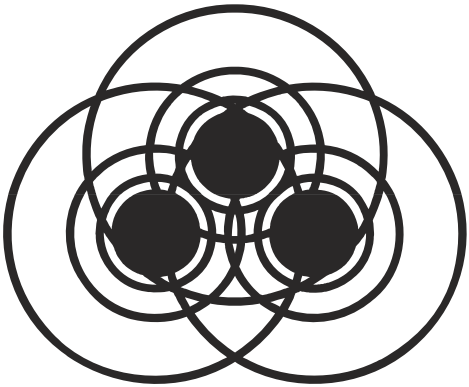
Мы пользуемся тем, что среда Builder позволяет в ее пространстве изучать C (C++), хотя это можно было бы делать и вне среды Builder. Но раз уж такая возможность есть, то почему бы ею не воспользоваться?

Для установки Builder требуется, чтобы ваш компьютер имел следующую конфигурацию:

- тактовая частота процессора Intel Pentium — не ниже 90 МГц;
- операционная система — Windows;
- оперативная память — не менее 32 Мбайт;
- свободное пространство на жестком диске — от 120 до 388 Мбайт в зависимости от параметров установки;
- дисковод CD-ROM;
- видеоадаптер не хуже VGA;
- мышь.

Желаю читателям приятного и плодотворного изучения.

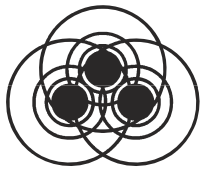
Автор



ЧАСТЬ I

Основы алгоритмических языков C и C++

Глава 1



Типы данных, простые переменные и основные операторы цикла

Цель этой главы — продемонстрировать начальные элементы программирования на языке С. Приложения строятся средой Borland C++Builder в виде специальных конструкций — *проектов*, которые выглядят для пользователя как совокупность нескольких файлов. Причем в проекте консольного приложения файлов меньше, чем в проектах приложений собственно Builder. Это мы увидим, когда начнем делать приложения для Builder.

Программа на языке С — это совокупность функций, т. е. специальных программ, отвечающих определенным требованиям. Запуск любой программы начинается с запуска *главной функции*. Внутри этой главной функции для реализации заданного алгоритма вызываются все другие необходимые функции. Часть функций создается самим программистом, другая часть — *библиотечные функции* — поставляется пользователю со средой программирования и применяется в процессе разработки программ.

Как перейти к созданию консольного приложения

1. Загрузите среду Borland C++Builder.
2. Выполните команды главного меню: **File/New**. Откроется диалоговое окно **New Items**, показанное на рис. 1.1.

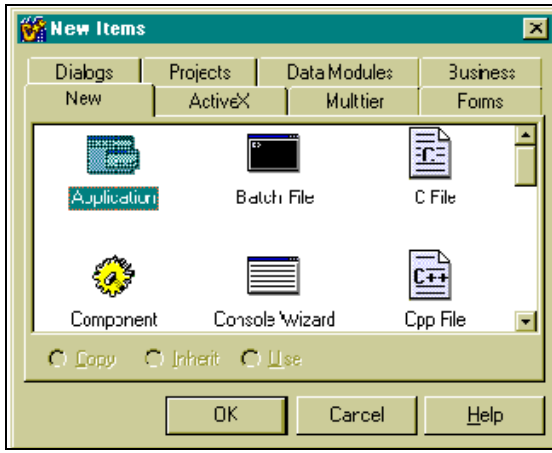


Рис. 1.1. Диалоговое окно **New Items**

В этом окне дважды щелкните кнопкой мыши на значке **Console Wizard** — Мастера построения заготовок консольных приложений. В результате появится диалоговое окно Мастера (рис. 1.2).

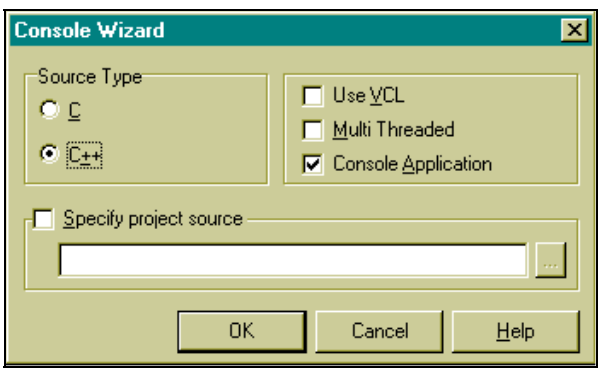


Рис. 1.2. Диалоговое окно Мастера построения заготовок консольных прилоржений

В этом окне активизируйте переключатель **C++**, установите флажок **Console Application** и нажмите **OK**. Мастер сформирует заготовку приложения. Ее вид показан на рис. 1.3.

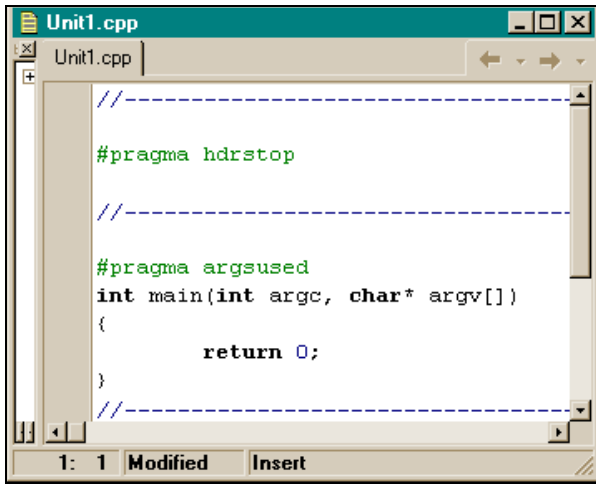


Рис. 1.3. Заготовка консольного приложения

Заготовка состоит из заголовка главной функции `int main(int argc, char* argv[])` и тела, ограниченного фигурными скобками. Преобразуем заголовок функции `main` к виду `main()`, а из тела удалим оператор `return 0`. Все это проделаем с помощью Редактора кода, который открывается одновременно с появлением заготовки консольного приложения на экране: щелкните кнопкой мыши в любом месте поля заготовки, и вы увидите, что курсор установится в месте вашего щелчка. Далее можете набирать любой текст, работать клавишами <Delete>, <Backspace>, клавишами-стрелками и другими необходимыми для ввода и редактирования клавишами.

Мы привели заголовок функции `main` к виду `main()`. Это означает, что наша главная функция не будет иметь аргументов, которые служат для связки консольных приложений. Этим мы заниматься не будем.

Формирование проекта консольного приложения

Теперь, прежде чем заполнять нашу заготовку какими-то кодами, следует сформировать проект консольного приложения, т. к.

приложение в среде Builder существует не само по себе, а в проекте. Для этого снова воспользуемся опцией **File** главного меню. Выполним команду: **File/Save Project As**. Откроется диалоговое окно для сохранения программного модуля заготовки (по умолчанию модулю присваивается имя Unit1, но вы можете дать ему свое имя). Следует выбрать папку, куда вы запишете свой проект, и нажать **OK**. После этого откроется диалоговое окно для сохранения заголовочного модуля проекта (с расширением bpr). Сохраните его, дав ему при необходимости свое имя (по умолчанию заголовочный модуль будет назван Project1). Организационная часть для будущего консольного приложения закончена. Начинаем формировать само приложение, а точнее — его программный модуль Unit1.

Создание простейшего консольного приложения

Запишем в теле функции `main()` следующие две строки:

```
printf("Hello!\n");
getch();
```

Это код нашего первого приложения. Он должен вывести на экран текст "Hello!" и задержать изображение, чтобы оно "не убежало", не исчезло, пока мы рассматриваем, что там появилось на экране. В итоге наше консольное приложение будет иметь вид, представленный на рис. 1.4.

Чтобы приложение заработало, его надо *откомпилировать*, т. е. перевести то, что мы написали текстом на языке C в машинные коды. Для этого запускается программа-компилятор. Запускается она либо нажатием клавиши <F9>, либо выполнением опции главного меню **Run/Run**. Если мы проделаем подобные действия, то получим картину, показанную на рис. 1.5.

На рисунке видно, что наша компиляция не удалась: в нижнем поле окна высветилось сообщение о двух ошибках: "Вызов неизвестной функции". Если кнопкой мыши дважды щелкнуть на каждой строке с информацией об ошибке, то в поле функции `main()`, т. е. в нашей программе, подсветится та строка, в кото-

рой эта ошибка обнаружена. Разберемся с обнаруженными ошибками.

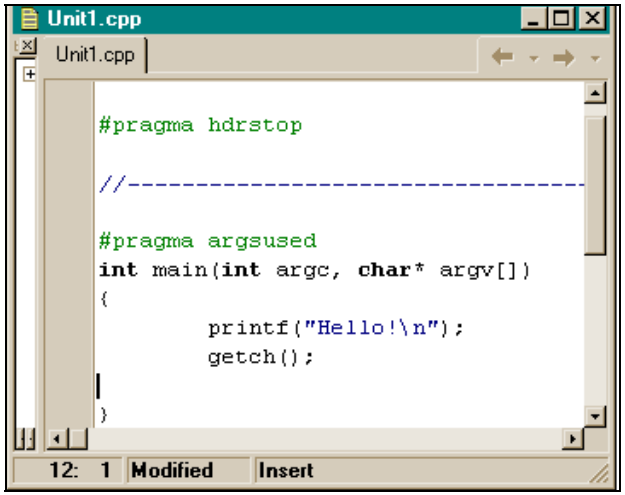


Рис. 1.4. Код консольного приложения до компиляции

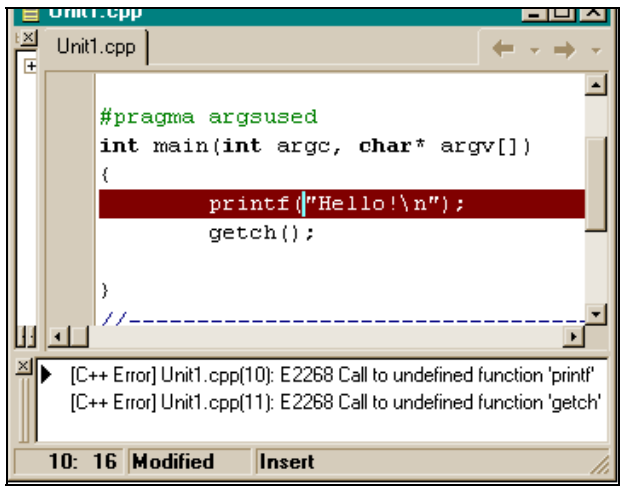


Рис. 1.5. Консольное приложение после неудачной компиляции

Выберем опцию **Help/C++Builder Help** главного меню. Откроется окно помощи. В нем выберем вкладку **Указатель** и в поле **1** наберем имя неизвестной (после компиляции программы) функции `printf`. В поле **2** появится подсвеченная строка с именем набранной в поле **1** функции. Нажмем <Enter>. Откроется окно помощи **Help**, в котором приводятся сведения о функции `printf`, в том числе, в каком файле находится ее описание (Header file — stdio.h), и как включать этот файл в текст программного модуля (`#include <stdio.h>`). `#include` — это оператор компилятора. Он включает в текст программного модуля файл, который указан в угловых скобках. Таким же образом с помощью раздела **Help** найдем, что для неизвестной функции `getch()` к программному модулю следует подключить строку `#include <conio.h>`. После этого текст нашей первой программы будет выглядеть, как на рис. 1.6.

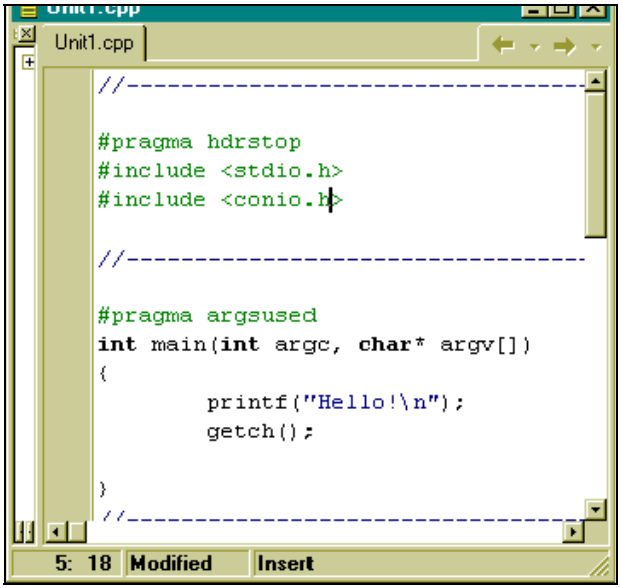


Рис. 1.6. Текст программы
после подключения необходимых библиотек

Запускаем клавишей <F9> компилятор, результат показан на рис. 1.7.

Рис. 1.7. Результат выполнения первой программы

Наша программа успешно откомпилировалась и сразу же выполнилась. В результате ее выполнения в окне черного цвета высветилось слово **Hello!**. Если теперь нажать любую клавишу, программа завершится, и мы снова увидим ее текст. Сохраним новый проект, выполнив опции **File/Save All**.

Поясним суть программы. Мы уже говорили выше, что любая С-программа строится как множество элементов, называемых функциями, — блоков программных кодов, выполняющих определенные действия. Имена этих блоков кодов, построенных по специальным правилам, задает либо программист, если он сам их конструирует, либо имена уже заданы в поставленной со средой программирования библиотеке стандартных функций. Имя главной функции, с которой собственно и начинается выполнение приложения, задано в среде программирования. Это имя — `main()`. В процессе выполнения программы сама функция `main()` обменивается данными с другими функциями и пользуется их результатами. Обмен данными между функциями происходит через *параметры функций*, которые указываются в круглых скобках, расположенных вслед за именем функции. Функция может и не иметь параметров, но круглые скобки после имени всегда должны присутствовать: по ним компилятор узнает, что перед ним функция, а не что-либо другое. В нашем примере две функции, использованные в главной функции `main()`: это функция `printf()` и функция `getch()`.

Функция `printf()` в качестве аргумента имеет строку символов (символы, заключенные в двойные кавычки). Среди символов этой строки есть специальный символ, записанный так: `\n`. Это

так называемый *управляющий символ* — один из первых 32-х символов таблицы кодировки символов ASCII. Управляющие символы не имеют экранного отображения и используются для управления процессами. В данном случае символ `\n` служит для выбрасывания *буфера функции* `printf()`, в котором находятся остальные символы строки, на экран и установки указателя изображения символов на экране в первую позицию — в начало следующей строки. То есть когда работает функция `printf()`, символы строки по одному записываются в некоторый буфер до тех пор, пока не встретится символ `\n`. Как только символ `\n` прочтен, содержимое буфера тут же передается на устройство вывода (в данном случае — на экран).

Функция `getch()` — это функция ввода одного символа с клавиатуры: она ждет нажатия какой-либо клавиши. Благодаря этой функции результат выполнения программы задерживается на экране до тех пор, пока мы не нажмем любой символ на клавиатуре. Если бы в коде не было функции `getch()`, то после выполнения `printf()` программа дошла бы до конца тела функции `main()`, до закрывающей фигурной скобки, и завершила бы свою работу. В результате черное окно, в котором вывелось сообщение **Hello!**, закрылось бы, и мы не увидели бы результата работы программы. Следовательно, когда мы захотим завершить нашу программу, мы должны нажать любой символ на клавиатуре, программа выполнит функцию `getch()` и перейдет к выполнению следующего оператора. А это будет конец тела `main()`. На этом программа и завершит свою работу. Следует отметить, что основное назначение функции `getch()` — вводить символы с клавиатуры и передавать их символьным переменным, о которых пойдет речь ниже. Но мы воспользовались побочным свойством функции — ждать ввода с клавиатуры и, тем самым, не дать программе завершиться, чтобы мы посмотрели результат ее предыдущей работы.

Программа с оператором *while*

Рассмотрим программу вывода таблицы температур по Фаренгейту и Цельсию.

Формула перевода температур такова: $C = (5 : 9) * (F - 32)$, где C — это температура по шкале Цельсия, а F — по шкале Фаренгейта. Задается таблица температур по Фаренгейту: 0, 20, 40, ..., 300. Требуется вычислить таблицу по шкале Цельсия и вывести на экран обе таблицы.

Создаем заготовку консольного приложения и сохраняем его описанным выше способом (как в простейшей программе, которую мы разработали выше).

Записываем код новой программы в тело главной функции (листинг 1.1).

Листинг 1.1

```
//---------------------------------------------------------------

#pragma hdrstop

#include <stdio.h>
#include <conio.h>

//---------------------------------------------------------------

main()
{
  int lower,upper,step;
  float fahr,cels;
  lower=0;
  upper=300;
  step=20;
  fahr=lower;
  while(fahr <= upper)
   {
    cels=(5.0/9.0)*(fahr-32.0);
    printf("%4.0f %6.1f\n",fahr,cels);
```

```
    fahr=fahr+step;
   }
  getch();

}
//---------------------------------------------------------
```

Запускаем компилятор клавишей <F9>. Программа откомпилируется и выполнится. Результат высветится в окне (рис. 1.8).

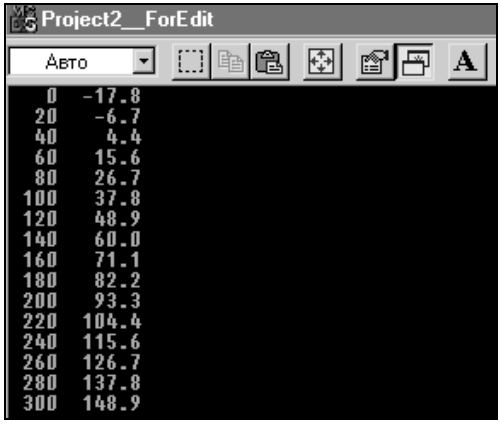


Рис. 1.8. Результат расчета таблицы температур по Цельсию

Имена и типы переменных

Поясним суть программы.

`int lower,upper,step;` — это так называемые "объявления переменных". `lower,upper,step` — имена переменных. Компилятор соотнесет с этими именами определенные адреса в памяти и, начиная с этих адресов, выделит участки памяти в соответствии с тем, какого типа объявлены переменные. В нашем случае тип переменных, заданный при их объявлении, — `int` (от англ. *integer* — целое число). Это означает, что все переменные имеют вид "целое число со знаком" и что под каждое значение числа,

которое будет записано на участках `lower`, `upper` или `step`, отведено по 2 байта. Таким образом, имена переменных — это названия тех "полочек" в памяти компьютера (а каждая "полочка" имеет свой адрес), где будут находиться данные (числа и не числа), с которыми программа будет работать при реализации алгоритма.

Имена переменным надо давать осмысленно — так, чтобы они отражали характер содержания переменной. В нашем случае `lower`, `upper` и `step` именуют соответственно нижнюю и верхнюю границы таблицы температур по Фаренгейту и шаг этой таблицы. Нижняя граница таблицы (`lower`) равна 0, верхняя (`upper`) — 300, а шаг таблицы (т. е. разность между соседними значениями — `step`) равен 20.

Перечень описываемых переменных одного типа (тип указывается в начале перечня) обязательно должен оканчиваться *точкой с запятой* — сигналом для компилятора, что описание переменных данного типа завершено. В языке C выражение, после которого стоит точка с запятой, считается *оператором*, т. е. законченным действием. В противном случае компилятор станет при компиляции искать ближайшую точку с запятой и объединять все, что до нее находится, в один оператор (объединятся разнородные данные) и, в конце концов, выдаст ошибку компиляции.

`float fahr,cels;` — описание переменных с именами `fahr`, `cels`, но тип этих переменных уже иной (`float`). Эти переменные — не целые числа, а так называемые числа "с плавающей точкой". "Полочки" в памяти, обозначаемые этими переменными, могут хранить любые вещественные числа, а не только целые. Под этот тип данных компилятор отводит по 4 байта.

Таким образом, перед составлением программы, которая будет оперировать данными (числовыми и нечисловыми), *эти данные следует описать*: им должны быть присвоены типы и имена. Присвоение переменным типов и имен фактически означает, что компилятор определит им место в памяти, куда данные будут помещаться и откуда будут извлекаться при выполнении операций над ними. Следовательно, когда мы пишем $c = a + b$, это означает, что одна часть данных будет извлечена с "полочки" с именем a, другая часть данных — с "полочки" с именем b,

произойдет их суммирование, и результат будет "положен" (записан) на "полочку" с именем *c*. Знак = означает "присвоить", это не знак равенства, а операция пересылки. Знак равенства выглядит иначе *(о знаке равенства см. в главе 2)*. Присваивать некоторой переменной можно не только значение с какой-либо "полочки", т. е. значение другой переменной, но и просто числа. Например, $a = 10$. В этом случае компилятор просто "положит на полочку" *a* число 10.

Оператор *while*

Чтобы вычислить температуру по Цельсию для каждого значения шкалы по Фаренгейту, не требуется писать программный код для каждой точки шкалы. В этом случае никакой памяти не хватило бы, поскольку шкала может содержать миллиарды точек. В таких случаях выходят из положения так: вычисляют для одной точки, используя некоторый параметр, а потом, изменяя этот параметр, заставляют участок расчета снова выполняться до тех пор, пока параметр не примет определенного значения, после которого повторение расчетов прекращают. Повторение расчетов называют *циклом расчетов*. Для организации циклов существуют специальные операторы цикла, которые "охватывают" участок расчета и "прокручивают" его необходимое количество раз. Одним из таких операторов в языке C является оператор `while` (англ. — до тех пор, пока). Тело этого оператора ограничивается парой фигурных скобок: начинается с открывающей фигурной скобки, а заканчивается закрывающей фигурной скобкой. В это-то тело и помещается прокручиваемый участок. А сколько раз "прокручивать" определяется *условием окончания цикла*, которое задается в заголовочной части оператора. Вид оператора `while` таков:

```
while(условие окончания цикла)
{
  Тело
}
```

Работает оператор так: в начале проверяется условие окончания цикла. Если оно истинно, то тело оператора выполняется. Если

условие окончания цикла ложно, то выполнение оператора прекращается, и начинает выполняться программный код, расположенный непосредственно после закрывающей скобки тела оператора.

Приведем пример истинности условия. Условие может быть записано в общем случае в виде некоторого выражения (переменные, соединенные между собой знаками операций). Например, $a < b$ (*a* меньше *b*). Значение переменной *a* — это то, что лежит на "полочке" с именем *a*, а значение переменной *b* — то, что лежит на "полочке" *b*. Если значение переменной *a* действительно меньше значения *b*, то выражение считается истинным, в противном случае — ложным.

Внимательно посмотрев на оператор `while`, можно сделать вывод: для завершения цикла (для этого условие окончания цикла должно стать ложным), надо, чтобы само условие окончания изменялось в теле оператора по мере выполнения цикла и в нужный момент стало бы ложным. Теперь рассмотрим, как это происходит в нашей программе.

Сперва определяются начальные значения переменных `lower`, `upper`, `step`. Параметром, задающим цикл, у нас является переменная `fahr`: ее значение будет меняться от цикла к циклу на величину шага шкалы по Фаренгейту, начиная от минимального, когда `fahr` = `lower` (мы присваиваем ей значение переменной `lower`, которая ранее получила значение ноля — начала шкалы по Фаренгейту), и заканчивая максимальным, когда значение переменной `lower` достигнет значения переменной `upper`, которое мы в начале указали равным 300. Поэтому условие окончания цикла в операторе цикла `while` будет таковым: "пока значение `fahr` не превзойдет значения переменной `upper`". На языке С это записывается в виде:

```
while(fahr <= upper)
```

В теле же самого оператора цикла мы записываем на языке С: формулу вычисления значения переменной `cels` (т. е. точки шкалы по Цельсию), функцию `printf()` для вывода значений точек по Фаренгейту и Цельсию, переменную `fahr` для изменения параметров цикла: она добавляет значение шага шкалы по Фаренгейту, что подготавливает переход к вычислению переменной `cels`

для нового значения переменной `fahr`. Это произойдет тогда, когда программа дойдет до выполнения конца тела оператора `while` (т. е. до закрывающей фигурной скобки) и перейдет к выполнению выражения, стоящего в заголовочной части `while` и проверке его на истинность/ложность. Если истинность выражения-условия не нарушилась, начнет снова выполняться тело оператора `while`. Когда же переменная `fahr` примет значение больше значения `upper`, цикл завершится: начнет выполняться код, следующий за телом оператора `while`. А это будет функция `getch()`, которая потребует ввода символа с клавиатуры, тем самым задерживая закрытие окна, где благодаря функции `printf()` появились результаты работы программы. Как только мы нажмем на любую клавишу, функция `getch()` получит то, что ждала, в результате чего она завершится. Затем начнет выполняться закрывающая скобка тела главной функции `main()`. После ее обработки наше приложение окончит свою работу.

Поясним операции, примененные при формировании переменной `cels`. Это арифметические операции деления (/), умножения (*), вычитания (-). У операции деления есть одна особенность: если ее операнды имеют тип `int`, то результат деления — всегда целое число, т. к. в этом случае остаток от деления отбрасывается. Поэтому, если бы мы в формуле для вычисления переменной `cels` записали 5/9, то получили бы 0, а не 0,55. Чтобы этого не случилось нам пришлось "обмануть" операцию деления: мы записали 5.0/9.0, как будто операнды — числа с плавающей точкой. Для таких операндов остаток от деления не отбрасывается.

Функция `printf()` в общем случае имеет такой формат:

```
printf(Control,arg1,arg2,…,argN);
```

`Control` — это строка символов, заключенных в двойные кавычки, `arg1,arg2,…,argN` — имена переменных, значения которых должны быть выведены на устройство вывода. Строка `Control` содержит в себе данные двух родов: указания на тип переменных `arg1,arg2,…,argN` (указания на тип расположены в том же порядке, что и переменные `arg1,arg2,…,argN`) и остальные символы, которые выводятся без всякого форматирования (т. е. без преобразования в другую форму).

Обозначение типа переменной всегда начинается с символа `%`, а заканчивается символом типа форматирования: `d` — для переменных типа `int`, `f` — для `float`, `s` — для строк символов и т. д.

Между символом `%` и символом типа форматирования задается ширина поля вывода, количество знаков после точки (для типа `f`) и т. д. Полное определение типов переменных можно посмотреть в разделе **Help** таким же образом, как ранее мы искали описания функций. Так как переменные `cels` и `fahr` относятся к типу `float`, то и в функции `printf()` указан соответствующий формат — `f`. Значение переменной `fahr` — целое число, занимающее 4 байта, а значение переменной `cels` — дробное число с одним знаком после точки, занимающее 6 байт.

Оператор *for*

Кроме оператора `while`, цикл позволяет организовать оператор `for`. Перепишем программу расчета температур, рассмотренную выше, в несколько другом виде (листинг 1.2).

Листинг 1.2

```
//---------------------------------------------------------

#pragma hdrstop

#include <stdio.h>
#include <conio.h>

//---------------------------------------------------------

main()
{
  int fahr;
  for(fahr=0; fahr <= 300; fahr= fahr + 20)
```

```
   printf("%4d  %6.1f\n",fahr,(5.0/9.0)*(fahr-32.0));
  getch();

}
//-----------------------------------------------------------
```

Здесь для получения того же результата, что и в предыдущем случае, применен оператор цикла `for`. Тело этого оператора, как и тело оператора `while`, циклически выполняется ("прокручивается"). В нашем случае тело `for` состоит всего из одного оператора — `printf()`, поэтому такое тело не берется в фигурные скобки (если бы тело оператора `while` состояло только из одного оператора, оно тоже не бралось бы в скобки).

Мы видим, что запись программы приобрела более компактный вид. В заголовочной части оператора `for` расположены три выражения, первые два из которых оканчиваются точкой с запятой, третье — круглой скобкой, обозначающей границу заголовочной части `for` (для компилятора этого достаточно, чтобы понять, что третье выражение завершилось). Как говорят, в данном случае "цикл идет по переменной `fahr`": в первом выражении она получает начальное значение, второе выражение — это условие окончания цикла (цикл закончится тогда, когда `fahr` примет значение большее 300), а третье выражение изменяет параметр цикла на величину шага цикла.

Работа идет так: инициализируется переменная цикла (т. е. получает начальное значение), затем проверяется условие продолжения цикла. Если оно истинно, то выполняется сначала тело оператора (в данном случае, функция `printf()`), затем управление передается в заголовочную часть оператора `for`, после чего вычисляется третье выражение (изменяется параметр цикла) и проверяется значение второго выражения: если оно истинно, то выполняется тело, затем управление снова передается на вычисление третьего выражения и т. д. Если же второе выражение становится ложным, то выполнение оператора `for` завершается и начинает выполняться оператор, следующий непосредственно за ним. А это — завершающая фигурная скобка `main()`, после чего функция `main()` завершается.

В данном примере следует обратить внимание на аргумент функции `printf()`: вместо обычной переменной там стоит целое выражение, которое сначала будет вычислено, а потом его значение выведется на устройство вывода. Выражение можно указывать в качестве аргумента функции, исходя из правила языка C: *"В любом контексте, в котором допускается использование переменной некоторого типа, можно использовать и выражение этого же типа"*.

Символические константы

Задание конкретных чисел в теле программы — не очень хороший стиль программирования, т. к. такой подход затрудняет дальнейшую модификацию программы и ее понимание. При создании программы надо стремиться задавать все конкретные данные в начале программы, используя специальный оператор компилятора `#define`, который позволяет соотнести с каждым конкретным числом или выражением набор символов — *символических* (не символьных! символьные — это другое) *констант*. В этом случае на местах конкретных чисел в программе будут находиться символические константы, которые в момент компиляции программы будут заменены на соответствующие им числа, — но это уже невидимо для программиста. Отсюда и название "символические константы": это не переменные, которые имеют свой адрес и меняют свое значение по мере работы программы, а постоянные, которые один раз получают свое значение и не меняют его. С учетом сказанного наша программа из листинга 1.2 примет следующий вид (листинг 1.3).

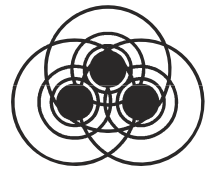
Листинг 1.3

```
#pragma hdrstop

#include <stdio.h>
#include <conio.h>
#define lower 0
#define upper 300
#define step   20
```

```
//-------------------------------------------------------------

main()
{
  int fahr;
  for(fahr=lower; fahr <= upper; fahr= fahr + step)
   printf("%4d  %6.1f\n",fahr,(5.0/9.0)*(fahr-32.0));
  getch();
}
//-------------------------------------------------------------
```

Теперь, когда начнется компиляция, компилятор просмотрит текст программы и заменит в нем все символические константы (в данном случае это: `lower`, `upper`, `step`) на их значения, заданные оператором `#define`. Заметим, что после этого оператора никаких точек с запятой не ставится, т. к. это оператор не языка C, а компилятора. И если нам понадобится изменить значения переменных `lower`, `upper`, `step`, нам не придется разбираться в тексте программы, а достаточно будет посмотреть в ее начало, быстро найти изменяемые величины и выполнить их модификацию.

Глава 2



Программы для работы с символьными данными

Рассмотрим некоторые полезные программы для работы с символьными данными, использующие операторы условного перехода. *Символьные данные* в языке С имеют тип `char`: переменная, которая будет содержать один символ (точнее — его код по таблице кодирования символов ASCII), должна описываться как `char c`.

Примечание

Здесь `c` — это обычное имя переменной, вместо которого можно было бы написать, например, `abcde`. Компилятор присвоит имени необходимый адрес, если количество символов в имени не превышает допустимое.

В библиотеке С наряду с функциями `getch()` и `printf()`, с которыми мы познакомились ранее, существуют и другие функции ввода-вывода. Две из них — это `getchar()` и `putchar()`. К переменной `char c;` эти функции применяются так:

```
c=getchar(); putchar(c);
```

Первая функция не имеет параметров — в круглых скобках ничего нет. Функция `getchar()`, начав выполняться, ожидает ввода символа с клавиатуры. Как только символ с клавиатуры введен, его значение присваивается переменной `c`.

Если говорить точнее, функция `getchar()` работает несколько иначе (это можно посмотреть в разделе **Help**). Фактическая об-

работка символа начнется только тогда, когда ввод закончится нажатием клавиши <Enter>: до этого вводимые символы накапливаются в буфере функции. Поэтому если присвоение символов происходит в некотором цикле (например, если мы хотим сформировать строку из символов, вводя их через `getchar()`, по одному символу в цикле), то следует ввести строку символов, нажать <Enter>, и только тогда мы увидим результат ввода: строка будет разобрана на символы, введенные функцией.

Тут возникает вопрос: а как дать знать программе, что ввод группы символов закончен? Ввод данных, как мы в дальнейшем увидим, может осуществляться из файла. В этом случае на окончание ввода указывает так называемый *признак конца файла*, который обнаруживается с помощью специальных данных (EOF — End Of File), определенных в соответствующем h-файле.

Примечание

В h-файлах описывают данные, которые используются функциями, применяемыми в программах, а также стандартные переменные, необходимые при составлении программ.

Но как быть с признаком конца файла при вводе данных с клавиатуры? Можно пользоваться EOF-данными, нажимая клавиши <Ctrl>+<z>. При этом выдается код 26, соответствующий EOF. Но можно и самостоятельно формировать такой признак.

Как мы теперь знаем, для того, чтобы функция `getchar()` приступила к вводу каждого набранного на клавиатуре символа, надо после набора группы символов (т. е. строки) нажать клавишу <Enter>. Нажатию этой клавиши соответствует символ `\n` — символ перехода на новую строку. Если мы хотим ограничиться вводом строки, то признаком конца ввода файла можно считать `\n`. Если же мы хотим ввести группу строк и после этого дать знать программе, что ввод завершен, то в качестве признака конца файла можно использовать некий управляющий код (из диапазона 0—31 таблицы ASCII). Таким управляющим кодом может быть, например, 27, соответствующий клавише <Esc>. После ввода последнего символа последней строки и нажатия

<Enter>, чтобы функция `getchar()` начала обработку введенных символов, требуется нажать <Esc>.

Если обозначить символическую константу, задающую значения признака конца ввода с клавиатуры через EOF, то, воспользовавшись оператором `#define`, в начале программ ввода данных с клавиатуры мы можем написать:

```
#define eof 27 // признак конца ввода символов
с клавиатуры
```

Здесь `//` — это признак комментария в программе на C. Такой знак ставится, если текст комментария занимает только одну строку. Если же ваш комментарий более длинный, то следует воспользоваться другой формой задания комментария — скобками `/* */`. Весь текст между этими скобками можно располагать на многих строках. Например,

```
/* признак конца ввода
   символов с клавиатуры */
```

Приступим к составлению простейших программ работы с символьными данными.

Программа копирования символьного файла. Вариант 1

Напишем программу, в которой входной файл будет вводиться с клавиатуры (входное стандартное устройство — клавиатура), а выводиться на экран (выходное стандартное устройство — экран). Текст программы представлен в листинге 2.1.

Листинг 2.1

```
//---------------------------------------------------------
#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof  27            //признак конца файла
```

```
//-------------------------------------------------------------
int main()
{
 int c;
 printf("Make input>\n");
 c=getchar();
 while(c != eof)
   {
    putchar(c);
    c=getchar();
   }
getch();  /*вводит символ, но без эхо-сопровождения
(для организации задержки экрана) */
//-------------------------------------------------------------
```

Переменная `c`, в которую вводится один символ, описана как `int`, а не как `char`. Почему? Дело в том, что в языке C типы `char` и `int` взаимозаменяемы (переменная `c` фактически содержит код символа, т. е. некоторое число). С другой стороны, чтобы определить момент наступления конца ввода с клавиатуры, мы должны сравнить содержимое переменной `c` с числом `eof` или, когда применяем EOF-данные, то с числом 26. Именно поэтому `c` объявлена как `int`.

Функция `printf()` выводит на экран запрос на ввод. Далее вводится первое значение переменной `c` (как мы договаривались выше, набираем на клавиатуре строку и нажимаем <Enter>, тогда `getchar()` начинает обработку по одному символу). Потом с помощью оператора цикла `while()` начинается циклическая обработка ввода-вывода символов. Пока условие в заголовочной части `while()` выполняется (т. е. пока мы не нажали <Esc>), выполняется тело оператора `while()`: все операторы, находящиеся внутри фигурных скобок — `putchar(c)` выводит введенный символ на экран, `c=getchar()` вводит новый символ в качестве значения переменной `c`. После этого программа доходит до конца тела оператора `while()` — закрывающей фигурной скобки — и снова возвращается в его заголовочную часть, где начи-

нает проверять выполнение записанного там условия. Если условие истинно, т. е. введенный в качестве значения переменной `c` символ — не код клавиши <Esc>, то снова выполняются операторы, находящиеся в теле `while()`.

Как только мы нажмем клавишу <Esc>, в переменной `c` появится ее код — число 27. Поскольку условие выполнения оператора `while()` нарушится (значение переменной `c` станет равно `eof`), управление будет передано следующему после `while()` оператору. Это будет функция `getch()`, которая потребует ввода символа с клавиатуры. Пока мы не нажмем какую-либо клавишу, `getch()` будет ждать ввода, а мы в это время будем рассматривать картинку на экране и гадать: что же у нас получилось? Если мы нажмем на любую клавишу, то функция `getch()` облегченно вздохнет, и наша программа завершится.

Следует кое-что уточнить: независимо от того, вывели мы символ на экран или нет, он будет отображен на экране функцией `getchar()`. В таком случае говорят, что `getchar()` работает с *эхо-сопровождением*. Поэтому когда сработает функция `putchar(c)`, то может показаться, что она повторно выведет введенный символ. На самом деле это не так, двойник символа на экране строкой выше появится из-за излишней плодовитости `getchar()`. Функция же `getch()` работает без эхо-сопровождения.

На рис. 2.1 приведен результат работы нашей программы.

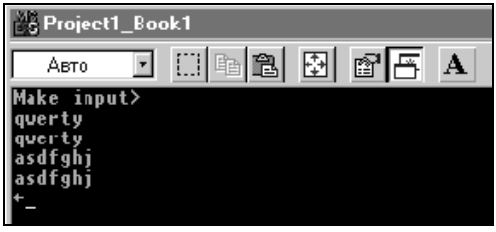


Рис. 2.1. Результат работы программы листинга 2.1

Программа копирования символьного файла. Вариант 2

Ранее мы познакомились с правилом C: вместо переменной некоторого типа можно использовать и выражение этого же типа. Воспользуемся этой возможностью и запишем нашу программу в другом виде (листинг 2.2).

Листинг 2.2

```
//---------------------------------------------------------

#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof  27       //признак конца файла

//---------------------------------------------------------

int main()
{
 int c;
 printf("Make input>\n");
 while((c=getchar())!= eof)
    putchar(c);
 getch();  //Вводит символ, но без эхо-сопровождения
}
//---------------------------------------------------------
```

Ввод символа мы внесли в заголовочную часть `while`, поскольку `(c=getchar())` — это выражение того же типа, что и `c`. Оператор `while` в общем случае работает так: он сначала вычисляет выражение, которое находится в его заголовочной части, при этом выполняется ввод символа с клавиатуры — что нам и нужно. Затем оператор `while` проверяет, что введенные символы не являются признаком конца файла. Так как в отличие от предыду-

щего варианта тело оператора `while` состоит только из одного оператора `putchar(c)`, фигурные скобки можно опустить. После того как выполнится `putchar(c)`, управление будет передано в заголовочную часть оператора `while`, где снова начнется вычисление выражения, которое, в свою очередь, потребует ввода символа с клавиатуры и т. д.

Подсчет символов в файле. Вариант 1

Напишем программу, в которой файл будет вводиться с клавиатуры. Программа представлена в листинге 2.3.

Листинг 2.3

```
//---------------------------------------------------------

#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof  27       //признак конца файла

//---------------------------------------------------------

int main()
{
 long nc;
 nc=0;
 printf("Make input>\n");
 while(getchar()!= eof)
       nc++;
 printf("Characters's number is: %ld\n",nc);
 getch();  //Вводит символ, но без эхо-сопровождения
}
//---------------------------------------------------------
```

Здесь мы встречаемся с новым типом данных: `long` — длинное целое. Этот тип применяется для описания больших целых чисел со знаком (для переменных типа `long` отводится в два раза больше памяти, чем для переменных типа `int`). Так как мы подсчитываем количество символов в файле, который в общем случае может быть и не "клавиатурный", то мы должны быть готовы к тому, что в нем будет много символов, и их число превысит допустимое значение переменной типа `int`. Количество вводимых символов подсчитывается в переменной `nc` по правилу: ввели один символ — значение `nc` увеличивается на единицу, ввели еще один — снова `nc` увеличивается на единицу и т. д. В начале каждого выполнения программы значение `nc` обнуляется.

Далее происходит запрос на ввод символов (текст выводится на экран). Затем опять с помощью цикла `while` организуется ввод символов до тех пор, пока не будет нажата клавиша <Esc>. Заметим, что тело оператора `while`, как и в предыдущем примере, содержит только одно выражение, поэтому фигурные скобки, ограничивающие тело оператора, излишни. Поскольку тело оператора составляет функция `getchar()`, то, когда оператор `while` начнет вычислять выражение, потребуется ввод символа с клавиатуры. После того как мы введем символ, значение `getchar()` станет равным коду введенного символа: эта функция всегда выдает код символа, который мы нажимаем на клавиатуре. Поэтому нет необходимости значение `getchar()` еще присваивать какой-либо переменной, тем более, что сам введенный символ по нашему алгоритму не требуется: значение `getchar()` сразу сравнивается с признаком конца файла. Если этот признак еще не введен (мы не нажали <Esc>), то условие выполнения оператора `while` не нарушается, и тело оператора `while`, состоящее всего из одного оператора `nc++`, начинает выполняться опять.

Новый оператор `nc++` равносилен выражению `nc=nc+1`. То есть к значению `nc` добавляется единица. Кстати, можно в данном случае писать и `++nc`: пока операция `++` не участвует в выражении типа `int x=nc++` или `int x=++nc`, это не имеет никакого значения. В последних же случаях положение символов `++` существенно: если они находятся до `nc`, то сначала операция выполня-

ется, а потом уже результат присваивается переменной x. Если — после nc, то сначала содержимое nc присваивается переменной x, а потом уже nc изменяется на единицу. Этой операции родственна операция --, которая себя ведет, как и ++, но не прибавляет, а вычитает единицу.

В функции printf() мы снова видим новый формат: %ld. В этом формате выводятся числа типа long.

Результат работы программы приводится на рис. 2.2.

Рис. 2.2. Результат работы программы листинга 2.3

Подсчет символов в файле. Вариант 2

Листинг 2.4

```
//---------------------------------------------------------------

#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof  27       //признак конца файла

//---------------------------------------------------------------

int main()
{
```

```
 double nc;
 printf("Make input>\n");
 for(nc=0; getchar() !=eof; nc++)
        ;
 printf("Characters's number is: %0.f\n",nc);
 printf("Characters's number is: %f\n",nc);
 getch();  //Вводит символ, но без эхо-сопровождения
}
//-------------------------------------------------------------
```

Для подсчета символов в файле мы применили переменную нового типа `double`. Этот тип данных занимает в два раза больше памяти, чем `float`. Накапливать по единице мы можем в любой числовой переменной, а не только в переменной типа `int`. Для обеспечения цикличности ввода символов в этой программе использован оператор цикла `for`, работу которого мы ранее рассматривали. В его заголовочной части имеются три выражения:

- инициализирующее выражение (`nc=0`) — в этом выражении задаются начальные значения переменных, которые будут участвовать в цикле (в нашем случае — переменная `nc`);
- выражение, определяющее условие окончания цикла (`getchar() !=eof`) — в нашем случае цикл окончится, когда будет нажата клавиша <Esc>;
- выражение, определяющее изменение переменной, по которой, как говорят, "идет цикл" — в нашем случае `nc++`.

Поскольку все удалось разместить в заголовочной части, в теле оператора `for` нет необходимости. Но формат оператора необходимо соблюсти. Поэтому тело `for` все-таки задано, но задано так называемым "пустым" оператором — точкой с запятой, которая стоит сама по себе.

Работа программы будет происходить так: сначала сработает первое заголовочное выражение и переменная цикла `nc` обнулится. Затем начнет проверяться на истинность/ложность второе заголовочное выражение. Но чтобы его проверить, надо выполнить функцию `getchar()`, т. е. нажать клавишу и ввести символ. Только после этого `getchar()` получит значение, и это значение

будет сравниваться с признаком конца файла. Если проверяемое выражение истинно, то начнет выполняться тело оператора `for`. Поскольку оно пусто, то управление передастся на вычисление третьего заголовочного выражения `nc++`. Для одного введенного символа в `nc` добавится одна единица. После этого управление передастся на вычисление второго заголовочного выражения, т. е. придется ввести следующий символ, который будет проверен на признак конца файла. Если введенный символ не код клавиши <Esc>, то снова будет выполнятся тело, и т. д. В конце концов, когда мы нажмем <Esc>, второе заголовочное выражение нарушится и оператор `for` будет пропущен, управление передастся на следующий за телом `for` оператор.

После этого начнет выполняться оператор вывода `printf()`. Мы привели в программе два таких оператора, чтобы показать различие в форматах вывода переменной типа `double`, которая всегда выводится в формате `f`. Если формат задан как `%0.f`, то дробная часть, которая является свойством чисел с плавающей точкой, будет отброшена и число выведется как целое. Если же задать формат в виде `%f`, не указав количество цифр в дробной части, то после точки выведется столько цифр, сколько определено по умолчанию.

Результат работы программы представлен на рис. 2.3.

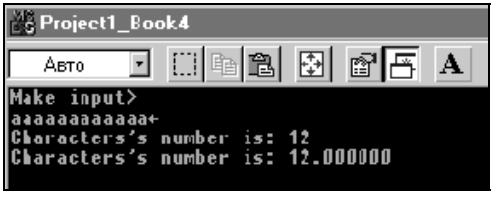


Рис. 2.3. Результат работы программы листинга 2.4

Подсчет количества строк в файле

Строки файла строк в языке C разделяются символом `\n`, поэтому программа ввода строк с клавиатуры и подсчета их количества будет выглядеть так, как показано в листинге 2.5.

Листинг 2.5

```
//---------------------------------------------------------

#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof  27       //признак конца файла

//---------------------------------------------------------

int main()
{
  int c,nl;
  nl=0;
  printf("Enter your string and press the key <Enter> >\n");
  while((c=getchar()) !=eof)
   if(c=='\n')
    nl++;
 printf("String's number is: %d\n",nl);
 getch();  //Вводит символ, но без эхо-сопровождения
}
//---------------------------------------------------------
```

Здесь новое по сравнению с предыдущими подобными программами только то, что появилась операция == (равно) и новый оператор `if`. Это оператор условного перехода: изменяет последовательное (сверху вниз) выполнение операторов программы в зависимости от истинности/ложности условия (оно записывается в круглых скобках в заголовочной части оператора и может представлять собой выражение). Если условие истинно, то выполняется тело оператора, которое обладает точно такими же свойствами, что и тела операторов `while` и `for`: если в теле всего один оператор, то этот оператор может не заключаться в фигурные скобки, в противном случае фигурные скобки обязательны. В нашем случае тело состоит из одного оператора `nl++`, который

выполняется всякий раз, когда введен символ конца строки. В противном случае тело `if` не выполняется. Тело оператора `while` тоже состоит из одного оператора `if` (неважно, сколько операторов включает тело `if`), поэтому `while` записан без фигурных скобок.

Программа работает так: обнуляется счетчик количества вводимых строк (`nl`), начинает выполняться оператор цикла `while`, обеспечивающий ввод с клавиатуры потока символов (вычисляется, как обычно, выражение в заголовочной части `while`, чтобы проверить условие на истинность/ложность, что требует нового ввода символа). Среди потока символов встречаются символы `\n`, сигнализирующие об окончании строки. Как только такой символ обнаруживается с помощью оператора `if`, в счетчик `nl`, расположенный в теле `if`, добавляется единица. Когда после последней строки, завершающейся символом `\n`, мы нажмем <Esc>, ввод строк завершится. Условие выполнения оператора `while` нарушится, и управление будет передано на оператор, следующий за его телом. Это будет оператор вывода `printf()`. Результат работы программы представлен на рис. 2.4.

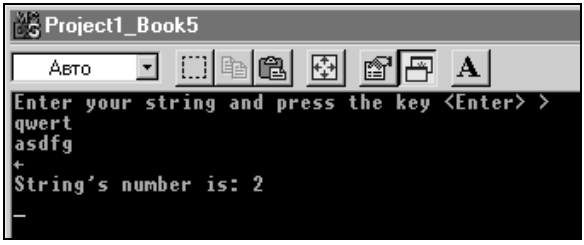


Рис. 2.4. Результат работы программы листинга 2.5

Подсчет количества слов в файле

Договоримся, что слово — это любая последовательность символов, не содержащая пробелов, символов табуляции (`\t`) и новой строки (`\n`). Наряду с количеством слов программа будет подсчитывать количество символов и строк.

Текст программы приведен в листинге 2.6.

Листинг 2.6

```
//---------------------------------------------------------

#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof 27        //признак конца файла

#define yes 1       // для придания значения переменной in
#define no  0      // для придания значения переменной in

//---------------------------------------------------------

int main()
{
  int c;         //для ввода символа
  int nc;        //для подсчета количества введенных символов
  int nl;       //счетчик строк
  int nw;      //счетчик слов
  int in;     /*флажок слежения за тем, находится ли в данный
                момент программа внутри слова или нет*/
  nc=nl=nw=0;      //обнуление счетчиков
  in=no;          // до ввода находимся вне слова
  printf("Enter your strings and press the key <Enter> >\n");
  while((c=getchar()) !=eof)
   {
    if(c != '\n')       //если символ - не конец строки
     nc++;           /*какой бы символ ни ввели (кроме кода
                       <Esc> и
                       '\n'), его надо учитывать в счетчике*/
     else      //иначе… (если введенный символ - конец строки)
```

```
      nl++;  /*Здесь c=='\n' и поэтому сколько раз нажали
               <Enter>, столько будет и строк*/
if(c==' ' || c=='\n' || c=='\t') //если символ хотя
                                 //бы один из…
     in=no;
/*сколько бы раз ни нажимали на клавиши "пробел","конец стро-
ки", "табуляция", всегда будем находиться вне слова*/
else if(in==no) /*сюда попадаем только тогда, когда нажали
любую клавишу, кроме пробела, <Enter> и конца строки*/
 {
   in=yes; /*если до этого мы были вне слова (in==no), то
                сейчас попали на начало слова*/
   nw++;       //и слово надо учесть в счетчике
 }
else    // иначе… если (in != no)
  ;     /*эта часть выполняется, когда мы, находясь внутри
слова (in !=no), ввели любой символ, кроме пробела, знака
табуляции и знака конца строки. В этом случае подсчет слов
не ведется, а программа возвращается на ввод следующего
символа*/
   } // закрывающая скобка оператора while()
    printf("Strings.…..=%d\n",nl);
    printf("Words.....…=%d\n",nw);
    printf("Characters..=%d\n",nc);
 getch();  /*Вводит символ, но без эхо-сопровождения (задер-
живает отображение результатов расчетов на экране) */
} //закрывающая скобка функции main()
//-------------------------------------------------------------
```

Весь ход работы программы ясен из подробного комментария. Стоит обратить внимание на некоторые нововведения.

- `nc=nl=nw=0;` Так можно писать, потому что операция "присвоить" (=) выполняется справа налево. Поэтому сначала выполнится `nw=0`, потом выражение `nl=nw`, уже равное нулю, затем выполнится `nc=nl`.
- Появился оператор `else`. Это — необязательная часть оператора `if`. Если условие в скобках `if` ложно и нет части `else`,

то тело оператора `if` не выполняется, а начинает работать следующий за `if` оператор. Если есть необязательная часть `else` и условие `if` ложно, то выполняется тело оператора `else` (тело `else` обладает такими же свойствами, как и тело `if`). Если условие `if` истинно, то выполняется тело этого оператора, а оператор `else` пропускается.

- Появилась комбинация `else if`. Она работает точно так же, как и оператор `if`: если в ее скобках условие выполняется, то выполняется ее тело, в противном случае тело пропускается.
- Появилась логическая операция `||` (ИЛИ) или операция дизъюнкции. Это — бинарная операция, результат которой истинен, когда истинен хоть один из операндов. В противоположность ей существует бинарная логическая операция `&&` (И) или операция конъюнкции. Ее результат истинен только тогда, когда оба операнда истинны. Если хотя бы один из них ложен, то ложен и результат.
- У последнего оператора `else` тело состоит из одного (пустого) оператора. Этот оператор `else` можно было бы опустить — он поставлен для лучшего понимания картины.

Результат работы программы показан на рис. 2.5.

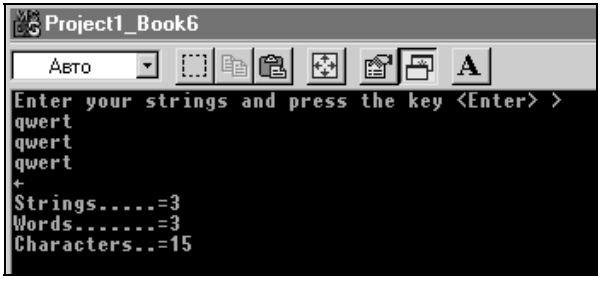
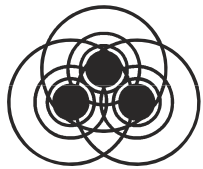


Рис. 2.5. Результат работы программы листинга 2.6

Глава 3



Работа с массивами данных

Одномерные массивы

Создадим программу, которая вводит файл с клавиатуры и подсчитывает, сколько раз в нем встречается каждая из цифр от 0 до 9. Если использовать тот опыт, который мы получили в предыдущих главах, то для составления такой программы нам потребуется объявить десять счетчиков, в каждом из которых мы станем накапливать данные о том, сколько раз встретилась каждая цифра. Первый счетчик — для накапливания данных о количестве имеющихся в файле нолей, второй — единиц и т. д. А представим себе, что нам надо подсчитать количество каких-то объектов, которых, скажем, сто, и каждый может встречаться в файле разное количество раз. Надо было бы объявить сто счетчиков? Это очень неудобно. Для нашей цели удобнее пользоваться конструкцией языка С, которая называется "массив".

Массив — это множество однотипных данных, объединенных под одним именем. Объявляется массив данных так:

```
<тип данных в массиве> <имя массива>
[количество элементов массива];
```

Последнее значение должно быть целым числом без знака. Например, массив 100 целых чисел можно объявить как `int m[100]`; массив символов как `char s[20]`. Так как элементы массива располагаются в памяти последовательно друг за другом, то массив символов — это не что иное, как строка символов. Причем в языке С принято, что строка символов обязательно оканчивается признаком конца строки, которым служит символ `\0`. Если мы сами формируем символьный массив, то сами же должны позаботиться, чтобы его последним элементом

был символ \0, иначе такую строку, полученную с помощью массива, никогда не распознает ни одна стандартная программа.

Как "доставать" элементы из описанной выше конструкции? Если имеем, например, массив чисел `int m[100]`, то любой его элемент — это `m[i] (i=0,1,…,99)`, где `i` — номер элемента. Видим, что нумерация элементов начинается с нуля: порядковый номер первого элемента массива — 0, второго — 1 и т. д. Порядковый номер элемента массива называют *индексом*.

Примечание

Элемент массива называют переменной с индексами. Если у такой переменной один индекс (в нашем случае именно так и есть), то массив таких переменных называют одномерным, если более одного индекса, то — многомерным (двумерным, трехмерным и т. д.). Часто количество индексов в массиве называют *длиной массива* или *размерностью*.

Чтобы массив инициализировать, т. е. присвоить его элементам какие-то значения, надо присвоить соответствующие значения каждому элементу массива. Если имеем массив `int m[2]`, то следует написать: `m[0]=1; m[1]=8`. Этот же эффект получим, если напишем: `int m[2]={1,8}`. Для символьного массива `char s[3]` можно писать либо `s[0]='a'; s[1]='b'; s[2]='c'`, либо `char s[3]={'a','b','c'}`, либо даже `s[3]="abc"`.

Примечание

В одинарных кавычках `'b'` записывают символы, в двойных кавычках — строки символов, потому что последний символ строки символов — признак конца строки `'\0'`. Поэтому `'b'` — это один символ, а `"b"` — два.

Коль скоро мы заговорили о символах, сделаем еще одно замечание. Если в переменной `c` находится символ цифры, точнее — код цифры, то выражение `c-'0'` дает значение самого числа, код которого находится в `c`. Действительно: таблица кодов ASCII построена так, что коды всех символов английского алфавита расположены в ней по возрастанию. Но цифры, как говорят, и в Африке — цифры. Мы ими пользуемся, несмотря на то, что они "английские". Отсюда следует вывод: за кодом нуля идет

код единицы. То есть разность между кодом нуля и кодом единицы равна единице. За кодом единицы следует код двойки. Разность между кодом двойки и кодом единицы тоже равна единице. Но разность между кодом двойки и кодом нуля равна двум. И так далее. Следовательно разность между кодом числа `i` и кодом нуля равна числу `i`.

Наша программа будет выглядеть так, как показано в листинге 3.1.

Листинг 3.1

```
#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#define eof  27       //признак конца файла
#define maxind 10    //количество элементов массива
//---------------------------------------------------------


int main()
{
  int c;            //для ввода символа
  int nd[maxind]; /*для подсчета количества обнаруженных
    в файле цифр: в nd[0] будет накапливаться количество
    встреченных нулей, в nd[1] - единиц, в nd[2] - двоек
    и т. д.*/
  int i;
  for(i=0; i<maxind; i++)
   nd[i]=0; /*обнуление элементов массива - заготовка их под
          счетчики*/
  printf("Enter your string and then press the key <Enter>
>\n");
  while((c=getchar()) !=eof)
   if(c >= '0' && c <= '9')
       ++nd[c-'0'];       //накопление в счетчике
    printf("Number of digits are:\n");
    for(i=0; i<maxind; i++)
```

```
    printf("for i=%d number of digits=%d\n",i,nd[i]);
   getch();  //задержка изображения на экране
} // от main()
```

Оператором `#define` мы определили символическую константу `maxind`, с помощью которой задаем (и легко можем изменять) размерность массива `nd[]`, элементами которого мы воспользовались в качестве счетчиков. Признаком конца файла служит нажатие <Esc>. В начале все счетчики, т. е. элементы массива, обнуляются, чтобы в них накапливать по единице, если встретится соответствующая цифра. Обнуление происходит в цикле с помощью оператора `for`. Цикл завершится, если нарушится условие продолжения цикла: номер элемента массива (им является значение переменной `i`) станет равным количеству элементов.

Примечание

Максимальный индекс всегда на единицу меньше количества элементов массива, указанного при объявлении массива, т. к. нумерация элементов начинается с нуля.

Далее следует уже знакомый нам цикл ввода символов, в котором проверяется, входит ли код каждого введенного символа в диапазон кодов от ноля до девятки. Если код введенного символа входит в диапазон кодов цифр, т. е. соответствует искомым цифрам, то в соответствующий элемент массива `nd[]` добавляется единица.

Примечание

Вспомним, что коды цифр идут по возрастанию и что, поскольку коды — это числа, мы можем выполнять над ними операцию "минус" или операции отношения (>, >=, <, <=, !=,==).

Заметим, что тело оператора `while` содержит всего один оператор (`if`), и потому не ограничено фигурными скобками. После обнаружения символа конца файла (<Esc>) происходит вывод поэлементно содержимого массива на устройство вывода. Здесь так же, как и при инициализации, организован цикл с помощью оператора `for`, телом которого является функция `printf()`.

Результаты работы программы приведены на рис. 3.1.

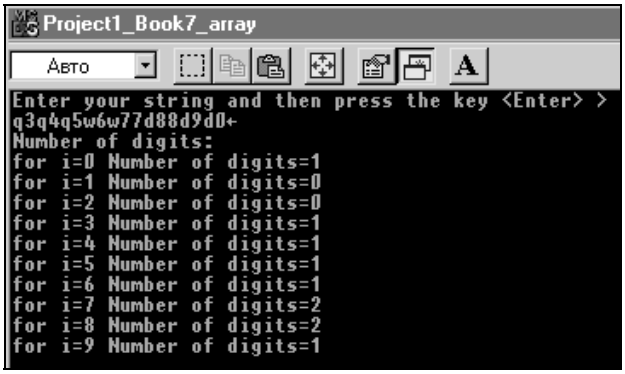


Рис. 3.1. Результат работы программы листинга 3.1

Многомерные массивы

Массивы размерности, большей, чем один, объявляются, как и массивы одинарной размерности, но справа от объявления количества элементов одинарной размерности в квадратных скобках указывают количество элементов для следующей размерности. И так далее. Например, двумерный массив целых чисел объявляется как `int m[10][20]`. В виде такого массива можно объявить прямоугольную матрицу чисел. Говорят, что это "массив десяти строк чисел по 20 чисел в каждой строке" или "массив из десяти строк и двадцати столбцов". Обращаться к элементам такого массива следует, указывая номера строк и столбцов. Например, `m[3][8]`. Или: `int i=3,j=8,k; k=m[i][j]`.

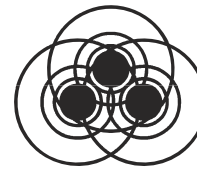
На примере двумерного массива покажем, как надо инициализировать такой массив (т. е. придавать его элементам начальные значения). Допустим, мы хотим составить программу расчета зарплаты работника. Для этой задачи нам понадобится справочник "Количество дней в каждом месяце високосного и невисокосного года". Такой справочник можно представить в виде двумерного массива `int m[2][13]`, в котором элементами первой

размерности будут две строки: одна будет содержать данные по месяцам невисокосного года, а вторая — по месяцам високосного. Элементами второй размерности — собственно количество дней в каждом из двенадцати месяцев. К тому же, т. к. элементы в массивах нумеруются с нуля, то для удобства пользования нашим массивом введем еще один искусственный элемент, равный нолю, и поместим его на нулевое место во второй размерности. Это обеспечит более приемлемое обращение к массиву: например, величина `m[2][2]` будет тогда означать "количество дней високосного года в феврале". Массив можно записать:

```
int m[2][13]={0,31,28,31,30,31,30,31,31,30,31,30,31,
 0,31,29,31,30,31,30,31,31,30,31,30,31};
```

Можно объявлять не только числовые, но и символьные массивы (и не только такие — об этом мы узнаем в следующих главах). Например, символьный массив `char s[20][50]` задает не что иное, как массив из десяти символьных строк, в каждой из которых по 50 символов. Переменное число символов в такой конструкции задать нельзя, т. к. нельзя будет определить положение элемента массива, которое вычисляется исходя из постоянного количества элементов в строках и столбцах.

Глава 4



Создание и использование функций

В процессе программной реализации алгоритмов часто возникает необходимость выполнения повторяющихся действий с разными группами данных. Например, требуется вычислять синусы заданных величин или начислять заработную плату работникам. Ясно, что глупо всякий раз, когда надо вычислить синус какого-то аргумента или начислить зарплату какому-то работнику, создавать заново соответствующую программу именно под конкретные данные.

Напрашивается вывод, что этот процесс надо как-то параметризовать, т. е. создать параметрическую программу, которая могла бы, например, вычислять синус от любого аргумента, получая извне его конкретное значение, или начислять зарплату любому работнику, исходя из его данных. Такие программы, созданные с использованием формальных значений своих параметров, при передаче им конкретных значений параметров возвращают пользователю результаты расчетов. Их называют *функциями* по аналогии с математическими функциями.

Если в математике определена некая функция $y = f(x1, x2, ..., xN)$, то на конкретном наборе данных $\{x11, x21, ..., xN1\}$ эта функция возвратит вычисленное ею значение $y1 = f(x11, x21, ..., xN1)$. В данном случае можем сказать, что аргументы $x1, x2, ..., xN$ — это формальные параметры функции $f()$, а $x11, x21, ..., xN1$ — их конкретные значения.

Функция в С объявляется почти аналогичным образом: задается тип возвращаемого ею значения (из рассмотренного материала

мы знаем, что переменные могут иметь типы `int`, `float`, `long` и т. д. Тип значения, возвращаемого функцией, может быть таким же, как и тип переменных); после задания типа возвращаемого значения задается имя функции (как и для математической функции). Затем в круглых скобках указываются ее аргументы — формальные параметры, каждый из которых должен быть описан так, как если бы он описывался в программе самостоятельно вне функции. Это только первая часть объявления функции в C.

Далее формируется тело функции: программный код, реализующий тот алгоритм, который и положено выполнять определяемой функции. Например, объявим условную функцию расчета зарплаты одного работника:

```
float salary(int TabNom, int Mes)
 {
   /*здесь должен быть программный код расчета зарплаты*/
  return(значение вычисленной зарплаты);
 }
```

Тип возвращаемого значения — `float`, т. к. сумма зарплаты в общем случае число не целое. Имя функции — `salary`. У функции два формальных параметра и оба целого типа: табельный номер работника и номер месяца, за который должен производиться расчет. Особым признаком функции является наличие оператора `return()`, который возвращает результат расчетов. После того как такая функция разработана, пользоваться ею можно так же, как мы пользуемся математической функцией. Для данного примера мы смогли бы записать:

```
float y; y=salary(1001,12);
```

или:

```
float y=salary(1001,12);
```

или:

```
int tn=1001; int ms=12; float y=salary(tn,ms);
```

Во всех случаях при обращении к функции в ее заголовочную часть мы подставляли вместо ее формальных параметров их конкретные значения. Каков внутренний механизм параметризации программы и превращения ее в функцию? Мы описываем в заголовке формальные параметры и затем в теле функции ис-

пользуем их при создании программного кода так, как будто известны их значения. Это возможно благодаря тому, что компилятор, когда начнет компилировать функцию, соотносит с каждым ее параметром определенный адрес некоторого места в так называемой стековой памяти, созданной специально для обеспечения вызова подпрограмм из других программ (а функция — это ведь тоже подпрограмма, которая еще и возвращает некоторое значение, а не только получает какой-то результат расчетов). Размер такой адресованной области для каждого параметра определяется типом описанного формального параметра. Выделяется место и для будущего возвращаемого результата. В теле функции будет построен программный код, работающий, когда речь идет о формальных параметрах, с адресами не обычной, а стековой памяти. Когда мы, обращаясь к функции, передаем ей фактические значения параметров, то эти значения пересылаются по тем адресам стека, которые были определены для формальных параметров (т. е. кладутся на "полочки" в стеке, которые отведены для формальных параметров). Но программный код тела функции как раз и работает с этими "полочками"! "Полочки" и содержат параметры: т. к. тело строится так, что оно работает с "полочками", то остается только класть на них разные данные и получать соответствующие результаты. Это и делается, когда мы передаем каждый раз функции конкретные значения ее параметров.

Отсюда можно сделать выводы: поскольку передаваемые функции значения пересылаются в стековую память (т. е. там формируется их копия), то сама функция, работая со стеком, не может изменить значения переменных, которые подставляются в ее заголовочную часть вместо формальных параметров (в примере мы писали `float y=salary(tn,ms)`, подставляя вместо формальных параметров `TabNom` и `Mes` значения переменных `tn` и `ms`. И мы утверждаем, что значения `tn` и `ms` не изменятся).

Если же передавать функции не значения переменных, а их адреса (об этом речь пойдет в следующих главах), то по адресу можно записать все, что угодно, где бы он ни находился (в стеке или в обычной памяти). В этом случае переменные, адреса которых переданы в качестве фактических параметров, могут изменяться в теле функции.

Функции должны описываться до основной программы: либо их текст располагается до `main()`, либо он подключается к `main()` оператором `#include` (который тоже стоит раньше `main()` в тексте), если функция расположена где-то в другом файле.

Создание некоторых функций

Перейдем теперь от столь пространного введения к созданию некоторых функций и проверке их работы в основной программе.

Ввод строки с клавиатуры

Создадим функцию, вводящую строку символов с клавиатуры и возвращающую длину введенной строки. Такая функция представлена в листинге 4.1.

Листинг 4.1

```
/*Возвращает длину введенной строки с учетом символа '\0';
lim-максимальное количество символов, которое можно ввести
в строку s[]  */


 getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества символов
    return(i);
  }
```

Тип выводимого функцией `getline()` результата не определен, поэтому по умолчанию он будет `int`. Входным параметром функции является `lim` — ограничитель на количество вводимых в строку символов. Дело в том, что в языке C строка символов

представляется в виде массива символов (об этом мы говорили в предыдущей главе), а любой массив имеет свою конкретную размерность (количество элементов). Поэтому, если символы будут вводится в массив `s[]`, размерность которого указана в вызывающей программе, то мы должны задавать параметр `lim`, причем значение `lim` не должно превосходить размерность массива (т. е. определенную длину строки). При определении функции можно писать `s[]`, не указывая конкретной размерности, что удобно, т. к. такую функцию можно использовать в различных случаях, задавая разные размерности.

Далее идет знакомый нам обычный цикл ввода по символу, организованный с помощью оператора `for`. Введенный функцией `getchar()` символ присваивается очередному элементу массива `s[]`, чем и формируется строка. Ввод обеспечивается необходимостью вычисления условия продолжения-завершения цикла (цикл идет по переменной `i`):

```
i<lim-1 && (c=getchar())  != eof && c != '\n';
```

В момент вычисления этого выражения требуется ввести один символ с клавиатуры, иначе выражение не может быть вычислено. Цикл ввода может завершиться при нарушении хотя бы одного из выражений, связанных операцией "И":

- или номер введенного символа (`i`) превзойдет ограничитель `lim` (в условии стоит `i<lim-1`, потому что номер последнего элемента массива, размерность которого `lim`, будет `lim-1`, а надо еще оставить место и для признака конца строки `\0`);
- или будет введен признак конца ввода (он должен быть задан в вызывающей программе так, чтобы был известен в этой функции);
- или ввод строки завершится, когда будет введен признак конца строки символ `\n`.

Первым в сложном условии продолжения-окончания цикла стоит выражение `i<lim-1`. В языке С при вычислении такого рода выражений действует правило: если при вычислении части выражения становится ясным его значение, то вычисления дальнейших частей не происходит и вычисление всего выражения завершается. Этот принцип обеспечивает повышение скорости

обработки. Поэтому если обнаружится, что выражение `i<lim-1` нарушено, т. е. количество вводимых символов превзойдет размерность массива, то не потребуется вводить очередной символ и проверять его на признак конца строки.

Когда цикл ввода закончится, к введенным символам надо добавить признак конца строки, иначе строка в дальнейшем не сможет обрабатываться как строка. Этот символ добавляется, и с его учетом корректируется количество введенных символов. К `i` добавляется единица, т. к. значение этой переменной фактически равно количеству введенных символов, хотя прямое назначение переменной `i` формировать порядковый номер элемента массива, в который будет записываться очередной введенный символ.

Оператор `return(i)` возвращает количество введенных символов. Здесь следует различать возвращаемые функцией значения. Результат ввода возвращается через массив `s[]` через свой выходной параметр.

Примечание

Это возможно, потому что `s`, точнее `s[0]`, — это адрес первого элемента массива, как определено в языке C. Мы выше говорили, что из тела функции можно изменять значения тех переменных, которые передают в функцию не свои значения, а свои адреса. Вот мы и изменили значение переменной `s[]`.

Но когда определяют функцию как подпрограмму, возвращающую обязательно какое-то значение, имеют в виду другое. В этом случае речь идет о значениях, которые возвращает оператор `return()`, а не о выходных параметрах. Если бы наша функция не возвращала количество введенных символов, то она "ничего бы не возвращала", и тогда бы мы определили тип возвращаемого значения как `void`. Функция, которая имеет тип `void`, не возвращает ничего. Функция может что-то возвращать, но не иметь совсем параметров. Тогда при ее создании пишут, например, `float aaa(void)`, а обращение к ней пойдет как `float y=aaa()`.

Итак, параметры функции могут быть входными и выходными, не следует путать с ними возвращаемые значения.

Детально посмотреть, как работает функция или любая другая программа, можно, воспользовавшись программой-отладчиком (Debugger). Чтобы включить отладчик, следует в любой точке программы щелкнуть мышью в поле подшивки редактора текстов (рис. 4.1).

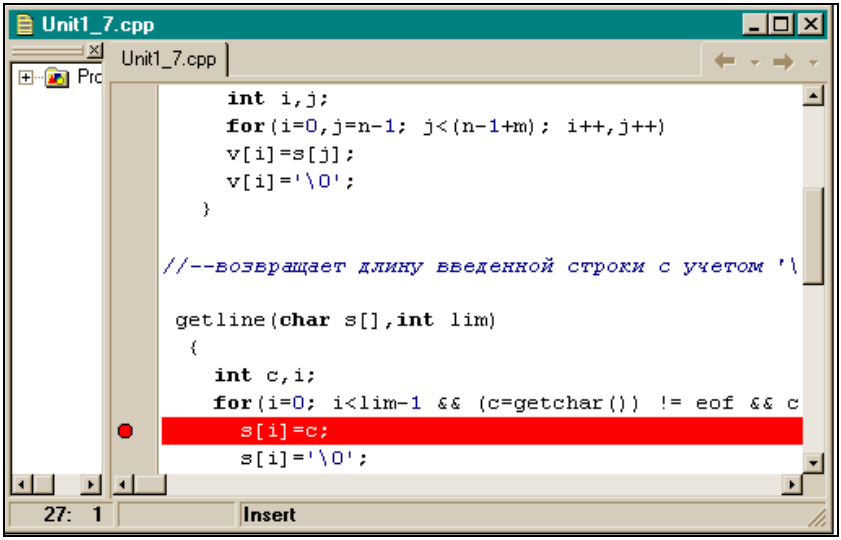


Рис. 4.1. Подключение отладчика к программе

При этом в поле подшивки редактора появится красный кружок, а соответствующая строка подсветится красным. Красный кружок показывает, какой участок кода выполнялся в момент останова программы, и называется *точкой останова (breakpoint)*. Убрать точку останова можно, повторно щелкнув на красном кружке.

От образовавшейся точки останова можно дальше двигаться по тексту программы шаг за шагом, нажимая клавишу <F8>. При этом будут одновременно выполняться все операторы, находящиеся в строке, на которую указывает зеленая стрелка. Эта зеленая стрелка образуется в поле подшивки, когда после пуска программы происходит останов на точке останова и когда мы начинаем двигаться дальше с помощью <F8>, выполняя одну

строку за другой. В этом случае мы имеем возможность просматривать значения переменных, что очень важно для процесса отладки программы: достаточно навести на имя переменной указатель мыши и немного подождать, как рядом с указателем мыши будет выведено содержимое переменной. Если в строке программы стоит обращение к некоторой функции, то, нажав <F8>, мы тут же получим ее выполнение. Но если мы хотим, чтобы произошла передача управления в тело функции, то вместо <F8> следует нажать <F7>.

Сказанное об отладчике — только часть его возможностей. Остальные его функции можно увидеть, воспользовавшись опциями **Run** и **Help** главного меню среды Borland C++Builder. Кстати, любую справку об элементе языка, как и о любом элементе Builder, можно быстро получить, воспользовавшись клавишей <F1>, предварительно либо наведя курсор мыши на интересующий элемент, либо выделив необходимое слово.

Функция выделения подстроки из строки

Пример программы с этой функцией представлен в листинге 4.2.

Листинг 4.2

```
void substr(char v[],char s[],int n,int m)
  {
    //n-й элемент находится в массиве на (n-1)-м месте
     int i,j;
     for(i=0,j=n-1; j<(n-1+m); i++,j++)
     v[i]=s[j];
     v[i]='\0';
   }
```

Эта функция с именем `substr()` ничего не возвращает (тип возвращаемого значения — `void`), а на свой вход получает:

- строку символов `char s[]` (не забудем, что строка символов в языке C задается массивом символов);

- порядковый номер символа, начиная с которого требуется выделить подстроку (`int n`);
- количество символов, которое требуется выделить (`int m`).

А выходным параметром будет выделенная подстрока (`char v[]`).

Алгоритм очень прост: в цикле участвуют две переменные: `i` и `j`. Цикл организован с помощью оператора `for`. Здесь мы видим пример того, что в первом (инициализирующем) выражении оператора `for` может быть более одной переменной (или одного выражения). Такие переменные должны отделяться друг от друга запятой, которую в этом случае называют *операция "запятая"*. Соответственно и в третьем выражении оператора `for`, в котором наращиваются переменные цикла, имеется более одной переменной, которые также разделены запятой.

Цикл начинается заданием значений переменных цикла: первое значение индекса `i` — 0, т. к. массив формируется с нулевого элемента. В нашем случае индекс `i` — порядковый номер символов, заносимых в массив `v[]`. Переменная `j` начинается с `n-1`, т. к. с ее помощью станут извлекаться элементы из массива `s[]` (т. е. из выделяемой строки). Первый элемент (в данном случае — символ) надо извлечь с места `n-1`, поскольку пользователь этой функции задает нумерацию в естественном порядке, т. е. считает, что строка начинается с первого, а не с нулевого символа.

Тело `for` состоит всего из одного оператора `v[i]=s[j]`. В нем происходит пересылка символа из входной строки с места `j` в выходную строку на место `i`. И так будет продолжаться, пока условие продолжения-окончания цикла не нарушится, пока значение переменной `j`, меняясь в третьем выражении от цикла к циклу, не станет равным `(n-1+m)`, т. е. не изменится `m` раз. Это будет означать, что все необходимые символы из `s[]` пересланы в `v[]`. Осталось только выполнить требование C в отношении признака конца строки символов: добавить символ `\0`.

Функция копирования строки в строку

Эта функция показана в листинге 4.3.

Листинг 4.3

```
void copy(char save[],char line[])
  {
   int i=0;
   while((save[i]=line[i]) != '\0')
     i++;
  }
```

Эта функция похожа на предыдущую (`substr()`), но пересылка символов начинается с нулевого элемента входного массива `line[]` в нулевой элемент выходного массива `save[]`. Цикл организован с помощью оператора `while`. Так как на входе имеется строка символов, то она обязательно заканчивается символом `\0`. В условие окончания цикла поставлено выражение `save[i]=line[i]) != '\0'`. Чтобы вычислить это выражение, потребуется, во-первых, перегнать сначала `i`-й символ из входного массива `line[]` в `i`-й элемент выходного массива `save[]` и после этого его значение проверить на совпадение с `\0`. Если совпадения не будет, выполнится тело `while`: индекс элементов массива возрастет на единицу, после чего станет готовым к тому, чтобы по нему перегнать следующий символ из `line[]` в `save[]`. Поскольку эта функция ничего не возвращает, то отсутствует оператор `return()`. Как только будет передан символ `\0`, цикл прекратится и программа "провалится" на закрывающую тело `while` фигурную скобку. Это означает, что функция завершилась.

Головная программа для проверки функций *getline(), substr(), copy()*

Составим теперь головную программу для проверки функций `getline()`, `substr()`, `copy()`. Эта программа приведена в листинге 4.4.

Листинг 4.4

```
#pragma hdrstop
#include <stdio.h>       //for getchar(),putchar()
#include <conio.h>       //for getch()
#include <stdlib.h>      //for exit()


#define eof  27          //признак конца ввода (<Esc>)
#define maxline 1000     //размерность массивов (максимальная
                         //длина строк)
#define from 4        /*константа для выделения подстроки (с
этого символа будет начинаться выделение) */
#define howmany 5      /*константа для выделения подстроки
(столько символов будет выделено) */


//-----substr(s,n,m)-----------------------------------------
void substr(char v[],char s[],int n,int m)
  {
    //n-й элемент находится в массиве на (n-1)-м месте
     int i,j;
     for(i=0,j=n-1; j<(n-1+m); i++,j++)
     v[i]=s[j];
     v[i]='\0';
   }
//---------------------------------------------------------------
 getline(char s[],int lim)
  {
    int c,i;
```

```
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n'; i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }

//------------Копирование строки в строку---------------
void copy(char save[],char line[])
  {
   int i=0;
    while((save[i]=line[i]) != '\0')
     i++;
  }
//-----------------------------------------------------------


main()
{
     char s[maxline],v[maxline],w[maxline];
     int i=getline(s,maxline);
     copy(v,s);
     substr(w,v,from,howmany);
     if((i-1) < from)
      {
       printf("Length of the entered string is not enough for
extraction from it");
       getch();
       exit(0);
      }
     printf("Entered string = %s\n",s);
     printf("Copied string .= %s\n",v);
     printf("substring..... = %s\n",w);
     getch();
}
//-----------------------------------------------------------
```

Смысл приведенной основной программы ясен из комментария к ней. Заметим только, что здесь встретилась новая библиотечная функция `exit()`, которая прерывает выполнение программы. Чтобы ее использовать, надо подключить файл stdlib.h. Так как строка символов может быть произвольной длины, то приходится проверять, достаточно ли в ней символов, чтобы выделить подстроку с указанием количества выделяемых символов и номера символа, с которого начнется выделение (`if((i-1) < from`).

Напомним, что длину строки возвращает функция `getline()`, а количество выделяемых символов мы задали с помощью оператора `#define`. В выражении `if((i-1) < from` мы записали `i-1`, чтобы длина проверялась без учета символа `\0`. Если длина введенной строки меньше номера символа, с которого надо выделять подстроку, то, естественно, надо об этом сообщить пользователю (что мы и делаем) и завершить программу. Это делает функция `exit()` (можно было бы добавить к программе блок возврата на повторный ввод новой строки, но у нас была задача другая: познакомиться с работой функций и новой функцией `exit()`). Результат работы основной программы приводится на рис. 4.2.

Рис. 4.2. Результат работы программы листинга 4.4

Внешние и внутренние переменные

Функции можно создавать и без параметров, если воспользоваться внешними переменными. *Внешняя переменная* — это переменная, значение которой известно во всех функциях, объявленных после нее, в том числе и в самой функции `main()`.

Этот прием "беспараметризации" удобно применять тогда, когда в действительности у функции надо вводить столько параметров, что это затруднит ее понимание, или когда две функции обмениваются общими данными, не вызывая друг друга. Удобно это применять и тогда, когда в теле функции существуют массивы, требующие инициализации. Если массив, объявленный в теле функции, требует инициализации, то каждый раз при входе в функцию эта инициализация будет происходить, что продлит время выполнения программы. Лучше объявить такой массив вне функции, но так, чтобы он был известен в этой функции, один раз проинициализирован. То есть массив надо объявить как внешнюю переменную по отношению к данной функции.

Следует сказать, что существуют и *внутренние* или *локальные* переменные (их еще называют "автоматическими"). Это такие переменные, которые объявлены в теле какого-либо оператора (`if`, `while`, `for`, `do-while`) или в теле функции. Такие переменные, как говорят, локализуются в блоке объявления, т. е. известны только в таком блоке и не известны за его пределами. Например, можно писать:

```
for(int i=0; i<10; i++)
{какие-то операторы}
i=0;
```

В этом случае первая переменная `i` известна только в цикле `for`, а в выражении `i=0`; это будет уже другая переменная, т. е. ей компилятор присвоит совсем другой адрес. Если внешняя переменная объявлена в некотором другом файле, подключаемом с помощью оператора `#include`, то она должна быть объявлена и той программе, которую мы составляем, но с атрибутом `extern`. Такая переменная может инициализироваться при объявлении только один раз и в месте ее основного объявления: например,

имеем `int a=5;` (объявление с инициализацией) в файле F1.h В нашей программе мы выполнили `#include "F1.h"`.

Примечание

Чтобы подключить библиотечный файл, его имя нужно указать в угловых скобках, а имя файла, созданного пользователем, заключают в двойные кавычки.

В программе мы должны написать: `extern int a;` (но не `extern int a=0;` никакое другое значение присвоить при этом "дополнительном" объявлении уже нельзя). Приведем виды программ `getline()`, `copy()` и `main()`, которые вместо параметров используют значения внешних переменных (листинг 4.5).

Листинг 4.5

```
#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>     //for getch()
#include <string.h>   //for strcpy()
#define eof  27       //признак конца ввода (<Esc>)
#define maxline 1000  //длина максимально возможной строки

//внешние переменные, но объявлены в этом же программном
//файле
char line[maxline];
char save[maxline];
int max;

//Объявление функций
/*Формирование строки ввода с клавиатуры в line[]:возвращает
длину введенной строки; lim-максимальное количество символов,
которое можно ввести в строку line[] */
 getline()
  {
    int c,i;
    extern char line[];
```

```
/*использование глобальной переменной в функции: т. к.
описание находится в этом же файле, то в функции такую
переменную можно было бы не описывать. Но мы это сделали
для общего случая. */


    for(i=0; i<maxline-1 && (c=getchar()) != eof && c !=
'\n'; i++)
      line[i]=c;
    i++;
    line[i]='\0';
    return(i);
  }
//------------Копирование строки в строку---------------
void copy()
  {
extern char line[];          /* писать общий атрибут extern
                                для нескольких объявляемых
                                переменных нельзя*/
extern char save[];
   int i=0;
    while((save[i]=line[i]) != '\0')
     i++;
  }
/*Основная программа: выбирает строку наибольшей длины из
всех, вводимых с клавиатуры*/
int main()
{


int len;          //длина текущей строки
extern int max; /*здесь будет храниться длина наибольшей из
2-х сравниваемых по длине строк*/
extern char save[];
max=0;
while((len=getline()) >1)
 {
     /*когда нажмем <Esc> или <Enter>, то длина строки станет
равной 1 за счет учета признака конца '\0' */
```

```
 if(len > max)
   {
    max=len;
    copy();
   }
 }
 /*когда мы нажмем <Esc> или <Enter> (конец ввода), то get-
line() выдаст единичную длину (с учетом символа '\0') и мы
попадем сюда*/
if(max > 0)     //была введена хоть одна строка
  printf("Max's string = %s\n",save);
 getch();
}
```

Результат работы программы показан на рис. 4.3.

Рис. 4.3. Результат работы программы листинга 4.5

Область действия переменных

Выше мы говорили, что существуют внешние переменные, которые известны во всех объявленных ниже функциях и блоках. Эти переменные могут быть объявлены как в тексте разрабатываемой вами программы, так и в текстах внешних файлов, которые следует к вашей программе подключать с помощью оператора `#include`. В таких случаях подобные переменные объявляются в

вашей программе, но с атрибутом `extern` (например, `extern int a;`) и не могут быть инициализированы при объявлении в вашей программе. Как создать внешний файл? И вообще, для чего он создается?

Предположим, что вы — участник разработки крупного проекта, выполняемого многими группами специалистов. При постановке задач такого проекта оказалось, что имеется довольно много общих для всех групп разработчиков данных, которые надо использовать в программах, создаваемых каждой группой. Было бы неверным, если бы программисты каждой такой группы описывали в своих программах такие общие данные. Легче все эти данные описать (объявить) в одном файле с расширением h, чтобы каждый программист мог им пользоваться. К тому же при таком подходе получится, что у всех программ, использующих общий файл, переменные имеют одинаковые наименования и смысл, что значительно облегчает дальнейшее сопровождение программ.

Как создать свой внешний файл

Приведем текст программы проверки на принадлежность к внешнему файлу некоторой переменной `a`, объявленной и инициализированной числом 20 (листинг 4.6).

Листинг 4.6

```
//---------------------------------------------------------


#pragma hdrstop
#include <stdio.h>
#include <conio.h>  //getch()

//---------------------------------------------------------
#include "C:\Мои документы\Unit1.h"
#pragma argsused
```

```
int main(int argc, char* argv[])
{
extern int a;    // в h-file'е a=20
      printf("a=%d\n",a);
      getch();
}
//---------------------------------------------------------
```

h-file формируется редактором кода: надо создать заготовку `main()` обычным способом, выполнив команду **File/New/Console Wizard**, очистить все поле редактора и записать в это поле необходимые данные. Затем выполнить опцию главного меню: **File/Save As** и выбрать для сохраняемого файла расширение h в раскрывающемся окне типов файла **C files**: c, cpp, h. Затем с помощью оператора `#include` файл следует включить в созданную заготовку `main()` для нового консольного приложения.

Атрибут *static*

Наряду с внешними переменными существуют, как мы видели, локальные переменные. Локальные переменные объявлены в какой-либо функции или блоке и известны только там. При этом каждый раз при вхождении в функцию (блок) такие переменные получают значения, а при выходе эти значения теряются. Как же заставить переменные сохранять свои значения после выхода из функции (блока)? Такая проблема существует во множестве алгоритмов. Решить ее можно, объявив переменную с атрибутом `static`.

Ниже приведен текст проверочной программы (листинг 4.7). Для простоты взята уже известная нам функция ввода строки с клавиатуры `getline()`, и в ее текст вставлено объявление статической переменной `j`, которой там же, в тексте, присваивается значение количества введенных символов `i`. Можно проследить в режиме отладчика, что локальная переменная `j` сохраняет свое значение и после выхода из функции, и после нового входа в функцию.

Листинг 4.7

```
//-------------------------------------------------------------
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>
#define eof 27          //Esc
#define maxline 100



//-------Ввод строки с клавиатуры---------------
/*в следующую функцию вставлены "лишние" операторы, чтобы
продемонстрировать действие атрибута static*/


getline(char s[],int lim)
  {
    int c,i;
  static  int j;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
     j=i;
    return(i);
  }
//----------------------------------------
main()
{
 /*для ввода используем объявление char *s;*/

 char s[maxline];
for(int i=0; i <3; i++)
   {
    getline(s,maxline);
    static int b;
```

```
    b=i;
   }
  getch();
}
```

В функцию `getline()` вставлены "лишние" операторы (`static int j` и `j=i`), чтобы продемонстрировать действие атрибута `static`. Они не меняют функциональности `getline()`. Поставьте точку останова отладчика на оператор `j=i` и в цикле работы `getline()` убедитесь, что переменная `j` сохраняет свое значение в других циклах запуска `getline()`.

Для ввода можно использовать объявление `char *s` (об этой записи мы поговорим в следующих главах). В этом случае перед обращением к функции ввода надо динамически выделить память функцией `s=(char*)malloc(maxline)`, а в конце программы выполнить оператор `free(s)`.

Рекурсивные функции

Рекурсивные функции (от лат. *recursio* — движение назад, возвращение) — такие функции, которые могут вызывать сами себя. При этом каждый раз под каждый вызов создается совершенно новый набор локальных переменных, отличный от набора вызывающей (т. е. этой же) функции. Рекурсия применяется при обработке так называемых "рекуррентных" (основанных на рекурсии) формул. Одной из таких формул является, например формула вычисления факториала числа: $n! = (n - 1)! * n$, где $0! = 1$. Чтобы вычислить факториал на шаге n, надо воспользоваться факториалом, вычисленным на шаге $n - 1$. Рекурсивная функция, реализующая алгоритм для вычисления факториала, показана в листинге 4.8.

Листинг 4.8

```
int fact(int i)
    {
      if(i==0)
```

```
        return(i);
        else
    {
        i=i * fact(i-1);
         return(i);
     }
      }
```

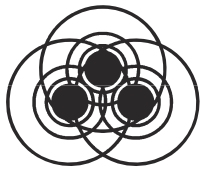
Быстрый вызов функций

В среде C++ Builder применяется так называемый *быстрый вызов функций*. При этом вызове функции приписывают атрибут `__fastcall` в виде:

```
<тип возвращаемого значения> __fastcall <имя функции>
(список аргументов)
```

Компилятор построит код, в котором значения аргументов размещаются не в стеке, а на регистрах процессора, если последние свободны. Это ускорит выполнение программы.

Глава 5



Основные стандартные функции для работы с символьными строками

Для работы с символьными строками в языке С существует ряд функций. Чтобы ими воспользоваться, надо в основную функцию `main()` включить файл string.h. Рассмотрим основные строковые функции.

Функция *sprintf (s, Control,arg1,arg2,...,argN)*

Эта функция родственна функции `printf()`, которую мы уже рассматривали. Она работает точно так же, как и `printf()`, но в отличие от функции `printf()`, которая выводит результат своей работы на стандартное выводное устройство (по умолчанию — экран), функция `sprintf()` результат своей работы выводит в строку `s`. Это очень полезная функция: с ее помощью мы можем собрать в одну строку совершенно разнотипные данные, расположенные в переменных `arg1`, `arg2`, ..., `argN`, да еще и вставлять между ними необходимый текст, который может находиться между форматами расположенных в управляющей строке `Control` данных.

Функция *strcpy(s1,s2)*

Эта функция выполняет то же, что и функция `copy()`, рассмотренная нами в *разд. "Головная программа для проверки функций*

getline(), substr(), copy()" главы 4: она копирует содержимое строки `s2` в строку `s1`. Признак конца строки — символ `\0` — тоже копируется. Напомним, что строка в языке C представляет собой массив символов (описывается как `char s[]`) и что имя массива является адресом его первого элемента `s[0]`. Это нам пригодится в дальнейшем, когда в качестве аргументов `strcpy()` будут выступать не имена массивов, а имена переменных, типы которых мы будем изучать позже.

Функция *strcmp(s1,s2)*

Эта функция сравнивает две строки (т. е. содержимое переменных `s1` и `s2`) и выдает результат сравнения в виде числового значения. Если `s1` = `s2`, функция возвращает ноль; если `s1` < `s2` — отрицательное число; если `s1` > `s2` — положительное число.

Функция возвращает число, поскольку сравнивает не сами символы, а их коды из таблице ASCII (обычные числа). Мы знаем, что коды символов в таблице кодирования символов ASCII, на основе которой кодируются символы в языке C, для английского алфавита расположены по возрастанию. Они занимают первую половину (первые 128 позиций) таблицы. Вторая половина таблицы (остальные 128 позиций) отдана под национальные кодировки, которые, в общем случае, неупорядочены, что касается и кириллицы. Этот момент надо учитывать при сравнении символьных строк с помощью `strcmp()`.

В теле функции коды строки `s1` с помощью вычитания посимвольно сравниваются с кодами строки `s2`. Такая обработка символов происходит до первого неравенства кодов, и результат вычитания выводится в качестве результата работы функции. Повторим, что такой подход возможен, потому что символы английского алфавита в таблице ASCII упорядочены по возрастанию: код символа `a` меньше кода символа `b`, и в этом смысле строка "`a`" меньше строки "`b`". Поэтому если все символы, расположенные в строках на одинаковых местах, равны, то строки считаются равными, в противном случае одна строка либо меньше, либо больше другой. Таким образом, строки, содержащие текст на кириллице, сравнивать с помощью этой функции

нельзя. Следует отметить, что одинаковые строки, отличающиеся только регистром символов, не будут равны. Это и понятно: прописные и строчные буквы в таблице ASCII имеют разные коды.

Функция *strcmpi(s1,s2)*

Эта функция работает так же, как и `strcmp()`, но регистров не различает: для нее, например, символ `a` совпадает с символом `A`.

Функция *strcat(s1,s2)*

Это функция сцепления (как говорят, *конкатенации*) двух строк. Содержимое строки `s2` дописывается в конец строки `s1`, и результат пересылается в `s1`.

Функция *strlen(s)*

Эта функция возвращает ("возвращает", значит можно писать, например, `int y=strlen(s)`) длину строки `s` (т. е. количество символов в строке) без учета символа `\0` — признака конца строки.

Пример программы проверки функций

Напишем программу, на примере которой проследим, как работают рассмотренные функции (листинг 5.1).

Листинг 5.1

```
//---------------------------------------------------------
#pragma hdrstop
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>      //for getch()
```

```
#include <string.h>     //for strcpy(),..
#include <stdlib.h>     //atoi(),atof()

#define eof 27          //<Esc>
#define maxline 1000

/* Функция getline(s,lim) вводит с клавиатуры строку в s и
возвращает длину введенной строки с учетом символа \0;
lim — максимальное количество символов, которое можно ввести
в строку s*/

 getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }
//-----------------------------------------
int main()
{
 //Программы работы со строками в C

  //----использование sprintf()------------
  int x; float y; char s1[maxline];
  char c,c1,ot[5],v1[maxline];
  do
  {
   printf("Enter int n for sprintf()...>");
   getline(ot,5);
   int x=atoi(ot);
```

```
   printf("Enter float m for sprintf() >");
   getline(ot,5);
   float y=atof(ot);

   printf("Enter string for sprintf(). >");
   getline(s1,maxline);

   sprintf(v1,"%d  %f  %s",x,y,s1);
   printf("v=%s\n",v1);
   printf("continue - Enter, exit - Esc >");
  }
  while((c1=getchar()) != eof)
    ;
  //--------использование strcpy()--------------

 char s2[maxline],v2[maxline];
 while((c=getchar()) != eof)
  {
    printf("\n\nEnter string for strcpy() >\n");
    getline(s2,maxline);
    strcpy(v2,s2);
    printf("Copied string=%s\n",v2);
    printf("Continue - Enter, exit - Esc >");
    getch();
  }

//-----------использование strcmp(), strlen()-----------

    char s3[maxline],v3[maxline];
    while((c=getchar()) != eof)
     {
      printf("\n\nEnter string1 for strcmp() >");
      getline(s3,maxline);
      printf("Enter string2 for strcmp() >");
```

```
      getline(v3,maxline);
      int i=strcmp(s3,v3);
      printf("strcmp's value=%d\nstring1's length=%d\n
",i,strlen(s3));
      if(i==0)
        printf("strin1 = string2\n");
      if(i>0)
        printf("strin1 > string2\n");
      if(i<0)
        printf("strin1 < string2\n");
      printf("continue - enter, exit - Esc >");
      getch();
     }

//------------использование strcat()--------------

char s4[maxline],v4[maxline];
    while((c=getchar()) != eof)
     {
      printf("\n\nEnter string1 for strcat() >");
      getline(s4,maxline);
      printf("Enter string2 for strcat() >");
      getline(v4,maxline);
      printf("strcat's value=%s\n",strcat(s4,v4));
      printf("continue - enter, exit - Esc >");
      getch();
     }
}

//-------------------------------------------------------
```

Для ввода данных мы использовали ранее рассмотренную нами функцию `getline()`, которая вводит строку символов. Но для наших целей нам требуется вводить числа. Поэтому мы их вводим с помощью `getline()` как текст, а потом преобразуем в тип

`int` с помощью функции `atoi(s)`, которая переводит строковые данные в целое число и возвращает это число. Если в строке — нечисловые данные, то функция возвратит ноль.

Другая функция, использованная нами при переводе строковых данных в тип `float`, это функция `atof(s)`. Она возвращает число типа `float`, преобразуя свою входную строку. И также возвращает ноль, если в строке — нечисловые данные. Для ввода чисел мы использовали массив `char ot[5]`, поскольку число, вводимое в примере, не превзойдет пяти цифр (ввести больше не позволит функция `getline()`: так мы ее сформировали).

В этой программе мы встретились с новым оператором `do-while`. Он работает, как оператор `while`, но с тем отличием, что оператор `while` проверяет условие продолжения-окончания цикла в своей заголовочной части, а оператор `do-while` — в конце. Если нарушается условие продолжения оператора `while`, то его тело пропускается. Но в жизни бывают случаи, когда требуется, чтобы цикл `while` выполнился хотя бы один раз. Для этого и используют пару `do-while`, в которой условие продолжения проверяется в конце цикла, поэтому тело оператора будет выполнено хотя бы один раз. Формат оператора таков:

```
do
{ тело оператора}
while (условие цикла)
;
```

Точка с запятой обязательна. Для ввода нескольких вариантов данных в этой проверочной программе потребовалось ввести так называемое *зацикливание*: поставить оператор `while`, который обеспечивает зацикливание за счет запроса ввода символа либо для продолжения ввода другого варианта данных, либо для выхода из участка проверки. Но на первом участке удобно проводить проверку на продолжение ввода вариантов данных не в начале участка, а в конце, чтобы первый вариант данных вводился без проверки. Иначе пошел бы запрос на ввод символа для проверки на продолжение ввода: программа ожидала бы ввода, на экране бы мигал один курсор, и пользователю было бы непонятно, что же надо дальше делать.

Поясним работу функции `sprintf()`. Для ее проверки мы ввели два числа: одно имело тип `int`, другое — `float` и строку (тип `s`), чтобы показать, что `sprintf()` обработает данные в соответствии с их типом (в управляющей строке функции мы задали эти типы) и соберет в единую строку, включив в нее и символы, которые находились между полями, задающими тип (т. е. между полями, начинающимися со знака `%`, и оканчивающимися одним из символов форматирования `d`, `f`, `s`).

Для функции `strcmp()` мы вывели значение, которое она возвращает, чтобы читатель мог удостовериться, что это есть разность между первыми неравными кодами символов. Попробуйте определить, коды каких символов первыми оказались неравными. Код символа можно найти, воспользовавшись стандартной функцией `char(имя символа)`, которая возвращает код символа, указанного у нее в аргументе, например так: `int a=char('a')`.

Результат работы проверочной программы приведен на рис. 5.1.

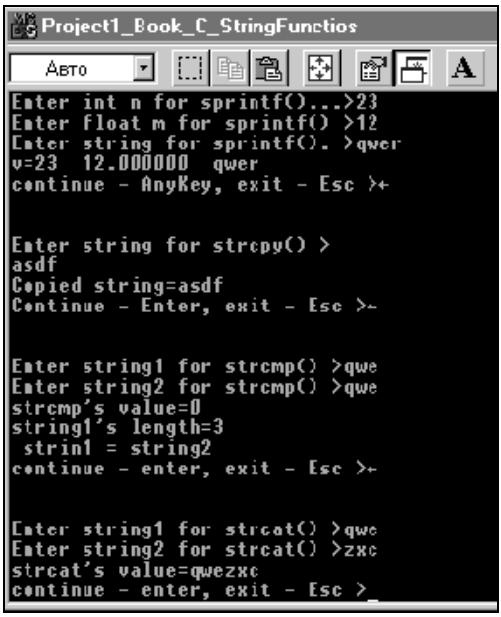
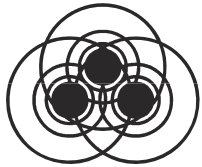


Рис. 5.1. Результат работы программы листинга 5.1

Глава 6



Дополнительные сведения о типах данных, операциях, выражениях и элементах управления

Новые типы переменных

В языке C наряду с рассмотренными ранее типами переменных (`int`, `char`, `float`) существуют и другие типы данных.

- `double` — указывает, что данные имеют тип "с плавающей точкой" двойной точности.
- `long` — указывает, что данные имеют тип "целое со знаком", но по сравнению с данными типа `int` занимают в 2 раза больше памяти.
- `short int` — короткое целое со знаком, занимает в 2 раза меньше памяти, чем `int`.
- `long double` — длинное удвоенное с плавающей точкой.
- `unsigned char` — если в переменной такого типа будет находиться число (которое, естественно, будет записано кодами цифр), то знаковый бит такого числа будет подавлен, т. е. не будет учитываться как знак числа, а только как элемент числа. Это исказит размер отрицательных чисел.

Примечание

Знаковый бит всегда располагается в старшем, находящемся слева разряде. Биты в записи числа нумеруются справа налево: 0, 1, 2, … Это означает, что если под число отведено 4 байта (32 бита), то самый старший бит 31-й.

Попробуйте выполнить следующую программу, используя проверку содержимого переменных с помощью точек останова, образованных отладчиком:

```
main()
{
   int i=-10;
     unsigned int j=i;
}
```

Вы убедитесь, что число `j` огромно. И все из-за того, что знаковый разряд, в котором была единица, т. к. число в переменной `i` отрицательное, стал участвовать в величине числа и превратился в его неотъемлемую часть.

- `unsigned int` аналогичен типу `unsigned char`.
- `unsigned long` — длинное целое без знака. Последствия того, что переменная имеет тип "без знака", для отрицательных чисел рассмотрены выше.
- `enum` — так называемый перечислимый тип данных. Он позволяет задавать мнемонические значения для множеств целых значений (т. е. обозначать значения в соответствии с их смыслом). Допустим, нам надо работать в программе с названиями дней недели, например, проверять, что текущий день — понедельник. Пока не было типа данных `enum`, надо было как-то задавать дни недели числами и работать с этими числами. Для дней недели это и не особенно сложно: каждый помнит, что седьмой день это воскресенье, а первый — понедельник (хотя в одном из компонентов Builder первым днем считается как раз воскресенье). Но бывают множества, как говорят, и "покруче", чем дни недели, элементы которых не упомнишь. С помощью типа `enum` можно добиться большей наглядности и лучшего понимания программы. Приведем пример программы с использованием типа `enum` (листинг 6.1).

Листинг 6.1

```
int main()
{
      enum days {sun, mon, tues, wed, thur, fri, sat}anyday;
      enum sex  {man, wom}pol;
      anyday=sun;
      pol=wom;
      if(anyday==0 && pol == wom)   /*можно писать либо
      anyday==sun, либо anyday==0; */
}
```

Запись `enum days {sun, mon, tues, wed, thur, fri, sat}anyday` — пример объявления переменной `days` перечислимого типа. В фигурных скобках находятся заданные заранее значения переменной `days`. Сама переменная `days` задает как бы шаблон, который самостоятельно применять нельзя. Следует объявить дополнительно другую переменную этого "типа-шаблона": ее имя можно записать сразу при объявлении шаблона между последней фигурной скобкой и двоеточием (у нас это переменная `anyday`). Можно было бы записать объявление так:

```
enum days {sun, mon, tues, wed, thur, fri, sat};
enum days anyday,otherdays;
```

В любом случае после этого переменной `anyday` (или, как во втором случае, и переменной `otherdays`) можно присваивать значения из определенного в переменной `enum` списка:

```
anyday=sun;
```

Если список приведен в виде `enum days {sun, mon, tues, wed, thur, fri, sat}`, то подразумевается, что его элементы имеют последовательные целые числовые значения: 0, 1, 2, То есть вместо `anyday==sun` можно писать `anyday==0`. Список, указанный в `enum`, можно при объявлении инициализировать другими целыми значениями. Например, имеем перечислимый тип "`кувшины(объемы в литрах)`":

```
enum {Vol1=5, Vol2=7, vVol3=9,Vol4,Vol5,Vol6}pitchers;
```

У первого, второго и третьего элементов этого множества числовые значения соответственно равны 5, 7 и 9. Остальным элементам, не определенным нами, компилятор присвоит последовательные (через единицу) значения, начиная со значения последнего определенного элемента, в нашем случае с `Vol3`. `Vol4` получит значение `Vol3+1`, т. е. 10, `Vol5` — 11, `Vol6` — 12. В приведенной выше программе заданы две переменные перечислимого типа: `days` (дни) и `sex` (пол). Это пока шаблоны. А на их основе определены собственно "рабочие" перечислимые переменные, т. е. те, с которыми и надо работать. Затем определенным таким образом переменным можно присваивать значения элементов перечислимых множеств и сравнивать их.

- `bool` — этот тип заимствован из языка C++, расширения C. Переменные этого типа могут принимать только два значения: `false` (ложь) и `true` (истина). Они используются для проверки логических выражений. Числовые значения `false` (ложь) и `true` (истина) заранее предопределены: значение `false` численно равно нулю, а значение `true` — единице. То есть эти так называемые *литералы* (постоянные символьные значения) сами выступают в роли переменных: им можно присваивать значения (соответственно ноль и единица). Вы можете преобразовать некоторую переменную типа `bool` в некоторую переменную типа `int`. Такое числовое преобразование устанавливает значение `false` равным нулю, а значение `true` — единице. Вы можете преобразовать перечислимые и арифметические типы в переменные типа `bool`: нулевое значение преобразовывается в `false`, а ненулевое — в `true`. Существует одноименная функция `bool()`, которая преобразует арифметические типы в булевы. Ниже приведен пример программы с использованием булевых переменных и булевой функции (листинг 6.2). Поставьте с помощью отладчика точку останова на первом операторе и двигайтесь по программе шаг за шагом, каждый раз нажимая клавишу <F8>. Перед каждым следующим шагом проверяйте содержимое переменных, наводя на них указатель мыши (при этом надо немного подождать, пока всплывет значение переменной). Смотрите, как изменяются значения переменных.

Листинг 6.2

```
int main()
{
//проверка преобразований типов int,float в bool
 int i=2;
 bool b=i;
 float j=2.2;
 bool a=bool(j);
 j=0.0;
 a=bool(j);

}
```

Константы

В языке C поддерживаются следующие типы констант (постоянных величин):

- *вещественные константы* — их значения это, в общем случае, нецелые числа с плавающей точкой;
- *перечислимые константы* — значения такого типа постоянны: это значения элементов перечислимого множества;
- *целая константа* записывается как:

  ```
  int i = 12;
  ```

  Плавающая точка может определяться так:

  ```
  float r=123.25e-4;
  ```

  или

  ```
  r=0.15;
  ```

  Обе записи равноценны. Целые числа, кроме десятичных, могут быть также восьмеричными и шестнадцатеричными. Первые пишутся как `int i = 027;` (т. е. с нулем впереди), а вторые как `int i = 0xa;` (т. е. с `0x` впереди);

- *символьные константы* — среди этого типа различают собственно символьные константы и строковые константы. Первые обозначаются символами в одинарных кавычках (апострофах), вторые — в двойных.

  Отличие строковых констант от символьных в том, что у строковых констант в конце всегда стоит признак конца строки — символ `'\0'` (так записывается ноль как символ). Этот признак формирует компилятор, когда встречает выражение вида `char s[]="advbn";` или вида `char *s="asdf";` (здесь для иллюстрации применена конструкция "указатель", о которой речь пойдет в следующих главах).

  Символьные константы имеют вид

  ```
  char a='b';
  ```

  или

  ```
  char c='\n'; char v='\010';
  ```

  В первом случае так задаются константы для символов, которые отображаются на экране (это все символы, закодированные в таблице ASCII кодами от 32 и далее). Во втором случае — для символов, которые не имеют экранного отображения и используются как управляющие (это символы с кодами 0—31 в таблице ASCII).

Новые операции

Поговорим об операциях, которые мы не встречали в рассмотренных ранее примерах.

Операции отношения

Операции *отношения*: `==` (равно), `!=` (не равно), `>=` (больше или равно), `>` (больше), `<=` (меньше или равно), `<` (меньше). Если две переменные сравниваются с помощью операций отношения, то результат сравнения всегда бывает булева типа, т. е. либо ложен, либо истинен. Поэтому мы можем, например, писать:

```
int i=34; int j=i * 25;
bool a=i>j;
```

Операции отношения бывают очень полезны. Допустим, в вашей программе цикл `while(выражение) {…}` работает не так, как вам бы того хотелось. Вы никак не можете разобраться в чем дело из-за того, что выражение в операторе `while` настолько громоздко, что вы не можете его вычислить в момент отладки программы. Тогда это выражение вы можете присвоить некоторой (объявляемой вами) булевой переменной. Теперь вы сможете в режиме отладки увидеть значение этого выражения и принять соответствующие меры.

Надо учитывать, что операции отношения по очередности их выполнения младше арифметических операций. То есть если вы напишете `if(i < j-1)`, то при вычислении выражения в `if()` сначала будет вычислено `j-1`, а затем результат будет сравниваться с содержимым переменной `i`.

Логические операции

Логические операции `&&` (И) и `||` (ИЛИ). Тип выражения, в котором будут участвовать эти операции, будет булевым, как и для операций отношения. Причем выражение `a && b` будет истинным только тогда, когда истинны оба операнда (операнды имеют булевы значения). Выражение `a || b` будет истинным только тогда, когда хоть один из операндов истинен. Следует иметь в виду, что компилятор строит такой алгоритм вычисления выражений, связанных этими операциями, что выражения вычисляются слева направо, и вычисление прекращается сразу, как только становится ясно, будет ли результат истинен или ложен. Поэтому при формировании выражений подобного рода следует их части, которые могут оказать влияние на полный результат первыми, располагать первыми, что экономит время.

Унарная операция отрицания

Унарная (воздействующая только на один операнд) операция `!` (НЕ) является операцией отрицания. Она преобразует операнд, значение которого не равно нулю или истине, в ноль, а нулевой или ложный — в единицу. Результат этой операции представляет собой булево значение. Иногда ее используют в записи вида:

`if(!a)` вместо `if(a==0)`. Действительно, по определению оператора `if()` условие в его скобках всегда должно быть истинным, поэтому получается, что `!a=1` (истина). Тогда `!!a=a=0`, поскольку мы применили к обеим частям равенства операцию "НЕ", а повтор этой операции возвращает число в исходное состояние.

Преобразование типов данных

Современные компиляторы многое берут на себя, неопытный программист этого не замечает и потому должным образом не оценивает происходящее. Но все же надо иметь представление о преобразованиях типов данных, потому что тот же неопытный программист часто приходит в тупик в очевидных ситуациях и недоуменно разводит руками: "Чего это оно не идет? Не понимаю". А не понимает, потому что не знает или не хочет вникать, избалованный возможностями современных компиляторов, при которых он родился и вырос. Но и эти его современники иногда подводят. И очень сильно.

Итак, при составлении программ все-таки надо знать, что в выражениях обычно участвуют данные разных типов, как и при операции присвоения, при которой левая часть имеет один тип, а правая другой. И чтобы как-то свести концы с концами, установлены соответствующие правила преобразований данных разных типов. Рассмотрим два случая преобразований: что делается при вычислении выражений, в которые входят данные разных типов, и что — если правая часть выражения присваивается левой, и обе имеют разные типы.

Примечание

Преобразования осуществляются для тех типов данных, для которых это имеет смысл.

При вычислении выражений, в которые входят данные разных типов, компилятор строит программу так, что все данные разных типов преобразуются к общему типу по следующим правилам:

- типы `int` и `char` могут свободно смешиваться в арифметических выражениях, т. к. перед вычислением переменная типа

`char` автоматически преобразуется в `int` (конечно, когда оба типа относятся к числу). Поэтому когда мы видим, что символ может быть отрицательным числом (например, −1), то его лучше помещать в переменную, объявленную как `int`;

- к каждой арифметической операции применяются следующие правила: низший тип всегда преобразуется в высший*:* `short` в `int`, `float` в `double`, `int` в `long` и т. д.;
- при присвоении тип значения правой части всегда преобразуется в тип левой части. Следовательно надо учитывать, что:
  - если переменная, расположенная справа от знака присвоения, имеет тип `float`, а переменная, расположенная слева — `int`, то произойдет преобразование в тип `int`, и дробная часть значения переменной типа `float` будет отброшена;
  - если справа расположена переменная типа `double`, а слева переменная типа `float`, то произойдет преобразование в тип `float` с округлением;
  - если справа расположена переменная типа `long`, а слева переменная типа `int`, то произойдет преобразование в тип `int`, при этом у значения переменной справа будут отброшены старшие биты (вот это может быть погрешность!);
- любое выражение может быть приведено к желаемому типу не автоматически при преобразованиях, а *принудительно* с помощью конструкции: `<(имя типа) выражение>`. Например, существует стандартная функция `malloc(число)`, которой выделено указанное в ее аргументе количество байт памяти и которая возвращает адрес выделенного участка. Но функция возвращает адрес "неопределенного типа", т. е. непонятно, данные какого типа мы можем размещать в выделенной области. Чтобы нам настроиться на выделенную область, в которой мы хотим, например, размещать данные типа `char`, мы должны неопределенный выход функции привести к нашему типу с помощью `(char*)malloc(число)` (об адресах мы будем говорить в последующих главах).

Побитовые логические операции

Эти операции выполняются над соответствующими битами чисел, имеющими целый тип. Если операнд — не целое число, то оно автоматически преобразуется в целое. Побитовые операции таковы:

- `&` — поразрядное умножение (конъюнкция) (формат: `int a,b=3,c=4; a=b & c;`);
- `|` — поразрядное сложение (дизъюнкция) или включающее "ИЛИ" (формат: `int a,b=3,c=4; a=b | c;`);
- `^` — поразрядное исключающее "ИЛИ" (формат: `int a,b=3,c=4; a=b ^ c;`);
- `~` — операция дополнения (формат: `int a,b=3,c=4; a=b ~ c;`);
- `>>` — сдвиг разрядов вправо (формат: `int a,b=3,c=4; a=b>>c;`);
- `<<` — сдвиг разрядов влево (формат: `int a,b=3,c=4; a=b <<c;`).

Побитовые логические операции выполняются по правилам, приведенным в табл. 6.1.

Таблица 6.1. *Правила выполнения побитовых логических операций*

Значения битов		Результат операции			
E1	E2	E1 & E2	E1 | E2	E1 ^ E2	~ E1
0	0	0	0	0	1
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0

Следует отличать побитовые операции над целыми числами от логических. Например:

```
int  x=1,y=2; bool a=x && y; bool b=x & y;
```

здесь `a` = 1, а `b` = 0, потому что: из `a=x && y` следует: т. к. `x` ≠ `0` и `y` ≠ `0`, то по определению операции `&&` результат будет ис-

тинен, а в языке С истинный результат имеет значение 1. С другой стороны, для поразрядного "И" получим: `x = 01`, `y = 10` в двоичной системе счисления, в которую надо перевести операнды, чтобы выполнить побитовую операцию `&`. Тогда получим: `01 & 10 = 00`.

Операции и выражения присваивания

Выражения вида `i=i+2;` можно записывать в виде `i+=2;` (читается: к `i` добавить 2) Это правило распространяется на операции: `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `~`.

Приведем текст программы, которая подсчитывает число ненулевых битов целого числа и использует операцию сдвига (листинг 6.3).

Листинг 6.3

```
//---------------------------------------------------------

#pragma hdrstop

#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>

//---Проверка побитовых логических операций-----------------
//--- Функция подсчета количества битов в целом числе--------

 int bitcount(unsigned int n)
  {
   int b;
   for(b=0; n != 0; n>>=1)
    if(n & 01)     //01 — восьмеричная единица
     b++;
```

```
   return(b);
  }
//-------------------------------------------------------------
int main()
{
  int n=017; //восьмеричное число (4 единицы)
  printf("The unit's quantity in n=%d\n",bitcount(n));
  getch();
}
//-------------------------------------------------------------
```

Суть алгоритма функции `bitcount()` состоит в следующем: в цикле, который идет по переменной цикла `b`, целое число `n` сравнивается с восьмеричной единицей (она же — двоичная единица) с помощью операции `&` (И). Так как число 01 можно представить и как, например, 00000001, то учитывая, как работает операция `&`, можно сказать, что результатом вычисления выражения `(n & 01)` будет значение последнего бита числа `n`. Действительно: если в последнем бите числа `n` — ноль, то выражение `(n & 01)` даст тоже ноль, если — единица, то `(n & 01)` даст единицу. Следовательно, выражение `(n & 01)` "просматривает" все биты числа `n`. То есть в теле оператора, содержащего это выражение, можно подсчитывать, сколько раз выражение `(n & 01)` было равно единице, что на самом деле означает, сколько единиц содержит число `n`. Просмотр содержимого числа `n` происходит за счет сдвига его содержимого вправо на один бит в каждом цикле. Мы видели, что выражение `n>>=1` равносильно выражению `n=n >>1`, т. е. "сдвинуть содержимое переменной `n` на один разряд вправо и результат записать в `n`". Поэтому действие будет происходить так: сначала в заголовочной части оператора `for` вычисляется значение переменной цикла `b`, которая получает значение ноль. Затем там же, в заголовочной части `for`, вычисляется выражение, определяющее условие продолжения-завершения цикла `(n !=0)`. Следовательно, пока мы подсчитываем значение переменной, в которой мы накапливаем единицы, должно быть ненулевым.

Нулевым оно может стать потому, что после каждого цикла до перехода на новый цикл мы сдвигаем вправо на один разряд содержимое n, что, в конце концов, приведет к тому, что в значении n останутся только нули. Но здесь надо иметь в виду следующее: число, которое хранится в переменной типа int, может иметь знак. Точнее знак есть всегда, но он может быть отрицательным. При сдвиге вправо освобождающиеся биты будут заполняться содержимым знакового разряда. Если число положительное, то ничего страшного в этом нет: его знаковый разряд содержит ноль, и освобождающиеся при сдвиге вправо биты будут заполнены нулем. Это так называемый арифметический сдвиг числа. Если же число отрицательное, то в его знаковом разряде будет единица, и ею станут заполняться освобождающиеся при сдвиге биты. Вот этого-то нам как раз и не надо! Чтобы избежать этой неприятности, следует сдвигаемое вправо число объявить с атрибутом unsigned (у нас так и объявлена переменная n). В этом случае освобождающиеся от сдвига вправо разряды станут заполняться не знаковым разрядом, а нулем. Такой сдвиг называют логическим.

Условное выражение

Это конструкция вида: e1 ? e2 : e3, где e1, e2, e3 — некоторые выражения. Читается эта конструкция так: "Если e1 отлично от нуля (истинно), то значением этой конструкции будет значение выражения e2, иначе — e3". Пользуясь условным выражением, можно упрощать некоторые операторы. Например, вместо того, чтобы писать `if(a<b) z=a; else z=b;`

можно записать: `z=(a>b) ? a : b;`

Операторы и блоки

Если за любым выражением, например, x=0, i++ или printf(…) стоит точка с запятой, то такое выражение в языке C называется *оператором*. Таким образом, x=0; — оператор, i++; — оператор, printf(…); — оператор. Фигурные скобки {} служат для объединения операторов в блоки. Такой блок синтаксически эквивалентен одному оператору, но точка с запятой после блока не

ставится. Исходя из этого определения, можно записать формат задания операторов `for`, `while`, `if`:

```
for(выражения)    while(выражение)    if(выражение)
   блок    блок    блок
```

Если в блоке всего один оператор, то фигурные скобки можно опустить. В блоке можно объявлять переменные, но следует помнить, что они будут локальными, т. е. неизвестными за пределами блока.

Конструкция *if-else*

Эта конструкция используется при необходимости сделать выбор. Синтаксис этой конструкции таков:

```
if(выражение)  блок  else блок
```

Работает эта конструкция так: вычисляется выражение в скобках оператора `if`. Если значение выражения истинно, то выполняется тело `if`: блок операторов, если значение выражения ложно выполняется тело `else`: блок операторов. Часть `else` является необязательной: если указанное в `if` выражение ложно, то управление передается на выполнение следующего за `if` оператора, т. е. тело `if` не выполняется (как ни покажется странным замечание, но его приходится делать: `if` и его тело — это одно целое! Их нельзя разрывать. Это же касается и операторов `while`, `for`). Часть `else` самостоятельно не применяется.

Конструкция *else-if*

Когда в соответствии с реализуемым в программе алгоритмом приходится делать многовариантный выбор, применяют конструкцию вида:

```
if(выражение)
        блок
        else if(выражение)
        блок
        else if(выражение)
```

```
блок
else if(выражение)
блок
else
блок
```

Работает эта конструкция так: последовательно вычисляется выражение в каждой строке: если оно истинно, то выполняется тело (блок операторов) и происходит выход из конструкции на выполнение следующего за ней оператора. Если выражение ложно, начинает вычисляться выражение в следующей строке и т. д. Последняя часть конструкции (`else блок`) не обязательна.

Приведем пример функции поиска заданного элемента в упорядоченном по возрастанию элементов числовом массиве. Даны массив `v[n]` и число `x`. Надо определить, принадлежит ли `x` массиву. Так как элементы массива предполагаются упорядоченными по возрастанию их значений, поиск проведем, применяя метод половинного деления (иначе называемый двоичным поиском). Суть метода такова: рассматривается отрезок, на котором расположены все элементы числового массива. Если массив `v[n]` имеет размерность `n`, то отрезок, на котором расположены номера его элементов, это `[0, n-1]`, потому что номер первого элемента массива будет 0, а последнего — `(n-1)`. Этот отрезок делится пополам. Средняя точка отрезка вычисляется как `j = (0 + (n-1))/2`. В этой средней точке вычисляется значение `v[j]` и проверяется: значение `x` больше, меньше или равно `v[j]`. Если `x < v[j]`, значит, `x` находится слева от середины отрезка, если `x > v[j]`, то `x` находится справа от середины отрезка, если `x = v[j]`, значит, `x` принадлежит массиву. В последнем случае программу надо завершить либо рассматривать ту из половин отрезка, в которой, возможно, содержится `x` (`x` может и не содержаться в массиве, т. е. не совпадать ни с одним из элементов массива), и делить половину отрезка пополам и проверять, как первый отрезок.

Когда же следует остановиться? Один вариант останова мы уже видели: когда значение `x` совпадет с одной из середин отрезка. А второй — когда отрезок "сожмется" так, что его нижняя граница (левый край) совпадет с верхней (правый край). Это про-

изойдет обязательно, потому что мы станем делить отрезок пополам и в качестве результата брать целую часть от деления, поскольку, как мы видели выше, ее надо использовать в качестве индекса массива. А индекс массива это величина обязательно целая, т. к. это порядковый номер элемента массива.

Например, массив из трех элементов будет иметь отрезок `[0, 2]`. Делим пополам и получаем (0 + 2) / 2 = 1. То есть имеем два отрезка `[0, 1]` и `[1, 2]`. Если станем искать в `[0, 1]`, то придется находить его середину: (0 + 1) / 2 = 0 (мы помним, что операция `/` при работе с целыми операндами дробную часть результата отбрасывает и оставляет только целую часть, что нам и требуется для нахождения индекса массива). Видим, что левый конец отрезка (0) совпал с правым (0), т. к. правый конец нового отрезка получился равным нулю. Вот в этот момент деление пополам надо прекратить и завершить программу. Если в нашем случае получится так, что отрезок сжался в точку, а число `x` не сравнялось ни с одним элементом массива `v[]`, надо вывести об этом событии информацию: например, `-1`. Если обнаружено, что `x` содержится в массиве `v[]`, то надо вывести номер элемента `v[]`, на котором и произошло совпадение.

Примечание

Значение индекса массива, на котором произошло совпадение, положительно. Если же совпадения не обнаружено, то значение "индекса" — отрицательно.

Это все необходимо для того, чтобы при обращении к функции поиска вхождения проверить результат поиска: входит ли число `x` в массив `v[]` или не входит. Текст программы представлен в листинге 6.4.

Листинг 6.4

```
//-------------------------------------------------------------
#include <stdio.h>      //for getchar(),…
#include <conio.h>
#include <stdlib.h>  //for atoi()
```

```
#define eof 27          //Esc
#define maxline 100

//-------Ввод строки с клавиатуры----------
getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }
//----------------------------------------
binary(int x,int v[], int n)

 /*ищет в массиве v[n] элемент со значением "x"
   n - размерность массива*/
 {
  int low,high,mid;
  low=0;
  high=n-1;
  while(low <= high)
   {
    mid=(low+high)/2;
    if(x < v[mid])
     high=mid - 1;
     else if(x > v[mid])
      low=mid + 1;
     else
      return(mid);       //found
    } //while
  return(-1); //not found
 }
```

```
//------------------------------
main()
{
  int v[maxline]={0,1,2,3,4,5,6,7,8,9};
  int c,i,x;
  char s[maxline];
  do
   {
    printf("enter your new <x> >");
    getline(s,maxline);
    x=atoi(s);
    i=binary(x,v,10);
    if(i != -1)
     printf("entrance found \n");
    else
     printf("entrance not found\n");
    printf("Continue-anykey, exit-Esc\n");
   }
  while((c=getchar()) != eof) ;
    /*это - конец оператора do-while. Нам требовалось,
      чтобы тело while выполнилось хотя бы один раз */
}    //main()
```

Рассмотрим работу функции `binary()`.

В переменных `low`, `high`, `mid` размещаются соответственно текущие значения нижней границы отрезка, верхней границы отрезка и его середины. Если значение числа `x` находится в левой половине поделенного отрезка, то изменяется отрезок поиска: переменная `low` остается без изменения, а переменная `high` сдвигается на середину (поэтому этой переменной присваивается значение середины, в результате получается отрезок, являющийся левой половиной предыдущего отрезка). Если значение числа `x` находится в правой половине поделенного отрезка, то изменяется отрезок поиска: переменная `high` остается без изменения, а переменная `low` сдвигается на середину, в результате чего получается правый отрезок. Если значение `x` совпадает со

значением `v[середина отрезка]`, то функция возвращает переменную `mid` и процесс поиска прекращается. Если цикл поиска закончился, это сигнал о том, что просмотрены все элементы, и совпадения не найдено. В этом случае будет возвращено отрицательное число.

Рассмотрим работу основной программы.

Выражение `int v[maxline]={0,1,2,3,4,5,6,7,8,9};` — это инициализация (определение элементов) массива. Для простоты проверки работы функции поиска мы задали значения элементов совпадающими с номерами своих элементов. Далее с помощью функции `getline()` в строку вводится значение `x` и переводится с помощью функции `atoi()` в целое число. Затем происходит обращение к функции `binary()` и проверяется результат ее работы: проверяется равно ли возвращенное ею значение −1.

Все эти операторы помещены в блок — тело оператора `do-while` — и выполняются в цикле, пока не будет нажата клавиша <Esc>. Такая структура дает возможность вводить разные значения переменной `x`. Результат работы программы приведен на рис. 6.1.

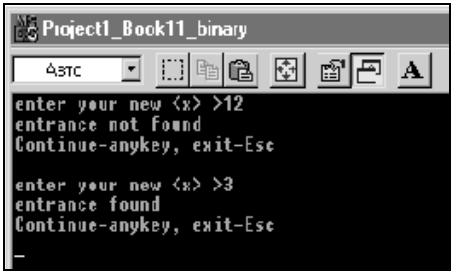


Рис. 6.1. Результат работы программы листинга 6.4

Переключатель *switch*

При большом многовариантном выборе использование комбинации `if-else` дает довольно запутанную картину. В таких ситуациях удобнее использовать специальный оператор `switch` —

оператор выбора одного из многих вариантов. Его называют также переключателем. Поясним работу оператора на примере программы подсчета количества встречающихся символов `a`, `b`, `c`, `d` во введенной с клавиатуры строке. Текст программы представлен в листинге 6.5.

Листинг 6.5

```
//-------------------------------------------------------------
#include <stdio.h>      //for getchar()….
#include <conio.h>


#define eof 27          //<Esc>
#define m 5             // количество счетчиков в операторе
switch


//----Функция подсчета символов ----------------------------
/* char c - входной символ, подсчет которого ведется(сколько
раз встретится)
    int v[] - с помощью элементов этого массива организованы
счетчики
    char s[]- сюда помещаются символы, которые подсчитываются
(для их последующей распечатки) */


CountSimb(char c,int v[],char s[])
 {
  int i;
  switch(c)
   {
    case 'a':
        v[0]++;
        s[0]=c;
        break;
    case 'b':
        s[1]=c;
```

```
        v[1]++;
        break;
    case 'c':
        v[2]++;
        s[2]=c;
        break;
    case 'd':
        v[3]++;
        s[3]=c;
        break;
    default:          //все прочие введенные символы попадают
                      //в этот блок
        v[4]++;
        s[4]='!';   /* признак "прочие символы" (введен для
                       печати: чтобы было понятно,
                       что счетчик относится к "прочим") */
       break;
   }
 }
//------------------------------


main()
{

  int c,i,a[m];
  char s[m];
  for(i=0; i < m; i++)
    a[i]=0;
  printf("enter your characters; the Esc is the last to be
entered >");
  i=0;
  while((c=getchar()) != eof)
   {
    CountSimb(c,a,s);
    i++;
   }
```

```
   for(i=0; i < m; i++)
    printf("Key=%c count =%d\n",s[i],a[i]);

   getch();
} //main()
```

Использование оператора `switch` демонстрируется с помощью его включения в функцию `CountSimb(char c,int v[],char s[])`, параметры которой описаны перед ее определением. У самого оператора `switch` есть заголовочная часть, заключенная в круглые скобки, и тело — блок операторов. В заголовочной части указано имя переменной, значение которой будет анализироваться оператором, и в зависимости от значения этой переменной произойдет передача управления в тот или иной участок блока.

Примечание

В заголовочной части оператора может быть расположено не только имя переменной, но и выражение целого типа, но никак не типа `float` или типа строки символов.

Участки блока определяются ключевым словом `case` (случай), после которого через пробел стоит конкретное значение анализируемой переменной. В нашем случае переменная описана как символ, поэтому конкретное ее значение, определяющее начало участка блока, написано в соответствии с правилами записи символьных констант: например, `'d'`. Если бы переменная была типа `int`, то надо было бы писать, например, `5`. После конкретного значения выражения в заголовочной части обязательно стоит символ двоеточия, обозначающий начало участка обработки данного случая: это как бы метка начала участка обработки, относящегося к данному случаю. Работа внутри тела оператора `switch` организована так: анализируется выражение заголовочной части и управление передается на выполнение того участка тела, значение в метке которого совпадает со значением выражения в заголовочной части. На участке могут находиться обычные операторы. Они должны быть завершены оператором `break` (прервать), который прервет выполнение `switch` и передаст управление следующему за телом `switch` оператору. Оператор

`break` прерывает не только оператор `switch`, но и `while`, и `for`. Если в конце участка не поставить `break`, то программа перейдет к следующему участку, потом к следующему и т. д. В конце концов она дойдет до конца тела оператора `switch` и выйдет из него. Этим свойством часто пользуются. Например, надо обработать какие-то значения переменной в заголовочной части, но для символов `a` и `b` требуется выполнить общий алгоритм. В этом случае в теле `switch` можно записать:

```
{
   case 'a':
   case 'b':
         операторы
       break;
}
```

Какой бы из символов — `a` или `b` — не поступил на вход переключателя, все равно будут выполняться одни и те же операторы.

Но в теле `switch` мы видим еще одно ключевое слово: `default`. Это "метка" участка "прочие": все значения заголовочного выражения, которые отличаются от значений, указанных в переменной `case`, будут приводить на этот участок, где можно располагать свои операторы, в том числе `break`.

В основной программе сначала инициализируется массив `a`, в элементах которого будут накапливаться данные по встреченным символам. Затем запрашивается строка символов (`getchar()` иначе и не работает: надо набрать строку, закончить ее клавишей <Esc>, а затем нажать <Enter>), после организуется обработка каждого символа в цикле и передача его в функцию, в которой находится оператор `switch`. В эту функцию также передается имя массива, в элементах которого установлены счетчики количества символов (массив `a`), и имя массива `s`, в который будут записываться сами символы, чтобы потом можно было их вывести на экран. Все прочие символы, не попадающие в переменную `case`, отмечаются в массиве `s` символом `!`. Когда встретится код <Esc>, цикл обработки символов завершится, и произойдет вывод счетчиков из массива `a`.

Уточнение по работе оператора *for*

Мы знаем, что заголовочная часть этого оператора содержит три выражения. Оказывается что любое из них может быть опущено. И даже все. Но с одним условием: точки с запятой должны остаться на своих местах. Это удобно для организации бесконечного цикла, выход из которого можно осуществить, проверяя в теле некоторые условия и пользуясь при этом оператором `break`.

Кроме того, в теле `for` могут находиться другие операторы `for`.

Оператор *continue*

Этот оператор родственен оператору `break`: он используется не для выхода из цикла, а для продолжения цикла (возврата на реинициализацию), не доходя до конца оператора цикла (`while`, `for`). Им удобно пользоваться в тех случаях, когда при выполнении тела цикла ясно, что не следует продолжать выполнение операторов дальше, а надо возвратиться на новый виток цикла. Например, имеем массив целых чисел `int A[n]`. Требуется выбрать из него только положительные числа и их обработать. Такой цикл можно построить следующим образом:

```
for(int i=0; i < n; i++)
 {
    if(A[i] <= 0)
    continue;
    другие операторы
  }
```

То есть сразу проверяем: если число отрицательное или нулевое, его не требуется рассматривать, а можно переходить к проверке следующего. Оператор `continue` передаст управление на реинициализацию цикла: на выражение `i++` в заголовочной части `for`.

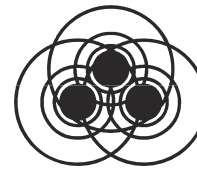
Оператор *goto* и метки

Это оператор безусловного перехода на участок программы, который помечен меткой: набором символов, оканчивающимся двоеточием и начинающимся с буквы. Структурное программирование своим появлением на свет во многом обязано этому оператору, который позволял делать такие петли в программе, что и сам автор не мог в них разобраться. Этот оператор может прервать цикл и выйти из него на метку. Например, можно написать:

```
main()
 {
     for(;;)
      {goto v1;}
     v1: printf("Hello\n");
     getch();
}
```

Увлекаться этим оператором нежелательно. В крайнем случае старайтесь передавать управление только вперед.

Глава 7



Работа с указателями и структурами данных

Указатель

Указатель — это переменная, которая содержит адрес другой переменной. Говорят, что указатель указывает на переменную того типа, адрес которой он содержит. Существует одноместная (унарная, т. е. для одного операнда) операция взятия адреса переменной `&`. Если имеем объявление `int a`, то можно определить адрес этой переменной: `&a`. Если `Pa` — указатель, который будет указывать на переменную типа `int`, то можем записать: `Pa=&a`. Существует унарная операция `*` (она называется операцией разыменования), которая воздействует на переменную, содержащую адрес объекта, т. е. на указатель. При этом извлекается содержимое переменной, адрес которой находится в указателе. Если, как мы видели, `Pa=&a`, то, воздействуя на обе части операцией `*`, получим (по определению этой операции): `*Pa=a;`

Исходя из этого, указатель объявляется так:

```
<тип переменной, на которую указывает указатель, т. е. тип переменной, чей адрес может находиться в переменной, которую мы задаем как указатель> * <имя указателя, т. е. имя переменной>;
```

Это и есть правило объявления указателя: указатель на переменную какого-то типа — это такая переменная, при воздействии на которую операцией разыменования получаем значение переменной этого же типа. Прежде чем использовать указатель, его необходимо инициализировать, т. е. настроить на какой-то кон-

кретный объект. Указатель может иметь нулевое значение, гарантирующее, что он не совпадает ни с одним значением указателя, используемого в данный момент в программе. Если мы присвоим указателю константу `ноль`, то получим указатель с нулевым значением. Такой указатель можно сравнивать с мнемоническим `NULL`, определенным в стандартной библиотеке stdio.h.

Указатель может иметь тип `void`, т. е. указывать на "ничто", но указатель этого типа нельзя путать с нулевым. Объявление `void *ptr;` говорит о том, что `ptr` не указывает на конкретный тип данных, а является универсальным указателем, способным настраиваться на любой тип значений, включая и нулевой. Примером указателя типа `void` может служить функция `malloc()`, возвращающая указатель на динамическую область памяти, выделяемую ею под объект. Она возвращает указатель типа `void`, и пользователь должен сделать приведение (casting) этого типа к типу объекта методом принудительного назначения типа (в скобках указать тип): если, например, мы выделяли память под объект типа `char`, то надо объявлять:

```
char object[]; char *P=(char *)malloc(sizeof(object));
```

Пусть некоторый указатель `Pc` указывает на переменную типа `char`, т. е. содержит адрес места памяти, начиная с которого располагается объект типа `char` (например, строка символов). Объявление такого указателя по определению будет выглядеть так: `char * Pc;` Здесь имя указателя — `Pc`, а не `*Pc`. Несмотря на такое объявление, сам указатель — это переменная `Pc`. Теперь воздействуем на него операцией разыменования. Получим `*Pc`. Это будет значение первого символа строки, на начало которой указывал указатель. Чтобы получить значение следующего символа строки, надо указатель увеличить на единицу: `Pc++` и применить `*(Pc++)`.

Вообще, какого бы типа не был объект, на начало которого в памяти указывает некоторый указатель `P` (а он указывает именно на начало объекта в памяти, когда говорят, что он указывает на объект), `P++` всегда указывает на следующий элемент объекта, `P+i` — на `i`-й элемент. Приращение адреса, который содержит указатель `P`, всегда сопровождается масштабированием размера памяти, занимаемого элементом объекта.

Указатели и массивы

Интересно соотносятся между собой указатели и массивы. Пусть имеем массив `int A[10];` и указатель, указывающий на какой-то объект типа `int: int Pa;` После объявления значение указателя никак не определено, как не определено и значение любой переменной (под них компилятор только выделяет соответствующую память). Настроим указатель на массив `A[]`. Адрес первого элемента массива занесем в указатель: `Pa=&A[0];` Как мы видели выше, `Pa+i` будет указывать на `i`-й элемент массива. То есть можно достать такой элемент из массива путем выполнения `int a=*(Pa+i);` Но по определению массива мы можем записать, что `int a=A[i];` Мы говорили ранее, что массив элементов строится в языке C так, что его имя это адрес первого элемента массива, в нашем случае `A=&A[0]` и `Pa=&A[0]`. Следовательно, `Pa=A и Pa+i = A+i` и `*(Pa+i)=*(A+i)=A[i]`. Более того, хотя `Pa` — это просто переменная, содержащая адрес, но когда она содержит адрес массива, то можно писать `Pa[i]=A[i]`, т. е. обращаться к элементам массива можно через индексированный указатель.

Пример программы, демонстрирующей вышесказанное, приводится в листинге 7.1 (все пояснения даны по тексту программы).

Листинг 7.1

```
//---------------------------------------------------------
#include <stdio.h>
#pragma argsused
#include <conio.h> //getch()
#include <stdlib.h> //atoi()
#pragma hdrstop
#define maxline 1000
#define eof   27

//-------Ввод строки с клавиатуры---------------

getline(char s[],int lim)
```

```
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }


int main(int argc, char* argv[])
{
 int A[maxline]={0,1,2,3,4,5,6,7,8,9};  //инициализация
                                        //массива
 int *Pa=&A[0];                         //настройка указателя
                                        //на массив
 char s[maxline];                       //для ввода номера
                                        //элемента массива
 int c;
 do               //для обеспечения цикличности ввода номеров
                  //элементов
  {
    printf("enter the element's number <0-9> >");
    //запрос на ввод номера элемента
    getline(s,maxline);
  //ввод номера элемента как строки символов
    int i=atoi(s);
 //преобразование номера элемента в число
  printf("i=%d A[i]=%d *(Pa+i)=%d *(A+i)=%d
%d\n",i,A[i],*(Pa+i),*(A+i),Pa[i]);
    getch();      //задержка изображения на экране
  }
  while((c=getchar() != eof)); /*для обеспечения цикличности
ввода номеров элементов: признак конца цикла ввода - <Esc>*/
}
//-------------------------------------------------------------
```

Операции над указателями

Над указателями, содержащими адрес одного и того же объекта, можно выполнять определенные операции.

- Операции отношения (>, <, и т. д.). Например, `P` и `Q` указывают на массив `A[]`. Тогда имеет смысл операция `P < Q`. Запись `P < Q` говорит о том, что `P` указывает на элемент с меньшим индексом, чем `Q`. Тогда имеет смысл и разность `Q - P`, которая определяет количество элементов между `P` и `Q`.
- Операции равенства и неравенства (==, !=).
- Указатель можно сравнивать с `NULL`, как мы видели выше.

Все остальные арифметические операции к указателям неприменимы.

Указатели и аргументы функций

Мы видели, что аргументы в функцию можно помещать либо передавая их значения, либо — ссылки на эти значения (т. е. адреса). В последнем случае значения переданных по ссылке переменных могут быть изменены в теле функции. Примером этого может служить программа, текст которой приводится в листинге 7.2.

Листинг 7.2

```
//---------------------------------------------------------


#pragma hdrstop
#include <stdio.h>
#include <conio.h>
/*функция, которая меняет местами значения переменных:
значение, которое было в переменной "a", переместится
в "b" и наоборот*/


int f(int *a, int *b)
 {
   int i=*a;
```

```
    *a=*b;
    *b=i;
  }

#pragma argsused
int main(int argc, char* argv[])
{
  int c=12;
  int d=120;
  printf("The (c,d)'s value before function application:
c=%d,d=%d\n",c,d);
  f(&c,&d);
  printf("The (c,d)'s value after  function application: c=%d
d=%d\n",c,d);
  getch();
}
//---------------------------------------------------------
```

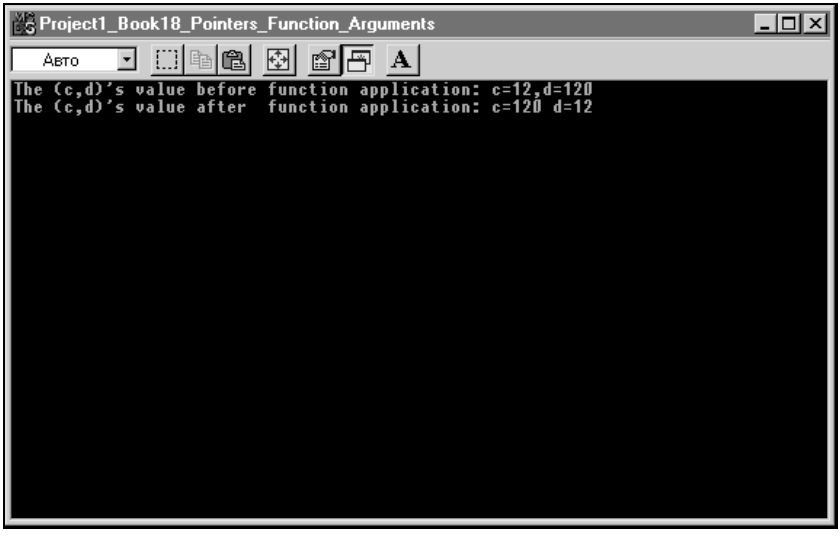


Рис. 7.1. Результат работы программы из листинга 7.2

В этой функции аргументы объявлены как указатели, следовательно при обращении к такой функции ей надо передать адреса переменных, а не их значения. А поскольку мы передали адреса переменных, то в теле функции по этим адресам можно изменять содержимое самих переменных. Например, когда мы пишем `*a=*b;` это и есть работа с адресами. Результат работы этой программы приведен на рис. 7.1.

Указатели символов и функций

Символьная константа или массив символов в языке C — это строка символов с признаком конца (символом `'\0'`). Если, например, имеем `char a[10];`, то `a` — это указатель на первый элемент массива `a[0]`. Если, с другой стороны, имеем `char *p=&a[0]`, то наряду с инициализацией `a[]="abc";` можем записать `*p="abc";`. Компилятор в обоих случаях, начиная с адреса, помещенного в указатель `p`, разместит символы `a`, `b`, `c`. Следовательно, оперирование именем массива и указателем на этот массив равносильно. Но за исключением некоторого небольшого обстоятельства: если мы хотим записать строку символов в некоторое место памяти, то при объявлении `char a[100];` компилятор выделит 100 байт для помещения строки символов, и мы сможем записать в массив `a[]` свои символы. Если же объявить указатель `char p`, то, чтобы записать символы, начиная с адреса, указанного в `p`, указатель должен быть предварительно инициализирован, т. е. ему должен быть присвоен некий адрес, указывающий на участок, где будут располагаться объекты типа `char`. Этот участок должен быть получен либо с помощью функции `malloc()`, которая возвратит указатель на выделенный участок, после чего значение этого указателя надо будет присвоить указателю `p`. Либо вы должны объявить массив символов размерности, соответствующей вводимой строке, и настроить указатель на этот массив. После этого можно работать с указателем.

Кстати, функции тоже могут быть "указателями", т. е. возвращать указатели на объекты заданного типа. В этом случае функция при ее объявлении имеет вид: `<тип объекта, на который указывает указатель> <*имя функции>(аргументы функции)`. Например, приводимая ниже функция `char *strsave(s)`. При-

ведем в качестве примера программу, работающую с указателями символьного типа (листинг 7.3). Программа содержит функции работы с символьными строками, в которых отражена работа с указателями.

Листинг 7.3

```
//-------------------------------------------------------------
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>
#include <stdlib.h>    //for atoi()
#include <string.h>
#include <alloc.h>     // for malloc()
#define maxline 1000
#define eof  27        //<Esc>



//-------Ввод строки с клавиатуры--------------

getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для подсчета количества
    return(i);
  }

//--Копирует t[] в s[] --------------------

void strcpy1(char s[],char t[])
  {
   int i=0;
```

```
   while((s[i]=t[i])!='\0')
    i++;
  }

/*--Копирует строку, на которую указывает указатель t,
 в строку, на которую указывает указатель s */

void strcpy2(char *s,char *t)
  {
      while((*s=*t) != '\0')
    {
              s++;
              t++;
    }
  }

//---------------------------------
/*выделяет по malloc() память по длине строки, на которую
 указывает s, и в эту память помещает саму строку, а затем
 выдает указатель на начало помещенной в буфер строки.*/

char *strsave(char *s)
 {
  char *p;
  int i=strlen(s)+1;
  p=(char *)malloc(i);
  if((p != NULL))
   strcpy2(p,s);
   return(p);
   /* т. к. malloc() выдает указатель типа void,
    то принудительно приводим его к типу char,
    чтобы согласовать с p */
 }

//-------------------------------------------------------
int main()
```

```
{
//Проверка strcpy1()
        printf("Enter string for strcpy1 >");
        char s[maxline],t[maxline];
        getline(t,maxline); //ввод строки с клавиатуры в t
        strcpy1(s,t);
        printf("inp.string=%s\nout.string=%s\n",t,s);

//Проверка strcpy2()
        printf("Enter string for strcpy2 >");
        getline(t,maxline); //ввод строки с клавиатуры в t
        strcpy1(&s[0],&t[0]);
        printf("input string=%s\noutput string=%s\n",t,s);

//Проверка strsave()
        printf("Enter string for strsave>");
        getline(s,maxline); //ввод строки в s
        char *p=strsave(&s[0]);
//в p указатель на память, куда записана строка из s
        printf("Saved string=%s\n",p);
        getch();
}
//-------------------------------------------------------
```

Функция `void strcpy1(char s[],char t[])` копирует массив в массив. Это происходит поэлементно с помощью оператора цикла `while`, в заголовочной части которого находится выражение, которое надо вычислить, чтобы принять решение о продолжении-завершении цикла. При вычислении выражения происходит поэлементная пересылка: пересылается один элемент, затем начинает работать тело `while`, в котором всего один оператор наращивания индекса элемента массива. Когда тело завершается, управление передается в заголовочную часть `while`, где снова вычисляется выражение, обеспечивающее пересылку следующего элемента массива в массив `s` и так далее, пока не будет переслан последний символ: `'\0'`. Когда это произойдет,

результат выражения станет равным нолю, и функция завершится. Функция не возвращает никакого значения (его тип `void`), но через параметр, адрес которого передается в функцию (массив `s[]`), возвращает копию массива `t[]`.

Параметрами функции `void strcpy2(char *s,char *t)` являются указатели типа `char`. Эта функция копирует символы, начиная с адреса, на который указывает указатель `t`, в область, на которую указывает указатель `s`. Перед применением этой функции, как отмечалось выше, надо определить область памяти, на которую указывает указатель `s` либо через массив, либо через функцию `malloc()`. В теле `strcpy2()` организован цикл посимвольного перемещения символов из одной области в другую с помощью указателей этих областей: указатель `*t` передает элемент входной области, а указатель `*s` — выходной. После того как первый элемент переслан, в теле `while` идет приращение значений указателей на единицу, которое, как мы видели выше, позволяет через указатель обратиться к следующему элементу объекта. Программа снова возвращается в заголовочную часть `while`, обеспечивающую пересылку второго элемента в область, на которую указывает `s` и так далее, пока не будет переслан последний элемент — признак конца массива. При этом значение выражения в заголовочной части `while` станет равным нулю, и оператор `while` завершит работу.

Функция `char *strsave(char *s)` копирует строку символов, на которую указывает указатель `s`, в буфер памяти, выделяемый стандартной функцией `malloc()`, и возвращает указатель на начало этого буфера, чтобы в дальнейшем можно было доставать сохраненную в нем строку. Копирование происходит с помощью ранее рассмотренной функции `strcpy2()`. Работа происходит следующим образом: длина исходной строки с учетом признака конца строки определяется с помощью стандартной функции `strlen()`. Затем по функции `malloc()` выделяется участок памяти такого же размера, как длина входной строки. Перед тем как начать копирование с использованием указателя, возвращенного функцией `malloc()`, проверяется, успешно ли сработала эта функция. То есть действительно ли выделила место в памяти. Это вопрос не праздный, т. к. свободной памяти в данный мо-

мент могло не оказаться или мог произойти какой-либо сбой, и память выделить не удалось. Поэтому проверять успешно ли сработала функция надо. Если память выделена, то `malloc()` возвращает ненулевой указатель типа `void`, который требуется привести к типу `char`, т. к. в выделенном буфере будет размещен объект типа `char`. После выделения памяти происходит собственно копирование в буфер и возврат указателя на область копирования.

При пояснении основной программы стоит обратить внимание только на проверку `strcpy2()`. Так как эта функция работает с указателями, то ей мы и передаем указатели: адреса первых элементов массивов. Результат расчета приводится на рис. 7.2.

Рис. 7.2. Результат выполнения программы листинга 7.3

Передача в качестве аргумента функции массивов размерности больше единицы

До сих пор мы передавали в качестве аргумента функции только одномерный массив. Можно передавать и массивы большей размерности. Если, например, двумерный массив передается в качестве аргумента функции, то его описание как аргумента функции может быть следующим:

```
int  m[2][13]; int m[][13];
```

Здесь `m` указывает на начало массива, поэтому компилятору достаточно знать только количество столбцов массива и начало его первого элемента.

Другой вариант описания:

```
int (*m)[13];
```

Здесь `m` указывает на начало массива.

Массивы указателей

Мы видели, что с помощью массива, объявленного, например, как `char M[n][m]`, можно задавать множество символьных строк постоянной длины. Иначе и не задать, потому что компилятор не сможет найти заданный элемент массива. Однако в жизни чаще всего приходится работать со строками переменной длины. Тогда жесткая конструкция двумерного массива для их хранения не подойдет.

Для решения этой проблемы существует конструкция, называемая *массивом указателей*. Создается одномерный массив, элементами которого служат указатели на заданный тип данных. Например, массив `char *s[10];` — это десять указателей (`s[0],s[1],…,s[9]`), каждый из которых указывает на строку, которая может быть переменной длины. Такой массив формируется так: в некоторой памяти размещается первая строка, ее адрес заносится в `s[0]`. Затем размещается вторая строка, и ее адрес заносится в `s[1]` и т. д. Чтобы обратиться к элементам такого массива, нужно воспользоваться определением указателя. Обратиться к нулевому элементу нулевой строки следует как `*s[0]`, к первому элементу той же строки как `*s[0]++` и т. д. К нулевому элементу первой строки как `*s[1]`, к ее первому элементу как `*s[1]++` и т. д.

Инициализация массива указателей на строки символов, например, `char *s[3];` будет выглядеть так:

```
char *s[3]={"Первая строка символов",
"Вторая строка символов", "Третья строка символов"};
```

В чем же различие между записями, например, `int n[10][20]` и `int *b[10]`? Под первый вариант компилятор выделяет 200 единиц памяти. И поиск элемента этого массива производится пу-

тем вычисления обычных прямоугольных индексов. При втором варианте, если предположить, что и там строки содержат по 20 элементов, под них также будет выделено 200 единиц памяти, но еще понадобится память для хранения десяти указателей. То есть памяти при втором варианте размещения данных требуется больше. Но это неудобство перекрывается тем, что в таких конструкциях можно хранить строки переменной длины, и что доступ к таким строкам происходит напрямую, по их адресам, без вычисления индексов массивов.

Указатели на функции

В языке С имеется возможность определять указатели на функции и, следовательно, обрабатывать указатели, передавать в качестве аргумента другим функция и т. д. При объявлении указатель на функцию записывается в виде:

```
<тип возвращаемого функцией значения>
(*имя функции) (список параметров);
```

Например:

```
int (*comp)(char s1,char s2);
```

Это указатель на функцию `comp(s1,s2)`, которая возвращает результат типа `int`. Если мы подействуем операцией разыменования (`*`) на этот указатель, по определению указателя записав воздействие в виде `(*comp)(s1,s2)`, то функция `comp(s1,s2)` выполнится и возвратит некое целое число. Ранее мы видели, что имена массивов можно передавать в качестве аргументов функции. Теперь, поскольку есть указатели на функции, функции можно передавать в качестве аргументов другим функциям. В таких функциях их аргументы-функции описываются как указатели на функции, а передача функций в качестве аргументов происходит указанием имени самой функции. Все остальное улаживает компилятор. То есть с функциями при передаче их в качестве аргументов другим функциям происходит то же, что и с массивами: их имена считаются внешними переменными. Приведем пример функции `gener()`, которая имеет своими параметрами две функции: ввода строки символов и подсчета количества ненулевых битов в целом числе (листинг 7.4).

Листинг 7.4

```
//-------------------------------------------------------------
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>
#include <stdlib.h>  //for atoi()
#define eof '?'
#define maxline 1000

//--- Функция подсчета количества битов в целом числе
 int bitcount(unsigned int n)
  {
   int b;
   for(b=0; n != 0; n>>=1)
    if(n & 01) //01-восьмеричная единица
     b++;
   return(b);
  }

//-------Ввод строки с клавиатуры

getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }

//функция вводит число n с клавиатуры и подсчитывает
//количество единиц в нем
```

```
 gener(int (*getline)(char s[],int lim),int
(*bitcount)(unsigned int n))
  {
    char s[maxline];
    int lim=100;
    (*getline)(s,lim);
      unsigned int n=atoi(s);
      n=(*bitcount)(n);
      return(n);
  }
//-------------------------------------------------------------


int main()
{
  int n=gener(getline,bitcount);
  printf("n=%o\n",n);
  getch();
        return 0;
}
//-------------------------------------------------------------
```

Мы уже знакомы с функциями `getline()` и `bitcount()`. Последнюю мы составляли, когда изучали операции сдвига. Она подсчитывает в целом без знака количество единиц (ненулевых битов).

Рассмотрим вызывающую функцию `gener()`. Мы видим, что оба ее аргумента описаны как указатели на функции: первый — на функцию `getline()`, второй — на `bitcount()`. Затем идет выполнение первой переданной в качестве аргумента функции. Чтобы заставить выполниться функцию, находящуюся по адресу, который содержится в указателе `getline`, надо подействовать на него операцией разыменования (по определению указателя). Получим `(*getline)(s,lim);`. Функция `getline()` выполнится и результатом ее работы станет введенное в строку `s` число, которое переводится в беззнаковое `n` с помощью функции `atoi()`. После этого выполнится функция `bitcount()`, тоже описанная

как указатель. Результат ее работы и возвращается в качестве результата функции `gener()`. Результат расчета показан на рис 7.3.

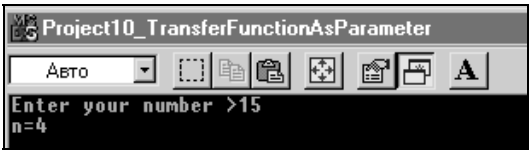


Рис. 7.3. Результат расчета программы листинга 7.4

Структуры

Объявление структур

Структуры — это такие конструкции языка C, которые объединяют в себе данные разных типов, в том числе и подструктуры (такие же структуры, но которые являются членами главной структуры). Эти конструкции полезны тем, что во многих ситуациях позволяют группировать связанные данные таким образом, что с ними можно работать как с единым целым. Как объявляется структура, покажем на примере объявления данных некоторого человека:

```
struct man
 {
   char name[80];
   char  phone_number[80];
   int  age;
int height;
};
```

Так задается шаблон будущего экземпляра структуры. Здесь `man` — имя шаблона. То, что находится в теле, ограниченном фигурными скобками, это члены структуры-шаблона. Под такое объявление компилятор память не выделяет. На основе такого шаблона создается экземпляр структуры, под который память

уже выделяется и с которым можно работать в программе. Экземпляры структуры создаются несколькими путями:

- по шаблону `man`:

```
struct man friends[100],others;
```

Здесь созданы два экземпляра структуры: один — это массив структур (каждый элемент такого массива представляет собой структуру шаблона `man`), другой — обычный экземпляр по шаблону `man`. В языке C++ ключевое слово `struct` можно опускать;

- при объявлении шаблона:

```
struct man
 {
   char name[80];
   char  phone_number[80];
    int  age;
   int height;
}others;
```

Здесь создан один экземпляр структуры — `others`;

- с помощью квалификатора типа `typedef`, который изменяет имя шаблона и позволяет воспользоваться новым именем в качестве типа данных:

```
char name[80];
typedef char phone_number[80];
   int age;
   int height;
}NewTempl;
```

Теперь можно писать: `NewTempl d1,d2[20],*p;` Здесь объявлено три переменных типа `NewTempl`: экземпляр `d1` структуры шаблона `man`, массив структур `d2[20]` и `p` — указатель на структуру.

Примечание

При объявлении шаблона структуры члены-данные структуры объявляются такого же формата, как если бы они были вне структу-

ры: тип, имя, точка с запятой. Отличие состоит в том, что вне структуры их нельзя сразу при объявлении инициализировать.

Приведем пример вложенной структуры, т. е. такой структуры, которая является членом другой структуры:

```
struct  date {
    int day;   //день недели
    int month;   //номер месяца
    int year;   //год
    char monthname[4];   //название месяца
    };
 struct person {
    char name[30];   //имя
    char adress[70];   //домашний адрес
    long mailcod;   //почтовый код
    float salary;   //заработная плата
    struct date birthdate;   //дата рождения
    struct date hiredate;   //дата поступления на работу
    }emp[1000],*p;
```

Это типичный пример объявления личной карточки работника (реальная карточка содержит намного больше данных). Здесь объявлен указатель на структуру и массив структур шаблона `person`. Если такой массив заполнить, то получим данные на 1000 работников.

Примечание

Указатель на структуру это не экземпляр структуры (экземпляр структуры объявляется как `emp[]`), а указатель, которому в дальнейшем будет присвоен адрес некоторой структуры, с элементами которой можно будет работать через указатель.

Обращение к элементам структур

Чтобы обратиться к элементам структуры, надо после имени экземпляра структуры поставить точку, а после имени указателя на данную структуру поставить стрелку вправо (->). Затем к этим

именам приписать имя члена структуры, к которому надо обратиться. Если требуется обратиться к членам вложенной структуры, то следует продолжить операции с точкой или стрелкой вправо с именем подструктуры, а затем с именем ее члена. Примеры обращения к членам экземпляров структуры:

`emp[0].name, emp[521].salary, emp[121].hiredate.year`

Допустим, `p=&emp[1]`. В этом случае `p->adress` — это адрес работника, который содержится в экземпляре структуры `emp[1]`, а год поступления на работу — `p-> hiredate->year`. Однако существуют некоторые ограничения.

- Членом структуры может быть любой тип данных (`int`, `float`, массив, структура), но элемент структуры не может иметь тот же тип, что и сама структура. Например:

  ```
  struct r {int s; struct r t};
  ```

  Такое объявление структуры неверно, т. к. `t` не может иметь тип `r`. При этом указатель на тот же тип разрешен. Например:

  ```
  struct r {int s; struct r *t};
  ```

  Такое объявление верно.

- В языке C членом структуры не может быть функция, а указатель на функцию может быть членом структуры. Например:

  ```
  struct r
    {
    int s;
      (*comp)(char *a,char *b);
    };
  ```

  Такое объявление верно.

- В C++ функция может быть членом структуры. Дело в том, что в C++ структуры рассматриваются как класс, т. е. для членов структуры могут быть определены спецификации `public` (всегда определена по умолчанию), `protected` и `private`.

В качестве примера приведем текст простейшей программы, в которой функция является членом структуры (листинг 7.5).

Листинг 7.5

```
//---------------------------------------------------------

#pragma hdrstop
#include <stdio.h>
#include <conio.h>

//---------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
  struct aaa
    {
  public:          //определена по умолчанию (приведена для
                   //примера)
     int i;
     int f(int a)   //функция-член структуры
        {           //объявлена прямо в структуре
         return(-a);
        };
    }*bbb;
      int a=bbb->f(15);
      printf("a=%d\n",a);
    getch();
}
//---------------------------------------------------------
```

Результат работы этой программы показан на рис. 7.4.

- В языке C внешнюю или статическую структуру можно инициализировать. Например, имеем шаблон:

```
struct  date {
  int day;   //день недели
  int month;   //номер месяца
```

```
    int year;    //год
    char monthname[4];    //название месяца
    };
```

В этом случае можем инициализировать структуру:

```
struct date d1={4,5,2003,sept};
```

Инициализация массива структур будет задаваться так:

```
struct a {char *s; int i;}m[3]={"u1",0,
   "u2",0,
   "u3",0
  };
```

Рис. 7.4. Результат работы программы из листинга 7.5

- Присваивать значения одной структуры другой структуре разрешено только для экземпляров одной структуры. Например, существует структура:

  ```
  struct A {int i; char d}a,a1; и struct B
  {int i; char d}b;
  ```

  В этом случае можно выполнить `a = a1;` или `a1 = a;` Но операцию `a=b;` выполнить нельзя, т. к. `a` и `b` считаются относящимися к шаблонам разного типа (у их шаблонов разные имена и этого достаточно, чтобы считать их разными, хотя по структуре они совпадают).

Структуры и функции

Функция может возвращать структуру или указатель на структуру. Например, если объявить структуру `mystruct func1(void);`, то функция `func1()` возвратит структуру. Для структуры `mystruct *func2(void);` функция `func2()` возвратит указатель на структуру.

Структура может передаваться в качестве аргумента функции следующими способами:

- непосредственно:

```
void func1(mystruct s);
```

- через указатель:

```
void func2(mystruct *sptr);
```

- в языке C++ через ссылку:

```
void func3(mystruct &sref);
```

Чем отличаются понятия "ссылка" и "указатель"? Ссылка — это непосредственно адрес, а указатель — переменная, содержащая адрес (подобное различие существует между константой и переменной).

Программы со структурами

Приведем примеры программ, где используются функции, имеющие на входе структуру и возвращающие либо саму структуру, либо указатель на нее.

Функция возвращает структуру

Приведем пример программы, в которой функция возвращает структуру (листинг 7.6).

Листинг 7.6

```
//-------------------------------------------------------------
#include <stdio.h>  //for getchar(),putchar()
#include <conio.h>
#include <stdlib.h>  //for atoi()
#include <string.h>
#include <alloc.h>  // for malloc()

#define eof  27
#define maxline 1000
```

```
//-------Ввод строки с клавиатуры


getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;  //для учета количества
    return(i);
  }
//----------------------------------------
struct key
 {
  char *keyword;
  int keycount;
 }tab[]={"break",0,
         "case",0,
         "char",0,
         "continue",0,
         "end",0
        },bbb;
struct key BinaryInStruc(char *word,struct key tab[],int n)
/*Ищет в массиве структур слово, находящееся в word
n - размерность массива, которая не должна быть задана
большей,чем количество инициализированных элементов массива.
Возвращает структуру (элемент массива tab[]), в которой
находится слово, заданное в word, либо tab[] последнего
обработанного индекса (можно было бы возвратить tab[] любого
существующего индекса), в котором значение keycount равно -1
(сигнал того, что заданное слово в массиве структур
не обнаружено)*/
 {
  int low,high,mid,cond;
  low=0;
```

```
  high=n-1;
  while(low < high)
   {
    mid=(low+high)/2;
    if((cond=strcmp(word,tab[mid].keyword)) < 0)
     high=mid - 1;
     else if(cond > 0)
      low=mid + 1;
     else
      return(tab[mid]);       //found
    } //while
    tab[mid].keycount=-1;
  return(tab[mid]); //not found
 }
//------------------------------


main()
{
  char s[maxline];
  int c;
  do
   {
    printf("enter your new string >");
    getline(s, maxline);
    bbb =BinaryInStruc(s,tab,5);
    if(bbb.keycount!= -1)
     printf("Found string = %s\n",bbb.keyword);
    else
     printf("not found\n");
 //   getch();
   }
  while((c=getchar()) != eof) /*здесь, как и в операторе get-
line(), надо использовать getchar(),а не getch(), иначе не
будет останова на вводе в getline() при повторном обращении
*/
```

```
    ; //конец оператора do-while
 //нам надо было, чтобы тело while выполнилось хотя бы один раз
}    //main()
//-------------------------------------------------------------
```

Здесь приведена функция `BinaryInStruc`, возвращающая структуру:

```
struct key BinaryInStruc(char *word,struct key tab[],int n)
```

Сама структура определена до объявления функции: объявлен шаблон `key` и по этому шаблону задан массив структур `tab[]` и один экземпляр `bbb`. Массив структур инициализирован: заданы значения только символьных строк, т. к. мы собираемся искать нужную строку, задавая на входе ее образец. Массив упорядочен по возрастанию строк, т. к. для поиска будет применяться метод деления отрезка пополам. Алгоритм этого метода рассматривался нами ранее. Значение `keycount` для поиска не используется, а используется только для возврата: если не найдена структура, в которой содержится заданное с клавиатуры слово, то возвращается структура, у которой значение `keycount` будет равно −1.

В теле функции реализован метод двоичного поиска: концы отрезка, на котором располагаются индексы массива `tab[]`, постоянно находятся в переменных `low` и `high`, а средняя точка — в переменной `mid`. Сравнение значений в `word` и `tab[]` осуществляется с помощью рассмотренной выше функции `strcmp(word,tab[mid].keyword)`. Если в средней точке строки в переменной `word` и в переменной `tab[mid]` значения `keyword` совпадают, возвращается таблица структур в этой точке, иначе возвращается таблица структур в последней средней точке со значением `keycount`, равным −1.

В основной программе запрашивается слово, которое вводится функцией `getline()`, а затем передается вместе с массивом структур (таблицей `tab[]`) в качестве параметров функции `BinaryInStruc()`, которая возвращает структуру, значение которой, в свою очередь, присваивается экземпляру `bbb` этого же шаблона (как мы видели ранее, эту операцию можно применить к структурам одинаковых шаблонов). Результат работы программы приводится на рис. 7.5.

Рис. 7.5. Результат работы программы листинга 7.6

Функция возвращает указатель на структуру

Изменим предыдущую программу так, чтобы функция возвращала вместо индекса массива указатель на структуру, в которой найдено или не найдено заданное слово (листинг 7.7).

Листинг 7.7

```
//-------------------------------------------------------------
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>
#include <stdlib.h>  //for atoi()
#include <string.h>
#include <alloc.h>     // for malloc()

#define eof 27
#define maxline 1000

//-------Ввод строки с клавиатуры

getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
```

```
      i++;     //для учета количества
    return(i);
  }
//----------------------------------------
struct key
 {
  char *keyword;
  int keycount;
 }tab[]={"break",0,
         "case",0,
         "char",0,
         "continue",0,
         "end",0
        },*bbb;

struct key *BinaryInStruc(char *word,struct key tab[],int n)

/*Ищет в массиве структур слово, находящееся в word
n - размерность массива, которая не должна быть задана
большей, чем количество инициализированных элементов массива.
Возвращает указатель на структуру типа key, в которой
находится слово, заданное в word, либо возвращает NULL (сиг-
нал того, что заданное слово в массиве структур
не обнаружено)*/
 {
  int cond;
  struct key low =&tab[0];    /*здесь low и high -
  это указатели на первый и последний элементы таблицы*/
  struct key *high=&tab[n-1];
  struct key *mid;  /*здесь будет указатель на средний
  элемент таблицы*/

  while(low < high)  /*указатели можно сравнивать*/
   {
    mid=low + (high - low)/2;
```

```
/*разность между указателями – это число элементов массива,
которое можно делить, а поскольку операция "/" даст целое
число, то его можно прибавить к указателю low*/

if((cond=strcmp(word,mid->keyword)) < 0)
     high=mid - 1;   /*от указателя можно вычесть целое
     число, в результате получим указатель на предыдущий
     элемент*/
else if(cond > 0)
      low=mid + 1;
     else
       return(mid);  /*found (возврат указателя на найденный
       элемент таблицы, т. е. на структуру)*/
} //while
  return(NULL); //not found
 }
//------------------------------

main()
{
  char s[maxline];
  int c;
  do
   {
    printf("enter your new string >");
    getline(s, maxline);
    bbb =BinaryInStruc(s,tab,5);
  //bbb объявлен как указатель на структуру типа key
    if(bbb != NULL)
     printf("Found string = %s\n",bbb->keyword);
    else
     printf("not found\n");
 //   getch();
   }
  while((c=getchar()) != eof) ; //конец оператора do-while
    //требовалось, чтобы тело while выполнилось хотя бы один раз

}    //main()
```

Пояснения к этой программе даны в ее тексте. Результат работы совпадает с результатом работы предыдущей программы.

Программы упрощенного расчета заработной платы одного работника

В этой программе создана функция расчета, которой передается в качестве параметра указатель на структуру (листинг 7.8).

Листинг 7.8

```
//-------------------------------------------------------
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>  //for atoi()

#define eof 27
#define maxline 1000



//-------Ввод строки с клавиатуры

getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }
//----------------------------------------
struct zrp             //структура данных по зарплате
 {
  char *name;         //имя работника
```

```
  float stavka;      //оплата за один рабочий день
  float nalog;      //величина налога
 }emp[]={"Ivanov",200,0.1,
         "Petrov",300,0.2,
         "Sidorov",400,0.3
        };
/*Здесь задан массив экземпляров структур: одна структура
содержит данные на одного работника. Массив сразу
проинициализирован.*/

/*Функция начисления зарплаты одному работнику.
RabDn - количество отработанных дней*/

float zarplata(struct zrp *z,int RabDn)
 {
  return(z->stavka * RabDn * (1 - z->nalog));
 }



//-------------------------------

main()
{
   char c,s[maxline];
   struct zrp *p1;
   /*определение размера массива:*/
   int RazmMas = sizeof(emp)/sizeof(zrp);
  do
   {
   /*ввод номера работника:*/
m:  printf("enter emp's number >");
    getline(s, maxline);
    int i=atoi(s);
    if(i < RazmMas)   //контроль количества заданных
                      //элементов массива
```

```
     p1=&emp[i];
    else  goto m;
    /*ввод количества отработанных дней:*/
    printf("enter work's days amount >");
    getline(s, maxline);
    i=atoi(s);
    float zp = zarplata(p1,i);   //обращение к функции
                                 //расчета зарплаты
    printf("%s %6.2f\n",p1->name,zp);
   }
 while((c=getchar()) != eof)
     ; //конец оператора do-while
}    //main()
```

Рекурсия в структурах

В структурах возможна рекурсия, т. е. структуры могут ссылаться сами на себя. Допустим, у нас имеется списковая структура типа: "Слово (строка символов)", "Счетчик количества каждого искомого слова в тексте", "Указатель на предыдущую структуру, в которой встречается данное слово", "Указатель на последующую структуру, в которой встречается данная строка".

Такая структура может быть представлена в виде рекурсивного шаблона с указателем на такую же структуру:

```
struct tnod                    (***)
 {
   char *word;
   int count;
   struct tnod *left;
   struct tnod *right;
 }t,*p;
```

Если, например, `p=&t`, то доступ к элементам структуры будет таким:

```
p->word, p->left, p->right
```

Приведем пример программы, которая подсчитывает количество встречающихся в некотором тексте слов. Эта программа передает введенные с клавиатуры слова специальной функции, которая по ним строит в памяти так называемое "двоичное дерево".

Примечание

Двоичное дерево — это такая конструкция, которая отображает связь между данными, исходя из того, что данные находятся в так называемых узлах. Под узлом понимается переменная типа "структура" (ее еще называют записью). Первая запись двоичного дерева (его корень) содержит один узел. В узле содержится некая полезная информация, для обработки которой мы и строим само дерево, а также два указателя на нижестоящие узлы. Из корневого узла "вырастают" всего две "веточки": левый нижний узел (первый указатель) и правый нижний узел (второй указатель). От каждого из этих узлов тоже "вырастает" по две "веточки" и т. д. Можно сказать, что в такой конструкции каждый узел — это тоже двоичное дерево (корень, из которого выходят две "веточки").

В нашем случае двоичное дерево строится так:

- никакой узел этого дерева не содержит более двух ближайших потомков (детей);
- в корневом узле слова нет;
- первое поступившее слово записывается в корневой узел, а остальные поступающие будут распределяться так: если поступившее слово меньше первого, то оно записывается в левое поддерево, если больше корневого — в правое;
- каждое слово будет находиться в структуре типа `(***)`;
- слово записывается в переменную `word`, а счетчик `count` в данной структуре устанавливается в единицу: слово пока что встретилось (в нашем случае введено) один раз.

Если встретится еще такое же слово, то к счетчику `count` в данной структуре добавится единица. При этом в корневой структуре в переменной `left` формируется указатель на структуру, в которую записалось слово меньшее, чем слово в корневой структуре. Если же поступило слово, которое больше слова в корневой структуре, то оно записывается, как уже говорилось, в

отдельную структуру (узел), а указатель на этот узел записывается в переменную `right` корневой структуры.

Потом вводится третье слово. Оно опять сравнивается со словами, находящимися в структурах дерева `(***)`, начиная с корневой структуры и далее по тем же принципам, что описаны выше. Если поступившее слово меньше корневого, то дальнейшее сравнение идет в левой части дерева. Это означает, что из корневой структуры выбирается указатель, хранящийся в переменной `left`, по нему отыскивается следующее меньшее, чем в данном узле слово, с которым и станет сравниваться поступившее. Если поступившее слово меньше, то из нового узла извлекается указатель, хранящийся в переменной `left`, и по нему выбирается следующий "левый" узел и т. д.

Если же больше узла нет (в последнем узле `left` будет ноль), то поступившее слово записывается в новый формируемый узел, в обе части `left` и `right` которого записываются нули (признак того, что этот узел последний). Если же на каком-то этапе поступившее слово оказывается больше, чем слово в левом узле, то рассматривается указатель `right` этого узла и по нему определяется структура (узел), где находится следующее меньшее этого слово, с которым станет сравниваться поступившее. Если оно окажется больше, то дальше пойдет проверка следующей "правой" структуры и сравнение с ее данными, если же меньше, то процесс перейдет на левую часть поддерева: движение по сравнению пойдет по "левым" структурам.

В итоге получим двоичное дерево: от каждого узла будут выходить не более двух подузлов и таких, что в левом подузле всегда будет слово меньшее, чем в узле, а в правом подузле всегда будет находиться слово большее, чем в подузле. Так как физически каждое слово будет находиться в отдельной структуре типа `(***)`, то в этой же структуре в переменных `left` и `right` будут располагаться указатели на структуры, где хранятся подузлы данной структуры. И все это дерево располагается в памяти.

Программа, реализующая рекурсию в структурах, приводится в листинге 7.9. За ней следует рис. 7.6, на котором отражен результат выполнения этой программы.

Листинг 7.9

```
//-------------------------------------------------------------
#include <stdio.h>
#include <conio.h>
#include <string.h>   //strcpy()
#include<alloc.h>

#pragma hdrstop

#define maxline 1000
#define eof 27
//-------Ввод строки с клавиатуры
getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }
//---------------------------------
struct tnod          //базовый узел
 {
  char *word;        //значение переменной — символьная
                     //строка в узле (слово)
  int count;        //здесь накапливается число встреч
                    //данного слова
  struct tnod left;  /*левый потомок: здесь хранится
  указатель на левый подузел, т. е. на структуру, в которой
  хранится слово меньшее данного. Если слов меньших данного
  нет, то здесь хранится ноль */
struct tnod right; /*правый потомок: здесь хранится указатель
на правый подузел, т. е. на структуру, в которой хранится
```

```
слово больше данного.  Если слов больших данного нет, то
здесь хранится ноль */
 };


char *strsave(char *s) //C++ позволяет объявлять структуру
                       //без слова struct


/*Функция выделяет по malloc() память по длине строки s, и в
эту память помещает саму строку s), а затем выдает указатель
на начало помещенной в буфер строки. */


 {
  char *p;
   p=(char *)malloc(sizeof(strlen(s)+1));


  if((p != NULL))
   strcpy(p,s);
   return(p);
   /*т. к. malloc() выдает указатель типа void, то
   принудительно приводим его к типу char, чтобы согласовать с p*/
 }
//---------------------------------
tnod *NodAlloc(void)
/*Функция выделяет память под структуру tnod и возвращает
указатель на эту память*/
 {
  tnod *p=p=(tnod *)malloc(sizeof(tnod));
  return(p);
 }
//---------------------------------


tnod *MakeTree(tnod *p,char *w)
/*Эта функция формирует двоичное дерево.
p - указатель на структуру, по которой будет создаваться
новый узел или будут проверяться старые узлы.
w - слово, поступающее для его поиска в дереве.
```

```
Если такое слово уже есть, то в счетчик структуры, где оно
обнаружено, добавляется единица: подсчитывается, сколько
одинаковых слов.

Если такого слова в дереве нет, то для него образуется новый
узел (новая структура)
*/
  {
   int cond;
   if(p==NULL) /*появилось новое слово. Для этого слова еще
   нет узла и его надо создать*/
     {
      p=NodAlloc();   //выделяется память под новый узел
      p->word=strsave(w);
/*введенное слово по strsave() записывается в память и на
него выдается указатель */
      p->count = 1; // слово пока единственное
      p->left=p->right= NULL;
/*т. к. это слово пока последнее в дереве, то этот узел не
указывает на следующие (меньшие — слева или большие — справа)
слова дерева*/
     }
    else if((cond=strcmp(w,p->word))==0)
/* слово не новое, тогда оно сравнивается со словом в узле,
на который указывает указатель p*/
       p->count++;
/*в этом узле - такое же слово, поэтому добавляем к счетчику
единицу*/
    else if(cond < 0)
/*слово меньше того, которое в узле, поэтому его помещаем в
левую часть дерева опять же с помощью этой функции. Ее первая
часть создаст узел в динамической памяти, поместит в него
слово, в левые и правые подузлы поместит нули, т. к. пока
дальше этого слова ни справа, ни слева ничего нет, а в левую
часть узла-предка поместит указатель на эту структуру */
       p->left=MakeTree(p->left,w);
          else if(cond > 0)
            p->right=MakeTree(p->right,w);
/*слово больше того, которое в узле, поэтому мы его помещаем
в правую часть дерева опять же с помощью этой же функции: ее
```

```
первая часть создаст узел в динамической памяти, поместит в
него слово, в левые и правые подузлы поместит нули, т. к.
пока дальше этого слова ни справа, ни слева ничего нет, а в
правую часть узла-предка поместит указатель на эту структуру */
return(p);
/*возвращает указатель на область, где создана новая
структура, или на структуру, в которую добавлена единица,
т. к. поступившее слово совпало со словом в этой структуре*/
  }
//-----------------------------
TreePrint(tnod *p)
/*Эта функция печатает или выводит на экран все дерево,
рекурсивно себя вызывая*/
 {
  if(p != NULL)
   {
     TreePrint(p->left); //выводит левый узел:
     printf("p->count=%d p->word=%s\n",p->count,p->word);
     TreePrint(p->right);  //выводит правый узел
   }

 }
//------------------------------------------------------------


#pragma argsused
int main(int argc, char* argv[])
{
 tnod *pp;
 char s[maxline];
 pp=NULL;
 int i=0;
 printf("Enter your words: >\n");
 while(getline(s,maxline)-1)
/*getline() возвращает количество введенных символов
Когда нажимаем только <Enter>, вводится всего один символ
('\n').
```

```
Если от единицы отнять единицу, получим ноль, а это для
оператора while сигнал о прекращении цикла*/
   {
    pp=MakeTree(pp,s); // формирует очередной узел
                       // pp всегда указывает на начало дерева
    } //while
  TreePrint(pp);  //вывод дерева
  getch();
}
//-------------------------------------------------------------
```

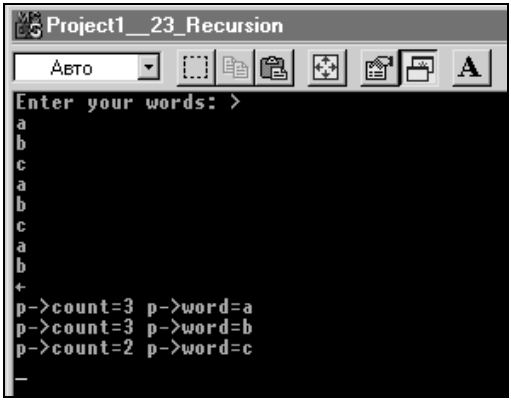


Рис. 7.6. Результат работы программы листинга 7.9

Физическая структура дерева, построенного программой, приведена в табл. 7.1, а логическая — на рис. 7.7.

Таблица 7.1. *Физическая структура дерева, построенного программой листинга 7.9*

`w1`: слово в динамической памяти	**Структура**			
	Указатель на слово `w1`	Количество слов `w1`: 1	Указатель на структуру, где находится `w2`	Указатель на структуру, где находится `w3`

Таблица 7.1 *(окончание)*

	Структура			
`w2`: слово в динамической памяти (`w 2 < w1`)	Указатель на слово `w2`	Количество слов `w2: 1`	0	0
`w3`: слово в динамической памяти (`w 3 > w1`)	Указатель на слово `w3`	Количество слов `w3: 1`	0	0

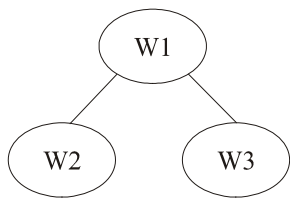


Рис. 7.7. Логическая структура дерева, построенного программой листинга 7.9

Битовые поля в структурах

Структуры обладают замечательным свойством: с их помощью можно задавать битовые поля: определять в переменной типа `int` группы подряд расположенных битов, значения которых можно задавать и с которыми можно работать как с элементами структуры. Долой всякие изощренности по засылке и "доставанию" битов из чисел: теперь все значительно проще! Описание битовых полей через структуру задается следующим образом:

```
struct
 {
   unsigned name1: size1;   //имя поля и его размер
   unsigned name2: size2;   //имя поля и его размер
```

```
   ...
   unsigned nameK: sizeK;   //имя поля и его размер
 }flags;
```

Это обычное задание шаблона и на нем экземпляра структуры.

Размер — это количество битов, расположенных подряд. Сумма всех полей не должна выходить за пределы размера переменной типа `int`. Если же это случится, то первое битовое поле, превысившее размер переменной `int`, расположится в следующем участке памяти, отведенном для переменной типа `int`. Пример:

```
struct
 {
   unsigned n1: 1;   //имя поля и его размер
   unsigned n2: 2;   //имя поля и его размер
}flags;
```

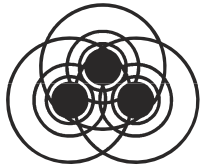
Здесь определен экземпляр структуры — переменная `flags`, содержащая два битовых поля: одно — однобитовое, другое — двухбитовое. На поля можно ссылаться как на отдельные члены структуры: `flags.n1`, `flags.n2`. Эти поля ведут себя подобно целым без знака и могут участвовать в арифметических операциях, применяемых к целым без знака. Например, для определенных выше полей можно записать:

```
flags.n1=1;   //включен бит
flags.n1=0;   //выключен бит
flags.n2=3;   //включены биты 1,2 поля (3 в десятичной
              //системе равно 11 в двоичной).
```

Для проверки значения битов можно писать:

```
if(flags.n2==1 && flags.n1==0) …
```

Глава 8



Классы в C++

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это естественное расширение структурного программирования. ООП требует сложных программных действий, но делает создание программы достаточно легким. В результате создаются совершенные коды, которые легко распространять и поддерживать. Однажды созданный для приложения объект вы можете использовать в других приложениях. Повторное использование объектов намного сокращает время разработки и увеличивает производительность труда. ООП основано на использовании *классов*. Классы — это основное отличие языка C++ от языка C.

Классы

Существуют разработчики компонентов и пользователи компонентов (разработчики приложений): если разработчик создает классы, то пользователь манипулирует классами и экземплярами классов.

Класс — это обыкновенный тип. Если вы программист, то вы всегда имеете дело с типами и экземплярами, даже если и не используете эту терминологию. Например, вы создаете переменные типа `int`. Классы обычно более сложны, чем простые типы данных, но они работают тем же способом: назначая различные значения экземплярам того же типа (как и переменным типа `int`), вы можете выполнять разные задачи. Поэтому наша зада-

ча — научиться пользоваться классами для написания приложений, а создание классов оставить их разработчикам.

Класс — это собрание связанной информации, которая включает в себя и данные, и функции (программы для работы с данными). Эти функции в классах называются *методами*. Класс — это дальнейшее развитие структур: в них тоже объединяются данные разных типов. Это такой же шаблон, под который, как и под структуру, память выделяется только тогда, когда мы создаем "переменную типа этого шаблона". Вспомним, что если у нас была некая структура `A`, то чтобы работать с ней, мы создавали экземпляр этой структуры `a` путем объявления `A a;` и затем уже работали с экземпляром `a`. То есть можно было сказать, что мы объявляли переменную `a` типа `A`. Точно так же поступают и для класса: если есть класс `A` (шаблон, под него память не выделяется), то объявляют переменную `a` типа `A` путем объявления `A a;` и могут работать уже как бы с самим классом, а на самом деле — с его экземпляром `a`. Как и при использовании структур, к членам класса (данным и методам) можно обращаться по тем же правилам: если объявлено `A a;`, то обращение к члену класса с именем `aa` будет записываться как `a.aa`, а если был объявлен указатель на класс, например, как `A *a;`, то обращение к члену класса с именем `aa` будет записываться как `a->aa`.

Класс — это некая конструкция, параметрически определяющая некоторую категорию объектов. Например, может быть класс компьютеров, который объединяет в себе компьютеры разных марок, разных возможностей. Может быть класс столов: столы письменные, обеденные и т. п. Класс столов может делиться на подклассы: столы письменные, которые, в свою очередь, могут делиться на столы письменные дубовые и столы письменные древесноволокнистые и т. д. То есть мы видим, что классы могут принадлежать некой иерархии классов. В каждом классе определены характеристики тех объектов, которые образуют этот класс. В классе также задаются программы, называемые *методами*, которые обрабатывают характеристики объектов, принадлежащих данному классу. Поведение объекта в реальном мире определяется его характеристиками. Изменяя значение характеристик, мы получим разное поведение объектов. Когда мы создаем эк-

земпляр класса и определяем значения его конкретных характеристик, мы получаем конкретный объект. В составе класса существует специальный метод (т. е. программа-функция), который формирует экземпляр класса. Этот метод носит название *конструктора*. В противоположность конструктору, существует *программа-деструктор*, которая уничтожает экземпляр класса в памяти.

Принципы построения классов

Основные принципы построения классов это: инкапсуляция, наследование и полиморфизм.

Инкапсуляция

Инкапсуляция — это принцип объединения в едином объекте данных и программ, обрабатывающих эти данные. В терминологии ООП данные называются членами-данными, а программы, их обрабатывающие (эти программы построены в виде функций), — членами-функциями, или методами.

Такой подход позволяет максимально изолировать объект от внешнего воздействия, что приводит к высокой надежности программ, использующих объекты. С другой стороны, классы используются, как ранее использовались стандартные программы, только с намного большей эффективностью в самых разных приложениях, что значительно повышает производительность труда программиста. При добавлении новых характеристик объектам программы, ранее использовавшие объекты, остаются без изменений.

В C++ Builder введено понятие *компонентов* — специальных классов, в которых объекты определяются такими характеристиками, как свойства, события и методы. Имена компонентов всегда начинаются с символа `T`, и их общим предком является класс `Tcomponent`. Причем в отличие от работы с обычными классами, при работе в C++ Builder возможно манипулировать видом и функциональным поведением компонентов и на стадии проектирования вашего приложения, и в момент его выполнения. Например, в C++ Builder существует компонент "форма"

(класс `TForm`) и "кнопка" (класс `TButton`), у которых есть свои свойства, методы и события. Если при проектировании приложения в форму поместить две кнопки, одну пометить **OK** и придать ее свойству `Caption` значение `true`, а ее свойству `Visible` — значение `false`; другую кнопку пометить **Cancel** и придать ее свойству `Caption` значение `false`, а ее свойству `Visible` — значение `true`, то каждая кнопка станет экземпляром класса `Tbutton`. С помощью назначения разных значений свойствам кнопок `Caption` и `Visible` вы получаете два экземпляра, которые ведут себя по-разному: первая кнопка при выполнении программы будет невидима в форме, а вторая останется видимой. При помощи события компонент сообщает пользователю, что на него произведено определенное воздействие (например, для компонента "кнопка" событием может быть нажатие кнопки — щелчок мышью), а методы служат для обработки реакции компонента на события.

Наследование

Наследование — второй принцип построения классов. Мы видели, что классы, в общем случае, могут составлять иерархию: столы — столы дубовые — столы дубовые письменные и т. д. Можно иерархию классов представить и как семью, где есть родители и дети, бабушки с дедушками, их внуки и т. д. Наследование предполагает, что все характеристики класса-родителя присваиваются классу-потомку. После этого потомку добавляют новые характеристики. Иногда некоторые методы в классе-потомке *переопределяются*, т. е. наполняются новым содержанием.

Наследование используется не только при разработке классов, но и при проектировании приложения. Например, в C++ Builder есть класс `TLabel` (метка). Если поместить экземпляр этой метки в форму (экземпляр `TForm`), то свойство "цвет" метки примет значение свойства "цвет" из экземпляра `TForm`, если не изменить свойство метки `ParentColor` (цвет, заданный у родителя). То есть метка автоматически будет наследовать это свойство от экземпляра `TForm`, на который она помещена.

Мы не ставим своей целью разработку классов, как было отмечено в начале этой главы. Но знать основные данные классов необходимо, т. к. иначе невозможно ими пользоваться при построении приложений в среде C++ Builder.

Рассмотрим структуру базового класса, из которого могут создаваться классы-потомки. Объявление класса представлено в листинге 8.1.

Листинг 8.1

```
class <имя>
 {
 private:  /*Имя секции. Данные и методы, помещенные в эту
секцию, будут доступны только методам этого класса. Доступ к
ним методов производных классов запрещен*/
 <Приватные данные>
 <Приватные конструкторы>
 <Приватные методы>


 protected:   /*Имя секции. Данные и методы, помещенные в эту
секцию, будут доступны методам этого класса и производным от
него*/
 <защищенные данные
 <защищенные конструкторы>
 <защищенные методы>


public:   /* Имя секции. Данные и методы, помещенные в эту
секцию, будут доступны методам всех классов.*/
 <общедоступные данные>
 <общедоступные конструкторы>
 <общедоступный деструктор>
 <общедоступные методы>
__published:   /*Имя секции. */
};   /*обратите внимание, что так же заканчивается
       и объявление структуры*/
```

Секция `__published` отличается от секции `public` тем, что свойства-данные из секции `__published` будут выведены в окно так называемого Инспектора объекта в том случае, если компонент находится в палитре компонентов. Кроме того, эта секция используется только для компонентов. В среде C++ Builder во время работы приложения (этот режим называют Run-Time, в отличие от режима разработки — Design-Time) текущая информация об объекте (RTTI — Object Pascal-style runtime type information — информация о типах в процессе исполнения) передается в Инспектор объекта для членов класса и методов, объявленных в данной секции.

Примечание

Инспектор объекта открывается для каждого компонента, с которым мы начинаем работать. В нем можно увидеть не только некоторые свойства объекта, но и его события. На *палитре компонентов* расположены компоненты среды C++ Builder (могут быть и не все).

Глядя на то, как объявляется класс, вспомним пример из *разд. "Структуры" главы 7*, где речь шла о том, что для структур в языке C++ можно задавать секции `private`, `public`, `protected`. Эти же секции можно задавать и для классов. Как и для структур, в этих секциях можно определять функции (в классах — это методы), и вызывать секции на выполнение можно только в соответствии с тем, в какой секции находится функция. В классах методы вызываются так же, как если бы они находились в структуре:

```
имя (экземпляра).f(фактические параметры функции);
```

А все эти сходства оттого, *что в C++ структуры рассматриваются тоже как классы.*

Компоненты перед использованием в среде C++ Builder должны быть помещены в форму (класс `TForm`), которая может передавать им свои свойства. Для формы создается два модуля. Один из них с расширением h, другой — с расширением cpp. В модуле с расширением h объявляется класс `Tform`, и в его секции `__published` хранятся указатели на все расположенные в форме компоненты и объявления методов-обработчиков событий компонентов формы. Там же существуют секции `public` и `private`,

в которых можно задавать пользовательские данные. В модуле с расширением cpp находится конструктор формы и модули обработки событий компонентов, расположенных в форме.

Метод для данного класса может определяться вне самого класса. Чтобы компилятор установил принадлежность такого метода к своему классу, перед именем метода указывают имя класса и двойное двоеточие (`::`).

Полиморфизм

Полиморфизм — это третий принцип, лежащий в основе создания класса. При полиморфизме (дословно: многоформие) родственные объекты (т. е. происходящие от общего родителя) могут вести себя по-разному в зависимости от ситуации, возникающей в момент выполнения программы. Чтобы добиться полиморфизма, надо один и тот же метод в классе-родителе переопределять в классе-потомке. Например, родственные классы, определяющие геометрические фигуры (точку, линию, прямоугольник, окружность и т. д.), имеют общий метод `draw`, который рисует соответствующую фигуру. Но все фигуры разные, т. к. метод `draw` в каждом из классов-потомков переопределен, т. е. в каждом классе-потомке ему назначена другая функциональность. Полиморфизм достигается за счет того, что функциям из класса-родителя позволено выполняться в классе-потомке. Такие функции должны объявляться в обоих классах с атрибутом `virtual`, записываемым перед атрибутом "возвращаемый тип данных". Если функция имеет атрибут `virtual`, она может быть переопределена в классе-потомке, даже если количество и тип ее аргументов те же, что и у функции базового класса. Переопределенная функция отменяет функцию базового класса.

Кроме атрибута `virtual`, у методов существует атрибут `friend`. Функции с таким атрибутом, расположенным, как и атрибут `virtual`, в объявлении функции-метода перед указанием типа возвращаемых данных, называются *дружественными*. Функция, объявленная с атрибутом `friend`, имеет полный доступ к членам класса, расположенным в секциях `private` и `protected`, даже если эта функция — не член этого класса. Это справедливо и

для классов: внешний класс имеет полный доступ к классу, который объявляет этот внешний класс дружественным.

Во всех остальных аспектах дружественная функция — это обычная функция. Подобные функции из внешних классов, имея доступ к секциям `private` и `protected`, могут решать задачи, реализация которых с помощью функций-членов данного класса затруднительна или даже невозможна.

Пример создания классов

Приведем пример двух программ, в которых объявлены и использованы простейшие классы.

Создание и использование класса "не компонента"

Программа показана в листинге 8.2.

Листинг 8.2

```
pragma hdrstop
#include <stdio.h>
#include <conio.h>

//---------------------------------------------------------------

#pragma argsused

class A
 {
  protected:
   int x;   /*к этим данным имеют доступ только методы
   данного класса и производных*/
   int y;
  public:
    int a;
```

```
    int b;
    int f1(int x, int y)
      {
       return(x-y);
      }
 };

 class B : A   // это объявляется класс, производный от A
  {
   public:
   int f2(int x)
    {
     A::x=20;     /* здесь могут использоваться члены-данные
     базового класса из секции protected */
     return(x+A::x);
    }
  };

int main(int argc, char* argv[])
{
  A min;  //создание экземпляров классов A, B
  B max;

  min.a=10;  //Работа с элементами класса A из секции public
  min.b=20;
 int x1=min.f1(min.a,min.b);
 int x2=max.f2(10);  // Работа с элементом класса B
 printf("x1=%d\nx2= %d\n",x1,x2);
     getch();
}
//---------------------------------------------------------
```

Результат работы программы показан на рис. 8.1.

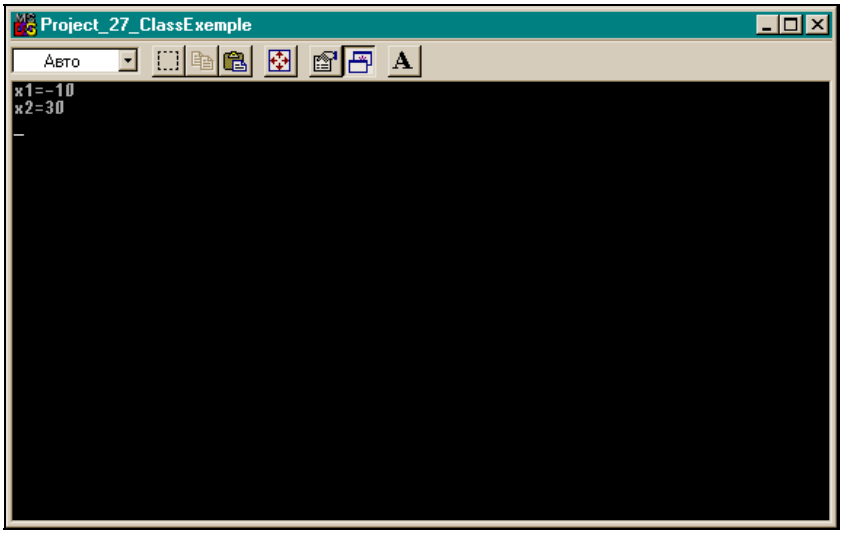


Рис. 8.1. Результат работы программы листинга 8.2

Создание и использование класса "компонента"

Создадим класс `TMyComponent` из базового класса `TComponent`. В создаваемом классе будет одно свойство типа "массив" и один метод, работающий с этим свойством. Правила создания классов-компонентов ввиду их специфики (их элементами являются свойства, события и методы) отличаются от правил создания обычных классов. Не вдаваясь в детали этих правил, приведем пример программы с классом-компонентом (листинг 8.3).

Листинг 8.3

```
//---------------------------------------------------------

#include <vcl.h> //for new class definition

#include <conio.h>
#include <stdio.h>
```

```
#include <stdlib.h> //atoi()
#include <alloc.h> //malloc(),free()



#pragma hdrstop

#define eof  27  //Esc
#define maxline 1000




//-------Ввод строки с клавиатуры

getline(char s[],int lim)
  {
    int c,i;
    for(i=0; i<lim-1 && (c=getchar()) != eof && c != '\n';
i++)
      s[i]=c;
      s[i]='\0';
      i++;     //для учета количества
    return(i);
  }

//---------------------------------------------------------------
class TMyComponent:public TComponent
 {
  private:
   String __fastcall GetSize(int ind);  /*может обрабатывать
   только член класса: это обычное объявление функции,
   которая будет использована здесь*/
  public:


    TMyComponent(TMyComponent *Owner); /*Конструктор создает
    класс в памяти (его вызывает компилятор). Память для
    экземпляра класса надо задавать самому. Указатель Owner
```

```
    задает собственника данного класса. При удалении
    собственника класса из памяти, все его потомки тоже
    удаляются из памяти*/


  __published:
    __property String Array[int ind]={read=GetSize};
/*свойство "Массив типа String". Значение этого свойства
будет определяться функцией GetSize(): т. е. свойство можно
только читать (read=GetSize), и в него нельзя писать.
Компилятор сам построит выполнение функции GetSize, которая
определяет характеристику размерности массива в зависимости
от ее величины*/
 };     //конец объявления класса


//Определение функции (метода)
  String __fastcall TMyComponent::GetSize(int ind)
     /*функция определена вне класса, и поэтому через ::
     показано, к какому классу она принадлежит. Функция
     выдает данные типа String */
   {
    String res;
    if(ind==0)
     res="size=0";
    else if(ind <=100)
     res="size<=100";
     else
       res="size > 100";
     return(res);
   }


int main()
{
TMyComponent *b=(TMyComponent*)malloc(sizeof(TMyComponent));
  /*в консольном (не оконном) приложении операторы new
  и delete не работают: их надо заменять на пару malloc()-
  free(): malloc() выдает указатель типа void, поэтому мы
  сами должны принудительно приводить указатель к нужному
  типу с помощью конструкции: (требуемый тип)malloc()*/
   char ss[maxline];
```

```
  int c;
  do
   {
    printf("enter your new size of Array >");
    getline(ss, maxline);

    int i=atoi(ss);      /* ввод размерности массива Array[],
    определенного как свойство через ввод строки символов
    и ее преобразование в число */

 String s=b->Array[i];
 /*когда мы здесь задаем обращение к массиву Array и его
 размерность, начинает выполняться метод GetSize(),
 который вычисляет характеристику(свойство) типа "массив" */

 char* cp = s.c_str();
 printf("cp=%s\n",cp);
 getch();
 }
  while((c=getchar()) != eof) ; //конец оператора do-while
  free(b); //освобождение памяти
}
//-------------------------------------------------------------
```

Основные пояснения даны в тексте программы. Остается пояснить следующие моменты:

- мы использовали тип `String`. Это класс, в котором определены данные и методы работы со строковыми величинами. Результат работы функции `GetSize()` имеет тип `String`. Этот результат мы хотим выдать на экран с помощью функции `printf()`, которая не может работать с данными типа `String`. Поэтому мы должны преобразовать строку этого типа в тип `char`, "понятный" функции `printf()`. Для этого мы воспользовались методом `c_str()` класса `String`, который, воздействуя на строку типа `String` (строка `s` принадлежит этому же классу), выдает указатель на начало строки типа `char` с таким же содержимым, что и в строке `s`;

- при создании консольного приложения для работы с компонентными классами мы должны включить переключатель **Use VCL**, чтобы подключилась библиотека, в которой находится описание компонентного класса `TComponent`.

Результат работы программы представлен на рис. 8.2.

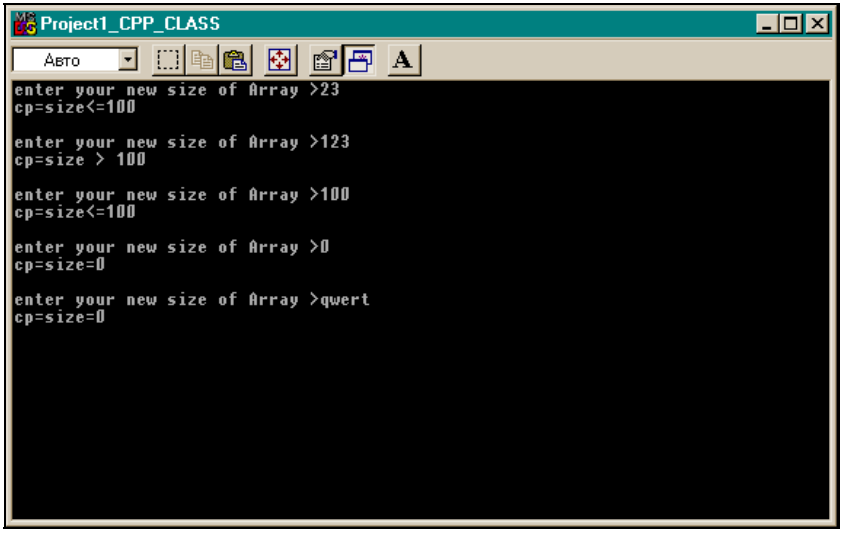
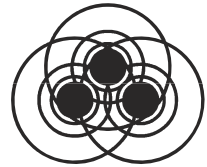


Рис. 8.2. Результат работы программы из листинга 8.3

Глава 9



Ввод и вывод в C и C++

Ввод и вывод в C

Ввод-вывод в языке C осуществляется функциями из стандартных библиотек. Чтобы ими пользоваться, в программу надо включать соответствующие h-файлы: stdio.h, stdlib.h, conio.h и др. Основная библиотека — stdio.h. В ней содержатся основные функции ввода-вывода, в том числе и обеспечивающие стандартный ввод-вывод.

Ввод-вывод файлов

Чтобы работать с файлом, его сначала следует открыть: связать со специальной структурой с именем `FILE`, которая описана в библиотеке stdio.h и в которой задаются характеристики файла (размер буфера ввода-вывода, состояние файла, последняя прочитанная запись и т. п.). Связь эта выполняется с помощью функции `fopen()`, которая тоже входит в библиотеку stdio.h и возвращает указатель на структуру `FILE`. Поэтому в программе прежде всего следует задать указатель на структуру `FILE` (например, `FILE *fp;`), а затем записать оператор собственно открытия файла:

```
fp=fopen(имя файла, способ открытия файла);
```

Функция открытия имеет два параметра: имя открываемого файла и способ открытия файла. Способ открытия файла определяет, как будет пользователь работать с файлом: читать его, писать в него или делать что-то еще. Рассмотрим способы открытия файла.

Их коды и значения приведены ниже:

- `r` — файл открывается только для чтения из него;
- `w` — файл открывается только для записи в него (если файл не существует, он создается);
- `a` — файл открывается для дозаписи информации в конец файла. Если файл не существует, он создается для записи в него;
- `r+` — существующий файл открывается для обновления: чтения и записи;
- `w+` — создается новый файл для работы в режиме обновления: такой файл можно будет и читать и записывать в него;
- `a+` — файл открывается для дозаписи информации в конец файла. Если файл не существует, он создается. Если существует, то открывается для дозаписи в конец файла.

Если по какой-либо причине открытия файла не произошло (например, задано имя несуществующего файла), то функция `fopen()` возвращает значение `NULL`. Поэтому открытие файла следует осуществлять так:

```
if((fp=fopen(name,mode)) == NULL)
     {операторы обработки ошибки открытия}
остальные операторы программы
```

После того как программа с данным файлом отработала, следует "отвязать" структуру `FILE` от отработавшего файла или, как говорят, закрыть файл. Это осуществляет функция `fclose(fp)`. Она не только разрывает связь структуры с файлом, но и записывает в память оставшееся содержимое буфера ввода-вывода, через который собственно и организуется ввод-вывод. Только после закрытия файла с ним можно выполнять какие-либо действия, т. к. он "свободен", "не привязан". Например, его можно удалить или заново открыть в другом режиме открытия и т. д.

Функции для работы с файлами

После того как файл открыт, то для чтения или записи используют специальные функции. В табл. 9.1 приведен перечень

функций для работы с файлами. Как работают эти функции, можно посмотреть в разделе **Help**.

Таблица 9.1. *Функции для работы с файлами*

Функция	Формат функции
fputc()	fputc(c, fp);
fputs()	fputs(s,fp);
fgetc()	c=fgetc(fp);
fgets()	fgets(s,maxline,fp);
fread()	fread(buf, m, n, fp);
fwrite()	fwrite(const void ptr, m, n, fp);
fseek()	fseek(fp,n,m);
ftell()	long int ftell(fp);
fscanf()	fscanf(fp,Control,arg1,arg2,…,argn);
fprintf()	fprinf(fp,Control,arg1,arg2,…,argn);
rewind()	rewind(fp);
remove()	remove(FILENameString);
FILElength()	long FILElength(fp);
ferror()	ferror(fp);
freopen()	FILE *freopen(const char *FILEname, const char *mode, FILE *stream);
feof()	feof(fp);
ferror()	ferror(fp);
exit()	exit(int status);

Стандартный ввод-вывод

При запуске любой программы автоматически открываются сразу 3 файла:

- файл стандартного ввода. Его указатель называется `stdin`;
- файл стандартного вывода. Его указатель называется `stdout`;

- файл стандартного вывода ошибок. Его указатель называется `stderr`.

При работе с файлами мы можем использовать эти указатели, чтобы направлять данные в стандартные потоки, в которых по умолчанию ввод идет с клавиатуры, а вывод — на экран. Например, чтобы ввести строку с клавиатуры, можно применить функцию `fgets()` в виде: `fgets(s,maxline,stdin);`, а для вывода строки на экран — функцию `fputs()`в виде: `fputs(s,stdout);`

Из приведенного выше перечня функций, обслуживающих ввод-вывод, мы видели, что существуют функции `getc(fp)`, `putc(c,fp)`, которые соответственно вводят один символ из файла с указателем `fp` и пишут один символ в файл с указателем `fp`. Если вместо указателя `fp`, который имеет тип `FILE`, в эти функции поместить указатели стандартного потока, то они станут соответственно вводить один символ с клавиатуры и выводить его на экран. Оказывается, что ранее применяемые нами в примерах функции `getchar()`, `putchar()` связаны в файле stdio.h со стандартными потоками следующим образом:

```
#define getchar() getc(stdin)
#define putchar() putc(stdout)
```

Поэтому, подключив файл stdio.h к своей программе, мы спокойно работали с этими функциями, а фактически — с символическими константами.

В C стандартный ввод можно перенаправлять на ввод из файла (выше мы видели, что это может делать функция `freopen()`). Если некоторая программа с именем p1.exe использует функцию `getchar()`, то с помощью выполнения командной строки `p1.exe < anyFILE` мы получим ввод не с клавиатуры, а из файла anyFILE. Командную строку в C можно выполнить с помощью системной функции `system()` в виде: `system("P1.EXE < ANYFILE");`. Причем символы должны быть верхнего регистра, т. к. выполняется команда DOS. Точно так же, как и для ввода, можно перенаправить стандартный вывод в файл. Если имеем программу p2.exe, которая использует стандартный вывод, то с помощью выполнения командной строки `p2.exe > anyFILE1` мы получим вывод в файл anyFILE2.

Функции стандартного ввода-вывода

В табл. 9.2 приведен только перечень функций и их форматов. Суть функций можно посмотреть, воспользовавшись разделом **Help**.

Таблица 9.2. *Функции стандартного ввода-вывода*

Функция	Формат
`getchar()`	`getchar();`
`putchar()`	`putchar(char c);`
`printf()`	`printf(Control,arg1,arg2,…,argn);`
`scanf()`	`scanf(Control,arg1,arg2,…,argn);`
`sprintf()`	`sprintf(string,Control,arg1,arg2,…,argn);`
`sscanf()`	`sscanf(string,Control,arg1,arg2,…,argn);`
`cprintf()`	`cprintf(Control,arg1,arg2,…,argn);` (совпадает с форматом функции `printf()`)
`gets()`	`gets(s);`
`puts()`	`puts(s);`
`cputs()`	`cputs(s);`
`gotoxy()`	`gotoxy(x,y);`
`clrscr()`	`clrscr();`
`window()`	`window(x1,y1,x2,y2);`

Ввод-вывод в C++

Общие положения

Ввод и вывод в C++ организован с помощью так называемых *поточных классов*, содержащих данные и методы работы с файлами по вводу-выводу. Поточные классы происходят от общего предка — класса `ios` и потому наследуют его функциональность. Чтобы начать писать программу с использованием ввода-вывода на языке C++, следует обязательно выполнить в программе `#include<fstream>`. Класс `fstream` является потомком классов

istream, ostream. Эти же классы являются родителями классов ifstream, ofstream. Класс fstream используется для организации ввода-вывода (т. е. чтения-записи) в один и тот же файл. Классы ifstream, ofstream — для организации соответственно ввода (т. е. чтения) файла и вывода (т. е. записи в файл). В свою очередь, экземплярами классов istream, ostream являются cin, cout, cerr, с помощью которых осуществляется так называемый "стандартный ввод-вывод" — ввод со стандартного вводного устройства, которым по умолчанию является клавиатура, и вывод на стандартное выводное устройство, по умолчанию — экран. Таким образом, включения в программу класса fstream оказывается достаточным для организации как стандартного, так и файлового ввода-вывода.

Файловый ввод-вывод организован с помощью переопределенных в поточных классах операций включения (<<) и извлечения (>>). Ранее мы видели, что это операции сдвига влево и сдвига вправо битов в переменной типа int, но в поточных классах C++ они обрели новую функциональность.

Чтобы работать с файлом, его сначала следует открыть: связать со специальной структурой, в которой задаются характеристики файла (размер буфера ввода-вывода, состояние файла, последняя прочитанная запись и т. п.). Связь эта выполняется с помощью функции open(), входящей в один из классов, который определяет ввод-вывод (fstream, istream, ostream). Поэтому, чтобы выполнить такую функцию, следует сначала создать экземпляр соответствующего класса, чтобы получить доступ к этой функции. Если мы, например, хотим выполнять вывод в файл (т. е. запись в него), то следует создать экземпляр класса ostream: ostream exp; и затем выполнить функцию exp.open(). В скобках должны быть указаны параметры этой функции: имя открываемого файла и способ открытия файла, в котором задаются сведения о том, как собирается пользователь работать с файлом: читать его, писать в него, или делать что-то еще.

После того как файл открыт собственно для чтения или записи, уже используют операции включения-извлечения (<<, >>). Если использовать пример с экземпляром exp класса ostream, то можно записать, например:

```
exp << "строка текста" << i << j << endl;
```

Здесь `i`, `j` — некоторые переменные (например, `int i; float j;`), `endl` — конец вывода и переход на новую строку.

После того как работа с файлом закончена, следует закрыть файл, чтобы разорвать связь с той структурой, с которой файл был связан при его открытии. Это необходимо, чтобы дать возможность другим файлам "открываться". Этот акт выполняется с помощью метода `close()` того же экземпляра класса, который мы создавали, чтобы выполнить функцию `open()`. В нашем случае следовало бы написать: `exp.close();`

Ввод-вывод с использованием разных классов

Итак, мы определили, что поточные классы это поставщики инструментов для работы с файлами. В поточных классах хранятся:

- структуры, обеспечивающие открытие-закрытие файлов;
- функции (методы) открытия-закрытия файлов;
- другие функции и данные, обеспечивающие, как мы увидим ниже, собственно ввод-вывод.

Пространства имен

Многие серьезные приложения состоят из нескольких программных файлов (с исходным текстом программ), которые создаются и обслуживаются отдельными группами программистов. И только после этого все файлы собираются в общий проект. Но как быть с тем фактом, что в таких файлах могут быть одинаково объявлены некоторые разные переменные? В C++ это неудобство разрешается с помощью так называемых *пространств имен*, которые вводятся в каждый текстовый программный файл проекта. Для ввода служит директива:

```
Namespace <имя пространства имен (идентификатор)> {в эти
скобки заключается весь программный текст}
```

Когда идет сборка общего проекта, то в итоговом тексте пишут директиву:

```
using namespace:: идентификатор пространства имен;
```

Это обеспечивает в итоговом проекте доступ к переменным файла с данным пространством имен. При использовании поточных классов языка C++ в основной программе требуется писать директиву:

```
using namespace::std;
```

В противном случае программа не пройдет компиляцию. В листинге 9.1 приводится пример использования директив пространства имен.

Листинг 9.1

```
#include <iostream>
#include <conio.h>


   namespace F
    {
      float x = 9;
    }

   namespace G
    {
      using namespace F; ./*здесь само пространство G
      использует пространство F и в нем же объявляется еще
      одно пространство: INNER_G */
         float y = 2.0;
         namespace INNER_G
          {
            float z = 10.01;
          }
     }  // G

   int main()
   {
      using namespace G;  /*эта директива позволяет
      пользоваться всем, объявленным в "G"*/
```

```
    using namespace G::INNER_G;  /* эта директива позволяет
    пользоваться всем объявленным только в "INNER_G" */
    float x = 19.1;     //локальное объявление
    переопределяет предыдущее
    std::cout << "x = " << x << std::endl;
    std::cout << "y = " << y << std::endl; // y берется из
    пространства F */
    std::cout << "z = " << z << std::endl; // z берется из
    пространства INNER_G
    getch();
    return 0;
  }

/*Результат:
   x = 19.1
   y = 2
   z = 10.01
   */

   getch();
}
```

`std::cout` — это стандартный вывод. Его мы рассмотрим ниже. Здесь показано, что объект `cout` принадлежит пространству имен `std`. Мы могли бы в основной программе записать: `using namespace::std;`. Тогда бы вместо `std::cout` можно было бы писать просто `cout`.

Итак, при составлении программы с использованием поточных файлов в начале основной программы следует записать директиву `using namespace std;`.

Работа с классом *fstream*

Члены этого класса позволяют открыть файл, записать в него данные, переместить указатель позиционирования в файле (указатель, показывающий, на каком месте в файле мы находимся) в то или иное место, прочитать данные.

Этот класс имеет такие основные функции (методы):

- `open()` — открывает файл.

  Параметры функции `open()`:

  - имя открываемого файла;
  - способ открытия файла;
- `close()` — закрывает файл;
- `is_open()` — если файл открыт, то она возвращает `true`, иначе — `false`;
- `rdbuf()` — выдает указатель на буфер ввода-вывода.

Способ открытия файла задается значением перечислимой переменной

```
enum open_mode {app,binary,in,out,trunc,ate};
```

Эта переменная определена в базовом классе `ios`, поэтому обращение к перечислимым значениям в классе `fstream`, с экземпляром которого мы работаем, должно идти с указанием класса-родителя: `ios::app, ios::binary` и т. д.

Назначение способов открытия файла:

- `app` — открыть файл для дозаписи в его конец;
- `binary` — открыть файл в бинарном виде (такие файлы были записаны по определенной структуре данных и поэтому должны читаться по этой же структуре);
- `in` — открыть файл для чтения из него;
- `out` — открыть файл для записи в него с его начала. Если файл не существует, он будет создан;
- `trunc` — уничтожить содержимое файла, если файл существует (очистить файл);
- `ate` — установить указатель позиционирования файла на его конец.

При задании режимов открытия файла можно применять оператор логического ИЛИ (`|`), чтобы составлять необходимое сочетание режимов открытия.

Приведем пример программы работы с классом `fstream` (листинг 9.2).

Листинг 9.2

```
#include<fstream>
#include <conio.h>
#include <stdio.h>
void main ( )
{
    using namespace std;     /* Используется стандартное
    пространство имен. Создание двунаправленного (чтение-
    запись в одном и том же файле) объекта (экземпляра) */
    fstream inout;
    inout.open("fstream.out",ios::out| ios::in);
    // вывод в файл
    inout << "This is the story1 of a man" << endl;
    inout << "This is the story2 of a man" << endl;
    inout << "This is the story3 of a man" << endl;
    char p[100];
    // установка указателя файла (позиционирование)
    // в его начало
    inout.seekg(0);

    // чтение 1-й строки (длиной не более 100 символов)
    inout.getline(p,100);
    // вывод 1-й строки на экран (stdout)
    cout << endl << "String1 :" << endl;
    cout << p;
    // запоминание текущей позиции в файле после 1-го вывода
      fstream::pos_type pos = inout.tellg();
    // чтение 2-й строки из файла
    inout.getline(p,100);
    //  вывод 2-й строки на экран (stdout)
    cout << endl << "String2 :" << endl;
    cout << p;
    // чтение 3-й строки из файла
    inout.getline(p,100);
```

```
    // вывод 3-й строки на экран (stdout)
    cout << endl << "String3 :" << endl;
    cout << p;
    // установка указателя перед 2-й строкой
    inout.seekp(pos);
    // запись на место 2-й строки
    inout << "This is the story2 of a man" << endl;
    // запись на место 3-й строки
    inout << "This is the story3 of a man" << endl;
    // установка на начало файла
    inout.seekg(0);
    // вывод всего содержимого потока на экран (stdout)
    cout << endl << endl << inout.rdbuf();
    inout.close();
    system("DEL FSTREAM.OUT");

    getch();
}
```

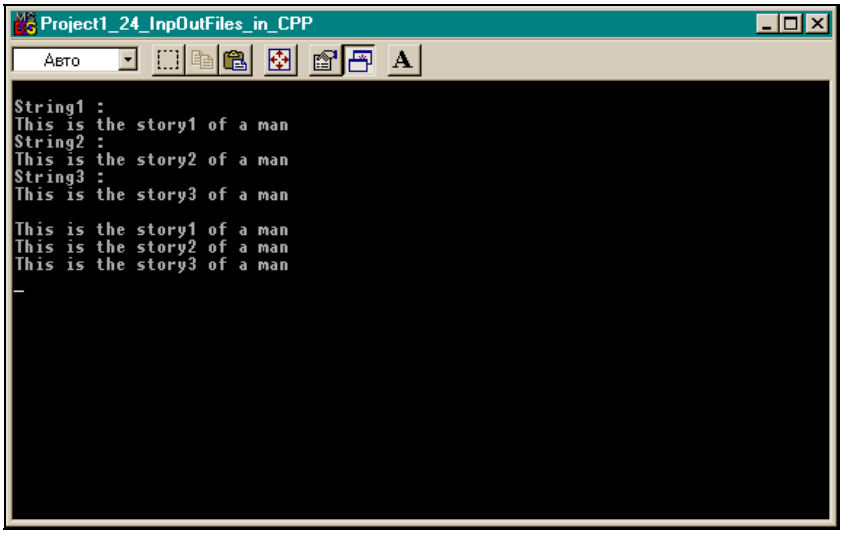


Рис. 9.1. Результаты работы программы листинга 9.2

Пояснения к программе даны по тексту. Результат работы показан на рис. 9.1.

Работа с классом *ofstream*

Этот класс предназначен для организации работ по выводу (записи) в файл с помощью методов этого класса:

- `open()` — открывает файл для записи в него информации;
- `is_open()` — возвращает `true`, если файл открыт, и `false` в противном случае;
- `put()` — записывает в файл один символ;
- `write()` — записывает в файл заданное число символов;
- `skeep()` — перемещает указатель позиционирования в заданное место файла;
- `tellp()` — выдает текущее значение указателя позиционирования;
- `close()` — закрывает файл;
- `rdbuf()` — выдает указатель на буфер вывода (этот буфер находится в структуре, с которой связывается файл при его открытии).

Ниже приведен пример использования класса `ofstream` (листинг 9.3).

Листинг 9.3

```
ofstream FILE; /*объявляем переменную FILE типа ofstream
(создаем экземпляр класса) */
FILE.open("a.txt");  //вызываем метод открытия файла
if(FILE ==NULL) return(0);  //неудачное открытие файла
for(int i=0; i<2; i++)
 FILE << "string " << i << endl;  //вывод в файл
FILE.close(); //закрытие файла
```

Работа с классом *ifstream*

Этот класс предназначен для организации работ по вводу (чтению) из файла с помощью методов этого класса:

- `open()` — открывает файл для чтения из него информации;
- `is_open()` — возвращает `true`, если файл открыт, и `false` в противном случае;
- `get()` — читает из файла один символ;
- `read()` — читает из файла заданное число символов;
- `eof()` — возвращает ненулевое значение, когда указатель позиционирования в файле достигает конца файла;
- `peek()` — выдает очередной символ потока, но не выбирает его (не сдвигает указатель позиционирования в файле);
- `seekg()` — перемещает указатель позиционирования в заданное место файла;
- `tellg()` — выдает текущее значение указателя позиционирования;
- `close()` — закрывает файл;
- `rdbuf()` — выдает указатель на буфер ввода (этот буфер находится в структуре, с которой связывается файл при его открытии).

Пример использования класса приведен в листинге 9.4.

Листинг 9.4

```
ifstream FILE; /*объявляем переменную FILE типа ifstream
(создаем экземпляр класса) */
char p[100];
FILE.open("a.txt");           //вызываем метод открытия файла
if(FILE ==NULL) return(0);  //неудачное открытие файла
while(!FILE.eof())          //проверка на признак конец файла
 {
  FILE >> p;               //чтение из файла
```

```
 cout << p  << endl;      // вывод прочитанных данных на экран
}
FILE.close();           //закрытие файла
```

Приведем пример программы с обоими классами `ofstream`, `ifstream` (листинг 9.5).

Листинг 9.5

```
#include<fstream>
#include <conio.h>

#define DelKey 's'   //этот символ будет удаляться из потока
#define maxline 1000

//---------------------------------------------------------

#pragma argsused

int main(int argc, char* argv[])
{
using namespace std;      // используется стандартное
                          // пространство имен

// Проверка вывода

ofstream FILE;
FILE.open("c:\\a.txt",ios::out);
char p[maxline];
int i,pos;
for(i=0; i<2; i++)
FILE << "string " << i;          /* << endl;  endl вводить не
надо, иначе и выводить его надо, и цикл будет длиннее */
FILE.close();
```

```
//Проверка ввода (чтения по записям)

ifstream FILE1;
FILE1.open("c:\\a.txt");
FILE1.seekg(0);    /* указатель - в начало(он и так будет в
начале, но это, чтобы посмотреть, как работает seekg()) */
if(FILE1 == NULL)   //так надо проверять на ошибку открытия
файла
   return(0);
while(!FILE1.eof())   //Так проверяется конец файла
 {
FILE1 >> p >> i;
cout << p << i << endl;
 }
FILE1.close();
getch();

  //Проверка посимвольного чтения

ifstream FILE2;
char c;
FILE2.open("c:\\a.txt");
if(FILE2 == NULL)   //так проверять на плохое открытие
   return(0);
while(!FILE2.eof())   //так проверяется конец файла
 {
c=FILE2.peek();  /*смотрит, какой следующий символ будет
считан, но указатель позиционирования при этом не сдвигается:
остается на этом символе */
streampos cgp=FILE2.tellg(); /* так определяется текущая
позиция в файле*/
   if(c==DelKey)     /*выбрасываются все символы DelKey
   из читаемого потока */
    {
     pos= cgp + 1; // готовимся пропустить символ по seekg()
     FILE2.seekg(pos); /*передвинули указатель позиционирования
     на один символ дальше, чтобы пропустить символ */
```

```
continue;    // на продолжение цикла
    }
FILE2.get(c);  //чтение символа в с
cout << c;
 } //while
cout << endl;
FILE2.close();
getch();
system("DEL C:\\A.TXT");  // удаление рабочего файла
     } //main()
```

Работа с бинарным файлом

Такие файлы, в отличие от потоковых, создаются в определенной логической структуре и поэтому должны читаться в переменную той же структуры. Пример программы приведен в листинге 9.6.

Листинг 9.6

```
#include<fstream>
#include <conio.h>
#include <stdio.h>
void main ( )
{
    using namespace std;      // используется стандартное
                              // пространство имен
    /*данные о сотрудниках*/
 struct Blocknotes
     {
      char name[30];
      char phone[15];
      int age;
     }b[2]={
            "Smit",  "123456",45,
```

```
            "Kolly", "456789",50
             };  //инициализация массива структур
//запись данных в файл
   ofstream FILE;
   FILE.open("Block",ios::binary);
   for(int i=0; i<2; i++)
    FILE.write((char *)&b[i],sizeof(b[i]));
    FILE.close();
//чтение данных из файла
    ifstream FILE1;
    FILE1.open("Block",ios::binary);
    Blocknotes bb[2];
    int i=0;
    while(!FILE1.eof())
     {
      if(i==2)
       goto m;
      FILE1.read((char *)&bb[i],sizeof(bb[i]));
      cout << "string" << i << "  " << bb[i].name << "  "
      << bb[i].phone <<"  " << bb[i].age << endl;
      i++;
     }
m:    FILE1.close();
     system("DEL BLOCK");
     getch();
}

//-------------------------------------------------------------
```

Пояснений требуют следующие моменты:

- `FILE.write((char *)&b[i],sizeof(b[i]));`

  Здесь для записи используется функция буферизированного вывода `write()`, где первым аргументом является указатель на структуру, из которой мы должны записывать данные. Этот указатель равен адресу структуры, т. е. `&b[i]`. Но в по-

токе все данные хранятся побайтно, поэтому тип указателя `char` (здесь идет принудительное преобразование типа). Второй аргумент — длина записи. Она определяется стандартной функцией `sizeof();`

- `system("DEL BLOCK");` Этой функцией удаляется рабочий файл;
- здесь применен оператор `goto` для подстраховки от превышения индекса массива `bb[]`.

Результат работы программы приведен на рис. 9.2.

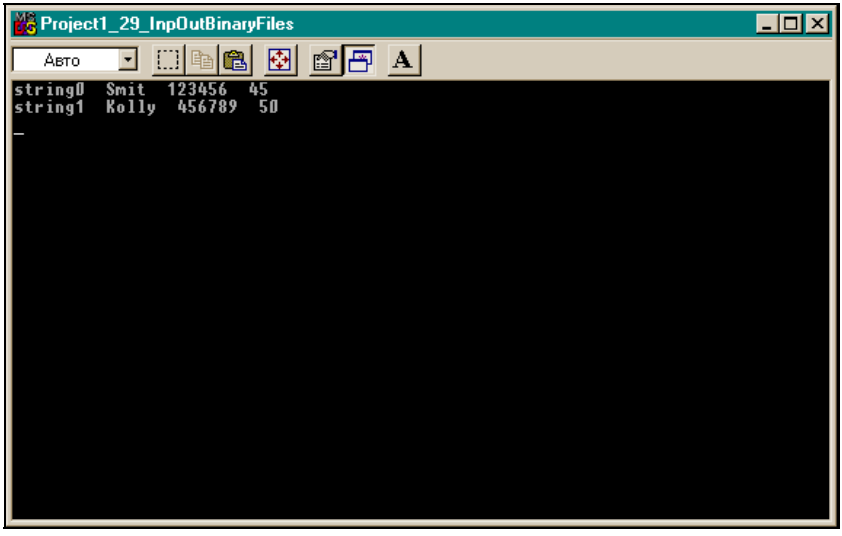


Рис. 9.2. Результат работы программы листинга 9.6

Стандартный ввод-вывод в C++

Общие положения

Стандартный ввод-вывод является частным случаем файлового ввода-вывода. При файловом вводе-выводе мы объявляли экземпляры соответствующих поточных классов и затем пользовались методами и операциями << и >>. Но как мы видели в нача-

ле этой главы, классы `istream`, `ostream`, лежащие в основе поточных классов, содержат стандартные объекты-экземпляры классов с именами `cout` (экземпляр класса для стандартного ввода), `cin` (экземпляр класса для стандартного вывода) и `cerr` (экземпляр класса для стандартного вывода сообщений об ошибках). При запуске любой программы на языке С++ эти стандартные потоки определены (открыты) и по умолчанию назначены на стандартное вводное устройство — клавиатуру (`cin`), на стандартное выводное устройство — экран (`cout` и `cerr`). Причем все эти устройства синхронно связаны с соответствующими указателями `stdin`, `stdout`, `stderr`. Так что работа со стандартным вводом-выводом сводится к тому, что вместо задаваемых пользователем имен экземпляров соответствующих классов задаются имена стандартных экземпляров классов: `cin`, `cout`. Открывать ничего не надо, надо только использовать операции `<<`, `>>` и операции форматирования. Если мы пишем имена переменных, из которых выводятся или в которые вводятся данные, то по умолчанию для ввода-вывода используются определенные форматы. Например, запишем:

```
cout << i;
```

В этом случае значение `i` выведется на экран в формате, определенном по умолчанию для типа `i` и в минимальном поле.

Запишем:

```
cin >> i >> j >> s;
```

где `i`, `j`, `s` описаны соответственно как `int`, `float`, `char`. В записи мы не видим форматов, но при вводе значений этих переменных с клавиатуры (после ввода каждого значения надо нажимать <Enter>) их форматы будут учтены.

Стандартный вывод *cout*

Объект `cout` направляет данные в буфер-поток, связанный с объектом `stdout`, объявленным в файле stdio.h. По умолчанию стандартные потоки С и С++ синхронизированы.

При выводе данные могут быть отформатированы с помощью функций-членов класса или манипуляторов. Перечень их можно посмотреть в разделе **Help**.

Стандартный ввод *cin*

Объект (экземпляр класса) `cin` управляет вводом из буфера ввода, связанного с объектом `stdin`, объявленным в файле stdio.h. По умолчанию стандартные потоки в языках C и C++ синхронизированы. При вводе используется часть тех функций и манипуляторов, которые определены для `cout`. Это такие манипуляторы, как `dec`, `hex`, `oct`, `ws` и др.

Пример программы с использованием объекта `cin` приведен в листинге 9.7.

Листинг 9.7

```
//
// cin example #1
//
#include <fstream>
#include <conio.h>
void main ( )
{
  using namespace std;

  int i;
  float f;
  char c;
  //ввод целого числа, числа с плавающей точкой и символа
  //c stdin
  cout << "Enter i,f,c and then input the string >" << endl;
  cin >> i >> f >> c;
  // вывод i, f и c на stdout
  cout << i << endl << f << endl << c << endl;

//
// cin example #2
//
```

```
  char p[50];
  // приказ на удаление из ввода всех пробелов
  cin >> ws >> p;
    cout << p << endl;
  //чтение символов с stdin, пока не будет нажата клавиша
  //<Enter> или не будут прочтены 49 символов
  cin.seekg(0);
  cin.getline(p,50);
  // вывод результата на stdout
  cout << p << endl;
  getch();
}
```

Результат работы программы приведен на рис. 9.3.

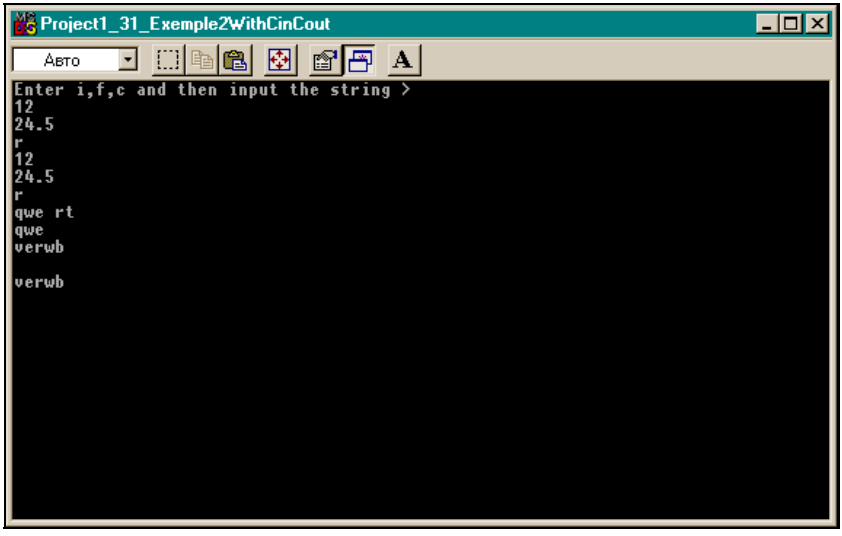
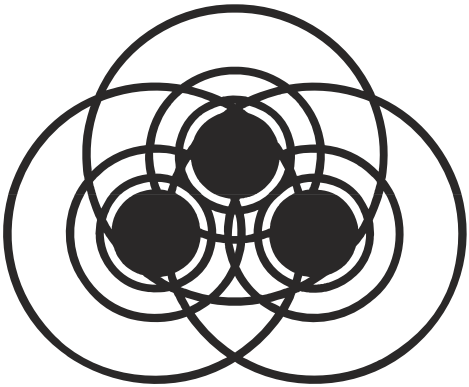


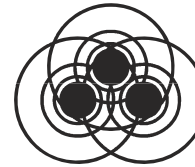
Рис. 9.3. Результат работы программы листинга 9.7



ЧАСТЬ II

Программирование в среде Borland C++ Builder

Глава 10



Начало изучения среды Borland C++ Builder

Как приступить к разработке нового приложения. Создание проекта

Все пользовательские программы в среде Borland C++ Builder называются *приложениями* (т. е. они прилагаются к самой среде) и оформляются в виде специальных структур, которые называются *проектами*. Ни одна программа не может существовать вне струтуры-проекта. Действия по управлению проектами осуществляет специальный программный комплекс — Менеджер проектов.

Чтобы приступить к разработке новой программы (т. е. приложения), надо сначала создать для нее новый пустой проект, пока не содержащий никаких элементов. Это делается с помощью подменю **File** главного меню среды. Надо выполнить команду **File/New Application**. Эта команда откроет на экране так называемую *форму*, экземпляр компонента `TForm`, и к ней создаст программный модуль — заготовку для помещения программ-обработчиков событий компонентов, которые будут размещены в форме. Дело в том, что форма — главное действующее лицо при создании проекта в среде Borland C++ Builder. Это главный контейнер, в котором размещаются компоненты самой среды. С помощью этих компонентов и реализуется какой-то конкретный алгоритм определенной задачи. Все построено именно так, что сначала надо открыть пустую форму — либо при первоначальном создании проекта, либо добавляя новую пустую форму

к уже существующим формам проекта, если этого требует алгоритм решения задачи. Но без открытия пустой формы не обойтись.

Когда форма появилась на экране, в нее в соответствии с имеющимся алгоритмом задачи помещают необходимые компоненты из палитры (т. е. из набора) компонентов среды, придают свойствам компонентов необходимые значения и определяют реакции на события компонентов. Реакции задаются в программах, которые называются обработчиками событий. Все программы-обработчики событий компонентов, расположенных в данной форме, помещаются в тот пустой программный модуль, который создается вместе с появлением формы на экране. Итак, мы с помощью команды **File/New Application** получили на экране пустую форму. Она показана на рис. 10.1.

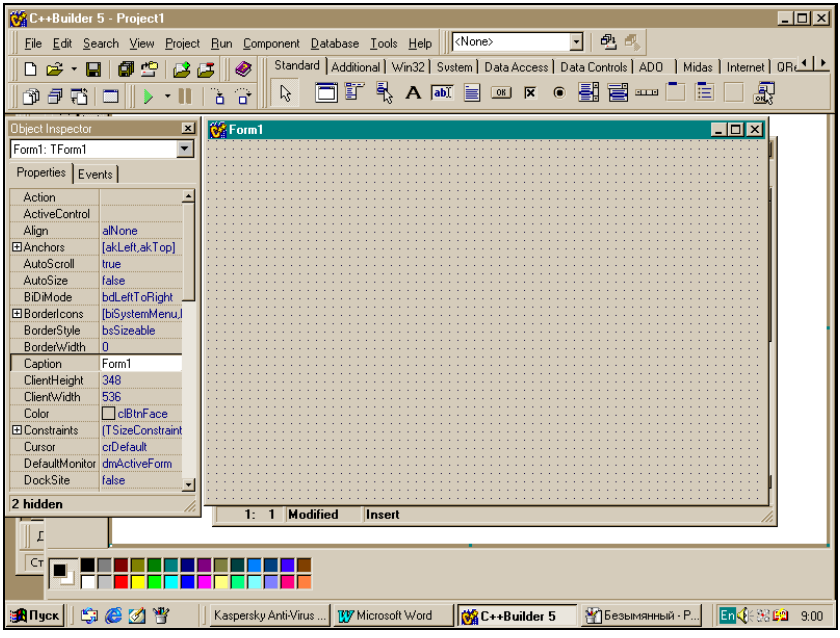


Рис. 10.1. Вид пустой формы приложения

После появления формы на экране в том же меню **File** следует выполнить команду **Save Project As**, позволяющую вам найти нужную папку и записать туда модуль, которому по умолчанию будет присвоено имя Unit1.cpp (вы вправе дать модулю свое имя). Затем появится окно для записи головного модуля проекта с именем, присвоенным ему по умолчанию: Project1.bpr. Головному модулю вы также можете дать свое имя. Пустой проект вы сотворили. Теперь вы можете начинать собственно разработку: помещать в форму (в ту, которая должна быть в вашем новом проекте — она осталась на экране после записей, что вы произвели) различные компоненты из палитры компонентов и делать с ними разные вещи, т. е. реализовывать алгоритм какой-то вашей задачи. Когда вы завершили проект, его надо сохранить (вообще по мере разработки его тоже периодически надо сохранять) командой **Save All** подменю **File** главного меню, после чего проект следует откомпилировать и выполнить (нажать клавишу <F9> или выполнить команду **Run/Run** главного меню).

Файлы проекта

C++ Builder связывает с каждым своим приложением, т. е. с программой, созданной пользователем и оформленной как проект, следующие исходные файлы:

Исходный файл — такой файл, который подлежит компиляции.

- Unit1.cpp — текст программы (если при разработке программы в ней появятся новые формы, то для каждой формы будут построены свои программные и h-модули: Unit2.cpp, Unit2.h и т. д. Но мы для простоты рассматриваем проект из одной формы);
- Unit1.h — интерфейсный модуль с объявлениями компонентов, расположенных в данной форме, глобальных переменных, функций и т. д.;
- Unit1.dfm — здесь хранится описание формы и всех расположенных в ней компонентов. Этот файл поддерживается и обновляется средой C++ Builder автоматически, но его можно

открыть и посмотреть (но не корректировать!) с помощью команды **File/Open** главного меню. Приведем вид такого файла для одной формы, содержащей компонент — кнопку **TButton** (рис. 10.2);

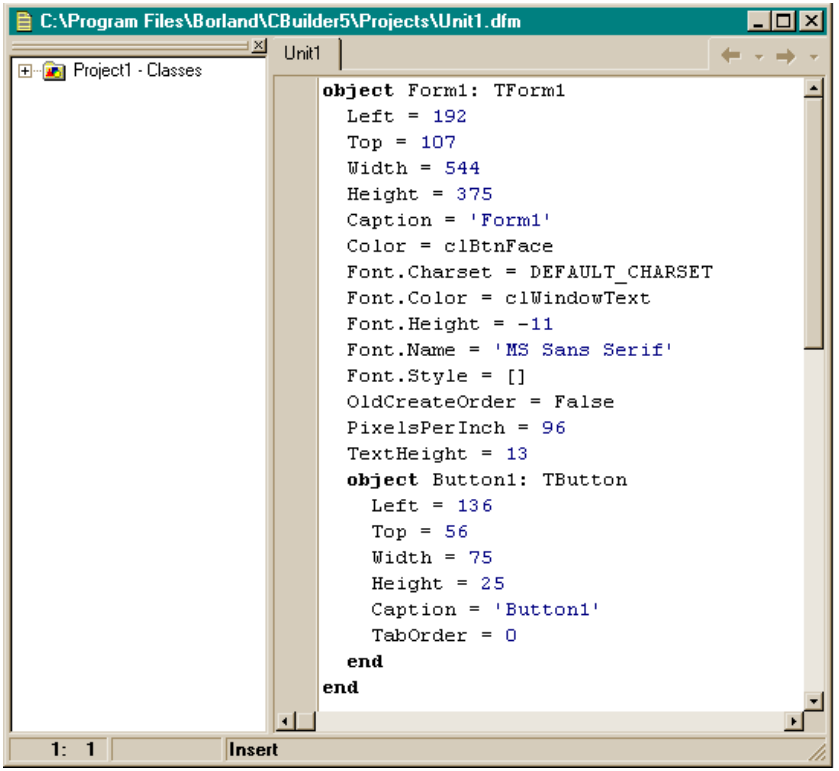


Рис. 10.2. Вид DFM-файла

- Project1.cpp — это главная управляющая программа проекта — проектный файл. Любая новая форма, программный модуль и другие элементы автоматически включаются в этот файл. Отсюда вызывается на выполнение, запускается и завершается исполняемый код проекта (после компиляции этот файл получит имя Project1.exe; он и вызывается на выполнение). Содержимое файла Project1.cpp можно посмотреть, вы-

полнив команду **View/Project Source** главного меню или вызвав проект с помощью Менеджера проектов: для этого требуется в подменю **Project** главного меню открыть контекстное меню элемента Project1.cpp и выбрать в нем команду **Open**;

- Project1.res — файл ресурсов проекта. В нем хранятся различные значки, используемые в проекте (часть из них могла быть задана разработчиком проекта, другие определены системой). Эти значки могут быть отредактированы с помощью редактора изображений, вызываемого командой **Tools/Image Editor** главного меню (рис. 10.3).

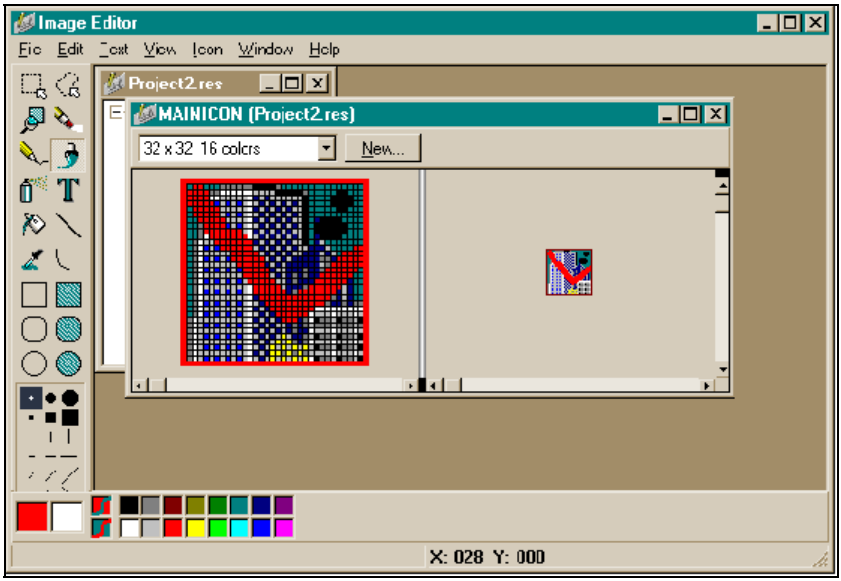


Рис. 10.3. Редактирование значков ресурсного файла

Видно, что рисунок отредактирован: на стандартный значок среды Builder помещена красная галочка и черные точки справа вверху;

- Project1.bpr — этот файл отвечает за проведение процесса сборки и компиляции проекта;

- если мы посмотрим в каталог, в котором находится весь проект, то обнаружим в нем еще другие файлы. Среди них файлы с расширением obj, из которых после компиляции модулей собирается exe-модуль.

Во время разработки приложения в целях защиты полезно время от времени делать промежуточные сохранения файлов проекта. Для этой цели служат команды подменю **File** из главного меню:

- **Save All** — сохраняет все исходные файлы под текущими именами;
- **Save** — сохраняет файлы с расширениями cpp и h под текущими именами;
- **Save As** — как и команда **Save**, сохраняет файлы с расширениями cpp и h, но позволяет при сохранении задать пользовательские имена модулям;
- **Save Project As** — сохраняет файлы проекта с расширениями bpr, cpp, res. Эту команду всегда надо выполнять первой при создании проекта (и консольного, и неконсольного). После того как выполнена эта команда, проект создан. А по ходу разработки следует применять команду **Save All**.

Инспектор объекта

Одновременно с открытием новой формы создается новый элемент, окно которого появляется на экране вместе с формой. Элемент называется **Object Inspector** или *Инспектор объекта* (рис. 10.4). Он появляется не только для каждой формы, но и для каждого помещенного в форму компонента: поместили в форму компонент — появился новый Инспектор. Вверху окна есть раскрывающееся поле (белая полоска, справа от которой находится кнопка в виде треугольника): в этом поле указывается имя по умолчанию и принадлежность к классу помещенного в форму компонента. Если щелкнуть мышью на кнопке, то появится выпадающий список, и в нем мы увидим имена всех помещенных в форму компонентов. Инспектор объекта позволяет увидеть основные свойства и события объекта, помещенного в форму. Полный перечень свойств, событий и методов компонента можно увидеть в справочной службе **Help**, нажав клавишу

<F1> при выделенном компоненте (чтобы выделить компонент, на нем надо щелкнуть левой кнопкой мыши). Инспектор объекта может быть скрыт и снова показан на экране с помощью клавиши <F11>.

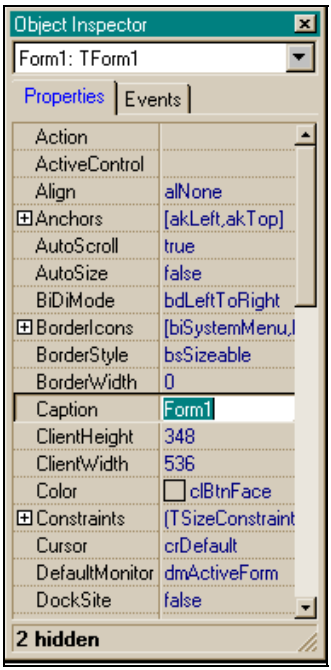


Рис. 10.4. Вид Инспектора объекта

Инспектор объекта имеет две вкладки: **Properties** и **Events**.

Вкладка *Properties*

Вкладка **Properties** позволяет манипулировать свойствами компонента (т. е. придавать им различные значения) на стадии проектирования приложения (как говорят, в режиме Design-Time). Чтобы устанавливать свойства компонента в режиме выполнения программы (или, как говорят, в режиме Run-Time), необходимо писать программы-обработчики событий компонента.

Свойство компонента может включать в себя другие свойства. Этот факт отмечается знаком + слева от имени базового свойства. Если на таком плюсе щелкнуть мышью, то выпадет список вложенных свойств, а вместо плюса появится минус. Если затем щелкнуть на этом минусе, то все встанет на свои прежние места: появится плюс и список вложенных свойств будет собран в одно основное свойство.

Примечание

Когда мы говорим о щелчке мышью, то имеем в виду щелчок левой кнопкой мыши. Щелчок правой кнопкой будем специально оговаривать, когда это потребуется.

Вкладка *Events*

Вкладка **Events** содержит список возможных событий, которые могут происходить с компонентом. Она дает возможность связать каждое событие с программой-обработчиком этого события: если дважды щелкнуть мышью в поле рядом с именем события, то C++ Builder создаст в модуле формы, в которую помещен компонент, программу-обработчик этого события. Это будет функция с заголовочной частью, но с пустым телом: не программа, а заготовка программы. В это пустое тело заготовки вы должны вписать свои команды, которые будут определять реакцию компонента на данное событие с учетом передаваемых функции фактических значений ее параметров. Вид пустого обработчика события `OnCreate` представлен на рис. 10.5, а вид окна Инспектора объекта для этого случая — на рис. 10.6.

Допустим, компонент `Button` помещен в форму. У этого компонента есть событие `OnClick`. Под нажатием имеется в виду щелчок мышью на кнопке. В созданной заготовке программы-обработчика вы можете написать реакцию компонента `Button` на событие `OnClick`: в форме должен быть нарисован красный шар (это уже делается с помощью другого компонента). При создании обработчика идет автоматическое переключение системы на вызов Редактора кода, курсор которого будет установлен на начало обработчика, чтобы вы могли вводить команды программы.

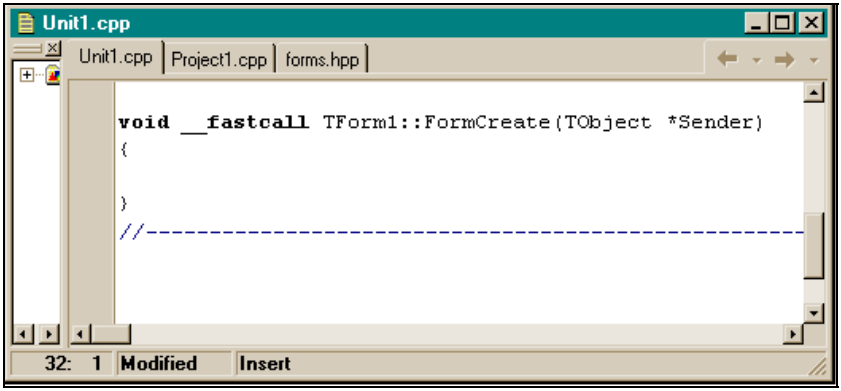


Рис. 10.5. Вид пустого обработчика события **OnCreate**

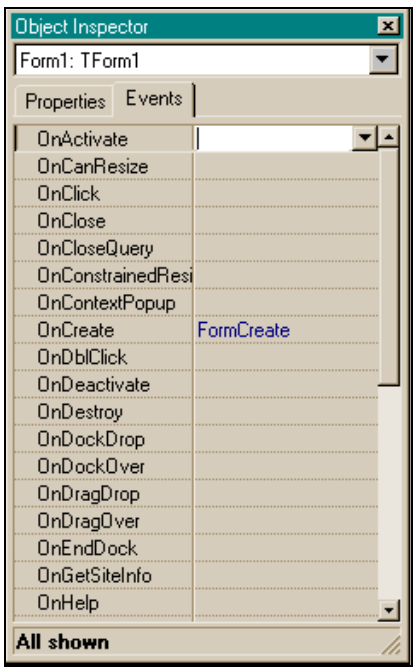


Рис. 10.6. Вид Инспектора объекта
с событием **OnCreate**

Мы уже отмечали, что в верхней части окна Инспектора объекта расположено поле, содержащее раскрывающийся список (рис. 10.7).

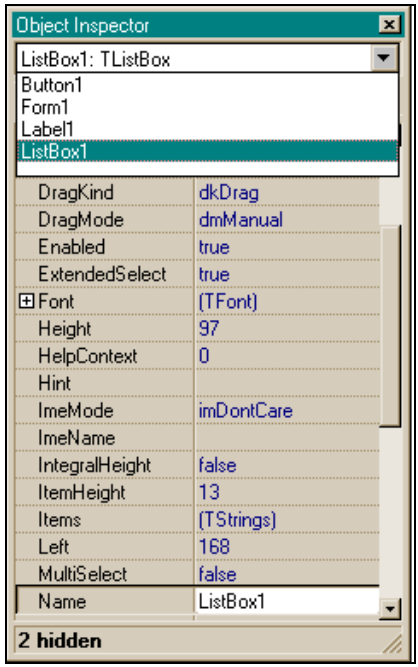


Рис. 10.7. Раскрывающийся список в окне Инспектора объекта

В этом списке находятся не только имена всех компонентов, помещенных в форму, но и имя активной в данный момент формы (той, с которой мы работаем в данный момент). С помощью этого списка мы можем выбирать любой компонент, помещенный в форму, и работать с ним. Для этого надо только щелкнуть на имени нужного компонента мышью, который сразу станет активным (помеченным), и в окне Инспектора объекта появятся данные уже по этому, активному, компоненту. Следовательно, с помощью списка можно легко переключаться с одного компонента на другой. В шестой версии среды одновре-

менно с окном Инспектора объекта появляется окно дерева просмотра объекта, в котором видна вся иерархия приложения и с помощью которого можно продвигаться по компонентам форм, не применяя раскрывающийся список Инспектора объекта.

Работа с Инспектором объекта

Ширину столбцов в окне Инспектора можно менять, перетаскивая мышью разделительные линии (как и само окно можно перетаскивать в любое место экрана по обычным правилам перетаскивания окон в Windows). Инспектор объектов имеет свое контекстное меню, которое, как и любое контекстное меню Windows, открывается с помощью правой кнопки мыши и содержит команды, смысл которых можно посмотреть в разделе **Help**, если установить курсор мыши на команду и нажать <F1>. Одной из команд меню является **Help** — вызывает соответствующую страницу справочной службы.

Редактор кода, cpp-модуль и h-файл

Когда открывается новая форма, к ней создается два файла: один — для программ-обработчиков событий (в нем также находится программа "конструктор формы"), его расширение — cpp, а другой, с расширением h, интерфейсный файл. Обоим файлам система присваивает имена по умолчанию. Например, для первой формы это будут Unit1.cpp и Unit1.h. Последний можно увидеть, если открыть контекстное меню Редактора кода, который открывается сразу, как только создается программный модуль Unit1.cpp. Попасть в программный модуль после загрузки проекта, когда на экране появится форма и окно ее Инспектора объекта, можно с помощью клавиши <F12>: нажмите эту клавишу и попадете в окно Редактора кода. В нем будет находиться модуль Unit1.cpp, готовый к редактированию, т. е. к внесению в него команд (рис. 10.8).

Как видим, cpp-модуль содержит указатель на класс `TForm` (т. е. на нашу пока единственную форму) и программу "конструктор"

для создания экземпляра класса `TForm`. Описание же класса находится в h-файле, вид которого показан на рис. 10.9.

Мы уже говорили, что добраться до h-файла можно, если установить указатель мыши в поле окна Редактора кодов, вызвать контекстное меню, нажав правую кнопку мыши, и выполнить команду **Open Source/Header File**.

Посмотрим на h-файл. Интересно, как среда C++Builder формирует программу-приложение. Главной при создании приложения является форма. С нее все начинается. Она первой вставляется в проект, а в нее уже помещаются другие компоненты. Когда создается новое приложение, форма, вставленная в проект, "рождается" довольно оригинально. С одной стороны, эта форма должна быть компонентом класса `TForm`, а с другой стороны, она впоследствии должна вместить в себя другие компоненты, из которых будет строиться приложение. Кроме того, приложение может содержать несколько форм с компонентами, и этот факт надо учитывать.

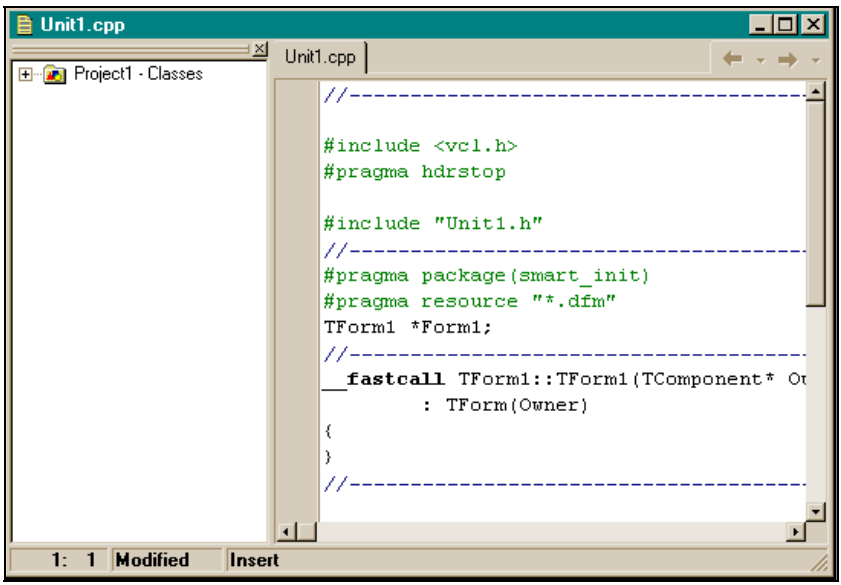


Рис. 10.8. Окно Редактора кода, в котором расположена заготовка модуля

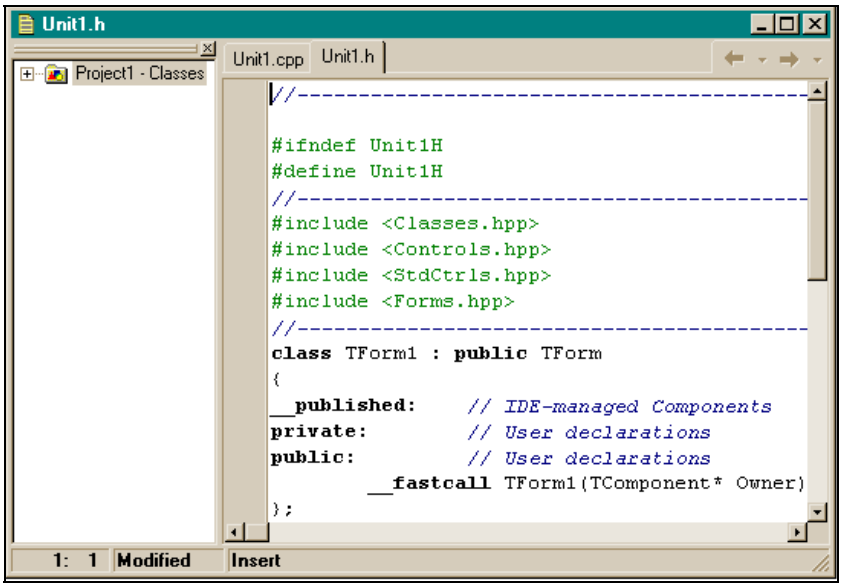


Рис. 10.9. Вид h-файла

Разработчики C++ Builder вышли из этого положения следующим образом. Любая форма, вставляемая в проект (в том числе, естественно, и первая), получает имя Form $_i$, где $i = 1, 2,$ Это делается для того, чтобы отразить, что вставляемая в проект форма будет связана с компонентом класса `TForm`. Затем в одном из создаваемых к каждой форме модулей — h-файле — объявляется рабочий класс (т. е. создаваемый на время действия будущего приложения) с именем `TForm` $_i$, наследник класса `TForm`. А это означает, что рабочий, вновь создаваемый класс, унаследует члены класса `TForm`. Для этого рабочего класса `TForm` $_i$ создаются свои секции, в том числе и секция `__published`, которая будет наполняться компонентами, помещаемыми в форму при проектировании приложения. Мы видим, что класс `TForm` $_i$ как раз и является тем классом, который решает проблему использования членов класса `TForm` и содержания в себе помещаемых в форму компонентов. Но чтобы пользоваться данными класса `TForm` $_i$, в другом из создаваемых к каждой форме модуле — cpp-файле — объявляется указатель с именем `Form` $_i$ на класс `TForm` $_i$, что по-

зволяет обращаться к членам класса TForm $_i$ в виде: Form$_i$->имя члена класса.

Такое обращение вполне естественно: пользователь ввел в проект, скажем, пятую форму и обращается к ее членам как Form5->...

Поместим теперь в форму один компонент, например, TButton, и посмотрим, что произойдет с h-файлом (рис. 10.10).

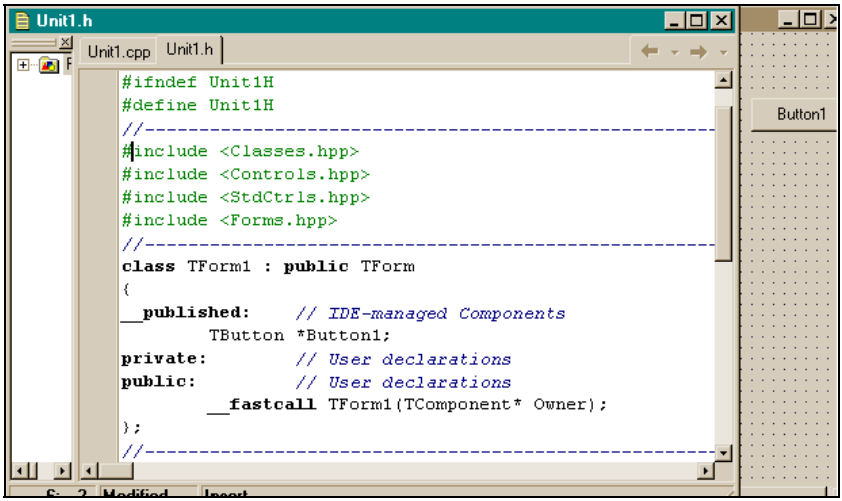


Рис. 10.10. Вид h-файла после помещения в форму компонента TButton

Мы видим, что в h-файле в секции __published появился указатель на класс TButton, т. е. в классе TForm1 появился элемент-член класса, и что имя, присвоенное ему по умолчанию, совпадает с именем самого компонента (Button), но к нему добавлен порядковый номер этого компонента в форме (можно убедиться, что если поместим в форму вторую кнопку, то ее именем станет Button2). Здесь действует тот же принцип именования, что и для формы.

Исходя из сказанного выше, к этой кнопке следует обращаться так: Form1->Button1. Но и к свойствам кнопки как класса, на который объявлен указатель Button1, надо обращаться в виде

`Button1->член класса TButton`. Чтобы обратиться, например, к свойству `Caption` кнопки, надо написать `Form1->Button1->Caption`. С другой стороны, поскольку члены класса попадают в секцию `__published`, их свойства и события будут отражаться в окне Инспектора объекта.

Создадим теперь обработчик события кнопки и посмотрим, как это отразится в h-файле. Возьмем событие `OnClick` — нажатие кнопки мышью. Сделаем так, чтобы при нажатии кнопки форма закрывалась. Вид обработчика показан на рис. 10.11, а вид h-файла — на рис. 10.12.

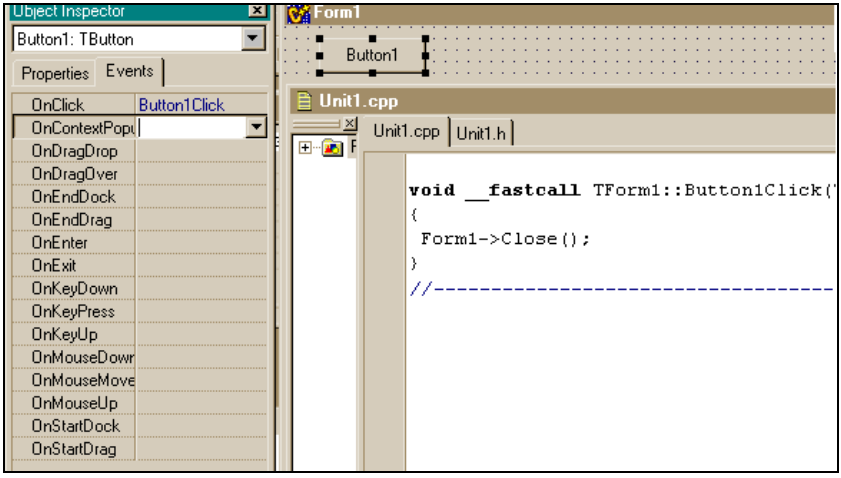


Рис. 10.11. Обработчик события `OnClick`

Мы видим, что в h-файле в секции `__published` появился новый член: это функция-метод-обработчик события `OnClick`. Следовательно, для выполнения этого метода к нему следует обращаться как к члену класса: `Form1->Button1Click(Button1)`. Заметим, что имена обработчикам событий даются системой тоже по определенному правилу: к имени компонента добавляется имя события. Это следует помнить, т. к. облегчается работа с cpp-модулем, содержащим много обработчиков.

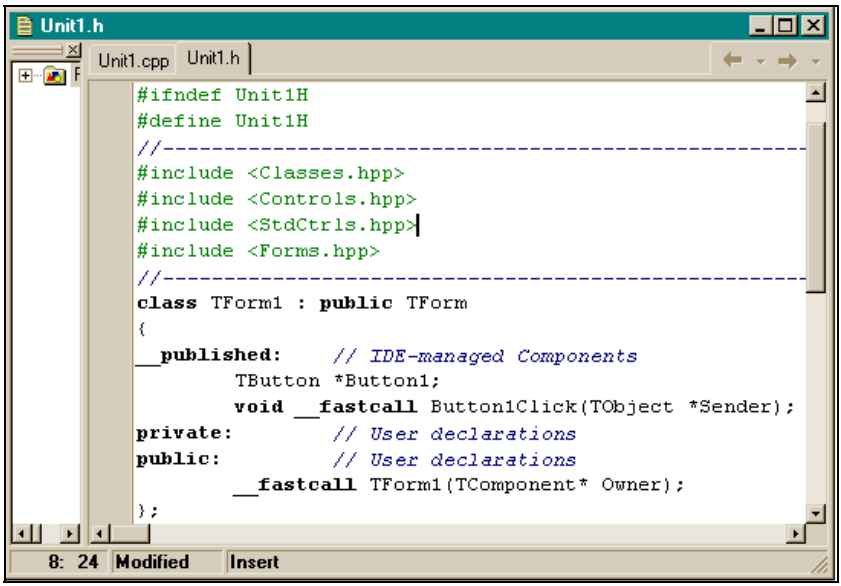


Рис. 10.12. Вид h-файла
с обработчиком события кнопки

Запишем в h-файле в секции `public` перед строкой:

```
__fastcAll TForm1 (TComponent *Ownner)
```

Объявление переменной: `int a;`. Вид h-файла после этого показан на рис. 10.13.

Эта переменная будет глобальной для всего cpp-модуля формы `Form1`, т. е. станет известной во всех программах-обработчиках событий компонентов формы.

Вернемся теперь к Редактору кода, который позволил нам попасть в h-файл и увидеть в нем немало интересного. Окно редактора может содержать много вкладок. Здесь отражаются вкладки всех модулей форм проекта. Переключение с вкладки на вкладку выполняется щелчком мыши на вкладке. Первой вкладкой всегда становится вкладка для cpp-модуля первой формы проекта.

По левому краю окна Редактора проходит так называемое поле подшивки. В этом поле расставляются точки останова (элементы программы-отладчика) для отладки программы. Здесь же расставляются и специальные закладки (bookmarks), которые обеспечивают удобную работу с текстом. Если в некотором месте текста сделать закладку, то потом из любого другого места текста одним щелчком мыши можно возвратиться туда, где сделана закладка.

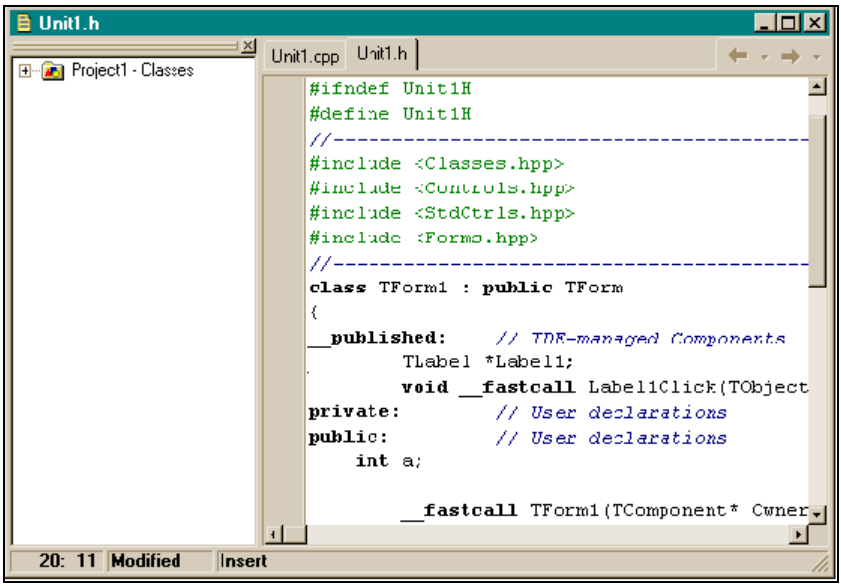


Рис. 10.13. Вид h-файла после помещения объявления переменной в секцию `public`

Новая вкладка в окне Редактора кода появляется при выполнении следующих команд главного меню:

- **File/New Application** — создание нового приложения. При этом создается новый проект (мы видели, что любое приложение оформляется в виде проекта) с новой (первой) формой. Для проекта создается вкладка Project1.cpp (ее можно увидеть, выполнив команду **Project/View Source** главного ме-

ню), а для формы — вкладки Unit1.cpp и Unit1.h. Project1.cpp и есть *главный модуль проекта*, содержащий *главную функцию* `WinMain()` — аналог консольной функции `main()`. В этом модуле создаются в памяти формы и запускается на выполнение само приложение;

- **File/New Form** — в проекте создается новая форма (она добавляется к проекту).

Как начать редактирование текста программного модуля

Чтобы начать редактировать текст модуля, надо:

1. Открыть вкладку с именем требуемого модуля в окне Редактора кода (для переключения между окном Редактора и формой надо нажать <F12>). Можно открыть диалоговое окно командой **View/Units** главного меню и в нем выбрать требуемый модуль (рис. 10.14).

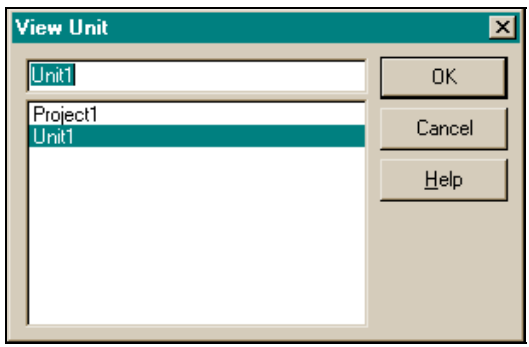


Рис. 10.14. Диалоговое окно для выбора модуля

2. Поместить курсор Редактора в нужное место текста и редактировать его по правилам стандартных текстовых редакторов Windows. Текст можно перетаскивать, если предварительно его выделить, а затем перетянуть мышью в необходимое место.

Примечание

Чтобы поместить курсор Редактора кода в нужное место текста, можно щелкнуть в этом месте мышью либо воспользоваться стрелками клавиатуры или клавишами <Home>, <End>, <Ctrl>+<Home>, <Ctrl>+<End>, <Page Up>, <Page Down> и т. п.

Контекстное меню Редактора кода

У Редактора есть свое контекстное меню, которое, как и все контекстные меню, открывается правой кнопкой мыши. Это меню содержит следующие команды:

- **Open Source/Header File** — переключает вкладки **cpp — h** активного модуля;
- **Close Page** — закрывает открытую (текущую) вкладку и убирает ее из состава вкладок Редактора. Если данный модуль имел до этого несохраненные изменения, система предложит их сохранить (рис. 10.15);

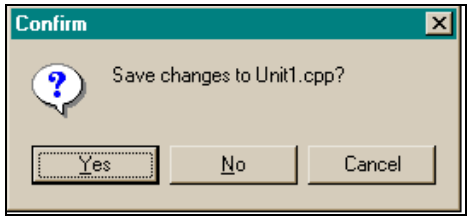


Рис. 10.15. Диалоговое окно для сохранения модуля при его закрытии

- **Open File at Cursor** — открывает диалоговое окно для поиска cpp-файла и после его открытия включает файл в Редактор кода, создавая для него новую вкладку. Однако этот файл в проект не включается. Из него можно брать готовые куски и включать в ваш модуль;
- **New Edit Window** — открывает новое окно Редактора кода, благодаря которому можно одновременно видеть части кода, расположенные далеко друг от друга, что удобно при редактировании;

- **Topic Search** — выдает справку о выделенном слове в тексте. Эта команда эквивалентна нажатию клавиши <F1>: слово надо выделить (при этом оно подкрасится синим цветом), а затем нажать <F1>, в результате чего начнет работать раздел системы **Help**;
- **Toggle Bookmarks** — вставляет и убирает закладку (действует, как переключатель: нажал — появилась, еще раз нажал — исчезла). В поле подшивки Редактора появляется зеленая книжечка с номером закладки (закладок можно делать много);
- **Goto Bookmarks** — если установить курсор мыши на этой команде, появится список закладок, из которого щелчком можно выбрать, на какую закладку перейти;
- **Debug** — открывает меню отладчика; пользуясь этим меню, можно создать точки останова программы, осуществлять переход от одной такой точки к другой и т. д.;
- **Read Only** — запрещает корректировку модуля;
- **Message View** — открывает окно, куда помещаются сообщения, полученные в ходе компиляции и сборки программы компилятором;
- **Propertis** — открывает диалоговое окно настроек Редактора, показанное на рис. 10.16.

Вкладки этого окна позволяют изменять параметры настройки Редактора:

- устанавливать общие настройки (автоматический отступ, вставка, табуляция, и т. д.);
- устанавливать режимы вывода на экран (форма курсора, ширина правого и левого полей отступа, шрифт);
- управлять цветом и размером шрифта для выделения различных синтаксических конструкций (ключевых слов, комментариев и т. п.);
- организовывать различные подсказки с помощью так называемого "суфлера кода" (используя, например, вкладку **Cod Insight**, можно в ней задать громоздкую конструкцию, которая часто встречается при составлении программ, а затем включать эту конструкцию в программу с помощью суфлера кода, т. к. она попадает в его список вставок) и т. п.

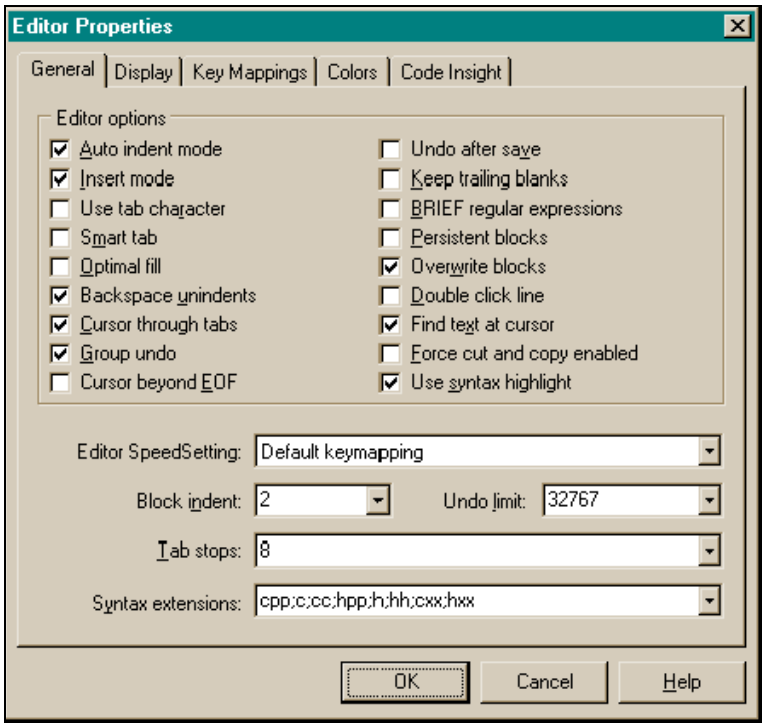


Рис. 10.16. Диалоговое окно для настройки свойств Редактора кода

Суфлер кода (подсказчик)

Суфлер кода помогает быстрее и правильнее набирать текст программы: выдает оперативную информацию (подсказку) при наборе текста. Основные подсказки приведены ниже.

- Если вы ввели имя некоторого объекта и поставили после него точку (а это — оператор прямого членства в классе или в структуре) или стрелку вправо (это оператор выбора члена класса через указатель на этот класс), то появится подсказка: имена всех членов данного класса. Остается установить, например, с помощью стрелок, линейку подсветки на нужный элемент класса и нажать <Enter>, в результате чего элемент

запишется в программу. Например, набираем элемент `Button1->`. Суфлер покажет список всех элементов класса `TButton`. Выбираем элемент `Width` и нажимаем <Enter>. Получим в программе:

```
Button1->Width
```

Мы обратились к свойству "ширина кнопки". Если записанному элементу требуется присвоить еще какие-либо свойства, набираем <Ctrl>+<пробел>. Справа от знака присвоения высветится перечень элементов, разрешенных к записи. Здесь нужно выбрать требуемый элемент и нажать <Enter>.

- Как только будет набрано имя метода и знак открывающей скобки, появится подсказка по параметрам: список имен и типов аргументов для вызова метода (функции). Подсказки выдаются также по параметрам функций `Windows API`.
- Достаточно набрать <Ctrl>+<J>, как появится подсказка для выбора кодовых шаблонов (различных операторов языка, его конструкций): высветится перечень готовых шаблонов, из которого необходимо выбрать требуемый и нажать <Enter>. Выше мы отмечали, что такие шаблоны можно задавать, используя команду **Propertis** контекстного меню Редактора и вкладку **Cod Insight** открывающегося при этом диалогового окна.
- Когда программа запущена и приостановлена с помощью точки останова, можно просмотреть значения идентификаторов программы. Для этого надо навести на имя идентификатора курсор мыши и немного подождать. Появится подсказка со значениями идентификатора (рис. 10.17).
- Если навести указатель мыши на имя переменной в выполняемой и приостановленной программе, то высветится значение этой переменной (на картинке выше мы навели мышь на экземпляр класса. Точно так же можно было бы навести мышь и на переменную `i` и увидеть ее значение).
- С помощью подсказчика можно определить характеристики объявленного в тексте программы идентификатора: это делается в режиме разработки. Наводим на имя объекта мышь и ждем. Появляется подсказка (рис. 10.18).

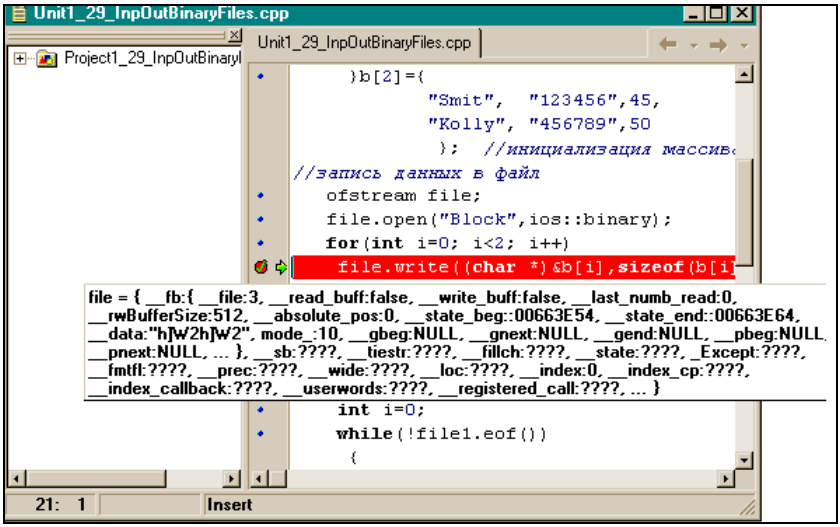


Рис. 10.17. Всплывающая подсказка

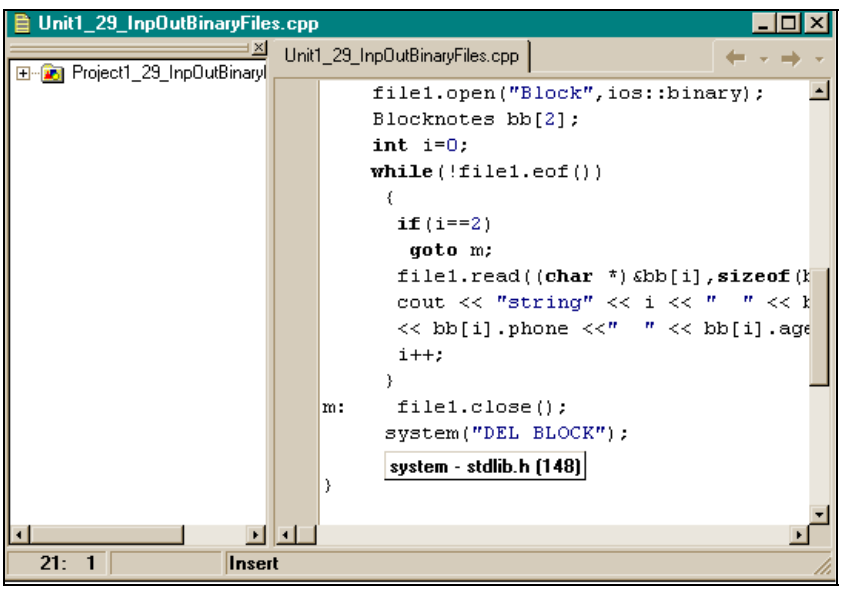


Рис. 10.18. Подсказка по идентификатору

В нашем примере мы навели курсор мыши на `system` — системную функцию. Появилась подсказка: описание этой функции находится в библиотеке stdlib.h в строке 148.

Перейдем далее к рассмотрению компонентов среды программирования Borland C++ Builder.

Класс *TForm*

Дизайнер форм

Пустая форма, которая появляется на экране после загрузки среды Builder или когда мы создаем новое приложение, фактически представляет собой окно Дизайнера форм (ДФ). Он позволяет работать с формой: помещать компоненты в форму, удалять компоненты с формы, закрывать форму, выделять компоненты в форме, перетаскивать их с одного места на другое и т. д.

Помещение компонента в форму

Поместить компонент в форму можно одним из следующих способов:

- найти нужный компонент на вкладках палитры компонентов, щелкнуть на нем мышью, перевести указатель мыши в нужное место окна Дизайнера форм и снова щелкнуть мышью. Значок компонента появится в форме в своем активном состоянии: он будет окружен черными квадратиками, за которые можно компонент растягивать. Если вы при этом тем же способом поместите другой компонент в форму, то последний станет активным, а все остальные — неактивными;
- дважды щелкнуть на нужном компоненте на вкладке;
- если требуется поместить несколько экземпляров одного компонента в форму, то надо нажать клавишу <Shift> и, не отпуская ее, щелкнуть на нужном компоненте на вкладке палитры, а затем (клавишу <Shift> уже можно отпустить) щелкать мышью в необходимых местах формы. Там станут появляться экземпляры компонента. Для прекращения такого размножения следует дважды щелкнуть на компоненте на

вкладке, он изменит свой выпуклый вид и "отвяжется" от мыши: при этом все-таки еще один лишний экземпляр в форме появится. Но вы его легко удалите: т. к. он — последний, то будет активным. Стоит нажать клавишу <Delete> и активный компонент исчезнет с формы. Так можно удалить любой компонент: щелкнуть на нем, сделав его активным, и нажать <Delete>. Как только компонент после щелчка становится активным, для него тут же открывается окно Инспектора объекта.

Другие действия с Дизайнером форм

- Закрыть активное окно Дизайнера форм можно либо нажав кнопку с крестом в верхнем правом углу формы, либо клавишами <Alt>+<F4>, либо командой **Close** подменю **File** главного меню.
- Чтобы выделить несколько компонентов (например, если требуется их групповое перемещение в другое место формы), можно либо при нажатой клавише <Shift> щелкать мышью на требуемых компонентах, либо заключить эти компоненты в так называемый прямоугольник выделения: надо установить курсор мыши левее и выше самого верхнего левого компонента формы, нажать на левую кнопку и, не отпуская ее, потянуть мышь вправо и вниз. Появится прямоугольник, который будет охватывать выделенные объекты. После того как вы отпустите клавишу мыши, объекты будут отмечены по периметрам серыми квадратиками — признак группового выделения. Снять групповое выделение компонентов можно щелчком мыши вне выделенного пространства.
- Можно изменять форму выделенного компонента на более мелком уровне, чем растяжка за черные квадратики. Для этого при нажатой клавише <Shift> надо нажимать нужную клавишу со стрелкой.
- Таким же способом, с помощью стрелок, можно мелкими долями изменять местоположение отмеченного компонента, но при нажатой клавише <Ctrl>.

Контекстное меню формы

Действия над формой и ее содержимым можно выполнять с помощью ее контекстного меню. Чтобы оно появилось, достаточно нажать правую кнопку мыши. Каждой команде контекстного меню соответствует одноименная команда из подменю **Edit** главного меню:

- **Align to Grid** — притягивает выделенный компонент или группу компонентов к ближайшему узлу координатной сетки формы (в форме расположены точки: это и есть координатная сетка);
- **Bring to Front** — перемещает выделенный компонент на самый верх всей перекрывающейся стопки компонентов;
- **Send to Back** — перемещает выделенный компонент под стопку компонентов (он будет в самом низу);
- **Revert to Inherited** — восстанавливает унаследованное поведение объектов компонента (мы подробно рассматривали эту команду в *разд. "Работа с Инспектором объекта" данной главы*);
- **Align** — открывает диалоговое окно для выравнивания отмеченных компонентов. Это окно показано на рис. 10.19. Такого же эффекта можно достичь, если выполнить команду **View/Alignement Palette** главного меню;

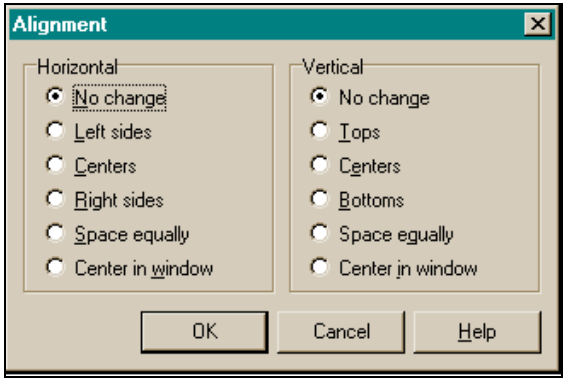


Рис. 10.19. Вид диалогового окна для выравнивания компонентов

- **Size** — открывает окно диалога установки одинаковых размеров компонента (рис. 10.20);

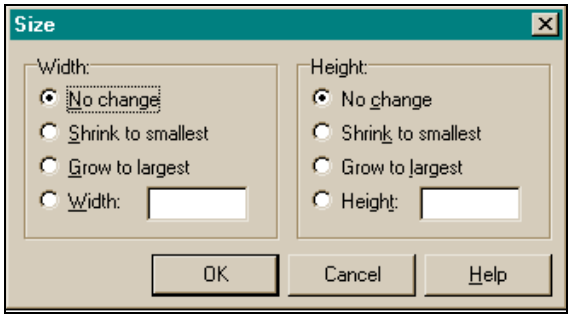


Рис. 10.20. Окно диалога установки одинаковых размеров компонента

- **Scale** — открывает диалоговое окно (рис. 10.21) для задания величины пропорционального масштабирования формы и всех ее компонентов: насколько при изменении размеров формы будут изменяться размеры всех компонентов, расположенных в ней (задается процент изменения компонентов по отношению к изменению размера формы);

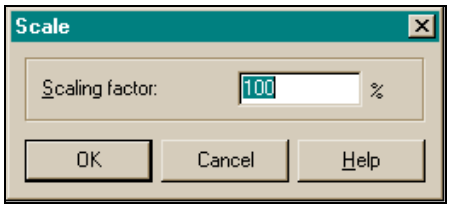


Рис. 10.21. Диалоговое окно для задания величины пропорционального масштабирования формы и всех ее компонентов

- **Tab Order** — открывает диалоговое окно (рис. 10.22), в котором устанавливается порядок перехода для видимых компонентов (те, что видны в форме, когда приложение исполняется) от компонента к компоненту при последовательном

нажатии клавиши <Tab>. По умолчанию порядок активизации компонентов с помощью <Tab> определяется порядком их помещения в форму. Изменение порядка бывает полезным, например, если в форме много полей ввода данных в базу данных и требуется определить порядок перехода от одного поля к другому (т. е. порядок их активизации). На рисунке показано, что порядок активизации трех кнопок таков: 1-я, 3-я, 2-я;

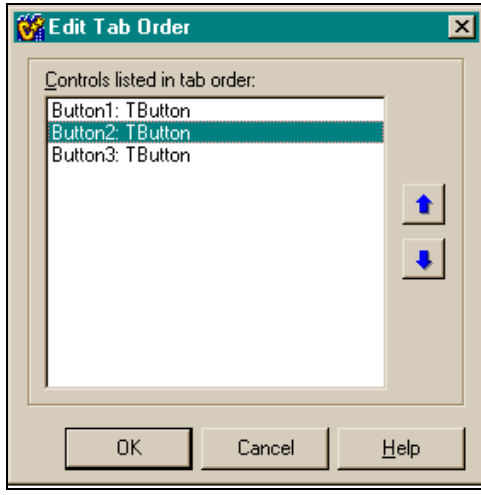


Рис. 10.22. Диалоговое окно для изменения порядка активизации видимых компонентов

- **Creation Order** — открывает окно диалога для определения последовательности создания в памяти невидимых компонентов либо при загрузке формы, либо в режиме проектирования, либо в режиме исполнения.

Примечание

Невидимые компоненты — это те, которых не видно в форме в момент исполнения приложения. Примером таких компонентов могут служить компоненты работы с базами данных.

На рис. 10.23 приведен пример переупорядочения невидимого компонента `TTable`: в форме расположены три объекта, полученные из `TTable`: `Table1`, `Table2`, `Table3`. В порядке, определенном в этом окне, ваше приложение будет создавать невидимые компоненты, когда вы загружаете форму, в режиме проектирования (Design-Time) или исполнения (Run-Time). По умолчанию такой порядок определяется порядком размещения этих компонентов в форме;

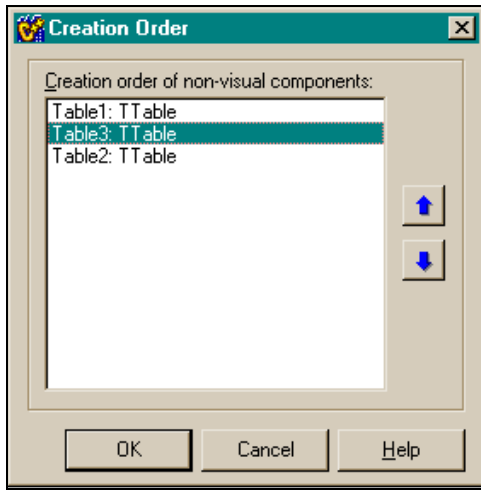


Рис. 10.23. Диалоговое окно для изменения порядка создания невидимых компонентов в памяти

- **Flip Children** — зеркально отобразить потомки: дает зеркальное отображение (перемещение) по горизонтали относительно середины формы либо всех компонентов, расположенных в форме, либо выделенных компонентов;
- **Add to Repository** — открывается окно диалога добавления формы в Хранилище объектов, чтобы впоследствии данную форму можно было извлекать из Хранилища и использовать в других проектах. Вид окна показан на рис. 10.24;
- **View as text** — эта опция позволяет просмотреть текстовое описание формы и всех компонентов, находящихся в ней;

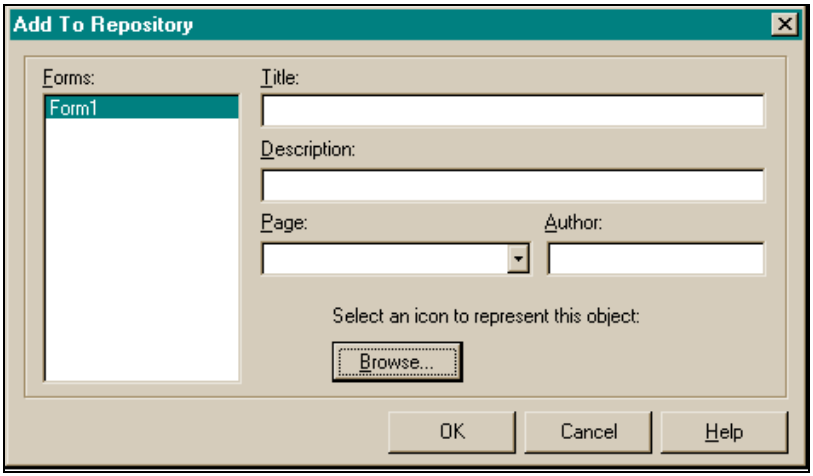


Рис. 10.24. Диалоговое окно для помещения объекта в Хранилище объектов

- **Text DFM** — это переключатель форматов, в которых ваша форма может быть сохранена. Файлы формы могут быть сохранены в одном из двух форматов: бинарном или текстовом. Текстовые файлы легче модифицировать разными средствами управляющей системы. Бинарные файлы имеют обратную совместимость с более ранними версиями C++ Builder (могут обрабатываться более ранними версиями). По умолчанию форма сохраняется в текстовом формате.

Добавление новых форм к проекту

Алгоритмы решения некоторых задач требуют, чтобы компоненты, обеспечивающие решение таких задач, размещались на разных формах и чтобы эти формы вызывались в процессе выполнения проекта в определенном порядке. Например, возьмем задачу "Управление кадрами". Лучше было бы, если бы все справочные данные по кадрам были размещены отдельно от аналитических форм: так удобнее поддерживать справочную информацию, потому что этим вопросом сможет заниматься отдельный работник, и ему нечего делать в разделе аналитических форм. Да и работнику, связанному с анализом кадров, легче ре-

шать свои проблемы, не заботясь о поддержке в актуальном состоянии нормативно-справочной информации.

Среда C++ Builder предполагает, что форма, которая появляется на экране, когда проект создается, будет главной в проекте, т. е. среди всех прочих форм проекта она будет вызвана на выполнение первой, но это в том случае, если не изменить порядок загрузки форм. Такая возможность в среде есть. Поэтому при разработке тех же "Кадров" вы можете в первой форме помещать справочную информацию, потом добавить к проекту новую форму, сделать ее главной и работать с ней: в ней размещать обработку аналитических таблиц (может даже и не в одной форме) и т. д. Чтобы добавить новую форму к проекту, надо выполнить команду главного меню среды **File/New Form**, в результате чего на экране появится новая (пустая) форма.

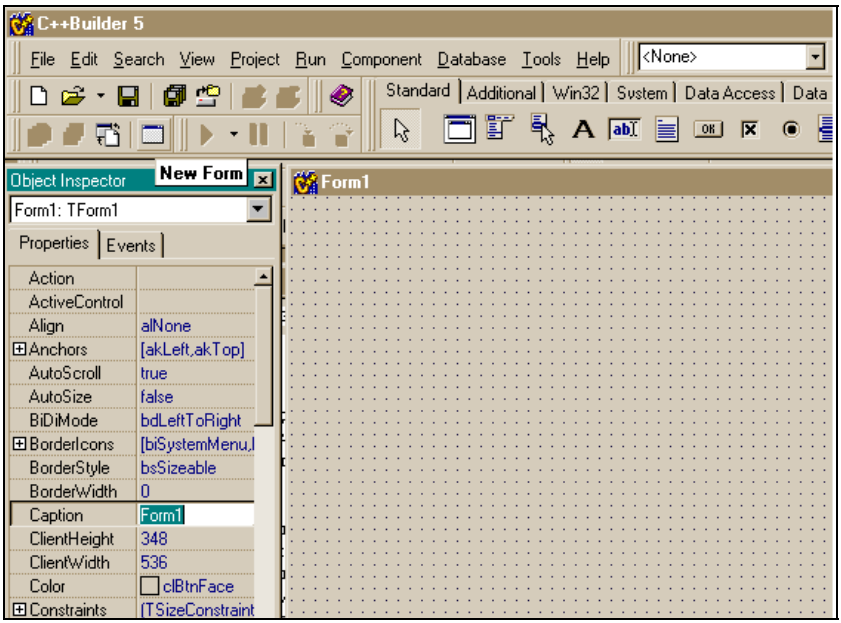


Рис. 10.25. Добавленная к проекту форма

Для добавки новой формы, как и для многих других операций, можно воспользоваться кнопками быстрого вызова: в частности, для добавки формы можно нажать мышью кнопку **New Form**. Результат этой операции представлен на рис. 10.25.

На картинке видна кнопка **New Form**: на панели меню и инструментов это третья строка сверху и четвертая кнопка слева. Если на кнопки, расположенные на этих панелях, навести курсор мыши и немного подождать, то появится подсказка с названием кнопки (в нашем случае эта подсказка видна).

Организация работы с множеством форм

Итак, форма, с которой был связан проект при его создании, имеет статус главной. Это означает, что она первой загружается и выполняется после того, как проект откомпилирован и запущен на выполнение. Если в проекте много форм, то из главной формы можно организовать вызов на выполнение остальных форм (а можно и из каждой формы вызвать любую другую). Эту задачу можно решить следующими способами:

- с помощью обработчиков кнопок вызова конкретных форм;
- с помощью меню.

C++ Builder позволяет сделать главной любую из форм проекта: достаточно выполнить команду **Projects/Options** главного меню. При этом откроется диалоговое окно со многими вкладками, из которых надо выбрать вкладку **Forms** (рис. 10.26)

В раскрывающемся списке **Main Form** надо выбрать ту форму, которую вы хотите сделать главной, и щелкнуть на ней мышью. Она попадет в окно, после чего следует нажать кнопку **OK**.

При разработке проекта на экране всегда находится какая-то одна форма. Если нам требуется поместить другую форму на экран из уже добавленных ранее в проект, то следует это сделать либо кнопкой быстрого вызова, либо командой **View/Forms** главного меню. В результате появится диалоговое окно, в котором вы выберете нужную вам форму (рис. 10.27).

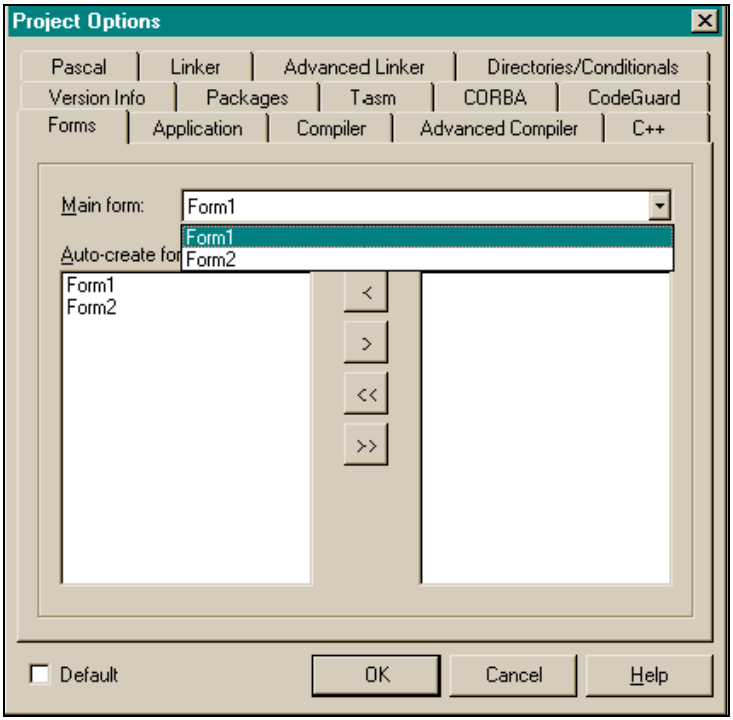


Рис. 10.26. Диалоговое окно для выбора главной формы

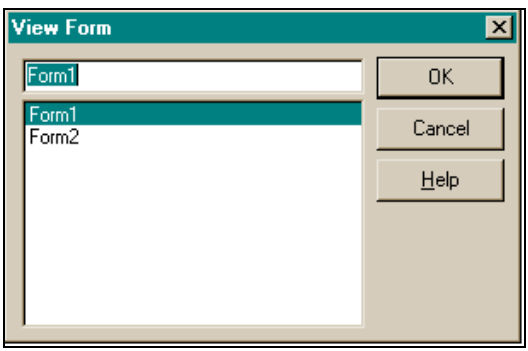


Рис. 10.27. Диалоговое окно для выбора формы

Вызов формы на выполнение

Вызов формы на выполнение может быть двух типов: *модальный и немодальный (обычный)*. При вызове формы в немодальном (обычном) режиме выполняется метод класса `TForm`, который называется `Show()`. То есть пишут: `Form1->Show();`. В обычном режиме можно вызывать сколько угодно форм одну за другой. При этом происходит большой расход памяти, т. к. все вызванные формы могут быть не закрыты (кстати, это делается методом `Close()`: `Form1->Close();`).

Примечание

Все методы, свойства и события любого компонента можно посмотреть, воспользовавшись справочной информацией среды. Для быстрого входа в нее следует активизировать компонент (щелкнуть на нем мышью), а затем нажать клавишу <F1>. Можно также воспользоваться командой **Help** главного меню.

Если же форма вызвана в так называемом модальном режиме (с помощью оператора `Form1->ShowModal();`), то приложение не сможет дальше нормально работать, пока модально вызванная форма не будет закрыта. В формах, которые вызываются модально, делают кнопки с именем **Close** и в их обработчиках помещают операторы:

```
ModalResult='OK'; Formi->Close(); Form1->Show();
```

Здесь предполагается, что `Formi` — это некая модальная форма (т. е. та, которая открывается в модальном режиме), а `Form1` — главная форма, из которой вызывалась форма `Formi`. Если же, например, форма `Formi` вызывалась из `Form1` обычным образом, то в обработчике ее кнопки `Close` можно было бы написать:

```
Formi->Close(); Form1->Show();
```

Более подробно о модальном открытии и закрытии формы будет сказано в *разд. "Компонент TButton" данной главы.*

Свойства формы

Эти свойства могут встречаться у многих компонентов, которые будут изучаться дальше, поэтому впоследствии будем говорить о

них кратко. В Инспекторе объекта, как известно, отображаются не все свойства объекта (только те, что попадают в секцию `published`). Все свойства объекта, в том числе отображаемые в Инспекторе объекта, можно посмотреть в разделе **Help** по данному объекту (надо при выделенном объекте нажать <F1>). Кроме того, о свойствах объекта можно узнать, если воспользоваться вкладкой **Legend**, находящейся в оглавлении списка свойств. Итак, свойства формы приведены ниже.

- `Caption` — сюда помещается название формы. По умолчанию первая форма проекта получает имя `Form1`, вторая — `Form2` и т. д.
- `ActiveControl` — это свойство определяет, какой компонент, помещенный в форму, в данный момент является активным(или, как говорят, имеет фокус).
- `AutoScroll` — это свойство определяет, будут ли автоматически появляться полосы прокрутки формы, если в ней не будут помещаться компоненты (т. е. чтобы их увидеть, надо будет форму "прокрутить"). Если значение этого свойства `true`, то полосы прокрутки будут появляться, а если `false` — то не будут).
- `AutoSize` — если значение свойства `true`, то форма автоматически принимает прежние размеры, как бы вы ее не растягивали. При свойстве, равном `false`, форма "позволяет" себя растягивать.
- `BorderStyle` — свойство задает появление и поведение границ формы: можно ли мышью менять размеры формы, когда приложение находится в режиме исполнения.
- `BorderWidth` — здесь задается в пикселах величина отступа координатной сетки формы от границ окна формы, т. е. фактически размеры формы можно изменить за счет изменения координатной сетки, задав ее отступ от границ окна. По умолчанию величина отступа равна нулю, т. е. форма занимает все пространство окна (рис. 10.28).

  На рисунке видно, что свойство имеет значение 100 пикселов (величину следует набрать в поле напротив названия свойства и затем нажать <Enter>). В форме находится кнопка, которая при таком отступе не умещается в новые разме-

ры формы. Поэтому у формы сразу появилась вертикальная полоса прокрутки, чтобы возможно было следить за "убежавшей" кнопкой.

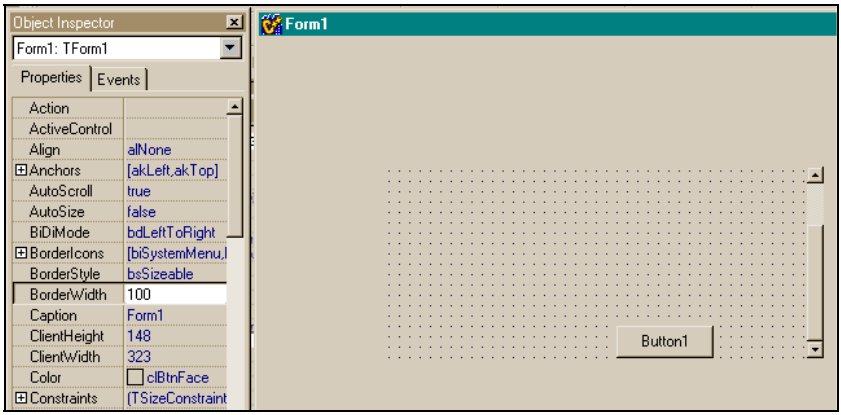


Рис. 10.28. Изменение размера формы в окне

- `Canvas` — обеспечивает пользователю возможность рисования в форме. Задает на плоскости формы битовый холст, на котором и можно рисовать. Это свойство само является классом и имеет свои свойства и методы, обеспечивающие рисование различных фигур (точек, линий, прямоугольников и т. п.). Нарисуем в форме график некоторой функции $y = f(x)$. Для создания графика потребуются два метода класса `TCanvas`:
  - `MoveTo(x,y)` — перейти к точке холста с координатами (x, y) (в пикселах);
  - `LineTo(x,y)` — нарисовать линию из предыдущей точки в точку с координатами (x, y).

  У класса `TForm` есть свойства `Width` (ширина) и `Height` (высота) (величины указываются в пикселах). Эти свойства определяют текущую ширину и высоту формы. В C++ Builder точка с координатами (0, 0) расположена в левом верхнем углу окна, а ось Y направлена вниз, значение координаты y изменяется от нуля через шаг, равный одному пикселу, до максимума. Значение координаты x изменяется слева направо

тоже через один пиксел. Если двигаться в форме, то текущие координаты точки, в которой мы будем находиться в данный момент, будут определяться величинами X, Height-Y. Чтобы нарисовать что-либо в форме, можно воспользоваться событием `OnPaint` (и не только им), которое возникает всякий раз, когда рисунок в форме изменяется. Общий вид обработчика этого события формы следующий:

```
{
int x,y;
x=y=0; /*начальные координаты графика функции y=f(x)*/
Form1->Canvas->MoveTo(0,Height); /*начальная точка
графика*/
Form1->Canvas->Pen->Color=clRed;
/*линия графика будет вычерчена пером красного цвета*/
/*Далее идут вычисления функции y=f(x), где x меняет
значения на один пиксел от нуля до максимального значения,
равного ширине формы:*/
while(x < Form1->Width && y < Form1->Height)
  {
  x++;  //следующая точка на оси X
  y=f(x);
  Form1->Canvas->LineTo(x,Height - y);
// рисуем линию в первую точку (x,y)
  }
} // конец обработчика.
```

Осталось написать конкретную программу для вычисления конкретной функции *y = f(x)*. Для этого в Unit1.cpp вставим h-файл, в котором описаны математические функции, и возьмем одну из них в качестве примера. Пусть это будет функция y = log(x). Учтем при этом, что мы работаем в пикселах (т.е. в целых величинах), а логарифмическая функция возвращает дробные числа. Поэтому дробная часть будет отбрасываться с помощью функции floor(x). Кроме того, чтобы график имел наглядность, увеличим его ординату в 50 раз. Теперь можно создать функцию, которая будет вызываться в основной программе и которая станет вычислять график логарифмической функции:

```
int f(int x)
 {
```

```
    int y;
    y=floor(50 * log(x));
    return(y);
}
```

Пример приложения, рисующего графики в форме, приведен в листинге 10.1, а нарисованный график — на рис. 10.29.

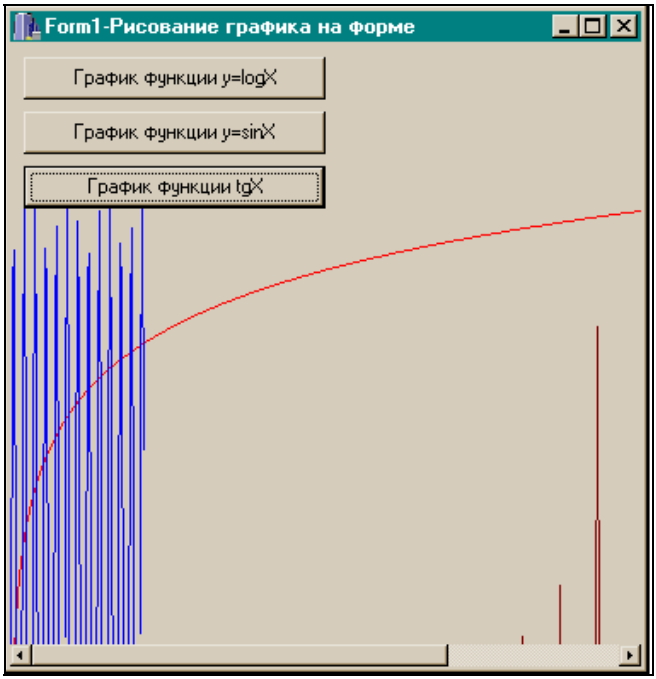


Рис. 10.29. Нарисованные в форме графики

Листинг 10.1

cpp-файл:

```
//---------------------------------------------------------------------
```

```
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include <Math.h> //для использования математических функций

//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;

// здесь мы определим функцию y=f(x);
//log(x)
int f(int x)
 {
  int y=floor(50*log(x));
/*коэффициент 50 необходим, чтобы установить ординату
побольше и график - повыше*/
  /* floor() - отбрасывается дробная часть, т. к. ордината
  должна быть целой*/
  return(y);
 }
//--sin(x)-------------------------------------------------
 int f1(int x)
  {
   int y=floor(300*sin(x));
  }
//---exp(x)
 int f2(int x)
  {
   int y=floor(tan(x));
  }

//---------------------------------------------------------
__fastcAll TForm1::TForm1(TComponent* Owner)
```

```
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcAll TForm1::Button1Click(TObject *Sender)
{
int x,y;          //текущие координаты в пикселах в форме
x=0; y=0;         //начальные координаты графика функции f(x)
Form1->Canvas->MoveTo(0,Height);
/*перейти в точку с координатами (0, Heigth-0) - начальную
точку графика*/
Form1->Canvas->Pen->Color=clRed;
//вычисление функции y=f(x) в точках x1,x2,...

while(x<Form1->Width && y<Form1->Height)
 /*абсцисса не выходит за пределы ширины формы, а ордината -
за высоту (она измеряется от левого верхнего угла вниз)*/
 {
  x++;           //следующая точка в пикселах по оси X
  y=f(x);        //эта функция определена в начале модуля cpp
  Form1->Canvas->LineTo(x,Height-y);
// рисуем линию до новой точки (x, y) из предыдущей (0,
Heigt-0)
 }
}
//---------------------------------------------------------
void __fastcAll TForm1::Button2Click(TObject *Sender)
{
int x,y;          //текущие координаты в пикселах в форме
x=0; y=0;         //начальные координаты графика функции f(x)
Form1->Canvas->MoveTo(0,Height);
/*перейти в точку с координатами (0, Heigth-0) - начальная
точка графика*/
Form1->Canvas->Pen->Color=clBlue;
//вычисление функции y=f(x) в точках x1, x2,...
```

```
while(x<Form1->Width-300 && y<Form1->Height)
{
  x++;              //следующая точка в пикселах по оси X
  y=f1(x);          //эта функция определена в начале
                    //модуля cpp
  Form1->Canvas->LineTo(x,Height-y);
}

}
//---------------------------------------------------------
void __fastcAll TForm1::Button3Click(TObject *Sender)
{
int x,y;              //текущие координаты в пикселах в форме
x=0; y=0;             //начальные координаты графика функции
f(x)
Form1->Canvas->MoveTo(0,Height);
Form1->Canvas->Pen->Color=clMaroon;
//вычисление функции y=f(x) в точках x1,x2,...
while(x<Form1->Width && y<Form1->Height)
  {
   x++;          //следующая точка в пикселах по оси X
   y=f2(x);      //эта функция определена в начале модуля .cpp
   Form1->Canvas->LineTo(x,Height-y);
  }

 }
}

//---------------------------------------------------------
```

- `Color` — это свойство задает цвет поля формы. Цвет можно выбрать из раскрывающегося списка, который появится, если нажать на кнопку в поле этого свойства. В этом списке следует установить стрелками или мышью линейку подсветки на нужный цвет и щелкнуть мышью или нажать <Enter>. Кроме того, цвет можно выбрать из палитры цветов, кото-

рая появится, если дважды щелкнуть кнопкой мыши в поле (рис. 10.30).

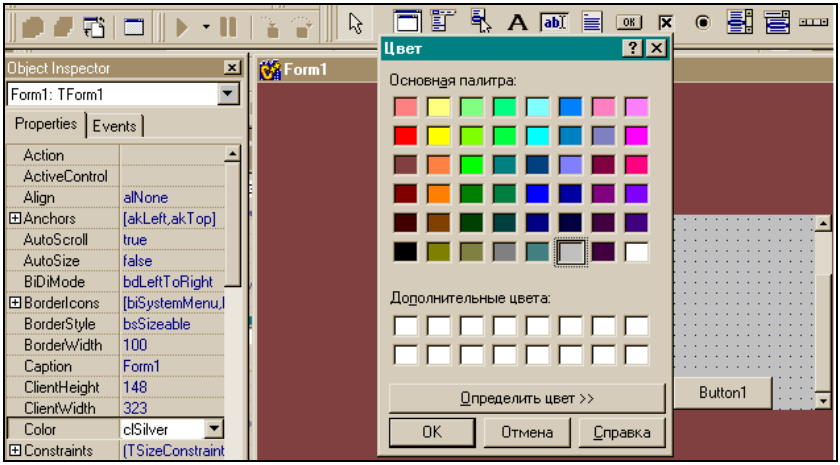


Рис. 10.30. Выбор цвета формы

- `Constraints` — здесь указываются ограничительные величины на размеры компонента (в данном случае — формы). Это свойство рекомендуется не изменять, т. к. его значения связаны со свойствами `Align` (выравнивание объекта относительно контейнера, его содержащего) и `Anchors` (анкеры), которые задают привязку компонента к родителю. Компонент поддерживает свое текущее месторасположение относительно угла родительского окна даже тогда, когда родительское окно изменяет размеры. В этом случае компонент удерживает свою позицию по отношению к тому углу, к которому он привязан. Анкеры и задают эту привязку.
- `Cursor` — в раскрывающемся списке можно выбрать форму курсора мыши. Форма курсора будет действовать над областью всего компонента. Выбор курсора аналогичен выбору цвета, но при двойном щелчке будут появляться новые формы курсора.
- `DockSite` — свойство задает способность компонента стать участком стыковки для других компонентов.

- `DragKind` — этим свойством обладают только некоторые компоненты. Оно задает способ перетаскивания компонента: если это свойство имеет значение `dkDrag`, то компонент участвует в операции перетаскивания, если — `dkDock`, то в операции стыковки. Если задать значение свойства `DragKind` равным `dkDock`, а значение свойства `DragMode` — `dmAutomatic`, то *в режиме исполнения программы* компонент приобретает способность перемещаться по форме: он становится потомком участка стыковки (в нашем случае — формы), и его можно перетаскивать мышью. По умолчанию `DragMode` = `dmManual`. Это означает, что компонент можно заставить двигаться, применяя обработчики событий нажатия мыши. Если свойство `DragKind` имеет значение `dkDock`, то с компонентом могут происходить следующие метаморфозы: он может быть захвачен участком стыковки, в этом случае компонент займет весь участок. Уже измененный в размере компонент можно вытащить из участка стыковки и двигать по форме. Если на компоненте щелкнуть, он может принимать вид обычного окна Windows с линейкой захвата и кнопкой закрытия окна на ней.

  Примечание

  Попробуем увидеть это на практике. Придайте свойству `Align` компонента `Tpanel` (панель) значение `Alleft`, чтобы панель прижалась к левому краю формы, установите свойство `DockSite` в значение `true`, чтобы панель стала стыковочным участком, поместите кнопку `TButton` в форму. Свойству `DragKind` этой кнопки придайте значение `dkDock`, а свойству `DragMod` — значение `dmAutomatic`. Откомпилируйте проект и подвигайте мышью кнопку: к началу панели, вытащите за пределы панели. Щелкните на кнопке, не создав ее обработчик. Посмотрите, как поведет себя кнопка.

- `DragMode` — см. пояснение к предыдущему свойству.
- `Enable` — задает право доступа к компоненту: значение `true` означает, что доступ разрешен, `false` — запрещен. В случае с формой значение свойства, равное `false`, приведет к блокированию формы: после компиляции ничто в ней не будет реагировать на мышь, даже закрыть форму будет невозможно.

- `Font` — задает характеристики шрифта формы. Все компоненты, расположенные в форме, унаследуют ее шрифт. Чтобы задать значение свойства `Font`, нужно два раза щелкнуть на свойстве, после чего откроется диалоговое окно выбора характеристик шрифта.
- `FormStyle` — определяет характеристики формы: является ли она одной из форм так называемого MDI-приложения. Мы сейчас говорим о создании приложений со стандартным документным интерфейсом (SDI) и поэтому свойства `FormStyle` и `DefaultMonitor` должны оставаться заданными средой по умолчанию.

Примечание

MDI-приложение формируется специальным Мастером, который позволяет создавать приложения на основе Многодокументного интерфейса (MDI). В таких приложениях вывод формы может идти на разные мониторы, для каждой формы вывод задается свойством `DefaultMonitor`.

- `Hint` — подсказка. Она появляется, как только мышь зависает над компонентом, но при условии, что значение свойства `ShowHint` (показать подсказку) установлено в `true`.
- `HorzScrollBar`, `VertScrollBar` — это составные свойства, которые позволяют задать характеристики вертикальной и горизонтальной полос прокрутки формы.
- `KeyPreview` — это свойство определяет, может ли форма получить событие с клавиатуры (`OnKeyUn`, `OnKeyPress` и т. п.) раньше, чем активный компонент в ней. Если значение свойства `KeyPreview` установлено равным `true`, то события с клавиатуры в форме возникают раньше, чем в активном компоненте (активный компонент выбирается из списка свойства `ActiveControl`). Если значение свойства `KeyPreview` установлено равным `false`, события клавиатуры возникают только в активном компоненте. Навигационные клавиши (<Tab>, клавиши со стрелками и т. д.) не действуют на свойство `KeyPreview`, потому что они не генерируют событий клавиатуры. По умолчанию свойство `KeyPreview` имеет значение `false`. Это означает, что форма и другой компонент, который

может в данный момент обрабатывать события, возникающие при вводе с клавиатуры имеют одинаковые события. Но при присвоении свойству `KeyPreview` значения `true` эти события можно перехватывать в форме, и выполнять определенные действия до того, как будут выполнены действия по этим же событиям в активном компоненте.

- `Menu` — если в форму поместить компонент `MainMenu`, то его имя попадет в это свойство, и при запуске формы главное меню будет готово к выполнению своих команд.
- `ModalResult` — это свойство используется для закрытия формы, когда она открыта в модальном режиме *(см. разд. "Методы формы" данной главы)*. По умолчанию `ModalResult` имеет значение `mrNone`. Установите `ModalResult` в любое ненулевое значение, чтобы закрыть форму. Значение, присвоенное `ModalResult`, становится возвращаемым значением функции (метода) `ShowModal()`, вызываемой, чтобы показать форму на экране. Свойство `ModalResult` имеет тип `TModalResult`. Но оно с помощью спецификатора класса памяти `typedef` представляет фактически тип `int`: `typedef int TModalResult;`. Чтобы было удобно наблюдать, как закрывается модальная форма, вместо чисел используют символические константы. Это возможно, т. к. значения, возвращаемые функцией `ShowModal()`, совпадают с теми, которые задавались в `ModalResult`. Эти константы и соответствующие им значения представлены в табл. 10.1. (Закрытие модальной формы происходит по нажатию кнопки, причем свойство `ModalResult` этой кнопки установлено в одно из этих значений.)

Таблица 10.1. *Значения свойства `ModalResult` и символические константы*

Константа	Значение	Смысл
`mrNone`	`0`	Это значение установлено по умолчанию до того, как форма закроется
`mrOk`	`idOK`	Форма закрылась по значению `OK` кнопки

Таблица 10.1 *(окончание)*

Константа	Значение	Смысл
mrCancel	idCancel	Форма закрылась по значению Cancel кнопки
mrAbort	idAbort	Форма закрылась по значению Abort кнопки
mrRetry	idRetry	Форма закрылась по значению Retry кнопки
mrIgnore	idIgnore	Форма закрылась по значению Ignore кнопки
mrYes	idYes	Форма закрылась по значению Yes кнопки
mrNo	idNo	Форма закрылась по значению No кнопки

- Name — задается имя формы.
- PopupMenu — если в форму поместить компонент PopupMenu (всплывающее меню), то его имя попадет в это свойство, и при запуске формы меню появится, если нажать правую кнопку мыши. Меню будет готово к выполнению своих команд.
- Position — определяет размер и размещение формы.
- Tag — сюда помещается некоторое целое число, которое можно потом доставать из формы во время исполнения программы.
- Visible — если это свойство имеет значение false, то форма становится невидимой при исполнении программы.

События формы

Перечень событий формы показан на рис. 10.31 и 10.32.

- OnActivate — возникает, когда форма активизируется.
- OnClick — возникает при щелчке мышью в форме.

Рис. 10.31. События формы (часть 1)

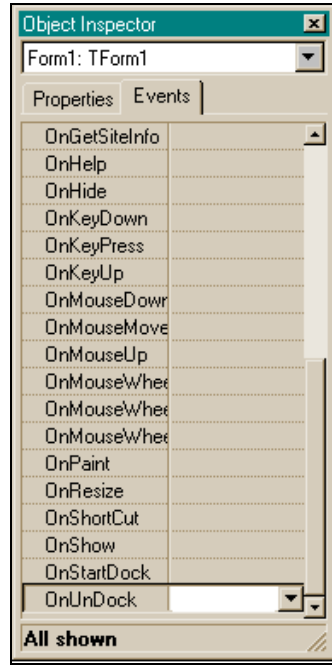


Рис. 10.32. События формы (часть 2)

- `OnClose` — возникает, когда форма закрывается.
- `OnCreate` — возникает, когда форма создается.
- `OnDeactivate` — возникает, когда форма перестает быть активной.
- `OnKeyDown` — возникает, когда пользователь нажимает некоторую клавишу на клавиатуре. Это событие может возникать в ответ на нажатие любых клавиш, включая функциональные (<F1>—<F12>) и комбинации с <Shift>, <Alt> и <Ctrl>.
- `OnKeyPress` — возникает, когда пользователь нажимает некоторую (только одну) клавишу на клавиатуре, кроме функциональных.

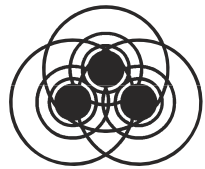
- `OnKeyUp` — возникает, когда пользователь отпускает клавишу клавиатуры, которая до этого была нажата. Это событие возникает и тогда, когда до этого были нажаты и комбинации клавиш с <Shift>, <Alt> и <Ctrl>, а также функциональные клавиши.
- `OnMouseDown` — возникает при нажатии любой кнопки мыши, а также при комбинации клавиш <Shift>, <Ctrl> и <Alt> и кнопок мыши. Обработчик этого события будет содержать координаты указателя мыши в пикселах — *X* и *Y*.
- `OnMouseMove` — возникает, когда пользователь двигает указатель мыши, пока он находится над объектом (в нашем случае — над формой). Обработчик этого события будет содержать координаты указателя мыши в пикселах — *X* и *Y*.
- `OnMouseUp` — возникает, когда пользователь отпускает клавишу мыши, которая была нажата на компоненте. Обработчик этого события будет содержать координаты указателя мыши в пикселах — *X* и *Y*.
- `OnPaint` — возникает, когда начинается прорисовка формы.
- `OnShow` — возникает, когда форма появляется на экране.

Методы формы

- `Close()` — закрывает форму. Если закрывается главная форма, приложение закрывается.
- `Hide()` — свойство `Visible` устанавливается в `false` и форма становится невидимой.
- `Print()` — форма печатается.
- `Release()` — форма разрушается и память, занятая ею, освобождается.
- `SetFocus()` — делает форму активной: свойства `Visible` и `Enabled` становятся равными `true`: форма становится видимой и доступной.

- `Show()` — показать форму: в этом случае свойство `Visible` устанавливается в `true`, и форма перемещается поверх всех форм на экране.
- `ShowModal()` — показать форму в модальном режиме. Когда форма показана в модальном режиме, приложение не может выполняться, пока форма не будет закрыта. Чтобы закрыть такую форму, надо установить ее свойство `ModalResult` в ненулевое значение.

Глава 11



Компоненты, создающие интерфейс между пользователем и приложением

В этой главе мы рассмотрим некоторые компоненты, с помощью которых разработчик приложения может создать удобный для пользователя интерфейс, позволяющий пользователю общаться с приложением и управлять им. Как известно из предыдущего материала, каждый компонент характеризуется тремя наборами данных, определяющих его функциональность: это свойства, события и методы. В данной главе мы рассмотрим не все компоненты, а компоненты первой необходимости, т. к. с течением времени разработчики среды пополняют ее все большим количеством компонентов, на описание которых потребуется не одна толстая книга. Но владея принципами работы с основными компонентами, пользователь среды Borland C++ Builder может самостоятельно осваивать новые компоненты, пользуясь справочной помощью, поставляемой со средой. Чтобы помочь такому пользователю в его дальнейшей самостоятельной работе по освоению вновь появляющихся компонентов среды, в этой главе мы сделаем упор не на подробном описании всех свойств, событий и методов рассматриваемых компонентов, а изучим лишь некоторые из них.

Компонент *TButton*

Этот компонент создает в форме, куда он помещен, элемент "кнопка", который надо нажимать щелчком мыши. Компонент

`TButton` обладает рядом свойств, определяющих его поведение. Вид его в форме показан на рис. 11.1.

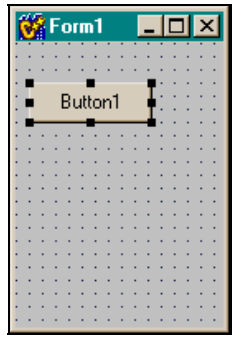


Рис. 11.1. Вид компонента `TButton`

Чтобы изучить характеристики компонента, обратимся к разделу **Help**. Щелкнем левой кнопкой мыши на компоненте, помещенном в форму, вызывая таким образом справочную помощь. Получим последовательно два диалоговых окна, показанных на рис. 11.2.

Во втором диалоговом окне, кроме описания сути компонента, в верхней его части имеются вкладки, содержащие разнообразную информацию о компоненте. Рассмотрим эти вкладки для кнопки.

- Вкладка **Hierarchy**. Ее вид показан на рис. 11.3. В ней отражена вся последовательность создания класса для кнопки из других классов. Когда мы рассматривали классы, то видели, что благодаря принципу наследственности при создании классов свойства родителей приобретаются потомками. Поэтому, глядя на иерархию, можно увидеть, какие свойства каких классов сосредоточены в кнопке. Эта же иерархия отражена (уже в деталях) на вкладке **Properties**. Ее вид также показан на рис. 11.3.
- На вкладке **Properties** видим, что собственно новых свойств у кнопки по сравнению с ее предками всего три: `Cancel`, `Default` и `ModalRezult`. Остальные унаследованы от предков (и показывается, какие свойства от каких предков попали в кнопку).

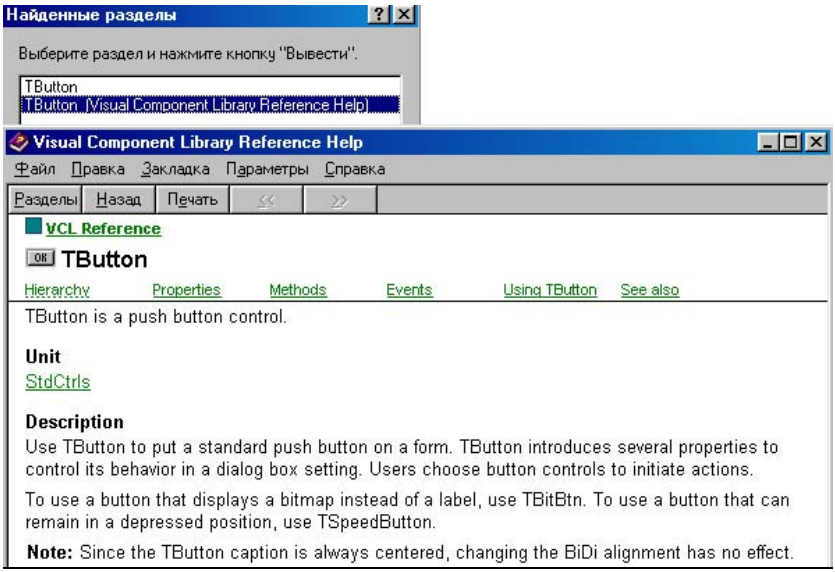


Рис. 11.2. Диалоговые окна для доступа к описанию характеристик компонента

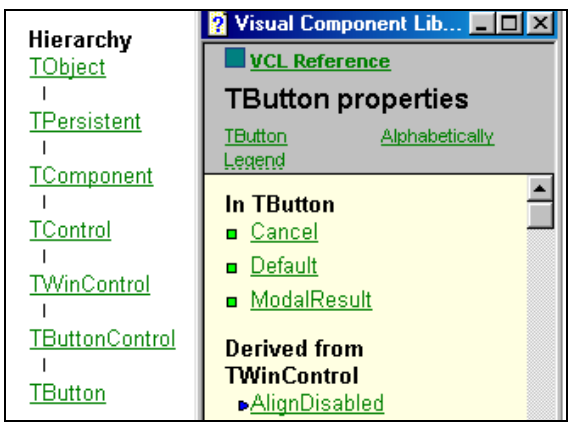


Рис. 11.3. Вкладки **Hierarchy** и **Properties**

- Вкладка **Methods**. Вид этой вкладки и вкладки **Events** показан на рис. 11.4. Структура вкладки **Methods** аналогична структу-

ре вкладки **Properties**. Мы видим, что у кнопки всего один ее собственный метод (`Click`), т. к. два остальных метода — это соответственно деструктор (первый) и конструктор (третий). Все остальные методы унаследованы от предков.

- Вкладка **Events** (События). По структуре она ничем не отличается от предыдущих двух вкладок. Здесь мы видим, что собственных (не унаследованных от предков) событий у кнопки нет.

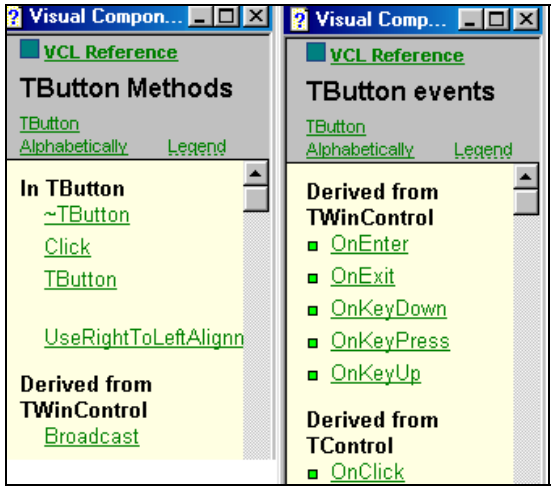


Рис. 11.4. Вкладки **Methods** и **Events**

- Вкладка **Using TButton**. Содержимое вкладки показано на рис. 11.5. Здесь даются краткие сведения, для чего и как используется кнопка. Там же присутствует и вкладка **Topic groups**, перейдя на которую можно получить список тем для более детального изучения элементов среды программирования.

Примечание

В списках, содержащихся во вкладках, слева от каждой позиции имеются пометки, суть которых можно понять, перейдя на вкладку **Legend** (ее название расположено в заголовочной части открытой вкладки). Значки определяют секцию памяти (в соответствии с

описанием класса-компонента), к которому относятся данные, например, свойство, и возможность его модификации.

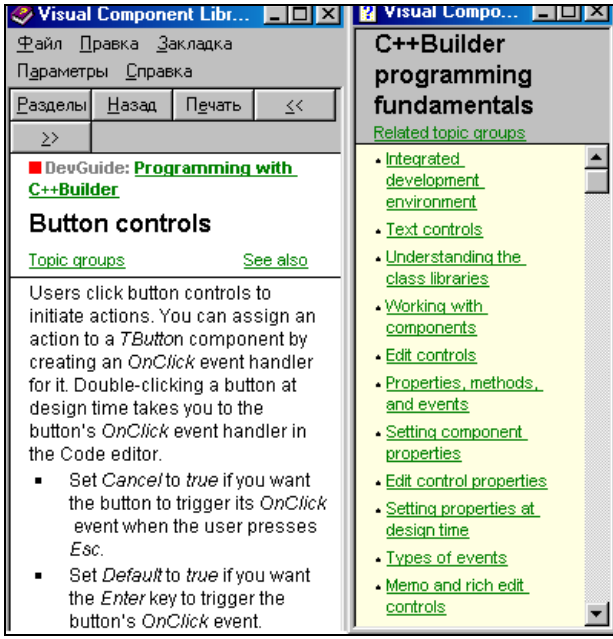


Рис. 11.5. Вкладка **Using TButton**

Некоторые свойства *TButton*

- `Cancel` — если значение свойства `true`, то, когда пользователь нажимает <Esc>, включается обработчик события `OnClick`. Так как приложение может иметь более одной кнопки **Cancel**, форма вызывает событие `OnClick` только для первой по порядку (в смысле `TabOrder`) видимой кнопки.
- `Caption` — сюда помещается название кнопки (только в одну строку).
- `ModalResult` — определяет, как кнопка закрывает модальную форму, на которой она находится. Установка этого компонента — легкий путь к закрытию модальной формы. Когда кнопку нажимают, свойство `ModalResult` формы, на которой

находится кнопка, устанавливается в то значение, которое имеет это свойство в кнопке. Поэтому перед закрытием модальной формы с помощью кнопки нет необходимости предварительно присваивать свойству `ModalResult` формы ненулевое значение: достаточно выбрать такое значение в свойстве `ModalResult` кнопки, и оно передастся одноименному свойству формы, в которой расположена эта кнопка. Поэтому достаточно выполнить `Formi->Close()`, и модальная форма закроется, как ей положено.

- `PopupMenu` — это свойство обеспечивает кнопке дополнительные возможности. Если в форму поместить всплывающее меню, то его имя можно включить в свойство `PopupMenu`. Тогда в режиме исполнения достаточно щелкнуть на кнопке правой кнопкой мыши (после щелчка левой кнопкой мыши будет запускаться обработчик `TButton`), чтобы рядом с кнопкой появилось всплывающее меню, команды которого можно выполнять.

Некоторые события *TButton*

Большинство событий аналогичны одноименным событиям формы:

- `OnClick` — возникает, когда на кнопке щелкают мышью;
- `OnEnter` — возникает, когда кнопка получает фокус ввода, т. е. становится активной: ее можно нажимать;
- `OnExit` — возникает, когда кнопка теряет фокус ввода.

Некоторые методы *TButton*

- `Click()` — имитирует нажатие кнопки. Выполнение этой функции назначает свойству `ModalResult` формы значение свойства `ModalResult` кнопки и вызывает событие `OnClick`.
- `SetFocus()` — делает кнопку активной: ее можно нажимать.

Как сделать вывод текста в поле кнопки многострочным

Для этого надо воспользоваться функциями Windows API: получить текущий стиль стандартной кнопки Windows (`GetWindowLong()`),

установить текущий стиль стандартной кнопки Windows (`SetWindowLong()`), а потом добавить к нему стиль `BS_MULTILINE`.

Пример, приведенный в листинге 11.1, демонстрирует, как сделать вывод текста в поле кнопки многострочным.

Листинг 11.1

```
void __fastcAll TForm1::Button1Click(TObject *Sender)
{
//возвращает текущий стиль кнопки
long NewStyle=GetWindowLong(Button1->Handle,GWL_STYLE);
//К существующему стилю добавляется многострочный стиль
NewStyle|= BS_MULTILINE;
//установка нового стиля
SetWindowLong(Button1->Handle,GWL_STYLE,NewStyle);
Button1->Caption="";
Button1->Caption="Иванов   Петров   Сидоров   Пахомов";
}
   //---------------------------------------------------------
```

Вид преобразованной кнопки в форме показан на рис. 11.6.

Рис. 11.6. Вид кнопки с многострочным названием

о он н *TPanel*

Панель — это компонент, который, как и форма, является контейнером, в который помещаются другие компоненты. Панели обеспечивают общее (родовое) поведение, введенное в классе-родителе `TCustomPanel`.

`TCustomPanel` — это базовый класс для всех панельных компонентов.

Он используется в качестве базового класса для определения объектов, которые включают в себя другие объекты. Панельные компоненты могут содержать в себе другие компоненты, объединяя их в единое целое. При перемещении панели такие компоненты перемещаются вместе с ней. Когда панель выравнивается в форме с помощью свойства `Align`, она сохраняет те же относительные позиции по отношению к форме, даже если форма перерисовывается. Например, панель может быть выровнена так, что всегда останется в вершине формы, даже когда пользователь меняет фигуру и размер формы. Это свойство панели делает ее полезной для расположения на ней компонентов, которые действуют как линейки инструментов или линейки состояний.

Некоторые свойства *TPanel*

У панели имеются свойства, определяющие ее обрамление:

- `BevelInner` — определяет стиль внутренней кромки. Если свойство имеет значение `bvLowred`, то внутренняя кромка будет опущена вниз, если — `bvNone`, то внутренней кромки у панели не будет, а если значение этого свойства — `bvRaised`, то внутренняя кромка панели будет поднята;
- `BevelOuter` — определяет стиль внешней кромки. Может принимать те же значения, что и `BevelInner`;
- `BevelWidth` — ширина кромок в пикселах;
- `BorderWidth` — расстояние в пикселах между внутренней и внешней кромками. На рис. 11.7 представлен вид панели внутренняя и внешняя кромки которой подняты, а расстояние между ними равно 5 пикселам;
- `BorderStyle` — стиль внешнего обвода. Обвод одиночной линией — `bsSingle`, нет внешнего обвода — `bsNone`;

- `Align` — размещает панель в форме в соответствии со значениями этого свойства: `alTop` — привязывает панель к верхней кромке формы, `alBottom` — к нижней, `alLeft` — к левой и `alRight` — к правой, `alNone` — панель не привязана ни к какой кромке формы, и ее можно двигать мышью, `alClient` — панель принимает размеры формы.
- `Alignment` — определяет, как выравнивается свойство `Caption`, в котором задается название панели, относительно самой панели: по центру, по левому краю или по правому краю. Например, если использовать панель как линейку состояния для вывода подсказки (подсказку надо помещать в свойство **Caption** панели), можно присвоить свойству `Alignment` значение `taLeftJustify` и текст подсказки разместится с левой стороны панели.

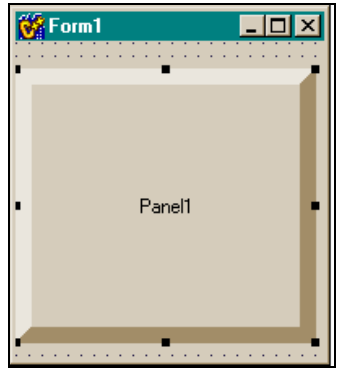


Рис. 11.7. Вид панели с кромками

Некоторые события *TPanel*

Из событий панели отметим следующие:

- `OnEnter` — возникает, когда панель становится активной — получает, как говорят, фокус ввода;
- `OnExit` — когда панель теряет фокус ввода — перестает быть активной;
- `OnClick` — когда на панели происходит щелчок мышью.

Во всех этих ситуациях управление в приложении передается на обработчик соответствующего события, и вы можете написать в обработчике необходимые вам команды, отражающие реакцию на произошедшее событие.

Методы *TPanel*

Собственных методов у компонента нет, а только унаследованные.

Компонент *TLabel*

Этот компонент выводит в форму текст, который пользователь в режиме исполнения приложения не может редактировать. Этот текст может использоваться как метка к другому компоненту и может устанавливать фокус этого компонента, когда пользователь нажимает "горячую" клавишу на клавиатуре.

Некоторые свойства *TLabel*

- `Alignment` — задает способ расположения (выравнивания) текста, записываемого в поле свойства `Caption`: будет ли текст выравниваться по левой или правой границе поля, или же — по центру поля.
- `FocusControl` — это свойство имеет раскрывающийся список, куда попадают компоненты, которые могут быть связаны с меткой и которые могут получать от нее фокус ввода (например, `TEdit`, `TButton` и др.). Выбрав из списка один из таких компонентов, который должна помечать метка, мы в ее тексте, записанном в свойстве `Caption`, в некотором месте указываем символ амперсанд (`&`). Символ, который следует за амперсандом, и станет "горячей" клавишей, когда приложение начнет исполняться. Если в момент исполнения приложения нажать "горячую" клавишу, то связанный с меткой по свойству `FocusControl` компонент станет активным (но только при условии, что свойство `ShowAccelChar` имеет значение `true`).
- `Layout` — размещение текста, заданного в свойстве `Caption`, в поле метки. Из раскрывающегося окна можно выбрать, как будет размещен текст в метке: в верхней части ее поля, в цен-

тре или в нижней части поля (метку можно растягивать за маркеры по ее краям, когда она активна).

- `Transparent` — если некоторый компонент будет расположен под меткой, то он может быть невидим. Чтобы этого не происходило, надо сделать метку прозрачной (транспарентной), т. е. установить это свойство в `true`.
- `WordWrap` — если это свойство равно `true`, а свойство `AutoSize` — `false`, то все слова текста в свойстве `Caption` станут располагаться в разных строках (так вводится многострочный текст в метках).

События *TLabel*

У компонента собственных событий нет: все унаследованы от предков.

Компонент *TEdit*

Этот компонент задает в форме однострочное редактируемое поле: через этот компонент вводят и выводят строчные данные.

Некоторые свойства *TEdit*

Собственных свойств у компонента нет, а унаследованные свойства следующие:

- `AutoSelect` — определяет, будет ли автоматически отмечен весь текст в поле этого компонента, когда компонент получает фокус ввода (значение `true`), или курсор ввода остановится в начале текста (значение `false`);
- `BorderStyle` — определяет, имеет ли компонент окантовку в виде одиночной линии или не имеет окантовки вовсе. Это видно в режиме проектирования при выборе того или иного значения;
- `CharCase` — задает регистр, в котором вводится текст в поле компонента. Значения свойства могут быть следующие:
  - `EcLowerCase` — текст преобразуется в символы нижнего регистра;

- `EcNormal` — в тексте присутствуют символы обоих регистров;
- `EcUpperCase` — текст преобразуется в символы верхнего регистра;

- `HideSelection` — задает, остается ли визуальная индикация выделенного текста, когда фокус ввода перемещается на другой компонент (`true` — выделенный текст не меняет подсветки, `false` — подсветка исчезает при выделении другого компонента);
- `PasswordChar` — если мы хотим, чтобы вводимые в поле `TEdit` символы не высвечивались, а заменялись неким другим символом, как это происходит при вводе пароля, в это свойство надо внести значения таких символов. Например, если записать: `Edit1->PasswordChar='*';`, то вместо вводимых в поле `TEdit` символов высветятся звездочки;
- `ReadOnly` — определяет, может ли пользователь менять текст в поле компонента: при значении этого свойства `true` — не может, `false` — может;
- `Text` — здесь задается текст, который мы видим в поле компонента. Если необходимо вывести текст через компонент `TEdit`, то текст надо предварительно записать, воспользовавшись свойством `Text`. Если же текст надо ввести через `TEdit`, то в поле компонента следует ввести текст, который попадет в свойство `Text`, а затем извлечь введенный текст из этого свойства.

События *TEdit*

Собственных событий у компонента нет. Но интересны некоторые унаследованные, такие как, например, `OnKeyDown`: оно возникает, когда пользователь приложения нажимает какую-либо клавишу на клавиатуре, когда сам компонент имеет фокус ввода. С помощью этого события можно отслеживать ввод данных через компонент. Например, вам надо, чтобы введенная через компонент строка символов уходила на обработку после нажатия клавиши <Enter>. Тогда в обработчик события, в который среда вас отсылает всякий раз, когда вы нажимаете любую клавишу при вводе, вы ставите проверку на нажатие клавиши <Enter> и при ее появлении прекращаете ввод. Детали можно посмотреть в

разделе **Help** этого события, нажав на элемент **Virtual Key codes** и посмотрев пример во вкладке **Example**.

Некоторые методы *TEdit*

Своих методов у компонента нет: только унаследованные.

- `Clear()` — очищает поле компонента: удаляет весь текст.
- `SelectAll()` — выделяет весь текст в поле компонента.
- `SetFocus()` — делает компонент активным: в его поле можно набирать и выводить текст.
- `GetTextLen()` — возвращает длину текста в поле компонента.
- `Hide()` — делает компонент невидимым.
- `Show()` — делает компонент видимым.

Компонент *TMainMenu*

Этот компонент создает главное меню приложения, с помощью которого управляют всей работой приложения и его частей. Разные части приложения запускаются на выполнение отдельными командами, собранными в эту структуру. Выход из приложения тоже происходит через меню. Структуру меню определяет заказчик приложения и его исполнитель. Меню формируется в форме после того, как его значок перенесен из палитры компонентов в форму. С этой формой меню будет связано через свойство формы `Menu`, в окне которого и появляется имя компонента `TMainMenu`. Когда меню сформировано, то после запуска приложения на выполнение в верхней левой части формы будет расположена строка, содержащая главные опции этого меню. Главные опции могут распадаться на более детальные команды (если таковые заданы), располагающиеся на этот раз уже в столбик: сверху вниз.

Для формирования опций меню надо воспользоваться Дизайнером меню. Он открывается одним из следующих способов: через команду **MenuDesigner** в контекстном меню компонента, двойным щелчком на компоненте, нажатием кнопки с многоточием в свойстве `Items` компонента. Во всех случаях открывается окно Дизайнера меню, который позволяет создавать опции меню. В окне Дизайнера меню в его левом верхнем углу появляется

синее поле, в которое должно быть помещено название первой опции. Чтобы поместить название в это поле, надо в открывшемся для этой опции окне Инспектора объекта в его свойстве `Caption` набрать необходимый текст и нажать на <Enter>. Тогда набранный в `Caption` текст появится в синем поле, а рядом с этим полем, справа от него, появится новое, но уже серое поле: заготовка для следующей опции меню.

Следует отметить, что каждая опция меню представляет собой новый объект — экземпляр класса. А раз так, то для экземпляра класса тут же, чтобы показать его свойства и события, появляется Инспектор объекта, в котором мы и задаем название опции. Если данная опция — последняя в иерархии опций (т. е. является исполнительной), то, воспользовавшись наличием Инспектора объекта, перейдем на вкладку **Events** и щелкнем мышью дважды в поле события `OnClick`. Откроется обработчик этого события, в который мы должны вписать те действия, которые будут выполняться, когда мы выберем данную опцию меню и щелкнем на ней. Теперь можем продолжить формировать следующую опцию горизонтальной строки на основе серого поля, расположенного справа от первой опции. На нем надо щелкнуть мышью, и оно станет активным: изменит свой цвет на синий. Тут же появится его Инспектор объекта, в свойство `Caption` которого мы впишем название этой опции и нажмем <Enter>.

В поле формируемой опции появится ее название, а рядом справа — новая (серого цвета) опция. Если новая сформированная опция — исполнительная, то с ней поступаем, как и с первой: открываем вкладку событий, выбираем событие `OnClick`. Создаем его обработчик и т. д. Остальные горизонтальные опции формируются аналогичным образом.

На рис. 11.8 приведены разные состояния окна Дизайнера меню: созданы две опции главного меню и каждая из них является исполнительной (у нее нет подменю и при нажатии на нее вызывается обработчик события `OnClick`, в который мы можем записывать действия приложения в ответ на нажатие этой опции).

Теперь посмотрим, как формируются команды подменю. Для этого достаточно щелкнуть мышью в поле опции главного меню: ниже нее появится пустая опция со своим Инспектором объекта. С ней надо поступать точно так, как мы поступали с опциями первой (главной) строки.

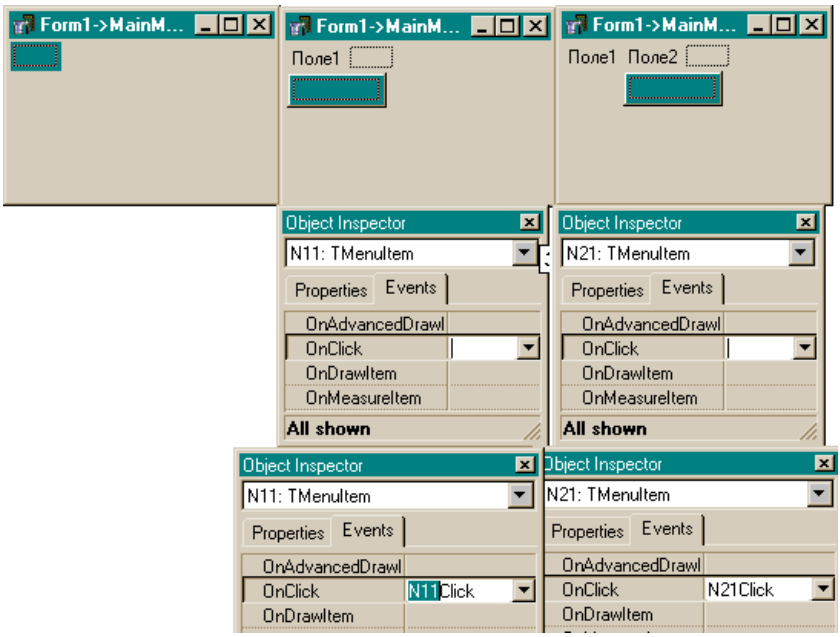


Рис. 11.8. Состояния окна Дизайнера меню при создании опций меню

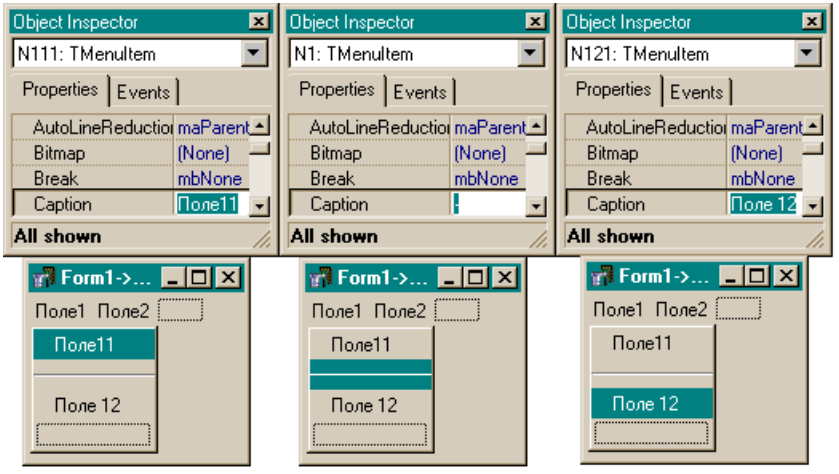


Рис. 11.9. Вид подменю

Здесь следует заметить, что можно улучшить внешний вид ниспадающего подменю, если ввести разделительные линии между опциями. Для этого при задании названия для очередной пустой опции надо в свойстве `Caption` набрать символ "минус". Тогда эта пустая опция превратится в разделительную черту между именованными опциями. Подменю для главного поля (**Поле1**), подменю, состоящее из двух опций **Поле11** и **Поле12**, и соответствующие Инспекторы объекта показаны на рис. 11.9.

Некоторые свойства *TMainMenu*

- `Images` — это свойство обеспечивает появление небольших изображений — значков — слева от названий команд и подменю. Чтобы значок появился рядом с названием опции, надо в форму поместить компонент `TImageList` (вкладка **Win32** палитры компонентов), с помощью которого можно сформировать список изображений. После помещения этого компонента в форму его имя появится в свойстве `Images` главного меню. Потом, когда будут формироваться опции (меню и подменю), у каждой из них в ее Инспекторе объекта надо установить свойство `ImageIndex`. Если щелкнуть на этом свойстве мышью, то появится раскрывающийся список значков, выбранных с помощью компонента `TImageList`. Когда вы выберете нужное изображение, оно появится слева от названия опции, для которой вы сформировали свойство `ImageIndex`.
- `Items` — в поле этого свойства есть кнопка с многоточием, с помощью которой открывается Дизайнер меню. Это свойство само является указателем на класс `TMenuItem`: (`TmenuItem * Items`) и поэтому может быть использовано для обращения к свойствам и методам этого класса. Кроме того, оно является массивом, хранящим все опции меню. Например, количество строк меню с именем `MainMenu1` можно подсчитать так: `MainMenu1->Items->Count`, где `Count` — это свойство `TMenuItem`, а к опции, которая является *i*-м элементом меню, можно обратиться: `MainMenu1->Items[i]`. Следует отметить, что все это относится только к подменю. Для команд подменю с именем `N11` (`N11` — это имя объекта — подменю главного меню) подсчитать количество опций можно сле-

дующим образом: `N11->Count`. Действительно, `N11` — это указатель на класс `TMenuItem`, а `Count` — это свойство этого класса, в котором определяется количество элементов.

Некоторые свойства опций *TMainMenu*

Рассмотрим некоторые из этих свойств.

- `Bitmap` — позволяет выбрать значок в открывающемся диалоговом окне. После этого значок появится слева от названия опции.
- `Checked` — с помощью этого свойства можно контролировать, была ли выбрана данная команда меню: в обработчике события `OnClick` этой опции надо присвоить свойству `Checked` значение `true`. При условии, что свойство `RadioItem` этой опции имеет значение `false`, слева от названия опции появится галочка. При щелчке мышью на такой опции рядом с ее названием появится галочка. Если же присвоить свойству `RadioItem` значение `true`, а свойству `Checked` — `false`, то опция не будет отмечена. Но при `RadioItem` = `true` и `Checked` = = `true` будет помечена всегда только одна опция из всех, у которых свойство `GroupIndex` имеет одинаковое значение (это сигнал, что такие опции относятся к одной группе) или — все, если присвоить свойству `GroupIndex` каждой опции свое значение. Пометка будет в виде жирной точки слева от названия опции.
- `ShortCut` — в раскрывающемся списке этого свойства надо выбрать комбинацию клавиш, которая в режиме исполнения приложения заменит нажатие мышью на опцию. При этом действие комбинации клавиш будет эффективным только при условии, что меню не будет открыто. Выбранная комбинация клавиш появится справа от названия опции. Причем выбор заменяющей комбинации клавиш имеет место только для команд подменю, но не для команд главного меню (рис. 11.10).

Примечание

Если к определенным командам не все работники имеют доступ, то можно закрыть доступ к этим опциям меню или вообще сделать

их невидимыми, воспользовавшись свойствами `Visible` и `Enabled` опций. Чтобы блокировать выполнение опции, надо свойству `Enabled` придать значение `false` (например, записать: `N111->Enabled=false;`), а чтобы скрыть опцию, следует значение свойства `Visible` опции сделать равным `false`.

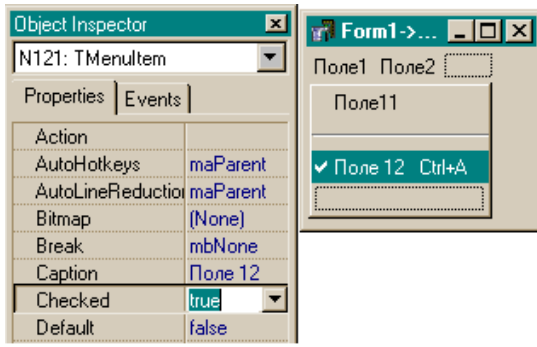


Рис. 11.10. Ввод комбинации клавиш, заменяющих нажатие опции мышью

События *TMainMenu*

- `OnChange` — возникает, когда изменяется содержимое меню (например, значение какого-то свойства).

Компонент *TPopupMenu*

Этот компонент может быть связан с любым другим компонентом (формой, кнопкой и т. д.), у которого имеется свойство `PopupMenu` (всплывающее меню). Когда компонент `TPopupMenu` помещается в форму, его имя будет видно в любом из компонентов формы, у которого есть свойство `PopupMenu`. Это обычное меню, в котором пользователь определяет порядок действий при активизации компонента, с которым данное меню связано. Если меню связано с формой, то оно появляется, когда пользователь нажимает в активной форме правую кнопку мыши. Меню появляется в том месте, где находится указатель мыши. Но точку

его появления в форме можно зафиксировать, если использовать метод `Popup()`, присущий компоненту `TPopupMenu`.

Используем свойства формы `Top` и `Left`. Если нам надо поместить меню в точку с координатами в пикселах (*m*, *n*), то следует выполнить команду:

```
Form1->PopupMenu1->Popup(Form1->Left+m,Form1->Top+n);
```

Например, чтобы поместить меню в левый верхний угол формы и чтобы оно не попадало на линейку заголовка формы, надо выполнить команду

```
Form1->PopupMenu1->Popup(Form1->Left,Form1->Top+25);
```

Задание опций выпадающего меню аналогично заданию команд подменю в главном меню. Чтобы задать вложенную опцию, надо открыть контекстное меню той опции, которую вы хотите поделить на вложенные опции, и в этом меню выполнить команду **CreateSubmenu**.

Формирование опций меню аналогично формированию соответствующих опций главного меню.

Свойства *TPopupMenu*

Если посмотреть на иерархию этого компонента, то можно видеть, что одним из его предков является компонент `TMainMenu` и поэтому `TPopupMenu` их наследует. Однако есть и новые:

- `Alignment` — задает расположение меню, когда оно появляется при нажатии на компоненте `TButton` правой кнопкой мыши (по центру компонента, слева или справа от него (на практике оказывается, что придание свойству значения `paRight` выводит отображение меню слева от компонента!));
- `AutoPopup` — если установить `AutoPopup` в `false`, то появление всплывающего меню надо будет устанавливать программно с помощью метода `Popup()`, если же `AutoPopup` имеет значение `true`, то меню появляется, автоматически привязанное к соответствующему компоненту;
- `MenuAnimation` — задает анимационный эффект появления меню на экране: либо его опции последовательно возника-

ют, появляясь сверху вниз, либо — слева направо и т. д. Ясно, что это будет красиво выглядеть для больших по объему меню;

- `TrackButton` — задает кнопку мыши, при нажатии которой меню появляется. По умолчанию — это правая кнопка.

События и методы *TPopupMenu*

Событие `OnPopup` возникает перед появлением меню на экране. Его полезно использовать для установки свойств `Checked`, `Enabled` или `Visible` конкретных опций меню.

События опций ввиду наследования совпадают с соответствующими событиями для опций главного меню. Метод `Popup()` выводит `PopupMenu` на экран.

Компонент *TMemo*

С помощью этого компонента в форме задается многострочное редактируемое текстовое поле.

`TMemo` — это массив пронумерованных текстовых строк типа `TStrings`. Этот массив находится в свойстве `Lines`, и чтобы обратиться к его *i*-й строке, следует писать:

```
Memo1->Lines->Strings[i];
```

Чтобы ввести строки в массив, надо воспользоваться Редактором строк, который открывается нажатием кнопки с многоточием, расположенной в свойстве `Lines`. `Lines` — это указатель на класс `TStrings`, поэтому с помощью `Lines` можно обращаться ко всем членам этого класса. Например, у этого класса есть свойство `Count`, в котором хранится количество строк в массиве. Если мы запишем `Memo1->Lines->Count`, то это и будет количество строк в `Memo`-поле.

Некоторые свойства *TMemo*

- `Lines` — открывает Редактор текстовых строк. В появившемся окне Редактора следует набрать текст таким же образом, как

в обычном текстовом редакторе Windows, и нажать **OK**. При этом набранный текст попадет в поле Memo. С помощью свойства `Lines` можно обращаться к строкам Memo-поля, как было показано выше. Можно также определять длину *i*-й строки:

```
Memo1->Lines->Strings[i].Length();
```

И все благодаря тому, что `Lines` — это указатель на класс `TStrings`, который, в свою очередь, является массивом экземпляров класса `AnsiString`:

```
AnsiString TStrings[int n];
```

То есть конкретная строка `Strings[i]` — это экземпляр класса `AnsiString`, и поэтому мы можем пользоваться методами этого класса, в частности, методом определения длины строки `Length()`. Но поскольку мы имеем дело не с указателем на класс, а с экземпляром этого класса, то при обращении к методу этого класса мы ставим после имени класса точку, а не стрелку вправо.

В классе `TStrings` есть методы `LoadFromFile("имя файла")` и `SaveToFile("имя файла")`, которые позволяют загружать Memo-поле из текстового файла или сохранять его в текстовом файле. Например, чтобы загрузить строки из текстового файла, нужно выполнить

```
Memo1->Lines->LoadFromFile("a.txt");
```

А чтобы очистить `Memo`-поле:

```
Memo1->Lines->Clear();
```

- `MaxLength` — здесь задается максимальная длина строки текста, вводимой в Memo-поле. Если значение этого свойства — ноль (по умолчанию так и есть), это означает, что длина строки ограничивается возможностями операционной системы.
- `ScrollBars` — здесь задаются полосы прокрутки окна Memo-поля.
- `WantTabs` — если это свойство равно `false` (принято по умолчанию), то при нажатии на клавишу табуляции табулятор обходит компоненты формы в порядке значений их свой-

ства `TabOrder`. Если же `WantTabs` имеет значение `true`, то дойдя до `TMemo`, табуляция захватывается этим компонентом и выполняется только внутри этого компонента.

События и методы *TMemo*

Собственным событием компонента является `OnChange`. Возникает при попытке изменить содержимое компонента. Эту ситуацию и можно обработать в обработчике события этого компонента, которому передается управление при попытке внести изменение в текст. Интересны методы:

- `CopyToClipBoard()` — копирование содержимого Memo-поля в буфер Windows без его уничтожения в самом поле;
- `CutToClipBoard()` — то же, но с уничтожением;
- `PastFromClipBoard()` — вставка в Memo-поле содержимого буфера.

О методах `SaveToFile()` и `LoadFromFile()` сказано выше. Это методы класса `TStrings`, указателем на который является свойство `Lines`. То есть по правилам использования указателей можно писать:

```
Lines->(элемент класса, на который указывает Lines,
например, SaveToFile(имя файла))
```

Задача регистрации пользователя в приложении

Когда разработанное приложение сдается в эксплуатацию, первое, на что обращает внимание заказчик, это то, как осуществлена защита приложения от постороннего вмешательства. В этой главе на основе уже изученных нами программных средств мы рассмотрим простейшую задачу защиты приложения от постороннего вмешательства — задачу регистрации пользователя в приложении. Здесь имеется в виду тот факт, что для входа в приложение пользователь должен, как и во всех порядочных системах, зарегистрироваться, т. е. набрать свое имя и пароль. Если все, что он набрал, верно, доступ к приложению открыт.

В качестве примера мы рассмотрим приложение, в котором используются: кнопки, меню, вызов форм в обычном и модальном режимах, ввод-вывод через компонент `TEdit`.

Регистрация пользователя

Форма с компонентами, реализующими регистрацию, приведена на рис. 11.11.

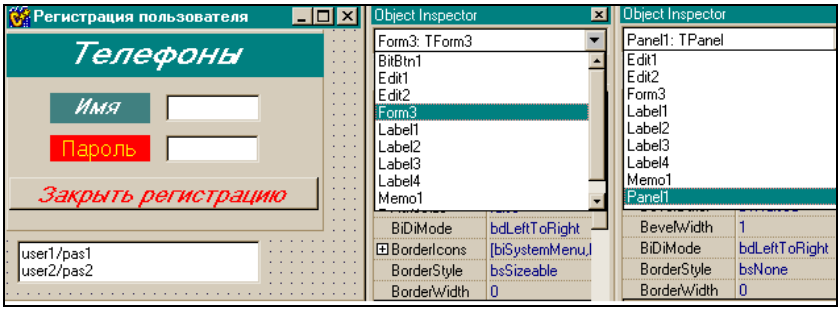


Рис. 11.11. Форма приложения для регистрации пользователя

В окне Инспектора объекта, открытом для формы, виден выпадающий список ее компонентов:

- панель, на которой расположены компоненты `TEdit` (через них пойдет ввод имени и пароля пользователя — поля **Имя** и **Пароль**);
- кнопка **Закрыть регистрацию** (задача завершает свою работу вместе с приложением);
- экземпляр компонента `TMemo` — `Memo1`, в котором заданы имя пользователя и пароль, с которыми надо сверять значения, введенные пользователем. Естественно, что `Memo1` на этапе разработки невидим.

Форма с этими компонентами имеет статус главной. Когда пользователь запускает весь проект, сначала вызывается форма с компонентами регистрации пользователя, пользователь вводит свое регистрационное имя и пароль и, если все набрано верно, вызывается первая форма собственно приложения.

Текст программы регистрации приведен в листинге 11.2.

Листинг 11.2

```
//cpp-модуль
//---------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit3.h"
#include "Unit1.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm3 *Form3;
//---------------------------------------------------------
__fastcall TForm3::TForm3(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------

void __fastcall TForm3::Edit1KeyDown(TObject *Sender, WORD
&Key,
      TShiftState Shift)
{
  if (Key == VK_RETURN)       //Завершение ввода в Edit1
                              //по нажатии <Enter>
        {
          AnsiString s=Edit1->Text;
//поиск строки в Memo с данным именем пользователя
          int j=0;
          for(int i=0;i<Memo1->Lines->Count; i++)
           {
```

```
            w=Memo1->Lines->Strings[i];
            pos=w.AnsiPos("/");
            us=w.SubString(1,pos-1);
            if(s!=us)
             continue;
            else
             { j++; break;}
           } //for
            if(!j)
             {
              ShowMessage("Не найдено имя пользователя.
                          Повторите ввод");
              Edit1->Text="";
              goto mm;
             }
 //здесь имя пользователя обнаружено в Memo, надо читать пароль
 //он находится в переменной w
          Edit1->Text="";
          Edit2->Text="";
          FocusControl((TWinControl*) Edit2);
      mm: ; //выход на ввод
        } //if
  } //void
//---------------------------------------------------------------


void __fastcall TForm3::Edit2KeyDown(TObject *Sender, WORD
&Key, TShiftState Shift)
{
    if (Key == VK_RETURN)    /*Завершение ввода в Edit2 по
                               нажатии <Enter>*/
       {
          AnsiString s=Edit2->Text;
//поиск строки в Memo с данным именем пользователя
          int j=0;
```

```
          pas=w.SubString(pos+1,100);
             if(s!=pas)
              {
               ShowMessage("Не верен пароль. Повторите ввод");
               Edit2->Text="";
               goto mm;
              }
 //здесь пароль совпал, надо вызывать форму 1
          Edit1->Text="";
          Edit2->Text="";
          Form1->Show();
       }//if
       mm: ;
}
//---------------------------------------------------------
void __fastcall TForm3::BitBtn1Click(TObject *Sender)
{
 Form3->Close();
}
//---------------------------------------------------------
```

//h-модуль

```
//---------------------------------------------------------
#ifndef Unit3H
#define Unit3H
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <ComCtrls.hpp>
#include <ExtCtrls.hpp>
#include <Buttons.hpp>
//---------------------------------------------------------
class TForm3 : public TForm
```

```
{
__published:  // IDE-managed Components
        TPanel *Panel1;
        TLabel *Label1;
        TLabel *Label3;
        TLabel *Label2;
        TLabel *Label4;
        TEdit *Edit1;
        TEdit *Edit2;
        TMemo *Memo1;
        TBitBtn *BitBtn1;
       void __fastcall Edit1KeyDown(TObject *Sender,
       WORD &Key,
          TShiftState Shift);
       void __fastcall Edit2KeyDown(TObject *Sender,
       WORD &Key,
          TShiftState Shift);
        void __fastcall BitBtn1Click(TObject *Sender);
private:   // User declarations
public:      // User declarations
  AnsiString us,pas,w; //Глобальные переменные
  int pos;
        __fastcall TForm3(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm3 *Form3;
//---------------------------------------------------------
#endif
```

Пояснения даны по тексту программы. Следует воспользоваться тем, что имя обработчика состоит из имени компонента и имени события этого компонента. Например, `BitBtn1Click`: это обработчик нажатия кнопки **BitBtn1**. Кроме того, ввод в `Edit1` и `Edit2` заканчивается нажатием <Enter>. Для этого создается обработчик события `OnKeyDown`, на вход которого в переменную

`Key` поступает значение нажатой клавиши. Если это любая клавиша, кроме <Enter>, то происходит выход из обработчика, т. к. не завершен ввод в `Edit1` и `Edit2` (`VK_RETURN (Virtual Key for Return)`).

> **Примечание**
>
> Виртуальное значение <Enter>, наряду со значениями многих специальных клавиш клавиатуры и кнопок мыши, определено в модуле Windows.

Если же сравнение с виртуальным ключом произошло, следовательно ввод в `Edit1`, `Edit2` завершен и надо приступать к сравнению с данными, расположенными в Memo-поле. Отметим также, что свойство `Text` компонентов `Edit1` и `Edit2` автоматически преобразуется при присвоении в класс `AnsiString`, методами которого для работы со строковыми данными мы здесь пользовались.

Приложение

Приложение состоит из двух форм. Вид формы 1 приведен на рис. 11.12. Вид главного меню — на рис. 11.13. В первой форме действия выполняются сначала кнопками, потом те же действия продублированы с помощью главного меню формы.

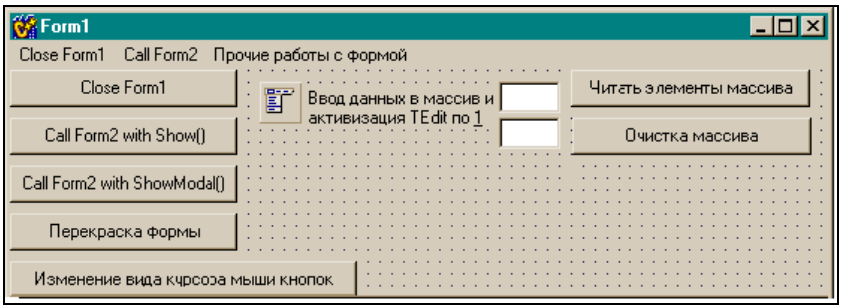


Рис. 11.12. Вид первой формы приложения, вызываемой после успешной регистрации пользователя

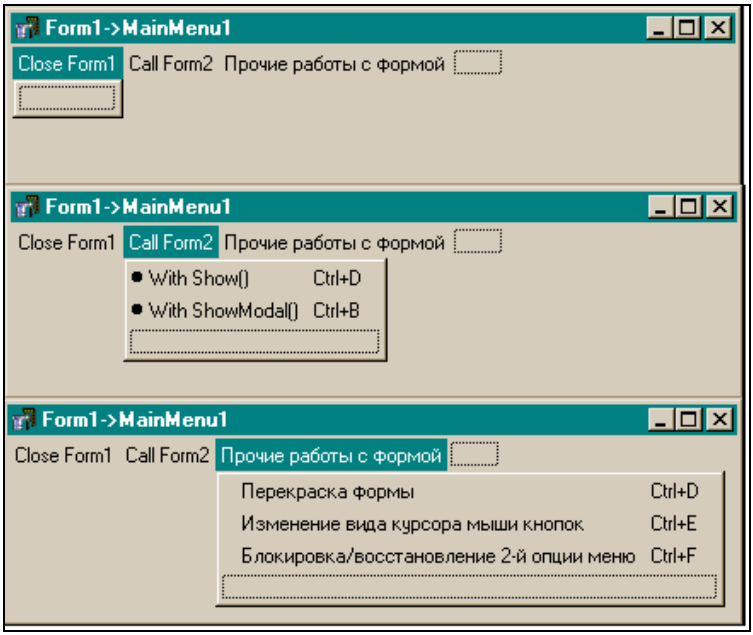


Рис. 11.13. Вид главного меню формы

Комбинацию клавиш для опции меню можно назначить в свойстве этой опции `ShortCut`. Поскольку опция выступает как отдельный объект, то имеет свой Инспектор объекта, в котором отражены ее свойства, в том числе и `ShortCut`. В этом свойстве есть раскрывающийся список, в котором и указаны все комбинации клавиш. Достаточно выбрать любую комбинацию мышью или с помощью стрелок клавиатуры. Заметим, что комбинация клавиш срабатывает только тогда, когда меню "закрыто".

Свойства меню `Checked`, `GroupIndex` и `RadioItem` предназначены для того, чтобы контролировать выбор соответствующих команд меню. Если в обработчике нажатия мышью опции (это равносильно нажатию комбинации клавиш) присвоить свойству `Checked` значение `true`, а свойству `RadioItem` — `false`, то слева от названия опции появится "галочка". Если же хотите пометить выполненную опцию жирной точкой вместо галочки, присвойте свойству `GroupIndex` каждой опции собственное значение, а

свойство `RadioItem` установите в `true`. Если же значение свойства `GroupIndex` у всех опций будет одинаково, это означает, что все опции принадлежат одной группе. В такой группе при `Checked = true` присвоить свойству `RadioItem` значение `true` можно только для одной опции, т. е. только одна опция группы может быть помечена жирной точкой.

Ввод и вывод данных в нашем приложении осуществляется через компоненты `Edit1`, `Edit2`. В листинге 11.3 приведен пример активизации поля ввода через задание в тексте метки "горячей" клавиши <1>. Здесь же — пример задания глобальных переменных в h-файле.

Листинг 11.3

```
//cpp-модуль
//---------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include "Unit2.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
 i=0;
 InputCount=0;
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
```

```
{
Form1->Close();
}
//---------------------------------------------------------
void __fastcall TForm1::Button2Click(TObject *Sender)
{
Form2->Show();
}
//---------------------------------------------------------
void __fastcall TForm1::Button3Click(TObject *Sender)
{
Form2->ShowModal();
}
//---------------------------------------------------------
void __fastcall TForm1::Button4Click(TObject *Sender)
{
        if(Form1->Color == clBtnFace)
           Form1->Color = clRed;
        else Form1->Color= clBtnFace;
}
//---------------------------------------------------------
void __fastcall TForm1::Button5Click(TObject *Sender)
{
        if(Form1->Button1->Cursor==crDefault)
          {
           Form1->Cursor=crHandPoint;
           Form1->Button1->Cursor=crHandPoint;
           Form1->Button2->Cursor=crHandPoint;
           Form1->Button3->Cursor=crHandPoint;
           Form1->Button4->Cursor=crHandPoint;
           }
        else
          {
           Form1->Cursor=crDefault;
```

```
            Form1->Button1->Cursor=crDefault;
            Form1->Button2->Cursor=crDefault;
            Form1->Button3->Cursor=crDefault;
            Form1->Button4->Cursor=crDefault;
            }
}
//---------------------------------------------------------

void __fastcall TForm1::N2Click(TObject *Sender)
{
        if(Form1->Color == clBtnFace)
           Form1->Color = clRed;
        else Form1->Color= clBtnFace;
}
//---------------------------------------------------------
void __fastcall TForm1::N3Click(TObject *Sender)
{
    if(Form1->Button1->Cursor==crDefault)
          {
           Form1->Cursor=crHandPoint;
           Form1->Button1->Cursor=crHandPoint;
           Form1->Button2->Cursor=crHandPoint;
           Form1->Button3->Cursor=crHandPoint;
           Form1->Button4->Cursor=crHandPoint;
           }
        else
          {
           Form1->Cursor=crDefault;
           Form1->Button1->Cursor=crDefault;
           Form1->Button2->Cursor=crDefault;
           Form1->Button3->Cursor=crDefault;
           Form1->Button4->Cursor=crDefault;
          }
}
```

```
//---------------------------------------------------------
void __fastcall TForm1::WithShow1Click(TObject *Sender)
{
Form2->Show();
}
//---------------------------------------------------------
void __fastcall TForm1::WithShowModal1Click(TObject *Sender)
{
Form2->ShowModal();
}
//---------------------------------------------------------
void __fastcall TForm1::CloseForm11Click(TObject *Sender)
{
Form1->Close();
}
//---------------------------------------------------------
void __fastcall TForm1::Button6Click(TObject *Sender)
{
       Label2->Visible=false;
       if(j >= SizeOfArray || j > InputCount)
        {
          j=0;
          Edit2->Text= m[j];
          j++;
        }
       else
        {
          Edit2->Text= m[j];
          j++;
        }

}
//---------------------------------------------------------
void __fastcall TForm1::N21Click(TObject *Sender)
```

```
{     if(N3->Enabled==true)
       N3->Enabled=false;
       else
        N3->Enabled=true;
}
//---------------------------------------------------------
void __fastcall TForm1::Edit1KeyDown(TObject *Sender, WORD
&Key,
      TShiftState Shift)
{
if (Key == VK_RETURN)      //Завершение ввода в Edit1
                           //по нажатии <Enter>
  {
    if(i<SizeOfArray)
     {
       m[i]=Edit1->Text; i++;
       Edit1->Text="";
       Edit1->SetFocus();
       InputCount=i;

     }
     else
      {
       Label2->Color=clRed;
       Label2->Visible=true;
       Label2->Caption="Дальше вводить нельзя!";
       j=0;
       Edit1->Text="";
       Button6->SetFocus();
      }
    }//VK_RETURN
}
//---------------------------------------------------------
void __fastcall TForm1::Button7Click(TObject *Sender)
{
```

```
        Label2->Visible=false;
        i=0; //это индекс массива-глобальный
        InputCount=0;
        Edit2->Text="";
        for(int i=0; i<SizeOfArray; i++)
                m[i]="";
}
//---------------------------------------------------------
//h-модуль
//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
#define  SizeOfArray 10 //my information
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TButton *Button4;
        TButton *Button5;
        TLabel *Label1;
        TMainMenu *MainMenu1;
        TMenuItem *CloseForm11;
        TMenuItem *CallForm21;
        TMenuItem *N1;
        TMenuItem *WithShow1;
```

```
        TMenuItem *WithShowModal1;
        TMenuItem *N2;
        TMenuItem *N3;
        TEdit *Edit1;
        TLabel *Label2;
        TEdit *Edit2;
        TButton *Button6;
        TMenuItem *N21;
        TButton *Button7;
        TMemo *Memo1;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Button4Click(TObject *Sender);
        void __fastcall Button5Click(TObject *Sender);
        void __fastcall N2Click(TObject *Sender);
        void __fastcall N3Click(TObject *Sender);
        void __fastcall WithShow1Click(TObject *Sender);
        void __fastcall WithShowModal1Click(TObject *Sender);
        void __fastcall CloseForm11Click(TObject *Sender);
        void __fastcall Button6Click(TObject *Sender);
        void __fastcall N21Click(TObject *Sender);
        void __fastcall Edit1KeyDown(TObject *Sender,
        WORD &Key,
          TShiftState Shift);
        void __fastcall Button7Click(TObject *Sender);
private:   // User declarations
public:   // User declarations
AnsiString m[SizeOfArray];
int i,j;
int InputCount; //количество введенных в массив чисел
        __fastcall TForm1(TComponent* Owner);
};
```

```
//---------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------
endif
```

Вторая форма нужна для демонстрации вызова форм в модальном виде. При закрытии второй формы, открытой в модальном виде, используются кнопка и раскрывающееся меню. При использовании кнопки в ее свойстве `ModalResult` установлено значение `mrOK`. Поэтому при закрытии формы ее свойству `ModalResult` ничего не присваивается, т. к. в него попадет то значение, которое установлено в этом свойстве у кнопки (если бы кнопка не использовалась для закрытия модальной формы, то для успешного закрытия такой формы ее свойству `ModalResult` надо было бы присвоить перед закрытием ненулевое значение). Вид второй формы с ее компонентами представлен на рис. 11.14.

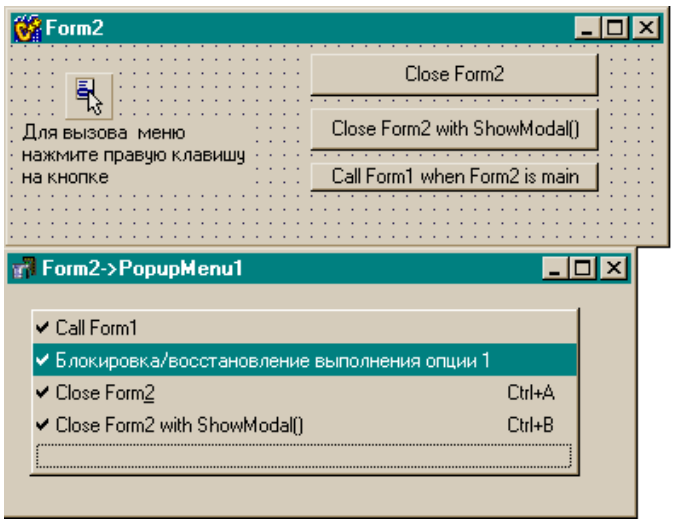


Рис. 11.14. Вид второй формы приложения

Текст модуля второй формы приведен в листинге 11.4.

Листинг 11.4

```
//cpp-модуль
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit2.h"
#include "Unit1.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm2 *Form2;
//---------------------------------------------------------
__fastcall TForm2::TForm2(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm2::Button1Click(TObject *Sender)
{
Form2->Close();
}
//---------------------------------------------------------
void __fastcall TForm2::Button2Click(TObject *Sender)
{
Form2->Close();
Form1->Show();
}
//---------------------------------------------------------
void __fastcall TForm2::Button3Click(TObject *Sender)
```

```
{
Form1->Show();
}
//---------------------------------------------------------
void __fastcall TForm2::Item1Click(TObject *Sender)
{
   Form2->Close();
}
//---------------------------------------------------------
void __fastcall TForm2::Item2Click(TObject *Sender)
{
ModalResult='mrOK';
Form2->Close();
Form1->Show();
}
//---------------------------------------------------------
void __fastcall TForm2::CallForm11Click(TObject *Sender)
{
Form1->Show();
}
//---------------------------------------------------------
void __fastcall TForm2::N11Click(TObject *Sender)
{
  if(Item1->Enabled)
    Item1->Enabled=false;
  else
     Item1->Enabled=true;
}
//---------------------------------------------------------

//h-модуль
//---------------------------------------------------------
#ifndef Unit2H
#define Unit2H
```

```
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
//---------------------------------------------------------
class TForm2 : public TForm
{
__published:  // IDE-managed Components
        TButton *Button1;
        TButton *Button2;
        TButton *Button3;
        TPopupMenu *PopupMenu1;
        TMenuItem *Item1;
        TMenuItem *Item2;
        TMenuItem *CallForm11;
        TMenuItem *N11;
        TLabel *Label1;
        void __fastcall Button1Click(TObject *Sender);
        void __fastcall Button2Click(TObject *Sender);
        void __fastcall Button3Click(TObject *Sender);
        void __fastcall Item1Click(TObject *Sender);
        void __fastcall Item2Click(TObject *Sender);
        void __fastcall CallForm11Click(TObject *Sender);
        void __fastcall N11Click(TObject *Sender);
private:      // User declarations
public:       // User declarations
        __fastcall TForm2(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm2 *Form2;
//---------------------------------------------------------
#endif
```

Некоторые функции выдачи сообщений и перевода данных из одного типа в другой

Здесь мы рассмотрим некоторые полезные функции, используемые для перевода строковых данных (чисел, записанных в виде строки данных) в числа и наоборот, а также функции вывода сообщений на экран, которые необходимы, если требуется не только сообщить пользователю какую-то информацию, но и потребовать его действий в ответ на такую информацию. Все строковые данные, которые мы будем рассматривать, это данные типа `AnSiString`. Если у пользователя возникнет необходимость перейти к строкам, задаваемым как `char * a;` или `char m[количество]`, то к таким строкам легко перейти, воспользовавшись методом `c_Str()` класса `AnSiString`. Итак, рассмотрим функции.

- `StrToCurr()` — преобразует строку `AnSiString` в объект типа `Currency` (валюта). Обращение:

  ```
  AnSiString  S; Currency x= StrToCurr(S);
  ```

  Строка `AnSiString` представляет собой число с плавающей точкой, соответствующее объекту типа `Currency`. Лидирующие и хвостовые пробелы в строке игнорируются.

- `CurrToStr()` — выполняет обратное преобразование. Объект типа `Currency` преобразуется в строку данных типа `AnSiString`.

- `ShowMessage()` — выводит сообщение в специальном окне с кнопкой **OK**, которая организует ожидание. Имя исполняемого приложения выводится в заголовочной части окна. Обращение к этой функции:

  ```
  AnSiString MSg; ShowMessage(MSg);
  ```

Пример использования всех рассмотренных выше функций показан в программе, текст которой приведен в листинге 11.5, а форма в режиме проектирования и исполнения показана на рис. 11.15.

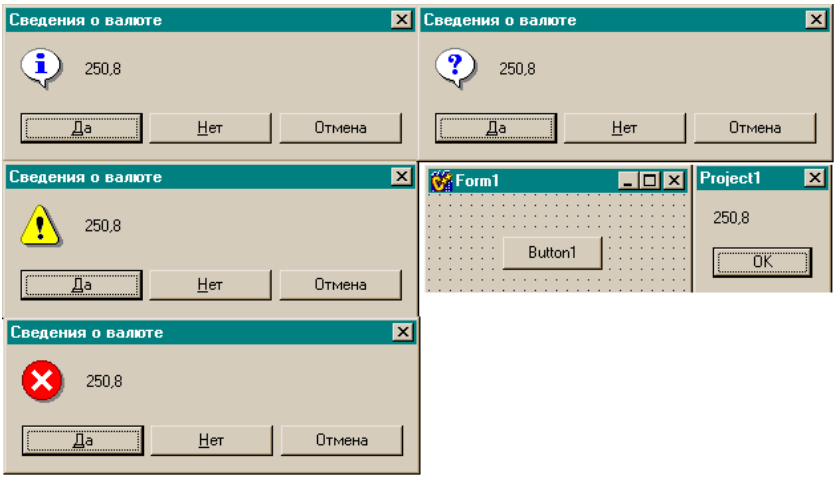


Рис. 11.15. Форма в режиме проектирования и исполнения для проверки функций и окна сообщений

Листинг 11.5

```
void __faStcall TForm1::Button1Click(TObject *Sender)
{
AnSiString  S="250,8"; Currency x= StrToCurr(S);
ShowMessage(CurrToStr(x));
}
```

- `FloatToStr()` — преобразует число с плавающей точкой в строку типа `AnSiString`. Обращение:

  ```
  float x=12.5; AnSiString S; S= FloatToStr(x);
  ```

- `StrToFloat()` — выполняет обратное преобразование: строковые данные преобразуются в число с плавающей точкой. Обращение:

  ```
  float x; AnSiString S="12.5";  x= StrToFloat(S);
  ```

- `IntToStr()` — преобразует целое число в строку типа `AnSiString`.

Обращение:

```
int x=-12; AnSiString S; S= IntToStr(x);
```

- `StrToInt()` — выполняет обратное преобразование: строковые данные преобразует в целое число. Обращение:

```
AnSiString S="-12"; int x=StrToInt(S);
```

- `MessageBox()` — эта функция выводит окно с сообщением. Окно содержит несколько кнопок, обеспечивающих выбор ситуации, которую можно обрабатывать. Вместе с окном сообщения функция выводит значение клавиши, которая была нажата в ответ на сообщение. Вид функции:

```
MessageBox(дескриптор окна, в которое попадет окно
сообщений, "Текст сообщения, которое попадет в поле
окна сообщения", "Текст сообщения, которое попадет
в заголовок окна сообщения", комбинация битовых
флажков, определяющих, какие кнопки и какие значки
должны появиться в окне сообщений)
```

Примечание

Дескриптор окна, в которое попадет окно сообщений, по умолчанию имеет значение `NULL`.

Возможные комбинации битов для определения кнопок задаются в виде:

`MB_YESNO`

`MB_OK`

`MB_OKCANCEL`

`MB_RETRYCANCEL`

`MB_YESNOCANCEL`

Какой значок появится в поле сообщения, определяется заданием следующих флажков, которые записываются после задания кнопок через операцию "ИЛИ"(`|`): `MB_ICONINFORMATION`, `MB_ICONEXCLAMATION`, `MB_ICONSTOP`, `MB_ICONQUESTION`.

Функция `MessageBox()` выдает значение кнопки, нажатой в ответ на сообщение. Значение можно применять, пользуясь табл. 11.1.

Таблица 11.1. *Значения, возвращаемые функцией* `MessageBox()`

Символическая константа	Значение
IDDOK	1
IDCANCEL	2
IDABORT	3
IDRETRY	4
IDIGNORE	5
IDYES	6
IDNO	7

В операторы можно вставлять как символические константы, так и их числовые эквиваленты. Приведем пример приложения, использующего функцию `MessageBox()` (листинг 11.6).

Листинг 11.6

```
void __faStcall TForm1::Button1Click(TObject *Sender)
{
AnSiString  S="250,8"; Currency x= StrToCurr(S);
int i=MessageBox(NULL,CurrToStr(x).c_Str(), "Сведения о
валюте",MB_YESNOCANCEL | MB_ICONSTOP);
if(i==IDYES || i ==IDNO)  return;
elSe
 exit(0);
}
```

Компонент *TListBox*

Этот компонент представляет собой простой список, пример которого приведен на рис. 11.16.

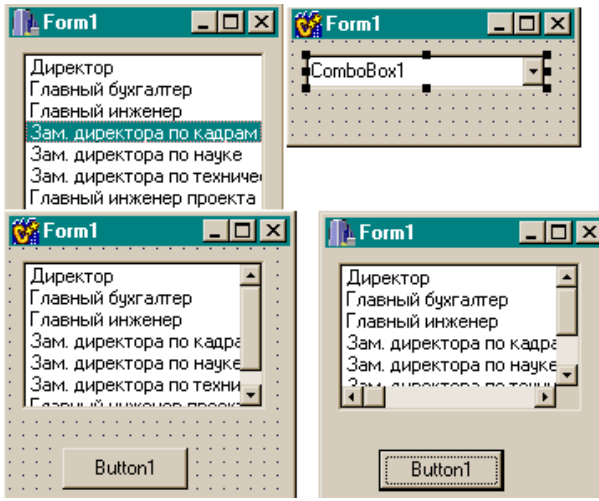


Рис. 11.16. Примеры `TListBox`

Как использовать *TListBox*

Компонент создает прямоугольную область, в которой отображается список текстовых строк. Эти текстовые строки можно добавлять в список, выбирать или удалять из него.

Как используется этот компонент? Например, в процессе решения некоторой задачи вводятся данные о сотрудниках предприятия, и каждый раз приходится вводить должность сотрудника. Список должностей помещается на этапе разработки приложения в некоторый файл, который затем поддерживается в актуальном состоянии. Когда приложение запущено, этот файл загружается в `ListBox`, и если необходимо ввести какую-либо должность в базу данных, то достаточно открыть список должностей и щелкнуть на требуемой должности — ее наименование перекочевывает в базу данных.

Как формировать список строк

На этапе разработки приложения можно сформировать так называемый отладочный список, который в дальнейшем неудобно

поддерживать в актуальном состоянии, когда приложение находится в эксплуатации, т. к. требуется корректировка списка и перекомпиляция приложения. Но для отладочных работ список надо сформировать. Это делается с помощью Редактора текста точно в таком же окне и по тем же правилам, что и для компонента `TMemo`, только Редактор открывается кнопкой не в свойстве `Lines`, а в свойстве `Items`. Для компонента `TListBox` свойство `Items` играет роль, аналогичную роли свойства `Lines` для компоненте `TMemo`. Редактор текста приведен на рис. 11.16.

Некоторые свойства *TListBox*

- `Items`, `ItemIndex`. Вид списка в форме после компиляции приложения будет таким, как это показано в начале раздела. Свойство `Items`, как и свойство `Lines` в компоненте `TMemo`, является указателем на класс `TStrings`, который, в свою очередь, имеет свойство `Strings`, являющееся массивом строк типа `AnsiString`. Поэтому, во-первых, в классе `TStrings` существует свойство `Count`, которое содержит количество строк в массиве `Strings`, т. е. если записать `ListBox1->Items->Count`, то получим количество строк, во-вторых, в классе `TListBox` существует свойство `ItemIndex`, в котором всегда находится номер выбранной в списке строки, и поэтому строку из списка после щелчка на ней мышью можно выбрать, написав

  ```
  ListBox1->Items->Strings[ListBox1->ItemIndex];
  ```

  Заметим, что в квадратных скобках мы не могли писать просто `ItemIndex`, т. к. это свойство не класса `TStrings`, а класса `TListBox`, поэтому мы и указали в квадратных скобках эту принадлежность. И в-третьих: т. к. строки в массиве — это строки `AnsiString`, то к ним применимы методы этого класса для их преобразования и вообще — для работы с ними. У класса `TStrings`, указателем на который является свойство `Items`, есть методы `Add()` — добавить строку в конец списка, `Delete()` — удалить строку, и другие методы, которыми можно воспользоваться для модификации списка `ListBox`. Например, можно написать

  ```
  ListBox1->Items->Add("Последняя строка списка");
  ```

или:

```
ListBox1->Items->LoadFromFile("a.txt");
```

- `Sorted` — устанавливает, будут ли строки списка упорядочены по алфавиту.

События *TListBox*

Собственных (не унаследованных) событий у компонента нет.

Некоторые методы *TListBox*

- `Clear()` — удаляет список.
- `SetFocus()` — делает список активным (с ним можно работать).
- `Hide()` — скрывает список.
- `Show()` — делает список видимым.
- `Sort()` — сортирует элементы списка в порядке их возрастания.

Включение горизонтальной полосы прокрутки списка

Когда в `TListBox` количество строк не помещается в отведенный размер прямоугольника, то автоматически включается вертикальная полоса прокрутки. Однако этого не происходит, когда длина строки больше ширины прямоугольника. Из этого положения можно выйти, применив функцию `Perform()` из Windows API следующим образом:

```
int MaxWidth=1000;
```

Мы взяли максимальную строку списка именно такого размера, хотя можно было бы установить автоматическое определение самой длинной строки. Но это лишнее, т. к. вряд ли на практике в списке будут такие большие строки. А если вы опасаетесь, что такое может случиться, задайте число 10 000.

```
ListBox1->Perform(LB_SETHORIZONTALEXTENT,MaxWidth,0);
```

Результат представлен на рис. 11.16.

Чтобы появилась полоса прокрутки, надо щелкнуть мышью на непомещающейся строке после выполнения команды `Perform`.

В листинге 11.7 представлен обработчик кнопки.

Листинг 11.7

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 int MaxWidth=1000;
ListBox1->Perform(LB_SETHORIZONTALEXTENT,MaxWidth,0);
 ListBox1->Items->LoadFromFile("c:\\a.txt");
}
```

Компонент *TComboBox*

Этот компонент (вид его в форме приведен на рис. 11.16) является комбинацией редактируемого поля и списка `TListBox`: в форме он представляется в виде редактируемого поля с треугольной кнопкой справа. Выбрать данные из списка можно двумя способами: открыть список и в нем щелчком выбрать необходимую строку (список тут же закроется) или ввести нужную строку в редактируемое поле. В последнем случае, если включено свойство `AutoComplete`, может осуществляться автоматический выбор из списка. Как только в редактируемом поле вы набрали первый символ, в нем автоматически появится строка из списка, имеющая такой же первый символ. Если набрали второй символ, появится строка, первые два символа которой совпадают с набранными, и т. д. То есть набираемые символы как бы дополняются автоматически символами подходящей строки. Когда пользователь вводит данные (набирает строку в редактируемом поле или выбирает ее из списка), данные попадают в свойство `Text`.

Существуют три типа комбинированных списков: drop-down (по умолчанию), drop-down list и simple. Все эти типы указываются при определении свойства `Style`.

- Значение `csDropDown` используется, когда вы применяете и редактируемое поле, и выпадающий список. Как им пользоваться, мы только что разобрали. Кроме того, если вы включите свойство `AutoDropDown`, то при вводе хотя бы одного символа в редактируемое поле список автоматически раскрывается.
- Значение `csDropDownList` открывает доступ к редактируемому полю только для чтения, тем самым заставляя пользователя выбирать из списка. Кстати, если вы хотите, чтобы в списке появлялось нужное вам количество строк, задайте это количество в свойстве `DropDownCount`.
- Значение `csSimple` позволит создать список, который не будет закрываться, когда вы щелкнете на выбранной строке (обычно список закрывается, и выбранная строка появляется в редактируемом поле, а для пользования ею — в свойстве `Text`). В данном случае список после компиляции и запуска приложения остается открытым. Удостоверьтесь при этом, что свойство `Height`, которое определяет величину окна списка, достаточно велико (его надо задать самому), иначе окно списка не появится, а на экране останется только редактируемое поле. Теперь, когда вы щелкнете на любой строке списка, выбранная строка появится в редактируемом поле, но список не закроется. Получается почти полный аналог компонента `TListBox`.

  Список формируется так же, как и для `TListBox`.

Компонент *TMaskEdit*

С помощью этого компонента создается редактируемое текстовое поле (*маска*) для ввода данных специфического формата: дат, времени, номеров телефонов и т. д. Если вы задали формат ввода данных по конкретной маске, то при вводе текста проверяется, соответствует ли он этому формату. Маска налагает ограничения на символы, вводимые по маске, и на формат данных. Контроль ввода осуществляется посимвольно: если пользователь попытается ввести запрещенный в маске символ, этот символ системой контроля будет отвергнут.

Маска — это строка, состоящая из трех полей, разделенных точкой с запятой. Первое поле — это собственно и есть сама маска. Вторая часть маски — это символ, который определяет, будут ли литеральные символы, присутствующие в маске, сохраняться как часть введенных данных. Третья часть маски — это символ, используемый для задания не входящих в маску символов.

- Первое поле маски.

  Ниже приведен перечень специальных символов, задающих собственно маску (т. е. первую часть всей маски).

  - `!` — если этот символ присутствует в маске, необязательные символы интерпретируются как лидирующие пробелы. Если же этот символ отсутствует, то необязательные символы интерпретируются как хвостовые пробелы.
  - `>` — если этот символ появляется в маске, это означает, что все последующие символы должны быть набраны в верхнем регистре клавиатуры до конца маски или до появления символа `<`.
  - `<` — если этот символ появляется в маске, это означает, что все последующие символы должны быть набраны в нижнем регистре клавиатуры до конца маски или до появления символа `>`.
  - `<>` — если оба эти символа появляются в маске, это означает, что можно использовать и верхний, и нижний регистры клавиатуры.
  - `\` — этот символ в маске означает, что символ ввода, следующий за ним, это символ — литерал (символьная константа).
  - `L` — требует ввода в позицию, в которой он находится, только символов алфавита установленного языка. Например, для Великобритании это будут символы A—Z, a—z. Ввод обязателен.
  - `l` — в позицию, которую он занимает, можно вводить только символы алфавита. Но если вы ничего не введете, система контроля разрешит вам перейти к следующим позициям.

- `A` — требует ввода в позицию, им занимаемую, только алфавитно-цифровых символов. Для Великобритании это символы A—Z, a—z, 0—9.
- `a` — по своей функции совпадает с символом `A`, но не требует обязательного ввода.
- `C` — требует обязательного ввода любого символа в позицию, которую он занимает в маске.
- `c` — по своей функции совпадает с символом `C`, но не требует обязательного ввода.
- `0` — требует обязательного ввода цифры в позицию, занимаемую им.
- `9` — функционально совпадает с символом 0, но не требует обязательного ввода.
- `#` — позволяет вводить в свою позицию цифру или знаки `+` и `-`, но не требует обязательного ввода.
- `:` — используется при вводе времени для разделения часов, минут и секунд. Если в установках вашего компьютера использован другой разделитель, преимущество имеет этот символ.
- `/` — используется в датах для разделения года, месяца и числа. В отношении установок вашего компьютера действует то же правило, что и для разделителя времени.
- `;` — используется для разделения частей маски.
- `_` — автоматически вставляет пробелы в текст: когда вы вводите символы в поле, курсор ввода автоматически перепрыгивает через символ _.

Символы, которых нет в приведенном списке, могут появляться собственно в маске (т. е. в первой части всей маски) в качестве литералов. Они должны быть точно определены. Такие символы вставляются автоматически, и курсор ввода перескакивает через них. Кроме того, установить нужные символы в маске можно, задав символ `\`, после которого система контроля будет "воспринимать" все последующие символы как литералы (если не встретится другой управляющий символ).

- Второе поле маски — это один символ, который указывает, могут ли символы-литералы из маски включаться в данные. Например, маска телефонного номера выглядит так: `(000)_000-0000;0;*`

  Символ 0 во втором поле (то, что после точки с запятой) показывает, что свойство `Text` компонента может включать в себя 10 цифр ввода вместо 14-ти, т. к. остальные (литералы) не будут включены в результат ввода этих данных. Такой символ можно изменять в окне редактора свойства `EditMask`.

- Третье поле маски — содержит символ, который увидит пользователь в поле редактирования до того, как будут введены данные. По умолчанию это символ _, который будет заменен на вводимый пользователем символ. Но если ввод данных по маске закончен, а символ _ остался в некоторых позициях, это означает, что ввод в эти позиции запрещен.

Задание маски

Маска (шаблон ввода) задается с помощью Редактора маски, который открывается нажатием кнопки с многоточием, расположенной в свойстве `EditMask`. Если нажать кнопку, появится диалоговое окно для задания маски.

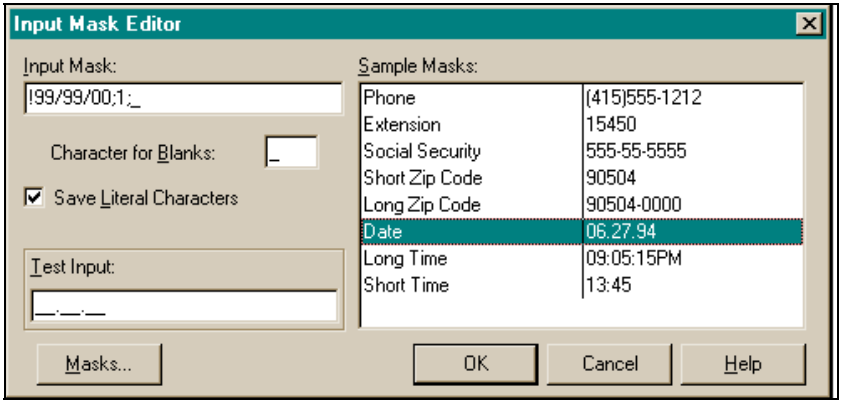


Рис. 11.17. Диалоговое окно для задания маски

Поля и кнопки в окне Редактора маски

- В поле **Input Mask** вводится сама маска. Вы можете выбрать формат данных в окне **Sample Masks** либо ввести маску самостоятельно.
- В поле **Character for Blanks** вы задаете символ, который увидит пользователь до ввода данных. Если вы работаете с полем **Sample Masks**, этот символ появляется сам.
- Если установлен флажок **Save Literal Characters**, то символы-литералы из маски могут включаться в данные, вводимые пользователем.
- После формирования шаблона можно ввести значение данных в поле **TestInput** и проверить, как работает маска.
- Кнопка **Masks** открывает диалоговое окно, в котором можно выбрать файл с масками, определенными для данной страны.

Куда помещается текст, введенный по маске

Текст, введенный, но еще не отформатированный, находится в свойстве `Text`. Отформатированный же текст (готовый к дальнейшему употреблению) находится в свойстве `EditText`.

Проверочная программа

Текст проверочного приложения приведен в листинге 11.8. Шаблон ввода и результат работы приложения показаны на рис. 11.18.

Листинг 11.8

```
//---------------------------------------------------------
//cpp-файл
//---------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------
```

```
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::MaskEdit1Exit(TObject *Sender)
{
  ListBox1->Items->Add(MaskEdit1->EditText);
/*  MaskEdit1->Text="";    если включить эту строку, то поле
ввода будет очищено, иначе там останется предыдущая введенная
дата, которая погасится при первом введенном символе новой
даты*/
  MaskEdit1->SetFocus();
}
//---------------------------------------------------------
//h-файл
//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Mask.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TMaskEdit *MaskEdit1;
        TListBox *ListBox1;
```

```
        void __fastcall MaskEdit1Exit(TObject *Sender);
private:       // User declarations
public:        // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------
#endif
```

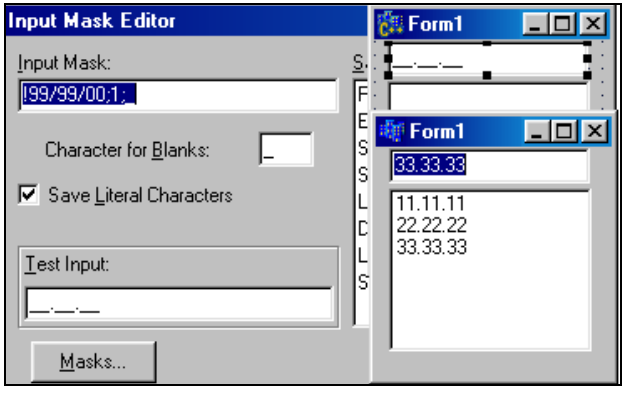


Рис. 11.18. Шаблон ввода
и результат работы приложения

Компонент *TCheckBox*

Это флажок, предназначенный для управления двумя состояниями. Имеет маленькое поле, в котором появляется галочка, если на нем щелкнуть первый раз. А при повторном щелчке галочка исчезает. Описательный текст флажка задается в свойстве `Caption`. Контроль осуществляется с помощью придания значения свойству `Checked` или проверки его содержимого: если оно равно `false`, галочка спрятана, если — `true`, галочка появляется в окне. Свойство `State` позволяет следить за состоянием флажка: включен ли он (`cbChecked`), выключен ли (`cbUnchecked`) или

закрашен серым цветом (`cbGrayed`). Если, скажем, мы запустили какой-то процесс, нажав на флажок, то в дальнейшем, проверив состояние флажка, мы можем этот процесс выключить. Три состояния флажка возможны, если свойство `AllowGrayed` включено. Текст проверочной программы приведен в листинге 11.9. Вид формы в режиме проектирования и результат выполнения программы показаны на рис. 11.19.

Листинг 11.9

```
//cpp-файл
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::CheckBox1Click(TObject *Sender)
{
if (CheckBox1->State==cbChecked)
   Label1->Caption="Флажок включен!";
  else if (CheckBox1->State==cbUnchecked)
        Label1->Caption="Флажок выключен!";
       else
        Label1->Caption="Флажок закрашен";
}
//---------------------------------------------------------
```

```
//h-файл
//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
        TCheckBox *CheckBox1;
        TLabel *Label1;
        void __fastcall CheckBox1Click(TObject *Sender);
private:   // User declarations
public:    // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

В программе обрабатывается событие `OnClick` компонента. Щелчок на компоненте переключает флажок из одного состояния в другое. Остальные нерассмотренные свойства компонента, а также его события и методы можно посмотреть с помощью раздела **Help**.

Компонент *TRadioButton*

Это переключатель, который имеет два состояния и описательный текст, указывающий на назначение переключателя. В отличие от флажков `TCheckBox`, которых в одном контейнере (на-

пример, на одной панели) может быть включено несколько, из группы переключателей в одном контейнере включенным может быть только один, остальные автоматически выключаются. Это свойство обоих компонентов проиллюстрировано на рис. 11.19.

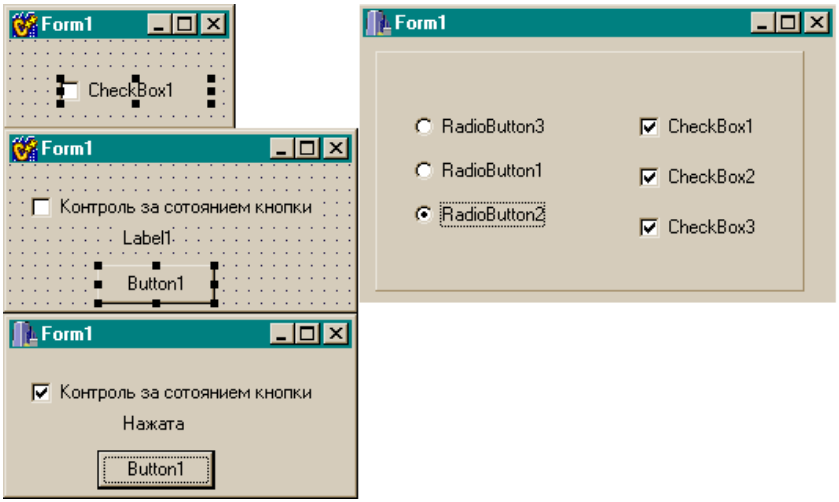


Рис. 11.19. Данные о работе `TCheckBox` и его взаимодействии с `TRadioButton`

Здесь все переключатели помещены на одну панель и составляют одну группу. События компонента `TRadioButton` почти такие же, как и у его предыдущего "коллеги", но отсутствует свойство `State`. Используется компонент `TRadioButton` в силу своей специфики несколько иначе. Допустим, форму надо перекрасить в один из трех цветов. Это делается с помощью переключателя `RadioButton`, т. к. он позволяет перекрасить только в один цвет: две кнопки нажать не удастся. Для более удобного использования переключателя `RadioButton` применяют групповой компонент `TRadioGroup`, позволяющий за одно обращение к нему задавать необходимое количество радиокнопок, а не помещать в форму для каждой кнопки свой компонент.

Компонент *TRadioGroup*

Довольно часто вместо отдельных переключателей используют их групповой контейнер — компонент `TRadioGroup`.

`TRadioGroup` представляет собой группу переключателей, которые функционируют совместно. `TRadioGroup` — это специальный контейнер, который содержит только переключатели. Такие кнопки называются *сгруппированными*. Когда пользователь нажимает на один переключатель, остальные переключатели этой группы выключаются. Два переключателя в форме могут быть включены одновременно, если они располагаются в разных контейнерах. Чтобы добавить новый переключатель в `TRadioGroup`, надо открыть окно свойства **Items** в Инспекторе объекта (как в `TListBox`). Каждая строка в **Items** обеспечивает появление переключателя в контейнере со своей строкой наименования. Значение свойства **ItemIndex** в режиме исполнения приложения определяет, какой переключатель в данный момент активен в контейнере (как в `TListBox`). Причем, если имеем `RadioGroup1->ItemIndex==0`, это означает, что включен первый переключатель. Изменить сопроводительный текст переключателя, на который указывает значение свойства `ItemIndex`, можно так:

```
RadioGroup1 ->Items->Strings[RadioGroup1-
>ItemIndex]="Текст";
```

Здесь мы видим полную аналогию с компонентом `TlistBox`. В *разд. "Компонент TlistBox" данной главы* дается объяснение, почему именно так надо записывать.

Изменение расположения переключателей в контейнере с горизонтального на вертикальное и обратно осуществляется с помощью свойства `Columns`.

Попробуем перекрасить в разные цвета сам компонент, помещенный в форму. Окно Редактора текста, которое мы открыли через свойство `Items` и с помощью которого задали три переключателя, приводится на рис. 11.20. Там же показан и результат перекраски.

Текст обработчика события — в листинге 11.10.

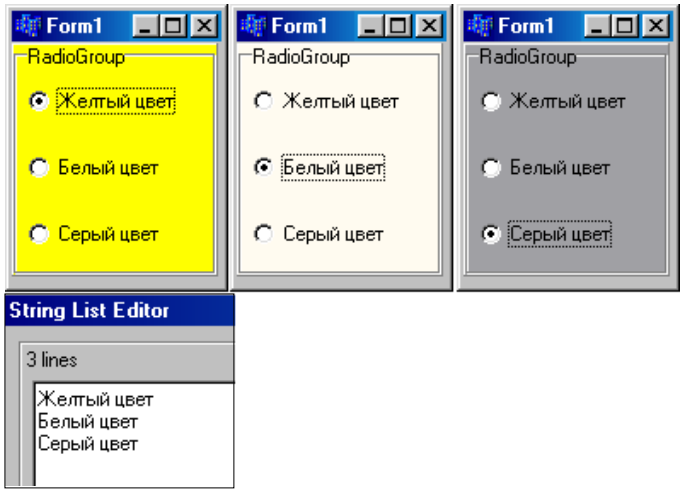


Рис. 11.20. Диалоговое окно для задания переключателей на панели `TRadioGroup` и результат перекраски панели

Листинг 11.10

```
void __fastcall TForm1::RadioGroup1Click(TObject *Sender)
{
if(RadioGroup1 ->ItemIndex==0)
  RadioGroup1->Color=clYellow;
 else if(RadioGroup1 ->ItemIndex==1)
  RadioGroup1->Color=clCream;
       else
         RadioGroup1->Color=clMedGray;
}
```

Компонент *TCheckListBox*

`TCheckListBox` выводит список с полосами прокрутки. Слева от каждого элемента списка располагаются флажки. Компонент `TCheckListBox` схож с `TListBox`. Отличие состоит в том, что каждый элемент имеет флажок, который может включаться или

выключаться пользователем. Это позволяет принимать решения после того, как элемент из списка выбран нажатием мыши. Например, мы выбрали некоторую строку из списка. Затем смотрим, включен ли у нее флажок. Если включен, то работаем со строкой по одному алгоритму. Если же флажок выключен — работаем со строкой по другому алгоритму.

Компонент *TImage*

Через этот компонент в форму выводится графическое изображение.

Какое изображение надо выводить, указывается в свойстве `Picture`. Если нажать кнопку с многоточием в поле этого свойства, откроется диалоговое окно для выбора объекта в формате BMP, JPEG, значка или метафайла. Компонент `TImage` содержит некоторые свойства, определяющие, как выводить изображение внутри границ самого этого объекта (в форме `TImage` отображается в виде пустого квадрата).

Форма с выбранным изображением показана на рис. 11.21.

Рис. 11.21. Результат выбора изображения

Некоторые свойства *TImage*

- `Picture` — задает изображение, которое появляется в компоненте. Это указатель на класс `TPicture`, объектами которого являются значки, метафайлы, растровые изображения, кото-

рые могут выводиться через свойство Picture. В поле этого свойства имеется кнопка с многоточием, с помощью которой открывается диалоговое окно для загрузки изображения в компонент. Можно загружать и сохранять изображение и в режиме исполнения приложения с помощью методов класса TPicture:

- LoadFromFile() — позволяет загрузить изображение из файла. Обращение к методу:

  ```
  Image1->Picture->LoadFromFile("имя файла");
  ```

- SaveToFile() — позволяет сохранить изображение в файле. Обращение к методу: I

  ```
  mage1->Picture->SaveToFile("имя файла");
  ```

- Assign() — о нем скажем чуть ниже, в абзаце, посвященном свойству Canvas.

❐ Stretch — указывает, может ли компонент растягиваться или сжиматься в соответствии с размерами изображения или, наоборот, изображение примет размеры компонента. Если свойство Stretch имеет значение true, то изображение принимает размеры и форму компонента. Если компонент изменит размеры, то изображение тоже изменит размеры. Свойство Stretch меняет размеры высоты и ширины изображения непропорционально. Поэтому если форма и размер компонента значительно отличаются от формы и размеров изображения, то растягивание может исказить изображение. Чтобы привести размеры компонента в соответствие с размерами изображения, надо воспользоваться свойством AutoSize вместо Stretch. Если свойство Picture содержит значок, свойство Stretch не работает. Если свойство AutoSize имеет значение false, то компонент можно растягивать и сжимать. Сожмется рамка компонента и при AutoSize = true. Вывод такой: если AutoSize = true, то рамка компонента всегда будет автоматически настраиваться на размер изображения и ее нельзя будет ни сжать, ни расширить. Свойство Stretch в этом случае отключено. Но если отключена автоматическая настройка рамки компонента на размеры изображения (AutoSize=false), то включается свойство Stretch.

- `Canvas` — битовая карта плоскости для рисования на ней различных изображений. Это свойство само является классом со своими свойствами и методами, позволяющими рисовать изображения. Орудия рисования — это `Pen`, `Brush` и методы. Свойство `Canvas` доступно только в случае, если свойство `Picture` содержит BMP-изображение. Приведем пример приложения, которое рисует заштрихованный круг на картинке, изображенной в компоненте `TIMage`, с помощью свойства `Canvas`, его методов и свойств. При первом щелчке на компоненте на нем появляется круг, а само изображение предварительно сохраняется в буфере Clipboard (чтобы работала функция запоминания в буфере, в программный модуль включен файл Clipbrd.hpp). Копирование в буфер происходит методом `Assign()` свойства `Picture` (которое само является классом) по правилу:

  ```
  Destination->Assign(Source);
  ```

  (`Куда->Assign(Откуда);`).

  При повторном щелчке на компоненте изображение восстанавливается. Код приложения приведен в листинге 11.11, а результат его работы показан на рис. 11.22.

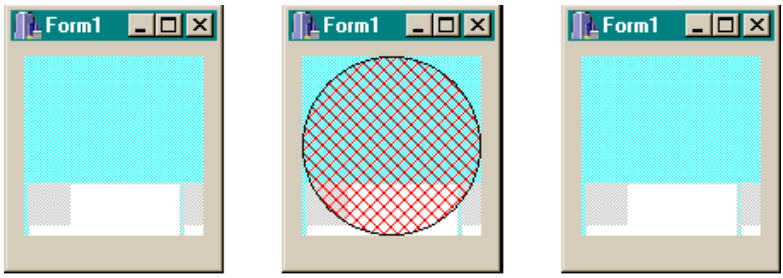


Рис. 11.22. Рисование на изображении

Листинг 11.11

```
//cpp-модуль
//---------------------------------------------------------------
```

```
#include <vcl.h>
#pragma hdrstop
#include "Unit1.h"
#include <Clipbrd.hpp>
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Image1Click(TObject *Sender)
{
  if(Image1->Tag == 0)
   {
     Clipboard ()->Assign(Image1->Picture);    //копирование
                                               //изображения
                                               //в буфер
     // Destination->Assign(Source);
     TCanvas *p=Image1->Canvas;    //Настроились на свойство
                                   //Canvas компонента Image1
      p->Brush->Color=clRed;       //установка цвета кисти
      p->Brush->Style=bsDiagCross;  /*установка стиля
      закрашивания кистью: штриховка поверхности*/
      p->Ellipse(0, 0, Image1->Width, Image1->Height);
/*рисование эллипса, вписанного в компонент*/
      Image1->Tag=1;
      return;
    }
      Image1->Tag=0;
      Image1->Picture->Assign(Clipboard ());
}
//---------------------------------------------------------
```

Компонент *TShape*

Этот компонент предназначен для рисования простых геометрических фигур, вид которых задается в его свойстве `Shape`. Цвет и способ штриховки фигуры задаются в свойстве `Brush` с помощью вложенных свойств `Color` и `Style`. Характеристики контура фигуры задаются во вложенных свойствах свойства `Pen`. Контур фигуры определяется свойством `Pen`, а заливка — свойством `Brush`. Пример нарисованной фигуры приведен на рис. 11.23.

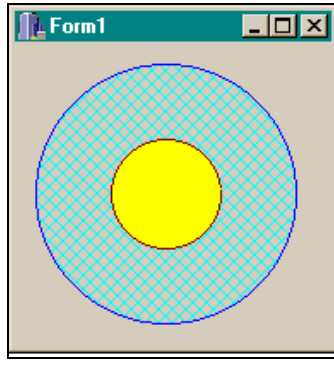


Рис. 11.23. Пример нарисованной фигуры

Компонент *TBevel*

Этот компонент задает обрамление: с его помощью задают рамки, кадры или линии, которые могут быть выпуклыми или вогнутыми, придавая рисунку более эстетичный вид. Обрамляющую фигуру задают в свойстве `Shape`, а в свойстве `Style` задают, будет ли фигура вогнутой или выпуклой.

Отметим, что событий у этого компонента нет.

Компонент *TPageControl*

Этот компонент позволяет построить набор страниц, которые друг друга перекрывают и которые можно перелистывать. Но главная их ценность в том, что на эти страницы в рамках одной формы, в которой размещен сам компонент `TPageControl`, можно помещать другие компоненты, тем самым расширяя возможности формы. Например, вы должны разработать приложение по управлению кадрами предприятия. Все компоненты, обеспечивающие решение вашей проблемы, вы можете разместить в одной форме, но на разных страницах компонента `TPageControl`, который достаточно поместить в форму. На каждой странице вы можете разместить по одному справочнику и организовать на ней ведение справочника, на одной странице вы можете разместить элементы управления для получения аналитических таблиц и т. д. Чтобы переключиться между страницами достаточно щелкнуть на любой из них.

Как задавать страницы

Как задавать страницы? Поместите компонент `TPageControl` в форму и откройте его контекстное меню. В первых четырех строках вы увидите команды **New Page**, **Next Page**, **Previous Page**, **Delete Page**, с помощью которых и можно управлять формированием страниц. С помощью **New Page** можно создавать новые страницы, а с помощью остальных команд — перемещаться по уже сформированным страницам или удалять их. Если страницы не умещаются в поле компонента, автоматически формируются полосы прокрутки.

Если вам неудобно работать с полосами прокрутки, и вы хотите видеть все страницы сразу, задайте значение свойства `MultiLine` равным `true`, и вы получите желаемый результат.

Вы можете изменить форму полей с названиями страниц. Для этого надо воспользоваться свойством `Style`. Поля могут иметь форму кнопки или вкладки.

Некоторые свойства страницы *TTabSheet*

Каждая страница компонента `TPageControl` представляет собой отдельный объект — экземпляр класса `TTabSheet` со своими свойствами и событиями. Рассмотрим некоторые свойства страницы.

- `PageIndex` — указывает индекс страницы (ее номер) в списке страниц, поддерживаемых компонентом `TPageControl`. Каждой странице, когда она вставляется в компонент, автоматически назначается индекс. При этом первая страница получает нулевой номер, вторая — первый и т. д. Значение `PageIndex` переопределяется, когда страницы удаляются или передвигаются. Например, значение `TabSheet1->PageIndex` дает номер страницы во множестве страниц компонента.
- `TabIndex` — показывает порядковый номер станицы среди всех видимых страниц (видимость определяется значением свойства `TabVisible`). Значение 0 определяет первую видимую страницу, значение 1 — вторую видимую страницу и т. д. Значение свойства `TabIndex` всегда меньше значения `PageIndex` или равно ему. Если свойство `TabVisible` некоторой страницы имеет значение `false`, а `TabIndex` значение `-1`, то для невидимой страницы ее `TabIndex` = `-1`.

Некоторые свойства *TPageControl*

- `ActivePage` указывает на активную в данный момент страницу (т. е. на ту, которая в данный момент открыта). Можно им воспользоваться, чтобы установить самому в режиме исполнения приложения активную страницу (т. е. переключаться между страницами):

```
PageControl1->ActivePage=TabSheet5;
```

- `RaggedRight`, равное `true`, сжимает поля с названиями страниц и делает их более компактными.
- `TabPosition` определяет, в какой части компонента будут появляться вкладки: в верхней (`Top`), нижней (`Bottom`), слева (`Left`) или справа (`Right`) . Проверьте сами.

- `ScrollOpposite` задает, как при многострочном режиме (когда все вкладки видны) будут прокручиваться вкладки. Когда `ScrollOpposite` имеет значение `false`, то все идет в обычном порядке: все вкладки после их переключения остаются на той же стороне компонента, на которой они находились и до переключения. Если свойству `ScrollOpposite` присвоено значение `true`, а свойству `TabPosition` — `Top` или `Bottom`, то ничего не меняется. Но если значение `ScrollOpposite` — `true`, а значение `TabPosition` — `Left` или `Right`, картина получается иная: все вкладки, которые находятся слева (справа) от переключаемой вкладки, перемещаются на противоположную сторону компонента.
- `ActivePageIndex` — значение этого свойства представляет собой индекс конкретной страницы. Множество страниц компонента определено в массиве `Page[]` — другом свойстве компонента. Значение `ActivePageIndex` всегда указывает на активную страницу. Но с помощью этого свойства можно и активизировать страницу, если задать индекс необходимой страницы в свойстве `Pages`: изменение значения `ActivePageIndex` изменяет значение свойства `ActivePage`, т. е. активной становится именно та страница, индекс которой вы задали с помощью `ActivePageIndex`. Если активных страниц в компоненте нет, то свойство `ActivePageIndex` имеет значение −1. Если установить `ActivePageIndex` вне границ, определенных количеством заданных страниц (в том числе и меньше нуля), то компонент не будет иметь ни одной активной страницы. Это есть средство деактивизации всех страниц.
- `Pages` — это массив страниц: описан как указатель на класс `TTabSheet`. С помощью этого свойства можно добраться до любой страницы, задав ее номер (счет начинается с нуля). Например, для пятой страницы получим: `PageControl1->Pages[4]`. Отметим, что любая страница компонента — это отдельный объект класса `TTabSheet` со своими свойствами, событиями и методами, которые позволяют управлять страницей.
- `PageCount` — в этом свойстве содержится число страниц компонента `TPageControl`. Если свойство `MultiLine` имеет зна-

чение `false`, то не все страницы могут иметь видимые вкладки, поэтому надо учитывать, что значение `PageCount` отражает полное количество страниц: видимых и невидимых.

- `Canvas` — дает доступ к битовой карте компонента, на которой можно рисовать, обрабатывая событие `OnDrawTab`.

Некоторые события *TPageControl*

Из событий компонента отметим событие `OnChange`. Оно возникает, когда выбрана новая вкладка (страница). Можно использовать свойство `TabIndex`, чтобы определить, какая страница выбрана. Перед тем как значение `TabIndex` изменится, возникает событие `OnChanging`.

Компонент *TOpenDialog*

Этот компонент предназначен для выбора файлов. Он предоставляет пользователю возможность продвигаться к файлам от каталога к каталогу в диалоговом окне, обычном для Windows, и выбирать необходимые файлы. Компонент выводит на экран стандартное окно Windows для выбора и открытия файлов. Диалоговое окно появляется в режиме исполнения приложения в момент выполнения метода `Execute()`. Когда пользователь нажимает на кнопку **Открыть** (или **Open** — это зависит от установленной у вас версии Windows) в диалоговом окне, метод `Execute()` возвращает значение `true`, окно закрывается и имя выбранного файла помещается в свойство `FileName` компонента.

Чтобы запустить диалоговое окно выбора некоторого файла, надо поместить в форму кнопку и компонент `TOpenDialog`, а в обработчике кнопки записать:

```
if(OpenDialog1->Execute());
```

После выполнения этого оператора откроется стандартное диалоговое окно Windows для выбора файла. С помощью этого окна вы выбираете нужный файл, нажимаете на кнопку **Открыть**, и имя выбранного файла оказывается в свойстве `FileName`. Теперь вы можете работать с этим файлом. Например, загрузить его в

Memo-поле. Для этого поместите в форму компонент `TMemo` и выполните команду:

```
Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
```

Для ускорения процесса выбора файла можно заранее задать начальный каталог в свойстве `InitialDir`. Этот каталог активизируется при первом обращении к методу `Execute()`. Можно также задать, какие файлы должны быть видимы в списке файлов. Это задается в свойстве `Filter`. В поле этого свойства имеется кнопка с многоточием, при нажатии на которую открывается диалоговое окно **Filter Editor**, в котором задаются имена фильтров и сами фильтры (рис. 11.24). Каждая строка, указанная в окне **Filter Editor**, попадает в свойство `Filter`. Каждая группа данных в свойстве `Filter` отделена от другой знаком "ИЛИ" (`|`). Например, если окно **Filter Editor** выглядит так, как показано на рис. 11.24, то свойство `Filter` будет содержать такую информацию:

```
aaaa|*.txt|bbbb|*.doc
```

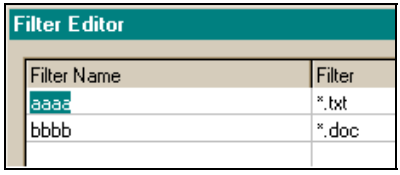


Рис. 11.24. Окно **Filter Editor**

Теперь в диалоговом окне для выбора файлов будут видимы только заданные в фильтре типы файлов.

Чтобы включить несколько масок фильтров в один фильтр, надо разделять маски точкой с запятой. Например, чтобы задать фильтры в режиме исполнения приложения, надо записать:

```
OpenDialog1->Filter="Pascal files|*.pas;*.dpk;*.dpr";
```

Если в свойстве `Filter` маска фильтрации не указана, то в диалоговом окне выбора файлов будут появляться все файлы, хранящиеся в открываемых каталогах.

Некоторые свойства *TOpenDialog*

- Свойство `Options` определяет появление и поведение компонента. Некоторые из значений этого свойства приведены в табл. 11.2.

Таблица 11.2. *Некоторые значения свойства* `Options`

Значение	Пояснение
`ofReadOnly`	Файл будет открыт только для чтения
`ofOverwritePrompt`	Если пользователь выбирает файл, который уже открыт, то появляется сообщение, предлагающее перекрыть существующий файл (этот режим надо использовать для компонентов `TSaveDialog` и `TSavePictureDialog`, которые мы рассмотрим ниже)
`ofHideReadOnly`	Отменяет режим "Только для чтения"
`ofNoChangeDir`	После того как пользователь нажмет **OK**, восстанавливается каталог, который был текущим до открытия диалогового окна выбора файла (этот режим надо использовать для компонентов `TSaveDialog` и `SavePictureDialog`, которые мы рассмотрим ниже)
`ofShowHelp`	В диалоговое окно выводится кнопка помощи **Справка**
`ofNoValidate`	Позволяет выбирать файлы со специальными символами (например, с символом &)
`ofAllowMultiSelect`	Позволяет одновременно выбрать несколько файлов
`ofPathMustExist`	Выводит сообщение об ошибке, если пользователь задает путь к несуществующему каталогу
`ofFileMustExist`	Выводит сообщение об ошибке, если пользователь пытается выбрать несуществующий файл (применяется только в диалогах открытия файла)

Таблица 11.2 *(окончание)*

Значение	Пояснение
`ofCreatePrompt`	Если пользователь пытается выбрать несуществующий файл, то появляется сообщение с предложением создать новый файл с указанным именем (этот режим надо использовать для компонентов `TSaveDialog` и `TSavePictureDialog`, которые мы рассмотрим ниже)
`OfShareAware`	Игнорируются ошибки разделения файлов. Позволяет выбирать файлы, даже если нарушается принцип разделения файлов

- `Files` — сюда попадает список имен выбранных файлов. `Files` — экземпляр класса `TStrings`, который содержит каждое выбранное имя файла и полный путь к нему (чтобы дать возможность пользователю выбрать много файлов, надо свойству `Options` присвоить значение `ofAllowMultiSelect`).
- `DefaultExt` — здесь задается расширение файла по умолчанию. Это расширение автоматически добавляется к имени выбранного файла, заменяя его собственное расширение. Если выбирается файл с незарегистрированным расширением, значение `DefaultExt` добавляется к незарегистрированному расширению. Расширения, состоящие более чем из трех символов, не поддерживаются. Нельзя включать в расширение символ "точка".
- `FileIndex` — это свойство определяет фильтр, который по умолчанию будет применен к списку файлов в диалоговом окне (от фильтра зависит, какие типы файлов будут показаны). Если присвоить свойству `FilterIndex` значение 1, то по умолчанию будет выбран первый фильтр в списке фильтров, если `FilterIndex` присвоить значение 2 — второй фильтр и т. д. Если значение `FilterIndex` выходит за пределы допустимого (не задано столько фильтров), то будет установлен первый фильтр из списка фильтров.
- `Title` — задает заголовок диалогового окна. Если заголовок не задан, то по умолчанию окно будет названо **Открытие**.

Некоторые события *TOpenDialog*

- `OnCanClose` — возникает тогда, когда пользователь пытается закрыть диалог без применения кнопки **Отмена**.
- `OnClose` — возникает при закрытии диалога.
- `OnFolderChange` — событие возникает, когда пользователь меняет каталог, содержимое которого показано в диалоговом окне (т. е. при переходе от каталога к каталогу).
- `OnIncludeItem` — событие возникает перед тем, как добавляется новый файл в список выбранных файлов (речь идет об именах файлов, естественно). Этим обстоятельством можно воспользоваться, чтобы проконтролировать занесение имен выбранных файлов в список. Описание параметров функции-обработчика этого события такое:

  ```
  OpenDialog1IncludeItem(const TOFNotifyEx &OFN, bool &Include).
  ```

  Параметр `OFN` — это Windows-структура, которая содержит в себе информацию об оболочке каталога. Если параметру `Include` присвоить значение `false`, то имя файла в список не добавится. Это событие не наступит, если значение свойства `Options` равно `ofEnableIncludeNotify`.
- `OnSelectionChange` — это событие наступает, когда пользователь что-то меняет в диалоговом окне (например, тут же создает новый каталог или выделяет файл или каталог).
- `OnShow` — событие возникает, когда открывается диалоговое окно.
- `OnTypeChange` — это событие возникает, когда пользователь выбирает новый фильтр в поле **Типы файлов**, расположенном в нижней части диалогового окна.

Компонент *TSaveDialog*

С помощью этого компонента можно сохранять файл в нужном месте файловой структуры. Свойства и события у этого компонента аналогичны соответствующим атрибутам компонента `TOpenDialog`. Чтобы сохранить файл с помощью `TSaveDialog`,

надо выполнить метод `Execute()` этого компонента. Выполнение метода вызывает открытие стандартного диалогового окна Windows для сохранения файла

Когда пользователь выбирает имя файла и нажимает на кнопку **Сохранить** (или **Save** — в зависимости от версии Windows) в диалоговом окне, метод `Execute()` возвращает значение `true`, окно закрывается и в свойство `FileName` компонента заносится имя файла и путь к нему. Никакой перезаписи файла не происходит. Отсюда следует, что для записи файла в необходимое место файловой структуры нужно применять методы сохранения файла.

Приведем пример обработчика кнопки, при нажатии которой некий строковый файл читается и переписывается в другое место под именем, выбранным нами в диалоговом окне **SaveDialog** (листинг 11.12).

Листинг 11.12

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
if(OpenDialog1->Execute())
 Memo1->Lines->LoadFromFile(OpenDialog1->FileName);
if(SaveDialog1->Execute())
 Memo1->Lines->SaveToFile(SaveDialog1->FileName);
}
```

Здесь в функции `OpenDialog1` выбирается файл для копирования (такой, чтобы к нему можно было применить метод `LoadFromFile()`). Когда диалоговое окно закроется после нажатия кнопки **Открыть**, метод `Execute()` возвратит значение `true`, и в свойство `FileName` диалогового компонента будет помещено имя выбранного файла и путь к нему. Так как метод `Execute()` возвратил значение `true`, то начинает выполняться тело оператора `if`, в котором происходит загрузка выбранного файла в Memo-поле (оно играет роль буфера). После этого начинает выполняться функция `SaveDialog1`. Она открывает окно для выбора имени файла, под которым надо записать содержимое Memo-

поля, и место, куда этот файл записать. Когда окно диалога закроется по кнопке **Сохранить**, заданное имя файла попадет в свойство `FileName` компонента `SaveDialog1`, и метод `Execute()` возвратит значение `true`. На этом роль функции `SaveDialog1` заканчивается.

Теперь мы должны сами организовать запись файла по выбранному месту и имени. Мы это и делаем: т. к. `Execute()` возвратил значение `true`, тело второго оператора `if` начинает выполняться. В теле `if` метод `SaveToFile()` сохраняет файл с именем, которое мы "добыли" в диалоговом окне (имя будет содержать и весь путь к файлу).

Компонент *TOpenPictureDialog*

Этот компонент обеспечивает диалоговый выбор и открытие файлов изображений с расширениями: bmp (битовые изображения), jpg (сжатые фотоснимки), ico (значки) и wmf, emf (метафайлы). Функции этого компонента совпадают с функциями компонента `TOpenDialog`, но в окне просмотра файлов справа появляется вертикальная полоса предварительного просмотра выбранного изображения.

Компонент *TSavePictureDialog*

Это аналог `TSaveDialog`, но, в отличие от последнего, `TSavePictureDialog` позволяет просматривать изображения в поле предварительного просмотра.

Компонент *TFontDialog*

Компонент обеспечивает выбор шрифта в диалоговом режиме: открывает диалоговое окно выбора шрифтов и их атрибутов (стиля, размера, цвета и т. п.).

После выбора нужного шрифта его название попадает в свойство `Font` компонента. Свойство `Device` определяет устройство, для которого устанавливается список доступных шрифтов (экран или принтер). В этом компоненте, как и в предыдущих диалого-

вых компонентах, действует метод `Execute()`. Например, для установки шрифта панели следует написать:

```
if(FontDialog1->Execute())
 Panel1->Font=FontDialog1->Font;
```

Некоторые свойства *TFontDialog*

Из свойств компонента рассмотрим свойство `Options`. Оно содержит в себе множество свойств, значения которых в совокупности и определяют все свойство в целом.

- `fdAnsiOnly` — в диалоге выводятся только те шрифты, которые используют набор символов Windows. Эти шрифты не видны в диалоге.
- `fdApplyButton` — в диалоге выводится кнопка **Apply** (Применить) независимо от наличия обработчика события `OnApply`.
- `fdEffects` — в диалоге выводятся переключатель эффектов (зачеркнутый и подчеркнутый) и список цветов.
- `FdFixedPitchOnly` — в диалоге выводятся только шрифты фиксированной ширины. Пропорционально изменяющиеся шрифты не выводятся.
- `fdForceFontExist` — этот атрибут позволяет пользователю вводить только те шрифты, которые появляются в диалоговом окне, в комбинированном списке шрифтов. Если пользователь выберет другой шрифт и попытается его ввести, появится сообщение об ошибке.
- `fdLimitSize` — разрешается доступ к свойствам `MaxFontSize` (максимальный размер шрифта) и `MinFontSize` (минимальный размер шрифта), тем самым дается возможность в диалоге ограничивать размеры шрифтов.
- `FdNoFaceSel` — в поле **Шрифты** диалогового окна не появляется заранее определенный шрифт: его надо выбрать самому из списка шрифтов.
- `FdNoOEMFonts` — удаляет OEM-шрифты (шрифты, зависящие от кодовых страниц операционных систем) из списка шриф-

тов в диалоговом окне. Выводятся все шрифты, кроме шрифтов OEM.

- `FdScalableOnly` — в диалоге выводятся только масштабируемые шрифты. Немасштабируемые (растровые) шрифты из списка вывода удаляются.
- `FdNoSimulations` — в диалоговом окне выводятся только те шрифты и стили, которые напрямую поддерживаются файлом определения шрифтов. Шрифты полужирного и курсивного начертания не выводятся в диалоге.
- `FdNoStyleSel` — в диалоговом окне в поле **Стиль шрифта** не появляется выбранный стиль шрифта.
- `fdShowHelp` — в диалоговом окне появляется кнопка **Справка**.
- `fdTrueTypeOnly` — в диалоге появляются только шрифты TrueType.
- `fdWysiwyg` — в диалоге появляются только шрифты, доступные как для экранного отображения, так и для распечатки на принтере. Шрифты, определенные только для одного устройства, не появляются в списке.

По умолчанию все режимы, кроме `fdEffects`, выключены.

Некоторые события *TFontDialog*

- `OnApply` — возникает, когда пользователь нажимает кнопку `Apply` в диалоговом окне.
- `OnClose` — возникает тогда, когда диалоговое окно закрывается.
- `OnShow` — возникает тогда, когда диалоговое окно открывается.

Компонент *TColorDialog*

Компонент делает возможным выбор цвета в диалоговом окне. Он работает в диалоге точно так же, как и остальные диалоговые компоненты: после выбора цвета (диалоговое окно открывается с помощью метода `Execute()` название цвета попадает в свойство `Color` компонента, и цвет может использоваться в дальнейшем).

Например, чтобы изменить цвет панели с помощью диалогового окна выбора цвета, надо написать:

```
if(ColorDialog1->Execute())
Panel1->Color=ColorDialog1->Color;
```

Некоторые свойства *TColorDialog*

- `CustomColors` — определяет цвета, заданные самим пользователем, которые будут доступны в диалоговом окне. Каждый цвет должен быть представлен в виде строки в формате `ColorX = HexValue` (шестнадцатеричное значение цвета). Например, `ColorA` = 808022. Можно задать до 16-ти цветов (от `ColorA` до `ColorP`). Задание происходит в окне Редактора, которое открывается нажатием кнопки с многоточием в поле этого свойства (рис. 11.25).

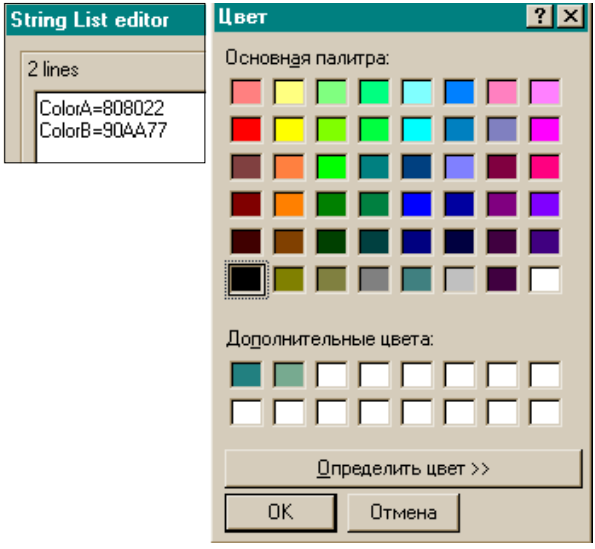


Рис. 11.25. Окно Редактора для задания цвета и выбранные цвета

Выбранные пользователем цвета появляются в разделе **Дополнительные цвета** диалогового окна (рис. 11.25).

- `Options` — это составное свойство, определяемое значениями подсвойств:
  - `CdFullOpen` — когда диалог открывается, выводятся все пользовательские режимы;
  - `CdPreventFullOpen` — запрещает пользователю определять дополнительные цвета;
  - `CdShowHelp` — добавляет в диалоговое окно кнопку **Справка**;
  - `cdSolidColor` — заставляет Windows использовать ближайший к выбранному цвет основной палитры;
  - `CdAnyColor` — позволяет пользователю выбирать дополнительные цвета (которые могут иметь приближения с помощью передачи полутонов).

По умолчанию все эти режимы выключены.

События *TColorDialog*

- `OnClose` — возникает, когда диалог закрывается.
- `OnShow` — возникает, когда диалог открывается

Компонент *TPrintDialog*

С помощью этого компонента в диалоговом режиме можно послать задание принтеру, выполнив, как и в предыдущих диалоговых компонентах, метод `Execute()`. Этот метод открывает диалоговое окно **Печать**, возвращает `true`, если пользователь нажимает **OK**, и `false` — если **Cancel**.

Свойства *TPrintDialog*

- `Collate` — показывает, выбран ли переключатель **Проверить**. Чтобы заставить диалог открыться с выбранным переключателем, надо установить `Collate` в `true`.

- `Copies` — показывает количество копий, определенное в диалоговом окне **Печать**. Это количество появляется в поле **Количество копий**.
- `FromPage` — показывает страницу, с которой начнется печать.
- `MaxPage`, `MinPage` — диапазон печатаемых страниц.
- Свойство `Options` имеет несколько подсвойств, которые задают режимы печати.

Компонент *TPrinterSetupDialog*

Этот компонент открывает диалоговое окно для задания конфигурации принтеров.

У компонента есть метод `Execute()`, который открывает диалог для установки принтера. Когда пользователь нажимает **OK**, метод `Execute()` заканчивает конфигурацию принтера, заданную в диалоге, и возвращает `true`. Если пользователь нажимает **Cancel**, метод `Execute()` возвращает `false`.

OLE-объекты

OLE — это контейнер, куда могут помещаться отдельные программы, которые затем могут обмениваться данными между собой. Например, в этот контейнер можно поместить документ программы Word и таблицу программы Excel, данные из таблицы передавать в документ Word. Или можно просто работать из вашего приложения в Word или Excel, вызвав их на выполнение в контейнер OLE. Обмен данными происходит через динамически отводимую память и не требует знания формата принимаемых данных. Компонент организует связь с объектами, которые в него помещаются двумя способами: либо напрямую, когда объект помещается непосредственно в контейнер OLE, либо косвенно, по ссылке на связываемый с OLE объект. Программа, которая передает данные, называется сервером, а которая принимает — контейнером. Включение объекта в контейнер можно провести на стадии проектирования вашего приложения, открыв контекстное меню компонента и выбрав там команду **Insert Object**.

Некоторые свойства OLE-контейнера

- `Align` — задает, как компонент выравнивается относительно контейнера, в который он помещен.
- `AllowInPlace` — показывает, как будет размещен вставленный в OLE объект. Если свойство `AllowInPlace` имеет значение `true`, а свойство `Iconic` — `false`, то OLE-объект будет активизирован на месте, в которое он помещен. Если `AllowInPlace` имеет значение `false`, OLE-объект будет активизирован в отдельном окне. Значение по умолчанию — `true`.
- `AutoActivate` — определяет, как активизируется объект в OLE-контейнере. Значения этого свойства приведены ниже.
  - `aaManual` — OLE-объект должен быть активизирован программно, вызовом метода `DoVerb()` с фактическим параметром `ovShow` (`DoVerb(ovShow)`) или с параметром `ovPrimary` (`DoVerb(ovPrimary)`). Пример активного OLE-объекта показан на рис. 11.26.
  - Свойство `AutoActivate` установлено в `aaManual`, и поэтому объект не активизируется при двойном щелчке на нем, как это происходит по умолчанию. Объект активизируется с помощью кнопок, обработчики которых показаны в листинге 11.13.

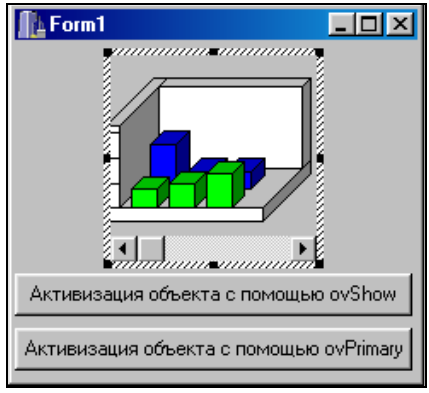


Рис. 11.26. Программная активизация OLE-объекта

Листинг 11.13

```
//---------------------------------------------------
//cpp-файл

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------

void __fastcall TForm1::Button1Click(TObject *Sender)
{
OLEContainer1->DoVerb(ovShow);
}
//---------------------------------------------------

void __fastcall TForm1::Button2Click(TObject *Sender)
{
 OleContainer1->DoVerb(ovPrimary);
}
//---------------------------------------------------

//h-файл

//---------------------------------------------------
#ifndef Unit1H
```

```
#define Unit1H
//---------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <OLECtnrs.hpp>
//---------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
    TOLEContainer *OLEContainer1;
    TButton *Button1;
    TButton *Button2;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall Button2Click(TObject *Sender);
private:   // User declarations
public:   // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------
#endif
```

`DoVerb()` требует, чтобы OLE-объект выполнил некое действие из списка действий, указанных в свойстве `ObjectVerbs` компонента OLE. Сам компонент OLE определяет некоторые действия, такие как `ovShow` (вывести OLE-объект) и `ovPrimary` (действие по умолчанию, которое обычно активизирует OLE-объект). OLE-объекты, помещенные в OLE-контейнер, могут определять свои собственные действия. Эти действия указываются в свойстве `ObjectVerbs` компонента OLE. Чтобы увидеть, какие действия заложены в свойство `ObjectVerbs` для данного OLE-объекта, надо выполнить операторы

```
AnsiString s; s=OLEContainer->ObjectVerbs->Strings[i];
```

Эти же действия появляются в контекстном меню, если его активизировать правой кнопкой мыши на OLE-объекте в режиме исполнения. Само же меню формируется, если значение свойства `AutoVerbMenu` = `true`.

- `aaGetFocus` — OLE-объект активизируется, когда OLE-контейнер получает фокус (когда на нем щелкают мышкой или нажимают клавишу <Tab>).
- `aaDoubleClick` (по умолчанию) — OLE-объект активизируется двойным щелчком на нем или нажатием <Enter>, когда контейнер имеет фокус.

- `AutoVerbMenu` — определяет, создает ли OLE-контейнер автоматически контекстное меню, содержащее список возможных действий с OLE-объектами. Если `AutoVerbMenu` имеет значение `true` (по умолчанию), OLE-объект заменяет значение свойства `PopupMenu`, в котором могло быть указано некоторое контекстное меню, на свое меню. Если значение `AutoVerbMenu` = `false`, контекстное меню не создается автоматически.
- `CopyOnSave` — в этом свойстве определяется, будут ли методы `SaveToFile()` и `SaveToStream()` создавать копию OLE-объекта. Когда свойству `CopyOnSave` присвоено значение `true`, методы `SaveToFile()` и `SaveToStream()` создают временный OLE-объект, удаляя избыточные данные в сохраняемом пространстве. Эта сжатая копия сохраняется затем в файле или в потоке. Когда же свойство `CopyOnSave` имеет значение `false`, OLE-объект не сохраняется в сжатом виде в файле или потоке. Присвоение свойству `CopyOnSave` значения `false` выгодно, когда OLE-контейнер содержит много объектов, и при сохранении временной копии могут возникнуть проблемы с памятью. По умолчанию это свойство имеет значение `true`.
- `Iconic` — определяет, будет ли OLE-контейнер выводить значок серверного приложения (т. е. помещаемого в него объекта), или изображение самого объекта. Например, если в OLE поместить программу WordPad, то при значении `true` свойства `Iconic` в поле OLE появится значок программы WordPad. Чтобы открыть программу в режиме проектирования, надо

воспользоваться контекстным меню, а при значении `false` свойства `Iconic` в поле OLE откроется сама программа WordPad.

- `SizeMode` — свойство определяет размеры OLE-объекта в OLE-контейнере. Ниже приведены возможные значения этого свойства:
  - `SmClip` (по умолчанию) — выводится только та часть OLE-объекта, которая помещается в OLE-контейнер.
  - `smCenter` — также выводится только та часть OLE-объекта, которая помещается в контейнере, но внутри контейнера изображение располагается по центру.
  - `smScale` — OLE-объект пропорционально масштабируется внутри контейнера.
  - `smStretch` — OLE-объект заполняет весь контейнер, сжимаясь или растягиваясь при необходимости, причем пропорции объекта при этом могут меняться.
  - `smAutoSize` — OLE-объект выводится весь, без изменения размеров, а контейнер принимает размеры OLE-объекта.

Выбор объекта для вставки в контейнер

Если мы поместим контейнер в форму, откроем его контекстное меню и выполним команду **Insert Object**, откроется диалоговое окно для выбора объекта.

Из появившегося списка и выбирается объект. После того как объект выбран, помещен в контейнер, и свойства контейнера определены, можно использовать контекстное меню объекта для его изменения, удаления, замены и т. д.

Компонент *TUpDown*

Эти компоненты представляют собой *счетчики*. С помощью `TUpDown` создаются спаренные кнопки, с помощью которых прокручивают некоторый компонент, который связывается с данными. Имя прокручиваемого компонента указывают в свойстве

Associate. Если щелкнуть на кнопке, то свойство Position увеличится (уменьшится) на единицу, в зависимости от того, какую кнопку нажали: со стрелкой вверх или со стрелкой вниз. Причем значение свойства Position изменяется само по себе в момент нажатия кнопки, поэтому нет необходимости изменять его в обработчике. Если, например, связать TUpDown с TEdit, то в окне последнего станет отражаться значение свойства Position (в текстовом виде).

Некоторые свойства *TUpDown*

- ArrowKes — определяет, действуют ли на компонент клавиши со стрелками вверх и вниз. Если свойство ArrowKeys имеет значение true, компонент отвечает на нажатие клавиш со стрелками, даже если связанный с ним другой компонент имеет фокус ввода. Когда ArrowKeys имеет значение false, пользователь должен нажимать кнопку мыши.

  Примечание

  Компонент может отвечать на нажатие только двух определенных выше клавиш со стрелками (вверх и вниз), даже если ориентация стрелок компонента — горизонтальная (значение свойства Orientation = udHorizontal. По умолчанию ориентация — вертикальная).

- Increment — определяет величину приращения свойства Position при каждом нажатии на кнопку компонента. Значение Position меняется в пределах, заданных в свойствах Min и Max.
- Thousands — определяет, появляется ли разделитель тысяч, когда счетчик превысит тысячную отметку.
- Wrap — если это свойство установлено в true, то в ситуации, когда пользователь пытается придать свойству Position значение, большее чем значение Max, свойству Position присваивается значение Min. Если пользователь пытается задать значение Position, меньшее чем Min, то свойство Position принимает значение свойства Max.

Компонент *TTimer*

Этот компонент задает счетчик времени. Свойство `Enabled` управляет запуском и остановкой таймера. Свойство `Interval` задает промежуток времени, через который возникает событие `OnTimer`. При использовании обработчика события `OnTimer` следует учитывать, что после возникновения события `OnTimer` новое событие не возникает, пока не выполнятся все команды обработчика. Как только все команды обработчика завершены, событие возникает не позднее, чем через интервал времени, заданный в свойстве `Interval`.

`TTimer` — удобное средство для организации процессов, автоматически повторяющихся через равные интервалы времени. Например, вы хотите, чтобы на экране вашего компьютера происходило движение различных окрашенных линий. Вставьте в обработчик события `OnTimer` формирование таких линий и запустите это приложение. Пока ваш компьютер будет включен (или пока вы не отключите таймер с помощью кнопки), его экран будет сверкать разноцветными линиями (рис. 11.27).

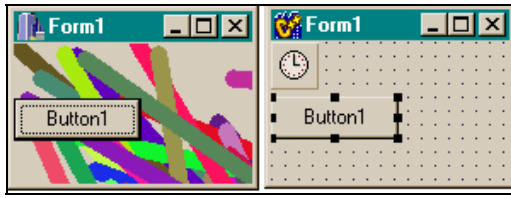


Рис. 11.27. Автоматическая разрисовка экрана

Текст программного модуля приводится в листинге 11.14.

Листинг 11.14

```
//cpp-файл
//---------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
```

```
#include "Unit1.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
   Canvas->Pen->Width=10;
   Canvas->Pen->Color=random(16000000);
   Canvas->Brush->Color=random(16000000);
   Canvas->LineTo(random(1200),random(1200));
   Canvas->MoveTo(random(1200),random(1200));
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
if(!Timer1->Enabled)
 Timer1->Enabled=true;
else
 Timer1->Enabled=false;
}
//---------------------------------
```

Компонент *TProgressBar*

Этот компонент создает индикатор некоторого процесса, благодаря чему можно наблюдать ход процесса во времени. Прямоугольный индикатор при достаточно длительном процессе по-

степенно заполняется символом-заполнителем слева направо, причем заполнение завершается с окончанием самого процесса. Это заполнение организовано с помощью свойств и методов компонента `TProgressBar`. Свойства `Min`, `Max` задают интервал значений индикатора. Свойство `Position` (в отличие от `UpDown`, в котором это свойство меняется автоматически, его надо изменять самому) определяет текущую позицию индикатора внутри интервала `Min`—`Max`. Метод `StepIt()` вызывает изменение свойства `Position` на величину свойства `Step`. Метод `StepBy(n)` вызывает изменение свойства `Position` на величину *n*.

Чтобы организовать работу `TProgressBar` по отображению хода процесса, надо использовать компонент `TTimer`: включить счетчик времени до начала процесса (`Timer1->Enabled=true;`), установить значение свойства `Position` компонента `TProgressBar` в ноль (`ProgressBar1->Position=0;`), а в обработчике события `OnTimer` наращивать значение `Position` (`ProgressBar1->Position++;`). После окончания контролируемого процесса надо выключить таймер и скрыть сам индикатор (`ProgressBar1->Visible=false;`).

Приведем пример совместной работы `TProgressBar` и `TTimer`. Формы в режиме проектирования и исполнения показаны на рис. 11.28.

Текст обработчиков событий приведен в листинге 11.15.

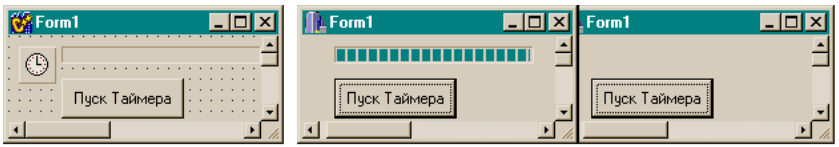


Рис. 11.28. Совместная работа `TProgressBar` и `TTimer`

Листинг 11.15

```
void __fastcall TForm1::Timer1Timer(TObject *Sender)
{
```

```
 for (int i=ProgressBar1->Min; i <= ProgressBar1->Max; i++)
  ProgressBar1->Position=i;
 Timer1->Enabled=false; //выключить таймер
 ProgressBar1->Position=0;
 ProgressBar1->Visible=false;
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
 Timer1->Enabled=true; //запустить таймер
 ProgressBar1->Visible=true;
}
//---------------------------------------------------------
```

Компонент *TDateTimePicker*

Этот компонент обеспечивает возможность ввода даты или времени (это регулируется заданием свойства `Kind`) в редактируемое прямоугольное поле. Введенная дата в формате `TDateTime` помещается в свойство `Date`, из которого ее можно извлекать и использовать по назначению. Дату можно также помещать в это свойство. Свойство `DateMode` дает возможность выбрать один из двух вариантов отображения даты: в виде одного поля со счетчиками `TUpDown` или в виде компонента наподобие `TComboBox`. В отличие от настоящего `TComboBox`, здесь при щелчке на кнопке в редактируемом поле выпадает не список строк, а экземпляр класса `TMonthCalendar` — календарь с указанием текущей даты.

Календарь можно перелистывать кнопками вперед и назад (в сторону увеличения или уменьшения даты), что позволяет легко подбирать необходимую дату. Как только вы щелкнете на выбранной дате, ее значение немедленно попадет в окно компонента `TDateTimePicker` и тут же произойдет событие `OnCloseUp`, в обработчике которого и можно работать с выбранной датой.

Некоторые свойства *TDateTimePicker*

- `CalAlignment` — определяет выравнивание выпадающего календаря относительно компонента: появится ли он слева от компонента или справа.
- `CalColors` — дает возможность выбора цветов различных элементов календаря (фона, цвета шрифта и т. д.).
- `Checked` — свойство задает, будет ли установлен флажок слева от выбранной даты (времени). Флажок появляется в зависимости от значения свойства `ShowCheckBox`. `Checked` возвращает `true`, когда флажок установлен. Установка `Checked` в `true` в режиме проектирования заставит флажок появляться автоматически.
- `Date` — здесь появляется дата, выбранная на календаре. Значение даты должно быть в интервале, заданном значениями свойств `MaxDate` и `MinDate`. Если значение свойства `MultiSelect` экземпляра календаря `TMonthCalendar` равно `true`, то в календаре можно выбрать группу дат, и значение последней даты будет находиться в свойстве `EndDate` календаря. Это дает возможность выбора группы дат и работы с ней через класс `TMonthCalendar`. Задать или получить дату можно также с помощью свойства `DateTime`.
- `DateFormat` — определяет, в каком формате будет появляться дата в поле компонента: в "коротком" виде (по два разряда на число, месяц, год) или в "длинном", когда появляется число, название месяца и четырехразрядное значение года.
- `Kind` — задает, может ли пользователь устанавливать дату или время. Если задать установку времени, то формат поля компонента автоматически отображается в виде поля со счетчиками, и к календарю доступа нет.
- `Time` — указывает время, введенное пользователем (если свойство `Kind` позволяет задавать время).
- `ParseInput` — разрешает возникновение события `OnUserInput`. Это событие возникает, когда пользователь напрямую вводит данные в редактируемое поле (это возможно, если `ParseInput` = `true`), что позволяет контролировать вводимые данные в обработчике этого события.

О работе с датами

Для работы с датами в C++ Builder существует специальный класс `TDateTime`, который содержит в себе методы работы с датами. Этот класс в качестве члена данных включает в себя переменную типа `Double`, которая в своей целой части содержит дату, а в дробной — время. Точка отсчета даты — 30.12.1899 г. 12 часов 00 минут. Значение даты в это время принято считать равным нолю. Ненулевое значение даты вычисляется так: к точке отсчета прибавляется целая часть даты (т. е. количество дней) и к дробной части точки отсчета прибавляется дробная часть даты (т. е. часы и минуты, конечно, с учетом перехода времени, большего 24 часов в сутки). Чтобы увидеть функции работы с датой и временем, щелкните мышкой, например, на форме, сделав ее активной и нажмите <F1>. Появится справочная служба по компоненту `TForm`. Откройте вкладку **Разделы**. При этом откроется диалоговое окно, в котором надо открыть вкладку **Указатель** и в поле поиска набрать слово `Date`. Среди списка тем в окне будет и позиция с названием **Date/time routines** (Подпрограммы даты-времени). Щелкнув мышью на этой позиции, вы увидите список подпрограмм, работающих с датой-временем. Например, среди них будут функции:

- `DateToStr()` — преобразует дату в строку типа `AnsiString`;
- `DateTimeToStr()` — преобразует дату и время в строку типа `AnsiString`;
- `TimeToStr()` — преобразует время в строку типа `AnsiString`.

Чтобы работать с датой и временем, следует объявить переменную типа `TDateTime`. Например, `TDateTime d1,d;` А затем использовать методы этого класса, которые обнаруживаются разделом **Help** таким же способом, как и подпрограммы, которые мы только что находили. Только в поле поиска надо набрать текст `TDateTime`. Например, среди методов этого класса будут такие:

- `CurrentDate()` — возвращает текущую дату как значение типа `TDateTime`. Значение времени возвращается равным нулю;
- `CurrenTDateTime()` — возвращает текущую дату и время как значение типа `TDateTime`;

- `CurrentTime()` — возвращает текущее время как значение типа `TDateTime` с нулевым значением даты;
- `DayOfWeek()` — возвращает день недели между 1 и 7. Воскресенье считается первым днем недели, суббота — последним;
- `DecodeDate()` — разделяет значение даты типа `TDateTime` на год, месяц и число и помещает эти значения в соответствующие параметры: `year`, `month`, `day`;
- `DecodeTime()` — выделяет значения часов, минут, секунд и миллисекунд из времени даты и помещает их в соответствующие параметры `hour`, `min`, `sec`, `msec`;
- `FormatString()` — форматирует `TDateTime`-объект, используя заданные форматы (см. раздел **Help**);
- `operator int` — преобразует `TdateTime`-объект в целое число. Например:

```
TDateTime df;
int i =df.CurrenTDateTime().operator int();
```

- `operator` — с его помощью можно из значения даты (времени), т. е. из объекта типа `TDateTime`, вычитать: даты, числа типа `double`, числа типа `int`.

Примеры использования дат

Пример 1

Приведем пример использования дат. В форму помещена кнопка, в обработчике которой и выполняются действия с рассмотренными методами. Код обработчика кнопки представлен в листинге 11.16.

Листинг 11.16

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TDateTime d1,d;
d1=d.CurrentDate();
```

```
AnsiString s = DateToStr(d1);
ShowMessage(s);
//-------------------------
d1=d.CurrentDateTime();
s = DateTimeToStr(d1);
ShowMessage(s);
//-------------------------
d1=d.CurrentTime();
s = TimeToStr(d1);
ShowMessage(s);
//-------------------------
}
```

Пример 2

В этом примере текущая дата разбивается на год, месяц и число, которые и выводятся на экран. Код обработчика кнопки, расположенного в форме, приведен в листинге 11.17.

Листинг 11.17

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TDateTime dt,d;
dt=d.CurrentDate();
Word y1,m1,d1;
DecodeDate(dt,y1,m1,d1);
AnsiString s=IntToStr(y1)+
 "**" + IntToStr(m1)+
 "**" + IntToStr(d1);
ShowMessage(s);
//-------------------------
Word ms;
dt=d.CurrentTime();
```

```
DecodeTime(dt,y1,m1,d1,ms);
s=IntToStr(y1)+
 "**" + IntToStr(m1)+
 "**" + IntToStr(d1);
ShowMessage(s);
}
```

Пример 3

В примере использован один из рассмотренных выше форматов. Текст обработчика кнопки в форме приведен в листинге 11.18.

Листинг 11.18

```
TDateTime df;
AnsiString ss = FormaTDateTime("d mmmm yyyy ***
h:mm",df.CurrenTDateTime());
ShowMessage(ss);
```

Пример 4

Ниже приведен пример использования формата вывода даты, задаваемого символом / (листинг 11.19).

Листинг 11.19

```
TDateTime df;
AnsiString ss = FormaTDateTime("d/ m/ yyyy/ ***
tt",df.CurrenTDateTime());
ShowMessage(ss);
```

Пример 5

Ниже показан пример вычитания из текущей даты числа 5. Обработчик кнопки показан в листинге 11.20.

Листинг 11.20

```
TDateTime df;
df= df.CurrenTDateTime().operator-(5);
 ss = FormaTDateTime("d/ m/ yyyy/ *** tt",df);
 ShowMessage(ss);
```

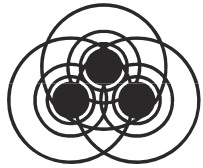
Пример 6

Приведен пример вычитания дат: взята текущая дата (на период разработки примера) и из нее вычтена дата, значение которой отличается от текущей даты в меньшую сторону на три дня с минутами. Обработчик кнопки приводится в листинге 11.21.

Листинг 11.21

```
TDateTime Df1=StrToDateTime("19.10.03 15:25");
Df= Df.CurrenTDateTime().operator-(Df1);
 ss = FormaTDateTime("D/ m/ yyyy/ *** tt",Df);
 ShowMessage(ss);
```

Глава 12



Базы данных

Что такое база данных

База данных — это специальным образом организованное пространство памяти для хранения определенных групп данных. Базы данных снабжены специальным программным обеспечением, которое поддерживает хранящуюся информацию в актуальном состоянии и позволяет пользователю получать данные и добавлять их.

Под группами данных в среде Borland C++ Builder мы понимаем, в первую очередь, такие элементы, как *прямоугольные таблицы данных* и *хранимые процедуры*. Прямоугольные таблицы данных состоят из *строк* и *столбцов* данных. Определяющими в таких таблицах являются столбцы (их еще называют *полями*). Столбцы задают основные характеристики объекта, сведения о котором хранятся в таблице. Например, мы хотим хранить в таблице данные по кадрам предприятия, в частности, "Личную карточку работника". Реквизиты этого документа (объекта) и определяют поля (столбцы) будущей таблицы: фамилия, имя, отчество, год рождения, должность, категория работника, оклад, дата поступления на работу и т. д. Строки же таблицы будут отражать данные о конкретном работнике. Строки таблиц еще называют *записями*. Далее мы будем пользоваться данными контрольного примера, поставляемого вместе со средой Borland C++ Builder, в частности, базой данных (БД) BCDEMOS, содержащей ряд таблиц.

Базы данных бывают *локальные*, расположенные на вашем же компьютере, и *удаленные*, которые расположены на других компьютерах, соединенных одной сетью с вашим компьютером.

Компьютеры, на которых располагаются удаленные базы данных, называются *серверами*, а ваш компьютер по отношению к таким базам выступает в роли *клиента*. Базы данных могут быть самых разных типов, поскольку они поддерживают различные структуры таблиц, которые содержат данные. Каждая база данных имеет свой *механизм ведения базы данных*, т. е. механизм поддержки информации в актуальном состоянии. Примерами баз данных, которые могут быть локальными, являются базы данных типа Paradox и dBase, а примерами удаленных — InterBase, Informix, SyBase, Oracle.

Создание базы данных

Для создания баз данных и организации их взаимодействия с приложениями пользователя (клиентскими приложениями) существуют различные механизмы. Одним из таких является механизм BDE (Borland Database Engine). С его помощью можно создавать базы данных многих типов (при их создании предлагается список поддерживаемых этим механизмом баз данных). Создадим свою, локальную, базу данных, в качестве которой выберем базу данных типа Paradox (она может быть и удаленной). Следует заметить, что механизм BDE устроен так, что работа с локальной или с удаленной БД для пользователя не имеет никакой разницы. При создании БД с помощью BDE задается так называемый псевдоним (alias) БД, по которому она обнаруживается клиентским приложением как на собственном компьютере, так и на другом, удаленном. Пользователь при создании БД только связывает присваиваемый им базе данных псевдоним с путем к месту, где будет расположена сама БД: если БД локальная, то указывается путь к каталогу, где выделено место для БД, если БД удаленная, то с помощью сетевых механизмов определяется путь к удаленному серверу.

Итак, создаем базу данных. Для этого делаем следующее. Запускаем утилиту BDE Administrator (она поставляется вместе со средой Builder и находится в одном каталоге с ней). Открывается диалоговое окно с двумя вкладками: **Databases** и **Configuration**. Первая вкладка по умолчанию открыта. На ней представлены псевдонимы всех баз данных, доступных с вашего компьютера.

В первой строке находится уже знакомая нам БД BCDEMOS. Устанавливаем курсор мыши на любое поле вкладки и открываем ее контекстное меню (правой кнопкой мыши). Чтобы создать новую базу данных выбираем команду **New**. Когда мы щелкаем в поле вкладки **Databases**, подсветится ближайшая к курсору мыши строка, и справа от нее появится поле **Definition**. В этом поле задаются характеристики создаваемой БД или высвечиваются характеристики уже созданной. Одновременно с этим полем откроется диалоговое окно для выбора типа БД. По умолчанию предлагается тип STANDARD. Это и есть тип базы данных Paradox. Но можно выбрать и другие типы из раскрывающегося списка, который появится, если нажать кнопку со стрелкой.

Мы выбираем то, что предлагается по умолчанию, и нажимаем кнопку **OK**. Среди множества псевдонимов на вкладке **Databases** появляется новая строка (она подсвечена) с названием STANDARD1. В подсвеченное поле надо вписать свое название создаваемой БД, например, `MyParadox`.

После этого следует задать характеристики создаваемой БД. Это делается в поле **Definition**. Для Paradox следует определить значение параметра **PATH**. Для этого надо щелкнуть мышью в строке параметра **PATH**, после чего появится спрятанная ранее кнопка с многоточием (если щелкать мышью в других строках, то могут появиться спрятанные кнопки и для других параметров БД. Нажатие на эти кнопки дает возможность пользователю выбирать необходимые значения параметров из выпадающих списков). Если нажать на кнопку с многоточием в поле параметра **PATH**, то откроется стандартное диалоговое окно для выбора пути к размещению создаваемой БД. С помощью списка **Drivers** надо открыть папку, в которой будет располагаться БД, и нажать **OK**. Сформированный путь станет значением параметра **PATH**.

Теперь надо открыть вкладку **Configuration** окна **BDE Administrator**, а на ней последовательно выбрать **Drivers/Native/Paradox**, раскрывая, при необходимости, составные элементы с помощью нажатия на знак "плюс" слева от элемента. Когда в конце концов мы выберем Paradox, в поле **Definition** появятся новые параметры Paradox, среди которых есть параметр **LANGDRIVER** (выбор драйвера, обслуживающего язык таблиц типа Paradox). В поле этого параметра нужно щелкнуть мышью, в результате

чего появится кнопка для выпадающего списка. Если ее нажать, раскроется список драйверов. Если вы хотите, чтобы информация в таблицах вашей БД хранилась на русском языке, выберите строку **dBase RUS 866**.

Теперь следует вернуться на вкладку **Databases** и нажать на синюю изогнутую вправо стрелку (применить) либо выбрать команду **Apply** из подменю **Object** меню **BDE Administrator**. Появится окно подтверждения. Можно отказаться от дальнейших действий, нажав кнопку **Cancel**. Но если нажать **OK**, то выбранный вами псевдоним вашей новой БД будет зафиксирован. БД создана.

Создание таблицы базы данных

Таблицы создаются с помощью специальной утилиты Database Desktop, доступ к которой осуществляется из подменю **Tools** главного меню среды Builder. После запуска утилиты открывается диалоговое окно, в котором открываем меню **File**, в нем выбираем подменю **New**, а затем команду **Table**. После этого откроется диалоговое окно для выбора типа таблицы.

Выбираем Paradox 7, предлагаемый по умолчанию, и нажимаем **OK**. Откроется окно для задания полей таблицы и их характеристик. В части окна, озаглавленной **Field roster**, задаем поля таблицы.

Задание полей таблицы

Поле *Field Name*

В поле **Field Name** вводим имя поля и нажимаем <Enter>. Подсветка переместится в поле **Type**, определяющее тип данных, которые будут помещаться в это поле.

Поле *Type*

Чтобы ввести тип, надо открыть правой кнопкой мыши контекстное меню поля **Type**. В меню будут находиться все типы данных, поддерживаемые таблицами Paradox. После выбора типа из меню его аббревиатура появится в поле **Type**, после чего следует

нажать <Enter>, чтобы перейти к работе с полем **Size**. Каждый тип таблицы, в том числе и Paradox, поддерживает свой тип данных. Типы данных, которые поддерживаются в таблицах Paradox, приведены в табл. 12.1.

Таблица 12.1. *Типы данных, поддерживаемые таблицами Paradox*

Обозначение	Размер	Тип
A	1—255	*Alpha*
N	—	*Number*
$	—	*Money*
S	—	*Short*
I	—	*Long Integer*
#	0—32 (количество цифр после точки)	*BCD*
D	—	*Date*
T	—	*Time*
@	—	*Timestamp*
M	1—240 (Memo и форматируемые Memo-поля могут быть неопределенной длины)	*Memo*
F	0—240 (Memo и форматируемые Memo-поля могут быть неопределенной длины)	*Formatted Memo*
G	0—240 (длина поля произвольная)	*Graphic*
O	0—240 (длина поля произвольная)	*OLE*
L	—	*Logical*
+	—	*Autoincrement*
B	0—240 (длина поля произвольная)	*Binary*
Y	1—255	*Bytes*

Вы можете задать размер Memo-поля в диалоговом окне **Create Table** в противовес определяемым по умолчанию самой утилитой: Database Desktop отводит для Memo-полей в таблице: от 1 до 240 символов для полей Memo и от 0 до 240 символов для форматируемых полей Memo. Само Memo-поле хранится вне таблицы. Например, если вы задали размер поля свыше 45, Database Desktop размещает первые 45 символов в таблице. Само же поле целиком размещается в другом файле (с расширением mb) и извлекается по мере продвижения по записям в таблице.

Рассмотрим типы полей подробнее.

- Alpha — поле может содержать строки, состоящие из букв, чисел, специальных символов, таких как %, &, #, или =, других печатаемых символов ASCII.
- Number — поле этого типа может содержать только числа (положительные и отрицательные, целые и дробные). Диапазон чисел: от -10^{307} до 10^{308} с 15-ю значащими цифрами.

Примечание

Считается хорошим стилем использовать alfa-поля вместо числовых для телефонных номеров или почтовых кодов. В alpha-поле вы можете помещать скобки и дефисы.

- Money — поля этого типа, как и числовые, могут содержать данные только числового типа. Но по умолчанию money-поля форматируются таким образом, чтобы выводились не только десятичные цифры, но и денежные символы. Невзирая на то, сколько выводится десятичных цифр, Database Desktop ограничивается только шестью десятичными цифрами при проведении внутренних расчетов с числами в таких полях. Чтобы изменить значение по умолчанию, надо самому задать размер при создании таблицы (если нажать <Enter> после ввода типа данных, то подсветка автоматически пропустит поле **Size** и попадет на поле **Key**).
- Short — это специальные числовые поля, которые могут содержать только целые числа в диапазоне от −32 767 до 32 767. Эти поля занимают меньше памяти на диске, чем обычные number-поля. Они используются только в таблицах типа Paradox.

- Long integer — эти поля содержат целые 32-битовые со знаком числа в диапазоне от −2 147 483 648 до 2 147 483 647. Поля типа long integer требуют больше пространства для хранения, чем short-поля.
- BCD — эти поля содержат числовые данные в формате BCD (Binary Coded Decimal). Использовать этот тип данных можно в случаях, когда вам надо выполнять вычисления с более высоким уровнем точности, чем тот, который обеспечивают поля других числовых типов. Расчеты с помощью BCD-полей происходят медленнее, чем с помощью полей других числовых типов. Кроме того, BCD-поля применяются для обеспечения совместимости с другими приложениями, использующими поля такого типа. При вычислениях с BCD-данными они преобразуются в данные типа `float`, а полученный результат преобразуется обратно в BCD.

Примечание

Хотя BCD-поля вмещают большой объем данных, вы можете ввести только число не более, чем с 15-ю значащими цифрами.

- Date — поля этого типа содержат даты в пределах от 1 января 9999 г. до н. э. до 31 декабря 9999 г. н. э. Database Desktop правильно управляет переходом между годами и веками.
- Time — поля этого типа содержат время дня, которое считается в миллисекундах, начиная с полуночи и до 24-х часов.
- Timestamp — эти поля содержат дату и время одновременно. Чтобы ввести текущую дату и текущее время, следует нажимать клавишу <Пробел> до тех пор, пока Database Desktop не введет данные. Правила для этого типа полей такие же, как и для date-полей и time-полей.
- Memo — этот тип полей содержит Memo-поля — текстовые строки. Memo-поля имеют виртуальную длину. Database Desktop размещает все поле вне таблицы (в MB-файле). Database Desktop извлекает данные из MB-файла, когда вы двигаетесь по записям таблицы. Количество данных в Memo-поле ограничено только дисковым пространством вашего компьютера.

- Formatted Memo — это такие же поля, как и Memo-поля, но за тем исключением, что вы можете форматировать текст в полях (в Paradox). Paradox распознает такие атрибуты текста, как шрифт, стиль, цвет и размер, и размещает их вместе с текстом.
- Graphic — в этот тип полей можно помещать графические изображения в форматах BMP, PCX, TIF, GIF, EPS. Когда вы вставляете графическое изображение в graphic-поле, Database Desktop преобразует его в BMP-формат. Graphic-поля не требуют задания размера, т. к. они хранятся не в таблице, а в отдельных файлах.
- OLE — это поле используется для размещения в нем различных данных, таких как изображения, звук, документы и т. д. OLE-поля обеспечивают работу с этими данными внутри базы данных Paradox. Database Desktop не поддерживает OLE-данные. Размер этих полей не задается, т. к. значения полей хранятся не в таблице, а вне ее, в отдельных файлах.
- Logical — эти поля содержат логические данные `true` или `false` (регистр клавиатуры роли не играет).
- Autoincrement — этот тип полей содержит данные типа `long integer`, `read-only`. Database Desktop начинает отсчет значения поля с 1 и добавляет по единице для каждой записи в таблице. То есть если это поле задать, то при движении по таблице в этом поле будет порядковый номер текущей записи в таблице. Удаление записи не изменяет значения поля для других записей. Когда создается таблица, можно задать начальный номер отсчета этого поля, указав его минимальное значение в поле **Minimum Value**.
- Binary — этот тип полей используют "продвинутые" пользователи, работающие с данными, которые Database Desktop не может интерпретировать. Database Desktop не может выводить или интерпретировать binary-поля. Обычно binary-поля используются при работе со звуком. Binary-поля не требуют задания размера, т. к. они хранятся не в самой таблице, а в отдельных файлах (MB-файлах).
- Bytes — как и в предыдущем случае, этот тип полей для "продвинутых" пользователей, работающих с данными, которые

Database Desktop не может интерпретировать. Обычное использование этих полей — для работы со штрихкодами и магнитными лентами. Эти поля хранятся в самой таблице, обеспечивая быстрый доступ к ним.

Таковы типы полей, поддерживаемые в Paradox. Продолжим рассмотрение полей для задания структуры таблицы.

Поле *Key*

Это одно поле или группа подряд расположенных полей таблицы Paradox, используемых для упорядочения записей в таблице или для проверки целостности таблицы. Пометка поля признаком **Key** дает следующие результаты:

- таблица застрахована от появления записей-дубликатов;
- записи будут упорядочены по ключевым полям;
- для таблицы создается так называемый *первичный индекс*, который определяет, в каком порядке Database Desktop осуществляет доступ к записям таблицы. Когда для поиска необходимой записи применяют соответствующие методы поиска по ключу, при обращении к этим методам задают ключ **Key**.

Другие элементы диалогового окна для создания таблицы

Рассмотрим теперь другие элементы диалогового окна **Create Paradox Table**, позволяющего создавать таблицы данных типа Paradox.

Элементы *Table properties*

Это раскрывающийся список свойств таблицы Paradox. При нажатии мышью на каждый элемент списка открывается диалоговое окно для определения соответствующих конкретных значений свойства.

- Свойство **Validity Checks**. Оно определяет требования к заполнению поля и значения по умолчанию. Способ задания контроля зависит от данных, которые должны вводиться в

поле. Database Desktop обеспечивает пять видов контроля ввода данных в поле:

- поле должно быть обязательно заполнено;
- минимальное значение вводимых данных;
- максимальное значение вводимых данных;
- определить значение по умолчанию, которое будет появляться, если другое значение не введено в поле;
- задать шаблон ввода данных в поле.

- Элемент **Table Lookup**. Если вы укажете свойство **Table Lookup** для некоторого поля, то это поле будет содержать только те значения, которые существуют в первом поле другой таблицы, уже существующей. Эту существующую таблицу можно назвать *поисковой* (lookup). Когда будем двигаться по создаваемой таблице, то в поисковой таблице будет доступна запись, у которой значение связанного поля совпадет со значением поля нашей таблицы. То есть это тоже элемент контроля ввода данных, определяемый при создании таблицы.
- Элемент **Secondary Indexes**. Этот элемент присваивает *вторичный индекс* в списке **Field Roster** создаваемому полю. Вторичный индекс позволяет сортировать данные не так, как они определены в ключевом поле. Кроме того, этот индекс позволяет связывать таблицы. Вторичный индекс — это поле или группа полей, с помощью которых вы можете указать:
  - альтернативный порядок сортировки записей таблицы;
  - поле, по которому вы хотите связать таблицу с другой таблицей;
  - способ ускорения некоторых поисковых операций.

  Таблица может иметь более одного вторичного индекса. Вы можете определить каждое поле как вторичный индекс, что позволит сортировать таблицу по любому ее полю. Вы можете создать составной вторичный индекс, комбинируя два поля или больше. Но для полей типа Memo, Formatted Memo, Binary, OLE, Graphic, Logical или Bytes вторичный индекс создавать нельзя. Когда вы используете вторичный индекс, вы изменяете только видимый порядок записей. Их физическое

размещение в таблице остается неизменным. Таблицы Paradox содержат следующие режимы для вторичных индексов: Unique, Case-sensitive, Maintained и Ascending/Descending. Рассмотрим эти режимы.

- Режим Unique определяет, могут ли записи иметь совпадающие значения в полях, составляющих вторичный индекс. Если режим Unique включен и две или более записи имеют совпадающие значения вторичного индекса, то попытка определить вторичный индекс не состоится. Вы должны удалить дублирующие записи и повторно определить вторичный индекс. Этим он и ценен: устраняет дублирование записей.
- Case-sensitive-индексы чувствительны к регистру. При сортировке таких индексов прописные символы располагаются перед строчными символами.
- Если вторичный индекс имеет режим Maintained, то Database Desktop автоматически обновляет индекс, когда обновляется таблица. Таблица должна иметь ключ до создания вторичного индекса типа Maintained. Индексы, не имеющие режима Maintained, не обновляются автоматически при обновлении таблицы. Когда вы хотите просмотреть таблицу с такими вторичными индексами, она временно блокируется и не может обновляться.
- С помощью режима Ascending/Descending вы можете задать сортировку по возрастанию/убыванию вторичных индексов.

Примечание

Индексы, имеющие режимы Unique и Descending, могут применяться только начиная с седьмой версии таблицы Paradox, но не в более ранних версиях. Когда сортируете таблицу с ключами, то используйте вторичный индекс. Только явно определенный вторичный индекс может перекрыть первично отсортированную по ключу таблицу.

- Элемент **Referential Integrity**. С помощью этой опции списка **Table properties** проверяется не разрушена ли связь между одинаковыми данными в разных таблицах, т. е. не нарушена

ли *целостность данных*. Целостность означает, что ключевые поля в обеих таблицах для одинаковых данных составляющих их полей должны совпадать. Используя заданное свойство целостности, Database Desktop проверяет достоверность значений в обеих таблицах. Например, вы задали свойство целостности данных таблицам Customer и Orders по их общему полю CustNo. Когда вы введете значение этого поля в таблицу Orders, Database Desktop ищет значение поля CustNo в таблице Customer и, если находит, принимает значение, введенное вами в Orders. И наоборот, отвергает введенное значение, если его не существует в таблице Customer. Не для всех файлов можно задавать целостность данных. Например, вы не сможете задать требование целостности данных между DBF-файлами, таблицами Paradox 3.5, или таблицами, не имеющими ключа (в которых не определены ключевые поля). Вы можете задавать это требование для DB-файлов и некоторых SQL-серверных таблиц.

- Элемент **Password Security**. Эта опция списка **Table properties** обеспечивает задание элемента защиты вашей таблицы от неавторизованного доступа. Это особенно важно при многопользовательском доступе к таблице. Вы можете не только установить пароль для входа в таблицу в целом, но и определить специфические права на отдельные ее поля. Если вы задали пароль, то в таблицу могут войти только те пользователи, которые его знают. Это касается и вас. Поэтому не забывайте ваш пароль. Каждый раз, когда пользователь пытается получить доступ к таблице, Database Desktop подсказывает ему, чтобы он ввел пароль. Database Desktop поддерживает два типа паролей:

  - *главные пароли* контролируют доступ ко всей таблице в целом. Вы должны задать главный пароль до создания паролей дополнительного доступа;
  - *вспомогательные пароли* обеспечивают различные уровни привилегий доступа пользователей одной группы.

  Обычно делают так: назначают главный пароль для администратора базы данных. А группе пользователей, которой необходимо обращаться к таблице и выполнять с ее данными

различные задачи, назначают различные вспомогательные пароли, обеспечивающие различные уровни доступа.

- Элемент **Table Language**. Эта опция списка **Table properties** позволяет изменять язык, принятый в таблице. Если выбрать эту опцию, станет доступной кнопка **Modify**, после нажатия на которую откроется окно для выбора нового языка.

Кнопка *Borrow*

Она расположена на панели, на которой задаются поля таблицы. Кнопка **Borrow** предоставляет возможность создать структуру вашей таблицы по образцу структуры другой таблицы. Кнопка открывает диалоговое окно для выбора подходящей таблицы из доступных вам баз данных. Кнопка становится недоступной, как только вы начинаете сами определять поля таблицы. После выбора и открытия подходящей таблицы ее структура вставляется в поле окна **Create Table**. После этого вы можете добавлять свои поля или воспользоваться заимствованными. Удаление полей происходит при нажатии <Ctrl>+<Delete>.

Компоненты работы с базой данных

Основные компоненты работы с созданной нами базой данных — это компоненты `TTable`, `TDataSource`, `TDBGrid`, `TDBNavigator`, `TQuery`, `TStoredProc`. Именно с их помощью происходит запись данных в БД, поддержка их в актуальном состоянии, выбор и обработка данных.

Компонент *TTable*

Этот компонент устанавливает прямую связь с таблицей БД, псевдоним которой указывается в компоненте. Связь устанавливается с помощью механизма BDE (Borland Database Engine). `TTable` обеспечивает прямой доступ к любой записи и полю таблицы БД, независимо от того, является ли она одной из баз данных типа Paradox, dBASE, Access, FoxPro либо управляется с помощью механизма ODBC, либо некоторой SQL-БД на уда-

ленном сервере (InterBase, Oracle, Sybase, MS-SQL Server, Informix или DB2).

Механизм ODBC (Open Database Connectivity) представляет из себя набор драйверов для организации работы со специфическими базами данных, которые напрямую не поддерживает механизм BDE. При организации работы с `TTable` используются свойства, события и методы этого компонента. Рассмотрим некоторые из них.

Некоторые свойства *TTable*

- Свойство `Active`. Здесь задается, открыть ли доступ к набору данных, представляемых таблицей, или нет. Значение `Active` определяет, устанавливается ли связь с набором данных в БД. Если свойство `Active` имеет значение `false`, то набор данных закрыт: к нему нет доступа. Если `Active` = `true`, набор данных открыт и данные из него могут читаться или записываться в него. Приложение пользователя должно установить свойство `Active` в `false` перед внесением изменений в другие свойства таблицы. Вызов метода `Open()` компонента `TTable` устанавливает `Active` в `true`. Вызов метода `Close()` — в `false`.
- Свойство `AutoCalcFields`. Это свойство определяет, произойдет ли событие `OnCalcFields`. В обработчике этого события обрабатываются так называемые *вычислимые поля*. Это такие поля, которые не присутствуют в структуре таблицы, а создаются для проведения каких-то временных, промежуточных операций. Их значения вычисляются на основе "родных" полей записи таблицы, активной в момент наступления события `OnCalcFields`. Если в приложении меняются данные, событие `OnCalcFields` постоянно включается. В этих случаях приложение может установить свойство `AutoCalcFields` в `false`, чтобы снизить частоту вызова события. Когда `AutoCalcFields` = `false`, событие `OnCalcFields` не вызывается, какие бы ни происходили изменения конкретных полей внутри записи. Как создавать вычислимые поля, мы покажем при рассмотрении свойств `Fields`.
- Свойство `Bof`. Это свойство показывает, установлен ли указатель записи таблицы на начало таблицы. Свойство `Bof` имеет

значение `true`, когда таблица открывается или мы находимся на первой записи таблицы. Если мы находимся не на первой записи таблицы, свойство `Bof` устанавливается в `false`.

- Свойство `DatabaseName`. Здесь задается имя базы данных (ее псевдоним), в которой находится требуемый набор данных (в нашем случае это — таблица).
- Свойство `DefaultIndex`. Это свойство определяет, будут ли записи в таблице упорядочены по индексу, заданному по умолчанию, когда таблица открывается. Если это свойство установлено в `false`, механизм BDE не сортирует данные, когда открывает таблицу. Когда же `DefaultIndex = true`, BDE упорядочивает данные на основе первичного ключа или unique-индекса при открытии таблицы. По умолчанию `DefaultIndex = true`.
- Свойство `Eof`. Это свойство определяет, находится ли указатель записи таблицы на ее последней записи. Если `Eof = = true`, то находится, в противном случае — нет. Когда приложение открывает пустую таблицу, то свойство `Eof` имеет значение `true`. Если и свойство `Eof`, и свойство `Bof` равны `true`, таблица пуста.
- Свойство `Exclusive`. Разрешает приложению (и только ему) доступ к данным таблиц Paradox или dBase. Свойство `Exclusive`, если оно установлено в `true`, делает таблицы Paradox и dBase недоступными, пока ваше приложение открыто. При многопользовательском режиме надо учитывать эту ситуацию с захватом доступа к таблице. Если кто-то уже захватил доступ к таблице, при обращении к такой таблице вашего приложения возникнет исключительная ситуация. Поэтому при многопользовательском режиме обращение к таблице должно идти в блоке обработки исключительных ситуаций `try...catch`. Не устанавливайте свойство `Exclusive` в `true` в режиме проектирования, если вам понадобится активизировать в этом же режиме таблицу. В этом случае возникает исключительная ситуация, т. к. таблица уже находится в использовании механизма IDE (Интегрированная среда разработки C++ Builder).

- Свойство `Fields`. Это свойство в Инспекторе объекта не отражено. Свойство `Fields` является указателем на класс `TFields`, который имеет свои свойства:
  - `Count` — представляет количество полей объекта (таблицы);
  - `DataSet` — идентифицирует набор данных (в нашем случае — таблицу), которому принадлежат поля. Являясь указателем на объект `TDataSet`, это свойство позволяет работать со свойствами объекта `TDataSet`, в нашем случае фактически со свойствами таблицы;
  - `Fields` — это массив указателей на объект `TField`, с помощью свойств которого мы можем работать с полями таблицы. Например, свойства `AsBoolean`, `AsCurrensy`, `AsDateTime`, `AsFloat`, `AsInteger`, `AsString`, `AsVariant` позволяют обращаться к полю таблицы с учетом его типа. Например, можно записать:

    ```
    TDateTime d=Table1->Fields->Fields[0]->AsDateTime;
    ```

    В переменной `d` получим значение даты, хранящееся в первом поле таблицы. Имя поля таблицы можно получить, записав:

    ```
    AnsiString name=Table1->Fields->Fields[0]->
    FieldName;
    ```

Как еще использовать это свойство? Дело в том, что ваше приложение работает не с полями, помещенными в саму структуру таблицы, а с их копиями, находящимися в свойстве `Fields`. Это делается автоматически в момент запуска приложения. Но часто возникает необходимость работы не со всеми полями, определенными в структуре таблицы, а с их подмножеством, да и то в заданном вами порядке расположения полей, а не так, как вы это определили при создании таблицы. Вот тут вам на помощь приходит свойство `Fields`. Редактор, с помощью которого можно открыть множество полей таблицы, а затем создать необходимое подмножество в заданном вами порядке, а также изменять свойства самих полей ввиду того, что каждое поле — это отдельный объект со своим Инспектором, этот Редактор можно открыть, если воспользоваться контекстным меню таблицы или дважды щелк-

нуть мышью на значке таблицы. По умолчанию, если не выбирать никакого подмножества полей, в свойстве `Fields` будут использоваться все поля, что естественно. Откроем теперь контекстное меню этого Редактора (уже в его поле щелкнем правой кнопкой мыши) и выполним команду **Add Fields**. Откроется новое окно, содержащее все поля таблицы. В этом окне вы можете выбрать необходимые для работы вашего приложения поля и поместить их в предыдущее окно: щелкните мышью на белом поле, чтобы снять выделение со всех полей (т. к. по умолчанию в свойство `Fields` должны попасть все поля, то и выделенными при появлении окна оказались все. Если теперь нажать **OK**, все они перенесутся в окно Редактора полей). Когда подсветка снята, выберите необходимые поля с помощью клавиши <Ctrl> и щелчка мышью на нужном поле, а затем нажмите **OK**. Выбранные поля появятся в окне Редактора полей, которое примет вид, как на рис. 12.1.

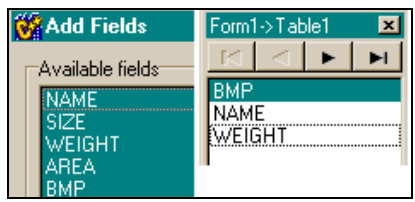


Рис. 12.1. Окно, открываемое контекстным меню Редактора полей, и результат выборки полей

Теперь, отмечая мышью нужную строку (поле), вы можете в появившемся Инспекторе объекта изменить некоторые свойства выделенного поля, а также задать порядок следования полей, который вам необходим при выводе таблицы на экран (надо мышью перетащить отмеченное поле в нужное место среди всех полей и отпустить кнопку). Например, в нашем случае порядок следования полей будет таким, как показано на рис. 12.2.

Изменим свойство поля `Weight`: пусть имя этого поля будет выводиться в компоненте, отображающем таблицу на экране,

по-русски. Последовательность и результат изменения свойства показаны на рис. 12.3.

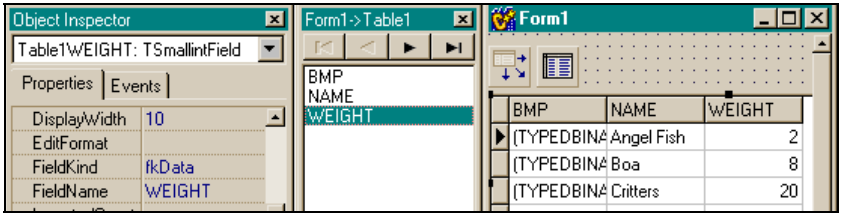


Рис. 12.2. Выбранные поля после изменения порядка их следования

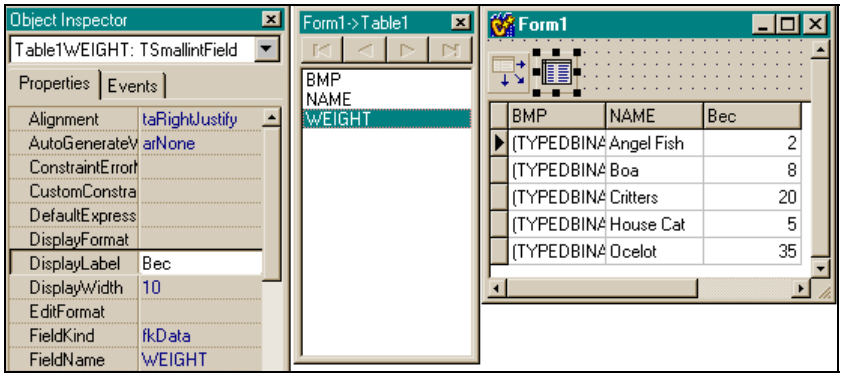


Рис. 12.3. Последовательность и результат изменения свойства `DisplayLabel` поля **Weight**

При использовании контекстного меню Редактора полей **Fields** можно было бы воспользоваться другой командой: **Add all Fields**. В этом случае в окне Редактора сразу появятся все поля таблицы. Но вы можете манипулировать полями с помощью все того же контекстного меню Редактора.

- Свойство `Filter` — здесь задается в виде строки символов правило фильтрования записей таблицы базы данных. Когда задан фильтр, то из таблицы выбираются только те записи, которые удовлетворяют условиям, указанным в строке-фильтре. Строка описывает условия фильтрования. В ней

указываются значения полей, записи с которыми должны выбираться. Остальные записи при выборке игнорируются. Значения полей задаются в одинарных кавычках. При создании строки используются операторы AND (И) и OR (ИЛИ), а также для обеспечения частичного сравнения — символ замещения *. Например, можно задать такой фильтр: "Выбрать все записи таблицы, в которых поле «Имя работника» начинается с буквы А". Тогда надо написать строку-фильтр в виде: `Имя поля='А*'`. Или другой пример: "Выбрать из таблицы — телефонного справочника — все записи, у которых поле Town (код города) содержит значения 373 или 024". Строку-фильтр тогда можно задать в виде: `Town='373' or Town='024'`. Приложения могут задавать фильтры в режиме исполнения, чтобы изменять условия фильтрации. Формирование строки-фильтра в свойстве `Filter` можно обеспечить вводом извне. Например, введя в редактируемое поле необходимую строку можно записать:

```
Table1->Filter=Edit1->Text;
```

Чтобы заданное значение свойства `Filter` заработало, следует установить свойство `Filtered` в `true` (разрешить фильтрацию). Вы можете сравнивать в строке-фильтре значения полей с литералами и константами, используя следующие операции сравнения и логические операции:

`<`, `>`, `>=`, `<=`, `=`, `<>` `and`, `not`, `or`.

- Свойство `Filtered` — включает/выключает фильтрацию, заданную в свойстве `Filter`.
- Свойство `FilterOptions` — это свойство определяет, чувствительна ли фильтрация к регистру данных в полях таблицы, а также разрешено ли частичное сравнение при фильтрации.
- Свойство `IndexDefs` — в этом свойстве задаются индексы, которые будут использованы в таблице для создания ее индексов, если она формируется в режиме исполнения приложения с помощью метода `CreateTable()`. (В таком же направлении используется и свойство `FieldDefs`).
- Свойство `MasterFields` — в этом свойстве задается связь между полями двух таблиц через диалоговое окно. Сначала

определяется свойство `MasterSource`, чтобы указать для подчиненной таблицы источник данных, через который будет устанавливаться связь с другой, главной, таблицей. В качестве источника данных выступает компонент `TDataSource`, связываемый через свое свойство `DataSet` с главной таблицей. `MasterFields` — это строка, содержащая одно или более имен полей в главной таблице. Имена полей разделены между собой точками с запятой. Каждый раз, когда меняется текущая запись в главной таблице, новые значения ее полей связи используются для выбора соответствующих записей подчиненной таблицы для их вывода. Чтобы установить связь между двумя таблицами в режиме проектирования приложения (design time), нужно открыть диалоговое окно Дизайнера связей полей. Оно открывается кнопкой с многоточием в свойстве `MasterFields`.

- Свойство `MasterSource` — его роль рассмотрена в пояснении к свойству `MasterFields`.
- Свойство `TableName`. Это свойство позволяет выбрать из раскрывающегося списка, содержащего перечень таблиц базы данных, заданной в свойстве `DatabaseName`, таблицу, с которой желает работать пользователь.
- Свойство `TableType`. Это свойство указывает структуру таблицы базы данных, которую компонент `TTable` представляет. Это свойство говорит о том, к какому типу базы данных принадлежит таблица: к dBASE, Paradox, FoxPro или это — ASCII-таблица.

Как настраивать компонент *TTable* на конкретную таблицу базы данных

Чтобы настроить `TTable` на конкретную таблицу БД, надо поместить компонент в форму, задать его свойство `DatabaseName` (выбрать из раскрывающегося списка псевдоним необходимой базы данных), а затем из базы данных выбрать нужную вам таблицу. Для этого в свойстве `TableName` нужно выбрать имя требуемой таблицы из раскрывающегося списка, в котором окажутся имена всех таблиц, содержащихся в заданном вами в свойстве

DatabaseName базы данных. После всего следует установить свойство TableType.

Некоторые методы *TTable*

- EmptyTable() — очищает таблицу (удаляет все ее записи). Речь идет о таблице, заданной свойствами DatabaseName и TableName компонента TTable. Но очистка таблицы может не произойти, если пользователь не обладает достаточными привилегиями, чтобы выполнить эту операцию (например, таблица захвачена другим пользователем).
- IsEmpty() — указывает, содержит ли таблица записи. IsEmpty() возвращает true, если таблица не содержит записей (пуста), иначе метод возвращает false.
- Open() — этим методом таблица открывается: ее свойство Active устанавливается в true. Когда Active = true, данные могут читаться из таблицы и записываться в нее. Свойство State таблицы устанавливается в dsBrowse. Если при открытии происходит какая-либо ошибка, свойство State переводится в dsInActive, и таблица не открывается.
- Close() — таблица закрывается: ее свойство Active устанавливается в false. После этого в таблицу нельзя ничего записывать и из нее нельзя ничего читать. Чтобы изменить какие-либо свойства таблицы, приложение должно предварительно ее закрыть.
- Refresh() — чтобы быть уверенным, что приложение работает с самыми поздними данными, их надо "освежать", т. е. приводить в актуальное состояние. Например, после отмены фильтрации в таблице она должна быть сразу подвержена обновлению с помощью этого метода, чтобы выводить все записи, а не те, что подвергались фильтрованию.
- First(), Last(), Next(), Prior() — с помощью этих методов указатель записи в таблице (ее курсор) устанавливается соответственно на первую, последнюю, следующую за текущей и предыдущую от текущей запись. Эти методы позволяют приложению перемещаться по записям в таблице.

- `MoveBy(n)` — передвигает указатель записи относительно текущей записи на `n` записей вперед (`n` > 0) или назад (`n` < 0).
- `Edit()` — разрешает редактирование данных в таблице. `Edit()` определяет текущее состояние набора данных. Если набор данных пуст, `Edit()` вызывает метод `Insert()` — вставку в набор данных. Если набор данных не пуст, переводит его в состояние `dsEdit`, разрешая приложению или пользователю модификацию полей в записи.
- `Insert()` — вставляет новую пустую запись в таблицу и делает ее активной (текущей, т. е. с ней можно работать). После вставки пустой записи в ее поля можно вводить данные и затем сохранять запись в таблице с помощью метода `Post()`.
- `Delete()` — удаляет активную запись и устанавливает указатель записи на следующую запись таблицы. Если таблица неактивна, возникает исключительная ситуация.
- `Append()` — добавляет новую пустую запись в конец таблицы и делает ее активной. Вновь добавленная запись записывается методом `Post()` в таблицу одним из следующих способов: для индексированных таблиц Paradox и dBase запись вставляется в таблицу в позицию, основанную на значении индекса. Для неиндексированных таблиц Paradox и dBase запись добавляется в конец таблицы. Для баз данных типа SQL физическое расположение записи имеет определенную специфику. Для индексированных таблиц индекс обновляется в соответствии с новой записью.
- `Post()` — записывает (отсылает) измененную запись в базу данных (в таблицу). Методы класса `TDataSet`, к которому принадлежат все методы `TTable`, изменяющие состояние таблицы (ее свойство `State`), такие как `Edit()`, `Insert()` или `Append()`, или те, что перемещают указатель записи в таблице, такие как `First()`, `Last()`, `Next()` и `Prior()`, автоматически вызывают метод `Post()`.
- `Cancel()` — отменяет изменения, внесенные в активную запись, если она еще не сохранена в таблице (методом `Post()`). Этот метод возвращает записи ее предыдущее состояние и переводит таблицу в состояние `dsBrowse`.

- `FieldByName()` — находит поле по его имени, заданному в параметре в виде строки `AnsiString`. Например,

  ```
  FieldByName("Price");
  ```

  `FieldByName()` — это указатель на класс `TField`. Поэтому приложение имеет прямой доступ к свойствам и методам этого класса. Например, следующий оператор определяет, является ли заданное поле вычислимым:

  ```
  if (Customers->FieldByName("FullName")->Calculated)
  Application.ShowMessage("This is a Calculated field",
  "FullName", MB_OK);
  ```

  Или, например, можно получить значение поля CustNo таблицы Customers так:

  ```
  Float Field=Customers->FieldByName("CustNo")->AsFloat;
  ```

  Чтобы получить или придать значение конкретному полю можно вместо метода `FieldByName()` использовать свойство `FieldValues`.

- `SetKey()` — этот метод употребляется для организации поиска записи в таблице по ключу записи. Чтобы организовать такой поиск, надо выполнить метод `SetKey()` — переключить таблицу в режим поиска и разрешить задать ключ поиска. Вызов этого метода переводит таблицу в состояние `dsSetKey` и очищает текущее содержимое буфера ключа. Затем следует задать значение ключа, по которому будет производиться поиск, и выполнить сам поиск методом `GotoKey()`. Например, так:

  ```
  Table1->SetKey(); Table1->FieldByName("State")-
  >AsString="CA"; Table1->FieldByName("City")-
  >AsString="Santa"; Table1-> GotoKey();
  ```

  Если не надо заменять весь существующий ключ, а только его изменить, следует вместо `SetKey()` применить `EditKey()`.

- `EditKey()` — разрешает модификацию буфера поискового ключа: переводит таблицу в состояние `dsSetKey` и сохраняет (в отличие от `SetKey()`) текущее содержимое буфера ключа. Чтобы определить текущее содержимое буфера ключа, можно использовать свойство `IndexFields`. `EditKey()` особенно полезен при применении многоразового поиска, когда каждый раз из всего множества ключевых полей при каждом новом

поиске изменяются только некоторые из них. Например, в таблице Customer ключевым полем является поле CustNo. Для поиска мы сформируем новый ключ, добавив к прежнему ключу еще два поля. Получим такую последовательность операторов:

```
Table1->EditKey();
Table1->FieldByName("State")->AsString="CA";
Table1->FieldByName("City")->AsString="Santa Barbara";
Table1->GotoKey();
```

- `GotoKey()` — устанавливает указатель записи таблицы (курсор) на запись, у которой значения ключевых полей совпадают со значением ключа, определенного с использованием методов `SetKey()` или `EditKey()`. Если `GotoKey()` находит подходящую запись, он возвращает `true`, иначе — `false`. Для таблиц типа Paradox и dBase ключом может быть индекс, указанный в свойстве `IndexName`. Если свойство `IndexName` не имеет значения, `GotoKey()` использует первичный индекс таблицы.

- `Locate()` — эта функция позволяет находить запись в таблице по совокупности значений ее полей, и найденную запись делает активной, позволяя тем самым доступ к ней. Функция описана как

```
Locate(const AnsiString KeyFields, const
System::Variant &KeyValues, TLocateOptions Options);
```

Функция возвращает `false`, если подходящая запись не обнаружена, и — `true` в противном случае. Если запись найдена, курсор таблицы указывает на найденную запись. Для обращения к функции следует задать фактические значения ее параметров. Параметр `Options` — это экземпляр класса `TLocateOptions`, в котором определены два параметра способа поиска: по частичному совпадению полей, выбранных для поиска (`loPartialKey`), и поиск без учета регистров клавиатуры (`loCaseInsensitive`). Чтобы придать значение параметру `Options`, надо воспользоваться операцией включения, переопределенной в классе: `<<` (переопределенной, потому что вообще-то в C — это операция сдвига битов числа влево, а

здесь она играет уже другую роль). Итак, задать параметр `Options` можно так:

- `TLocateOptions Options; Options.Clear();` (очищаем режимы);
- `Options << loPartialKey << loCaseInsensitive;` (задаем, если хотим использовать оба правила).

Теперь надо задать поля, по которым пойдет поиск, а затем и соответствующие им конкретные значения. Поля можно задать их именами, разделенными точкой с запятой (имена должны быть в кавычках, т. к. первый параметр — это строка символов), а значения полей (второй параметр) — через массив типа `Variant`: например, для двух полей можем написать:

```
Variant m[2];
m[0]=Variant("Sight Diver");
m[1]=Variant("P");
CusTTable->Locate("Company;Contact", VarArrayOf(m, 1),
Options);
```

Здесь `VarArrayOf()` — функция, которая создает и заполняет одномерный массив типа `Variant`. Второй параметр этой функции (у нас это 1) — это индекс последнего элемента массива.

Указать параметры можно и с помощью имен переменных. Например, для поиска по одному полю, имя которого находится в `Edit1->Text` (вводится в это поле), а значение — в `Edit2->Text` (тоже вводится в это поле), вид обращения к функции `Locate()` будет таким:

```
Locate(Edit1->Text,Edit2->Text,Options);
```

- `SetRangeStart()`, `SetRangeEnd()`, `ApplyRange()` — эти три метода (их можно заменить одним — `SetRange()`) задают диапазон для выборки записей:
  - `SetRangeStart()` переводит таблицу в режим задания нижней границы диапазона выборки. После этого метода надо установить начальное значение нижней границы диапазона: задать начальные значения полей, которые определят первую запись будущей выборки;

- `SetRangeEnd()` переводит таблицу в режим задания верхней границы диапазона выборки. После этого метода надо установить начальное значение верхней границы диапазона: задать начальные значения полей, которые определят последнюю запись будущей выборки;
- `ApplyRange()` формирует саму выборку. Если начать обрабатывать после этого записи таблицы, то ее первой записью будет запись, которая определена нижней границей диапазона, а последняя — верхней границей. Например, требуется задать диапазон выборки таблицы Orders (из BCDEMOS) по полю OrderNo (причем оно индексировано):

```
Table1->SetRangeStart();
Table1->FieldValues["OrderNo"]=1015;
Table1->SetRangeEnd();
Table1->FieldValues["OrderNo"]=1020;
Table1->ApplyRange();
```

- `GetBookmark()`, `GotoBookmark()`, `FreeBookmark()` — эти методы обеспечивают работу с закладками (bookmark) — средствами, которые позволяют пометить текущую запись. Благодаря закладке можно вернуться на запись, на которой эта закладка установлена, обработав какое-то количество записей, при необходимости можно убрать закладку с записи. Как работать с закладками, видно из следующего примера:

```
TBookmark p;  /объявляем указатель типа закладка*/
p=Table1->GetBookmark();  /*функция возвращает
указатель типа TBookmark */
Table1->Prior();  //уходим с помеченной записи
Table1->GotoBookmark(p);  /*возвращаемся на помеченную
запись*/
Table1->FreeBookmark(p);  /*освобождаем помеченную
запись от закладки*/
```

Компонент *TDataSource*

Этот компонент — посредник между набором данных и другими компонентами, работающими с набором данных (компоненты, обеспечивающие вывод данных, перемещение по ним, их редак-

тирование). Если данные набора данных должны выводиться или с ними в этих компонентах должны производиться какие-нибудь действия, то наборы данных связываются с *источником данных* `TDataSource` (его название так и переводится — источник данных). Компоненты-источники данных также связывают наборы данных отношениями "главный-подчиненный". Основное свойство компонента — свойство `DataSet` — здесь из выпадающего списка, в который попадают имена всех компонентов-наборов данных, присутствующих в форме, выбирается тот, с которым связывается источник данных. В режиме исполнения формирование свойства выглядит так:

```
DataSource->DataSet=Table1;
```

Компонент *TDBGrid*

Этот компонент создает в форме таблицу для отображения данных из таблиц базы данных, информация из которых выбирается с помощью компонентов `TTable` и др. Он используется не только для вывода данных на экран, но с его помощью можно редактировать данные и вводить новые. Предварительно компонент должен быть настроен на тот источник данных, который связан с таблицей, данные которой будут выводиться через `TDBGrid`.

Некоторые свойства *TDBGrid*

- `DataSource` — здесь из выпадающего списка всех источников данных, расположенных в форме, выбирается тот, который связан с таблицей (т. е. с компонентом `TTable`, ее представляющим), данные которой надо вывести в таблицу `TDBGrid`.
- `Columns` (колонки, столбцы таблицы) — в этом свойстве определяются параметры столбцов компонента. Если в поле этого свойства щелкнуть мышью, появится кнопка с многоточием, при нажатии которой откроется диалоговое окно для редактирования свойств столбцов таблицы `TDBGrid` (этого эффекта можно достичь, если открыть контекстное меню компонента).

- `FixedColor` — задает цвет верхней и боковой линеек таблицы `TDBGrid`.
- `Options` — представлено набором режимов, которые определяют виды вывода данных и поведенческие свойства таблицы `TDBGrid`.

Компонент *TDBNavigator*

Компонент `TDBNavigator` используется для перемещения по набору данных и для выполнения определенных операций над данными этого набора. Это операции "вставка пустой записи" и "сохранение записи". Навигатор применяется к таким компонентам, как `TDBGrid` или `TDBEdit`. Когда пользователь нажимает на одну из кнопок Навигатора, выполняется соответствующее ей действие, аналогичные функциям `TTable`, связанным с перемещением по таблице, вставкой в нее, удалением из нее и др.

Как используется *TDBNavigator*

`TDBNavigator` связывают с источником данных через свойство Навигатора `DataSource`. (Напомним, что источник данных — это компонент `TDataSource`, который связан с конкретным набором данных.) Если с этим же источником данных связана таблица `TDBGrid`, то в результате получается механизм управления таблицей `TDBGrid`: щелкая на соответствующих кнопках Навигатора, мы обеспечиваем движение по строкам `TDBGrid`, а следовательно и по записям связанной с ней таблицы. При этом можем работать с активными записями.

О компонентах работы с полями набора данных

Для работы с отдельными полями таблицы существуют соответствующие компоненты, в названии которых первыми буквами являются `DB`. Они и указывают на принадлежность компонента к рассматриваемой группе. Все они, если их поместить в форму,

содержащую таблицу (компонент `TTable`, через который осуществляется связь с таблицей базы данных) — все они настраиваются с помощью своих свойств в поле таблицы, которое они представляют. Настройка идет, во-первых, через свойство `DataSource`, в котором указывается источник данных (а по сути — таблица с данными, т. к. этот "источник" связан напрямую с таблицей БД), а во-вторых — через свойство `DataField`, в раскрывающемся окне которого надо выбрать необходимое поле. Список полей в этом окне появляется, когда вы свяжете компонент с необходимым источником данных, через него и будет передан в окно список полей таблицы.

С компонентами-полями вы можете работать так, как вы работаете с полями таблицы: читать и записывать данные через них. Например, через `TDBCheckBox` можно изменять значение булевых полей таблицы:

```
DBCheckBox1->Field->AsBoolean=true;
```

Мы используем компонент `TDBEdit` для ввода данных в таблицу, привязав его к Навигатору. Это делается для того, чтобы двигаться с помощью Навигатора по записям таблицы и выполнять с ними определенные действия.

Пример ввода данных в таблицу

Таблица создана из трех полей таблицы Clients из БД BCDEMOS.

Форма приложения показана на рис. 12.4.

Здесь использованы компоненты `TDBEdit`, `TDBImage`. Первый — для ввода текстовых данных, второй — для ввода изображения. Компонент `TDBGrid` взят для наглядности отображения ввода. Для поиска изображения для ввода в таблицу используется компонент `TOpenPictureDialog`. Как происходит ввод? Кнопкой со знаком + Навигатора формируется пустая запись таблицы, в которую надо вписать данные (через поля ввода — компоненты `DBEdit`). Затем следует с помощью кнопки **Добавка картинки из файла bmp** выбрать необходимое изображение (оно поместится в компонент `TImage`) и нажать на кнопу **Запись** Навигатора. Изображение вставляется в таблицу (компонент `DBImage1`) через буфер памяти, куда предварительно по-

мещается из объекта `Image1`, в котором оказывается после выбора по `OpenPictureDialog1`.

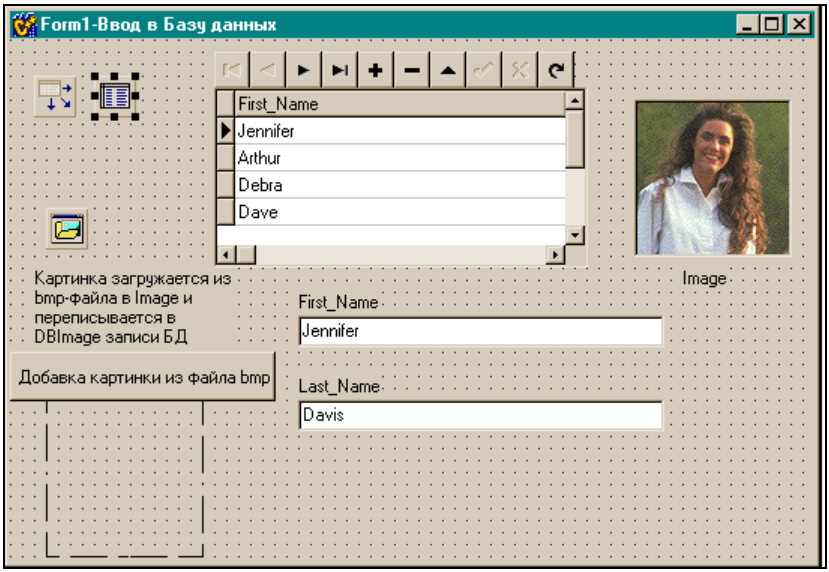


Рис. 12.4. Форма приложения для ввода данных в таблицу

Программный модуль приложения приводится в листинге 12.1.

Листинг 12.1

```
//cpp-файл
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
#include <vcl\Clipbrd.hpp>  //чтобы использовать работу
                            //с функцией вставки-выбора
                            //из буфера
```

```
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
if(OpenPictureDialog1->Execute()) //Когда выбрали файл, его
                                  //имя находится в FileName,
                                  //он загружается
                                  //в компонент Image1
 {
 Image1->Picture->LoadFromFile(OpenPictureDialog1->FileName);
 }
 Clipboard()->Assign(Image1->Picture); //запись в буфер
 //Image1->Picture->Assign(Clipboard()); //запись из буфера
 DBImage1->PasteFromClipboard(); //вставка в DBImage из буфера
}
```

//**h-файл**

```
//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Db.hpp>
#include <DBCtrls.hpp>
#include <DBGrids.hpp>
```

```
#include <DBTables.hpp>
#include <ExtCtrls.hpp>
#include <Grids.hpp>
#include <Mask.hpp>
#include <Graphics.hpp>
#include <ImgList.hpp>
#include <Dialogs.hpp>
#include <ExtDlgs.hpp>
#include <Buttons.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
    TDataSource *DataSource1;
    TTable *Table1;
    TDBNavigator *DBNavigator1;
    TDBGrid *DBGrid1;
    TStringField *Table1First_Name;
    TStringField *Table1Last_Name;
    TBlobField *Table1Image;
    TLabel *Label1;
    TDBEdit *DBEdit1;
    TLabel *Label2;
    TDBEdit *DBEdit2;
    TLabel *Label3;
    TDBImage *DBImage1;
    TOpenPictureDialog *OpenPictureDialog1;
    TLabel *Label4;
    TBitBtn *BitBtn1;
    TImage *Image1;
    void __fastcall BitBtn1Click(TObject *Sender);
private:    // User declarations
public:    // User declarations
    __fastcall TForm1(TComponent* Owner);
};
```

```
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Компонент *TQuery*

Компонент предназначен для работы с таблицами базы данных с помощью запросов, составленных на языке SQL. Компонент позволяет организовать работу одновременно с несколькими таблицами. Среда Builder обеспечивает выполнение следующих операторов SQL:

- `select` — обеспечивает выбор данных из таблиц (может выбирать сразу из нескольких таблиц, в том числе и по отдельным полям таблиц);
- `insert` — обеспечивает вставку записи в таблицу;
- `update` — обеспечивает обновление записи в таблице;
- `delete` — обеспечивает удаление записи.

`Query`-компоненты могут использоваться с удаленными серверами баз данных таких, как Sybase, SQL Server, Oracle, Informix, DB2, InterBase, с локальными таблицами (Paradox, InterBase, dBase, Access, FoxPro) и с базами данных, поддерживаемыми механизмом ODBC.

`Query`-компоненты полезны тем, что они позволяют одновременно работать со многими таблицами и делать выборку не только отдельных полей, но и отдельных строк. Кроме того, при разработке приложения с использованием `TQuery` на локальном компьютере благодаря свойствам `TQuery` приложение может быть легко распространено для работы с удаленными базами данных.

Некоторые свойства *TQuery*

- `DatabaseName` — здесь указывается псевдоним базы данных, к таблицам которой будет сформирован запрос.
- `SQL` — если щелкнуть в поле значения этого свойства, то появится кнопка с многоточием, при нажатии которой открыва-

ется диалоговое окно для записи в нем запроса на языке SQL. Пример заполненного диалогового окна показан на рис. 12.5.

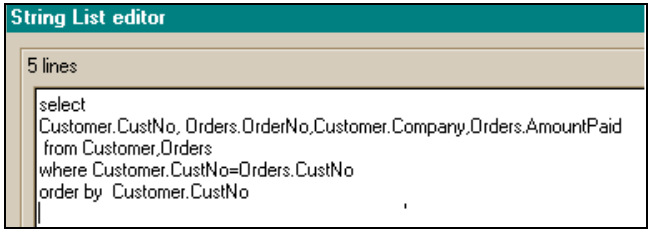


Рис. 12.5. Формирование запроса к БД в диалоговом окне

В окне показано, что запрашивается выборка из двух таблиц Customer и Orders по их отдельным полям. В первой строке запроса находится оператор `select`. Во второй строке указано, что выбирать: из таблицы Customer выбрать поле CustNo, из таблицы Orders — поле OrderNo, из таблицы Customer — поле Company, из таблицы Orders — поле AmountPaid. В третьей строке задано, откуда производить выборку: из таблиц Customer, Orders. В четвертой строке задается условие выборки: выбрать из обеих таблиц такие записи, значения полей CustNo которых совпадают. В пятой строке задано условие упорядочения результата выборки: записи должны быть упорядочены по возрастанию значений поля CustNo.

- `Params` — когда в свойстве `SQL` определяется запрос, он может быть построен так, что, например, выборка будет выполняться не для конкретных значений полей, а для значений параметризированных. Мы можем сформировать, например, такой запрос:

```
select * from Customer where CustNo=1221.
```

Это означает: "Выбрать из таблицы Customer все записи, у которых поле CustNo содержит число 1221". Но когда нам надо сделать такую же выборку, но для поля со значением 1222, придется снова формировать запрос, снова компилировать приложение и т. д. Возникают определенные неудобства. Чтобы избежать подобных неприятностей, применяют средство, которое называется *параметризация запроса*. Вместо

конкретного значения поля пишут имя некоторого параметра, впереди имени ставят двоеточие, чтобы показать, что данное имя — параметр SQL. Предыдущий запрос можно записать параметрически так:

```
select * from Customer where CustNo =:p
```

где `p` — это параметр запроса. Поскольку мы задали параметр, то необходимо определить его характеристики. Это и делается в свойстве `Params`. Если щелкнуть на этом свойстве, появится кнопка с многоточием, при нажатии которой открывается окно, где появится заданный нами параметр и его порядковый номер в запросе (в запросе может быть более одного параметра). Теперь следует щелкнуть мышью на этом параметре, в результате чего откроется окно Инспектора объекта со свойствами параметра, которые и надо определить (рис. 12.6).

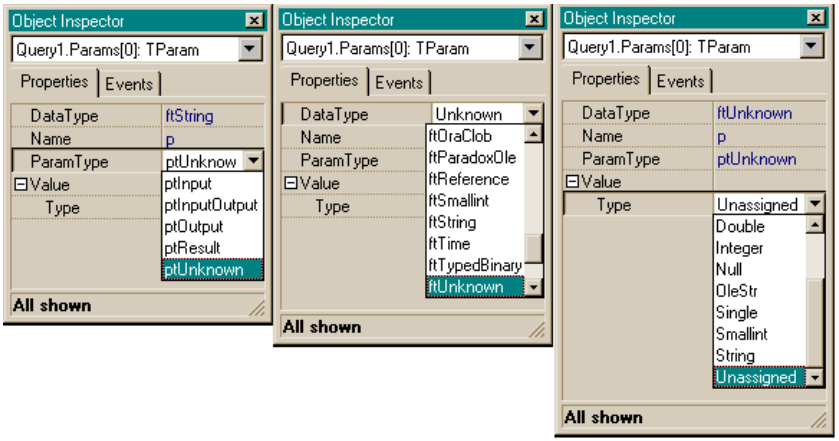


Рис. 12.6. Инспектор объекта для определения свойств параметра

Следует задать тип поля (**DataType**), чье значение параметр представляет (или выбрать из выпадающего списка **Value Type** значение **Variant**), задать тип параметра (входной, выходной и т. д.) из выпадающего списка. После всех этих действий остается присвоить параметру конкретные значения в

режиме исполнения приложения и получить результаты выборки в зависимости от значения параметра. Это можно выполнить с помощью ввода конкретного значения параметра в компонент `TEdit` и обработчика кнопки.

Приведем пример приложения, использующего запрос с параметром (в форму помещены: кнопка и компоненты `TEdit`, `TQuery`, `TDBGrid`, `TDataSource`). Форма приложения и запрос показаны на рис. 12.7.

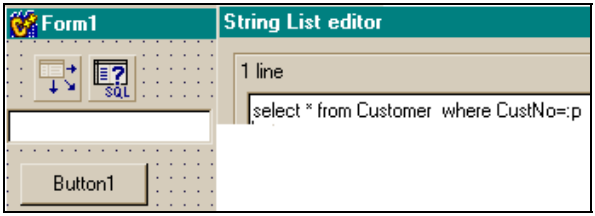


Рис. 12.7. Форма приложения для параметрического запроса

Текст программы приложения приведен в листинге 12.2.

Листинг 12.2

```
//cpp-файл

#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
```

```
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Query1->Close();
if(Edit1->Text != "")
 {
  if(!Query1->Prepared) /*Готов ли запрос для исполнения?
  Если не готов - подготовить */

  Query1->Prepare();
    /* Подготовка запроса к исполнению: Prepare() запускает
    BDE и удаленный сервер или локальную БД, чтобы были
    выделены ресурсы для запроса. Вызов Prepare() до выполнения
    запроса улучшает работу приложения за счет выполнения
    определенных действий по оптимизации процесса. Дело
    в том, что C++ Builder автоматически подготавливает
    запрос, если он выполняется без предварительного
    выполнения метода Prepare(). После выполнения запроса
    C++ Builder освобождает ресурсы, выделенные для исполнения
    запроса. Когда требуется выполнять запрос несколько раз,
    приложение должно само явно подготовить среду к запросу
    (выполнить метод Prepare()), чтобы избежать многочисленных
    и ненужных подготовок и удалений результатов подготовок,
    выполняемых Builder, когда нет явного выполнения метода
    Prepare(). Для подготовки запроса требуются определенные
    ресурсы БД, поэтому в приложении желательно после
    исполнения запроса освободить эти захваченные ресурсы,
    выполнив метод UnPrepare(). Когда вы изменяете текст
    запроса в режиме исполнения приложения, запрос
    автоматически закрывается и все ресурсы, отданные ему
    для выполнения, освобождаются */

Query1->ParamByName("p")->AsFloat=StrToFloat(Edit1->Text);
/*Придание параметру введенного извне значения*/
Query1->Open(); /*Запрос открывается и исполняется*/
 if(Query1->Eof)
 /*При выполнении запроса достигнут конец таблицы,
 это означает, что по запросу ничего не найдено */
 {
    ShowMessage("Не найдена информация по запросу CustNo=" +
Edit1->Text);
```

```
    return;
 }
 }
 else
 {
     ShowMessage("Введите значение параметра");
     return;
 }
}
//---------------------------------------------------------
void __fastcall TForm1::DBGrid1DblClick(TObject *Sender)
{
 Query1->Close();
 Query1->UnPrepare(); /*Освобождение ресурсов, выделенных для
 исполнения запроса*/
}
//---------------------------------------------------------
```

//h-файл

```
//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Db.hpp>
#include <DBGrids.hpp>
#include <DBTables.hpp>
#include <Grids.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:   // IDE-managed Components
    TDataSource *DataSource1;
```

```
    TQuery *Query1;
    TEdit *Edit1;
    TButton *Button1;
    TDBGrid *DBGrid1;
    void __fastcall Button1Click(TObject *Sender);
    void __fastcall DBGrid1DblClick(TObject *Sender);
private:   // User declarations
public:    // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Примечание

Освобождение ресурсов, выделенных базой данных запросу, осуществляется в обработчике таблицы `TDBGrid` по двойному щелчку в ее поле, который означает, что результаты запроса не нужны, поэтому запрос закрывается. Ресурсы, выделенные ему, освобождаются (см. комментарии к тексту программы).

Методы *TQuery*

- `ExecSQL()` — применяется для запросов, которые модифицируют базу данных (для выборки данных по оператору `select` используется метод `Open()`). Например, имеем:

```
Query1->Close();  //метод закрытия запроса
Query1->SQL->Clear(); //метод очистки запроса — это метод SQL
Query1->SQL->Add("Delete Kadry where TabNom=1000");
Query1->ExecSQL();
```

По этому запросу строка таблицы Кадры с табельным номером 1000 будет удалена из БД.

- `Open()` — это эквивалент установления свойства `Active` в `true`. Используется для запросов на выборку данных (оператором `select`).
- `Close()` — эквивалентен установлению свойства `Active` в `false`. Делает запрос неактивным.
- `Prepare()` — выделяет ресурсы для запроса. Подробно объяснен при рассмотрении свойства `Params`.
- `First()`, `Last()`, `Next()`, `Prior()`, `MoveBy(n)`, `Append()`, `Insert()`, `Edit()`, `Delete()`, `Post()`, `Cancel()`, `GetBookmark()`, `GotoBookmark()`, `FreeBookmark()`, `IsEmpty()`, `Locate()` воздействуют на результат запроса и по своей функциональности совпадают с соответствующими методами для `TTable`.

Общие сведения о хранимых процедурах

Хранимые процедуры — это средства оптимизации работы клиента с базами данных, расположенными на удаленных серверах. Если клиенту требуется обработать некоторые данные, находящиеся в БД на удаленном сервере, он может запросить эти данные и обработать их. Это один путь.

Возможно, что данных очень много, а в качестве результата должно быть получено только одно значение: например, клиенту надо получить среднее значение некоторой величины на основе данных, хранящихся на удаленном сервере, и записей. В таком случае надо обработать несколько миллионов (это может быть, например, статистика переписи населения страны). Чтобы перегнать для расчета на свой компьютер такое количество записей, потребуется значительное количество ресурсов сети. Поэтому сразу возникает вопрос: а нельзя ли все расчеты производить на месте, т. е. в базе данных удаленного сервера, а результат расчета, объем которого несравненно меньше, чем данные для его расчета, пересылать клиенту? Оказывается, можно. И это — второй путь работы клиента с удаленными данными.

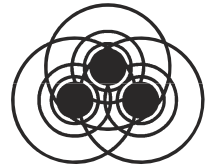
Определенные базы данных поддерживают средства создания специальных процедур (SQL-сервер, Oracle и др.). С их помощью в базе данных, расположенной на сервере, создается программа — *хранимая процедура* (хранимая на сервере), которая выполняет необходимые расчеты. Хранимые процедуры — это именованный набор SQL-операторов, помещаемый в базу данных на SQL-сервере. Результаты расчета могут быть получены любым клиентом этого сервера с помощью компонента `TStoredProcedure`, устанавливаемого в форме клиентского приложения. В свойствах этого компонента определяется удаленная база данных, выбирается имя хранимой процедуры в удаленной базе, определяются параметры, через которые будут передаваться в процедуру величины, требуемые для проведения расчетов, а также параметры, через которые клиенту будут возвращаться результаты расчетов. Хранимую процедуру можно задать и с использованием компонента `TQuery` клиентского приложения, определив запрос в свойстве `SQL` (в `TQuery` надо задать имя БД, поддерживающей аппарат создания хранимых процедур):

```
Query1->ParamCheck=false;
Query1->SQL->Clear();
Query1->SQL->Add("CREATE PROCEDURE GET_MAX_EMP_NAME");
Query1->SQL->Add("RETURNS (Max_Name CHAR(15))");
Query1->SQL->Add("AS");
Query1->SQL->Add("BEGIN");
Query1->SQL->Add(" SELECT MAX(LAST_NAME)");
Query1->SQL->Add(" FROM EMPLOYEE");
Query1->SQL->Add(" INTO :Max_Name;");
Query1->SQL->Add(" SUSPEND;");
Query1->SQL->Add("END");
Query1->ExecSQL();
```

Здесь средствами языка SQL в режиме исполнения приложения создается хранимая процедура `GET_MAX_EMP_NAME`. С помощью этой процедуры можно получить последнюю по алфавиту фамилию (значение поля `LAST_NAME`) служащего из таблицы EMPLOYEE и результат записать в параметр `Max_Name`.

Хранимая процедура в базе данных — это тоже элемент базы данных, каким является и таблица базы данных.

Глава 13



Вывод отчетов

Вывод отчетов (видеограмм, если речь идет о выводе на экран, и машинограмм, если речь идет о печати результатов работы приложений на бумаге) осуществляется с помощью компонента `TQuickRep` (вкладка **QReport**). Смысл его работы состоит в том, что вы соединяете этот компонент с наборами данных (представляемыми компонентами `TQuery`, `TTable`), располагаете на его листе другие компоненты из вкладки **QReport**, которые формируют интерфейс компонента с наборами данных и через которые осуществляется собственно вывод данных.

Получение простейшего отчета

Составим отчет на основании данных двух таблиц из БД BCDEMOS: Customer и Orders, выведя на экран поля, заданные запросом в компоненте `TQuery`:

```
select Customer.CustNo, Customer.Company, Customer.City,
Orders.OrderNo, Orders.SaleDate,Orders.AmountPaid
from Customer,Orders
where Customer.CustNo=Orders.CustNo
order by Customer.CustNo
```

Ниже определен порядок подготовки отчета. Размещаем в форме компонент `TQuery` и в его свойстве `SQL` задаем вышеприведенный запрос. Свойство `DatabaseName` устанавливаем равным `BCDEMOS`. Размещаем в форме компонент `TQuickRep`, а его свойство `DataSet` определяем как `Query1` (это и есть связка компонента вывода отчетов с конкретным набором данных). Формирование вывода возможно, поскольку на компонент `TQuickRep`

помещают горизонтальные полосы (bands), которые сами являются компонентами TQRBand, для них существуют свои Инспекторы объекта, где определяются свойства этих полос. Компоненты TQRBand — это контейнеры для помещения в них различных данных: заголовка, верхнего и нижнего колонтитула страницы, верхнего и нижнего колонтитула группы данных, компонентов, через которые собственно и выводятся данные, и т. д. Рассмотрим свойства TQRBand.

- AlignToBottom — если это свойство равно true, а свойство BandType равно rbDetail, то полоса с таким типом будет распечатываться одна на страницу.
- BandType — это свойство задает тип полосы, определяет ее роль в выводе данных отчета. Некоторые типы полос зарезервированы для специальных компонентов и не используются. Типы полос:
  - rbTitle — эта полоса служит для задания названия отчета. Она выводится всегда на первой странице отчета после заголовка страницы (в котором выводятся номер страницы, дата вывода и т. п.);
  - rbPageHeader — эта полоса задает заголовок страницы и выводится всегда самой первой на каждой странице. Печатание ее может быть отключено с помощью свойства Options;
  - rbDetail — эта полоса выводится один раз для каждой записи (строки) выводимого набора данных;
  - rbPageFooter — полоса выводится в нижней части каждой страницы. Вывод может быть отключен через свойство Options;
  - rbSummary — полоса выводится после всех полос типа rbDetail и групповых полос-подписей в конце отчета (служит для вывода итоговых данных по отчету);
  - rbSubDetail — зарезервирована для компонента TQRSubDetail. Вручную не рекомендуется устанавливать;
  - rbColumnHeader — полоса выводится в вершине каждого столбца на каждой странице в многоколоночном отчете. Используется для задания заголовков столбцов. Выводится

один раз после заголовка страницы при отчете с одним столбцом;

- `rbGroupFooter` — это полоса для задания подписей при использовании компонентов `TQRGroup` и `TQRSubDetail`. Выводится по окончании группы или всех полос типа `rbDetail`, выводимых для компонента `TQRSubDetail`;
- `rbGroupHeader` — заголовочная полоса для компонентов `TQRGroup` и `TQRSubDetail`;
- `rbOverlay` — тип включен для обратной совместимости с `QuickReport` версии 1;
- `rbChild` — тип зарезервирован для компонента `TQRChildBand`. Рекомендуется не использовать его вручную.

- `ForceNewColumn` — если это свойство имеет значение `true`, то полоса всегда будет выводиться с новой колонки.
- `ForceNewPage` — если установить это свойство в `true`, то полоса всегда будет выводиться с новой страницы. Если заголовок страницы не выключен, то он будет печататься первым.
- `Frame` — это составное свойство, определяющее оформление кадра всего отчета или полосы с помощью подчиненных свойств: `DrawBottom` (`Left`, `Right`, `Top`) — в нижней (левой, правой, верхней) части кадра отчета рисуется линия, цвет которой задан в подсвойстве `Color`, ширина — в посвойстве `Width`, а стиль — в подсвойстве `Style`. То есть отчет можно оформить помещенным в цветную рамку (кадр) определенной ширины.
- `HasChild` — если потомком данной полосы является полоса типа `TQRChildBand`, то свойство должно быть установлено в `true`, иначе — в `false`. Такие структуры используются для обеспечения вывода групповых подчиненных данных типа "в том числе".
- `LinkBand` — это свойство используется, чтобы обеспечить вывод двух различных полос на одной странице, например, чтобы не "потерять" полосы-подписи, полосы-итоги. Соединяя данную полосу с полосой определенного типа, мы гарантируем, что связанная полоса не появится в качестве первой на

новой странице, что текущая полоса всегда будет перемещена на следующую страницу и выведена перед связанной с ней полосой.

Свойства *TQuickRep*

- `Bands` — это составное свойство, определяющее, какие полосы будут появляться в отчете:
  - `HasColumnHeader` — если это свойство установлено в `true`, полоса для заголовка колонки будет включена в отчет. Эта полоса выводится в вершине каждого столбца отчета;
  - `HasDetail` — если это свойство установлено в `true`, эта полоса включается в отчет. Полоса выводится для каждой записи набора данных;
  - `HasPageFooter` — если это свойство установлено в `true`, то полоса включается в отчет. Полоса выводится в конце каждой страницы. С помощью свойства `Options` можно управлять выводом этой полосы на последней странице отчета;
  - `HasPageHeader` — если это свойство установлено в `true`, то полоса включается в отчет. Полоса выводится в начале каждой страницы. С помощью свойства `Options` можно управлять выводом этой полосы на первой странице отчета;
  - `HasSummary` — если это свойство установлено в `true`, то полоса включается в отчет. Полоса выводится после всех полос со свойством `rbDetail`;
  - `HasTitle` — если это свойство установлено в `true`, то полоса включается в отчет. Полоса выводится перед выводом полос со свойством `rbDetail`. Чтобы полоса появлялась вместо полосы заголовка страницы на первой странице отчета, установите свойство `Options/FirstPageHeader` в `false`.
- `DataSet` — используется для связи отчета с набором данных. Набором данных могут быть компоненты `TQuery`, `TTable`, `TRemoteDataset` или любой потомок компонента `TCustomDataset`.

- `Description` — используется для задания комментария в окне, раскрываемом кнопкой с многоточием в поле свойства. Само свойство не используется в `QuickReport`, но с его помощью перед выводом отчета вы можете сообщить пользователям дополнительную информацию.
- `Frame` — свойство аналогично одноименному свойству полосы.
- `Options` — это свойство составное. Его подсвойства используются для установки некоторых флажков, определяющих поведение отчета:
  - `FirstPageHeader` — задает, будет ли полоса заголовка страницы выводиться на первой странице;
  - `LastPageHeader` — задает, будет ли полоса подписи страницы выводиться на последней странице;
  - `Compression` — задает, будет ли отчет сжиматься в памяти во время генерации.
- `Page` — это составное свойство, подсвойства которого определяют характеристики страницы для ее вывода на печать: расстояние между колонками отчета, отступы справа и слева, формат листа бумаги и т. п.
- `PrinterSettings` — это составное свойство, подсвойства которого задают характеристики печати: количество напечатанных копий, диапазон печатаемых страниц, тип кармана (лотка) печатающего устройства, а с помощью свойства `Duplex` можно задать печать с обеих сторон бумаги (если принтер поддерживает такой режим).
- `PrintIfEmpty` — свойство определяет, как происходит печать пустого отчета. Если свойство `PrintIfEmpty` имеет значение `true`, страница выводится со своим заголовком, заголовком отчета, итоговой полосой и полосой-подписью. В противном случае отчет вообще не генерируется.
- `ReportTitle` — это свойство определяет имя задания для принтера и идентификатор отчета в операциях по генерации и печати отчета.
- `ShowProgress` — если это свойство установлено в `true`, то во время печати отчета в прямоугольном окне будет виден ход выполнения печати (работает компонент `TProgressBar`).

Формирование отчета

Поместим в поле TQuickRep три полосы и зададим их типы как rbPageHeader (заголовок страницы), rbTitle (заголовок отчета) и rbDetail (полоса для вывода значений полей набора данных). Они будут расположены на компоненте в соответствии со своим статусом. В полосу для вывода значений полей набора данных поместим компоненты TQRDBText, через которые выводятся поля набора данных: для каждого поля должен существовать свой компонент, который настраивается на поле с помощью свойств DataSet (набор данных: выбирается из раскрывающегося списка) и DataField (имя поля набора данных: выбирается из раскрывающегося списка полей, появляющегося после задания свойства DataSet). Кроме этих основных свойств, TQRDBText имеет другие свойства редакционного характера.

Свойства *TQRDBText*

- AutoSize — аналогично одноименному свойству компонента TLabel.
- AutoStretch — это свойство определяет, как поведет себя компонент, если текст, помещенный в него, будет выходить за границы поля вывода. Если AutoStretch = true, поле будет автоматически расширяться вниз. Если недостаточно места на одной странице, поле может перейти на следующую страницу. Если внизу находится другой такой же компонент, то расширившийся компонент его перекроет. Поэтому в таких случаях надо помещать нижний компонент в полосу-потомка.
- Color — задает цвет выводимого поля.
- Font — задает шрифт выводимого поля.
- Frame — задает окантовку выводимого поля (как и для полосы и отчета в целом).
- Mask — задает формат выводимого поля. Форматы для чисел задаются в соответствии с форматом для функции FormatFloat(), а форматы для дат задаются в соответствии с форматом для функции FormatDateTime().

- `Size` — здесь автоматически отражаются размеры и отступы от края компонента печати, если вы изменяете его конфигурацию (но эти размеры можно и самому задавать в соответствующих полях).
- `Transparent` — свойство задает прозрачность компонента.
- `WordWrap` — свойство определяет, как будет размещен текст, превышающий размер поля вывода. Если `WordWrap` = `true`, текст будет перенесен на новую строку. Если свойство `AutoStretch` имеет значение `true`, то поле расширится вертикально. Если `WordWrap` = `false`, текст будет обрезан.

Итак, поместим в полосу вывода значений полей набора данных шесть компонентов `TQRDBText` и настроим их свойства на соответствующие поля, выводимые запросом `Query1`. Получим отчет, показанный на рис. 13.1.

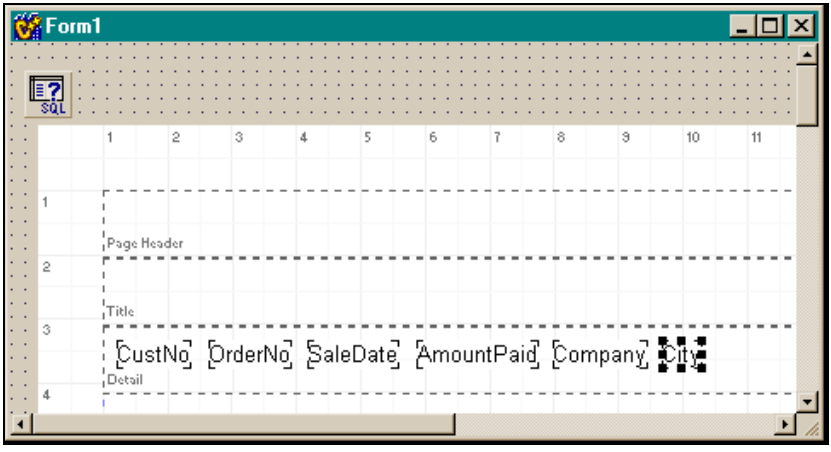


Рис. 13.1. Отчет после помещения в полосу `Detail` элементов `TQRDBText`

Здесь мы настроили только свойства `DataSet` и `DataField`. Зададим теперь заголовок отчета. Для этого в полосу `Title` надо поместить столько компонентов `TQRLabel`, сколько строк будет в заголовке. `TQRLabel` — это обычная метка с теми же свойствами, только для использования в компоненте вывода отчетов. Зане-

сем в ее свойство `Caption` название отчета. Поместим на полосу заголовка страницы еще один компонент `TQRLabel`, в свойство которого `Caption` занесем заголовок страницы.

Получим отчет, показанный на рис. 13.2.

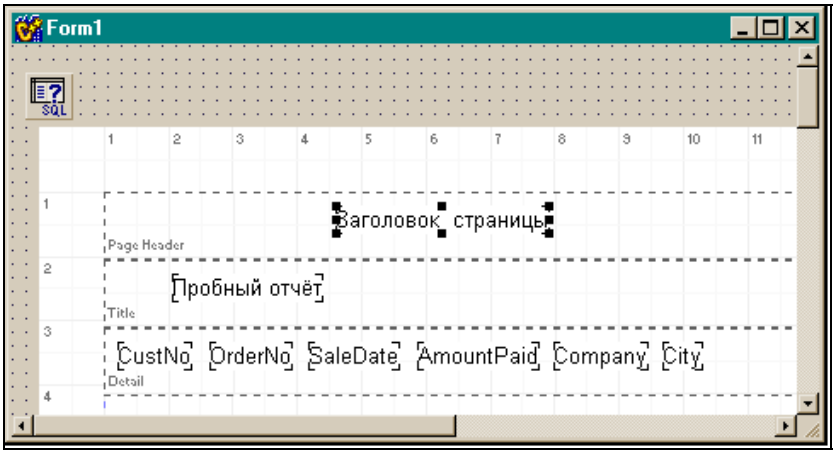


Рис. 13.2. Формирование заголовка страницы и заголовка отчета

Теперь нам надо озаглавить поля вывода. Для этого поместим в поле `QuickRep1` полосу `TQRBand`, дадим ей тип `rbColumnHeader`, свяжем ее с полосой вывода значений полей (`QRBand3`) через свойство `LinkBand`. В этой полосе над каждым полем полосы `Detail` поместим компонент `QRLabel`. Получим отчет, показанный на рис. 13.3.

Теперь можно просмотреть созданный отчет. Для этого надо открыть контекстное меню и выполнить в нем опцию **Preview** (проверьте, чтобы свойство `DataSet` компонента `QuickRep1было` было установлено в `Query1`, а свойство `Active` объекта `Query1` — в `true`). Получим результат, показанный на рис. 13.4.

Чтобы отчет выглядел более привлекательно, можно определить свойства `Frame` для соответствующих компонентов, задать метки для формирования горизонтальных разделителей и т. д.

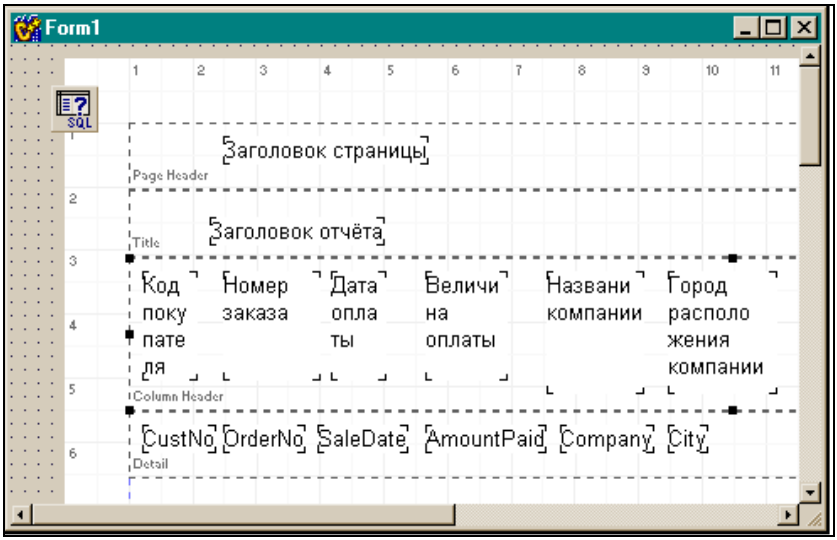


Рис. 13.3. Отчет
с заголовками столбцов

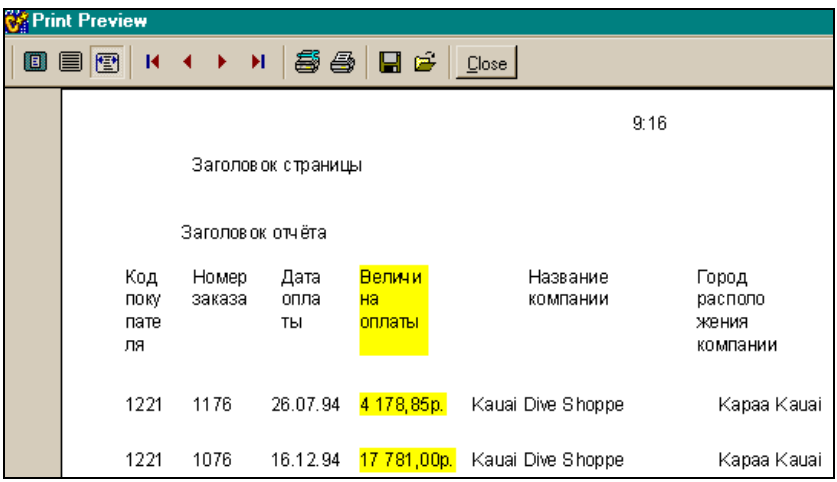


Рис. 13.4. Результат предварительного просмотра
сформированного отчета

В верхней части отчета можно поместить компонент (или несколько для каждого типа данных) `QRSysData`. В свойстве этого компонента `Data` следует выбрать из раскрывающегося списка тип отображаемых данных: будет ли это дата, либо номер страницы и т. п. Чтобы запускать приложение, надо определить обработчики двух кнопок:

- для кнопки **Предварительный просмотр** записать в обработчик:
  ```
  QuickRep1->Preview();
  ```
- для кнопки **Печать** в обработчик записать:
  ```
  QuickRep1->Print();
  ```

Можно печатать отчет и после предварительного просмотра, воспользовавшись кнопками окна **Print Previw**. С их помощью отчет можно писать в файл и загружать из файла, уменьшать или увеличивать его изображение на экране.

Пример отчета, печатающего изображения

Мы выведем данные таблицы Клиенты из БД BCDEMOS. Вид формы с компонентами показан на рис. 13.5.

Для вывода изображения (поле `Image`) использован компонент `TQRDBImage`, настраиваемый так же, как и `TQRDBText`.

Для вывода номера страницы и даты вывода в форму вставлены два компонента `TQRSysData`. В их свойство `Text` внесены названия элементов (номер страницы — `Стр N` и `Дата`). Из выпадающего списка свойства `Data` в одном случае выбрано значение `qrsPageNumber`, а в другом — `qrsDate` (для отображения системной даты).

В полосу `rbDetail` вставлен в иллюстративных целях компонент `QRImage`, который выводит логотип продукта, который мы изучаем.

Результат печати данных таблицы Клиенты показан на рис. 13.6.

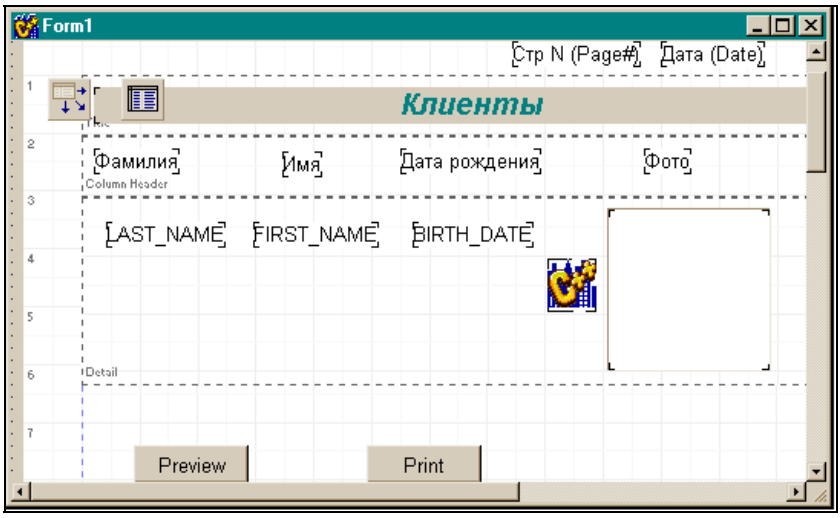


Рис. 13.5. Вид формы приложения для печати данных таблицы Клиенты

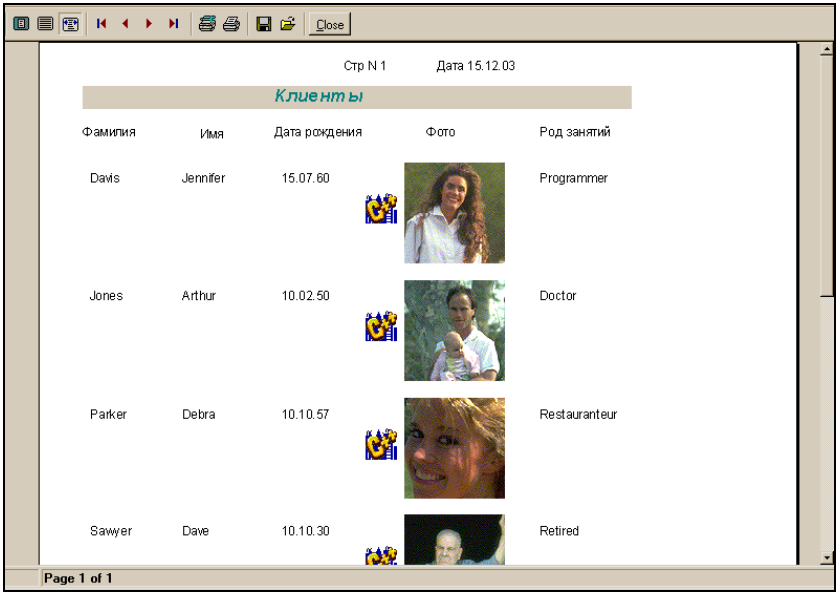
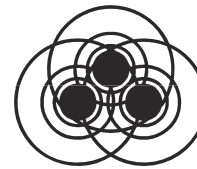


Рис. 13.6. Результат печати данных таблицы Клиенты

Глава 14



Некоторые компоненты вкладки *Internet*

Чтобы сетевые приложения (которые формируются с помощью компонентов вкладки **Internet**) работали корректно, на вашем компьютере должен быть установлен Персональный Web-сервер (PWS), который находится на диске с дистрибутивом Windows.

Компонент *TServerSocket*

Этот компонент делает ваше приложение сервером, работающим по протоколу TCP/IP (Transfer control protocol/internet protocol). *Протокол* — это язык, позволяющий компьютерам взаимодействовать с другими устройствами сети. Для взаимодействия друг с другом компьютеры должны использовать одинаковый протокол. Протокол TCP/IP применяется для подключения к Интернету и другим глобальным сетям. Серверное приложение содержит удаленный модуль данных, который включает в себя компонент "Провайдер набора данных" — механизм, обеспечивающий поставку данных клиентскому приложению. Для каждого набора данных существует свой провайдер. Провайдер набора данных делает следующее:

- получает запросы от клиента, выбирает затребованные клиентом данные из базы данных сервера, упаковывает данные для их передачи и пересылает данные клиентскому набору данных. Эти действия называются *провайдингом*;
- получает обновленные данные от клиентского набора данных, осуществляет на их основе корректировку базы данных (или источника данных), регистрирует поступившие обнов-

ления, которые не могут быть применены, и возвращает их клиенту для дальнейшей доработки. Эта часть работы провайдера называется *разрешением*.

Свойства *TServerSocket*

- `Active` — это свойство указывает, открыто ли так называемое *стыковочное соединение* и доступно ли оно для связи с другими машинами. Это свойство унаследовано от абстрактного класса `TAbstractSocket`, который содержит свойства и методы, позволяющие работать приложению со стыковочными компонентами (sockets) Windows, которые, в свою очередь, включают в себя набор коммуникационных протоколов (т. е. протоколов связи), позволяющих приложению соединяться с другими машинами для чтения и записи информации. Стыковочные компоненты Windows обеспечивают соединения, основанные на протоколе TCP/IP. Стыковочные компоненты позволяют приложению формировать соединения с другими машинами, не заботясь о деталях существующего сетевого программного обеспечения. Свойства компонента `TAbstractSocket` описывают IP-адрес стыковочного соединения и обслуживают его (IP-адрес — это код сетевого узла, представляющий собой строку чисел из четырех групп, разделенных между собой точками, например, 192.200.1.15). Не все потомки `TAbstractSocket` используют все эти свойства. Например, серверные стыковочные узлы не ищут IP-адрес, т. к. он явно читается из системы, запускающей приложение. Экземпляры класса `TAbstractSocket` явно не задаются, для этой цели используются его потомки: чтобы создать стыковочное соединение с другой машиной, используется класс `TClientSocket`, а для создания стыковочного соединения, которое отвечает на запросы о соединении, поступающие от других машин, используется класс `TServerSocket`, который мы и рассматриваем. Перед тем как использовать или изменить стыковочное соединение, надо посмотреть его свойство `Active`, чтобы определить, открыто ли соединение и готово ли оно к работе. Для клиентских стыковочных соединений (компонент `TClientSocket`) установка свойства `Active` открывает или закрывает стыковочное соединение с другой

машиной. Для серверных же стыковочных соединений установка свойства `Active` открывает или закрывает прослушивание запросов от клиентов (прослушивание — это процесс, когда сервер ждет появления запросов от других машин и устанавливает соединение с той из них, от которой он принял запрос). В режиме проектирования приложения установка свойства `Active` в `true` заставляет стыковочное соединение сервера открыть соединение (т. е. будет установлено прослушивание запросов от других машин), когда приложение запустится.

- `Port` — здесь задается числовой идентификатор, который идентифицирует серверное стыковочное соединение. Номера портов позволяют отдельной системе, заданной на стороне клиента свойствами `Host` или `Address`, принимать одновременно множество соединений.

Примечание

В свойстве `Host` задается URL, синоним IP-адреса, например, **www.mail.ru**, а в свойстве `Address` задается IP-адрес.

У свойства `Port` может быть много значений: это связано с обслуживанием данных, передаваемых по разным протоколам: например, по протоколу FTP (file transfer protocol — протокол передачи файлов) или HTTP (hypertext transfer protocol — протокол передачи гипертекста). Для серверных стыковочных соединений свойство `Port` — это идентификатор соединения, по которому серверное стыковочное соединение прослушивает клиентские запросы. Серверные стыковочные соединения устанавливают свойство `Port` в определенное значение, которое могут использовать клиенты для инициирования соединений. Для клиентских стыковочных соединений `Port` — это идентификатор порта сервера, с которым клиент желает осуществить соединение. Значение свойства `Port` обычно связано с видом обслуживания, которое требуется клиенту от серверного приложения. Изменение значения свойства `Port` должно происходить при неактивном соединении, иначе возникает исключительная ситуация типа `EsocketError`.

- `ServerType` — указывает, каким будет каждое принятое сервером соединение: либо оно автоматически выделяется в отдельный исполняемый поток, либо нет. Если `ServerType` установлено в `stThreadBlocking`, то для каждого стыковочного соединения, принятого серверным стыковочным соединением, автоматически выделяется новый поток. В этом случае исполнение потока не происходит, пока не будет прочитана или записана вся информация, проходящая через соединение. Для каждого соединения такой поток генерирует события `OnClientRead` или `OnClientWrite` всякий раз, когда серверный стыковочный узел требует чтения или записи. Если свойство `ServerType` имеет значение `stNonBlocking`, то обработка всех процессов чтения и записи информации, проходящих через стыковочные соединения, происходит асинхронно. В этом режиме все клиентские соединения обрабатываются по умолчанию в одном исполнительном потоке.
- `Service` — здесь задается имя службы, для которой используется стыковочное соединение. *Служба* — это подсистема сетевого программного обеспечения, которая выполняет некую конкретную задачу. Например, служба общего доступа к файлам и принтерам, служба автоматического резервного копирования и т. д. Windows обеспечивает множество стандартных служб: FTP, HTTP, служба времени и др. Серверы могут задавать дополнительные службы и связанные с ними порты в services-файлах. Некоторые номера портов зарезервированы для специальных служб. Для серверных стыковочных соединений использование свойства `Service` вместо свойства `Port` означает, что сервер будет прослушивать TCP/IP-запросы соответствующего ему порта.
- `Socket` — указывает на `TServerWinSocket`-объект, который описывает концевую точку прослушиваемого соединения. С помощью этого свойства можно получить (пользуясь свойствами и методами класса `TServerWinSocket`) следующие данные:
  - информацию о потоках соединения, которые сформированы для обработки серверным стыковочным соединением;

- информацию о соединениях в виде `Socket->Connections[int Index]` — массива открытых соединений с клиентскими стыковочными узлами для данного прослушивающего стыковочного узла;
- другие данные.

- `ThreadCacheSize` — здесь задается максимальное число потоков, используемых для новых клиентских соединений (см. комментарий к свойству `ServerType`).

Компонент *TClientSocket*

Этот компонент превращает ваше приложение в клиента сетевого соединения по протоколу TCP/IP.

Свойства *TClientSocket*

- `Active` — это свойство аналогично одноименному свойству компонента `TServerSocket`.
- `Address` — здесь задается IP-адрес серверного приложения. `Address` — это строка четырех чисел, разделенных между собой точкой в стандартной интернет-нотации. Например, 123.197.1.2. Для каждого стыковочного клиентского соединения надо придать свойству `Address` значение IP-адреса того сервера, с которым это стыковочное соединение должно быть установлено. Когда соединение открыто, значение `Address` связывается с соединением. Если стыковочное соединение содержит значение свойства `Host` вместо значения свойства `Address`, то IP-адрес извлекается из специальной таблицы. Одна серверная система может поддерживать более одного IP-адреса (об этом сказано выше).
- `ClientType` — определяет, как осуществляет клиентский стыковочный узел чтение и запись информации: асинхронно или нет. Если `ClientType` = `ctNonBlocking`, то клиентский стыковочный узел отвечает на асинхронные события записи и чтения.

- `Host` — это URL, синоним IP-адреса серверного приложения, например, **www.rambler.com**. Если задан IP-адрес, это свойство указывать не надо.
- `Port` — это номер порта стыковочного узла сервера, с которым клиент собирается осуществить соединение.
- `Service` — свойство аналогично одноименному свойству компонента `TServerSocket`.
- `Socket` — это свойство описывает клиентский стыковочный узел с помощью свойств и методов класса `TClientWinSocket`, указателем на который оно является. Например, свойство позволяет:
  - определить адрес и порт стыковочных соединений клиента и сервера, связанных с соединением;
  - читать и писать информацию через стыковочное соединение:
    ```
    AnsiString S=Socket->ReceiveText(),
    Socket->SendText(S);
    ```
  - определять, на какие замечания сервера должен отвечать клиент.

События *TClientSocket*

- `OnConnect` — происходит сразу после того, как установлено соединение со стыковочным узлом сервера. В зависимости от службы, заданной в соответствующем свойстве, в этот момент могли бы начаться чтение или запись через стыковочный узел клиента. Вообще, когда стыковочный узел открывает соединение, возникают следующие события:
  - `OnLookup` — возникает раньше попытки найти стыковочный узел сервера;
  - `OnConnecting` — возникает после того, как найден стыковочный узел сервера, требование соединения принято сервером и подготовлено клиентским стыковочным узлом;
  - `OnConnect` — возникает после того, как соединение установлено.
- `OnDisconnect` — происходит прямо перед тем, как клиентский стыковочный узел закрывает соединение с сервером.

OnDisconnect происходит после проверки того, что свойство Active установлено в false, но до закрытия существующего соединения.

- OnError — это реакция на ошибки, возникающие при работе стыковочного соединения. При обработке ошибки надо установить параметр ErrorCode обработчика в 0, иначе после выхода из обработчика возникнет исключительная ситуация типа ESocketError.
- OnLookup — это событие возникает перед попыткой найти серверное стыковочное соединение, с которым собирается соединиться данный клиент.
- OnRead — возникает при чтении информации из стыковочного соединения. Если стыковочное соединение — блокированное, то для чтения из него надо использовать объект TWinSocketStream. В противном случае надо использовать методы параметра Socket.

Примечание

В стыковочных соединениях Nonblocking для последнего бита данных, передаваемых через это соединение, не всегда возникает событие OnRead. Поэтому в этих случаях надо проверять непрочитанные данные в обработчике события OnDisconnect, чтобы быть уверенным, что все обработано верно.

- OnWrite — возникает тогда, когда клиентское стыковочное соединение пишет информацию в стыковочное соединение сервера. Если стыковочное соединение блокировано, то для записи следует использовать методы объекта TWinSocketStream. В противном случае используются методы параметра Socket.

Пример соединения по протоколу TCP/IP

Приведем пример использования компонентов TServerSocket и TClientSocket в приложении, обеспечивающем обмен сообщениями между клиентом и сервером. Форма приложения вместе с

Инспекторами объекта для компонентов формы и главным меню приложения показана на рис. 14.1.

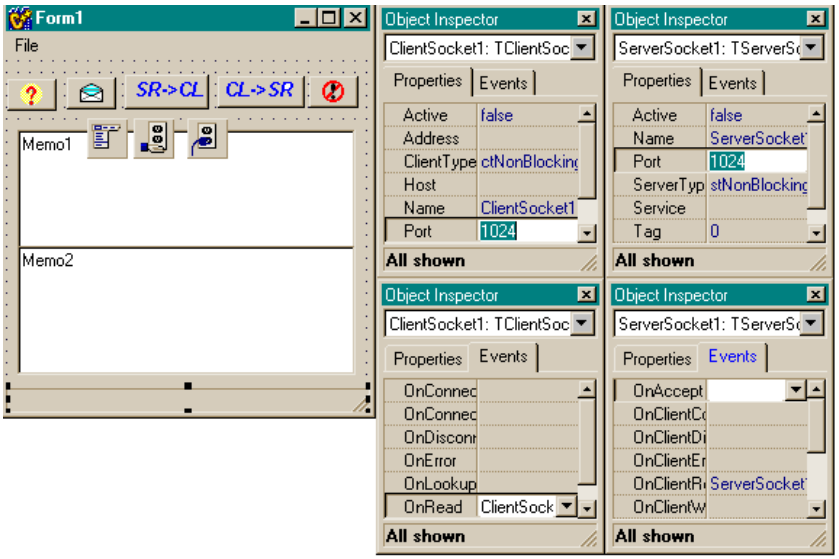


Рис. 14.1. Форма приложения, использующего `TServerSocket` и `TClientSocket`

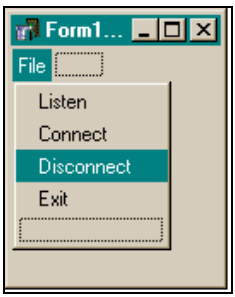


Рис. 14.2. Команды подменю **File** главного меню приложения

Суть работы приложения состоит в том, что после запуска приложения выполняется прослушивание сервером клиентов. Ре-

жим запускается либо командой **File/Listen** меню приложения (рис. 14.2), либо дублирующей эту команду кнопкой со знаком вопроса (все кнопки имеют всплывающие подсказки).

В нижней части формы расположен компонент `TStatusBar`, в поле которого отображаются производимые действия. После запуска прослушивания можно запускать соединение с сервером со стороны клиента (командой **File/Connect** меню или кнопкой с изображением конверта). В этом случае запрашивается URL сервера с помощью функции `InputQuery()` (рис. 14.3).

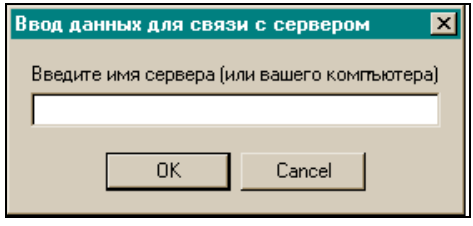


Рис. 14.3. Запрос на ввод URL сервера

Если вы хотите проверить работу компонентов на своей (локальной) машине, то в ответ на запрос следует задать имя вашего компьютера, которое можно найти либо в реестре Windows (окно **Редактор реестра** можно открыть, если выбрать опции **Пуск/Выполнить** и в поле **Открыть** задать regedit), либо с помощью команды **Сведения о системе**, до которой можно добраться так: **Пуск/Программы/Стандартные/Служебные**.

В приведенном примере сервер и клиент находятся на одной машине. Когда соединение между клиентом и сервером установлено, можно отсылать сообщения от сервера клиенту и наоборот, используя соответствующие кнопки (***SR->CL***, ***CL->SR***). В первом случае сначала текст сообщения набирается в поле **Memo1** и после этого нажимается кнопка отсылки. Во втором — текст сообщения набирается в поле **Memo2** и после этого нажимается кнопка отсылки. Результаты от сервера попадают в **Memo2**, и соответственно от клиента — в **Memo1**. Сообщения посылаются методом `SendText()` подсвойства `Connections[]`, которое представляет собой массив соединений. Из этого массива мы выбираем только одно, наше, единственное соединение

свойства `Socket` серверного компонента. Таким же методом сообщения пересылаются от клиента серверу через свойство `Socket` клиентского компонента. После того как сообщение отослано, соответствующий абонент может его получить методом `ReceiveText()` свойства `Socket`.

Форма в режиме исполнения после нажатия кнопки со знаком вопроса приводится на рис. 14.4.

После нажатия кнопки с изображением конверта или выбора команды **Connect** меню **File** на экране появится запрос на ввод имени сервера (рис. 14.5).

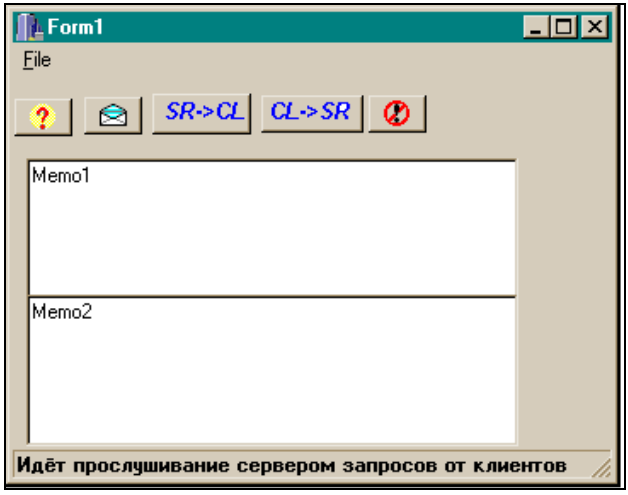


Рис. 14.4. Форма в режиме исполнения после нажатия кнопки со знаком вопроса

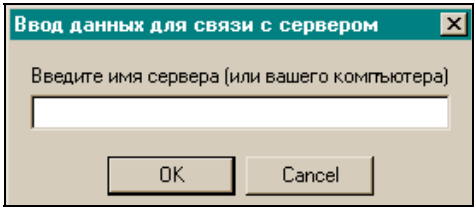


Рис. 14.5. Запрос на ввод имени сервера

Если соединение установлено, то появится картинка, изображенная на рис. 14.6.

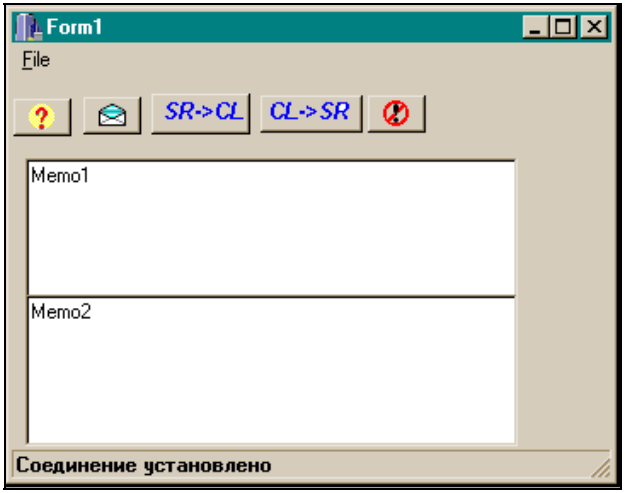


Рис. 14.6. Форма при установленном соединении

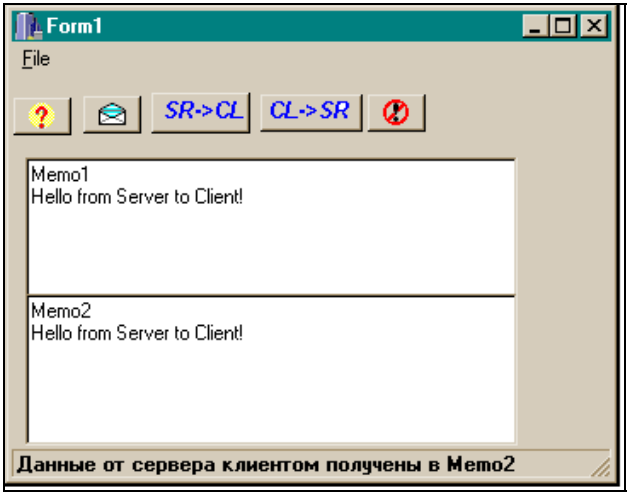


Рис. 14.7. Получение клиентом сообщения от сервера

Теперь можно отсылать сообщения. Сообщение от сервера клиенту пойдет после нажатия кнопки ***SR->CL***. Предварительно текст сообщения набирается в поле **Memo1**. Результат представлен на рис. 14.7.

Ответное сообщение (от клиента серверу) надо набрать в поле **Memo2** и нажать на кнопку отсылки сообщения ***CL->SR***. Результат представлен на рис. 14.8.

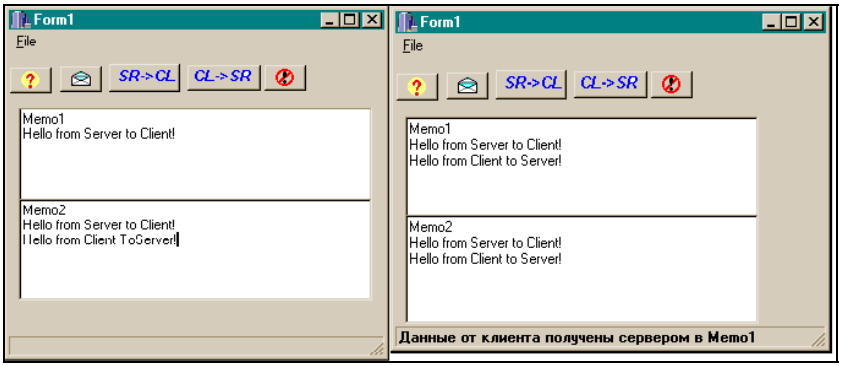


Рис. 14.8. Получение сервером ответа клиента

Разорвать соединение можно с помощью опции меню **Disconnect** или нажав кнопку с рисунком в виде красной окружности, внутри которой находится перечеркнутый восклицательный знак.

Текст программы приведен в листинге 14.1.

Листинг 14.1

```
//cpp-файл

//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1_ServerSocket.h"
//---------------------------------------------------------------
```

```
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Listen1Click(TObject *Sender)
{
ClientSocket1->Active=false;
ServerSocket1->Active=true;
StatusBar1->SimpleText="Идет прослушивание сервером запросов
от клиентов";
}
//---------------------------------------------------------
void __fastcall TForm1::SpeedButton1Click(TObject *Sender)
{
//соединение с сервером, если он в состоянии прослушивания
 if(ClientSocket1->Active)
  ClientSocket1->Active=false; //Для возможности изменения
                               //свойств этого компонента
 InputQuery("Ввод данных для связи с сервером","Введите имя
сервера (или вашего компьютера)",ClientSocket1->Host);
 ClientSocket1->Active=true;
 StatusBar1->SimpleText="Соединение установлено";
}
//---------------------------------------------------------
void __fastcall TForm1::Connect1Click(TObject *Sender)
{
 //соединение с сервером, если он в состоянии прослушивания
 if(ClientSocket1->Active)
  ClientSocket1->Active=false; //Для возможности изменения
                               //свойств этого компонента
 InputQuery("Ввод данных для связи с сервером","Введите имя
сервера (или вашего компьютера)",ClientSocket1->Host);
```

```
 ClientSocket1->Active=true;
 StatusBar1->SimpleText="Соединение установлено";
}
//---------------------------------------------------------
void __fastcall TForm1::Exit1Click(TObject *Sender)
{
 Form1->Close();
}
//---------------------------------------------------------
void __fastcall TForm1::SpeedButton2Click(TObject *Sender)
{
 //Разрыв соединения
 ClientSocket1->Close();
 ServerSocket1->Close();
 StatusBar1->SimpleText="Соединение разорвано";
}
//---------------------------------------------------------
void __fastcall TForm1::Disconnect1Click(TObject *Sender)
{
 //Разрыв соединения
 ClientSocket1->Close();
 ServerSocket1->Close();
 StatusBar1->SimpleText="Соединение разорвано";
}
//---------------------------------------------------------


//---------------------------------------------------------
void __fastcall TForm1::ClientSocket1Read(TObject *Sender,
   TCustomWinSocket *Socket)
{
//Клиент читает текст, посланный ему сервером
//через свое стыковочное соединение в поле Memo2
 Memo2->Lines->Add(Socket->ReceiveText());
 StatusBar1->SimpleText="Данные от сервера клиентом получены
 в Memo2";
}
```

```
//---------------------------------------------------------
void __fastcall TForm1::SpeedButton3Click(TObject *Sender)
{
ClientSocket1->Active=false;
ServerSocket1->Active=true;
StatusBar1->SimpleText="Идет прослушивание сервером запросов
от клиентов";
}
//---------------------------------------------------------
void __fastcall TForm1::SpeedButton4Click(TObject *Sender)
{
//Обработка ввода в поле Memo1 для отсылки сообщения клиенту

 ServerSocket1->Socket->Connections[0]->SendText(
 Memo1->Lines->Strings[Memo1->Lines->Count - 1]);
 StatusBar1->SimpleText="Ввод данных в Memo1 для их отсылки
 клиенту";
}
//---------------------------------------------------------
void __fastcall TForm1::SpeedButton5Click(TObject *Sender)
{
//Обработка ввода в Memo2-поле для отсылки сообщения серверу

 ClientSocket1->Socket->SendText(
 Memo2->Lines->Strings[Memo2->Lines->Count - 1]);
 StatusBar1->SimpleText="Ввод данных в Memo2 для их отсылки
 серверу в Memo1";
}
//---------------------------------------------------------
void __fastcall TForm1::ServerSocket1ClientRead(TObject
*Sender,
   TCustomWinSocket *Socket)
{
//Сервер читает текст, посланный ему клиентом
//через свое стыковочное соединение в поле Memo1
 Memo1->Lines->Add(Socket->ReceiveText());
```

```
 StatusBar1->SimpleText="Данные от клиента получены сервером
 в Memo1";
}
//---------------------------------------------------------------
//h-модуль

//---------------------------------------------------------------
#ifndef Unit1_ServerSocketH
#define Unit1_ServerSocketH
//---------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Menus.hpp>
#include <ScktComp.hpp>
#include <ComCtrls.hpp>
#include <Buttons.hpp>
//---------------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
    TClientSocket *ClientSocket1;
    TServerSocket *ServerSocket1;
    TMainMenu *MainMenu1;
    TMenuItem *File1;
    TMenuItem *Connect1;
    TMenuItem *Disconnect1;
    TMenuItem *Listen1;
    TMenuItem *Exit1;
    TStatusBar *StatusBar1;
    TMemo *Memo1;
    TMemo *Memo2;
    TSpeedButton *SpeedButton1;
    TSpeedButton *SpeedButton2;
    TSpeedButton *SpeedButton3;
```

```
    TSpeedButton *SpeedButton4;
    TSpeedButton *SpeedButton5;
    void __fastcall Listen1Click(TObject *Sender);
    void __fastcall SpeedButton1Click(TObject *Sender);
    void __fastcall Connect1Click(TObject *Sender);
    void __fastcall Exit1Click(TObject *Sender);
    void __fastcall SpeedButton2Click(TObject *Sender);
    void __fastcall Disconnect1Click(TObject *Sender);
    void __fastcall ClientSocket1Read(TObject *Sender,
     TCustomWinSocket *Socket);
    void __fastcall SpeedButton3Click(TObject *Sender);
    void __fastcall SpeedButton4Click(TObject *Sender);
    void __fastcall SpeedButton5Click(TObject *Sender);
    void __fastcall ServerSocket1ClientRead(TObject *Sender,
     TCustomWinSocket *Socket);
private:   // User declarations
public:   // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Компонент *TCppWebBrowser*

С помощью этого компонента из вашего приложения можно запустить обозреватель Internet Explorer. Это можно сделать с помощью метода `Navigate`, который находит ресурс либо по пути к нему (для локального документа), либо по URL (для сетевого документа). Для локального документа можно задать параметры метода `Navigate` так:

```
WideString Path=Edit1->Text;
   Variant flag=1;
   CppWebBrowser1->Navigate(Path,flag,NULL,NULL,NULL);
```

Здесь `Edit1->Text` должен содержать введенный вами путь к файлу с расширением html, `flag=1` — параметр, говорящий о том, что ресурс (локальный или сетевой) будет открыт в новом окне.

Для сетевого документа можно записать:

```
WideString URL="www.mail.ru";
   Variant flag=1;
   CppWebBrowser1->Navigate(URL,flag,NULL,NULL,NULL);
```

Пример приложения, запускающего Internet Explorer для вывода локального документа

Форма приложения показана на рис. 14.9, форма приложения после компиляции — на рис. 14.10, а на рис. 14.11 показан результат работы приложения. Текст программы приведен в листинге 14.2.

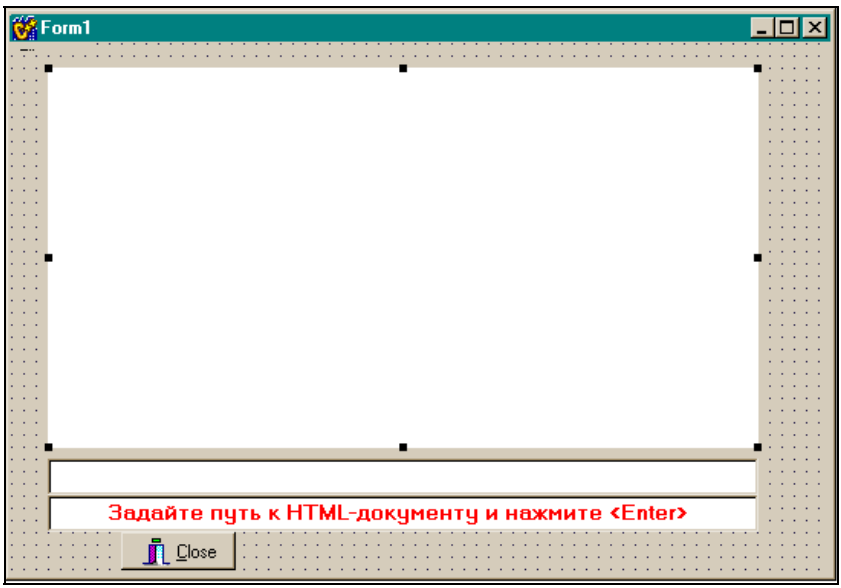


Рис. 14.9. Форма приложения для запуска Internet Explorer

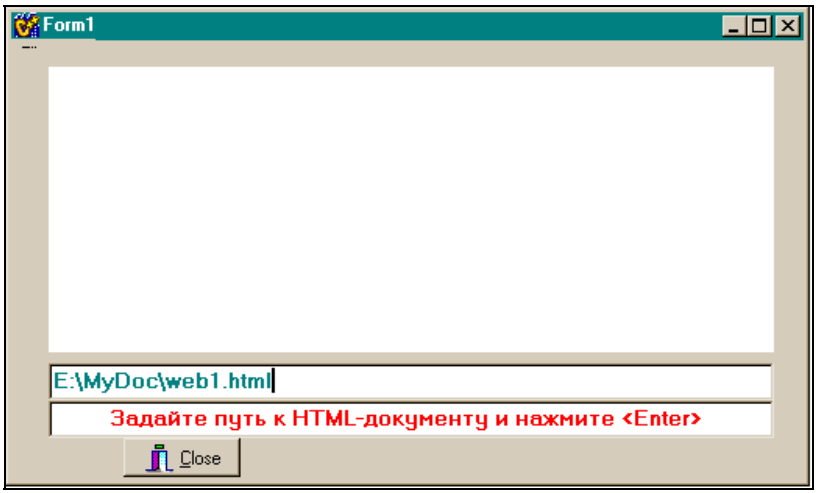


Рис. 14.10. Форма скомпилированного приложения для запуска Internet Explorer

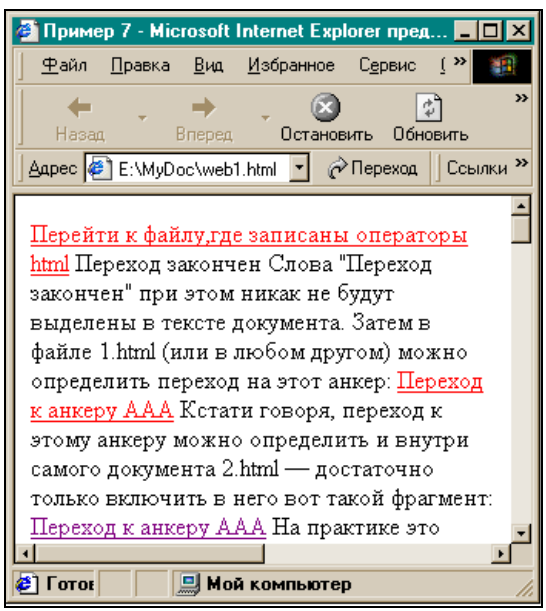


Рис. 14.11. Результат вызова Internet Explorer из приложения

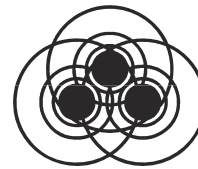
Листинг 14.2

//**cpp-файл**

```
//---------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop

#include "Unit1.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma link "SHDocVw_OCX"
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{
}
//---------------------------------------------------------------
void __fastcall TForm1::Edit1KeyDown(TObject *Sender, WORD
&Key,
   TShiftState Shift)
{
 if(Key==VK_RETURN)
 {
   WideString Path=Edit1->Text;
   Variant flag=1;
   CppWebBrowser1->Navigate(Path,flag,NULL,NULL,NULL);
 }
}
//---------------------------------------------------------------
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
```

```
 Form1->Close();
}
//---------------------------------------------------------
//h-файл

//---------------------------------------------------------
#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include "SHDocVw_OCX.h"
#include <OleCtrls.hpp>
#include <Buttons.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:   // IDE-managed Components
    TCppWebBrowser *CppWebBrowser1;
    TEdit *Edit1;
    TEdit *Edit2;
    TBitBtn *BitBtn1;
    void __fastcall Edit1KeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift);
    void __fastcall BitBtn1Click(TObject *Sender);
private:   // User declarations
public:   // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Глава 15



Примеры из технологии MIDAS

Технология MIDAS (службы многоуровневых распределенных приложений) позволяет построить систему "клиент-сервер" без использования механизмов — посредников BDE/ADO. Для этой цели служат несколько компонентов. В этой системе строится отдельно серверная, отдельно клиентская часть приложения. Серверная часть может быть на удаленном компьютере, клиентская — на вашем. Систему можно смоделировать и на одном компьютере. Рассмотрим необходимые компоненты.

Компонент *TDataSetProvider*

Этот компонент (провайдер набора данных) располагается в серверной части приложения и служит для передачи информации из набора данных BDE/ADO клиентскому приложению, на котором установлен компонент `TClientDataSet`. В клиентском приложении набор данных может быть модифицирован и отослан назад на сервер через провайдер.

Свойства *TDataSetProvider*

- `DataSet` — здесь задается набор данных, из которого провайдер пересылает данные и в который он пересылает модифицированные клиентом данные. Провайдер упаковывает записи из набора данных и пересылает их в клиентский набор данных. Набор данных должен быть задан до установки провайдера. Если свойство `ResolveToDataSet` установлено в

`true`, провайдер применяет обновления данных от серверного приложения к набору данных, указанному в свойстве `DataSet`. Иначе провайдер применяет обновления данных к основному серверу базы данных, связанному с набором данных.

- `Exported` — позволяет клиентскому приложению передавать вызовы провайдеру. `Exported` используется, чтобы указать, становится ли провайдер доступным клиенту с помощью удаленного модуля данных, который содержит этот провайдер. Когда `Exported` имеет значение `true`, клиентское приложение может указать провайдер в качестве цели вызова для интерфейса IAppServer (этот интерфейс предоставляется в среде MIDAS для связи клиента с сервером). Когда же `Exported` имеет значение `false`, указание провайдера для интерфейса `IAppServer` вызывает исключительную ситуацию.

  Примечание

  В дополнение к свойству `Exported`, провайдер должен быть зарегистрирован с удаленным модулем данных для клиентских приложений, чтобы иметь к нему доступ с использованием интерфейса модуля данных IAppServer. Когда собственником провайдера является удаленный модуль данных, провайдер регистрируется автоматически (при первом запуске серверного приложения). Если же провайдер имеет другого собственника, серверное приложение должно явно (т. е. само) регистрировать провайдера, используя метод `RegisterProvider` из удаленного модуля данных (в примере, который будет приведен ниже, серверное приложение как раз и формируется с использованием удаленного модуля данных, который извлекается из хранилища объектов).

- `Options` — это свойство задает режимы, определяющие, что будет включено в пакет данных, формируемый провайдером, и как информация из пакета будет использована. В этом свойстве настраиваются связи провайдера с клиентскими наборами данных.

  Режимы задают:

  - будут ли BLOB-данные включаться в пакет или выбираться отдельно;

- будут ли пакеты включать такие свойства полей компонента, как форматы, выводимые имена, минимальные и максимальные значения;
- будут ли пакеты предназначены только для чтения и запрещены ли специфические изменения (вставки, редактирования или удаления);
- будут ли изменения в полях главной таблицы вызывать соответствующие изменения в связанной таблице;
- воздействует ли единственное клиентское обновление данных на множество серверных записей;
- станут ли клиентские записи обновляться после их модификации, может ли клиент послать SQL-операторы, которые перекрывают провайдерский набор данных.

- `Resolver` — обеспечивает доступ к компоненту `TDataSetResolver`, который применяет обновления к набору данных и разрешает ошибки обновления.
- `ResolveToDataSet` — свойство определяет, будут ли применены обновления данных к самому набору данных, с которым связан провайдер, или прямо к серверу базы данных. Когда `ResolveToDataSet` имеет значение `true`, то свойство `Resolver` устанавливается в компоненте `TDataSetResolver`, который применяет обновления непосредственно к набору данных, указанному в свойстве `DataSet`. Это может быть полезным в случае, когда приложение использует обработку событий набора данных или если набор данных не представляет данные с сервера базы данных (например, клиентский набор данных). Когда же `ResolveToDataSet` установлено в `false`, свойство `Resolver` устанавливается в компоненте `TSQLResolver`, который применяет обновления непосредственно к серверу базы данных, связанному с набором данных, указанным в свойстве провайдера `DataSet`.
- `UpdateMode` — это свойство задает критерии поиска провайдером записей, когда надо применить обновления к набору данных. `UpdateMode` указывает, будут ли модифицируемые записи искаться на базе всех полей, только с использованием ключевых полей или с использованием ключевых полей и полей, которые были модифицированы.

Компонент *TClientDataSet*

Это клиентский набор данных. Компонент устанавливается в клиентском приложении. Он может выполнять две функции: быть обычным полнофункциональным набором данных, в который можно записывать данные и из которого их можно читать, минуя механизмы BDE/ADO. Или быть копией набора данных, получаемых клиентским приложением от серверного приложения.

Свойства *TClientDataSet*

- `Agregates` — если открыть кнопкой с многоточием, расположенной в поле этого свойства, окно и выполнить команду **Add** его контекстного меню, то в окне появится компонент `TClientDataSet.Aggregates[0]` класса `TAggregate`, свойства которого отобразятся в Инспекторе объекта.

  С помощью свойств массива `Aggregates[]` можно задать агрегированную обработку записей клиентского набора данных. Задавая разные строки в окне редактора агрегированных данных (назовем их "агрегирования") и определяя различные формулы вычислений над данными клиентского набора через свойство `Expression`, можно проводить групповую обработку данных не только всех записей набора, но и его групп и подгрупп, имеющих одинаковые значения по установленным полям. "Агрегирования", которые выполняют обработку групп данных, связаны индексами. Они могут применяться только в том случае, когда связанный индекс — текущий. "Агрегирования" не используются для значений, поддерживаемых клиентским набором данных. Свойство `Expression` задает, как будут агрегироваться значения. Это свойство включает в себя следующие операторы агрегирования данных:

  - `Sum` — суммирует значения для числовых полей или выражений;
  - `Avg` — вычисляет среднее значение выражений, числовых или полей, содержащих дату или время;

- `Count` — указывает число непустых значений для полей или выражений;
- `Min` — указывает минимальное значение для выражения, строки, числа или поля даты или времени;
- `Max` — указывает максимальное значение для выражения, строки, числа или поля даты или времени.

Операторы агрегированной обработки действуют на значения полей или выражений, построенных из значений полей, и используют те же операторы, которые применяются для создания фильтров. Вы можете создавать выражения с использованием операторов над итоговыми значениями с другими итоговыми значениями или с константами. Однако нельзя комбинировать итоговые значения и значения полей, потому что такие выражения являются неоднозначными (по ним не видно, из какой записи надо выбрать значение поля). Эти правила иллюстрируются следующими выражениями:

```
Sum(Qty * Price)    // правильно — суммирование
                    // выражения из полей

Max(Field1) - Max(Field2)     // правильно — выражение
                              // из функций

Avg(DiscountRate) * 100    // правильно — выражение из
                           // функции и константы

Min(Sum(Field1))    // неправильно — вложенные функции

Count(Field1) - Field2    // неправильно — выражение
                          // из функции и поля
```

- `AgregatesActive` — указывает, вычисляет ли клиентский набор данных агрегированные значения.
- `CommandText` — здесь задается команда на языке SQL для исполнения на сервере. Это свойство используется, когда надо переопределить SQL-запрос в серверном приложении или заменить таблицу или хранимую процедуру на серверном приложении. Если задано свойство `CommandText`, его значение пересылается серверному приложению, когда клиентское приложение открывается и когда вы вызываете метод `Execute()`. Заданные в свойстве команды SQL переопределяют таблицу или хранимую процедуру набора данных, связанного с провайдером на серверном приложении. Если строка-

команда содержит параметры, убедитесь, что они расположены в правильном порядке, т. к. серверное приложение применяет эти параметры только по их индексу. `CommandText` не работает, если не включен режим `poAllowCommandText` в свойстве провайдера `Options`.

- `DataSetField` — это свойство используется, если клиентский набор данных связывается (подчиняется) набору другого (главного) клиентского набора данных. Тогда оба набора данных будут связаны между собой по этому полю. Когда устанавливается свойство `DataSetField`, свойства `ProviderName`, `RemoteServer` и `FileName` клиентского набора данных очищаются, потому что клиентские данные теперь будут поддерживаться главной таблицей, в которой эти свойства определены.
- `Params` — с помощью этого свойства задаются параметры, которые пересылаются серверному приложению. Значения параметров используются, чтобы передать некоторые величины либо компоненту `TQuery`, либо хранимой процедуре (`TStoredProcedure`) серверного приложения или передать лимит на число записей пакета, формируемого провайдером. После задания параметров они автоматически пересылаются серверному приложению, когда клиентский набор данных выбирает данные или выполняет запросы через компоненты `TQuery` или `TStoredProcedure` на удаленном сервере. Чтобы сформировать сами параметрические значения, надо открыть окно Редактора параметров, нажав кнопку с многоточием в поле свойства `Params`. При этом откроется окно Редактора, в котором надо нажать кнопку **Add** или выполнить такую же опцию в контекстном меню Редактора. В окне появится новая строка-объект со своим Инспектором объекта (рис. 15.1).

  Структура объекта `TParams` и его свойства аналогичны рассмотренным для компонента `TQuery`. Чтобы отослать параметры, ограничивающие количество значений в пакете (лимит на данные), надо создать параметр для каждого поля записи таблицы, из которой пойдет выборка, дать такому параметру те же значения `Name` и `DataType`, что и в соответствующем поле таблицы. Эти параметры будут автоматически пересланы серверному приложению, когда клиентский набор данных начнет выборку с серверного набора. Тогда в пакете,

сформированном провайдером серверного приложения, будут находиться только те записи, у которых поля имеют значения, заданные в параметрах. Вы можете добиться того же результата, установив соответствующий фильтр.

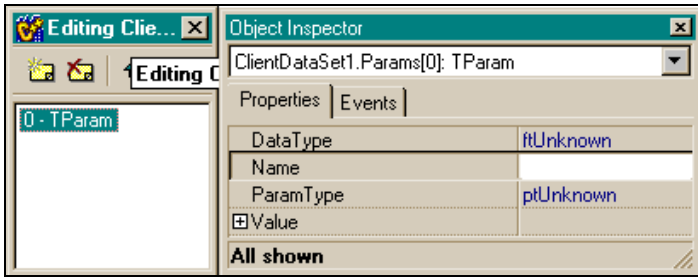


Рис. 15.1. Задание свойств параметра

- `ProviderName` — здесь задается имя провайдера, установленного на серверном приложении. Связь с провайдером обеспечивает интерфейс `IAppServer`, принадлежащий удаленному модулю данных, на котором установлен провайдер.
- `RemoteServer` — здесь задается имя компонента, обеспечивающего соединение с удаленным сервером со стороны клиента (это один из компонентов: `TDCOMConnection`, `TSocketConnection`, `TWebConnection` — из них выбирается тот, который обеспечивает соединение по заданному протоколу: DCOM, TCP/IP или HTTP).

Компонент *TDCOMConnection*

Этот компонент обеспечивает соединение клиентского приложения с удаленным сервером при поддержке службы безопасности MTS (Сервер транзакций компании Microsoft).

Свойства *TDCOMConnection*

- `ComputerName` — здесь указывается имя машины, на которой расположено серверное приложение (имя выбирается в окне

Выбор удаленного сервера, открываемом нажатием кнопки с многоточием в поле этого свойства). Если свойство `ComputerName` пусто (по умолчанию), `TDCOMConnection` "считает", что серверное приложение находится на машине пользователя (т. е. на той, на которой установлен этот компонент) или что имя машины будет образовано с помощью системного регистра на основе DCOM-конфигурации. Если же `ComputerName` не пусто, оно перекрывает любые данные системных регистров. В нормальных случаях клиентское и серверное приложения находятся на разных машинах, поэтому `ComputerName` должно быть установлено клиентским приложением или доступно как часть зарегистрированной DCOM-конфигурации.

- `Connected` — указывает, соединен ли объект `TDCOMConnection` с серверным приложением. Установить `Connected` в `true`, значит установить соединение с серверным приложением. Установить `Connected` в `false`, значит разорвать соединение. Когда мы устанавливаем значение свойства `Connected` в `true`, `TCOMConnection` может выводить диалог регистрации в зависимости от значения свойства `LoginPrompt`. Свойство `Connected` использует методы `DoConnect` и `DoDisconnect`, чтобы соединиться или разорвать соединение с серверным приложением.
- `LoginPromt` — определяет, появится или нет диалог регистрации прямо перед открытием нового соединения. Установите `LoginPrompt` в `true`, чтобы обеспечить поддержку регистрации во время установки соединения. В этом случае появляется диалог для задания имени пользователя и пароля для доступа к соединению. Появление диалога зависит также от типа компонента соединения: для компонентов соединения из MIDAS диалог появляется после наступления события `OnGetUsername` и перед событиями `BeforeConnect`, `AfterConnect` и `OnLogin`. Если пользователь отменяет диалог регистрации, то попытки выполнить соединение не производится.
- `ObjectBroker` — здесь задается имя объектного брокера, который позволяет компоненту соединения выбирать, с каким серверным приложением соединяться. Этот компонент под-

держивает список серверов, которые, в свою очередь, поддерживают приложение, указанное в свойствах `ServerName` или `ServerGuid`. Когда компоненту соединения необходимо открыть некоторое соединение, он требует информацию для идентификации серверной машины от объектного брокера. Приложения могут использовать в качестве объектного брокера компонент `TSimpleObjectBroker` или создать свой собственный объектный брокер на основе компонента `TCustomObjectBroker`.

- `ServerGuid` — здесь задается имя идентификатора (GUID) серверного приложения, с которым требуется соединение. Использование `ServerGuid` вместо `ServerName` для идентификации серверного приложения более предпочтительно, так как не требует регистрации серверного приложения в клиентской системе.

Примечание

Мы работаем с COM-объектами (Component Object Model) по протоколу DCOM. COM-клиенты общаются с объектами через COM-интерфейсы. *Интерфейсы* — это группы связанных подпрограмм, которые обеспечивают связь между провайдером некоторой службы (серверным объектом) и ее клиентами. Интерфейсы гарантируют каждому серверу уникальную идентификацию, названную GUID (Globally Unique Identifier — глобальный уникальный идентификатор) — это 128-битовое случайно сгенерированное число.

- `ServerName` — здесь из раскрывающегося списка выбирается имя серверного приложения (так как такое приложение должно быть создано заранее), с которым требуется установить соединение. В раскрывающемся списке имена серверов появятся при выполнении следующих условий:
  - для DCOM-соединений серверы должны быть зарегистрированы в системном Реестре (клиентской машины). Для Socket-соединений (соединения по протоколу TCP/IP) на серверной машине должна быть запущена программа ScktSrvr.exe;

- для Web-соединений (соединения по протоколу HTTP) компонент соединения должен иметь URL для поиска программы httpsrvr.dll.

Если заданное значение `ServerName` правильно, то свойство `ServerGuid` устанавливается в значение идентификатора GUID, который соответствует имени серверного приложения.

Компонент *TSocketConnection*

Этот компонент предназначен для организации соединения с серверным приложением на основе протокола TCP/IP и без принятых в DCOM средств защиты от коммуникационных сбоев (служба MTS).

`TSocketConnection` используется в клиентской части многозвенного приложения базы данных для установки и поддержки соединения клиентского приложения с удаленным серверным приложением. Чтобы использовать `TSocketConnection`, на сервере приложений должна быть запущена программа ScktSrvr.exe.

Свойства *TSocketConnection*

- `Adress` — здесь задается IP-адрес стыковочного соединения серверного приложения. Если компонент соединения использует объектный брокер (например, компонент `TSimpleObjectBroker`), задавать свойство `Address` не надо. Объектный брокер сам найдет адрес сервера в режиме исполнения.
- `Connected` — если придать этому свойству значение `true`, то будет установлено соединение с серверным приложением. Чтобы разорвать соединение, надо установить `Connected` в `false`.
- `Host` — здесь задается доменное имя, синоним IP-адреса серверного приложения, с которым надо установить связь. Свойства `Adress` и `Host` — взаимоисключающие.
- `InterceptGUID` — перехватчик GUID.

TSocketConnection использует класс IDataIntercept перед тем, как сообщения посланы серверу и после того, как ответы от сервера получены. Разработчики могут создать свой компонент (COM-объект), который содержит IDataIntercept, чтобы обрабатывать данные, проходящие через интерфейс между сервером и клиентом. Например, можно использовать методы из класса IDataIntercept, чтобы шифровать и дешифровывать данные или сжимать и восстанавливать их (методы DataIn, DataOut, информацию о которых можно посмотреть в разделе **Help** системы). В свойстве InterceptGUID задается идентификатор (GUID) этого COM-объекта. Тогда этот объект станет перехватывать сообщения, проходящие через стыковочное соединение перед их отсылкой от клиента и после того, как они получены клиентом. При использовании свойства InterceptGUID со стороны клиента сервер должен использовать тот же самый COM-объект, чтобы иметь возможность отменять всякие трансформации, произведенные с данными перехватчиком InterceptGUID. Для этого надо установить серверное свойство InterceptGUID в то же значение GUID, используя при этом диалоговое окно, появляющееся, когда на сервере запускают программу ScktSrvr.exe.

- LoginPromt — определяет, появится или нет диалог регистрации прямо перед открытием нового соединения.
- Port — здесь задается номер порта, который использует диспетчер стыковочного соединения (он устанавливается на сервере), чтобы прослушивать клиентов. По умолчанию значение свойства Port равно 211. Это номер порта по умолчанию, установленный в программе ScktSrvr.exe. Если диспетчер использует другой порт, в свойство Port надо записать значение, используемое диспетчером. Но если соединение использует объектный брокер (например, компонент TSimpleObjectBroker), свойство Port задавать не надо. Объектный брокер сам определит динамически номер порта сервера.
- SupportCallbacks — поддержка повторных вызовов. Это свойство задает, может ли обрабатывать компонент повторные вызовы в интерфейсе серверного приложения.

Компонент *TWebConnection*

Этот компонент обеспечивает соединение клиентского приложения с Web-сервером. Соединение идет по протоколу HTTP.

Свойства *TWebConnection*

- `Proxy` — здесь перечисляются доменные имена (синонимы IP-адресов) вспомогательных серверов, с которыми надо установить соединение. Это свойство используется, когда `TWebConnection` не может обработать свойство `Host`. Если сервер зарегистрирован со стороны клиента, свойство `Proxy` может иметь нулевое значение.
- `ProxyByPass` — обход `Proxy`. Если свойство `Host` указывает на сервер из списка, заданного в этом свойстве, `WebConnection` осуществляет соединение непосредственно с данным сервером, даже если свойство `Proxy` задано. Если свойство `ProxyByPass` не задано, `WebConnection` проверяет Регистр на предмет локальной регистрации сервера и обходит вспомогательный сервер, если свойство `Host` задает зарегистрированный на машине клиента сервер.
- `URL` — здесь задается поисковый адрес для библиотеки httpsrvr.dll на сервере. Это библиотека DLL, которая всегда находится в каталоге BIN, располагается на той же машине, что и Web-сервер, поэтому она может принимать HTTP-запросы и преобразовывать их в COM-вызовы, которые связываются с серверным приложением. URL включает протокол (HTTP или HTTPS), доменное имя и сценарное имя для httpsrvr.dll. Обычно URL имеет вид:

  ```
  http://MIDASHost.org/scripts/httpsrvr.dll
  ```

- `UserName` — указывает имя пользователя, под которым клиент регистрируется на серверном приложении.

Использование компонента *TClientDataSet*. Пример 1

Это пример использования компонента `TClientDataSet` в качестве обычного источника данных без сервера. К нему подключается компонент `TDataSetProvider`, который извлекает из таблиц или запросов наборов BDE/ADO данные и передает их компоненту `TClientDataSet`. Чтобы из последнего вывести данные на экран, используется компонент `TDBGrid`, который не может работать без источника данных `TDataSource`. Поэтому источник данных связывается с компонентом `TClientDataSet`. Все это происходит на локальном компьютере. Цепочка получается такая:

```
DBGrid <--- DataSource <--- ClientDataSet <---
DataSetProvider <--- TTable
```

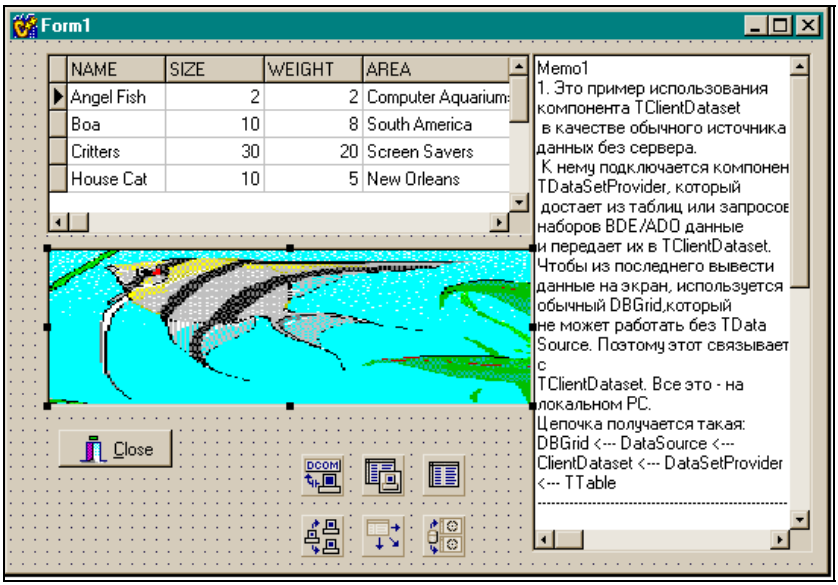


Рис. 15.2. Вид формы приложения с компонентами в ней

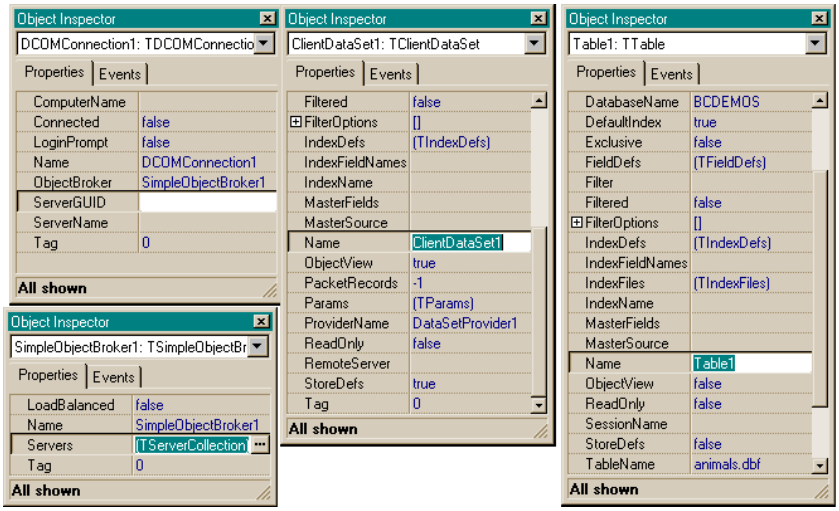


Рис. 15.3. Инспекторы объекта компонентов приложения

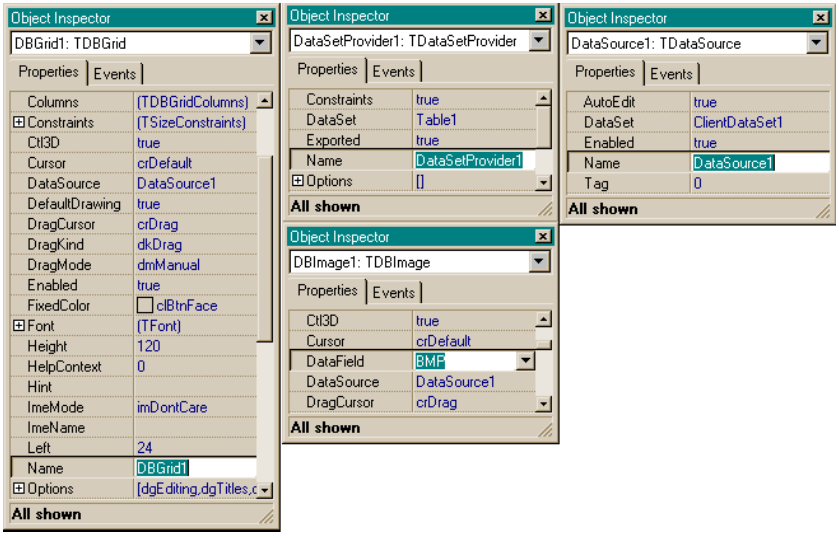


Рис. 15.4. Инспекторы объекта компонентов приложения (продолжение)

Форма с компонентами показана на рис. 15.2. Инспекторы объекта компонентов показаны на рис. 15.3 и 15.4, а форма в режиме исполнения — на рис. 15.5.

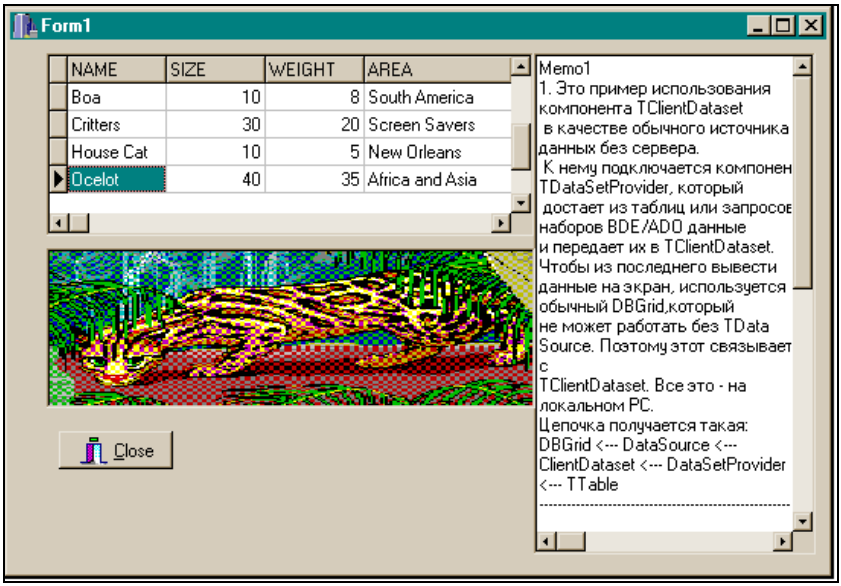


Рис. 15.5. Вид формы приложения с компонентами в ней в режиме исполнения

Использование компонента *TClientDataSet.* Пример 2

Попробуем использовать компонент для получения данных через сервер.

Как строится пара "клиент-сервер" на одной локальной машине? Сначала создается серверная часть, так как ее данные (имена провайдеров, каждый из которых извлекает данные только из связанной с ним таблицы или запроса) и имя сервера должны

указываться в компоненте TClientDataSet клиентской части. Сервер строится так: сначала создаем пустой проект из одной формы и сохраняем его. В форме расположим клиентскую часть. Серверную же часть построим с помощью компонента Remote Data Module, который вызывается командой **File/New/Multitier/ Remote Data Module**.

Одновременно с открытием этого компонента в хранилище объектов открываются два окна. В окне, расположенном слева, будет отражаться иерархия данных, задаваемых в правом окне. Правое окно играет роль контейнера, в который помещаются VCL-компоненты. Когда мы будем сохранять проект по команде **File/Save All**, система запросит имя вставленного удаленного модуля и потребует определить место для него. Расположим удаленный модуль вместе с проектом в некотором каталоге. Этот каталог и будет высвечиваться как имя удаленного сервера в клиентском компоненте TClientDataSet.

После того как мы сохранили пустой удаленный модуль как часть проекта, начинаем наполнять его содержанием, т. е. строить собственно серверную часть. Допустим, что нам надо, чтобы сервер извлекал данные из одной таблицы полностью (с использованием компонента TTable), а вторую свою функцию выполнял как выборку из некоторой таблицы (с использованием компонента TQuery). Помещаем компоненты TTable и TQuery в правое окно тем же способом, как и в форму. К каждому из них добавляем по одному компоненту TDataSetProvider. Эти последние связываем через их свойства DataSet с соответствующим компонентам (TTable, TQuery). Серверная часть готова. (В TQuery надо, конечно, задать запрос соответствующим образом).

Затем строим клиентскую часть. Кнопкой главной панели **View-Forms** открываем форму нашего проекта, в которой будем размещать компоненты клиентской части. Помещаем компоненты TClientDataSet, которые играют роль TTable в клиентской части, и в них определяем имена провайдеров на сервере, которые будут извлекать данные из соответствующего компонента на сервере и передавать их в компонент TClientDataSet. Имена всех провайдеров на сервере высветятся в раскрывающемся окне свойства **ProviderName**. Но это будет только в том случае, если сервер хоть раз был запущен на выполнение, потому что именно

при первом выполнении сервер регистрируется в Реестре Windows, а его имя и провайдеры становятся известны компоненту `TClientDataSet`. Кроме компонентов `TclientDataSet` в форму помещаем компонент `TDCOMConnection`, который собственно и организует связь с серверной частью. Связь устанавливается через свойства `ServerName` и `ServerGuid`. Свойство `ServerGuid` автоматически заполняется, когда заполнено `ServerName`, а значение `ServerName` выбирается из раскрывающегося списка, в котором увидим имя сервера (для локальной машины это путь к удаленному модулю и через точку — имя этого модуля. Если же мы собираемся строить сервер в сети, то надо пользоваться свойством `ComputerName`).

Если мы хотим отразить результаты работы с помощью компонента `DBGrid`, то так как он работает через источник данных `TDataSource`, то оба эти компонента помещаются в форму и `TDataSource` связывается с `TClientDataSet` через свое свойство `DataSet`.

Как обработать запрос с параметром? Надо сформировать на сервере в `Query`-объекте запрос с параметром обычным образом (например, `where CustNo=:p`), там же задать его характеристики через свойство `Params` из `Query`-объекта. Затем перейти к клиентской части. В свойстве `Params` компонента `ClientDataSet`, который через провайдера связан с данным `Query`-объектом (в нашем случае это `ClientDataSet2`), определить все необходимые параметры точно с такими же именами и характеристиками, которые были определены для `Query`-объекта на сервере. Затем в клиентской части построить обработчик, который бы вводил значения параметров и присваивал их параметрам по правилу (в примере всего один параметр типа `float`):

```
ClientDataSet2->Params->Items[0]-
>AsFloat=StrToFloat(Edit1->Text);
```

Далее провайдер за вас все сделает сам: подставит в `Query`-объект на сервере значение, которое получит из компонента `TClientDataSet`, и выполнит выборку.

На рис. 15.6 показана форма приложения с компонетами. Инспекторы объекта компонентов показаны на рис. 15.7 и 15.8, а удаленные модули — на рис. 15.9 и 15.10.

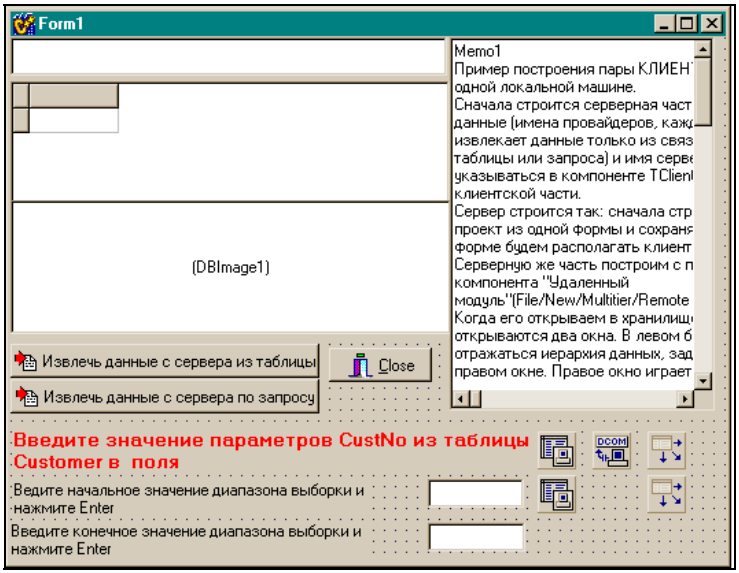


Рис. 15.6. Форма приложения с компонентами

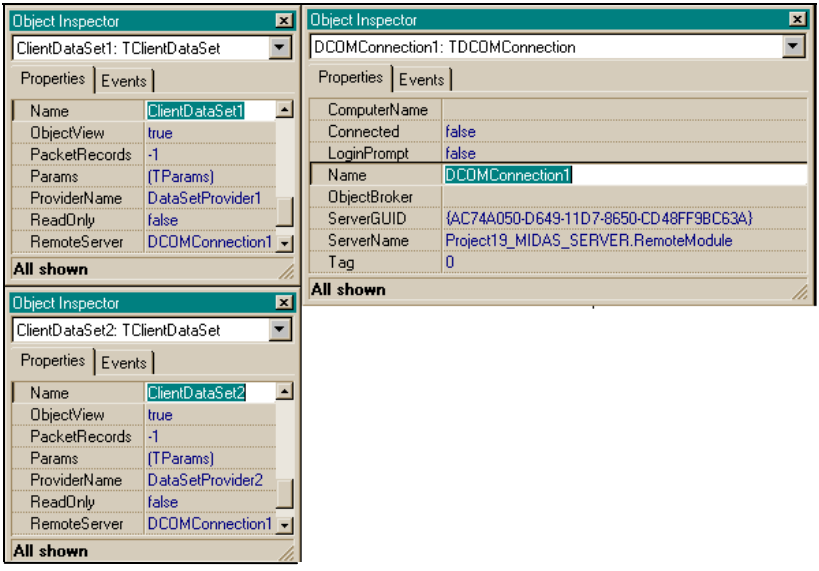


Рис. 15.7. Инспекторы объекта

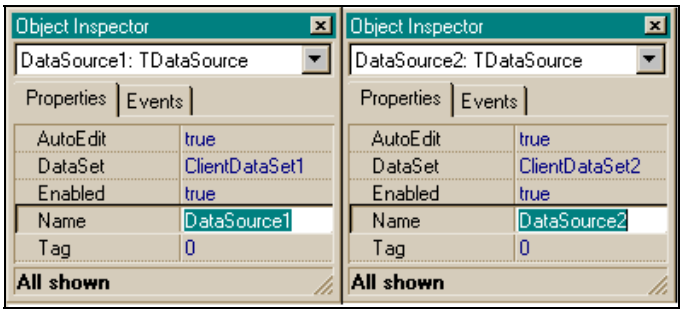


Рис. 15.8. Инспекторы объекта (продолжение)

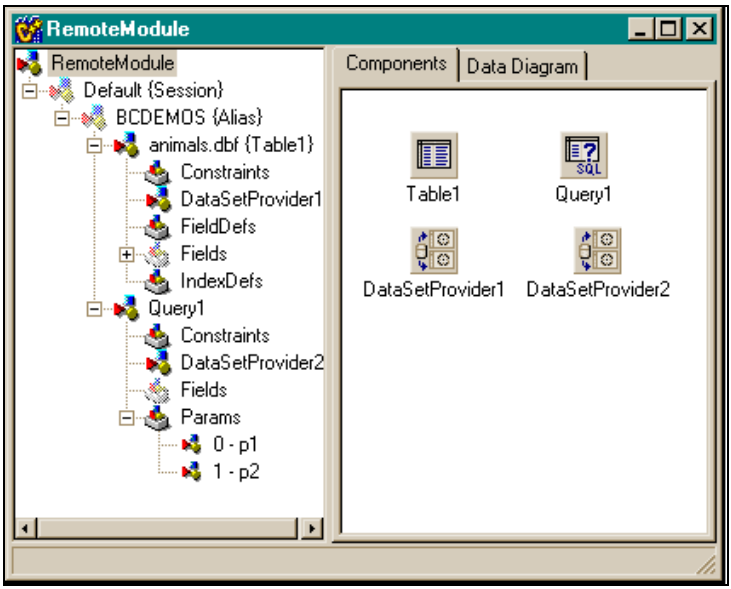


Рис. 15.9. Удаленный модуль (вкладка **Components**)

Форма приложения в режиме исполнения показана на рис. 15.11 и 15.12.

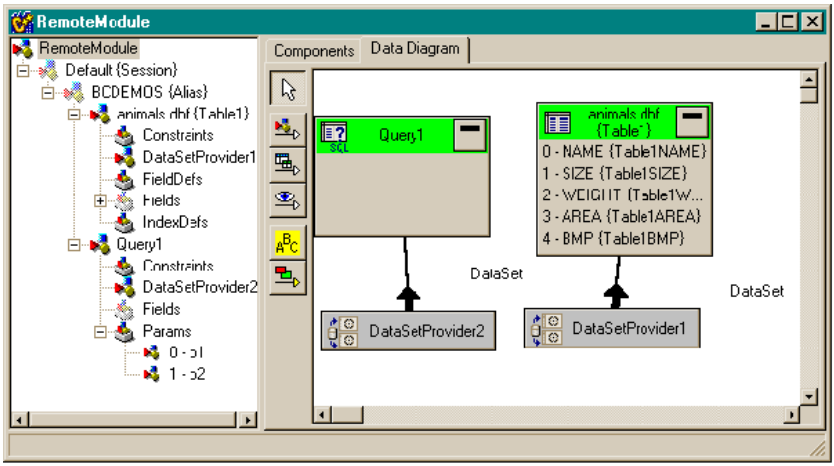


Рис. 15.10. Удаленный модуль
(вкладка **Data Diagram**)

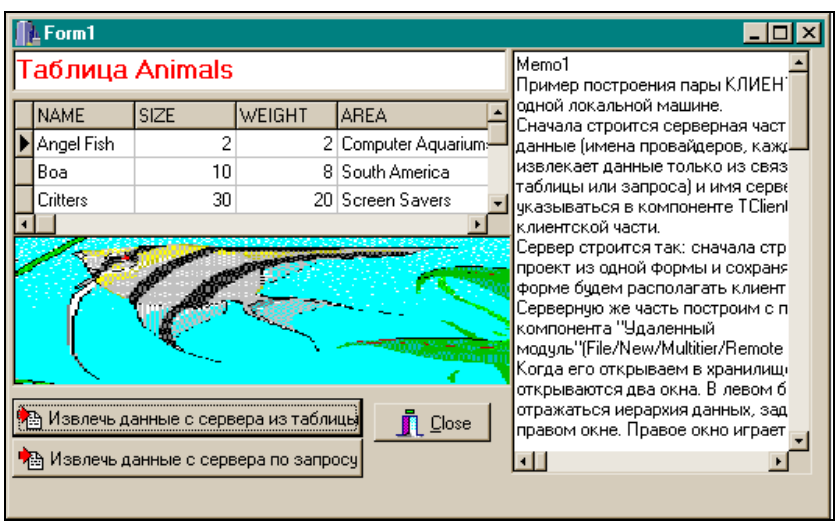


Рис. 15.11. Форма приложения в режиме исполнения
(извлечение данных по `TTable`)

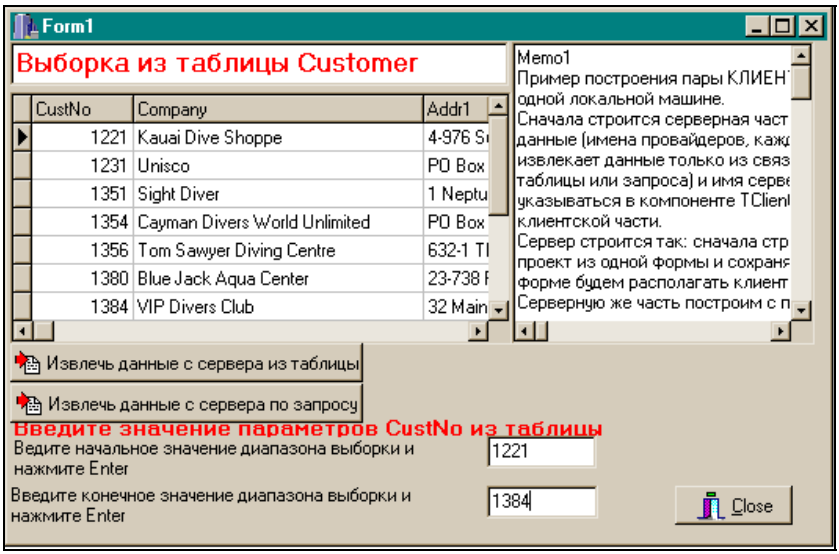


Рис. 15.12. Форма приложения в режиме исполнения (извлечение данных по `TQuery`)

Текст программы приложения приведен в листинге 15.1.

Листинг 15.1

```
//cpp-файл

//---------------------------------------------------------------
#include <VCL.h>
#pragma hdrstop

#include "Unit1_Server.h"
//---------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
int fix=0;
```

```
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
    : TForm(Owner)
{


}
//---------------------------------------------------------
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
 Label1->Visible=false;
 Label2->Visible=false;
 Label3->Visible=false;
 Edit2->Visible=false;
 Edit3->Visible=false;
 ClientDataSet2->Close();
 Edit1->Text="Таблица Animals";
 DBImage1->Visible=true;
 DBGrid1->DataSource=DataSource1;
 DCOMConnection1->Connected=false;  //Отключить предыдущее
                                    //соединение
 ClientDataSet1->Active=true; //Подключить текущее соединение
}
//---------------------------------------------------------
void __fastcall TForm1::BitBtn2Click(TObject *Sender)
{
 if(fix==0)
 {
  Edit1->Text="";
  Label1->Visible=true;
  Label2->Visible=true;
  Label3->Visible=true;
  Edit2->Visible=true;
  Edit3->Visible=true;
  Edit2->Text="";
```

```
  Edit2->SetFocus();
  Edit3->Text="";
  DBGrid1->DataSource=DataSource1; //чтобы очистить Grid при
                                   //повторе выборки
  DBGrid1->DataSource=DataSource2; //чтобы очистить Grid при
                                   //переходе от таблицы
                                   //к выборке
  ClientDataSet1->Close();
  ClientDataSet2->Close();  //Для разрыва соединения с
                            //сервером, чтобы при повторе новый
                           //параметр попадал на сервер
  fix=1;
  return;
 }
 if(Edit2->Text == "" || Edit3->Text == "")
 {
  Edit1->Text="Введите значение параметра!";
  Edit2->SetFocus();
 // fix=0;
  return;
 }
 ClientDataSet2->Params->Items[0]->AsFloat=StrToFloat(Edit2
 ->Text);
 ClientDataSet2->Params->Items[1]->AsFloat=StrToFloat(Edit3
 ->Text);
 ClientDataSet1->Close();
 DCOMConnection1->Connected=false;
 Edit1->Text="Выборка из таблицы Customer";
 DBGrid1->DataSource=DataSource2;
 ClientDataSet2->Active=true;
 fix=0;
}
//---------------------------------------------------------
void __fastcall TForm1::BitBtn3Click(TObject *Sender)
{
Form1->Close();
}
```

```
//---------------------------------------------------------
void __fastcall TForm1::ClientDataSet1AfterClose(TDataSet
*DataSet)
{
DBImage1->Visible=false;
}
//---------------------------------------------------------
void __fastcall TForm1::Edit2KeyDown(TObject *Sender, WORD
&Key,
   TShiftState Shift)
{
 if(Key == VK_RETURN)
  Edit3->SetFocus();
}
//---------------------------------------------------------
void __fastcall TForm1::Edit3KeyDown(TObject *Sender, WORD
&Key,
   TShiftState Shift)
{
 if(Key == VK_RETURN)
  BitBtn2Click((TBitBtn*) Sender); //запуск(нажатие) кнопки
                                   //BitBtn2
}
//---------------------------------------------------------
```

//h-файл

```
//---------------------------------------------------------
#ifndef Unit1_ServerH
#define Unit1_ServerH
//---------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Db.hpp>
```

```
#include <DBClient.hpp>
#include <DBGrids.hpp>
#include <Grids.hpp>
#include <MConnect.hpp>
#include <Provider.hpp>
#include <Buttons.hpp>
#include <DBCtrls.hpp>
//---------------------------------------------------------
class TForm1 : public TForm
{
__published:  // IDE-managed Components
    TDataSource *DataSource1;
    TDBGrid *DBGrid1;
    TClientDataSet *ClientDataSet1;
    TDCOMConnection *DCOMConnection1;
    TBitBtn *BitBtn1;
    TBitBtn *BitBtn2;
    TClientDataSet *ClientDataSet2;
    TEdit *Edit1;
    TBitBtn *BitBtn3;
    TDBImage *DBImage1;
    TMemo *Memo1;
    TDataSource *DataSource2;
    TLabel *Label1;
    TLabel *Label2;
    TEdit *Edit2;
    TLabel *Label3;
    TEdit *Edit3;
    void __fastcall BitBtn1Click(TObject *Sender);
    void __fastcall BitBtn2Click(TObject *Sender);
    void __fastcall BitBtn3Click(TObject *Sender);
    void __fastcall ClientDataSet1AfterClose(TDataSet
*DataSet);
```

```
    void __fastcall Edit2KeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift);
    void __fastcall Edit3KeyDown(TObject *Sender, WORD &Key,
     TShiftState Shift);
private:  // User declarations
public:  // User declarations
    __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------
#endif
```

Предметный указатель

U

A

Б

В

Г