Борис Пахомов

C/C++ и MS Visual C++ 2012

Основные элементы языков C/C++
Визуальная среда программирования
Создание основных типов приложений
Работа с наборами данных

ДЛЯ НАЧИНАЮЩИХ

Борис Пахомов

C/C++ и MS Visual C++ 2012 для начинающих

Санкт-Петербург
«БХВ-Петербург»
2013

УДК 004.4
ББК 32.973.26-018.2
П12

Пахомов Б. И.

П12 C/C++ и MS Visual C++ 2012 для начинающих. — СПб.: БХВ-Петербург, 2013. — 512 с.: ил.

ISBN 978-5-9775-0881-0

Книга является руководством для начинающих по разработке приложений в среде Microsoft Visual C++ 2012. Рассмотрены основные элементы языков программирования C/C++ и примеры создания простейших классов и программ. Изложены принципы визуального проектирования и событийного программирования. На конкретных примерах показаны основные возможности визуальной среды разработки Microsoft Visual C++, назначение базовых компонентов и процесс разработки различных типов консольных и Windows-приложений.

Для начинающих программистов

УДК 004.4
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 30.11.12.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 41,28.
Тираж 1800 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-0881-0

© Пахомов Б. И., 2013
© Оформление, издательство "БХВ-Петербург", 2013

Оглавление

Введение **1**

ЧАСТЬ I. ИЗУЧЕНИЕ ЯЗЫКА C/C++ **3**

Глава 1. Общие сведения о среде Visual C++ 2011. Создание консольного приложения **5**

Общие положения 5
Структура рабочего стола среды программирования 7
Главное окно 7
Некоторые замечания 9
О рабочем столе 9
О справочной системе Help 13
Структура программ в VC++ 15
Переход к созданию консольного приложения 17
Типы данных, простые переменные и основные операторы цикла. Создание простейшего консольного приложения 23
Программа с оператором *while* 29
Имена и типы переменных 30
Оператор *while* 32
Оператор *for* 34
Символические константы 35

Глава 2. Программы для работы с символьными данными **37**

Программа копирования символьного файла. Вариант 1 39
Программа копирования символьного файла. Вариант 2 42
Подсчет символов в файле. Вариант 1 42
Подсчет символов в файле. Вариант 2 44
Подсчет количества строк в файле 47
Подсчет количества слов в файле 48

Глава 3. Работа с массивами данных **51**

Одномерные массивы 51
Многомерные массивы 54

Глава 4. Создание и использование функций ... 57
Создание некоторых функций ... 59
Ввод строки с клавиатуры ... 59
Функция выделения подстроки из строки ... 62
Функция копирования строки в строку ... 63
Головная программа для проверки функций *getline(), substr(), copy()* ... 64
Внешние и внутренние переменные ... 66
Область действия переменных ... 69
Как создать свой внешний файл ... 69
Атрибут *static* ... 70
Рекурсивные функции ... 72
Некоторые итоговые данные по изучению функций ... 72
Перегрузка функций ... 75
Использование шаблонов функций ... 76
Создание простого шаблона функции ... 76
Шаблоны, которые используют несколько типов ... 77

Глава 5. Функции для работы с символьными строками ... 79
Основные стандартные строковые функции ... 79
Функция *sprintf()* ... 79
Функция *strcpy()* ... 79
Функция *strcmp()* ... 80
Функция *strcmpi()* ... 80
Функция *strcat()* ... 80
Функция *strlen()* ... 80
Пример программы проверки функций ... 81

Глава 6. Дополнительные сведения о типах данных, операциях, выражениях и элементах управления ... 85
Новые типы переменных ... 85
Константы ... 88
Новые операции ... 89
Преобразование типов данных ... 91
Побитовые логические операции ... 92
Операции и выражения присваивания ... 93
Условное выражение ... 95
Операторы и блоки ... 95
Конструкция *if-else* ... 95
Конструкция *else-if* ... 96
Переключатель *switch* ... 100
Уточнение по работе оператора *for* ... 103
Оператор *continue* ... 103
Оператор *goto* и метки ... 104

Глава 7. Работа с указателями и структурами данных ... 105
Указатель ... 105
Указатели и массивы ... 109

Операции над указателями ... 111
Указатели и аргументы функций ... 111
Указатели символов и функций ... 113
Передача в качестве аргумента функции массивов размерности больше единицы ... 117
Массивы указателей ... 117
Указатели на функции ... 118
Структуры. Объявление структур ... 120
Обращение к элементам структур ... 122
Структуры и функции ... 125
Программы со структурами ... 125
Функция возвращает структуру ... 125
Функция возвращает указатель на структуру ... 128
Программа упрощенного расчета заработной платы одному работнику ... 131
Рекурсия в структурах ... 133
Битовые поля в структурах ... 138
Категории памяти ... 139

Глава 8. Классы в C++. Объектно-ориентированное программирование ... 141
Классы ... 143
Принципы построения классов ... 144
Инкапсуляция ... 144
Наследование ... 145
Полиморфизм ... 146
Примеры создания классов ... 147
Пример 1 ... 147
Пример 2 ... 150
Пример 3 ... 151
Конструктор класса ... 153
Деструктор класса ... 156
Классы и структуры в среде CLR ... 156
Классы и структуры ... 156
Абстрактные классы ... 158
Статические функции и элементы данных ... 158
Использование элементов с атрибутами *public static*, если объекты не существуют ... 161
Частные и общие данные. Интерфейсные функции ... 163
Использование оператора глобального разрешения для элементов класса ... 163

Глава 9. Ввод и вывод в языках C и C++ ... 165
Ввод и вывод в C ... 165
Ввод/вывод файлов ... 165
Основные функции для работы с файлами ... 166
Стандартный ввод/вывод ... 172
Функции стандартного ввода/вывода ... 172
Ввод/вывод в C++ ... 178
Общие положения ... 178
Ввод/вывод с использованием разных классов ... 179
Пространства имен ... 180
Работа с классом *fstream* ... 181

Работа с классом *ofstream* ... 184
Работа с классом *ifstream* ... 185
Работа с бинарным файлом ... 187
Стандартный ввод/вывод в C++ ... 189
Общие положения ... 189
Стандартный вывод *cout* ... 189
Стандартный ввод *cin* ... 193

ЧАСТЬ II. ПРИЛОЖЕНИЯ WINDOWS FORM ... 195

Глава 10. Продолжение изучения среды Visual C++ ... 197
Создание проекта ... 197
Некоторые файлы проекта ... 202
Окно сведений об объекте ... 204
Вкладка *Events* ... 205
Вкладка *Property Pages* ... 207
Работа с окном сведений об объекте ... 207
Редактор кода, h-модуль и режим дизайна (проектирования). Указатель *this* ... 208
Контекстное меню редактора кода ... 210
Суфлер кода (подсказчик) ... 212
Настройка редактора кода ... 212
Управление окнами редактора ... 212
Настройка опций редактора через команду *Tools* главного меню ... 213
Изменение шрифта и цвета ... 215
Начало редактирования кода программного модуля ... 215
Компоненты среды программирования VC++ ... 216
Класс *Form* ... 216
Дизайнер форм ... 216
Помещение компонента в форму ... 218
Другие действия с дизайнером форм ... 218
Контекстное меню формы ... 219
Добавление новых форм к проекту ... 220
Организация работы с множеством форм ... 221
Вызов формы на выполнение ... 221
Свойства формы ... 221
События формы ... 234
Некоторые методы формы ... 235
Рисование графиков в форме ... 237

Глава 11. Компоненты, создающие интерфейс между пользователем и приложением ... 245
Пространство имен *System* ... 246
Работа с переменными некоторых типов ... 247
Компонент *Button* ... 250
Свойства *Button* ... 250
События *Button* ... 254
Методы *Button* ... 255

Компонент *Panel* ... 255
Некоторые свойства *Panel* ... 256
Некоторые события *Panel* ... 256
Компонент *Label* ... 258
Некоторые свойства *Label* ... 258
События *Label* ... 259
Компонент *TextBox* ... 259
Некоторые свойства *TextBox* ... 260
События *TextBox* ... 263
Некоторые методы *TextBox* ... 265
Компонент *MenuStrip* ... 266
Некоторые свойства *MenuStrip* ... 272
События *MenuStrip* ... 273
Компонент *ContextMenuStrip* ... 273
Компонент *ListView* ... 274
Некоторые свойства *ListView* ... 278
События *ListView* ... 280
Компонент *WebBrowser* ... 282
Компонент *ListBox* ... 288
Как работать с *ListBox* ... 288
Свойства *ListBox* ... 289
Как использовать *ListBox* ... 292
Как формировать список строк ... 292
Компонент *ComboBox* ... 298
Свойства *ComboBox* ... 299
События *ComboBox* ... 301
Некоторые методы *ComboBox* ... 301
Примеры использования *ComboBox* ... 303
Пример 1 ... 303
Пример 2 ... 308
Пример 3 ... 312
Компонент *MaskedTextBox* ... 317
Свойства *MaskedTextBox* ... 319
Компонент *CheckedListBox* ... 321
Пример: домашний телефонный справочник ... 324
Компоненты *CheckBox* и *RadioButton* ... 338
Компонент *GroupBox* ... 342
Компонент *LinkLabel* ... 343
Компонент *PictureBox* ... 354
Некоторые свойства компонента *PictureBox* ... 354
Компонент *DateTimePicker* ... 357
Форматные строки даты и времени ... 359
Стандартное и пользовательское форматирование ... 360
Некоторые сведения о работе с датами ... 365
Компонент *TabControl* ... 373
Как задавать страницы ... 374
Некоторые методы *TabControl* ... 376

Некоторые свойства страницы *TabPage* ... 377
Как защитить страницу от неавторизованного доступа ... 378
Задача регистрации пользователя в приложении ... 380
Компонент *Timer* ... 390
Компонент *ProgressBar* ... 394
Компонент *OpenFileDialog* ... 395
Компонент *SaveFileDialog* ... 401
Компонент *ColorDialog* ... 407
Компонент *FontDialog* ... 407
Компонент *PrintDialog* ... 408
Компонент *ToolStrip* ... 409
Некоторые свойства *ToolStrip* ... 410
Использование *ToolStrip* ... 411

Глава 12. Работа с наборами данных. Общие сведения о базах данных ... 413
Проектирование баз данных ... 414
Модель базы данных ... 415
Структура проектирования базы данных ... 415
Идентификация сущностей и атрибутов ... 416
Проектирование таблиц ... 417
Определение неповторяющихся атрибутов ... 418
Набор правил при разработке таблицы ... 419
Определение ограничений на целостность данных ... 419
Принудительное обеспечение целостности данных ... 420
Выбор индексов ... 420
Язык SQL ... 420
Примеры оператора *SELECT* ... 422
Наборы данных (компонент *DataSet*) ... 423
Общая технология организации работы с базой данных в приложении ... 424
Пример работы с базой данных ... 425

Глава 13. Управление исключительными ситуациями ... 459
Операторы *try*, *catch* и *throw* ... 459
Пример 1 ... 461
Пример 2 ... 462
Классы типов исключений ... 464
Пример 3 ... 466
Функции, выдающие исключения ... 468

Глава 14. Преобразование между нерегулируемыми и регулируемыми (режим CLR) указателями ... 471
Пример 1. Перевод строки *String* ^ в ASCII-строку ... 472
Пример 2. Перевод ASCII-строки в строку *String* ^ ... 474
Пример 3. Преобразование строки *String* ^ в строку *wchar_t* ... 475
Пример 4. Преобразование строки *wchar_t* в строку *String* ^ ... 477
Пример 5. Маршалинг native-структуры ... 478
Пример 6. Работа с массивом элементов native-структуры в managed-функции ... 480

Пример 7. Доступ к символам в классе *System::String* 482
Пример 8. Преобразование *char* * в массив *System::Byte* 483
Пример 9. Преобразование *System::String* в *wchar_t* * или *char* *..... 484
Пример 10. Преобразование *String* в *string*..... 485
Пример 11. Преобразование *string*-строки в *String*-строку 489
Пример 12. Объявление дескрипторов в native-типах 490
Пример 13. Работа с дескриптором в native-функции 491

Предметный указатель 493

Введение

Книга предназначена для начинающих программистов, поэтому многие вопросы раскрыты не во всей глубине, ибо читатель должен научиться пользоваться продуктом, а не тонуть в подробностях. Того, что дано здесь, достаточно для написания консольных приложений, а также многих приложений типа Windows Form. Автор надеется, что полученные знания явятся ступенью для изучения интересного и полезного (несмотря на его многие недостатки) программного продукта, каким, несомненно, является MS Visual C++ 2012. Для начинающего вполне достаточно изучить язык и Windows-формы, чтобы в дальнейшем, имея определенный багаж знаний, заниматься даже проблемами Интернета. Кстати, и в этом урезанном варианте некоторые выходы в Интернет имеются (например, через компонент `LinkLabel`).

Примечание

В этой книге рассматривается программирование в Microsoft Visual C++ 2012, которая на этапе разработки и тестирования носила кодовое название Visual C++ 11.0 или Visual C++ 2011. Поскольку окончательная версия продукта появилась только в середине 2012 года, то она и получила название Microsoft Visual C++ 2012. При написании книги была использована пробная версия Microsoft Visual C++ 2011 Ultimate (в дальнейшем VC++).

Существенным недостатком этой версии, как и версий, начиная с 2008 г., является отсутствие возможности работы с базами данных в том варианте, как это было в версии 2005 г. Представитель разработчика еще в марте 2010 г. заверил, что указанный недостаток будет устранен. Но, к сожалению, и в этой новой версии 2011 г. все в плане работы с базами данных осталось по-старому. Более того, в данной версии отключена возможность работы с подсказчиком (надеюсь, что только потому, что она пробная), что вообще не поддается никакому объяснению.

Однако все же есть небольшая лазейка в этом безнадежном деле: в *главе 12* приводится метод работы с базой данных типа MS Access на основе построения таблиц данных в самом приложении. Но и здесь стоит отметить очень плохого качества такие компоненты, как `dataSet`, `bindingSource` и особенно `dataGridView`: на одном наборе данных они могут работать, а на другом с такими же настройками компонентов, как и на предыдущем наборе, не работают.

Хочу отметить, что данная книга уточняет многие моменты, опущенные в прежнем издании, и дополнена новыми данными, например, по классам. Поэтому на ее фундаменте начинающему программисту будет значительно легче войти в среду VC++, нежели с помощью предыдущей книги автора.

C/C++

Microsoft® Visual

ЧАСТЬ I

Изучение языка C/C++

Глава 1. Общие сведения о среде Visual C++ 2011. Создание консольного приложения

Глава 2. Программы для работы с символьными данными

Глава 3. Работа с массивами данных

Глава 4. Создание и использование функций

Глава 5. Функции для работы с символьными строками

Глава 6. Дополнительные сведения о типах данных, операциях, выражениях и элементах управления

Глава 7. Работа с указателями и структурами данных

Глава 8. Классы в C++. Объектно-ориентированное программирование

Глава 9. Ввод и вывод в языках C и C++

C/C++ Microsoft Visual

ГЛАВА 1

Общие сведения о среде Visual C++ 2011. Создание консольного приложения

Общие положения

Вам требуется решить некоторую задачу. С помощью компьютера, конечно. Например, рассчитать движение материальных ценностей по некоторому складу: сколько чего было на данную дату, сколько чего поступило, сколько ушло, сколько осталось. С чего обычно начинают? Ясно сразу, что если задачу надо решать на компьютере, то следовало бы ее решение как-то формализовать, т. е. алгоритм ее решения (набор последовательных действий, исполнение которых приводит к решению задачи) требуется привести к последовательности неких формальных действий, понятных потом машине (говорят, что надо построить *машинный алгоритм решения задачи*). Затем надо продумать форму общения *(интерфейс)* того, кто станет решать эту задачу на компьютере *(пользователя)* с самим компьютером, исходя из максимального удобства общения. Это чуть ли не одна из главных трудностей проектирования решения задачи, ибо неудобство общения раздражает пользователя, который начинает совершать ошибки, что может привести, в конечном счете, к тому, что ваш проект просто будет отвергнут. Имеются еще некоторые шаги по подготовке к решению, но мы их здесь опустим, т. к. это не наша проблема.

Итак, мы изучили задачу, создали машинный алгоритм ее решения на компьютере, разработали интерфейс взаимодействия будущего пользователя с компьютером по решению данной задачи. А что дальше? А дальше все эти разработки надо перевести на понятный компьютеру язык, т. е., как говорят, *запрограммировать* наши действия, составить машинную программу, которая представляет собой последовательность определенных команд, записанных на выбранном для решения задачи языке (*алгоритмическом языке*, как принято называть). Для решения конкретной задачи с учетом разработанного интерфейса подходит не всякий алгоритмический язык. Поэтому разработчику и надо выбирать подходящий к данной ситуации язык, который бы отвечал требованиям к решению задачи. И не просто обеспечивал бы ее программирование, но и отвечал бы еще множеству других условий: позволял бы программисту быстро и надежно создавать программу, обеспечивал бы удобное сопровождение программы в период ее эксплуатации и т. д.

Когда программа написана на некотором алгоритмическом языке, она должна быть переведена в *машинный язык*, в язык, на котором работает компьютер, в его систему команд. Для этого существуют специальные программы, называемые *компиляторами*. Эти программы имеют параметры, задание которых позволяет компилятору создавать машинные программы в той или иной плоскости. Например, существуют параметры, позволяющие компилятору минимизировать размер памяти, которую станет занимать скомпилированная программа. Или, как мы будем рассматривать в данной книге, существует параметр, позволяющий компилятору создавать программы (часто говорят просто "коды"), состоящие из так называемых управляемых или неуправляемых кодов. Параметры компилятора называют по-разному: ключами, опциями.

Но программу мало откомпилировать. Компилирование — это только первый этап создания машинной программы. Дело в том, что в общем случае, для решения конкретной задачи, т. е. реализации ее машинного алгоритма, требуется подключение неких стандартных *библиотек*, содержащих стандартные программы, которые разрабатываются один раз и используются во многих алгоритмах. Например, перевод десятичных чисел в двоичные или шестнадцатеричные. Каждый программист, пишущий программу, не станет всякий раз заниматься этим переводом. Поэтому в подобных случаях разработчик программной среды, в рамках которой создается программный продукт (в нашем случае — это Visual Studio 2011), сам создает подобные библиотеки и поставляет их со средой разработки, которая содержит и компиляторы с разных языков среды в машинные коды. В свою очередь, и опытный программист может самостоятельно создавать такие библиотеки и включать их в общий перечень библиотек среды программирования, чтобы они в дальнейшем подключались автоматически к решению задач. Для этой цели среда поставляет специальные средства.

Но вернемся к компиляции. Компилятор, просматривая программу (код, как говорят), переводит ее (код) в машинные команды. Но не просто в набор команд, а формирует это все множество в виде отдельных (*объектных*) *модулей*. В каждом таком модуле создается своя таблица имен со ссылками на их месторасположение. То есть компилятор создает не исполняемый код, а так называемый *объектный код*, содержащий неконкретные (как говорят "неразрешенные", т. е. неопределенные) ссылки. На этом его работа завершается. Чтобы получить *исполняемый модуль*, надо "разрешить", т. е. конкретизировать ссылки, сформированные компилятором с учетом конкретного размещения данного кода в выделенной для него памяти компьютера. Эту работу выполняет специальная программа, которая называется по-разному: *редактор связей*, *компоновщик*, *линковщик*, *построитель*. После работы этой программы получается набор машинных команд, готовых к исполнению на компьютере.

В свете всего вышесказанного можно утверждать, что для того чтобы создать программу, требуется еще иметь средства ее компиляции и компоновки. Эти средства поставляются в рамках изучаемой нами среды MS Visual C++ 2011. То есть, прежде чем перейти к изучению собственно языка C/C++, надо познакомиться хотя бы в общих чертах с интерфейсом среды обработки данных MS Visual C++ 2011, что-

бы иметь возможность с его помощью компилировать и строить исполняемые программы в рамках языка C/C++.

Структура рабочего стола среды программирования

Цель этой главы — продемонстрировать начальные элементы программирования на языке C/C++. Язык C++ является дальнейшим развитием языка C, поэтому все конструкции языка C поддерживаются в C++. Однако в C++ появились новые возможности синтаксиса, не имеющиеся в C. Это мы увидим по мере рассмотрения материала.

Чтобы построить программу на этом языке, нам надо воспользоваться средой программирования Visual C++ 2011 (ее английская аббревиатура — IDE: Integrated Development Environment), которая содержит средства создания программы, ее компиляции, отладки и запуска на выполнение. В этой связи рассмотрим кратко структуру этой среды, а точнее, ее интерфейс с нами, пользователями. Интерфейс — это аппарат, который позволяет удобно взаимодействовать пользователю со средой.

После установки на своем компьютере среды Visual C++ 2011 она запускается на выполнение. Здесь следует отметить один момент.

Среда Visual Studio 2011 Professional состоит из многих подсред, подразделов, каждый из которых нацелен функционально обрабатывать определенную область задач. Чтобы настроиться на работу с конкретным подразделом, разработчики предусмотрели для пользователя возможность выбора одной конкретной подсреды, с которой он станет работать после запуска основной среды. Когда пользователь выберет такую подсреду (в данном случае автор выбрал подсреду Visual C++), основная среда настраивается на выбранный подраздел и пользователь получает к нему доступ. Адрес загрузки такой подсреды, как и при установке любой программы, попадает в меню кнопки **Пуск**, откуда вы можете ее загрузить, воспользовавшись командой главного меню **Пуск | Программы**. Для удобства дальнейшей работы с установленным программным продуктом следует мышью перетянуть его значок на линейку быстрого запуска программ, которая находится на рабочем столе операционной системы (обычно ее располагают в нижней части стола). Находящийся на этой линейке любой программный продукт запускается одинарным щелчком мыши на значке соответствующего продукта. Итак, загружаем наш продукт Microsoft Visual C++ 2011 (для краткости в дальнейшем станем его называть VC++). На экране появится главное окно — рабочий стол, структуру которого мы и рассмотрим.

Главное окно

Общий вид окна показан на рис. 1.1. Этот формат интерфейса принят в среде по умолчанию и может быть всегда восстановлен через соответствующую опцию главного меню: **Windows | Reset Window Layout**.

Рис. 1.1. Вид главного окна IDE после загрузки VC++

В верхней части окна расположена строка с *командами главного меню* среды (команды: **File**, **Edit**, ...) — это строка горизонтального меню. При вызове этих команд (их еще называют опциями, т. е. элементами выбора из нескольких значений) открываются так называемые *"выпадающие меню"* — это вертикальные меню, представляющие собой набор команд, располагающихся на экране сверху вниз. Пример такого меню показан на рис. 1.2.

Ниже строки главного окна находятся *кнопки быстрого вызова* некоторых команд на исполнение. Все эти кнопки имеют всплывающие подсказки (надо навести курсор мыши на кнопку, немного подождать, после чего появится подсказка о том, для чего предназначена данная кнопка). Рядом с такими кнопками могут быть дополнительные кнопки для раскрытия списка значений основной кнопки. Так как все кнопки не помещаются в отведенное им место на рабочем столе, то они свернуты в небольшие полосы с кнопками их развертывания точно так же, как это выполнено во всем известной программе Word (рис. 1.3).

Вид главного окна, в свою очередь, изменяется при задании типа создаваемого приложения. С этим мы познакомимся, когда начнем создавать приложения.

Рис. 1.2. Пример выпадающего меню

Рис. 1.3. Кнопки быстрого вызова

Некоторые замечания

О рабочем столе

Рабочий стол формируется из набора окон. Каждое окно — это обычное Windows-окно, имеющее стандартную заголовочную полосу в своей верхней части. За эту полосу можно окно перемещать с помощью протягивания мышью. У окон имеются свойства, которые открываются, если на заголовочной части щелкнуть правой кнопкой мыши. Перечень свойств окна **Solution Explorer** показан на рис. 1.4.

Рис. 1.4. Меню свойств окна

Кстати сказать, стартовая страница, исключая главное меню и набор кнопок быстрого запуска, — это тоже обычное окно со своим набором свойств.

Особенностью набора окон является возможность разбирать этот набор на подокна и группировать последние между собой в соответствии с желаниями пользователя. Для этого у каждого окна имеются определенные свойства. Чтобы лучше понять,

как работают некоторые свойства окон, введем такое понятие, как *причаливание*. По самому смыслу этого слова понятно, что объект, обладающий свойством причаливания, может по аналогии с морским или речным портом причаливать либо к берегу, либо к другому объекту. В нашем случае таким берегом является полоса стартовой страницы, содержащая главное меню и кнопки быстрого запуска. К ней могут причаливать другие окна, образуя у этого "берега" свои вкладки, как это делается в интернет-браузерах, когда открывается та или иная Web-страница. Но причаливание окна может осуществляться и к другому окну или к группе окон, образующих из своих вкладок новый берег, когда одно или несколько окон занимают всю горизонтальную полосу страницы, а остальные окна располагаются в нижней полосе, образуя своими вкладками свой "берег причаливания". При этом окно, к которому причаливает другое окно, как бы захватывает, проглатывает причаливающее к нему окно, оставляя от последнего только вкладку с именем "проглоченного" окна, по которой можно открыть такое причалившее и "проглоченное" окно.

Как визуально определить, захватится ли перемещаемое вами окно "берегом" или другим окном-объектом? Когда вы перемещаете по экрану окно, вы видите некие направляющие "кресты", возникающие на вашем пути и синие подсветки, всплывающие в некоторых областях перемещения. Кресты возникают по центру активной вкладки, через территорию которой вы перемещаетесь. *Вкладка* — это какое-то окно, причаленное к берегу или объекту. Синяя подсветка же появляется в момент, когда подсвеченная область готова захватить ваше перемещаемое окно, как только вы отпустите левую кнопку мыши. Подсветки возникают справа, слева, сверху, снизу у креста, показывая область размещения будущего захваченного объекта. Уточним, чтобы "вытащить" окно из какой-то группы, надо зацепиться мышью за его заголовок и начать протяжку окна. Если получилось так, что окна расположились горизонтально в несколько слоев (например, первый слой содержит вкладки двух окон, второй — двух окон, третий — одного окна), а вы хотите, чтобы все вкладки были в одном слое, вытащите, ухватив мышью заголовок вкладки, окно в область, где оно не попадает на синее пространство, уменьшите мышью горизонтальный размер окна и снова вставьте окно (через его попадание в поле, в котором появится синяя подсветка) в место его причаливания. Так поступите с каждым из окон во всех слоях, добиваясь, чтобы они по ширине заголовка помещались в причаливамый "берег". Если вы случайно закрыли какое-то окно (оно при этом, естественно, исчезло с экрана), следует найти его с помощью опций главного меню **Windows** и **View**, на которых надо просто щелкнуть мышью.

Вот опции окна (открываются правой кнопкой мыши):

- **Float** (плавающее). Такое окно можно перетягивать в любую часть рабочего стола. Оно имеет вид обычного Windows-окна. Подобное окно как бы отвязано от набора окон и может перемещаться мышью в любое место экрана. При этом, как только протяжка окна завершается (напомним, что протяжка — это захват окна за его заголовочную полосу мышью и удержание левой кнопки мыши в момент продвижения окна по экрану; завершение протяжки — отпускание нажатой левой кнопки мыши), окно остается в том же месте, где оно находилось в момент завершения протяжки. Но так получается только тогда, когда окно в мо-

мент остановки не попало на поле синей подсветки, иначе окно захватится и причалит к области, определенной синей подсветкой;

- **Dock** (причаливающее). Вспомните морское понятие "док" — место для причаливания судов. Это аналогия с нашим случаем. Окно, у которого выбрана подобная опция, моментально причалит к какому-то "берегу" (зависит от его местоположения в момент выбора опции **Dock**);
- **Dock as Tabbed Document** (причаливать в качестве вкладки к основному причалу рабочего стола — к группе опций главного меню). Как только мы выбираем эту опцию у какого-то окна, оно моментально причаливает к основному причалу и становится вкладкой, располагающейся первой слева. Таким способом можно переупорядочить последовательность вкладок, назначая каждой в заданном порядке рассматриваемое свойство окна.

Пример показан на рис. 1.5. На рис. 1.5, *а* приведен вид рабочего стола VC++, в котором четыре окна представлены в виде своих вкладок, расположенных под полем главного меню (в данном случае оно будет играть роль причала). С помощью опции **View** главного меню (**View | Other Windows | Server Explorer**) выбрано окно **Server Explorer**, которое появилось на экране и свойства которого показаны на рис. 1.5, *б*. Затем выбирается опция **Dock as Tabbed Document**. Окно моментально становится вкладкой того причала, который показан ранее на рис. 1.5, *а*. В результате имеем вид уже пяти окон, приведенный на рис. 1.5, *в*;

Рис. 1.5. Изменение рабочего стола VC++: *а* — первоначальный вид VC++

б

в

Рис. 1.5. Изменение рабочего стола VC++:
б — окно **Server Explorer** и его свойства-опции;
в — окно **Server Explorer** после выбора его свойства **Dock as Tabbed Document**

- **Auto Hide** (автоматически исчезать). В этом случае окно автоматически "прячется" (причаливает) в качестве вкладки к ближайшей боковой стороне основного окна рабочего стола;
- **Hide** (спрятать). При выборе этого свойства окно исчезает с экрана. Чтобы оно снова появилось, надо воспользоваться либо опцией **View** главного меню, либо соответствующей данному окну опцией **Other Windows** опции **View**.

Все рассмотренные выше свойства надо хорошо понимать, чтобы манипулировать положением окон на рабочем столе, иначе может сложиться ситуация, когда на рабочем столе соберется множество окон, которые просто станут мешать работать. Их надо будет "разогнать" по боковым сторонам основного окна, а другие просто спрятать. Например, как установить справа сбоку основного окна рабочего стола окна **Toolbox** и **Server Explorer**? Предварительно надо щелкнуть мышью на вкладке окна, и оно появится на рабочем столе. Тогда у него можно увидеть его опции. У рассматриваемых окон надо выбрать (через правую кнопку мыши) свойство **Dock** (причаливание). Тогда в заглавной строке окна появится значок **Auto Hide** (скрывать автоматически). Значок имеет вид вертикального полупроводника. Если на этом значке щелкнуть, то окно причалит к правой стороне главного окна среды и спрячется: останется видно только его имя.

О справочной системе Help

Все всегда начинается с вопроса: "А где посмотреть, как это делается?" Ясно, что в справочной системе к программному продукту. Откроем опцию **Help** главного меню VC++ (рис. 1.6).

Рис. 1.6. Меню **Help**

В рассматриваемой нами среде программирования VC++ предусмотрено использование справочной информации из двух источников: из Интернета и из своего компьютера. Для этого справочную систему следует предварительно настроить на желаемый источник информации.

Почему из двух источников? Если иметь справку на своем компьютере, ее можно очень быстро загружать. Но за это надо платить: приходится самому поддерживать

ее в актуальном состоянии. Для этого в меню **Help** служит опция **Add and Remove Local Help Content**. Если же пользоваться базой данных разработчика среды, то база данных, естественно, поддерживается самим разработчиком, что удобно. Но и за это удобство есть своя плата: приходится загружать справку через Интернет, что, во-первых, дольше и дороже, а во-вторых, Интернет может быть в нужный момент и недоступен.

Настройка источника информации осуществляется через опцию **Set Help Preference** меню **Help** (рис. 1.7).

Вот теперь, когда вы начнете выполнять опцию **View Help**, показанную на рис. 1.6, ваш браузер откроет вам доступ к справочной системе — рис. 1.8.

Рис. 1.7. Настройка среды программирования на источник справочной информации

Рис. 1.8. Окно MSDN

Обратим внимание на опцию **Samples** (образцы) в подменю опции Help, которая открывает доступ к примерам приложений различного направления, заготовленным разработчиком.

Структура программ в VC++

Программы в VC++ называются приложениями (очевидно, приложениями к среде IDE). Мы так и дальше станем их называть. Приложения строятся средой в виде специальных конструкций — проектов, которые выглядят для пользователя как совокупность нескольких файлов.

Программа на языке C, расширением которого является язык C++ (далее не станем их пока разделять и будем писать C/C++), — это совокупность функций, т. е. специальных программных образований, отвечающих определенным требованиям. Причем приложение — это главная функция, внутри которой помещаются операторы, реализующие алгоритм приложения. Среди операторов имеются такие, которые служат для вызова других функций, требующихся при реализации алгоритма. Запуск любой программы начинается с запуска *главной функции*, содержащей всю остальную часть программы. Часть функций создается самим программистом, другая часть — библиотечные функции — поставляется пользователю со средой программирования и используется в процессе разработки программ. При изучении C/C++ мы будем пользоваться специальным видом приложений — консольными приложениями, которые формируются на основе заранее заготовленных в среде проектирования шаблонов.

Консольные (т. е. опорные, базовые) *приложения* — это приложения без графического интерфейса, которые взаимодействуют с пользователем через специальную командную строку или (если они работают в рамках IDE) запускаются специальной командой из главного меню среды. Такие приложения создаются с помощью специального шаблона, доступного из диалогового окна, открывающегося после выполнения команды **File | New | Project**.

Шаблон консольного приложения добавляет в создаваемое приложение необходимые элементы (создается заготовка будущего приложения), после чего разработчик вставляет в этот шаблон свои операторы на языке C/C++. Затем приложение компилируется в автономный исполняемый файл и может быть запущено на выполнение. Общение с пользователем происходит через специальное так называемое консольное окно, открывающееся средой после запуска приложения (в это окно выводятся сообщения программы, через него вводятся данные для расчета и в него же выводятся результаты расчетов).

Компиляция и сборка проекта осуществляется через опцию **Build** главного меню среды (структура главного меню меняется в зависимости от совершаемых действий: при загрузке среды она одна (в ней нет опции **Build**), а например, если мы создадим проект, такая опция появляется). После компиляции и сборки проект можно запустить на выполнение. Запуск на выполнение осуществляется с помощью опции **Debug** главного меню среды.

Изучение C/C++ мы станем осуществлять на примерах: будем создавать программы, разбирать, как они работают с параллельным изучением их структуры. Консольные приложения, которые мы начнем создавать, имеют шаблон вида CLR Console Application. Последние два слова этого названия понятны — консольное приложение. А что означает аббревиатура CLR? В настоящей редакции среды разработки авторы создали два варианта консольного приложения, только разнесли их использование по разным диалоговым окнам: шаблон с CLR задан в папке CLR, а шаблон для обычного консольного приложения задан в папке Win32.

Так что же такое — CLR (Common Language RunTime)? Это специальная среда, которая управляет исполнением программного кода, памятью, потоками данных и работой с удаленными компьютерами, при этом строго обеспечивая безопасность и создавая надежность исполнения кода. CLR является добавкой-расширением C++, введенной фирмой Microsoft, начиная с версии VC++ 2005. В версиях 2005, 2008 подключение к вашему проекту этой среды осуществлялось на этапе компиляции. Там для консольного приложения (по умолчанию) этот режим не поддерживался (т. е. консольное приложение как бы оставалось в старой версии C++, в "родной" (native), как определили ее авторы добавки CLR). В этой старой версии, когда вы в своей программе работали с некоторыми объектами (здесь нам приходится забегать вперед, ибо объекты — это предмет более позднего изучения, когда мы станем знакомиться с классами), то должны были сами заботиться об их размещении в памяти, выделяемой средой. Память для вашего приложения выделяется в так называемой *куче*: в ней вы размещаете свои объекты, там же сами освобождаете память, когда перестаете работать с объектом, иначе куча может переполниться и процесс выполнения приложения прервется. Это так называемая неуправляемая куча. Указатели (о них — позже) на участки памяти в такой куче обозначаются символом "*".

Другое дело, когда включается режим CLR. Такое приложение отличается от обычного тем, что его заготовка обеспечивает подключение к приложению специального системного пространства `System`, содержащего объекты, размещение в памяти которых надо автоматически регулировать. Так вот: режим CLR работает уже с управляемой кучей памяти, в которой размещение объектов и ее освобождение от них происходит под управлением среды. Такой сервис входит в язык Java, где не надо делить кучу на управляемую и неуправляемую. В этой среде употребляются так называемые регулируемые указатели на объекты.

Регулируемый указатель — это тип указателя, который ссылается на объекты (адреса памяти, по которым можно обращаться к объектам), расположенные в общей регулируемой куче памяти, предоставленной приложению в момент его исполнения. Для таких указателей принято специальное обозначение: вместо символа "*" применяется символ "^".

Создание CLR привело к необходимости разработки аппарата преобразования переменных, относящихся к одной куче, в адреса в другой куче и т. п. Этот процесс назвали *маршализацией*. Существует специальная библиотека, обеспечивающая этот процесс. Однако все это довольно усложнило программирование.

Переход к созданию консольного приложения

Для создания консольного приложения воспользуемся шаблоном CLR-приложения. Для этого необходимо выполнить следующие шаги:

1. Загрузить среду VC++.
2. Выполнить команды главного меню **File | New | Project**. Откроется диалоговое окно, показанное на рис. 1.9. В этом окне выберите опцию **CLR Console Applicaton**, задайте в его нижней части имя будущего проекта в поле **Name**, которое потом перекочует в поле **Solution name**. С помощью кнопки **Browse** установите папку, в которую будет помещен ваш проект. Теперь окно рис. 1.9 станет выглядеть так, как показано на рис. 1.10.

Рис. 1.9. Выход на шаблон создания консольного приложения

3. Нажать кнопку **OK**. В результате получится то, что показано на рис. 1.11.

Заготовка консольного приложения состоит из заголовка главной функции

```
int main(array<System::String ^> ^args)
```

и тела, ограниченного фигурными скобками.

Преобразуем заголовок функции `main(множество аргументов функции)` к виду `main()`, т. е. к виду без аргументов, а из тела удалим оператор `return 0`.

Все это проделаем с помощью Редактора кода, который открывается одновременно с появлением заготовки консольного приложения на экране (заготовка сразу помещается в поле Редактора кода). Чтобы убедиться, что вы находитесь в Редакторе, щелкните кнопкой мыши в любом месте поля заготовки и увидите, что курсор

Рис. 1.10. Формирование консольного приложения

Рис. 1.11. Вид заготовки консольного приложения

установится в месте вашего щелчка (Редактор ждет в этой точке ваших дальнейших действий). Далее можно набирать любой текст, как в обычном современном текстовом редакторе, работать клавишами <Delete>, <Backspace>, клавишами-стрелками и другими необходимыми для ввода и редактирования клавишами.

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

Когда мы формировали заготовку консольного приложения, мы видели, что при задании имени (**Name**) приложения формировалось и некое поле **Solution name** (решение). Дело в том, что среда VC++ оформляет создаваемое приложение в виде двух контейнеров, вложенных один в другой. Один (главный контейнер) называется **Solution** (решение), а другой — **Project** (проект). Проект определен как конфигурация (каркас, контейнер), объединяющий группу файлов.

В рамках проекта создается программа, в том числе и подлежащая исполнению, т. е. откомпилированная и построенная. Каждый проект содержит по крайней мере две подконфигурации: отладочную и обычную (исполнительскую). Это задается в выпадающем меню, которое по умолчанию установлено на опцию **Debug** (отладка). Это выпадающее меню находится в строке окна среды, расположенной ниже строки главного меню (рис. 1.12).

Рис. 1.12. Установка режима исполнения программы

Проекты являются частью другого каркаса, другого контейнера, который называется **Solution** (решение) и который отражает взаимосвязь между проектами: одно решение может содержать множество проектов, а проект содержит множество элементов, обеспечивающих существование приложения как такового. Можно сказать,

что *решение* — это не что иное, как группа объединенных проектов. Назовем его просто *группа проектов*, чтобы термин "решение" не вводил нас в заблуждение. Существует специальный инструмент работы с группой проектов, называемый **Solution Explorer**. К нему можно добраться через опцию **View** меню среды разработки. Сама среда автоматически формирует создаваемое приложение как группу проектов, содержащих собственно проект (это видно из рис. 1.13).

Рис. 1.13. Группа проектов одного решения

Такой подход к оформлению приложения позволяет работать с группой проектов как с одним целым, что ускоряет процесс разработки приложений.

Добавка нового или существующего проекта к уже существующему решению осуществляется через опцию главного меню **File** (рис. 1.14).

Добавленный проект, как и любой другой, входящий в группу проектов, можно удалить. Для этого надо встать мышью на имя этого проекта, открыть его контекстное меню и выполнить в нем команду **Remove**. (Контекстное меню, как известно, открывается правой кнопкой мыши, когда ее курсор находится на имени объекта, чье контекстное меню требуется открыть.)

Примечание

Часто бывает необходимо быстро загрузить ранее сохраненные проекты (не искать их в каталогах). Для этого существует команда главного меню **File | Recent Projects and Solutions**, которая открывает меню с названиями ранее выполненных проектов и путями к ним.

Удалим из группы проектов проект 2 и оставим в ней проект 1. Откроем опцию **View** главного меню и выполним в ней команду **Start Page**. Получим вид рабочего стола среды, показанный на рис. 1.15.

Рис. 1.14. Добавление существующего проекта в решение

Рис. 1.15. Вид рабочего стола среды после удаления одного проекта из группы

Из рисунка мы видим, что под заголовком **Recent Projects** (где указаны проекты, с которыми недавно работала среда) имеется список проектов. К каждому из них можно вернуться, если щелкнуть на его имени. Выполнив команду **Remove** (Удалить проект), мы всего лишь удалили сведения о проекте из окна **Solution Explorer**, но не из памяти, поэтому-то и возможно восстановление проекта.

Чтобы удалить проект из памяти, надо найти папку, в которой создан проект, удалить его из папки, а потом щелкнуть на его имени под заголовком **Recent Projects**, показанном на рис. 1.15. Если проект из памяти был вами удален, то после щелчка среда выдаст сообщение, что такого проекта нет, и попросит подтвердить его удаление. При подтверждении имя проекта удалится из окна **Recent Projects**. Таким способом вы сможете удалять ненужные проекты не только из группы, но и из окна **Recent Projects** на странице **Start Page**, чтобы они вас не раздражали.

После всех манипуляций с проектами мы получим созданную заготовку нашего приложения, показанную на рис. 1.16.

Рис. 1.16. Заготовка консольного приложения и стартовая страница

Мы видим, что над самой заготовкой консольного приложения имеются две вкладки: **Start Page** и **ConsoleApplication1.cpp**. Если щелкнуть мышью на 2-й из них, то на рабочем столе появится консольная заготовка-проект. Расширение cpp означает, что это исходный текст проекта на языке C++. Если щелкнуть мышью на вкладке **Start Page**, то на рабочем столе появится стартовая страница среды. Таким способом можно переключаться с одного объекта на другой. Вообще, если вы добавите

к проекту еще какой-нибудь элемент, воспользовавшись контекстным меню проекта и выполнив команду **Add** (например, новый CPP-файл), то для добавленного элемента тут же на рабочем столе откроется новая вкладка с именем добавленного элемента, чтобы можно было переключаться между элементами проекта.

Теперь посмотрим некоторые файлы, попавшие в проект, показанный на рис. 1.16:

- **ConsoleApplication1.cpp** — главный исходный файл и точка входа в создаваемое приложение (**ConsoleApplication1** — это в данном случае имя проекта);
- **stdafx.cpp** — подключает для компиляции файл **stdafx.h**;
- **stdafx.h** — если для проекта требуются какие-то дополнительные файлы оглавления, то они задаются пользователем в этом файле;
- **ReadMe.txt** — файл, описывающий некоторые из созданных по шаблону консольного приложения файлов проекта.

Посмотреть содержимое указанных файлов можно либо через их контекстные меню, если выполнить в них команду **Open**, либо просто щелкнуть мышью на имени файла. В этом случае в окне справа от окна **Solution Explorer** появляется содержимое файла, а над содержимым — вкладка с именем открываемого файла.

Типы данных, простые переменные и основные операторы цикла. 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 исчезло, пока мы рассматриваем, что там появилось на экране. Вывод на экран выполняет оператор `printf("Hello!\n");`, а задержку изображения — оператор `_getch();`.

Заметим, что *оператором* в C/C++ называют некоторое выражение C/C++, оканчивающееся точкой с запятой. В первый оператор входит функция `printf("Hello!\n")`, а во второй — функция `_getch()` (эта функция C/C++ введена вместо ранее использовавшейся функции `getch()`).

В итоге наше консольное приложение будет иметь вид, представленный на рис. 1.17.

Чтобы приложение заработало, его надо *откомпилировать*, т. е. перевести написанное на языке C/C++ в машинные коды. Для этого запускается программа-компилятор. Запускается она либо нажатием клавиши <F7>, либо выполнением опции главного меню **Build | Solution**. Если мы проделаем подобные действия, то получим картинку, показанную на рис. 1.18.

Рис. 1.17. Вид консольного приложения до компиляции

Рис. 1.18. Результат 1-й компиляции приложения

Рисунок показывает, что наша компиляция не удалась: в окне вывода высветились сообщения об ошибках. В частности, мы видим сообщение: "error C3861: '_getch': identifier not found". Это означает, что компилятор не узнает функцию `_getch()`. Точно так же компилятор ничего не нашел относительно функции `printf()`. Если кнопкой мыши дважды щелкнуть на каждой строке с информацией об ошибке, то в поле функции `main()`, т. е. в нашей программе, в поле подшивки (вертикальная полоса слева от текста) отметится та строка, в которой эта ошибка обнаружена. Этот процесс также показан на рис. 1.19.

Рис. 1.19. Обнаружение операторов с ошибками в теле программы по результатам компиляции

А теперь разберемся с обнаруженными ошибками.

Щелкнем дважды на имени функции `_getch()`, чтобы пометить ее, и нажмем клавишу <F1>. Откроется окно помощи. В появившемся окне мы найдем сведения о необходимой нам функции, в том числе и о местонахождении ее описания: Required header file conio.h (Требуется файл оглавления conio.h). Header file — файл оглавления, отсюда и символ h в его расширении. Если этот файл подключить к нашей программе, то компилятор станет находить сведения об этой функции и ошибка устранится.

Включение файла с описанием функции осуществляется оператором `#include` — это оператор компилятора. Он включает в текст программного модуля файл, который указан в угловых скобках. Вы и сами можете задавать в подобных файлах необходимую для вашей программы информацию. Тогда для ее подключения к вашей

программе имя такого файла в операторе `#include` должно быть в двойных кавычках.

Итак, в заголовке программы пишем `#include <conio.h>`. Аналогично поступаем и со второй ошибкой компиляции: находим, что чтобы компилятор "узнал" функцию `printf()`, надо к программе подключить файл оглавления `#include <stdio.h>`.

Вид появившейся на экране справки после нажатия клавиши <F1> на отмеченной функции `printf()` показан на рис. 1.20.

Рис. 1.20. Справка по функции `printf()`

Теперь запускаем компилятор клавишей <F7>. Результат показан на рис. 1.21. Заметим, что Редактор текста программы рассматривает две косые черты `//` как начало комментария — набора символов, не участвующих в выполнении программы. Заметим также, что на самом деле не запускают отдельно компиляцию, а выполняют режим **Build Solution** (откомпилировать и построить). Такой режим мы использовали, когда первый раз запускали нашу программу.

Как видно из рисунка, ошибки исчезли. Теперь можно запускать программу на выполнение. Для этого в главном меню среды требуется открыть выпадающее меню опции **Debug** и выбрать из него желаемое действие по запуску сформированной для выполнения программы:

- старт с отладчиком (**Start Debugging F5**);
- старт без подключения отладчика (**Start Without Debugging Ctrl + F5**).

Рис. 1.21. Текст программы после подключения необходимых библиотек и результат его компиляции

Если программу выполнить, то в окне черного цвета (рис. 1.22), которое называется консольным окном, высветится текст "Hello!". Если теперь нажать любую клавишу, то программа не завершится, а выдаст сообщение "Для продолжения нажмите любую клавишу". Нажимаем любую клавишу и снова видим текст программы, т. к. окно вывода закрывается.

Рис. 1.22. Результат выполнения консольной программы

Примечание

Здесь надо отметить, что в этом примере видна новинка версии 2011 по сравнению с версией 2010: в последней приходилось вставлять в программу функцию `_getch()`,

которая ждет ввода любого символа с клавиатуры и этим задерживает закрытие экрана. Мы так и поступили, когда составили текст программы. Однако это оказалось излишним: разработчики среды сами позаботились, чтобы окно вывода не закрывалось, не убегало, а задерживалось до нажатия любой клавиши на экране. На такое усовершенствование фирме Microsoft понадобилось около десятка лет, если не больше. Поздравим фирму с новым достижением.

Сохраним новый проект, выполнив команду **File | Save All**.

Поясним суть программы. Любая программа на C/C++ строится из множества 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 в главной функции `main()` использованы две функции — это `printf()` и `_getch()`.

Функция `printf()` в качестве аргумента имеет строку символов (символы, заключенные в двойные кавычки). Среди символов этой строки есть специальный символ: `\n`. Это так называемый *управляющий символ* — один из первых 32 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` будет прочтен, содержимое буфера тут же передастся на устройство вывода (в данном случае — на экран). При создании консольного (неграфического) интерфейса с окном консольного приложения всегда автоматически связываются два файла: один из них — для ввода данных с клавиатуры, а другой — для вывода данных на экран. О функции `_getch()` мы говорили выше. При наличии `_getch()` программа станет ждать, пока будет введен хотя бы один символ. Как только такой символ будет введен (например, символ 'a'), функция завершит свою работу, а вместе с ней завершит работу и вся программа, т. к. функция была последним исполняемым оператором (выполнение программы идет сверху вниз подряд, если не встречаются операторы, изменяющие последовательность исполнения).

Функция `_getch()` — это функция ввода одного символа с клавиатуры: она ждет нажатия какой-либо клавиши. Благодаря этой функции результат выполнения про-

граммы задерживается на экране до тех пор, пока мы не нажмем любую клавишу. Начинающий программист должен знать о подобном приеме задержки "убегания" экрана. Следует отметить, что основное назначение функции `_getch()` — вводить символы с клавиатуры и передавать их символьным переменным, о которых речь пойдет чуть позже.

Теперь, чтобы начать создавать новую программу, надо старую закрыть. Для этого выполняем опцию **File | Close Solution**.

Программа с оператором *while*

Рассмотрим программу вывода на экран 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.

Для этого производим следующие действия:

1. Создаем заготовку консольного приложения. Приложение автоматически сохраняется. Но можно его сохранять и пользуясь командами **Save**, **Save All** опции **File** главного меню среды. Кроме того, если вы забыли сохранить программу и попытаетесь закрыть проект, то среда запросит у вас подтверждение на сохранение, если в проекте были какие-либо изменения.
2. Записываем код новой программы в тело главной функции (листинг 1.1).

Листинг 1.1

```
// 2_2011.cpp

#include "stdafx.h"
#include <stdio.h> //для функции printf()
using namespace System;

int 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;
   }
}
//-------------------------------------------------------------
```

3. Запускаем компилятор и построитель одновременно клавишей <F7>. Программа откомпилируется, построится. Для ее выполнения нажмем комбинацию клавиш <Ctrl>+<F5>. Результат высветится в окне (рис. 1.23).

Рис. 1.23. Результат расчета таблицы температур по Цельсию

Имена и типы переменных

Поясним суть программы из листинга 1.1.

Строка:

```
int lower,upper,step;
```

это так называемые *объявления переменных*, где `lower`, `upper`, `step` — *имена переменных*.

Компилятор соотнесет с этими именами определенные адреса в памяти и, начиная с этих адресов, выделит участки памяти (в байтах) 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 определенное количество байтов памяти (в данном случае 4 байта).

Рассмотрим, как определяется количество памяти, отводимое под переменную, в данной среде программирования.

`int` — этот тип данных занимает место большее или равное, чем тип `short int`, и меньшее или равное, чем тип `long` (т. е. мы видим, что данные этого типа получаются как бы безразмерными в заданных границах). Объекты типа `int` могут объявляться как `signed int` *(целое со знаком)*, так и `unsigned int` *(целое без знака)*.

`Signed int` — это синоним `int`. В дополнение к `int` существует тип данных, названный `__intn`. Здесь `n` задает размер данных. Значения `n` могут быть равными 8, 16, 32 или 64.

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, чтобы облегчить их запоминание (еще говорят, что переменным надо давать мнемонические обозначения) — 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`) равна 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 С 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 (в общем, объединятся разнородные данные) и, в конце концов, выдаст ошибку компиляции.

`float fahr,cels;` — это описание переменных с именами `fahr`, `cels`, но тип этих переменных уже иной. 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.

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" `a`, другая часть данных — с "полочки" `b`, произойдет их суммирование, а результат будет "положен" (записан) на "полочку" `c`.

Знак "=" означает "присвоить", это не знак равенства, а знак операции 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.

Примечание

В VC++ 2011 имеется еще один тип — `auto`. Он ставится перед именем переменной, которая определяется правой частью некоторого выражения. Компилятор по значению правой части для этого типа определяет истинный тип переменной. Например:

```
auto a=1.5;
```

Компилятор присвоит переменной `a` тип `float`.

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 не хватило бы никакой памяти, поскольку шкала может содержать миллиарды точек. В подобных ситуациях выходят из положения так: делают вычисления 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/C++ является оператор `while` (англ. *while* — 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`:

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

Работает оператор так: вначале 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 нуля — 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 C/C++ это записывается в виде:

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

В теле же самого оператора цикла мы записываем на языке C/C++ 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, 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, то начнет снова выполняться тело оператора `while`. Когда же переменная `fahr` примет значение, большее значения `upper`, цикл завершится: начнет выполняться код, следующий за телом оператора `while`. А это будет закрывающая скобка тела главной функции программы, функции `main()`. То есть сама программа фактически завершится, но т. к. консольное окно, в которое выведены результаты расчета, еще не закрыто, то программа не выходит из выполнения. А выйдет только после нажатия любой клавиши на клавиатуре. Только после этого начнет выполняться закрывающая скобка тела главной функции `main()`. После ее обработки наше приложение закончит свою работу.

Поясним операции, примененные при формировании переменной `cels`. Это арифметические операции деления (`/`), умножения (`*`), вычитания (`-`). Операция деления имеет одну особенность: если ее операнды типа `int`, то и результат — всегда целое число, т. к. в этом случае остаток от деления отбрасывается. И 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 так, будто операнды в формате плавающей точки (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 указания на формат переменных `arg1`, `arg2`, ..., `argN` (к какому виду эти переменные должны быть преобразованы). Указания на формат расположены точно в том же порядке, что и сами переменные `arg1`, `arg2`, ..., `argN`.

Обозначение формата всегда начинается с символа `%`, а заканчивается символом типа форматирования: `d` — для переменных типа `int`, `f` — для `float`, `s` — для строк символов и т. д. То есть если в строке `Control` группа символов не начинается с символа `%` и не заканчивается символом типа форматирования, то такие символы выводятся в том виде, как они записаны (например, некоторый текст). Между символом `%` и символом типа форматирования задается ширина поля вывода, количество знаков после точки (для типа `f`) и т. д. Полное определение форматов можно посмотреть в разделе **Help** для функции `printf()` среды разработки (напишите, например, в любом месте текста вашей программы `printf()`, щелкните дважды по этой записи, нажмите клавишу <F1> и вам откроется страница Help, касающаяся функции `printf()`). Так как переменные `cels` и `fahr` относятся к типу `float`, то и в функции `printf()` указан соответствующий формат — `f`. Значение переменной `fahr` выводится целым числом в поле шириной 4 байта, а значение переменной `cels`, имеющее в результате расчетов дробное значение, выводится в поле шириной 6 байтов с одним знаком после точки (формат `%6.1f`).

Строка `Control` может содержать и другие символы, которые выводятся без всякого форматирования (т. е. без преобразования в другую форму).

Оператор *for*

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

Листинг 1.2

```
// 3_2011.cpp

#include "stdafx.h"
#include <stdio.h> //для функции printf()
using namespace System;

int main()
{
    int fahr;
    for(fahr=0; fahr <= 300; fahr= fahr + 20)
    printf("%4d  %6.1f\n",fahr,(5.0/9.0)*(fahr-32.0));
}
```

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, из которых первые два оканчиваются точкой с запятой, третье — круглой скобкой, обозначающей границу заголовочной части `for` (компилятор понимает, 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 происходит так: инициализируется переменная цикла (т. е. получает начальное значение), затем проверяется условие продолжения цикла. Если оно истинно, то сначала выполняется тело оператора (в данном случае, функция `printf()`), затем управление передается в заголовочную часть оператора `for`. После этого 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()`, означающая прекращение работы функции `main()`).

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 С: *"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, имеющие свой адрес и меняющие 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

```
// 4_2011.cpp

#include "stdafx.h"
#include <stdio.h> //для функции printf()
using namespace System;

#define lower 0
#define upper 300
#define step 20

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

int 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 данного оператора никаких точек с запятой не ставится, т. к. это оператор не языка C/C++, а компилятора. И если нам понадобится изменить значения переменных `lower`, `upper`, `step`, то не придется разбираться в тексте программы, а достаточно будет посмотреть в ее начало, быстро найти изменяемые величины и выполнить их модификацию.

Результат расчета температуры по всем трем вариантам, естественно, одинаков и показан на рис. 1.23.

C/C++ Microsoft Visual

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 C имеют тип `char`: переменная `c`, 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, определенное средой программирования.

Среда VC++ обладает еще одним типом данных для описания символа, но по таблице Unicode. В отличие от таблицы ASCII, в соответствии с которой код символа занимает один байт, кодирование по Unicode размещает символ в двух байтах. С таким кодированием мы встретимся позднее.

Компилятор C++ трактует переменные типа `char` как `signed char` (символьная со знаком) и как `unsigned char` (символьная без знака), причем подразумевается, что это данные разных типов. По умолчанию переменные типа `char` рассматриваются компилятором как переменные типа `signed char`. Такие переменные могут преобразовываться в переменные типа `int`. Теперь после введения типа `char` можно определить и тип `short` или `short int`. Этим типом описывают "короткие" переменные целого типа. Такие переменные по определению занимают памяти больше или равно, чем переменные типа `char`, и меньше или равно, чем переменные типа `int`.

В библиотеке C 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);
```

Функции `getchar()` и `putchar()` описаны в файле stdio.h, который в данной версии среды не подключается к заготовке консольной программы автоматически. 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. Фактическая обработка символа начнется только тогда, когда ввод закончится нажатием клавиши <Enter>, а 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 функцию `getchar()` по одному символу в цикле), то следует ввести всю строку символов, нажать клавишу <Enter>, и только тогда мы увидим результат ввода — строка будет состоять из символов, введенных функцией.

Тут возникает вопрос: а как дать знать программе, что ввод группы символов закончен? Ввод данных, как мы в дальнейшем увидим, может осуществляться и из файла. В этом случае на окончание ввода указывает так называемый *признак конца файла*, присутствующий в файле, который формируется функциями ввода файла после окончания его ввода в момент выполнения функции закрытия файла. Этот признак потом при чтении файла обнаруживается с помощью специальной функции `_eof()` (ее описание находится в файле <io.h>). Аббревиатура eof означает end of file.

Но как быть с признаком конца ввода при вводе данных с клавиатуры? Хотелось бы, чтобы и при вводе с клавиатуры был бы такой же порядок формирования признака конца данных. И он действительно имеется: среда рассматривает ввод символов с клавиатуры в виде специального потока данных, который можно завершить нажатием комбинации клавиш <Ctrl>+<z>. При этом формируется символ, значение которого равно –1. Этот символ и будет признаком конца данных, вводимых с клавиатуры функцией ввода. Когда мы в программе "поймаем" этот символ, то должны завершить ввод. Это и будет для нас `eof`. Но такой признак можно и самостоятельно сформировать в программе: задать в ней некий символ (например, знак вопроса (?)), который будет означать конец ввода с клавиатуры. Тогда при вводе символов последним надо будет набрать символ знака вопроса (?). А в самой программе проверять каждый введенный символ на совпадение его со знаком вопроса и при обнаружении такого совпадения завершать ввод.

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 одной строки, то признаком конца ввода файла можно считать `\n`. Если же мы хотим вводить 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) или, как мы видели, какой-либо символ, который имеет отображение на экране в отличие от управляющих. Если воспользоваться стандартным признаком конца файла (нажатием <Ctrl>+<z>), то следует поступать так: после ввода последнего символа последней строки и нажатия клавиши <Enter> (чтобы функция `getchar()` начала обработку введенных символов), требуется нажать комбинацию клавиш <Ctrl>+<z>, затем клавишу <Enter> (чтобы сим-

вол от <Ctrl>+<z> попал на обработку, и тем самым цикл `while` завершился). Если обозначить символическую константу, задающую значения признака конца ввода с клавиатуры через `eof`, то, воспользовавшись оператором `#define`, в начале программы ввода данных с клавиатуры мы можем написать:

```
#define eof -1 // признак конца ввода символов с клавиатуры
```

Здесь `//` — это признак комментария в программе на языке С. 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

```
// 2.1_2011.cpp : main project file.

#include "stdafx.h"
#include <stdio.h>     //для getchar(), putchar(), printf()
#include <conio.h>     //для _getch()
using namespace System;

#define eof  -1        //признак конца файла: Ctrl+Z

int main()
{
 int c;
 printf("Make input>\n");
 c=getchar();
 while(c != eof)
   {
    putchar(c);
    c=getchar();
   }
 _getch();
}
```

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`. Именно поэтому `c` объявлена как `int`.

Функция `printf()` выводит на экран запрос на ввод. Далее вводится первое значение переменной `c` (как мы договаривались ранее, набираем на клавиатуре строку и нажимаем клавишу <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()` истинно (т. е. пока мы не нажали комбинацию клавиш <Ctrl>+<z>), 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 начинает проверять выполнение записанного там условия. Если условие истинно, т. е. введенный в качестве значения переменной `c` символ не соответствует <Ctrl>+<z>, то снова выполняются операторы, находящиеся в теле `while()`.

Как только мы нажмем комбинацию клавиш <Ctrl>+<z>, в переменной `c` появится ее код — число –1. Поскольку условие выполнения оператора `while()` нарушится (значение переменной `c` станет равно `eof`), то управление будет передано следующему после `while()` оператору. В этом случае в консольном окне появится сообщение-задержка закрытия экрана "Для продолжения нажмите любой символ", которое потребует ввода символа с клавиатуры. Пока мы не нажмем какую-либо клавишу, наша программа-приложение будет ждать ввода, а мы в это время будем рассматривать картинку на экране и гадать, что же у нас получилось. Если мы нажмем любую клавишу, то 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()`.

Для закрепления материала желательно выполнить программу в режиме отладки, чтобы детально посмотреть, как все происходит на самом деле. Для этого щелкните мышью в поле подшивки слева от оператора `while(c != eof)`. В этом поле появится пометка в виде красного кружочка. Это так называемая точка прерывания программы (в опции **Debug** главного меню среды вы ее обнаружите под названием **breakpoint**). Запуск программы на выполнение осуществляется в режиме отладки: либо по нажатию клавиши <F5>, либо командами **Debug | Start Debugging**.

Введите, например, символы `123`. Их коды по таблице ASCII: 49, 50, 51. После нажатия клавиши <Enter> программа остановится в точке прерывания (в красном кружочке появится желтая стрелка вправо — это сигнал о том, что произошел останов программы). Начинается цикл обработки введенной последовательности 1, 2, 3. Наведите мышь на переменную `c`. Тут же всплывет подсказка, показывающая значение этой переменной. Вы увидите, что первое значение переменной будет равно 49 — это код единицы, т. е. в обработку поступил первый введенный символ (запомните, что в переменных находятся коды, а не сами значения!). Чтобы выполнить следующий оператор, откройте опцию **Debug** главного меню и посмотрите, какую команду надо в ней выполнить. Это будет команда **Step Over** (или нажатие клавиши <F10>). При этой команде выполняется всегда целый оператор. Но в качестве оператора может выступать, как мы увидим далее, и некоторая функция (или подпрограмма), имя которой с параметрами указано в тексте программы (заголовок) и после имени которой стоит точка с запятой. А вам хочется посмотреть по шагам, как выполняется функция. Чтобы в отладочном режиме попасть внутрь функции, надо после останова на ее имени выполнить опции **Debug | Step into** или нажать клавишу <F11>. Таким образом, последовательно нажимая клавишу <F10>, вы пошагово выполните всю программу, проследив, как и почему выполняются в определенной последовательности операторы программы. Именно так отлаживается любая программа.

В нашем конкретном случае, если вы работали в режиме отладки, надо не запутаться, когда нажимать комбинацию клавиш <Ctrl>+<z>, иначе ничего не получится, потому что среда передает управление то консольному окну, то самой программе. Во-вторых, когда вы попали на шаг обработки закрывающей скобки функции `main()`, *следует отключить отладку* (это делается нажатием <Shift>+<F5>), иначе вам придется пошагово выполнять еще много операторов, завершающих программу, которых мы обычно вне отладки не видим. И еще один момент. В данной программе придется все-таки самому организовывать задержку экрана с помощью функции `_getch()`, потому что после ввода символов в буфере ввода остается один символ <Enter>, которым мы завершаем обработку введенного признака конца ввода <Ctrl>+<z>.

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, когда вместо переменной некоторого типа можно использовать и выражение этого же типа. Воспользуемся данной возможностью и запишем нашу программу в другом виде (листинг 2.2).

Листинг 2.2

```
// 2.2_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(), putchar(), printf()
#include <conio.h>      //для _getch()
using namespace System;
#define eof  -1       //признак конца файла

int main()
{
 int c;
 printf("Make input>\n");
 while((c=getchar())!= eof)
  putchar(c);
 _getch();
}
```

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, не являются ли введенные символы признаком конца файла. Так как в отличие от предыдущего варианта тело оператора `while` состоит только из одного оператора `putchar(c)`, то 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. Вид программы представлен в листинге 2.3.

Листинг 2.3

```
// 2.3_2011.cpp
#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(), printf()
```

```
#include <conio.h>      //для _getch()
using namespace System;
#define eof  -1       //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();
}
```

Здесь мы встречаемся с новым типом данных: `long` — длинное целое. Этот тип (можно и `long int`) применяется для описания больших целых чисел со знаком. По занимаемой памяти этот тип переменных больше или равен типу `int`. Переменные типа `long` могут объявляться как `signed long` или `unsigned long`. В первом случае у переменной будет присутствовать знак, во втором — знак будет подавлен. Это означает в обоих случаях, что знаковый разряд числа с квалификатором `unsigned` будет рассматриваться компилятором как обычный разряд данных, т. е. станет участвовать в операциях наряду с другими разрядами данных.

`signed long` — это синоним `long`. Существует также тип данных `long long`. Переменная этого типа занимает память больше, чем `unsigned long`. Переменные типа `long long` могут быть со знаком или без знака. Соответственно они объявляются как `signed long long` и `unsigned long long`. `signed long long` — это синоним `long long`.

Так как мы подсчитываем количество символов в файле, который (в общем случае) может быть и не "клавиатурным", то должны быть готовы к тому, что в нем будет много символов, и их число превысит допустимое значение, помещающееся в переменной типа `int`. Количество вводимых символов подсчитывается в переменной `nc` типа `long` 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 комбинация клавиш <Ctrl>+<z>. 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 введен-

ный символ по нашему алгоритму не требуется — значение `getchar()` сразу сравнивается с признаком конца файла. Если этот признак еще не введен (мы не нажали комбинацию клавиш <Ctrl>+<z>), 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` (т. е. 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 выполняется, а потом уже результат присваивается переменной `x`. Если находятся после `nc`, то сначала содержимое `nc` присваивается переменной `x`, а потом уже `nc` изменяется на единицу. Этой операции родственна операция `--`, которая ведет себя как и `++`, но не прибавляет, а вычитает единицу.

В функции `printf()` мы снова видим новый формат: `%ld`. По этому формату выводятся числа типа `long`.

Результат работы программы приведен на рис. 2.2. Заметим, что в подсчете символов участвует и символ комбинации клавиш <Ctrl>+<z>, и символ, который вводится как признак конца ввода. Поэтому результат — количество введенных символов — будет на единицу больше, что и видно из рисунка. Чтобы этого не происходило, в программе перед ее завершением из `nc` надо было бы отнять единицу, но тогда это было бы не наглядно для обучающегося.

Рис. 2.2. Результат работы программы листинга 2.3

Подсчет символов в файле. Вариант 2

Изменим несколько предыдущую программу, применив новый оператор и новую операцию (см. по тексту). Программа будет выглядеть, как показано в листинге 2.4.

Листинг 2.4

```
// 2.4_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(),printf()
#include <conio.h>      //для _getch()
```

```
using namespace System;

#define eof  -1       //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();
}
```

Для подсчета символов в файле мы применили переменную нового типа `double` (это длинное `float`), т. к. таких символов, возможно, окажется больше, чем может поместиться в переменную, описанную в предыдущем примере. Размер занимаемой памяти для переменной типа `double` больше или равен, чем для типа `float`, но меньше или равен, чем для типа `long double`. Последний тип совпадает с типом `double`.

В табл. 2.1 приведены значения размеров памяти для основных типов данных в Microsoft C++.

Таблица 2.1. *Размеры памяти для основных типов данных*

Тип	**Размер памяти**
`bool`	1 байт
`char`, `unsigned char`, `signed char`	1 байт
`short`, `unsigned short`	2 байта
`int`, `unsigned int`	4 байта
`__int`*n*	8, 16, 32, 64 или 128 битов (зависит от значения `n`)
`long`, `unsigned long`	4 байта
`float`	4 байта
`double`	8 байтов
`long double1`	8 байтов
`long long`	8 байтов

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 введен символ конца файла, т. е. комбинация клавиш <Ctrl>+<z>;
- выражение, определяющее изменение переменной, по которой, как говорят, "идет цикл" — в нашем случае `nc++`.

Поскольку все удалось разместить в заголовочной части, то в теле оператора `for` нет необходимости. Но формат оператора нужно 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 соответствует комбинации клавиш <Ctrl>+<z>, то снова будет выполняться тело и т. д. В конце концов, когда мы нажмем <Ctrl>+<z>, 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 выдается по формату `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.

Рис. 2.3. Результат работы программы листинга 2.4

Результат работы программы представлен на рис. 2.3. Отметим, что программа подсчитывает и вспомогательные символы.

Подсчет количества строк в файле

Строки файла строк в языке С 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

```
// 2.5_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(),printf()
#include <conio.h>      //для _getch()

using namespace System;

#define eof  -1       //признак конца файла

int main()
{
  int c,nl;
  nl=0;
  printf("Enter your string and press <Enter> and <Ctrl + z> >\n");
   while((c=getchar()) !=eof)

      if(c=='\n')
       {
        nl++;
       }

 printf("String's number is: %d\n",nl);
 _getch();
}
```

Здесь новое по сравнению с предыдущими подобными программами только то, что появилась операция == (равно) и новый оператор `if` — это оператор условного перехода, изменяющий 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 од-

ного оператора `nl++`, который выполняется всякий раз, когда введен не символ конца строки. В противном случае тело `if` не выполняется. Для простоты понимания мы ввели фигурные скобки там, где их можно опустить, выделяя тем самым тело оператора. Тело оператора `while` тоже состоит из одного оператора `if` (неважно, сколько операторов включает тело `if`), поэтому оператор `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: когда мы набираем строки в консольном окне, мы заканчиваем их ввод символом <Enter>. 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 комбинацию клавиш <Ctrl>+<z> (символ конца ввода), 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

```
// 2.6_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(), printf()
#include <conio.h>     //для getch()
```

```
using namespace System;

#define eof -1       //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 <Enter> >\n");
  while((c=getchar()) !=eof)
   {
    if(c != '\n')      //если символ — не конец строки
     nc++;         /*какой бы символ ни ввели (кроме Ctrl+z 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 клавиши "пробел","конец строки", "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 (задерживает отображение
результатов расчетов на экране) */
} //закрывающая скобка функции main()
```

Весь ход работы программы ясен из подробных комментариев. 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 хотя бы 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 более ясного понимания картины).

Результат расчетов приведен на рис. 2.5.

Рис. 2.5. Результат работы программы листинга 2.6

C/C++ Microsoft Visual

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 станем накапливать данные о том, сколько раз встретилась каждая цифра. Первый счетчик — для накапливания данных о количестве имеющихся в файле нулей, второй — для единиц и т. д.

А представим себе, что 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 воспользоваться 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 из 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 у строки символов обязательно есть признак конца — символ с кодом `\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];`, где `i` — номер элемента `(i=0,1,...,99)`.

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. придать его элементам какие-то значения, надо придать 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 так:

```
s[0]='a'; s[1]='b'; s[2]='c';
```

либо так:

```
char s[3]={ 'a','b','c'};
```

либо даже так:

```
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'`: когда мы пишем строку в двойных кавычках, компилятор по этому признаку сам формирует признак конца строки — символ `'\0'`. Поэтому `'b'` — это один символ, а `"b"` — два символа: собственно символ b и признак конца строки \0, сформированный компилятором, когда он распознал запись `"b"`.

Коль скоро мы заговорили о символах, то 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. За кодом нуля идет код единицы (т. е. 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 т. д. Следовательно, разность между кодом числа `i` и кодом нуля равна числу `i`.

Составим программу, подсчитывающую количество нулей в строке символов, количество единиц и т. д. Наша программа будет выглядеть так, как показано в листинге 3.1.

Листинг 3.1

```
// 3.1_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(),printf()
#include <conio.h>      //для _getch()

using namespace System;

#define eof  -1       //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 <Enter> and <Ctrl+z> >\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();
} // 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 комбинации клавиш <Ctrl>+<z>. Вначале все счетчики (т. е. элементы массива) обнуляются, чтобы в них накапливать по единичке, 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 идет с нуля.

Далее идет уже знакомый нам цикл ввода символов, в котором проверяется, входит ли код каждого введенного символа в диапазон кодов от нуля 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 (комбинация клавиш <Ctrl>+<z>) происходит поэлементный вывод 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 1, объявляются так же, как и одинарной размерности, но справа (в квадратных скобках) к объявленному количеству элементов одинарной размерности добавляется количество элементов для следующей размерности и т. д.

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];
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 по каждому месяцу в високосном и невисокосном году".

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 будет собственно количество дней в каждом из двенадцати месяцев. К тому же (т. к. элементы в массивах нумеруются с нуля) для удобства пользования нашим массивом введем еще один искусственный элемент, равный нулю, и поместим его на нулевое место во второй размерности. Это обеспечит более приемлемое обращение к массиву, т. к. не надо каждый раз помнить, что индексы массива идут с нуля. 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 двадцати 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.

C/C++ Microsoft Visual

ГЛАВА 4

Создание и использование функций

В процессе программной реализации алгоритмов часто возникает необходимость выполнения повторяющихся действий на разных группах данных. Например, требуется вычислять синусы заданных величин или начислять заработную плату работникам. Ясно, что неразумно 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 создать программу, которая бы начисляла зарплату любому работнику, получая данные конкретного работника. 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 $y = f(x_1, x_2,..., x_N)$, то на конкретном наборе данных $\{x_{11}, x_{21},..., x_{N1}\}$ эта функция возвратит вычисленное ею значение $y1 = f(x_{11}, x_{21},..., x_{N1})$. В данном случае можем сказать, что аргументы $x_1, x_2,..., x_N$ — это формальные параметры функции $f()$, а $x_{11}, x_{21},..., x_{N1}$ — 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, 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 языке С.

Примечание

В языке С типом функции может быть специальный тип `void()`, когда функция ничего не возвращает. Такая функция вырождается в подпрограмму — конструкцию, которая просто производит некоторые вычисления.

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, мы в ее заголовочную часть подставляли вместо 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 используем их с их объявленными именами при создании программного кода так, как будто известны их значения. Это возможно благодаря тому, что компилятор, когда начнет компилировать функцию, соотнесет 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 своего рода подпрограмма, да еще и возвращающая некоторое значение, а не только получающая 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, отведенные для формальных параметров). Но программный код тела функции как раз и работает с этими "полочками", содержащими параметры. Поскольку тело строится так, что оно работает с "полочками", то остается только класть на них разные данные и получать соот-

ветствующие результаты. Вот это и осуществляется, когда мы каждый раз передаем 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 и ни с чем другим, не может изменять 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, то переменные, адреса которых переданы в качестве фактических параметров, смогут изменяться в теле функции. Ведь по адресу можно записать все, что угодно, где бы он ни находился (в стеке или в обычной памяти).

Функции в основной программе должны описываться в том месте программы, чтобы компилятор мог ее обработать до момента ее использования в программе. В частности, если функцию описать до начала самой программы (до `main()`), либо текст функции подключается к `main()` оператором `#include` (который тоже стоит раньше `main()` в тексте), если функция расположена где-то в другом файле, то проблем с компиляцией программы не будет. Некоторые понятия, касающиеся функций, мы рассмотрим в этой же главе дальше.

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 `getline()` представлена в листинге 4.1.

Листинг 4.1

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

 int 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);
  }
```

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 вводиться в массив `s[]`, размерность которого указана в вызывающей программе, то мы должны задавать параметр `lim`, значение которого не должно превосходить размерности массива (т. е. определенной длины 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":

- 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`);
- будет введен признак конца ввода (он должен быть задан в вызывающей программе так, чтобы был известен в этой функции);
- ввод строки завершится, когда будет введен признак конца строки символ `\n`.

Первым в сложном условии продолжения/окончания цикла стоит выражение:

```
i<lim-1
```

поскольку в C при вычислении такого рода выражений действует правило: *если при вычислении части выражения становится ясным его значение, то 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 не только через возврат количества введенных символов, но и через массив `s[]`, который играет роль выходного параметра.

Примечание

Это возможно, потому что `s` (точнее `s[0]`) — это адрес первого элемента массива, как определено в языке С. Мы уже говорили, что в теле функции можно изменять значения тех переменных, которые передают в функцию не свои значения, а свои адреса. Массив `s[]` передан в функцию своим начальным адресом. Вот мы и изменяли значение переменной `s[]`.

Когда определяют функцию как подпрограмму, возвращающую обязательно какое-то значение, то имеют в виду, что 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, мы заметили, что параметры функции могут быть входными и выходными, и что не следует путать с ними возвращаемые значения, которые идут через оператор `return()`.

Детально посмотреть, как работает функция или любая другая программа, можно, воспользовавшись программой-отладчиком (Debugger), о которой мы говорили в предыдущей главе. Напомним, что включить его можно в любой точке программы, щелкнув мышью в поле подшивки редактора текстов (Text Editor).

При этом в поле подшивки редактора появится красный кружочек. Убрать точку останова (Breakpoint) можно, повторно щелкнув на красном кружочке. Запуск программы на выполнение должен осуществляться либо с помощью нажатия клавиши <F5>, либо посредством команды **Debug | Start Debugging**. В обоих случаях включается режим отладки программы. От образовавшейся точки останова можно либо войти внутрь некоторой функции, нажав клавишу <F11>, если мы остановились на этой функции, и тем самым проследить, как она пошагово выполняется, либо дальше двигаться по тексту программы шаг за шагом, нажимая клавишу <F10>. При этом будут одновременно выполняться все операторы, находящиеся в той строке, на которую указывает желтая стрелка. Эта стрелка образуется в поле подшивки, когда после пуска программы происходит останов на заданной точке, и когда мы начинаем двигаться дальше с помощью клавиши <F10>, выполняя одну строку за другой. В этом случае мы имеем возможность просмотра значений переменных, что очень важно для процесса отладки программы: достаточно навести на имя переменной указатель мыши и немного подождать, как рядом с указателем мыши появится изображение содержимого переменной (рис. 4.1).

Рис. 4.1. Подключение отладчика к программе

Функция выделения подстроки из строки

Пример программы с функцией `substr()` 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[]` (вспомним, что строка символов в языке C задается массивом символов), порядковый номер символа, 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 имеем 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, ко-

торую в этом случае называют *"операция Запятая"*. Соответственно и в третьем выражении оператора `for`, где наращиваются переменные цикла, имеется более одной переменной, которые также разделены запятой.

Цикл начинается заданием значений переменных цикла: индекс `i` начинается с нуля, т. к. массив формируется с нулевого элемента. В нашем случае индекс `i` — это 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 до тех пор, 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 языка С в отношении признака конца строки символов: добавить в конец символ `\0`.

Функция копирования строки в строку

Функция `copy()` 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`. Поскольку на входе имеется строка символов, то она обязательно заканчивается символом `'\0'`. В условии окончания цикла имеется выражение:

```
save[i]=line[i]) != '\0'
```

Чтобы вычислить это выражение, потребуется, во-первых, переслать 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 элемента массива возрастет на единицу, после чего станет готовым к тому, чтобы по нему переслать сле-

дующий 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 работы функций `getline()`, `substr()`, `copy()`. Эта программа приведена в листинге 4.4.

Листинг 4.4

```
// 4.1_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar().printf()
#include <conio.h>      //для _getch()
#include <stdlib.h>     //для exit()

using namespace System;

#define eof  -1       //признак конца ввода (Ctrl+z)
#define maxline 1000  //размерность массивов (максимальная длина строк)
#define from 2       /*константа для выделения подстроки (с этого символа будет
начинаться выделение) */
#define howmany 3     /*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';
   }
//-------------------------------------------------------------
 int 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++;
  }
//-------------------------------------------------------

int main()
{
     char s[maxline],v[maxline],w[maxline];
     printf("Enter your string\n");
     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 разной 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 другая задача: познакомиться с работой функций и новой функцией `exit()`). Результат работы основной программы приведен 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 и один раз был проинициализирован (т. е. 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 самом этом 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. к. 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 в 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, в файле F1.h имеем: `int a=5;` (объявление с инициализацией). В нашей программе мы должны выполнить `#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 этом случае в программе мы должны написать: `extern int a;` (но не `extern int a=0;`, поскольку 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). Основная программа будет вычислять строку максимальной длины из введенного множества строк.

Листинг 4.5

```
// 4.2_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(),printf()
#include <conio.h>     //для _getch()
#include <string.h>   //для strcpy()
using namespace System;

#define eof  -1       //признак конца ввода (Ctrl+z)
#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[].
getline() возвращает длину введенной строки; lim — максимальное количество
символов, которое можно ввести в строку line[] */
 int 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 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;
printf("Enter some strings >\n");
while((len=getline()) >1)
 {
     /*Когда введем Ctrl+z или Enter, то длина строки станет = 1
      за счет учета признака конца '\0' */
 if(len > max)
   {
    max=len;
    copy();
   }
 }
 /*Когда мы нажмем комбинацию клавиш <Ctrl>+<z> или <Enter>
  (конец ввода), то getline() 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 приведен на рис. 4.3.

Рис. 4.3. Результат работы программы листинга 4.5

Область действия переменных

Мы уже говорили, что существуют внешние переменные, известные во всех объявленных ниже их функциях и блоках. Эти переменные могут быть объявлены как в тексте разрабатываемой программы, так и в текстах внешних файлов, которые следует к этой программе подключать операторами `#include`. В таких случаях подобные переменные объявляются в программе, но с атрибутом `extern` (например, `extern int a;`) и не могут быть инициализированы при объявлении в 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 неверно, если бы программисты каждой группы описывали в своих программах общие данные. Гораздо проще все эти данные описать (объявить) в одном файле с расширением h, чтобы каждый программист смог им пользоваться (подключать к своей программе). К тому же при таком подходе получится, что у всех программ, использующих общий файл, переменные будут иметь одинаковые наименования и смысл, что значительно облегчит дальнейшее сопровождение программ.

Как создать свой внешний файл

В листинге 4.6 приведем текст программы проверки на принадлежность к внешнему файлу некоторой переменной `a`, объявленной и инициализированной числом 20.

Листинг 4.6

```
// 4.4_2010.cpp
//

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>  //getch()
#include "D:\\2011.h"
```

```
using namespace System;

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

Общий h-файл формируется с расширением h обычным Блокнотом. В данном примере редактором был записан файл с расширением h, содержащий всего одну строку: `int a=20;`. В результате работы программы в окно вывелось сообщение (рис. 4.4).

Рис. 4.4. Результат работы программы с внешним файлом, подготовленным пользователем

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`.

В листинге 4.7 приведен текст проверочной программы. Для простоты взята уже известная нам функция ввода строки с клавиатуры `getline()`, и в ее тело вставлено объявление статической переменной `j`, которой там же присваивается значение количества введенных символов `i`. В режиме Отладчика (**Start Debugging**) можно увидеть, что локальная переменная `j` сохраняет свое значение и после выхода из функции, и после нового входа в функцию.

Листинг 4.7

```
// 4.3_2011

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar(),printf()
#include <conio.h>
```

```
#define eof -1          //Ctrl+z
#define maxline 100

using namespace System;

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

int 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);
  }
//----------------------------------------
int 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 (режим **Start Debugging**) на оператор `j=i` и в цикле работы `getline()`, убедитесь, что переменная `j` сохраняет свое значение в других циклах запуска `getline()`. Не забудьте, что чтобы попасть в отладочном режиме внутрь функции, надо при останове на ее имени нажать клавишу <F11>, а дальше двигаться внутри по строкам путем нажатия <F10>.

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)` — освободить память.

Рекурсивные функции

Рекурсивные функции — 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: $n! = (n-1)! \cdot n$, где $0! = 1$. Чтобы вычислить факториал на шаге n, надо воспользоваться факториалом, вычисленном 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+1);
      else
       {
         i=i * fact(i-1);
          return(i);
        }
      }
```

Некоторые итоговые данные по изучению функций

Функция в С объявляется, определяется, вызывается. Определение функции состоит из заголовка и тела. Заголовок функции состоит из спецификаторов объявления, имени функции и списка параметров. Тело функции образуется блоком операторов.

Вызов функции — это выражение со списком (возможно пустым) выражений в круглых скобках. При разборе выражения вызова транслятору С требуется информация об основных характеристиках вызываемой функции. К таковым, прежде всего, относятся типы параметров, а также тип возвращаемого значения функции. При этом тип возвращаемого значения оказывается актуален лишь в том случае, если выражение вызова оказывается частью более сложного выражения. Если определение функции встречается транслятору до выражения вызова, никаких проблем не возникает. Вся необходимая к этому моменту информация о функции оказывается доступной из ее определения. При этом не принципиально фактическое расположение определения функции (в начале программы или в ее конце). Главное, чтобы в момент разбора выражения вызова транслятор знал бы все необходимое об этой функции. Но как только в исходном файле возникает ситуация, при которой вызов функции появляется в тексте программы до определения функции, разбор выраже-

ния вызова завершается ошибкой. Итак, каждая функция, перед тем как она будет вызвана, по крайней мере, должна быть объявлена. Это обязательное условие успешной трансляции.

Объявление и определение функции — разные вещи. Объект может быть много раз объявлен, но только один раз определен. Итак, в файле программы созданная функция может располагаться в любом месте файла (до `main()`, после `main()`). Однако с одним ограничением: в момент ее вызова на выполнение сведения об этой функции программе должны быть известны. Как этого добиться? Можно помещать определение функции перед `main()`. Но вдруг вам это покажется по каким-то причинам неудобным и вы захотите расположить определение функции где-то после конца `main()`. Тогда для компилятора вы должны оставить информацию об этой функции. Это делается с помощью так называемого прототипа функции.

Прототип функции — это заголовок функции с точкой запятой после него. Например, `void ZZ(int ppp);`. Информации прототипа достаточно для компилятора, чтобы он создал сведения об этой функции. Когда задан прототип, спокойно можно поместить определение функции в место файла, где вам удобно с ней работать: хоть перед, хоть после `main()`. На практике обычно при создании функции до начала функции `main()` пишут прототип функции, а за ним — само определение функции. Чтобы не запутаться. Особенно, когда программа большая и всяких функций — тьма. Даже если в дальнейшем вы переместите определение функции в конец файла программы, прототип останется до `main()` и тем самым обеспечит успешную компиляцию программы. Вот пример задания функции:

```
void  ZZ(int ppp);
void  ZZ(int ppp)
{
  Тело функции
}
```

Прототип функции — информация для транслятора. В объявлении функции сосредоточена вся необходимая транслятору информация о функции — о списке ее параметров и типе возвращаемого значения. И это все, что в момент трансляции вызова необходимо транслятору для осуществления контроля над типами. Несоответствия типов параметров в прототипе и определении функции выявляются на стадии окончательной сборки программы. Несоответствие спецификации возвращаемого значения в объявлении прототипа и определении функции также является ошибкой.

Значения параметров по умолчанию — прерогатива не C, а его дальнейшего развития — C++. Этот вопрос мы рассматриваем в данном контексте, потому что сегодня в чистом C вряд ли кто работает. В C++ при вызове функций можно вообще опускать параметры. В этом случае компилятор будет брать значения параметров по умолчанию. Значения по умолчанию для параметров указываются в заголовке функции при ее определении. Правила работы с функциями с указанием не всех параметров такие:

- значения по умолчанию можно задавать не для всех параметров функции;
- если вызов функции опускает значения одного или нескольких параметров, C++ будет использовать значения по умолчанию;

- если вызов функции опускает значение определенного параметра, то должны быть опущены и значения всех последующих параметров.

Параметры по умолчанию (напоминаю: в C++) задаются в виде:

```
<тип выхода> <имя функции>(<тип параметра> = <значение параметра
по умолчанию>, ...)
```

Например, `void f(int a=25, float b=12) { Операторы функции}`

Возможные формы вызова функции:

```
f(); f(1002); f(14,153);
```

Статическая переменная обычно инициализируется. Инициализация выполняется один раз при первом вызове функции. Например,

```
double average (double x)
{
  static double count = 0;
  static double sum = 0 ;
                ++count;
   sum += x;
   return( sum);
}
```

В дальнейшем при обращении к этой функции переменные `count`, `sum` будут текущими и инициализироваться не будут.

Рассмотрим правила, определяющие область действия идентификатора переменной. Область действия идентификатора — это часть программы, в которой на идентификатор можно ссылаться. Существуют четыре области действия идентификатора:

- функция;
- файл программы;
- блок;
- прототип функции.

Идентификатор, объявленный вне любой функции (на внешнем уровне), имеет область действия — файл программы. Такой идентификатор известен всем функциям от точки его объявления до конца файла программы. Глобальные переменные, описания функции, прототипы функции, находящиеся вне функции, — у всех область действия — файл программы.

Ключевое слово `inline` в C++. Когда выполняется программа, в которой имеется много обращений к некоторой функции в разных местах программы, то во всех местах, где ваша программа вызывает функцию, компилятор C++ (на этапе компиляции, конечно) помещает в программу специальные инструкции, которые заносят параметры функции в стек и затем выполняют переход к командам этой функции. Когда операторы функции завершаются, выполнение программы продолжается с первого оператора, который следует за вызовом функции. Помещение аргументов

в стек и переход в функцию и выход из нее вносят издержки, из-за которых ваша программа выполняется немного медленнее, чем если бы она размещала те же операторы тела функции без ее вызова прямо внутри программы при каждой ссылке на функцию. Для этой цели существует специальное ключевое слово `inline`, которое пишется перед именем функции и которое обрабатывает компилятор, формируя внутри программы не вызовы функции, а проставляя на месте ее вызова операторы тела функции с учетом значений параметров, конечно. Для программиста остается видимым только вызов функции, а на самом деле в случае `inline` в теле программы компилятором проставлены операторы тела функции и при выполнении программы не тратится время на дополнительные операции по вызову функции, что ускоряет процесс расчета. Функции с ключевым словом `inline` называются встроенными. К встроенной функции относится и возможность включения блока на языке ассемблера в вашу программу. Существует специальная конструкция

```
asm
{
      MOV AH,2
      MOV DL,7
      INT 21H
   }
```

которая позволяет это делать. Конструкция состоит из ключевого слова `asm`, после него идет тело конструкции, ограниченное фигурными скобками, внутри которых помещаются команды ассемблера.

Перегрузка функций

Это тоже прерогатива C++. Изучение понятия "Перегрузка функции" нам будет полезна при изучении понятия "Классы".

При определении функций в своих программах вы должны указать тип возвращаемого функцией значения, а также количество параметров и тип каждого из них. В прошлом (если вы программировали на языке C), когда у вас была некая функция `f()`, которая работала с двумя целыми значениями, а вы хотели бы использовать подобную функцию для сложения трех целых значений, вам следовало создать функцию с другим именем. Аналогично, если бы вы хотели использовать подобную функцию для сложения значений типа `float`, то вам была бы необходима еще одна функция с еще одним именем. Чтобы избежать дублирования функции, C++ позволяет вам определять несколько функций с одним и тем же именем. В процессе компиляции компилятор C++ принимает во внимание количество аргументов, используемых каждой функцией, и затем вызывает именно требуемую функцию. Предоставление компилятору выбора среди нескольких функций с одним и тем же именем, но с разным количеством и типами параметров называется *перегрузкой.* Для перегрузки функций просто определите две функции с одним и тем же именем и типом возвращаемого значения, которые отличаются количеством параметров или их типом.

Перегрузка функций является особенностью языка C++, которой нет в языке C.

Перегруженные функции не обязаны возвращать значения одинакового типа по той причине, что компилятор однозначно идентифицирует функцию по ее имени и набору ее аргументов. Для компилятора функции с одинаковыми именами, но различными типами аргументов — разные функции, поэтому тип возвращаемого значения — прерогатива каждой функции.

Использование шаблонов функций

При создании функций иногда возникают ситуации, когда две функции выполняют одинаковую обработку, но работают с разными типами данных (например, одна использует параметры типа `int`, а другая типа `float`). Мы уже знаем, что с помощью механизма перегрузки функций можно использовать одно и то же имя для функций, выполняющих разные действия и имеющих разные типы параметров. Однако, если функции возвращают значения разных типов, приходится использовать для них уникальные имена. Предположим, например, что у вас есть функция с именем `max()`, которая возвращает максимальное из двух целых значений. Если позже нам потребуется подобная функция, которая возвращает максимальное из двух значений с плавающей точкой, придется определить другую функцию, например `fmax()`. В C++ существуют специальные средства — шаблоны, которые помогают решить затронутую выше проблему. Оказывается, что:

- шаблон определяет набор операторов, с помощью которых ваши программы позже могут создать несколько функций;
- программы часто используют шаблоны функций для быстрого определения нескольких функций, которые с помощью одинаковых операторов работают с параметрами разных типов или имеют разные типы возвращаемых значений;
- шаблоны функций имеют специфичные имена, которые соответствуют имени функции, используемому вами в программе;
- после того как ваша программа определила шаблон функции, она в дальнейшем может создать конкретную функцию, используя этот шаблон для задания прототипа, который включает имя данного шаблона, возвращаемое функцией значение и типы параметров;
- в процессе компиляции компилятор C++ будет создавать в вашей программе функции с использованием типов, указанных в прототипах функций, которые ссылаются на имя шаблона.

Создание простого шаблона функции

Шаблон функции определяет типонезависимую функцию. С помощью такого шаблона ваши программы в дальнейшем могут определить конкретные функции с требуемыми типами. Например, ниже определен шаблон для функции с именем `max()`, которая возвращает большее из двух значений:

```
template<class T> T max(T a, T b)
{
   if (a > b) return(a);
   else return(b);
}
```

Буква `T` в данном случае представляет собой общий тип шаблона. После определения шаблона внутри вашей программы вы объявляете прототипы (заголовки) функций для каждого требуемого вам типа. В случае шаблона `max` следующие прототипы создают функции типа `float` и `int`.

```
float max(float, float);
int max(int, int);
```

Когда компилятор C++ встретит эти прототипы, то при построении функции он заменит тип шаблона `T` указанным вами типом. В случае с типом `float` функция `max` после замены примет следующий вид:

```
template<class T> T max(T a, T b)
{
   if (a > b) return(a) ;
   else return(b);
}
float max(float a, float b)
{
   if (a > b) return(a) ;
   else return(b);
}
```

В процессе компиляции компилятор C++ автоматически создает операторы для построения одной функции, работающей с типом `int`, и второй функции, работающей с типом `float`. Поскольку компилятор C++ управляет операторами, соответствующими функциям, которые вы создаете с помощью шаблонов, он позволяет вам использовать одинаковые имена для функций, которые возвращают значения разных типов. Вы не смогли бы это сделать, используя только перегрузку функций.

Шаблоны, которые используют несколько типов

Предыдущее определение шаблона для функции `max` использовало единственный общий тип `T`. Очень часто в шаблоне функции требуется указать несколько типов. Например, следующие операторы создают шаблон для функции `show_array`, которая выводит элементы массива. Шаблон использует тип `T` для определения типа массива и тип `T1` для указания типа параметра `count`:

```
template<class T,class T1> void  show_array(T *array,T1 count)
{
   T1 index;
```

```
   for (index =0; index < count; index++)
  /*далее идет оператор вывода  array[index], который мы еще не знаем, т. к.
C++ станем изучать позже */
}
```

Как и ранее, программа должна указать прототипы функций для требуемых типов:

```
void show_array(int *, int);
void show_array(float *, unsigned);
```

C/C++ Microsoft Visual

ГЛАВА 5

Функции для работы с символьными строками

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

Основные стандартные строковые функции

Функция *sprintf()*

`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()*

`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 С 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()*

`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`, то функция возвращает ноль; если `s1<s2` — возвращает отрицательное число; если `s1>s2` — возвращает положительное число.

Это происходит оттого, что функция сравнивает коды символов. 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 значений) таблицы. Вторая половина таблицы (остальные 128 позиций) отдана под национальные кодировки, которые, в общем случае, не упорядочены, это касается и кириллицы тоже. Данный момент надо учитывать при сравнении символьных строк с помощью `strcmp()`.

В теле функции коды строки `s1` посимвольно сравниваются с кодами строки `s2` посредством вычитания, как обычные числа (а коды и есть числа). Такая обработка символов происходит до первого несовпадения, 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 символы, введенные в разных регистрах (т. е. большие и маленькие буквы), различаются. Это и понятно: у них в таблице ASCII разные коды.

Функция *strcmpi()*

`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()*

`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()*

`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

```
// 5.1_2011.cpp

#include "stdafx.h"
#include <string.h>
#include <stdio.h>      //для getchar(),putchar(),printf()
#include <conio.h>      //для _getch()
#include <string.h>     //для strcpy(),..
#include <stdlib.h>     //atoi(),atof()
using namespace System;
#define eof -1          //Ctrl+z
#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*/

 int 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 С

  //----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 <Ctrl+z> >");
  }
  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 <Ctrl+z> >");
  }
 _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("string1 = string2\n");
      if(i>0)
        printf("string1 > string2\n");
      if(i<0)
        printf("string1 < string2\n");
      printf("continue <Enter>, exit <Ctrl+z> >");
    }
    _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 <Ctrl+z> >");
     }
      _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 формат `int` с помощью функции `atoi(s)`, которая преобразует строковые данные в целое число и возвращает это число. Если в строке не числовые данные, то функция возвратит ноль.

Другая функция, использованная нами при переводе строковых данных в формат `float`, — это функция `atof(s)`. Она возвращает число в формате `float`, преобразуя свою входную строку. И также возвращает ноль, если в строке не числовые 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 внедрить 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 продолже-

ние 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 немного, что сделала функция `sprintf()`.

Для ее проверки мы ввели два числа: одно в формате `int`, другое в формате `float` и строку (в формате `s`), чтобы показать, что `sprintf()` их обработает по форматам (в управляющей строке функции мы задали эти форматы) и соберет в единую строку, включив в нее и символы, которые находились между полями, задающими форматы (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 несравнившимися кодами символов. Попробуйте определить, какие символы первыми не сравнились, найдите их коды: это можно сделать, воспользовавшись стандартной функцией `char(имя символа)`, которая возвращает код символа, указанного у нее в аргументе (например, `int a=char('a')`).

Результат работы проверочной программы приведен на рис. 5.1.

```
Enter int n for sprintf()...>7
Enter float m for sprintf() >6.
Enter string for sprintf(). >12345
v=7  6.000000  12345
continue <Enter>, exit <Ctrl+z> >^Z



Enter string for strcpy() >
12345
Copied string=12345
Continue <Enter>, exit <Ctrl+z> >^Z



Enter string1 for strcmp() >123
Enter string2 for strcmp() >23
strcmp's value=-1
string1's length=3
 string1 < string2
continue <Enter>, exit <Ctrl+z> >^Z



Enter string1 for strcat() >123
Enter string2 for strcat() >45
strcat's value=12345
continue <Enter>, exit <Ctrl+z> >^Z_
```

Рис. 5.1. Результат работы программы листинга 5.1

C/C++ Microsoft Visual

ГЛАВА 6

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

Новые типы переменных

Мы уже знаем, что в языке С наряду с рассмотренными типами переменных (`int`, `char`, `float`) существуют и другие типы данных, сведения о которых необходимо уточнить:

- `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, объявленной с таким типом данных, будет находиться число (которое, естественно, будет изображено 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 (т. е. 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 — стал его неотъемлемой частью (в этом и смысл квалификатора `unsigned`: он подавляет знак в числе, т. е. число становится числом без знака. Но, как известно, знак числа находится в старшем разряде ячейки, в которой находится число, и значение этого знакового разряда физически никуда не девается, когда число получает квалификатор `unsigned`. Поэтому, если число положительное, т. е. в его знаковом разряде стоит ноль, то квалификатор `unsigned` не меняет значения числа, т. к. этот ноль не повлияет на величину числа (вспомните, что любое число разлагается по степеням, например, двойки, и если знаковый разряд находится в разряде n, то слагаемое разложения будет иметь вид $0 * 2^{n-1} = 0$). Если же число отрицательное, то в его знаковом разряде будет стоять единица, а величина слагаемого в разложении числа по степеням двойки станет уже равной $1 * 2^{n-1} = 2^{n-1}$. Вот на эту величину и изменится отрицательное число, если к нему применить атрибут `unsigned`;

- `unsigned int` — аналогичен типу `unsigned char`;
- `unsigned long` — длинное целое без знака. Последствия того, что переменная имеет тип "без знака", для отрицательных чисел уже рассмотрены.

Кроме того, существуют также следующие типы данных: `enum`; `bool`.

`enum` — так называемый перечислимый тип данных. Он позволяет задавать мнемонические значения для множеств целых значений (т. е. обозначать данные в соответствии с их смыслом). Допустим, нам надо работать в программе с наименованиями 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 английском варианте первым днем считается как раз воскресенье). Но бывают множества, как говорят, и "покруче", чем дни недели, элементы которых не упомнишь. Поэтому 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. Все, что в фигурных скобках, — это заданные заранее значения переменной `days`. Сама переменная `days` задает как бы шаблон, который самостоятельно применять нельзя: это обычный тип данных. Следует объявить дополнительно другую переменную этого типа: 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}; //ввели тип
days anyday,otherdays; //объявили 2 переменные данного типа
```

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,... (т. е. 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, Vol3=9,Vol4,Vol5,Vol6} pitchers;
```

У первого, второго и третьего элементов этого множества числовые значения соответственно равны 5, 7 и 9. Остальным элементам, неопределенным нами, компилятор присвоит последовательные (через единицу) значения, начиная со значения последнего определенного элемента (т. е. 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 (типы). 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. Эти так называемые *литералы* (постоянные символьные значения) сами выступают в роли переменных — им можно присваивать значения (соответственно, нуль и единицу).

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. В листинге 6.2 приведен пример программы с использованием булевых переменных и булевой функции. Поставьте с помощью отладчика точку останова на первом операторе и двигайтесь по программе пошагово, каждый раз нажимая клавишу <F10>. Перед каждым следующим шагом проверьте содержимое переменных. Понаблюдайте, как изменились 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): целые, вещественные, перечислимые и символьные.

Вещественные константы — это, в общем случае, нецелые числа, которые представлены в виде чисел с плавающей точкой.

Перечислимые константы — это значения элементов перечислимого множества (значения такого типа постоянны).

Целые константы. Например:

```
int i = 12;
```

Плавающая точка может определяться так:

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

или

```
r=0.15;
```

Обе записи равноценны.

Целые числа, помимо десятичных, могут быть также восьмеричными и шестнадцатеричными. Первые пишутся как `int i = 0-1;` (записано число –1 в 8-ричной системе, т. е. с нулем впереди), а вторые как `int i = 0xa;` (т. е. с `0x` впереди).

Среди констант *символьного* 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 нуль 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 таблицы ASCII с 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).

Второй вид записи — это так называемые ESC-последовательности. С их помощью можно записывать не только управляющие, но и любые символы. В этом случае в качестве элементов последовательности выступят сами коды символов. Например, код символа 0 по таблице ASCII равен 48. В виде ESC-последовательности его можно записать как `char v='\060';` (48 = 060).

Новые операции

Поговорим об операциях, с которыми еще не встречались в рассмотренных ранее примерах.

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 булевого типа, т. е. либо ложен, либо истинен. Поэтому мы можем, например, писать:

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

В данном примере, кстати, `a=0`, т. к. `i<j` и результат равен `false`.

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, поскольку выражение в операторе `while` настолько громоздко, что вы не можете его вычислить в момент отладки программы. Тогда это выражение можно присвоить некоторой (объявляемой вами) булевой переменной. Теперь вы сможете в режиме отладчика увидеть значение этого выражения и принять соответствующие меры.

Надо учитывать, что операции отношения по очередности (приоритету) их выполнения младше арифметических операций. 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:

- `&&` (и);
- `||` (или).

Тип выражения, в котором участвуют эти операции, будет булевым, как и для операций отношения.

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 так строит 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 (т. е. воздействующая только на один операнд) операция `!` (не). Это логическая операция отрицания. Она преобразует операнд, значение которого не равно нулю или истине, в нуль, 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=0`, т. е. в записи по правилам C это будет выглядеть как `a==0`.

К унарным относятся также операции "одиночный плюс" и "одиночный минус", явно устанавливающие знаки чисел или значений переменных. Например:

```
int x = -1;
float z=12.5;
float y = -z;
```

Преобразование типов данных

Современные компиляторы многое берут на себя, а 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 заходит в тупик в очевидных ситуациях и недоуменно разводит руками: "Чего это оно не идет? Не понимаю...". А не понимает, потому что избалован возможностями современных компиляторов, при которых он родился и вырос. Но они его (если программист не очень грамотный) иногда подводят, и бывает очень сильно.

Итак, при составлении программ все-таки надо знать, что в выражениях обычно участвуют данные разных типов, как и в операциях присвоения, при которых левая часть имеет один тип, а правая другой. И чтобы как-то свести концы с концами, установлены соответствующие правила преобразований данных разных типов.

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, если 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. Отсюда 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>`. Такие преобразования называют *кастингом*.

Например, существует стандартная функция `malloc(число)`, которая выделяет указанное в ее аргументе количество байтов памяти и которая возвращает адрес выделенного участка. Но функция возвращает адрес "неопределенного типа" (ее тип — `void`), т. е. непонятно, данные какого типа мы можем размещать в выделенной области. Чтобы настроиться на выделенную область, в которой хотим, например, размещать данные типа `char`, мы должны неопределенный выход функции привести к нашему типу с помощью кастинга `(char*)malloc(число)` (об адресах будем говорить в следующих 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`, `b = 0`, потому что из `a = x && y` следует: т. к. `x≠0` и `y≠0`, то по определению операции `&&` результат будет истинен, 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: `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `|`, `~`.

В листинге 6.3 приведен текст программы, которая подсчитывает число ненулевых битов целого числа и использует операцию сдвига.

Листинг 6.3

```
// 6.1_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar()
#include <conio.h>      //для _getch()
using namespace System;

//---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();
}
//-----------------------------------------------------------
```

Рис. 6.1. Результат работы программы листинга 6.3

Результат работы этой программы показан на рис. 6.1.

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 то, как работает операция `&`, можно сказать, что результатом вычисления выражения `(n & 01)` будет значение последнего бита числа `n`.

Действительно:

- если в последнем бите числа `n` — нуль, то выражение `(n & 01)` даст тоже нуль;
- 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 нуль. Затем там же, в заголовочной части `for`, вычисляется выражение, определяющее условие продолжения/завершения цикла `(n !=0)`. Следовательно, 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 нуль, 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 такой 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 (т. е. 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;.
```

Эти выражения, как обычные выражения, можно помещать в качестве аргументов функции `printf()` и т. д.

Операторы и блоки

Если за любым выражением (например, `x=0`, `i++` или `printf(...)`) стоит точка с запятой, то такое выражение в языке С 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 (компилятор определяет конец блока по его закрывающей скобке).

Исходя из этого определения, можно записать формат задания операторов `for`, `while`, `if` таким образом:

```
for(выражение)        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:

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

Работает эта конструкция так:

1. Вычисляется выражение в скобках оператора `if`.
2. Если значение выражения истинно, то выполняется тело `if` (блок операторов).
3. Если значение выражения ложно, то 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, то применяют конструкцию вида:

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

Работает эта конструкция так:

1. Последовательно вычисляется выражение в каждой строке.
2. Если выражение истинно, то выполняется тело (т. е. блок операторов) и происходит выход из конструкции на выполнение следующего за ней оператора.
3. Если выражение ложно, то 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.

Пусть даны массив `v[n]` и число `x`. Надо определить, принадлежит ли `x` массиву. Так как элементы массива предполагаются упорядоченными по возрастанию их значений, то поиск проведем, применяя метод половинного деления (иначе называемый двоичным поиском).

Суть метода такова:

1. 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)`.

2. Этот отрезок делится пополам.
3. Средняя точка отрезка вычисляется как `j = (0 + (n-1))/2`.
4. 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 половинок отрезка, в которой, возможно, содержится `x` (`x` может и не содержаться в массиве, т. е. не совпадать ни с одним из элементов массива), затем делить половину отрезка пополам и проверять, как и 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 произойдет обязательно, потому что мы станем делить отрезок пополам и в качестве результата брать целую часть от деления, поскольку, как мы видели, 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` (т. е. 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[]`, то 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

```
// 6.2_2011.cpp
#include "stdafx.h"
#include <stdio.h>      //для getchar(),...
#include <conio.h>      //для _getch()
#include <stdlib.h>     //для atoi()
using namespace System;

#define eof -1         //Ctrl+z
#define maxline 100

//-------Ввод строки с клавиатуры----------
int 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 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);       //нашли
    } //while
  return(-1); //не нашли
 }

//------------------------------
int 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-Enter, exit-Ctrl+z\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 данной переменной присваивается значение середины, в результате чего 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: равно ли возвращенное ею значение –1.

Все эти операторы помещены в блок — тело оператора `do...while` и выполняются в цикле, пока не будет нажата комбинация клавиш <Ctrl>+<z>. Такая структура

Рис. 6.2. Результат работы программы листинга 6.4

дает возможность вводить разные значения переменной `x`. Результат работы программы приведен на рис. 6.2.

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, результат ее работы — на рис. 6.3.

Листинг 6.5

```
// 6.3_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar()....
#include <conio.h>      //для _getch()
using namespace System;

#define eof -1          //Ctrl+z
#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) */
int 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;
       case '\012':     //чтобы не учитывался символ <Enter>
               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;
   }
  return(0);
 }
//-------------------------------
int main()
{
  int c,i,a[m];
  char s[m];
  for(i=0; i < m; i++)
    a[i]=0;
  printf("Enter your characters,<Enter>,<Ctrl+z>,<Enter> >");
  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()
```

Рис. 6.3. Результат работы программы листинга 6.5

Использование оператора `switch` демонстрируется посредством 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:

1. Анализируется выражение заголовочной части.
2. Управление передается на выполнение того участка тела, значение которого в метке 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 выполнение оператора `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, если 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, а 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`, куда будут записываться сами символы, чтобы потом можно было их вывести на экран. Все прочие символы, не попадающие в предложение `case`, отмечаются в массиве `s` символом `!`. Когда встретится символ <Ctrl>+<z>, 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`). Оператором `continue` удобно пользоваться в тех случаях, когда при выполнении тела цикла ясно, что не следует продолжать выполнение операторов дальше, а надо возвращаться на новый виток цикла. Например,

имеем массив целых чисел `int A[n]`. Требуется выбрать из него только положительные числа и обработать их. 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, то 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 самому автору трудно было 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.

C/C++ Microsoft Visual

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> * <имя указателя>
```

Здесь:

- `<тип переменной>` — это тип той переменной, на которую указывает указатель, т. е. тип переменной, чей адрес может находиться в переменной, которую мы задаем как указатель;
- `<имя указателя>` — это 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 конкретный 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 `нуль`, 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;
```

сообщает 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. То есть указатель при увеличении его значения на единицу передвигается на величину, равную длине элемента объекта, устанавливаясь на начало следующего элемента объекта.

Мы только что разобрали понятие указателя в общем смысле. Однако в VC++ это понятие несколько расширено, что вполне естественно для развивающихся систем.

Здесь рассматриваются указатели трех типов:

- регулируемые указатели;
- нерегулируемые указатели;
- нерегулируемые указатели функций.

Чтобы понять суть сказанного, вспомним, что ранее упоминалась среда CLR — это менеджер, который управляет исполнением кода, памятью, потоками и работой с удаленными компьютерами, при этом строго обеспечивая безопасность и создавая надежность исполнения кода.

CLR является добавкой-расширением C++, введенной фирмой Microsoft, начиная с версии VC++ 2005. Подключение к вашему проекту этого менеджера осуществляется на этапе компиляции. Если вы посмотрите свойства вашего проекта с помощью опции **Project | Properties** и в открывшемся диалоговом окне выберете папку **Configuration Properties | General**, то в правой части окна увидите настраиваемое свойство **Common Language Runtime support** (поддержка режима CLR).

Среда версии 2011 в режиме CLR понимает программы, использующие "родные" указатели, обозначаемые символом "*" (это видно из примера листинга 7.2, расположенного ниже по тексту), хотя эта же среда предусматривает совсем иное обозначение указателя, что мы увидим далее.

Без режима CLR, когда вы в своей программе работаете с некоторыми объектами (здесь нам приходится забегать вперед, ибо объекты — это предмет более позднего изучения, когда мы станем знакомиться с классами), вы должны сами заботиться об их размещении в памяти, выделяемой средой. Память для вашего приложения выделяется в так называемой "куче" (heap): в ней вы размещаете свои объекты, там же сами должны освобождать память, когда перестаете работать с объектом, иначе куча может переполниться, и процесс выполнения приложения прервется. Это так называемая *неуправляемая куча*. Указатели на участки памяти в такой куче обозначаются символом "*" (т. е. выше мы как раз рассматривали указатели для неуправляемой памяти). Вот два последних типа указателя из приведенного выше перечня типов указателей и являются традиционными указателями C/C++ на объекты в нерегулируемом объеме памяти, выделяемой для исполнения приложения.

Другое дело, когда включается режим CLR. Такое приложение отличается от обычного тем, что его заготовка обеспечивает подключение к приложению специального системного пространства `System`, содержащего объекты, размещение в памяти которых надо автоматически регулировать. Если вы внимательно смотрели тексты ранее изученных нами программ, то в каждой из них имеется строка `using namespace System`, что свидетельствует о том, что программа работает в режиме CLR (а мы и выбирали такой шаблон-заготовку при создании проекта программы). Так вот: режим CLR работает уже с *управляемой кучей памяти*, в которой размещение объектов и ее освобождение от них происходит под управлением среды. Такой сервис входит, например, в язык Java, где не надо делить кучу на управляемую и неуправляемую.

Регулируемый указатель — это тип указателя, который ссылается на объекты (адреса памяти, по которым можно обращаться к объектам), расположенные в общей

регулируемой куче памяти, предоставленной приложению в момент его исполнения. Для таких указателей принято специальное обозначение: вместо символа "*" применяется символ "^".

Создание CLR привело к необходимости разработки аппарата преобразования переменных, относящихся к одной куче, в адреса в другой и т. п. Этот процесс назвали *маршаллизацией*. Существует специальная библиотека, обеспечивающая этот процесс. Таблица некоторых преобразований приведена в *главе 14*.

При рассмотрении работы с компонентами так называемых форм *(см. главы 10—12)* вы увидите, что регулируемые указатели создаются в обработчиках событий компонентов и что свойства проектов, работающих с объектами, содержат автоматическое подключение при компиляции поддержки режима CLR. В среде VC++ существует специальная утилита gcnew, которая формирует экземпляр какого-то объекта, выделяя ему (экземпляру) некоторую память, и возвращает ссылку на этот экземпляр (аналог функции `malloc()`).

В листинге 7.1 демонстрируется работа с объектом "Структура", суть которого мы опишем позже. Результат работы программы представлен на рис. 7.1.

Листинг 7.1

```
// 7.1_2011.cpp

#include "stdafx.h"
ref struct Message
{
 System::String ^sender, ^receiver, ^data;
};
using namespace System;

int main()
{
    Message ^M= gcnew Message ;
    M->sender="The message to all";
    M->data="11.03.2012";
    Console::WriteLine(M->sender);
    Console::WriteLine(M->data);
    Console::WriteLine(L"Hello World");
    Console::ReadLine(); //для задержки экрана
}
```

Здесь объявлена некая структура `Message` ссылочного (`ref`) типа. Элементами ее являются указатели-ссылки `^sender`, `^receiver`, `^data` на объект типа `String` (строковые данные). В головной программе `main()` для структуры `Message` утилитой gcnew выделяется память и возвращается указатель `M` типа `Message` на эту структуру. Это аналог известной функции `malloc()`. Алгоритм программы интуитивно поня-

тен: по адресу, на который указывает указатель `sender`, пересылается сообщение "The message to all", по адресу, на который указывает указатель `data`, пересылается строка с датой, затем эти два текста выводятся на консольное устройство вывода (у нас в консольных приложениях консольное устройство вывода — экран). После этого туда же выводится сообщение "Hello World" и выполняется оператор чтения строки с клавиатуры для обеспечения задержки убегания экрана.

Рис. 7.1. Результат работы программы листинга 7.1

Шаблон CLR, когда мы хотим использовать регулируемые указатели, предполагает подключение к тексту программы специального пространства `System`, содержащего классы, которые задают ссылочные типы данных и функции работы с ними. В частности, функция `WriteLine()` (вывод на печать) принадлежит этому же пространству. А нам как раз и надо работать с такими данными.

Указатели и массивы

Продолжим рассмотрение нерегулируемых указателей. 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 ранее, `Pa+i` будет указывать на `i`-й элемент массива, т. е. можно достать такой элемент из массива путем выполнения оператора:

```
int a=*(Pa+i);
```

Но по определению массива мы можем записать:

```
int a=A[i];
```

Мы говорили ранее, что массив элементов строится в языке С так, что его имя — это адрес первого элемента массива, в нашем случае `A=&A[0]` и `Pa=&A[0]`.

Следовательно:

```
Pa=A
Pa+i = A+i
*(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]` (компилятор обрабатывает эту конструкцию), т. е. обращаться к элементам массива можно через *индексированный указатель*.

Пример программы, демонстрирующей вышесказанное, приводится в листинге 7.2 (все пояснения даны по тексту программы), результат работы программы — на рис. 7.2. Заметим, что программа строится в среде CLR, хотя содержит нерегулируемые указатели (мы раньше говорили о том, что среда при работе в режиме CLR узнает "родные указатели").

Листинг 7.2

```
// 7.2_2011.cpp

#include "stdafx.h"
#include <stdio.h>
#include <conio.h> //_getch()
#include <stdlib.h> //atoi()
using namespace System;
#define maxline 1000
#define eof   -1

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

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

int main()
{
 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]);
  }
  while((getline(s,maxline) != eof)); /*для обеспечения цикличности ввода
номеров элементов: признак конца цикла ввода — Ctrl+z*/
 _getch();      //задержка изображения на экране
}
```

Рис. 7.2. Результат работы программы листинга 7.2

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`. Это 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`.

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. Приме-

ром этого может служить программа, текст которой приводится в листинге 7.3 (результат работы программы — на рис. 7.3).

Листинг 7.3

```
// 7.3_2011.cpp

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
using namespace System;

/*функция, меняющая местами значения переменных: значение, которое было в
переменной "a", переместится в "b" и наоборот*/

int f(int *a, int *b)  //параметры - указатели
 {
   int i=*a;
   *a=*b;
   *b=i;
   return(0);
 }

int main()
{
  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);  //передача адресов переменных
  printf("The (c,d)'s value after  function application: c=%d d=%d\n",c,d);
  _getch();
}
```

Рис. 7.3. Результат работы программы из листинга 7.3

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.

Указатели символов и функций

Символьная константа или массив символов в языке С — 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 байтов 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, рассмотрим функцию `char *strsave(s)`. Приведем в качестве примера программу, работающую с указателями символьного типа (листинг 7.4). Программа содержит функции работы с символьными строками, в которых отражена работа с указателями.

Листинг 7.4

```
// 7.4_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar()
#include <conio.h>
#include <stdlib.h>    //для atoi()
#include <string.h>
#include <malloc.h>     //для malloc()
using namespace System;
```

```
#define maxline 1000
#define eof  -1        //Ctrl+z

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

int 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);  //копирует строку в кучу
```

```
   return(p);  //возвращается указатель на строку, помещенную в кучу
   /* 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 его заголовочной части 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 т. д., пока не будет переслан последний символ: `'\0'`. Когда это произойдет, результат выражения станет равным нулю, и функция завершится. Функция не возвращает никакого значения (ее 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 ранее) 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 перешлется, в теле `while` произойдет приращение значений указателей на единицу, которое позволит обратиться к следующему элементу объекта. Программа снова возвратится в заголовочную часть `while`, обеспечивающую пересылку второго элемента в область, на которую указывает `s` (и т. д., 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 извлекать сохраненную в нем строку.

Копирование производится 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 (т. е. действительно ли выделено место в памяти). Это существенный вопрос, т. к. свободной памяти в данный момент могло не оказаться или мог произойти какой-либо сбой. 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 приведен на рис. 7.4.

Рис. 7.4. Результат выполнения программы листинга 7.4

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 них 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, 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 нужно обратиться 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 путем 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 C возможно определять указатели на функции и, следовательно, обрабатывать указатели, передавать их в качестве аргумента другим функциям 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 в качестве своих параметров имеет две функции: ввода строки символов и подсчета количества ненулевых битов в целом числе (листинг 7.5).

Листинг 7.5

```
// 7.5_2011.cpp

#include "stdafx.h"
#include <stdio.h>      //для getchar(),putchar()
#include <conio.h>
#include <stdlib.h>  //для atoi()
using namespace System;

#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

int 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

 int gener(int (*getline)(char s[],int lim),int (*bitcount)(unsigned int n))
  {
       char s[maxline];
       int lim=100;
       printf("Enter any number >\n");
    (*getline)(s,lim);
      unsigned int n=atoi(s);
      n=(*bitcount)(n);
      return(n);
  }
//-------------------------------------------------------------

int main()
{
  int n=gener(getline,bitcount);
  printf("The amount of ones in the input n=%o\n",n);
  _getch();
        return 0;
}
```

Мы уже знакомы с функциями `getline()` и `bitcount()` (последнюю составляли, когда изучали операции сдвига — она подсчитывает в целом числе 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. 7.5.

Рис. 7.5. Результат расчета программы листинга 7.5

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 являющиеся 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];  //имя
   char  phone_number[80];  //телефон
   int  age;  //возраст
   int height; //рост
 };
```

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. Чтобы начинающему изучать C было более понятно, скажем, что объявленный шаблон — это тип "структура". А имея

такой тип, станем объявлять данные этого типа. Точно так же, как имея, например, тип `int`, начинаем объявлять данные типа `int`.

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`. Можно сказать так: `friends` — это переменная типа `man`), другой — обычный экземпляр по шаблону `man`. В языке C++ ключевое слово `struct` можно опускать, т. е. в C++ уже пришли к аналогии с простыми типами данных: там экземпляр структуры уже можно объявить так:

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

  Видите сходство с объявлением простых переменных? И `friends` и `others` объявлены как переменные типа `man`;

- 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. Этот квалификатор позволяет вводить в объявления имена-синонимы, более удобные в использовании в программе. Например, можно записать объявление некой переменной в виде `typedef int step;` и далее в программе при объявлении какой-то переменной писать вместо `int` ее синоним `step: step aa;`. Компилятор потом все вернет на свои места. По этому же принципу можно сократить работу со структурой, объявив, например,

  ```
  typedef struct
  {
     char phone_number[80];
     int age;
     int height;
  }aa;
  ```

  То есть заменили всю структуру на один тип `aa`.

  Теперь можно писать: `aa d1,d2[20],*p;`. Здесь объявлено три переменных типа `aa`: 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 структуры: тип, имя, точка с запятой.

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 ее элементами можно будет работать через указатель.

Обращение к элементам структур

Мы видели, что после объявления структуры (а это фактически тип данного, имя которого равно имени структуры) можно объявить некую переменную типа этой структуры или указатель этого типа (указатель на эту структуру).

Если вы объявили переменную типа структуры, то чтобы обратиться к элементам структуры, надо после имени переменной поставить точку, а если вы объявили указатель на структуру, то после имени указателя на данную структуру надо поставить стрелку вправо (->). Затем нужно 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 С 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 С++ функция может быть членом структуры. Дело в том, что в С++ структуры рассматриваются как класс, т. е. для членов структуры могут быть определены спецификации доступа к памяти, определяемые для членов класса: `public` (всегда определена по умолчанию), `protected` и `private` (их мы рассмотрим при изучении классов).

  В листинге 7.6 приведен текст простейшей программы, в которой функция является членом структуры (результат работы этой программы показан на рис. 7.6);

Листинг 7.6

```
// 7.6_2011.cpp

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
using namespace System;

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

int main()
{
  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;
     a=bbb.f(15);
     printf("a=%d\n",a);
   _getch();
}
```

Рис. 7.6. Результат работы программы из листинга 7.6

- 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
  };
```

- присваивать значения одной структуры другой 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 (функция сама ничего не возвратит: мы сами должны этого добиться с помощью C, который это позволяет сделать). 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

В листинге 7.7 приведен пример программы, в которой функция возвращает структуру.

Листинг 7.7

```
// 7.7_2011.cpp

#include "stdafx.h"
#include <stdio.h>   //для getchar(),putchar()
#include <conio.h>
#include <stdlib.h>  //для atoi()
#include <string.h>
#include <malloc.h>  //для malloc()
using namespace System;
#define eof  -1
#define maxline 1000
```

```
//-------Ввод строки с клавиатуры

int 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 должна быть задана не больше, 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;

       if(cond < 0)
        {
          high=mid — 1;  continue;
        }
       if(cond > 0)
         {
           low=mid + 1;   continue;
         }
```

```
            tab[mid].keycount=0;
            return(tab[mid]);       //найдено
        } //while
         tab[mid].keycount=-1;
         return(tab[mid]); //не найдено
 }
//-------------------------------
int 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 getline(), 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()
```

Рассмотрим функцию `BinaryInStruc`, возвращающую 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 применяется 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 уже рассмотренной функции `strcmp(word,tab[mid].keyword)`. Если в средней точке строки в переменной `word` и в переменной `tab[mid]`

значения `keyword` совпадают, то 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()`, возвращающей 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 показан на рис. 7.7.

Рис. 7.7. Результат работы программы листинга 7.7

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 7.8).

Листинг 7.8

```
// 7.8_2011.cpp

#include "stdafx.h"
#include <stdio.h>    //для getchar(),putchar()
#include <conio.h>
#include <stdlib.h>   //для atoi()
#include <string.h>
#include <malloc.h>   //для malloc()
using namespace System;
#define eof -1
#define maxline 1000

//-------Ввод строки с клавиатуры

int 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 должна быть задана не больше, чем количество
инициализированных элементов массива.
Возвращает указатель на структуру типа key, в которой находится
слово, заданное в word, либо возвращает NULL (сигнал 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 деления "/" 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*/
  continue;
 }
```

```
if(cond > 0)
 {
       low=mid + 1;  continue;
 }
return(mid);  /*найдено (возврат указателя на найденный элемент таблицы, т. е.
на структуру)*/
} //while
  return(NULL); //не найдено
 }
//-------------------------------
int 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 и показан на рис. 7.8.

Рис. 7.8. Результат работы программы листинга 7.8

Программа упрощенного расчета заработной платы одному работнику

В этой программе создана функция расчета, которой передается в качестве параметра указатель на структуру (листинг 7.9). Результат расчета показан на рис. 7.9.

Листинг 7.9

```
// 7.9_2011.cpp
//

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>  //для atoi()
using namespace System;

#define eof -1
#define maxline 1000

//-------Ввод строки с клавиатуры

int 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));
 }
//-------------------------------
int 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()
```

Рис. 7.9. Расчет зарплаты одному работнику

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, подсчитывающей 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). Из 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 ранее. 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 нуль), 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 самом узле, а в правом подузле всегда будет находиться слово большее, чем в в самом узле. 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 7.10. На рис. 7.10 отражен результат выполнения этой программы.

Листинг 7.10

```
// 7.10_2011.cpp

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <string.h>   //strcpy()
#include <malloc.h>
using namespace System;
```

```
#define maxline 1000
#define eof -1
int fix=0; //глобальная для печати дерева
//-------Ввод строки с клавиатуры
int 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 нуль */
struct tnod *right; /*правый потомок: здесь хранится указатель на правый
подузел, т. е. на структуру, в которой хранится слово, большее данного.
Если слов больших данного нет, то здесь хранится нуль */
 };

char *strsave(char *s) // C++ позволяет объявлять 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));
     /*т. к. malloc() выдает указатель типа void, то принудительно
       приводим его к типу char, чтобы согласовать с p*/

  if((p != NULL))
   strcpy(p,s);
   return(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*/
  }
//----------------------------
int TreePrint(tnod *p)
       //left_right=1 при выводе левого узла
       //left_right=2 при выводе правого узла
/*Эта функция печатает или выводит на экран все дерево, рекурсивно себя
вызывая*/
 {
  if(p != NULL) //p всегда указывает на начало дерева
                //p!=NULL — дерево существует
   {

          /*Left = ссылка на левый подузел, right — на правый. Эти ссылки могут
быть нулевыми: дерево дальше не идет*/
     printf("Count word repetition=%d word's value=%s\n",p->count,p->word);
     TreePrint(p->left); //выводит левый узел:
     TreePrint(p->right);  //выводит правый узел
    }

return(0);
 }
//-------------------------------------------------------------

#pragma argsused
int main()
{
 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 нуль, 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();
}
```

Физическая структура дерева приведена на рис. 7.11.

Рис. 7.10. Результат работы программы листинга 7.10

Рис. 7.11. Физическая структура дерева, построенного программой листинга 7.10

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. 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 — двубитовое. 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) ...
```

Категории памяти

Каждая объявленная переменная относится к той или иной категории памяти. Ранее мы видели, что переменные бывают глобальные или локальные. Принадлежность к одной из этих категорий определяет время жизни переменной. Действительно, локальные (или как их еще называют, автоматические) переменные "живут", пока они находятся в теле функции, оператора или блока. По выходе из этих конструкций переменная уничтожается: ее память становится доступной к перераспределению для других переменных.

Глобальные (их еще называют статическими) переменные "живут", пока работает ваша программа. *Все функции относятся к глобальным переменным.* Язык С, кроме того, имеет ряд спецификаторов категорий памяти, которые явно указывают на это компилятору. Мы уже встречались с некоторыми из них — это спецификаторы `extern` и `static`. Первый применяется при определении так называемых внешних переменных (они определяются вне вашей программы, например, в подключаемом к ней h-файле). Внешние переменные тоже являются глобальными. Второй спецификатор позволяет "глобализовать" локальную переменную: при выходе из локализующей ее конструкции такая переменная не теряет своего значения, как не теряет его она и при входе в свою конструкцию.

К другим спецификаторам категории памяти относятся спецификаторы: `auto`, `register`, `typedef`. В объявлении переменной вы можете использовать только один спецификатор категории памяти. Если категория памяти в объявлении переменной не указана, то категория принимается равной `auto`, и объявление переменной внутри блока, функции или оператора создает автоматические переменные. Переменные, объявленные с категорией `register`, будут также отнесены к автоматическим (т. е. локальным). Если переменной при ее объявлении присваивается категория `register`, то это означает, что при промежуточных операциях с такой переменной ее содержимое станет размещаться не в оперативной памяти, а на регистрах компьютера, что значительно увеличивает скорость работы с такой переменной.

Спецификатор `typedef` фактически вводит синоним типа, давая возможность программисту вводить типы данных с подходящими ему наименованиями, чтобы улучшить читаемость программы. Вот структура его спецификации:

```
typedef [заменяемые типы: void, char, short, int, long, float, double, signed,
unsigned, struct, enum] новое имя;
```

Например:

```
typedef char Flag;
```

Здесь тип `char` заменен синонимом `Flag`. Теперь в вашей программе можно вместо `char` везде писать `Flag`. Например, `Flag b='b';`. Если же вы используете имя переменной, совпадающее с синонимом, определенным по `typedef`, такая переменная, как обычно, должна иметь тип данного. Например: `int Flag=12;`.

Другой пример:

```
typedef struct club
{
    char name[30];
    int size, year;
} GROUP;
```

Здесь вся структура заменена синонимом `GROUP`. Теперь переменные типа структуры `club` можно объявлять переменными типа `GROUP`:

```
GROUP a,b,*c;
a.size=123;
b.year=2007;
c= (GROUP *) malloc(sizeof(GROUP)); //выделили память под структуру
                                    //(инициализировали указатель)
c->name[0]='a';
c->name[1]='b';
//Или:
enum days {sun, mon, tues, wed, thur, fri, sat}; //ввели тип
typedef enum days d;
d dd;
dd=sat;
//Или:
typedef char * cc;  //замена указателя
cc str;
str="123";
int i=atoi(str);
```

C/C++ Microsoft Visual

ГЛАВА 8

Классы в C++. Объектно-ориентированное программирование

Концепция объектно-ориентированного программирования (ООП) включает в себя понятия объектов, классов, инкапсуляции и наследования.

Объект — это некая математически-программно описанная сущность, элемент окружающего нас мира, с которым мы встречаемся в повседневной жизни. Например, ваша конкретная собака, ваш конкретный телевизор, ваш конкретный автомобиль — это все объекты. Реальные объекты имеют две характеристики: состояние, которое определяется набором свойств объекта, и поведение. Например, собака имеет состояние, определяемое следующим набором ее свойств: имя, цвет шерсти, порода, характер и т. д. А ее поведение определяется тем, что она в данный момент может лаять, вилять хвостом и т. д. Поведение объекта определяется набором функций, заданных в объекте, которые здесь называются методами.

Класс — это некий чертеж, некий проект, из которого создается объект. По аналогии с простыми переменными, рассмотренными ранее, класс — это некий тип данных, некий шаблон, из которого затем формируется конкретный объект. Если видели, что имеется тип `int` для описания переменных, то задав `int A;` мы создаем переменную (можем назвать ее объектом для общности) `A`. Написав `A=0;` мы этот объект инициализируем. С классами происходит точно такая же история, как мы в дальнейшем увидим. Класс — это дальнейшее обобщение понятия объекта программирования, с которым приходится работать. В классе заложены свойства и поведение будущего объекта, который получается из класса как из проекта. Например, "Автомобили" — это класс. "Тойота" — это объект класса "Автомобили", конкретное воплощение класса в конкретную модель. Или, например, проект дома серии 135 — это класс, а сам конкретный дом, построенный по конкретному адресу, — это объект класса домов серии 135. Таким образом, когда мы смотрим на окружающие нас объекты реального мира в плане их состояния и поведения, то готовы к пониманию объектно-ориентированного программирования.

Объектно-ориентированное программирование — это способ программирования с ориентацией на объекты. При таком способе создаются крупные программные образования — классы, куда закладываются общие свойства будущих объектов,

которые станут получаться по определенным правилам из этих классов, куда закладываются варианты поведения будущих объектов через создаваемые в классах параметрические программы (методы классов). Такой способ значительно ускоряет разработку программного обеспечения и облегчает труд программиста. Вспомним, что одним из первых средств автоматизации труда программиста были так называемые стандартные программы, которые выполняли часто встречающиеся действия (например, перевод десятичных чисел в двоичные, вычисление тригонометрических функций и т. п.). Такие программы объединялись в библиотеки стандартных программ. Сегодня на более высоком уровне мы имеем библиотеки классов, которые поставляют нашим программам необходимые им объекты (рисунки, фотографии, средства мультимедиа и т. п.), тем самым повышая качество и эффективность современного программирования. Например, у вас есть класс `Arr`, создающий массив, и в самом классе имеются методы для вычисления суммы и среднего значения хранимых в массиве чисел. Вам в вашей программе надо работать с неким массивом чисел. Не имея класса `Arr`, вам надо было бы объявить массив, проинициализировать его (наполнить его элементами), затем создать функции для вычисления суммы и среднего значения хранимых в массиве чисел, а уже после всего этого приступать к обработке массива и вычислению суммы его элементов и среднего значения этих элементов. Назавтра другой программист может столкнуться с точно такой же проблемой. А если воспользоваться уже готовым классом `Arr`, то работа с созданием и обработкой массива значительно сокращается: из класса создается объект (сам конкретный массив), а затем обращением к методам класса (примерно, как к элементам знакомых нам структур) вычисляются необходимые величины.

Программно в классах задаются элементы, называемые *свойствами*. Они описывают состояние объекта и хранятся в специальных элементах, называемых *полями*. Поведение же объекта описывается специальными функциями, которые в классах носят название *методов*. Когда свойствам класса присваиваются какие-то конкретные значения, то тем самым из класса создается конкретный объект. Объекты создаются специальным методом класса, называемым *конструктором*.

Методы обрабатывают внутренние состояния объекта и обеспечивают механизм взаимодействия между объектами. Например, возьмем класс велосипедов. Свойствами, характеризующими состояния объектов этого класса, будут: текущая скорость, текущее состояние переключателя педали (изменение скорости), количество шестеренок и текущая шестеренка, за счет которой скорость изменяется. Методами, которые изменяют состояние велосипеда, будут: смена шестеренки, переключение педали и изменение скорости. Весь этот механизм изменения свойств методами класса спрятан внутри самого класса. Такой принцип взаимодействия элементов класса носит название *инкапсуляции данных*. Это фундаментальный принцип объектно-ориентированного программирования.

В чем же фактическая польза от механизма классов-объектов?

Во-первых, обеспечивается модульность программирования: исходный код объекта написан и поддерживается независимо от исходных кодов других объектов (т. е. повышается надежность всей задачи, программа которой состоит из цепочки таких независимых модулей).

Во-вторых, механизм работы такого модуля скрыт внутри самого модуля и не отвлекает программиста на выяснение различных мелких деталей алгоритма.

В-третьих, имеется возможность многократного использования элемента (как и когда-то многократное использование библиотеки стандартных программ).

В-четвертых, обеспечивается легкая сменяемость элементов в общей программе (в приложении): если такой элемент выходит из строя, его можно легко заменить аналогичным элементом, не разрушая всю задачу.

Итак, ООП основано на использовании классов. Использование классов — это основное отличие языка C++ от языка C.

Классы

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

Класс — это обыкновенный тип. Если вы программист, то всегда имеете дело с типами и экземплярами, даже если и не используете эту терминологию. Например, вы создаете различные переменные типа `int`. Можно сказать, что вы фактически создаете различные экземпляры переменных этого типа. Классы обычно более сложны, чем простые типы данных, но они работают тем же способом. Создавая переменные типа заданного класса, вы создаете экземпляры этого класса, а назначая различные значения экземплярам того же типа (как и переменным типа `int`), вы можете выполнять разные задачи. Поэтому наша цель — научиться пользоваться классами для написания приложений, а создание самих 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;`, а затем уже работали с экземпляром `a`. Можно 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;`, после чего можно 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 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 древесно-волокнистые и т. д. 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 экземпляр класса и определяем значения его конкретных характеристик, то 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, освобождает память, которая может использоваться для других программных целей. А если вспомнить, что память — величина не беспредельная, то становится понятной и роль деструктора.

Принципы построения классов

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

Инкапсуляция

Инкапсуляция — это принцип объединения в единой конструкции и 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, получаемый из класса, от внешнего воздействия, что приводит к высокой надежности программ, использующих объекты. С другой стороны, классы используются так, как ранее использовались стандартные программы, только с еще большей эффективностью в самых разных приложениях, что значительно повышает производительность труда программиста. При добавлении новых характеристик классам программы, ранее использовавшие объекты, построенные из них, остаются без изменений.

В VC++ введено понятие *компонентов* — специальных классов, в которых объекты определяются такими характеристиками, как свойства, события и методы. Причем, в отличие от работы с обычными классами, при работе в VC++ возможно манипулировать видом и функциональным поведением компонентов и на стадии проектирования приложения, и в момент его выполнения. Например, в VC++ существует компонент "форма" (класс `Form`) и компонент "кнопка" (класс `Button`), у которых есть свои свойства, методы и события. Если при проектировании приложения

в форму поместить две кнопки, то с помощью задания двух разных значений свойствам кнопок `Text` (название кнопки) и `Visible` (значения `false` и `true` определяют видимость кнопки при исполнении приложения) вы получаете два экземпляра, которые ведут себя по-разному: первая кнопка при выполнении программы будет невидима на 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 кнопкой мыши), 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: один класс получается из другого, на основании другого получается третий и т. д. То есть речь идет о том, что и в классах существуют родители и дети, бабушки с дедушками, их внуки и т. д. Наследование предполагает, что все характеристики класса-родителя присваиваются классу-потомку. После этого потомку при необходимости добавляют новые характеристики. Иногда некоторые методы в классе-потомке, полученном от предков, *переопределяются*, т. е. наполняются новым содержанием.

Наследование используется не только при разработке классов, но и при проектировании приложения. Например, в VC++ есть класс `Label` (метка). Если поместить экземпляр этой метки на форму (экземпляр `Form`), то свойство "шрифт" метки примет значение свойства "шрифт" из экземпляра `Form`. Меняя параметры шрифта у родителя (`Form`), мы добиваемся изменения этих параметров у потомков (наследников). То есть метка автоматически будет наследовать это свойство от экземпляра `Form`, на который она помещена. Это же относится, например, и к классу `Button` (кнопка). Пример только что сказанного показан на рис. 8.1, из которого видно, как изменяется шрифт компонентов `Кнопка` и `Метка` в зависимости от изменения шрифта на форме.

Рис. 8.1. Наследование свойства "шрифт" потомками от родителя

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 VC++.

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 методам 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, т. е. его потомкам*/
 <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>
};   /*обратите внимание, что также заканчивается и объявление структуры*/
```

Глядя на то, как объявляется класс, вспомним пример из *разд. "Структуры. Объявление структур" 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), а вызывать методы на выполнение можно только в соответствии с тем, в какой секции находится функция. Атрибуты `private`, `public`, `protected` называются атрибутами доступа к членам класса. 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.

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 иметь возможность один и тот же метод в классе-родителе переопределить в классе-потомке. Например, все классы имеют общего прародителя — класс `Object`.

В этом классе определен метод `draw` (рисовать фигуру). Классы, рисующие различные фигуры и произошедшие от `Object` — родственные классы. Каждый из них определяет рисование своей фигуры методом `draw`, унаследованным от `Object`: точку, линию, прямоугольник, окружность и т. д. Но все фигуры разные, хотя метод общий. Но этот метод `draw` в каждом из классов-потомков переопределен, т. е. в каждом классе-потомке ему назначена другая функциональность.

Полиморфизм достигается за счет того, что методам из класса-родителя позволено выполняться в классе-потомке, а там оставляют только его имя, но при этом дают ему необходимую для данного класса функциональность. Такие методы должны объявляться в обоих классах с атрибутом `virtual`, записываемым перед атрибутом "возвращаемый тип данных". Если функция имеет атрибут `virtual`, то она может быть переопределена в классе-потомке, даже если количество и тип ее аргументов такие же, что и у функции базового класса. Переопределенная функция отменяет функцию базового класса.

Кроме атрибута `virtual`, у методов существует атрибут `friend`. Методы с таким атрибутом, расположенным (как и атрибут `virtual`) в объявлении метода перед указанием типа возвращаемых данных, называются *дружественными*. Метод, объявленный с атрибутом `friend`, имеет полный доступ к членам класса, расположенным в секциях `private` и `protected`, даже если этот метод — не член этого класса. Это справедливо и для классов: внешний класс (т. е. его методы) имеет полный доступ к классу, который объявляет этот внешний класс дружественным.

Во всех остальных аспектах дружественный метод — это обычный метод. Подобные методы из внешних классов, имея доступ к секциям `private` и `protected`, могут решать задачи, реализация которых с помощью методов-членов данного класса затруднительна или даже невозможна.

Примеры создания классов

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

Пример 1

Программа показана в листинге 8.2, результат работы программы — на рис. 8.2.

Как работает программа, видно из строк комментария в теле программы.

Листинг 8.2

```
//8.1_2011.cpp

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
using namespace System;
```

```
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) //метод класса
      {
       return(x-y);
      }
 };
 class B : A   //это объявляется класс, производный от A
  {            //при этом наследуются члены класса A
   public:
   int f2(int x)  //метод класса
    {
     A::x=20;     /* здесь могут использоваться члены-данные базового
                  класса из секции protected. Так как имя аргумента метода
                  f2() совпадает с именем поля x из класса A,
                  унаследованного классом B, то чтобы различить, какая
                  переменная к какому классу относится, потребовалось
                  уточнить с помощью спецификатора ::. Показано, что в
                  теле метода x берется тот, что унаследован от A, и
                  собственный аргумент x самого метода f2(): */
     return(x+A::x);  //возврат суммы x из B и x из A
    }
  };

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();
}
```

Обратите внимание, что когда вы создаете переменную типа класса, т. е. экземпляр класса (например, `A min;`, где `min` — экземпляр класса), то для этого экземпляра никакой памяти не выделяете, а компилятор все-таки размещает этот экземпляр где-то в памяти. Почему?

Рис. 8.2. Результат работы программы листинга 8.2

Среда исполнения приложений в C/C++ выделяет два вида памяти: статическую и динамическую. Последняя носит название "куча". Куча может быть управляемой и неуправляемой. Если компилятор способен определить размер памяти под объявленную переменную, то он выделяет для нее место в статической памяти (например, под переменную типа `int` он выделит 4 байта, а под массив типа `int` из 10 элементов он выделит 40 байтов).

Если же компилятор не в состоянии определить размер памяти под объявленную переменную, то он потребует от программиста поместить такую переменную в куче — в динамической памяти. Кстати, если программист сам хочет работать с динамической памятью (даже когда компилятор может поместить переменную в статической памяти), то язык это ему позволяет. Для простых переменных при этом используются: функция `malloc()`, которая выделяет область памяти в динамической области и возвращает указатель на эту область, и функция `free()`, которая освобождает занятую переменной область и передает освобожденную память в общее пользование.

Если же работа происходит с объектами, то здесь используются операторы `new` — аналог `malloc()`, и `delete` — аналог `free()`. Указанные функции и аналогичные им операторы работают с неуправляемой кучей. С управляемой кучей работает утилита gcnew, и нет необходимости освобождать самому память от объекта, поскольку в этом случае работает так называемая автоматическая сборка мусора: когда объект становится ненужным, память от него освобождается. Чтобы работать с такой кучей, надо перейти в режим CLR. Что касается вопроса "почему", то очевидно, что компилятор размещает переменную типа класса, объявленного нами, в статической области памяти. Но приведем пример этой же программы, в которой используется (по нашему желанию) динамическая память (мы, естественно, приводим только тело функции `main()`, где это отражается).

```
int main()
 {
  A *min =(A *) new A();  //создание экземпляров классов A, B
  B *max= (B *) new B();
  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();
```

```
 delete min;
 delete max;
}
```

В данном случае оператор `new` размещает объект в куче, выдает адрес этого объекта, а конструктор инициализирует объект.

Пример 2

Создадим класс, членами которого будут изделия, состоящие из деталей и их стоимостей, а также методы, первый из которых присваивает значения изделию, детали и стоимости через свои параметры, а второй выводит на экран значения, присвоенные первым методом. Текст программы приведен в листинге 8.3, результат работы программы представлен на рис. 8.3.

Листинг 8.3

```
//8.2_2011.cpp

#include "stdafx.h"
#include <stdio.h> //для printf()
#include <conio.h> //для _getch()

using namespace System;

class produce //начало определения класса
{
private:
//поля класса:
int modelnumber; //номер изделия
int partnumber;  //номер детали
float cost;      //стоимость детали

public:
//установка данных с помощью метода
//присваивает данным класса значения своих параметров
void setpart(int mn, int pn, float c)
{
  modelnumber = mn;
  partnumber = pn;
  cost = c;
}
void show() //вывод данных
{
 printf("The Number of the Model is  %d\n",modelnumber);
 printf("The Number of the Part is   %d\n",partnumber);
 printf("The Cost of the Part is    %.2f\n",cost);
}
}; //конец описания класса
```

```
//обработка класса в головной программе
int main()
{
 produce izd; //определение объекта из класса (экземпляр класса)
 izd.setpart(100, 200, 250.5); //вызов метода класса
 izd.show(); //вывод данных
 _getch();
}
```

Рис. 8.3. Результат работы программы листинга 8.3

Этот небольшой созданный нами класс позволяет выводить на экран характеристики изделия, описанного в нем.

Пример 3

Используем класс, созданный в примере 2, для создания нового класса — наследника класса из примера 2. Новый класс должен будет задавать дополнительную характеристику изделия — его форму. Пример программы приведен в листинге 8.4, результат работы программы — на рис. 8.4.

Листинг 8.4

```
// 8.3_2011.cpp

#include "stdafx.h"
#include <stdio.h> //для printf()
#include <conio.h> //для _getch()
using namespace System;

//детали изделия в качестве объектов (экземпляров класса)

class produce //начало определения класса
{
private:
int modelnumber; //номер изделия
int partnumber; //номер детали
float cost; //стоимость детали
public:
//установка данных
//функция-член класса
//присваивает данным класса значения своих параметров
```

```
void setpart(int mn, int pn, float c)
{
modelnumber = mn;
partnumber = pn;
cost = c;
}
void show() //вывод данных
{
printf("The Number of the Model is  %d\n",modelnumber);
printf("The Number of the Part is   %d\n",partnumber);
printf("The Cost of the Part is     %.2f\n",cost);
}
};

//объявление класса-наследника с новыми членами:
class MoreProduce : public produce
 {
 public:
    char *ProduceForm; //описание формы изделия
    void FormDecl(char *s)
   {
     ProduceForm=s;
   }
   void show1()
   {
       printf("The ProduceForm is  %s\n",ProduceForm);
   }
 };

//обработка класса в головной программе
int main()
{
 MoreProduce newizd;

 newizd.setpart(100,200,250.5);
 newizd.FormDecl("Square");
 newizd.show();
 newizd.show1();
 _getch();
}
```

Рис. 8.4. Результат работы программы листинга 8.4

Конструктор класса

Конструктор класса тоже член класса, но специфический — это метод с таким же именем, что и класс. Такие методы не имеют право возвращать какие-либо значения: если вы написали в конструкторе оператор `return`, компилятор выдаст ошибку. Конструктор выполняет различные задачи и не виден вам как программисту. Даже если вы сами не писали его текст, конструктор по умолчанию всегда присутствует в классе, ибо основная его задача — создавать в памяти экземпляр класса (т. е. как бы по проекту (классу) построить дом (экземпляр класса)). В этом случае конструктор берется из общего прародителя классов — класса `Object`, в котором он имеется.

Вторая задача конструктора — придавать всем членам-данным класса начальные значения — инициализировать класс. Если вы сами не построили конструктор, который станет инициализировать члены-данные вашего класса вашими же значениями, то этим данным невидимый для вас конструктор (конструктор по умолчанию) придаст свои, принятые в системе для данного типа величин значения (например, данные типа `int` получат нулевые значения и т. д.).

В листинге 8.5 приведен пример класса с конструктором, созданным для инициализации членов-данных класса теми значениями, которые задаст пользователь класса при работе с ним.

Листинг 8.5

```
// 8.4_2011.cpp
#include "stdafx.h"
#include <stdio.h> //для printf()
#include <conio.h> //для _getch()
using namespace System;
class Date
{
public:
/*Это члены-данные класса. С ними будут работать функции-члены класса: get... и
set...*/
   int month;
   int day;
   int year;

 Date( int dy, int mn, int yr )//конструктор класса
 {
/*члены-данные day, month, year будут принимать значения, которые поступят в
конструктор как в функцию при его использовании где-нибудь в приложении*/

  day=dy;
  month=mn;
  year=yr;
 }
```

```
//внешние функции класса:
 int getMonth() const;  /*это только прототип функции - информация для
                         компилятора*/

/*Функция getMonth() объявлена с атрибутом const — может данные только
поставлять, не изменяя их (read-only), а функция setMonth() не имеет
квалификатора const и поэтому данные внутри себя может изменять: виды функций
определены вне класса (см. ниже)*/

   void setMonth(int mn ); /*это только прототип функции — информация для
                         компилятора*/

// методы класса:
   int getDay();
   void setDay(int dy);
   int getYear();
   void setYear(int yr);

~Date()//Деструктор класса
  {
  }
};  //конец описания класса

/* создание функций (вне класса, поэтому надо указывать имя класса, для
которого эти функции создаются)*/

int Date::getMonth() const  //функция для класса Date
{
       return month;        //функция ничего не изменяет
}
//------------------
void Date::setMonth(int mn )
{
       month = mn;  //Функция изменяет член-данное класса
}
//-----------------------
int Date::getDay()
{
      return Date::day;
}
//---------------------------
void Date::setDay(int dy )
{
    day = dy;
}
//------------------------
int Date::getYear()
```

```
{
    return Date::year;
}
//--------------------------------
void Date::setYear(int yr )
{
    Date::year = yr;
}
//=========================================================

int main()  //Работа с созданным классом
{

/*здесь с помощью конструктора в экземпляре класса (экземпляр с именем MyDate)
устанавливаются заданные пользователем значения:*/

       Date MyDate(12,3,2012);  //здесь определяется и инициализируется
                                //экземпляр MyDate класса Date
       int d,m,y;
       d=MyDate.getDay(); //d=12
       m=MyDate.getMonth(); //m=3
       y=MyDate.getYear();  //y=2012
       printf("d=%d, m=%d, y=%d\n",d,m,y);

    Date BirthDate ( 1, 12, 1938 );
/*изменить значение месяца на значение 4 в экземпляре MyDate:*/
    MyDate.setMonth( 4 );
    m=MyDate.getMonth();   // m=4

/*изменить значение месяца на значение 5 в экземпляре BirthDate: */
    BirthDate.setMonth( 5 );
    m=BirthDate.getMonth(); //m=5
       _getch();
}
```

Результат работы приложения показан на рис. 8.5.

Рис. 8.5. Работа конструктора класса

Заметим, что у класса может быть много конструкторов. Когда создается класс, то для инициализации его различных полей и создаются различные конструкторы. Хотя все они имеют одинаковое имя, совпадающее с именем класса, тем не менее компилятор их различает по различным наборам параметров и типов.

Деструктор класса

Суть этой функции обратная сути функции конструктора. Она призвана разрушить созданный конструктором экземпляр класса и освободить от него память. Имя деструктора совпадает с именем класса, но перед именем указывается знак "тильда" (~). Для предыдущего примера деструктор будет иметь вид: `~Date()`. Деструктор у класса должен быть один, в то время как конструкторов может быть много.

Классы и структуры в среде CLR

Классы и структуры

Наряду с обычными классами и структурами (их еще называют native-классы и native-структуры или "родные"), существуют классы и структуры для работы в среде CLR — Common Language Runtime (их еще называют managed-классы (управляемые классы) и managed-структуры (управляемые структуры)). Эти конструкции будут располагаться в выделенной средой CLR памяти, и поэтому приложения с применением этого типа выделенной памяти должны компилироваться с ключом `/clr`. Напомним, что в версии среды 2011 консольные приложения строятся по шаблону сразу с ключом `/clr`. Приложения типа Windows Form этому условию удовлетворяют (мы их рассмотрим позже). Напомним также, что объекты, попадающие в память под управлением CLR, требуют в программе специального выделения памяти, т. к. они должны попасть на самом деле в динамическую память. Мы уже знаем, что указатели на такую управляемую память обозначаются символом `'^'`. Выделяют такую память с помощью утилиты gcnew.

Структуры и классы для работы в режиме CLR могут быть ссылочного типа (перед именем класса или структуры стоит квалификатор `ref`) и типа "значение" (перед именем класса или структуры стоит квалификатор `value`). При объявлении такой конструкции первым квалификатором идет квалификатор доступа к данной конструкции. Для класса по умолчанию идет квалификатор `private`, а для структуры — `public`, что также имеет место и для эквивалентов этих конструкций в режиме `native` (это для предыдущих версий среды). В чем смысл этих конструкций?

Например, возьмем классы:

```
ref class MyClass
{
public:
   int m_i;
};

value class MyClassV
{
public:
   int m_i;
};
```

Это обычные классы, только атрибут `ref` указывает, что его класс надо размещать в управляемой куче.

Например:

```
MyClass ^mc = gcnew MyClass();
```

Здесь `gcnew` для экземпляра класса выделяет память и размещает объект в куче и выдает указатель `mc` на начало объекта.

Если же попробовать объект разместить в native-куче, то компилятор выдаст ошибку. А для класса с атрибутом `value` компилятор разрешает это делать, т. е. оператор:

```
MyClassV *vv=new MyClassV();
```

срабатывает.

Пример:

```
int main() //Головная функция
 {
   MyClass ^mc = gcnew MyClass();//идет
   // MyClass *mc=new MyClass();//не идет
   mc->m_i=111; //присвоили значение члену класса
   int mci=mc->m_i;  //достали значение из класса
   //MyClassV ^vv=gcnew MyClassV(); //идет
   MyClassV *vv=new MyClassV(); //идет
   vv->m_i=222;
   int mv=vv->m_i;
   Console::WriteLine("mci=" + mci + " mv= " + mv);
   delete vv; //сами должны удалять объект из кучи
}
```

Результат: `mci=111` и `mv= 222`.

Оказывается, что можно ссылаться из managed-типа на native-тип. Например, функция в managed-типе (классе или структуре) может принимать native-параметр (например, `struct`). При этом если managed-тип и функция имеют атрибут доступа `public`, то и native-тип должен иметь атрибут `public`.

Например:

```
include "stdafx.h"
using namespace System;
//native-тип
public struct N
{
  int i;
};
public ref struct R //managed-тип
{
   // функция, у которой параметр nn  имеет native-тип N,
   //т. е. параметр – структура
```

```
 int f(N nn)
   {
          nn.i++;
          return(nn.i);
   }
};
int main()  //Головная функция
 {
   R ^r = gcnew R;
   N n;
   n.i=333;
   int ni=r->f(n);
   Console::WriteLine("ni=" + ni);
}
```

Абстрактные классы

Методы класса могут быть объявлены как абстрактные. Это означает, что в этом классе нет реализации этих методов. Абстрактные методы пишутся с модификатором `abstract`. Класс, в котором есть хотя бы один абстрактный метод, называется *абстрактным* (в таком классе могут быть и обычные методы). Нельзя создавать экземпляры абстрактного класса — такой класс может использоваться только в качестве базового класса для других классов. Для потомка такого класса есть две возможности: или он реализует все абстрактные методы базового класса (и в этом случае для такого класса-потомка мы сможем создавать его экземпляры), или он реализует не все абстрактные методы базового класса (в этом случае он остается тоже абстрактным классом и единственная возможность его использования — производить от него классы-потомки).

Статические функции и элементы данных

До настоящего момента каждый создаваемый нами объект имел свой собственный набор элементов данных. В зависимости от назначения вашего приложения могут быть ситуации, когда объекты одного и того же класса должны совместно использовать один или несколько элементов данных. Например, предположим, что вы пишете программу платежей, которая отслеживает рабочее время для 1000 служащих. Для определения налоговой ставки программа должна знать условия, в которых работает каждый служащий. Пусть для этого используется переменная класса `state_of_employee`. Однако, если все служащие работают в одинаковых условиях, ваша программа могла бы совместно использовать этот элемент данных для всех объектов типа `employee` (для всех служащих). Таким образом ваша программа уменьшает необходимое количество памяти, выбрасывая 999 копий одинаковой информации. Для совместного использования элемента класса вы должны объявить этот элемент как `static` (статический).

C++ позволяет иметь объекты одного и того же типа, которые совместно используют один или несколько элементов класса. Если ваша программа присваивает значение совместно используемому элементу, то все объекты этого класса сразу же получают доступ к этому новому значению. Итак, для создания совместно используемого элемента данных класса необходимо предварять имя элемента класса ключевым словом `static`.

После того как программа объявила элемент класса как `static`, она должна объявить глобальную переменную (вне определения класса), которая соответствует этому совместно используемому элементу класса.

Программы могут использовать ключевое слово `static`, чтобы сделать метод класса вызываемым в то время, как программа, возможно, еще не объявила каких-либо объектов данного класса (статический метод могут вызывать только другие статические методы).

Пример:

```
private:
static int shared_value;
```

Элемент `shared_value` некоего класса будет совместно использоваться. После объявления класса необходимо объявить элемент как глобальную переменную вне данного класса, как показано ниже:

```
int class_name::shared_value;
```

Пример программы по совместному использованию элемента данных в разных объектах показан в листинге 8.6.

Листинг 8.6

```
//8.5_2011.cpp

#include "stdafx.h"
#include <string.h>  //для strcpy()
#include <stdio.h>   //для printf()
#include <conio.h>   //для _getch()
using namespace System;

class book_series
{
public:
   book_series(char *, char *, float);
   void show_book(void);
   void set_pages(int);
private:
   static int page_count;  //это будет общий элемент
   char title[64];
   char author[ 64 ];
```

```
   float price;
};
int book_series::page_count; //объявление глобальной переменной вне
                             //класса
void book_series::set_pages(int pages)
 {
   page_count = pages;
 }
book_series::book_series(char *title, char *author, float price) //Конструктор
класса
 {
   strcpy(book_series::title, title); /*т. к. используется strcpy(),
                                       надо подключать класс string*/
   strcpy(book_series::author, author);
   book_series::price = price;
  }
void book_series:: show_book (void)
 {
   printf("Title: %s\n",title);
   printf("Author:%s\n",author);
   printf("Price: %.2f\n",price);
   printf("Pages: %d\n",page_count);
  }
void main()
 {
   book_series programming("Studiing C++", "Author1", 22.95);
//создаем объект programming с помощью конструктора
 book_series word( "Studiing to work with Word for Windows 7", "Author2",
19.95);
//создаем объект word с помощью конструктора

   word.set_pages(256); /*задаем кол-во страниц в объекте word. Оно
                         должно отразиться в объекте programming*/
   programming.show_book ();
   word.show_book() ;
   programming.set_pages(512);  //изменили page_count
   programming.show_book();     //вывод на экран данных объекта
                                //programming
   word.show_book();            //вывод на экран данных объекта
                                //word
   _getch();
}
```

Результат работы программы приведен на рис. 8.6.

Как видите, класс объявляет `page_count` как `static int`. Сразу же за определением класса программа объявляет элемент `page_count` как глобальную переменную.

Рис. 8.6. Работа со статическими элементами данных в классе

Когда программа изменяет элемент `page_count`, изменение сразу же проявляется во всех объектах класса `book_series` (ввели новое количество 512 в одном объекте, такое же количество появилось и в другом объекте).

Использование элементов с атрибутами *public static*, если объекты не существуют

Как мы только что узнали, при объявлении элемента класса как `static` этот элемент совместно используется всеми объектами данного класса. Однако возможны ситуации, когда программа еще не создала объект, но ей необходимо использовать некий элемент. Для использования элемента программа должна объявить его как `public` и `static`. Например, следующая программа (листинг 8.7) использует элемент `page_count` из класса `book_series`, даже если объекты этого класса не существуют.

Листинг 8.7

```
//8.7_2011.cpp

#include "stdafx.h"
#include <stdio.h>   //для printf()
#include <conio.h>   //для _getch()
using namespace System;

class book_series
{
       book_series();
```

```
   public:
       static void show_book(void);  //для вывода статического элемента
                                     //у функции д.б. такой же атрибут
    static int page_count;

   private:
    char title [64];
    char author[64];
    float price;
 };
int book_series::page_count; //Объявление переменной как глобальной,
                             //т. к. у нее атрибут static
 void book_series::show_book (void)
 {
   printf("page_count=%d\n",page_count);

 }

void main(void)
 {

   book_series::page_count = 256;  /*идет присвоение значения 256 переменной*/
   //employee worker("Happy Jamsa", 101, 10101.0);
//   book_series B();  //создаем объект, чтобы вывести элемент через
                       //функцию Show(). Но программа работает и без
                       //объекта
   book_series::show_book();
   _getch();
  }
```

Результат работы программы показан на рис. 8.7.

Рис. 8.7. Работа с элементом класса, когда объект класса еще не создан

В данном случае, поскольку класс определяет элемент класса `page_count` как `public`, программа может обратиться к этому элементу класса, даже если объекты класса `book_series` не существуют.

Предыдущая программа иллюстрировала использование *статических* элементов данных. Подобным образом C++ позволяет определить *статические* функции-элементы (методы). Если вы создаете *статический* метод, ваша программа может вызывать такой метод, даже если объекты не были созданы. Этот факт и был про-

демонстрирован предыдущей программой: сначала в программе был создан объект `B`. Программа отработала только тогда, когда методу вывода на экран был присвоен атрибут `static`. Затем строка, в которой был объявлен объект, была закомментирована. Программа прошла компиляцию и при выполнении показала тот же результат.

Частные и общие данные. Интерфейсные функции

При определении класса мы видели, что в нем задаются секции `public` и `private`, которые разграничивают доступ к данным класса. То есть атрибуты `public` и `private` управляют доступом к элементам класса со стороны основной программы (`main()`). В частности, программа может обратиться к общим (`public`) элементам с помощью конструкции `<имя объекта.имя члена класса>`. Такая конструкция не поможет достать данные из секции `private` в соответствии с определением секции. Поэтому программа может обращаться к частным (`private`) элементам только с использованием функций данного класса. То есть, вместо того чтобы, например, присвоить какому-то элементу из этой секции класса некоторое значение, программа должна вызвать метод класса. Такие методы называются *интерфейсными функциями*. Предотвращая с их помощью прямой доступ к элементам данных, "спрятанных" в секции `private`, вы таким образом можете гарантировать, что данным класса всегда будут присваиваться допустимые значения.

Интерфейсную функцию всегда надо помещать в секцию `public` класса, чтобы к ней был доступ и чтобы с ее помощью получать доступ к данным из секции `private`.

Использование оператора глобального разрешения для элементов класса

Чтобы избежать конфликта между именами параметров методов класса и именами элементов класса, применяют так называемый *оператор глобального разрешения* для элементов класса, который обозначается двумя двоеточиями подряд (`::`). Если перед именами элементов класса ставить имя класса, к которому принадлежит элемент, а за именем — оператор глобального разрешения, то конфликтов между именами можно избежать. Например, выражение `employee::employee_id = employee_id;` показывает, что идентификатор слева от знака равенства — элемент класса `employee`, а справа — классу `employee` не принадлежит.

C/C++ Microsoft Visual

ГЛАВА 9

Ввод и вывод в языках 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. Главная 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.

Основные функции для работы с файлами

После того как файл открыт, для чтения или записи используют специальные функции. Приведем перечень функций для работы с файлами.

- Функция `fputc()`.

  Формат:

  ```
  fputc(c, fp);
  ```

  Выводит символ в файл (`c` — выводимый символ, `fp` — указатель файла).

- Функция `fputs()`.

  Формат:

  ```
  fputs(s,fp);
  ```

  Выводит строку в файл (`s` — выводимая строка, `fp` — указатель файла).

- Функция `fgetc()`.

  Формат:

  ```
  c=fgetc(fp);
  ```

  Читает один символ из файла с указателем `fp`. Переменная `c` описана как `char c`. В случае ошибки или достижения конца файла возвращает `EOF`.

- Функция `fgets()`.

  Формат:

  ```
  fgets(s,maxline,fp);
  ```

Читает строку в `s` (`s` — массив символов или указатель типа `char` (предварительно должна быть выделена память для чтения с использованием указателя), `maxline` — максимальное число символов, которое требуется читать из файла с указателем `fp`). В случае ошибки или достижения конца файла возвращает `NULL`.

- Функция `fread()`.

  Формат:

  ```
  fread(buf, m, n, fp);
  ```

  Читает из файла с указателем `fp` `n` элементов данных, каждый из которых имеет длину `m` байтов. Чтение происходит в буфер, на который указывает указатель (например, `char buf[50]` или `char *buf` (но в этом случае надо выделить память для буфера)). Общее количество байтов чтения составит `n×m`. Функция возвращает количество прочитанных элементов, а по достижении конца файла или возникновении ошибки чтения возвращает `NULL`.

- Функция `fwrite()`.

  Формат:

  ```
  fwrite(const void ptr, m, n, fp);
  ```

  Пишет в файл с указателем `fp`: добавляет `n` элементов в выходной файл, каждый элемент длиной в `m` байтов. Данные записываются из буфера, на который указывает указатель `ptr` (этот указатель указывает на некоторый объект, например, на структуру). Общее число записанных байтов равно `n×m`. В случае ошибки записи функция возвращает ноль, в противном случае — количество записанных элементов.

- Функция `fseek()`.

  Формат:

  ```
  fseek(fp,n,m);
  ```

  Устанавливает указатель в файле в позицию, отстоящую на `n` байтов от текущей, а направление перемещения (вперед или назад) задается параметром `m`, который может быть одним из значений: 0, 1, 2 или одной из трех символических констант, определенных в файле stdio.h:

  - `SEEK_SET 0` — к началу файла;
  - `SEEK_CUR 1` — указатель в текущей позиции файла;
  - `SEEK_END 2` — к концу файла.

  Функция `fseek()` используется для ввода/вывода потоком.

  Для работы не с потоковыми данными следует использовать функцию `lseek()`.

  После функции `fseek()` можно выполнять операции обновления в файлах, открытых для обновления. При удачном завершении работы `fseek()` возвращает ноль, в противном случае — иное значение. Функция `fseek()` возвращает код

ошибки, только если файл или устройство не открыты. В этих случаях глобальная переменная `errno` принимает одно из следующих значений:

- `EBADF` — неверный указатель файла;
- `EINVAL` — неверный аргумент функции;
- `ESPIPE` — поиск на устройстве запрещен.

♦ Функция `ftell()`.

Формат:

```
long int ftell(fp);
```

Возвращает текущее значение указателя файла `fp` (т. е. номер текущей позиции) в виде значения типа `long int`. Отсчет идет в байтах от начала файла. Возвращаемое значение может быть использовано в функции `fseek()`. Если обнаружены ошибки, функция возвращает значение `-1L` и присваивает глобальной переменной `errno` положительное значение.

♦ Функция `fscanf()`.

Формат:

```
fscanf(fp,Control,arg1,arg2,...,argn);
```

Вводит данные из файла с указателем `fp`, преобразует их по форматам, записанным в управляющей строке `Conrtol`, и отформатированные данные записывает в аргументы `arg1,...,argn`. Подробные сведения о работе этой функции можно получить, ознакомившись с работой функции `scanf()` (функцию `scanf()` мы рассмотрим в *разд. "Функции стандартного ввода/вывода" далее в этой главе*).

♦ Функция `fprintf()`.

Формат:

```
fprinf(fp,Control,arg1,arg2,...,argn);
```

Выводит данные в файл с указателем `fp`, преобразует аргументы `arg1,...,argn` к форматам, которые записаны в управляющей строке `Conrtol`, и отформатированные данные записывает в файл. Подробные сведения о работе этой функции можно получить, ознакомившись с работой функции `printf()` (об этой функции *см. разд. "Функции стандартного ввода/вывода" далее в этой главе*).

♦ Функция `rewind()`.

Формат:

```
rewind(fp);
```

Устанавливает указатель позиционирования в файле с указателем `fp` на начало потока. Функция `rewind(fp)` эквивалентна функции `fseek(fp, 0L, SEEK_SET)` за исключением того, что `rewind()` сбрасывает индикатор конца файла и индикаторы ошибок, а `fseek()` сбрасывает только индикатор конца файла. После функции `rewind()` можно выполнять операции обновления для файлов, открытых для обновления. Возвращаемого значения нет.

- Функция `ferror()`.

  Формат:

  ```
  ferror(fp);
  ```

  Функция тестирует поток на ошибки чтения/записи. Если индикатор ошибки устанавливается, то он остается в таком положении, пока не будут вызваны функции `clearerr()` или `rewind()`, или до того момента, пока поток не закроется. Если в файле была обнаружена ошибка, то функция `ferror()` возвращает ненулевое значение.

- Функция `freopen()`.

  Формат:

  ```
  FILE *freopen(const char *FILEname, const char *mode, FILE *stream);
  ```

  Функция `freopen()` подставляет файл, заданный в первом параметре, вместо уже открытого потока. Она закрывает поток независимо от того, открыт ли он. Эта функция полезна для замены файла, связанного со стандартными устройствами ввода/вывода: `stdin`, `stdout` или `stderr`. Способы открытия файла аналогичны таким же в функции `fopen()`. При успешном завершении функция возвращает указатель типа `FILE` (как и функция `fopen()`), при неудачном — `NULL`.

  Пример перенаправления потока с помощью функции `freopen()` приведен в листинге 9.1.

Листинг 9.1

```
/* перенаправление стандартного вывода в файл */
   if (freopen("OUTPUT.FIL", "w", stdout) == NULL)
      fprintf(stderr, "error redirecting stdout\n");

   /* этот вывод пойдет уже в файл */
   printf("This will go into a FILE.");

   /* закрытие стандартного потока*/
   fclose(stdout);
```

- Функция `feof()`.

  Формат:

  ```
  feof(fp);
  ```

  Обнаруживает конец файла с указателем `fp`: тестирует поток на возникновение индикатора конца файла (который наступает после прочтения последней записи). Как только индикатор установлен, операции чтения файла возвращают индикатор до тех пор, пока не выполнена функция `rewind()` — "перемотка" в начало файла или пока файл не будет закрыт. Индикатор конца файла переустанавливается с каждой операцией ввода. Функция возвращает ненулевую величину,

если индикатор конца файла был обнаружен при последней операции чтения, в противном случае — ноль.

- Функция `ferror()`.

  Формат:

  ```
  ferror(fp);
  ```

  Здесь `fp` — указатель файла (`FILE *fp;`). Функция выдает ненулевое значение, если операция с файловым потоком завершается с ошибкой (например, возникает ошибка чтения файла). Для обработки ошибок ввода/вывода следует записать эту функцию перед блоком работы с файлом в виде:

  ```
  if(ferror(fp)) {команды обработки ошибок ввода/вывода}.
  ```

  Как только произойдет ошибка, выполнится эта функция, и начнут работать команды обработки ошибок.

- Функция `exit()`.

  Формат:

  ```
  exit(int status);
  ```

  Эта функция используется для срочного завершения работы программы при обработке ошибок открытия файла (и не только для этого, а для прерывания работы программы по каким-либо другим причинам). Но перед завершением все файлы закрываются, остатки данных, находящиеся в буфере вывода, записываются в память и вызываются функции обработки ошибок, предварительно зарегистрированные специальной функцией `atexit()`. Эти функции обработки ошибок надо самому написать и зарегистрировать их с помощью вызова функции `atexit()`.

  Для вызова функции `atexit()` требуется выполнить команду `#include <stdlib.h>`. Каждый вызов `atexit()` регистрирует новую функцию `exit()`. Можно зарегистрировать до 32 функций `exit()`. Они будут выполняться по принципу работы стековой памяти: "последний вошел — первый вышел" (т. е. последняя зарегистрированная функция будет выполнена первой). Поясним сказанное на примере программы, приведенной в листинге 9.2 (результат работы программы показан на рис. 9.1).

Листинг 9.2

```
// 9.1_2011.cpp

#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
using namespace System;
```

```
/*это первая функция exit() */
 void exit_fn1(void)
 {
    printf("Exit function #1 called\n");
       _getch();
 }

/*это вторая функция exit() */
 void exit_fn2(void)
 {
    printf("Exit function #2 called\n");
       _getch();
 }

/*Это основная программа, в которой происходит регистрация заданных вами
функций exit(). Здесь же применяется и сама функция exit(), которая перед
завершением работы программы станет вызывать зарегистрированные функции,
определенные выше */

 int main(void)
 {
    /* регистрация функции #1 */
   atexit(exit_fn1);
    /* регистрация функции #2 */
   atexit(exit_fn2);

/*exit() сначала вызовет функцию #2, т. к. она была зарегистрирована последней,
а затем — функцию #1. После этого программа завершится.*/
   printf("fn2 will be called first\n");
 exit(0);
 _getch(); //здесь задержки экрана не получится, т. к. exit() закрывает
программу

 }
```

Рис. 9.1. Результат работы программы листинга 9.2

Какое бы числовое значение вы не подставили вместо аргумента функции, вызов зарегистрированных функций `exit()` все равно произойдет. Если же не зарегистрирована ни одна функция, то произойдет завершение программы. Регистрация функций `exit()` действительна только в пределах одной программы.

Стандартный ввод/вывод

При запуске любой программы автоматически открываются сразу три 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.

Функции стандартного ввода/вывода

- Функция `getchar()`.

  Формат:

  ```
  getchar();.
  ```

  Вводит с клавиатуры один символ и выдает его. Обращаться к этой функции можно так:

  ```
  char c; c=getchar();
  ```

  или

  ```
  int c; c=getchar();
  ```

- Функция `putchar()`.

  Формат:

  ```
  putchar(char c);
  ```

Выводит значение переменной `c` (один символ) на стандартное выводное устройство. Обращение:

```
putchar(c);
```

♦ Функция `printf()`.

Формат:

```
printf(Control,arg1,arg2,...,argn);
```

Функция форматного вывода. Выводит на экран содержимое `arg1,arg2,...,argn` и возвращает количество выводимых байтов.

Здесь `Control` — управляющая символьная строка, в которой находятся форматы вывода на экран для соответствующих аргументов `arg1,arg2,...,argn`, т. е. первый формат — для вывода `arg1`, второй — для `arg2` и т. д.

Все символы, находящиеся между форматами, выводятся один к одному (т. е. не форматируются), что дает возможность вывода дополнительного текста для улучшения читаемости результата вывода.

Форматы вывода задаются так: любой формат начинается с символа `%` и заканчивается одним из символов форматирования:

- `d` — аргумент преобразуется к десятичному виду (с учетом знака);
- `o` — аргумент преобразуется к восьмеричному беззнаковому виду;
- `x` — аргумент преобразуется в беззнаковую шестнадцатеричную форму (с символами `a`, `b`, `c`, `d`, `e`, `f`);
- `X` — аргумент преобразуется в беззнаковую шестнадцатеричную форму (с символами `A`, `B`, `C`, `D`, `E`, `F`);
- `U` — аргумент преобразуется в беззнаковую десятичную форму;
- `c` — аргумент рассматривается как отдельный символ;
- `s` — аргумент рассматривается как строка символов; символы строки печатаются до тех пор, пока не будет достигнут нулевой символ или не будет напечатано количество символов, указанное в спецификации точности (о спецификаторе точности скажем далее);
- `e` — аргумент рассматривается как переменная типа `float` или `double` и преобразуется в десятичную форму в экспонентном виде `[-]`*m*`.`*nnnnnn* `e[+-]`*xx*, где длина строки из `n` определяется указанной точностью. По умолчанию точность равна 6;
- `E` — то же, что и `e`, но с `E` для экспоненты;
- `f` — аргумент рассматривается как переменная типа `float` или `double` и преобразуется в десятичную форму в виде `[-]`*mmm*`.`*nnnnnn*, где длина строки из *n* определяется указанной точностью. По умолчанию точность равна 6;
- `g` — используется либо формат `%e`, либо `%f`: выберется тот формат, который даст изображение числа меньшим количеством знаков с учетом заданной точности. Незначащие нули не печатаются;

- `G` — то же, что и `g`, но с `E` для экспоненты, если используется формат `e`;
- `n` — указатель на целое со знаком;
- `p` — входной аргумент выводится как указатель. Формат зависит от модели используемой памяти. Он может быть вида *XXXX*:*YYYY* или *YYYY* (только смещение).

Между границами формата вывода находятся:

`[флажки] [ширина] [.точность] [F|N|h|l|L].`

- *квадратные скобки* означают, что элемент, входящий в них, может отсутствовать для какого-то формата. Например, если выводится десятичное число, то точность для него не имеет смысла;
- *флажки* определяют выравнивание выводимого значения (по правому или по левому краю поля вывода), знаки числа, десятичные точки, конечные нули, восьмеричные и шестнадцатеричные префиксы. Флажки имеют следующий смысл:
  - выравнивание результата по левому краю поля вывода (число будет прижато к левой границе поля вывода) и заполнение поля вывода справа пробелами. Если этот флаг не задан, то результат выравнивается по правому краю поля вывода, а оставшееся слева пространство заполняется пробелами или нулями;
  - `+` — преобразование результата к виду со знаком: результат всегда начинается со знака "+" или "–";
  - *пробел* — если значение неотрицательное, то вместо плюса выводится пробел. Для отрицательных чисел выводится минус;
  - `#` — указывает, что аргумент должен быть преобразован с использованием альтернативной формы. Это означает, что если флажок `#` используется вместе с символом преобразования (форматирования), то при преобразовании аргумента для символов `c`, `s`, `d`, `u`, `i`, `o`, `x`, `X` символ `#` не влияет на результат. Для символов `e`, `E`, `f` результат всегда будет содержать десятичную точку, даже если за точкой не следует никаких цифр (обычно десятичная точка появляется, если за ней следует цифра). Для символов `g`, `G` результат будет как для символов `e`, `E`, но с тем отличием, что хвостовые нули не будут удаляться.

Примечание

Если заданы и пробел, и знак "+", то преимущество имеет знак "+".

- *спецификатор ширины* определяет размер поля для выходного значения. Ширину можно задать двумя способами:
  - напрямую — строкой десятичных цифр;
  - косвенно — через символ `*` (в этом случае аргумент должен иметь тип `int`). Если для задания ширины используется символ `*` (звездочка), то спе-

цификация ширины указывается не в строке `Control`, а в списке аргументов перед соответствующим аргументом.

Спецификаторы ширины:

- `n` — в этом случае выведется не менее `n` символов. Если в выводимом числе символов меньше, чем `n`, то оставшаяся часть поля заполнится пробелами справа (если задан флажок), и слева — в противном случае;
- `0n` (например, `04`) — будет выведено не менее `n` символов. Если в выводимом числе символов меньше, чем `n`, то оставшаяся часть поля заполнится слева нулями;
- `*` (для формата `d`) — если для задания ширины используется символ `*`, то спецификация ширины указывается не в строке `Control`, а в списке аргументов перед тем аргументом, для которого она определена. Причем ширина представляет собой отдельный аргумент. Например, выводим число `i=2` по функции `printf("%*d\n",5,i);`. Результат будет `пппп2` (где `п` — пробел). Здесь ширина задана равной 5 и указана в списке аргументов отдельно, но перед тем аргументом, для которого она определена;

- *спецификатор точности* задает число выводимых символов после точки (дробная часть числа). Задание спецификатора точности всегда начинается со знака точки (чтобы отделить его от спецификатора ширины). Как и спецификатор ширины, точность может задаваться двумя способами:
  - напрямую — заданием числа;
  - косвенно — указанием символа `*`. Если вы используете `*` для спецификатора точности, то спецификация точности должна указываться не в строке `Control`, а в списке аргументов перед соответствующим аргументом как отдельный аргумент (через запятую).

Примечание

Аргумент, для которого указывается спецификация точности, может быть только вещественного типа.

Например:

```
float nn=12.567;
printf("%.*f\n",2,nn); //вывести 2 знака после точки
```

Результат: `12.57`

- *модификаторы размера*, задаваемые в формате, определяют, как функция интерпретирует аргумент. Действие модификаторов показано в табл. 9.1.

Таблица 9.1. *Действие модификаторов размера*

Значение	Формат	Интерпретация
h	d, i, o, u, x, X	short int
l	d ,i ,o, u, x, X	long int

Таблица 9.1 *(окончание)*

Значение	Формат	Интерпретация
l	e, E, f, g,G	double
L	e, E, f, g,G	long double
L	d ,i ,o, u, x, X	__int64
h	c, C	1 символьный байт
l	c, C	2 символьных байта
h	s, S	1 строка символов по 1 байту на символ
l	s, S	1 строка символов по 2 байта на символ

Например, если мы выводим данные типа `long`, то должны задавать вместе с форматом `d` и модификатор типа `l`, т. е. общий вид формата будет `ld`.

- Функция `scanf()`.

Формат:

```
scanf(Control,arg1,arg2,...,argn);
```

Функция форматного ввода с клавиатуры. Осуществляет посимвольный ввод данных с клавиатуры, преобразует их для каждого значения в соответствии с форматом, указанным в управляющей (форматной) символьной строке `Control`, и результат преобразования записывает в аргументы `arg1, arg2,...,argn`. Смысл строки `Control` тот же, что и для функции `printf()`.

Так как `arg1, arg2,..., argn` — это выходные параметры функции, то при обращении к функции они должны задаваться своими адресами: имена массивов — именами (т. к. имена массивов — это указатели на их первые элементы), а те аргументы, что не являются указателями, задаются как `&arg`.

Форматная строка — это символьная строка, содержащая три типа объектов: незначащие символы, значащие символы и спецификации формата.

Незначащие символы — это пробел, знак табуляции (`\t`), символ перехода на новую строку (`\n`). Как только функция встречает незначащий символ в строке формата, она считывает (но не сохраняет) все последующие незначащие символы до тех пор, пока не встретится первый значащий символ (т. е. пропускает незначащие символы).

Значащие символы — это все символы кода ASCII, кроме символа `%`. Если функция встречает в форматной строке значащий символ, она его считывает, но не сохраняет.

Спецификация формата функции имеет вид:

```
%[*]  [ширина] [F/N] [h/l] символ формата
```

После символа начала формата `%` в определенном порядке следуют остальные спецификации.

`[*]` — это необязательный символ подавления ввода: весь входной поток функция рассматривает как совокупность *полей ввода*: значащих символов. Если в спецификации указан символ `*`, то все поле, которое должно в данный момент обрабатываться функцией по заданному формату, пропускается.

Ввод происходит так: в соответствии со спецификатором ширины первого формата из входного потока выбирается очередное поле ввода (т. е. значащие символы до первого незначащего), которое интерпретируется в соответствии с форматом и записывается в соответствующий аргумент.

Если при этом запрошенная ширина оказалась меньше поля ввода, то остаток поля обрабатывается функцией по следующему формату.

Если запрошенная ширина оказалась больше поля ввода, то все поле ввода обрабатывается по данному формату.

Если же в объявлении формата присутствовал символ подавления ввода `*`, то все поле, предназначенное для обработки данным форматом, пропускается.

Модификаторы размера аргумента и символы форматирования функции `scanf()` аналогичны модификаторам и символам форматирования функции `printf()`.

Рассмотрим пример работы функции `scanf()`.

Допустим, задано:

```
int i; float x; char m[100];
```

На клавиатуре набираем последовательность:

```
56789 0123 45a72
```

Выполняем:

```
scanf("%2d %f %*d %2s", &i,&x,m);
```

Как будет идти ввод?

В примере имеются три поля ввода: `56789`, `0123` и `45a72`. Наберем их на клавиатуре и нажмем клавишу <Enter>. Функция в соответствии с первым форматом (`%2d`) выбирает из первого поля первые два символа. Функция интерпретирует их как десятичное число и присваивает значение первому аргументу: `i = 56`.

В первом поле остались необработанными символы `789`. Они попадают в работу функции по второму формату: `%f`. Второй аргумент получит значение `x = 789`. Далее должно обрабатываться поле `0123` по третьему формату, но в нем есть символ подавления. Поэтому поле пропускается и начинает обрабатываться поле `45a72` по формату `%2s`. Из этого поля будут выбраны только первые два символа и строка `m` получит значение `45`.

- Функция `sprintf()`.

  Формат:

```
sprintf(string,Control,arg1,arg2,...,argn);
```

Эта функция аналогична `printf()`, за исключением того, что результат своей работы она выводит не на стандартное устройство вывода, а в строку `string`. Это позволяет собирать в одну строку данные совершенно разных типов.

- Функция `sscanf()`.

  Формат:

  ```
  sscanf(string,Control,arg1,arg2,...,argn);
  ```

  Эта функция аналогична `scanf()`, за исключением того, что входные данные для ее работы поступают не со стандартного устройства ввода, а из строки `string`. Это позволяет выделять в строке различные группы данных совершенно разных типов и помещать их в отдельные переменные.

- Функция `gets()`.

  Формат:

  ```
  gets(s);
  ```

  Вводит строку символов с клавиатуры и записывает ее в строку `s`, которая может быть объявлена как `char *s` или `char s[]`.

- Функция `puts()`.

  Формат:

  ```
  puts(s);
  ```

  Выводит содержимое строки `s` на устройство стандартного вывода (экран) (`s` может быть объявлена как `char *s` или `char s[]`).

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` используются 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 кла-

виатура) и вывод на стандартное выводное устройство (которым по умолчанию является экран). Таким образом, включение 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 (для получения доступа 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 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 так:

```
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 далее, 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 вводят в каждый составляющий единый проект текстовый программный файл с помощью директивы:

```
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 9.3 приводится пример использования директив пространства имен, результат работы программы показан на рис. 9.2.

Листинг 9.3

```
// 9.3_2011.cpp
//

#include "stdafx.h"
#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();
     }
```

Рис. 9.2. Результат работы программы листинга 9.3

`std::cout` — это стандартный вывод. Его мы рассмотрим чуть позже. 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 (указатель, показывающий, в каком месте файла мы находимся), прочитать данные. Этот класс имеет такие основные функции (методы):

- `open()` — открывает файл;
- `close()` — закрывает файл;

- `is_open()` — если файл открыт, то функция возвращает `true`, иначе — `false`;
- `rdbuf()` — выдает указатель на буфер ввода/вывода.

Параметры функции `open()`:

- имя открываемого файла;
- способ открытия файла.

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.

В листинге 9.4 приведен пример программы работы с классом `fstream`, результат работы показан на рис. 9.3.

Листинг 9.4

```
// 9.4_2011.cpp

#include "stdafx.h"
#include<fstream>
#include<iostream>
#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_base::in | ios_base::out | ios_base::trunc);
```

```
    // 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();
}
```

Алгоритм примера ясен из комментария к тексту программы: три строки записываются в файл, затем оттуда читаются и выводятся на экран. После этого на место 2-й и 3-й строк пишутся новые данные (на самом деле — те же самые) и снова содержимое файла выводится на экран.

Рис. 9.3. Результаты работы программы листинга 9.4

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).

В листинге 9.5 приведен пример использования класса `ofstream`.

Листинг 9.5

```
#include <fstream>
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 следующих методов:

- `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 достигает конца файла;
- `peek()` — выдает очередной символ потока, но не выбирает его (не сдвигает указатель позиционирования данного 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 9.6.

Листинг 9.6

```
#include <fstream>
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 конца файла
 {
  FILE >> p;                //чтение из файла
 cout << p  << endl;      // вывод прочитанных данных на экран
}
FILE.close();           //закрытие файла
```

В листинге 9.7 приведен пример использования классов `ofstream` и `ifstream` (оба находятся в `fstream`), результат работы программы показан на рис. 9.4.

Листинг 9.7

```
// 9.5_2011.cpp

#include "stdafx.h"
#include<iostream>
```

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

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

int main()
{
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 надо 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 */
```

```
streamoff 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 c
cout << c;
 } //while
cout << endl;
FILE2.close();
_getch();
system("DEL C:\\A.TXT");  // удаление рабочего файла
} //main()
```

Рис. 9.4. Результат работы программы листинга 9.7

Работа с бинарным файлом

Бинарные файлы, в отличие от потоковых, создаются в определенной логической структуре и поэтому должны читаться в переменную той же структуры. Пример программы приведен в листинге 9.8, результат работы программы показан на рис. 9.5.

Листинг 9.8

```
// 9.6_2011.cpp

#include "stdafx.h"
#include<iostream> //для cin, cout
#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();
}
```

Рис. 9.5. Результат работы программы листинга 9.8

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 за-

писывать данные. Этот указатель равен адресу структуры, т. е. `&b[i]`. Но в потоке 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;
- оператор `goto` применен для подстраховки от превышения индекса массива `bb[]`.

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, а затем пользовались методами и операциями: << , >>. Но как мы видели в начале 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 C++ 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, нужно 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 не видно форматов, но при вводе значений этих переменных с клавиатуры (после ввода каждого значения надо нажимать клавишу <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 C и C++ синхронизированы.

При выводе данные могут быть отформатированы с помощью функций-членов класса или манипуляторов. Их перечень приводится в табл. 9.2.

Примечание

Манипуляторы, начинающиеся с "no" (noshowpos и т. п.), имеют обратное действие по отношению к манипуляторам с такими же именами, но без приставки "no". В графе "Описание" у таких манипуляторов проставлены пробелы.

Таблица 9.2. *Манипуляторы и функции стандартного ввода/вывода в C++*

Манипуляторы	Функции-члены класса	Описание
showpos	setf(ios::showpos)	Выдает знак плюс у выводимых положительных чисел
noshowpos	unsetf(ios::showpos)	
showbase	setf(ios::showbase)	Выдает базу системы счисления в выводимом числе в виде префикса
noshowbase	unsetf(ios::showbase)	
uppercase	setf(ios::uppercase)	Заменяет символы нижнего регистра на символы верхнего регистра в выходном потоке
nouppercase	unsetf(ios::uppercase)	
showpoint	setf(ios::showpoint)	Создает символ десятичной точки в сгенерированном потоке с плавающей точкой (в выводимом числе)
noshowpoint	unsetf(ios::showpoint)	
boolalpha	setf(ios::boolalpha)	Переводит булевый тип в символьный
noboolalpha	unsetf(ios::boolalpha)	
unitbuf	setf(ios::unitbuf)	Сбрасывает буфер вывода после каждой операции вывода
nounitbuf	unsetf(ios::unitbuf)	
internal	setf(ios::internal, ios::adjustfield)	Добавляет символы-заполнители к определенным внутренним позициям выходного потока (речь идет о выводе числа в виде потока символов). Если такие позиции не определены, поток не изменяется
left	setf(ios::left,ios::adjustfield)	Добавляет символы-заполнители с конца числа (сдвигая число влево)
right	setf(ios::right,ios::adjustfield)	Добавляет символы-заполнители с начала числа (сдвигая число вправо)

Таблица 9.2 *(окончание)*

Манипуляторы	Функции-члены класса	Описание
dec	setf(ios::dec,ios::basefield)	Переводит базу вводимых или выводимых целых чисел в десятичную (введенные после этого манипулятора данные будут выводиться как десятичные)
hex	setf(ios::hex,ios::basefield)	Переводит базу вводимых или выводимых целых чисел в шестнадцатеричную (введенные после этого манипулятора данные будут выводиться как шестнадцатеричные)
oct	setf(ios::oct,ios::basefield)	Переводит базу вводимых или выводимых целых чисел в восьмеричную (введенные после этого манипулятора данные будут выводиться как восьмеричные)
fixed	setf(ios::fixed,ios::floatfield)	Переводит выход с плавающей точкой в выход с фиксированной точкой
scientific	setf(ios::scientific,ios:: floatfield)	Выдает числа с плавающей точкой в виде, используемом в научных целях: например, число 23450000 будет записано как: 23.45e6
	setbase(int base)	Преобразует ввод целых чисел в тип base, где параметр base может быть одним из чисел 8, 10 или 16
fill(c)	setfill(char_type c)	Задает символ заполнения при выводе данных
precision(n)	setprecision(int n)	Задает точность вывода данных (количество цифр после точки)
setw(int n)	width(n)	Задает ширину поля для выводимых данных (количество символов)
endl		Вставляет символ новой строки ('\n') в выходную последовательность символов и сбрасывает буфер ввода
ends		Вставляет символ '\0' в выходную последовательность символов
flush	flush( )	Сбрасывает буфер вывода
ws		Задает пропуск пробелов при вводе

Значения по умолчанию:

- precision() — 6;
- width() — 0;
- fill() — пробел.

В листинге 9.9 приведен пример программы с применением объекта `cout` (все пояснения можно найти в комментариях). Результат работы программы представлен на рис. 9.6.

Листинг 9.9

```
// 9.7_2011.cpp

#include "stdafx.h"
#include <iostream>
#include <iomanip>  //включение манипуляторов
#include <conio.h>

// пример использования cout

void main ( )
{

  using namespace std;
  int i;
  float f;
  cout << "Enter i and f >" << endl;
  // чтение целого числа и числа с плавающей точкой с устройства stdin
  cin >> i >> f;
  // вывод целого и переход на новую строку
  cout << i << endl;
  // вывод числа с плавающей точкой и переход на новую строку
  cout << f << endl;
  // вывод в шестнадцатеричной системе
  cout << hex << i << endl;
  // вывод в восьмеричной и десятичной системах
  cout << oct << i << dec << i << endl;
  // вывод i с указанием его знака
  cout << showpos << i << endl;
  // вывод i в шестнадцатеричной системе
  cout << setbase(16) << i << endl;
  /*вывод i в десятичной системе и дополнение справа символом @ до ширины
в 20 символов (заполнение начинается от правой границы к левой). Если вы
вводите, например, 45, то выведется 45@@@@@@@@@@@@@@@@@@@ */
  cout << setfill('@') << setw(20) << left << dec << i;
  cout << endl;
  // вывод того же результата в том же формате,
  // но с использованием функций вместо манипуляторов
  cout.fill('@');
  cout.width(20);
  cout.setf(ios::left, ios::adjustfield);
```

```
  cout.setf(ios::dec, ios::basefield);
  cout << i << endl;
  // вывод f в научной нотации с точностью 10 цифр
  cout << scientific << setprecision(10) << f << endl;
  // изменение точности до 6 цифр
   cout.precision(6);
  // вывод f и возврат к нотации с фиксированной точкой
  cout << f << fixed << endl;
 _getch();
} //main()
```

Рис. 9.6. Результат работы программы листинга 9.9

Стандартный ввод *cin*

Объект (экземпляр класса) `cin` управляет вводом из буфера, 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 9.10, результат работы программы — на рис. 9.7.

Листинг 9.10

```
// 9.8_2011.cpp

#include "stdafx.h"
#include <fstream>
#include <iostream>
#include <conio.h>
```

```
// 1-й пример использования cin

void main ( )
{
  using namespace std;

  int i;
  float f;
  char c;
  //ввод целого числа, числа с плавающей точкой и символа с 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;

//
// 2-й пример использования cin
//

  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();
}
```

Рис. 9.7. Результат работы программы листинга 9.10

C/C++

Microsoft® Visual

ЧАСТЬ II

Приложения Windows Form

Глава 10. Продолжение изучения среды Visual C++

Глава 11. Компоненты, создающие интерфейс между пользователем и приложением

Глава 12. Работа с наборами данных. Общие сведения о базах данных

Глава 13. Управление исключительными ситуациями

Глава 14. Преобразование между нерегулируемыми и регулируемыми (режим CLR) указателями

ГЛАВА 10

Продолжение изучения среды Visual C++

В *главе 1* мы рассмотрели структуру главного окна среды разработки программ, а также увидели, что все программы пользователя оформляются в качестве приложений к среде и помещаются в специальные конструкции, называемые проектами. Там же мы научились составлять консольные приложения. В этой и последующих главах мы изучим, как разрабатывать уже более сложные приложения с графическими интерфейсами. В частности, мы займемся созданием приложений с так называемыми формами. В изучаемой среде такие приложения называются Windows Form Application. Окно такого приложения показано на рис. 10.1.

Из *главы 1* нам известно, что множество окон, с которыми работает среда программирования, может перемещаться в рамках рабочего стола совершенно причудливым образом в рамках того или иного проекта. Причем положение окон, как и сам проект, автоматически сохраняется средой после каждого изменения содержимого рабочего стола. Поэтому если вы случайно разместили окна на рабочем столе так, что вам становится неудобно с ними работать, вернитесь к виду рабочего стола, предусмотренному по умолчанию. Это выполняется опцией главного меню **Window | Reset Window | Layout**. Вот от этого вида мы и станем отталкиваться (рис. 10.2).

В правой части этого главного окна прямо по его торцу расположены две вкладки, обеспечивающие появление соответственно двух окон (рис. 10.3). Первое окно информирует о связях баз данных, с которыми работает ваше приложение, второе окно содержит перечень компонентов среды проектирования приложений, которые могут помещаться в формы.

Более подробно работу каждой из частей главного окна (рабочего стола) мы будем рассматривать по мере дальнейшего изучения материала.

Создание проекта

Чтобы приступить к разработке новой программы (т. е. приложения), надо выполнить команду **File | New | Project**. Откроется диалоговое окно для выбора типа приложения, где нужно выполнить пронумерованные на рис. 10.4 действия.

Рис. 10.1. Вид главного окна для приложений типа Windows Form Application

Рис. 10.2. Вид рабочего стола, принятый по умолчанию

Рис. 10.3. Окно вкладок **Toolbox** и **Server Explorer**

Рис. 10.4. Диалоговое окно для выбора типа приложения

После нажатия кнопки **OK** на рабочем столе появится объект, который называется *формой* (рис. 10.5).

Если окна **Properties** по каким-то причинам на экране нет, надо открыть контекстное меню формы и в нем выполнить команду (опцию) **Properties**. Окно свойств формы появится на экране (рис. 10.5).

Рис. 10.5. Загрузка формы на рабочий стол

Новый проект можно создать и с помощью кнопки быстрого вызова **New Project** (третья слева в строке, расположенной ниже строки с опциями главного меню).

Итак, в результате действий по созданию приложения, мы увидим на экране объект, называемый формой, — это экземпляр класса `Form`. Одновременно с этими действиями среда IDE автоматически создает файлы проекта (структура проекта видна в окне **Solution Explorer**) и среди них — программный модуль-заготовку для помещения в него программ-обработчиков событий компонентов, которые будут размещены в форме.

Чтобы увидеть этот модуль, надо открыть контекстное меню формы и выполнить в нем команду **View Code** (показать код). При этом в верхней части окна проектирования, где находится форма, появится вкладка **Form1.h**, а в ней — программный модуль. Заметьте, что экземпляр класса `Form`, который виден на рабочем столе (т. е. сама форма), действительно является наследником класса `Form` и получил имя `Form1`. Кроме того, класс `Form1` имеет квалификатор `ref`, говорящий о том, что приложение будет работать в среде CLR, т. е. с автоматическим сборщиком мусора. Если по каким-либо причинам форма не видна (например, случайно была закрыта

вкладка **Form1.h [Design]**), то, чтобы форма появилась на рабочем столе, следует выполнить опцию **View Designer** контекстного меню элемента **Form1.h** (курсор мыши надо установить в поле с операторами программы и нажать правую кнопку мыши). Как известно, контекстное меню любого элемента открывается, если щелкнуть на этом элементе правой кнопкой мыши.

Форма — это главное действующее лицо при создании проекта в среде VC++. Это главный контейнер, в котором размещаются компоненты самой среды. С помощью этих компонентов и реализуется 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 появится 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 же программный модуль, который создается вместе с появлением формы на экране (**Form1.h**, например).

Итак, мы получили на экране пустую форму. Поскольку по правилам проектирования в дальнейшем в форму надо будет помещать компоненты (а их мы будем брать из палитры компонентов, расположенных в окне вкладки **Toolbox**), то может потребоваться изменение размеров формы. Это делается протяжкой формы за анкерные точки — небольшие квадратики, выделенные по углам и сторонам формы. Кроме того, зацепившись за заголовки окон, можно перетащить окна в удобное для вас место рабочего стола.

Когда вы создадите проект, его надо сохранить (вообще-то среда сама автоматически все сохраняет) командой **Save All** (подменю опции **File** главного меню), после чего проект следует откомпилировать и построить (нажав клавишу <F7> или выполнив команду **Build | Build Solution** главного меню), а затем проект следует выполнить, воспользовавшись командами опции **Debug** (клавишами <Ctrl>+<F5> или <F5>). То есть здесь отличий от того, что мы видели при работе с консольными приложениями, нет.

Некоторые файлы проекта

Язык VC++ в момент создания проекта (приложения) создает массу различных файлов. Но не все файлы, которые создает среда разработки, включаются в тот или иной проект. Это зависит от типа создаваемого проекта и от тех опций, которые вы выбираете, когда пользуетесь Мастером создания проекта. Изучая предыдущий материал, вы, наверное, заметили, что при создании проекта только отвечали на вопросы, делая тот или иной выбор. Это работала специальная программа, которая и называется Мастером.

В табл. 10.1 приведены собственно проектные файлы. Следует отметить, что не все проектные файлы отражены в **Solution Explorer** (точнее сказать, там отражен только самый минимум). Если перевести проект в режим **Show All Files** (показать все файлы) с помощью аналогичной опции из подменю **Project** главного меню среды, то в **Solution Explorer** файлы покажутся все равно не все: некоторые файлы будут скрыты даже тогда, когда проект находится в режиме **Show All Files.**

Таблица 10.1. *Проектные файлы*

Имя файла	**Описание**
Form1.h	h-файл, содержащий описание формы и всех ее компонентов
app.rc	Файл ресурсов проекта, записанный в виде сценария, который в зависимости от типа проекта содержит описание диалоговых окон, панелей инструментов, пиктограмм, версии проекта и др.
app.ico	Здесь хранится пиктограмма программы
resource.h	Этот заголовочный файл содержит определения ресурсов, используемых в проекте, сгенерированных из файла app.rc
stdafx.h	Этот файл используется для построения предкомпиляционного заголовочного файла и предкомпиляционных объектных файлов
AssemblyInfo.cpp	Этот файл содержит информацию по сборке проекта (файлы, ресурсы, типы и т. п.)
stdafx.cpp	Здесь содержится информация для создания предкомпиляционных файлов
Имя проекта.cpp	Содержит обычную команду `int main()`, с которой мы встречались при изучении C++, оператор `include`, подключающий h-файл, содержащий описание формы и всех ее компонентов. То есть это программа проекта на языке C++
Имя Решения.sln	Этот файл относится к категории группы проектов, объединенных в одно Решение. Он организует все элементы проекта (проектов) в одно общее Решение
Projname.vcproj	Это главный файл проекта для приложений VC++, генерируемых с использованием Мастера Приложений — программы, формирующей приложения на основании диалога с пользователем, создающим приложение. Он содержит информацию о версии среды разработки, платформе, на которой создается приложение, и свойствах созданного проекта
Readme.txt	В этом файле описываются некоторые файлы созданного проекта (рис. 10.6)

Уточним, что файлы stdax.h, stdafx.cpp используются для создания предварительно компилируемого заголовочного файла Projname.pch и объектного файла stdafx.obj.

```
=========================================================================
    APPLICATION : 10.1-2011 Project Overview
=========================================================================

AppWizard has created this 10.1-2011 Application for you.

This file contains a summary of what you will find in each of the files that
make up your 10.1-2011 application.

10.1-2011.vcxproj
    This is the main project file for VC++ projects generated using an Application Wizard.
    It contains information about the version of Visual C++ that generated the file, and
    information about the platforms, configurations, and project features selected with the
    Application Wizard.

10.1-2011.vcxproj.filters
    This is the filters file for VC++ projects generated using an Application Wizard.
    It contains information about the association between the files in your project
    and the filters. This association is used in the IDE to show grouping of files with
    similar extensions under a specific node (for e.g. ".cpp" files are associated with the
    "Source Files" filter).

10.1-2011.cpp
    This is the main application source file.
    Contains the code to display the form.

Form1.h
    Contains the implementation of your form class and InitializeComponent() function.

AssemblyInfo.cpp
    Contains custom attributes for modifying assembly metadata.

/////////////////////////////////////////////////////////////////////////////
Other standard files:

StdAfx.h, StdAfx.cpp
    These files are used to build a precompiled header (PCH) file
    named 10.1-2011.pch and a precompiled types file named StdAfx.obj.

/////////////////////////////////////////////////////////////////////////////
```

Рис. 10.6. Содержание файла Readme.txt

Окно сведений об объекте

Любой объект, находящийся на рабочем столе, обладает какими-то свойствами. Например, файл из **Solution Explorer** имеет такое свойство, как путь к папке, в которой он расположен, компонент из палитры компонентов (например, кнопка) характеризуется своим именем, видимостью в момент исполнения, координатами в форме и т. д.

Все свойства активного объекта (т. е. того, на котором в данный момент произведен щелчок мыши) мгновенно отражаются в специальном окне под названием **Properties** (свойства). Это окно может быть видимо или нет. Сделать окно видимым можно, открыв контекстное меню любого объекта в форме (или самой формы) и выполнив в нем опцию **Properties**. Вид этого элемента рабочего стола показан на рис. 10.7.

В окне имеется 5 вкладок:

- — упорядочивает содержимое окна по категориям информации;
- — упорядочивает содержимое окна по алфавиту;
- — показывает перечень свойств (это видно на рисунке);
- — показывает перечень событий, связанных с объектом;
- — выводит на экран так называемые "страницы свойства".

Рис. 10.7. Окно сведений об объекте

Вкладка *Events*

Вкладка **Events** содержит список возможных событий, которые могут происходить с компонентом. Она позволяет связывать каждое событие с программой-обработчиком этого события. Если для активного компонента щелкнуть мышью на вкладке **Events**, то появится окно со списком событий для активного компонента. Из списка выбирается то событие, которое требуется обработать. Например, нажатие кнопки, помещенной в форму (т. е. щелчок мыши на кнопке). После определения события следует дважды щелкнуть мышью в поле рядом с именем события. При этом VC++ создаст в модуле программы 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 `OnClick` кнопки `Button1` будет таким:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
}
```

Вид окна **Properties** объекта для этого случая показан на рис. 10.8.

Рис. 10.8. Вид окна **Properties** объекта с событием `OnClick`

Событие `OnClick` означает, что произошла реакция на щелчок мыши на кнопке. В созданной заготовке программы-обработчика вы можете написать реакцию объекта `Button1` на событие `OnClick`. Допустим, мы хотим, чтобы реакция на нажатие кнопки была такой: в форме должен быть нарисован красный круг (это уже будет делаться с помощью другого компонента).

При создании обработчика идет автоматическое переключение системы на вызов редактора кода, который установит курсор на начало обработчика, чтобы вы могли вводить команды программы. Кстати, создать обработчик события для кнопки можно было бы более простым способом — дважды щелкнуть на кнопке.

В окне **Properties** для формы ниже слова **Properties** справа расположена кнопка раскрывающегося (выпадающего) списка (с треугольной стрелкой). В этом списке находятся не только имена всех компонентов, помещенных в форму, но и имя активной в данный момент формы. 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 помеченным (т. е. активным), и в окне **Properties** появятся данные уже по этому (активному) компоненту. Таким образом, в списке можно легко переключаться с одного компонента на другой.

Заранее скажем, что приложение может содержать и более одной формы. Для каждой из них создается свой программный модуль, имя которого высвечивается на вкладке страниц проекта. Вкладки расположены в заголовочной части окна, в которое помещается форма. Переключая вкладки модулей, мы тем самым переключаемся с одной формы на другую.

Вкладка *Property Pages*

Это интерфейсный элемент (т. е. элемент, обеспечивающий общение между средой разработки и пользователем), который позволяет задавать установки элементов проекта. Когда мы дважды щелкаем на этой вкладке (если она доступна для данного объекта: для некоторых объектов эта страница блокирована), то открывается диалоговое окно, в котором можно задавать определенные настройки.

Чтобы открыть диалоговое окно для проекта, проще войти в опцию **Project** главного меню рабочего стола и выполнить в выпавшем меню команду **Properties**. Диалоговое окно вкладки **Property Pages** показано на рис. 10.9.

Рис. 10.9. Диалоговое окно для настройки свойств проекта

Работа с окном сведений об объекте

Ширину столбцов в окне **Properties** можно менять, перетаскивая мышью разделительные линии (как и само окно можно перетаскивать в любое место экрана по обычным правилам перетаскивания окон в Windows). Окно **Properties** имеет свое контекстное меню, которое, как и любое контекстное меню Windows, открывается с помощью щелчка правой кнопки мыши (предварительно надо поместить курсор мыши в поле окна) и содержит такие же команды, относящиеся к перемещению

этого документа, как и любое другое окно. То есть это окно может перемещаться, захватываться другим окном, само захватывать другое окно.

Если вам неудобно работать с документами, постоянно перекрывающимися другими окнами (когда приходится какой-то документ постоянно перемещать в удобное для обозрения место экрана), то лучше прикрепить его к общему документу проекта (к главному окну рабочего стола) вкладкой и работать с ним, переключаясь на его вкладку.

Но если вам требуется обращаться к документу во время работы с другими вкладками проекта, то это свойство должно быть отключено — и документ тогда будет постоянно на виду.

Выбор опции **Hide** контекстного меню делает документ невидимым на экране. Чтобы снова увидеть документ (в частности, его свойства), надо открыть контекстное меню (например, формы) и выбрать команду **Properties**. Для документа **Solution Explorer** надо выполнить команду с таким же названием в опции **View** главного меню проекта.

Редактор кода, h-модуль и режим дизайна (проектирования). Указатель *this*

Под редактором кода мы понимаем программное средство, обеспечивающее работу с текстом программного модуля. Когда говорят "программный код" или "код программы", имеют в виду программу, написанную на каком-либо языке программирования, т. е. текст, закодированный на конкретном языке. Часто можно встретить эквивалентное понятие "текст программы".

Когда открывается новая форма, к ней создается специальный программный модуль с именем формы и расширением h. В этом модуле находится описание формы и здесь же располагаются обработчики событий компонентов, участвующих в проекте. Попасть в программный модуль после загрузки проекта (когда на экране появится форма) можно с помощью комбинации клавиш <Ctrl>+<Alt>+<0> (нажмите вместе первые две клавиши и, не отпуская их, нажмите клавишу <0> на основной клавиатуре). Вы попадете в окно редактора кода, позволяющего набирать команды в будущих обработчиках событий и вообще изменять содержимое h-модуля. Можно воспользоваться и командой **Code** в опции **View**. На документе, у которого в начале была вкладка **Form1[Design]**, появится новая вкладка **Form1.h**. Имея эти две вкладки, можно переключаться из одного режима проектирования в другой. Из режима работы с модулем в режим дизайна можно переключиться комбинацией клавиш <Shift>+<F7>.

Посмотрим на h-файл. Интересно, как среда VC++ формирует программу-приложение. Главным 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 наследником

класса `Form`, чтобы в нее в результате наследования попали все члены класса `Form`, а с другой стороны, она впоследствии должна вместить другие компоненты, из которых будет строиться приложение. Кроме того, приложение может содержать несколько форм с компонентами, и этот факт надо учитывать.

Разработчики VC++ вышли из этого положения следующим образом. Первая форма, вставляемая в проект, получает имя **Form1**. Это делается для того, чтобы отразить тот факт, что вставляемая в проект форма будет связана с наследником класса `Form`. Вторая и последующие формы добавляются к проекту с помощью выполнения опций **Project | Add New Item**, в результате чего появляется диалоговое окно (рис. 10.10), где мы и задаем самостоятельно имя новой формы (в нашем случае задано имя Form2.h).

Вид рабочего стола после подключения второй формы показан на рис. 10.11.

Рис. 10.10. Диалоговое окно для подключения к проекту новой формы

Если посмотреть внимательно на программный модуль второй формы, то форма 2 построена тоже как класс-наследник класса `Form` по тому же принципу, что и форма 1.

Отметим, что все обработчики событий формируются в рамках одного класса-формы. Поэтому и обращение к любым членам класса будет идти через указатель на экземпляр этого класса. В среде VC++ это указатель `this`. Он всегда содержит ссылку на текущий объект. Если нам надо обратиться, например, к свойству `Visible` метки `Label1`, помещенной в форму, то это обращение будет выглядеть так (мы находимся в объекте `Label1`, адрес которого помещен в указатель `this`):

```
this->Label1->Visible=true;
```

Рис. 10.11. Вид рабочего стола после подключения второй формы

Это говорит о том, что имена экземпляров компонентов (`Label1` — это имя экземпляра класса `Label`) формируются как указатели на экземпляры.

Контекстное меню редактора кода

У редактора кода программы, как и у каждого объекта, есть свое контекстное меню, которое, как и все контекстные меню, открывается правой кнопкой мыши. Содержимое контекстного меню может изменяться в зависимости от места расположения курсора в тексте, выделения блока строк и т. д. Вариант контекстного меню редактора кода показан на рис. 10.12.

Некоторые команды из этого меню:

- **View Designer** — открывает окно дизайна, в котором находится форма и ее компоненты;
- **Go To Declaration** — если вы установили курсор на строке с элементом `Button1`, то после выбора этой опции курсор установится на строке с объявлением экземпляра `Button1` класса `Button` (рис. 10.13).
- **Find All References** — если вы установили курсор на строке с элементом `Button1`, то после выбора этой опции в окне вывода информации, расположенном под окном редактора кода, появится перечень ссылок на строки, в которых упоминается `Button` (рис. 10.14). Если дважды щелкнуть на какой-то строке этого перечня, курсор установится на соответствующей строке текста в поле редактора.

Рис. 10.12. Контекстное меню редактора кода

Рис. 10.13. Результат выполнения опции **Go To Declaration**

Рис. 10.14. Формирование ссылок на элемент текста в поле редактора кода

- **Outlining** — использование этой опции позволяет свертывать/развертывать код модуля, переключаться между этими режимами, делать выбранный участок невидимым и др.

Суфлер кода (подсказчик)

Этот инструмент среды проектирования помогает быстрее и правильнее набирать код программы — он выдает оперативную информацию (подсказку) при наборе кода.

К сожалению, в Beta-версии подсказчик не работает.

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 или указать на него мышью и нажать клавишу <Enter>, в результате чего элемент запишется в программу. Например, надо набрать:

```
this->button1->BackColor=clRed;
```

Как только мы наберем `this->`, сразу же откроется окно подсказчика, в котором будут все члены объекта `Form1`. Выберем код `Button1` (мы только начинаем набирать этот код, как тут же появляются строки, соответствующие набранным символам, что ускоряет процесс поиска в окне).

Когда наберем код `this->Button1`, а затем — стрелку вправо, то после стрелки вновь появится окно подсказчика с перечнем свойств, событий. Суфлер покажет список всех элементов класса `Button`.

Если записанному элементу потребуется присвоить еще какие-либо свойства, то нужно воспользоваться комбинацией клавиш <Ctrl>+<пробел>. Справа от знака присвоения высветится перечень элементов, разрешенных к записи. Здесь нужно выбрать требуемый элемент и нажать клавишу <Enter>.

Настройка редактора кода

Управление окнами редактора

Вы можете работать с кодом программы в нескольких местах одновременно. Чтобы этого добиться, надо разделить окно редактора, открыв одновременно необходимое количество экземпляров окон. Один экземпляр окна может быть разбит на два отдельных окна для более удобного редактирования. Чтобы раздвоить окно, надо выполнить следующее:

1. Щелкнуть мышью в поле редактора, чтобы он получил фокус ввода.
2. Выполнить команду главного меню **Window | Split**.

   Редактируемое поле при этом разбивается на два окна, разделенных горизонтальной полосой. Оба образовавшихся окна можно прокручивать независимо,

чтобы просматривать и редактировать их содержимое. Любые изменения, сделанные в одном окне, тут же отображаются в другом, потому что это фактически два экземпляра одного и того же окна. Чтобы сделать одно окно длиннее или короче другого, следует потянуть мышью за разделительную полосу вверх или вниз.

Чтобы вернуться к единственному окну, надо выполнить команду **Windows | Remove Split** главного меню среды.

Вы также можете создать множество экземпляров редактора кода. Эта возможность позволяет открывать длинные документы в нескольких экземплярах редактора (чтобы просмотреть и отредактировать по отдельности различные участки полноразмерного окна редактора).

Чтобы создать новое окно, следует выполнить команду **Windows | New Window** главного меню среды проектирования. На рис. 10.15 показаны три созданных окна редактора. Все они разместились в виде вкладок на главном окне рабочего стола.

Рис. 10.15. Три окна редактора кода

Настройка опций редактора через команду *Tools* главного меню

Настройка опций редактора выполняется через диалоговое окно, которое открывается, если выполнить команду **Tools | Options** главного меню среды. В открывшемся окне в левой его части надо дважды щелкнуть мышью на опции **Text Editor** (рис. 10.16).

Рис. 10.16. Редактирование опций редактора кода

В раскрывшемся списке выбираем язык **C/C++** и справа получаем три группы установок с переключателями. Рассмотрим некоторые из них:

- **Word wrap**. Включенная опция позволяет "заворачивать" длинную строку, не умещающуюся в поле редактора, на вторую строку. При этом можно задать появление пиктограммы в конце заворачиваемой строки;
- **Line numbers**. При включенной опции строки кода будут пронумерованы.

Рис. 10.17. Вид текста в поле редактора кода после установки свойств **Word wrap** и **Line numbers**

Результат установок обоих опций показан на рис. 10.17. Видим, что справа от текста появилась нумерация строк, а в самом тексте длинная строка текста стала переноситься на другую строку.

Изменение шрифта и цвета

Для изменения шрифта редактора следует выполнить команду главного меню **Tools | Options**, в открывшемся диалоговом окне дважды щелкнуть мышью на папке **Environment** и в появившемся ниже списке выбрать **Fonts and Colors** (рис. 10.18).

Рис. 10.18. Изменение шрифта редактора

Для изменения шрифта следует выбрать соответствующий элемент редактирования, для которого требуется изменить шрифт, и подобрать ему шрифт с использованием выпадающих списков **Font** и **Size**. А цвет шрифта выбирается посредством выпадающих списков **Item foreground** (элемент переднего плана) и **Item background** (элемент фона) или после нажатия кнопки **Custom**, когда появляется палитра цветов для выбора необходимого цвета.

Начало редактирования кода программного модуля

Чтобы начать редактировать код модуля, надо открыть вкладку с именем требуемого модуля на рабочем столе, поместить курсор редактора в нужное место кода и редактировать его по правилам стандартных текстовых редакторов Windows. Код можно перетаскивать (с помощью мыши), если предварительно его выделить.

Примечание

Чтобы поместить курсор редактора в нужное место кода, можно щелкнуть в этом месте мышью либо воспользоваться стрелками клавиатуры или клавишами <Home>, <End>, <Ctrl>+<Home>, <Ctrl>+<End>, <Page Up>, <Page Down>.

Компоненты среды программирования VC++

Среда программирования VC++ содержит множество компонентов, обеспечивающих решение различного рода практических задач: формы, панели, кнопки, метки, меню, компонент для выхода в Интернет, контейнеры для изображений и текста и т. д. Но главным среди всех компонентов, базовым компонентом является компонент "Форма" (Form), в который помещаются остальные компоненты, определяющие функциональность создаваемого приложения. Рассмотрим основной компонент среды программирования — класс `Form`.

Класс *Form*

Класс `Form` определяет элемент будущего приложения, называемый формой. Это контейнер, в который помещаются остальные элементы приложения, определяющие впоследствии всю функциональность данного приложения. Естественно, что для работы с формой должен существовать некий инструмент. В этой среде он существует и называется дизайнером форм.

Дизайнер форм

Пустая форма, которая появляется на экране после загрузки среды VC++, или когда мы создаем новое приложение, фактически представляет собой окно дизайнера форм. Дизайнер позволяет работать с формой: помещать компоненты в форму, удалять и выделять компоненты, перетаскивать их с одного места на другое, закрывать форму и т. д.

Компоненты выбираются из палитры компонентов, расположенной на вкладке **Toolbox**, и появляются в ней при загрузке формы. Многозначные компоненты разбиты на функциональные группы, которые могут сворачиваться и разворачиваться, если щелкать на треугольнике слева от названия группы (рис. 10.19).

Содержимое палитры компонентов регулируется списком, который открывается после выполнения команды **Tools | Choose Toolbox Items** (рис. 10.20 и 10.21). Каждая строка списка снабжена слева квадратным окошком, в котором можно щелкать мышью. При этом если окошко было пустым (это означает, что компонент, указанный в данной строке, не подключен к палитре компонентов), то после щелчка в нем появляется галочка (это означает, что компонент, указанный в данной строке, будет подключен к палитре компонентов), а если окошко содержало галочку, она исчезнет. Это означает, что компонент, указанный в данной строке, будет выведен из палитры компонентов. Поставляемый продукт содержит минимальное количество компонентов. Остальные вы должны сами подключить к палитре.

Рис. 10.19. Палитра компонентов

Рис. 10.20. Регулирование палитры компонентов. Часть 1

Рис. 10.21. Регулирование палитры компонентов. Часть 2

Помещение компонента в форму

Поместить компонент в форму можно одним из следующих способов:

- найти нужный компонент в палитре 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 — по его периметру (или в точке периметра) появятся квадратики. 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.

Другие действия с дизайнером форм

Кроме помещения компонента в форму, при конструировании приложения требуется совершать и другие действия, которые сейчас рассмотрим.

- Закрыть все проекты данного решения можно с помощью команд меню **File | Close Solution**, открыть — с помощью **File | Recent Projects and Solutions**. В открывшемся списке надо выбрать последний проект, с которым вы работали.

  Можно открыть все проекты и другим путем: выполнить команду **File | Open | Project/Solution** и далее в появившемся диалоговом окне выбрать папку, где

хранится файл проекта. Файл имеет вид 10.1-2011, где после пиктограммы идет имя проекта.

Если необходимо закрыть проект не целиком, а только одну его форму, то это можно сделать, выполнив команду **File | Close**, а снова открыть — командами **File | Recent Files**, после чего из выпадающего списка выбрать строку с именем нужной формы (например, **Form1.h**).

- Выделить несколько компонентов (например, если требуется их групповое перемещение в другое место формы) можно следующими способами:
  - при нажатой клавише <Shift> щелкать мышью на требуемых компонентах;
  - 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 кнопку мыши, можно, установив курсор мыши на любой из отмеченных объектов, потянуть за него в нужном направлении и вся отмеченная ранее группа переместится вместе с этим объектом.

  Снять групповое выделение компонентов можно щелчком мыши вне выделенного пространства.
- Можно изменять форму выделенного компонента на более мелком уровне, чем растяжка за точки выделения. 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

Некоторые действия над формой и ее содержимым можно выполнять через ее контекстное меню. Чтобы оно появилось, достаточно нажать правую кнопку мыши. Рассмотрим команды контекстного меню формы.

- **View Cod** — открывает файл модуля формы (имя формы.h), т. е. переводит работу из режима дизайна формы в режим работы с кодом модуля формы. Этого, кстати, можно добиться, дважды щелкнув на форме.
- **Lock Controls** — блокирует любой компонент, имеющий данное свойство (в том числе и форму).

  У компонента в его верхнем левом углу появляется пиктограмма замка, после чего компонент закрыт для разного рода действий с ним. Например, форму уже нельзя растягивать, а компоненты, помещенные в нее, обретают это же свойство — при их активизации у них тоже появляется пиктограмма замка. Такие компоненты теперь уже не перемещаются в форме и не растягиваются/сжимаются (т. е. они заблокированы). Снять блокировку можно повторным выполнением этой же опции.

- **Properties** — эта опция открывает окно **Properties** свойств объекта (в данном случае — формы).

Примечание

Если вы вдруг по каким-то причинам "потеряли" на экране дизайнер формы или код ее модуля, их легко восстановить с помощью информации из окна **Solution Explorer**: откройте контекстное меню h-файла формы и в нем увидите команды **View Code**, **View Designer**.

Добавление новых форм к проекту

Алгоритмы решения некоторых задач требуют, чтобы компоненты, обеспечивающие решение таких задач, размещались в разных формах, и чтобы эти формы вызывались в процессе выполнения проекта в определенном порядке. Например, возьмем задачу управления кадрами. Желательно, чтобы все справочные данные по кадрам были размещены отдельно от аналитических форм — так удобнее поддерживать справочную информацию (этим вопросом должен заниматься отдельный работник, которому нечего делать в разделе аналитических форм). Да и работнику, связанному с анализом кадров, легче решать свои проблемы, не заботясь о поддержке в активном состоянии нормативно-справочной информации.

Как добавить к проекту новую форму? Этот вопрос нами рассматривался выше.

Как выбрать, какую форму запускать первой?

Когда вы добавляете форму к проекту, она не станет по умолчанию сама выводиться в момент исполнения приложения — ее надо вызвать.

Форма, которая появляется в проекте, когда мы выбираем режим **New Project**, станет запускаться первой по умолчанию, т. е. всегда считается главной, стартовой.

А как быть, если мы имеем несколько форм и хотим одну из них (не обязательно главную) запустить первой, а потом уже запускать в заданном порядке остальные формы? В версии VC++ 2005 такая возможность была: чтобы установить приоритет запуска форм, надо было:

1. Открыть контекстное меню проекта в окне **Solution Explorer** и выполнить опцию **Properties**. Откроется диалоговое окно **Property Pages**.
2. В этом окне слева надо выбрать папку **General** и в окне справа из выпадающего меню **Startup Object** выбрать стартовую форму.

Однако в версиях 2008 Express Edition и 2010 beta2, 2011 beta автор не обнаружил такой возможности. Поэтому приходится считать, что главной все-таки будет первая форма проекта, из которой следует организовать вызов остальных форм.

Но есть выход: в **Solution Explorer** дважды щелкните на строке **Source Files | Имя_проекта.cpp**. Откроется cpp-файл стартовой формы. В нем найдите строку `Application::Run(gcnew Form1());`, которая запускает форму 1, и в ней замените `Form1` на `Form2`, если хотите, чтобы стартовой была форма 2: `Application::Run(gcnew Form2());`. Перекомпилируйте проект и запускайте на выполнение. Увидите, что первой запустилась форма 2.

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. Но мы увидели, как можно стартовой формой сделать любую другую. Если в проекте много форм, то из главной формы можно организовать вызов на выполнение остальных форм (а можно из каждой формы вызвать любую другую). Эта задача решается с помощью обработчиков кнопок вызова конкретных форм.

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 просто переключиться на другую вкладку, отражающую данную форму в режиме дизайна. Например, если у нас две формы и вкладки, соответственно, помечены как **Form1.h[Design]** и **Form2.h[Design]**, то для вызова на экран 2-й формы надо щелкнуть на вкладке **Form2.h[Design]**.

Вызов формы на выполнение

Форма вызывается на выполнение в двух режимах: модальном и немодальном (обычном). Вызов на выполнение осуществляется двумя разными командами. В модальном режиме — методом формы `ShowDialog()`, в обычном — методом формы `Show()`. Детальное описание обоих режимов вызова будет пояснено далее в *разд. "Некоторые методы формы" этой главы*.

Свойства формы

Эти свойства могут встречаться у многих компонентов, которые будут изучаться далее, поэтому впоследствии будем говорить о них кратко. Перечень свойств формы, отображенных в окне **Properties**, приведен на рис. 10.22.

Рассмотрим некоторые свойства формы.

- `ApplicationSettings` — установки приложения.

  Эта возможность типа Windows Form позволяет легче создавать, хранить и поддерживать приложения и различные преференции пользователя. С установками, о которых речь пойдет чуть позже, вы сможете не только хранить данные приложения (такие как строки соединения с базой данных), но и специфические данные пользователя, в частности его преференции (специфические установки параметров). Вы сможете создавать новые установки, читать и писать их, привязывать к свойствам ваших форм и определять их до загрузки и сохранения приложения. Фактически аппарат установок приложения — это аппарат параметризации этого приложения, что всегда дает возможность перестройки его без перекомпиляции.

  Так что же такое эти установки? Ваши приложения Windows Form почти всегда содержат такие данные, которые не желательно хранить в самом приложении, т. к. их изменение влечет за собой перекомпиляцию. Если ваше приложение

эксплуатируется не вами, а другим пользователем, то перекомпиляция из-за какой-то мелочи может повлечь за собой его неработоспособность. Ведь пользователь не знает вашей программы и не сможет ее правильно изменить (подстроить), даже если у него имеется код программы, что весьма сомнительно (обычно разработчики поставляют пользователю только исполняемые модули без кода).

Рис. 10.22. Перечень свойств формы, отображенных в окне **Properties**

Следовательно, такую настроечную информацию желательно хранить в отдельном файле, который подключается к приложению. Если смотреть на этот процесс с точки зрения работы приложения в рамках системы "клиент-сервер", то аппарат установок дает возможность хранить на компьютере клиента как установки клиента, так и установки приложения. В частности, вы можете определить установки (т. е. значения заданных величин, влияющих на ход выполнения программы-приложения) для данного свойства, указывая его имя, тип данного и контекст (это создается для приложения или для пользователя). Такие установки не изменяются программой и читаются ею в память автоматически в момент исполнения программы-приложения.

В большинстве случаев программные установки имеют статус "только для чтения — read only", это программная информация и нет необходимости ее изме-

нять в самой программе. И наоборот, пользовательские установки могут и читаться, и быть безопасно измененными в момент исполнения программы.

Установки приложения сохраняются в двух XML-файлах: в файле app.config (app — это имя exe-модуля приложения), который создается во время работы дизайнера форм, т. е. во время проектирования приложения. Формирование этого файла происходит в момент создания первой установки приложения. Другой файл — user.config (этот файл формируется в момент исполнения приложения — когда пользователь изменяет значение любой пользовательской установки).

Установки определяются дизайнером форм заданием свойства `ApplicationSettings` в окне **Properties** формы. Когда вы определяете установки, среда программирования автоматически создает специальный управляемый пользователем класс-оболочку, в котором каждая установка связывается со свойством класса.

Как осуществить установки с помощью дизайнера форм? В следующем порядке действий вы выполняете конфигурацию установок и связей, используя редактор свойства для Windows Form. Когда вы используете этот редактор, то среда программирования генерирует класс-оболочку, происходящий от класса `ApplicationSettingsBase`, и определяет все ваши установки в качестве свойств класса-оболочки. Чтобы создать новые установки приложения, выберите форму или компонент, чьи свойства хотите связать с новыми установками. Далее требуется:

1. Добавить в проект файл app.config (**Add | New Item | Configuration File**), а затем переименовать его в app.settings. Раскрыть свойство `ApplicationSettings` (если слева от него стоит знак "+") и щелкнуть мышью на кнопке с многоточием на подчиненном свойстве `PropertyBindings`. Откроется диалоговое окно, показанное на рис. 10.23.

Рис. 10.23. Диалоговое окно для задания установок приложения

2. В диалоговом окне надо открыть выпадающее меню для того свойства (кнопка с галочкой справа), которое требуется связать с объектом (для нашего рисунка — это `Font`). В открывшемся списке выполняем команду **New**. В результате откроется окно **New Application Setting**, в котором и следует задать установки, т. е. значения указанных в нем элементов. Во-первых, надо задать имя установки, ее значение по умолчанию, ее предназначение (для приложения или для пользователя). Если установка предназначена для приложения, то это свойство станет глобальным для всех пользователей приложения и не сможет изменяться в режиме исполнения. Если же контекст выбран `User` (пользователь), то свойство будет иметь статус `read/write` (его можно не только читать, но и записывать, т. е. изменять). В нашем случае сделана установка шрифта по умолчанию (рис. 10.24).
3. Щелкнуть на кнопке **OK** последовательно в обоих диалоговых окнах.

Рис. 10.24. Установка шрифта по умолчанию

Пример задания установки для свойства `Font` (шрифт) показан на рис. 10.25.

- `AccessibleDescription` — в этом свойстве отражается (или задается) текст описания компонента для приложений, имеющих к нему доступ.

  Служит для доступа клиентских приложений. Это свойство необходимо для случаев, когда описание объекта неочевидно (кнопка с надписью **OK** не нуждается в дополнительной информации, но кнопка, на которой изображен, например, рисунок верблюда, требует пояснений).

  Свойства `AccessibleName` и `AccessibleRole` для такой кнопки опишут ее назначение, а рассматриваемое свойство `AccessibleDescription` выразит саму идею этой кнопки. При этом `AccessibleName` задает имя компонента для приложений, имеющих к нему доступ, а `AccessibleRole` (выбирается из выпадающего списка)

содержит тип элемента пользовательского интерфейса (`ComboBox`, `DropList` и т. п.). Если конкретно это свойство не может быть определено, то оно устанавливается в значение `Default`.

Рис. 10.25. Пример задания установки для свойства `Font`

- `AllowDrop` — свойство определяет возможность вывода у себя данных, когда пользователь перемещает их над этим компонентом.

  Например, если компонент `A` имеет такую возможность, то когда вы начнете перемещать мышью некоторое изображение над ним, это изображение выведется на этом компоненте, когда будет отпущена кнопка мыши (изображение "капнет" на поверхность компонента, например, формы).

- `AutoScaleMode` — это свойство задает так называемое автоматическое масштабирование.

  Автоматическое масштабирование дает возможность форме и расположенным в ней компонентам, сконструированным на одном компьютере с некоторым разрешением вывода (т. е. с фиксированным значением числа точек изображения, которое станет выводиться на единицу поверхности вывода) или с определенным шрифтом, выводиться, соответственно, на другом компьютере с другим разрешением или шрифтом.

  Например, элементы пользовательского интерфейса, содержащие текст, такие как заголовочные полосы, меню и др., полностью зависят от используемого шрифта и могут быть искажены. Это же касается и экранного разрешения. Большинство дисплеев имеет разрешающую способность в 96 точек на дюйм. Но уже существуют дисплеи с 120, 133, 170 и более высокими значениями DPI (количество точек на один дюйм), и поэтому здесь также возможно искажение экранных изображений на разных компьютерах, если не отрегулировать масштабирование.

  Значение свойства выбирается из выпадающего списка, в котором определяется автоматический режим масштабирования для данного компонента.

Например, если выбрать значение `Font` (шрифт), то автоматическое изменение шрифта будет полезным тогда, когда вы хотите, чтобы форма или компонент растягивались или сокращались в соответствии с размерами шрифта в операционной системе и использовались в случаях, когда их абсолютные размеры не играют роли.

Если выбрано значение `DPI`, то эта величина полезна, если вы хотите изменять размеры формы или компонента относительно экрана. Например, вы выводите на экран диаграмму или другую графику и хотите, чтобы это изображение всегда занимало определенный процент экрана.

Если выбрать значение свойства, равное `Inherit`, то другой компьютер станет наследовать шрифт и разрешение базового компьютера.

- `AutoScroll` — это свойство задает возможность автоскроллинга компонента, т. е. появление полос прокрутки, если из-за изменения его размера появляются такие компоненты, которые полностью не видны на данном компоненте (в частности, на форме).
- `AutoScrollMargin` — свойство задает возможность автоскроллинга компонента, но с учетом отступов по ширине и высоте от внутреннего компонента, вызывающего скроллинг.

  Например, вы хотите, чтобы полосы прокрутки появлялись не тогда, когда при изменении формы ее нижний край достигает края самого нижнего ее компонента, а, например, не доходя (по высоте) 300 пикселов. Тогда подчиненное свойство `Height` надо установить на значение 300. Реакция формы на задание этого свойства показана на рис. 10.26 — на правом рисунке отступ до изображения без полос прокрутки больше, чем на левом рисунке.

Рис. 10.26. Результат задания свойства `AutoScrollMargin`

- `AutoScrollMinSize` — свойство задает минимальный размер (в пикселах) ширины и высоты полос прокрутки.
- `AutoSizeMode` — свойство задает способ автоматического изменения формой своих размеров. Значение выбирается из выпадающего списка. На рис. 10.27 показаны варианты изменения формы в зависимости от совместных значений данного свойства и свойства `AutoSize`.

Рис. 10.27. Демонстрация свойства `AutoSizeMode`

- `BackColor` — свойство дает возможность выбора из выпадающего списка цвета фона компонента.

  Например, если выбрать для кнопки синий цвет, то все ее поле будет закрашено синим.
- `BackgroundImage` — дает возможность выбора изображения, которое станет фоновым в компоненте.

Например, если выбрать для кнопки ваше фото, то оно появится по всему полю кнопки.

- `BackgroundImageLayout` — задает тип размещения фонового изображения в компоненте: подгоняется под размер окна компонента, разрешает zoom-действие (масштабирование) и т. д. Значение свойства выбирается из выпадающего списка.
- `CancelButton` — это свойство позволяет создавать имитацию нажатия кнопки с помощью нажатия клавиши <Esc> в момент работы приложения.

  Свойство удобно использовать для обеспечения быстроты работы с приложением (просто клавишу удобнее и быстрее нажимать, чем разыскивать кнопку в окне и щелкать на ней мышью). При нажатии <Esc> приложение выполняет такое же действие, как будто вы щелкнули на кнопке мышью. Свойство не станет работать, если другой компонент в форме работает с клавишей <Esc>.
- `CausesValidation` — включает/выключает необходимость проверки на достоверность компонента во время получения им фокуса ввода (т. е. когда компонент становится активным).

  На самом деле это свойство подавляет или не подавляет возникновение события `Validating`. Если это событие не подавлено, то в его обработчике можно проверить на достоверность некоторые данные, когда компонент, содержащий свойство `CausesValidation` и событие `Validating`, получает фокус ввода. Например, в компоненте находится адрес электронной почты. Когда компонент становится активным, можно проверить, содержит ли адрес электронной почты символ @, обязательный для такого адреса.
- `ContextMenuStrip` — через это свойство к компоненту подключается его контекстное меню. Меню (компонент `ContextMenuStrip`) должно быть определено в форме, и тогда оно станет "видимо" в этом свойстве.

  Контекстное меню представляет собой меню, которое выводится, когда пользователь нажимает правую кнопку мыши во время нахождения курсора мыши над компонентом. Например, можно задать такое меню для компонента, через который вводится текст, с тем, чтобы иметь возможность в момент ввода изменять шрифт, искать фрагменты текста или иметь возможность копирования текста в буфер системы, вставки его оттуда в необходимое место другого текста, и т. д.
- `ControlBox` — свойство предоставляет возможность вывода в различном виде заголовочной полосы формы: если свойство установлено в `false`, то в заголовочной полосе кнопки не выводятся.
- `Cursor` — задает путем выбора из выпадающего списка форму курсора мыши, когда он появляется над формой.
- `DoubleBuffered` — свойство задает возможность снижения мерцания изображения компонента при его перерисовке за счет использования дополнительного буфера памяти.
- `Enabled` — свойство задает право доступа к компоненту: значение `true` означает, что доступ разрешен, `false` — запрещен. В случае с формой значение свойства,

равное `false`, приведет к блокировке 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 щелкнуть на кнопке с многоточием в поле свойства, после чего откроется диалоговое окно выбора характеристик шрифта. На рис. 10.28 видна новая установка шрифта, которую мы выполнили, работая со свойством `ApplicationSettings`.

Рис. 10.28. Появление нового значения свойства `Font` после пользовательской установки

- `ForeColor` — это свойство задает цвет переднего плана компонента. Цвет можно выбрать из раскрывающегося списка, который появится, если нажать на кнопку в поле этого свойства.
- `FormBorderStyle` — задает стиль окантовки формы, который выбирается из выпадающего списка. По умолчанию принято значение `Sizable` (форма может изменять свои размеры в режиме исполнения: откомпилируйте приложение с формой и попробуйте потянуть мышью за стороны формы — форма растянется/сузится). Другие значения этого свойства не допускают такой "вольности".
- `HelpButton` — задает возможность вывода кнопки помощи в заголовке компонента. Эта кнопка (с вопросительным знаком в ее поле) появляется в заголовке формы слева от кнопки (закрыть форму) и может служить средством вывода сообщения пользователю, имеющего характер помощи (если вы, конечно, сделаете обработку этой кнопки). Обработка кнопки состоит в создании обработчика события `HelpRequested` формы.

  Как создавать обработчики событий, мы рассмотрим позже. Однако следует помнить, что кнопка помощи появится в заголовке формы при условии, что кнопки (это свойства формы) `MaximizeBox` и `MinimizeBox` будут отключены (т. е. в заголовочной полосе формы не будет кнопок свертывания и развертывания окна формы (рис. 10.29)). Кроме того, после создания обработчика кнопка реагирует только на нажатие клавиши <F1>, т. е. обычной клавиши помощи. Код обработчика события `HelpRequested` приведен в листинге 10.1, а результат нажатия клавиши <F1> — на рис. 10.30.

Листинг 10.1

```
private: System::Void Form1_HelpRequested(System::Object^  sender,
System::Windows::Forms::HelpEventArgs^  hlpevent)
{
  MessageBox::Show("Проверка действия кнопки Help",
   "Приложение 10.1_2010,
   MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
   }
```

Рис. 10.29. Задание кнопки помощи в заголовке формы

Рис. 10.30. Проверка действия кнопки **Help**

- `Icon` — дает возможность подключения к форме пиктограммы: если нажать кнопку с многоточием в поле этого свойства, то откроется диалоговое окно для выбора пиктограммы (файла с расширением ico). Выбранная пиктограмма попадет в заголовок формы.
- `ImeMode` — подключает к компоненту (посредством выбора из выпадающего списка) редактор с различными режимами обработки входных данных Input Method Editor (IME) (это специальная программа, которая дает возможность

пользователям вводить различные нестандартные символы, например японские, с помощью стандартной клавиатуры).

- `IsMdiContainer` — свойство, показывающее, является ли форма контейнером для многодокументного интерфейса (т. е. является ли она одной из форм так называемого MDI-приложения). Мы будем строить приложения со стандартным документным интерфейсом (SDI).
- `KeyPreview` — пояснение этого свойства связано с понятием "событие формы". С большинством компонентов связаны ситуации, названные "событиями", которые происходят в момент воздействия на компонент чего-либо.

  Например, когда форма начинает изменять свои размеры, происходит событие `Resize`, когда форма начинает на экране прорисовываться, наступает событие `Paint`, когда над формой появляется курсор мыши, возникает событие `MouseMove`, и т. д.

  Чтобы отреагировать на появление события, формируют специальные участки программы, называемые обработчиками событий. В этих обработчиках программисты пишут команды на C/C++, которые и отражают реакцию на событие.

  Например, мы хотим, чтобы при щелчке мышью на форме форма покраснела (не от стыда, конечно). Для этого мы должны создать обработчик события формы с именем `Click`. Заготовка обработчика события формируется довольно просто: надо открыть вкладку **Events** в окне **Properties** (щелкнуть мышью на кнопке), выбрать нужное событие, на которое мы хотим предусмотреть реакцию (допустим, это событие `Click`, возникающее при щелчке мышью в поле формы), и дважды щелкнуть мышью в поле этого события. Среда программирования мгновенно создаст в h-файле формы заготовку: заголовок функции, которая должна будет выполниться сразу, как только наступит это событие, и пустое тело функции, в которое мы должны вписать свои команды на C/C++, отражающие алгоритм реакции на событие.

  Для нашего примера надо изменить цвет формы, а это можно сделать, если ее свойству `ForeColor` придать значение "красный цвет".

  А теперь, что касается свойства `KeyPreview`. Когда это свойство принимает значение `true`, то форма получает все события `KeyPress` (событие `KeyPress` возникает, когда нажимается несимвольная клавиша (например, управляющая), однако вместе с этим возникают и события `KeyDown` и `KeyUp`).

  Событие `KeyDown` чаще всего возникает при нажатии клавиш <Tab>, <Enter>, <Esc>, а событие `KeyUp` возникает, когда нажатую клавишу отпускают. Причем эти события наступают для формы раньше, чем они наступают для тех ее компонентов, которые в данный момент породили такие события. Например, существует компонент `TextBox`, с помощью которого вводят данные с клавиатуры. При включенном свойстве `KeyPreview` у формы, как только нажали клавишу для ввода символа в поле компонента `TextBox`, сразу возникают указанные события. И только после этого они (эти события) возникнут у самого компонента `TextBox`.

Если же свойство `KeyPreview` отключено, то при нажатии клавиш в поле `TextBox` рассмотренные события наступят только для `TextBox`.

- `Language` — позволяет выбрать из выпадающего списка язык, на котором предполагается работать с формой.
- `Localisable` — это свойство связано с настройкой приложения на различные "культуры", т. е. на специфику данных отдельных стран.
- `Locked` — включение блокировки компонента, в результате чего компонент теряет свойство переместимости или изменения размеров. При этом в левом верхнем углу компонента появляется пиктограмма замка.
- `MainMenuStrip` — через это свойство к компоненту подключается главное меню: его компонент с именем `menuStrip` надо поместить, например, в форму и он станет "виден в свойстве".
- `MaximumSize` — максимальный размер формы. Если задано `0;0`, то форма безразмерная.
- `MaximizeBox`, `MinimizeBox` — рассмотрены в свойстве `KeyPreview`.
- `Opacity` — задает уровень затемнения (прозрачности) формы. Значение задается в процентах. Чем ниже процент, тем более прозрачна форма в режиме исполнения.
- `Padding` — задает отступы внутри компонента. Чтобы лучше понять смысл этого свойства, разберем одновременно и свойство `Margin` (оно определяет пространство *вне* компонента, которое "держит" другие компоненты на заданной дистанции от границ данного компонента). Свойство `Padding` определяет пространство *внутри* компонента, которое "держит" на заданной дистанции от границ компонента содержимое компонента (например, значение его свойства `Text`). Разница между свойствами `Padding` и `Margin` показана на рис. 10.31.

Рис. 10.31. Демонстрация разницы между свойствами `Padding` и `Margin`

- `RightToLeft` — это свойство введено с учетом особенностей письменности народов, у которых текст идет справа налево (например, у арабов или евреев). Если значение установлено в `Yes`, то компоненты, содержащие текст, станут выводить его справа налево.
- `Size` — задает размеры компонента. Попробуйте растянуть форму и посмотрите, как изменятся значения этого свойства.
- `SizeGripStyle` — это свойство позволяет задать вывод/невывод калибровочной полоски (захватки) в правом нижнем углу формы (рис. 10.32).

Рис. 10.32. Демонстрация задания свойства `SizeGripStyle`

- `StartPosition` — свойство задает стартовую позицию формы в режиме исполнения приложения. Значение выбирается из выпадающего меню. Если, например, задать значение `CenterScreen`, то форма начнет выводиться по центру экрана.
- `Tag` — в этом свойстве можно задавать некоторые данные, чтобы потом ими пользоваться при решении тех или иных задач. Это нечто вроде буферной памяти, связанной с компонентом.
- `Text` — в этом свойстве задают название компонента. Например, для кнопки можно написать: "Завершение работы приложения".
- `TopMost` — свойство определяет, будет ли данная форма всегда помещаться над другой.
- `TransparencyKey` — свойство задает цвет, которым будут высвечиваться прозрачные области формы.

- `UseWaitCursor` — свойство задает, будет ли использоваться курсор в виде песочных часов (курсор ожидания) для данного компонента и всех его потомков или нет.
- `WindowState` — свойство задает состояние окна формы. Перед тем как форма выведется, это свойство всегда сохраняет значение `Normal`, несмотря на первоначальное значение данного свойства. Это состояние отражается в свойствах `Height`, `Left`, `Top` и `Width`. Если форма "прячется" после того как она была показана, то эти свойства отражают предыдущее состояние до тех пор, пока форма снова не покажется, несмотря на изменения, сделанные в свойстве `WindowState` (изменения свойств можно делать и в режиме исполнения).

События формы

События формы показаны на рис. 10.33.

Опишем некоторые из них.

- `Activated` — возникает, когда форма активизирована.
- `Click` — возникает при щелчке мышью в форме.
- `ControlAdded` — возникает, когда в форму добавлен новый компонент.
- `ControlRemoved` — возникает, когда компонент удален из формы.
- `CursorChanged` — возникает, когда у формы изменяется свойство `Cursor`.
- `DoubleClick` — возникает после двойного щелчка в форме.
- `FormClosed` — возникает после закрытия формы.
- `FormClosing` — возникает перед закрытием формы.
- `HelpButtonClicked` — возникает после щелчка на кнопке `HelpButton`.
- `HelpRequested` — возникает при нажатии клавиши <F1>.

Рис. 10.33. События формы

- `Load` — возникает перед первым выводом формы.
- `Paint` — возникает, когда форма перерисована.
- `Scroll` — возникает, когда в форме начинается прокрутка.
- `Shown` — возникает, когда форма впервые выведена.

Некоторые методы формы

Форма имеет большое количество методов, которые можно посмотреть, открыв Help и набрав в поле поиска появившегося окна **Form Members**. Рассмотрим только некоторые из методов формы.

- `Close()` — закрывает форму.

  Если закрывается главная форма, приложение закрывается. Ресурсы, занятые формой, освобождаются.

- `Hide()` — форма становится невидимой.
- `Show()` — выводит форму на экран.
- `ShowDialog()` — показывает форму в модальном режиме.

  Если форма показана в модальном режиме, то приложение не может выполняться, пока форма не будет закрыта. Чтобы закрыть форму, открытую в модальном режиме, надо назначить свойству `DialogResult` кнопку, которая должна закрыть форму (например, **OK**), и проверить это свойство на совпадение его значения с соответствующим значением такого же свойства кнопки. Дело в том, что когда метод `ShowDialog()` выполнится, то он возвратит именно это заданное значение свойства в свойство формы с тем же наименованием `DialogResult`. Это и станет сигналом того, что форма была открыта в модальном режиме и ее можно закрыть.

  Как вызывать из главной формы другие и как возвращаться в главную, видно из листинга 10.2, который показывает работу приложения с тремя формами: `Form1`, `Form2`, `Form3`. Из главной (стартовой) `Form1` вызываются остальные, причем `Form2` — в модальном режиме, а `Form3` — в обычном (немодальном). Если `Form3` можно закрывать, нажимая кнопку **Вызов Form1** или кнопку закрытия окна, и при этом все проходит успешно, то для `Form2` это не имеет места: пока вы "правильно" не закроете эту форму, нажав кнопку **Вызов Form1**, форма не закроется.

  Отметим: чтобы формы были видны одна из другой, надо в h-файл для главной формы перед командой `using namespace` поместить операторы:

  ```
  #include "Form2.h"
  #include "Form3.h"
  ```

- `Dispose()` — форма разрушается и память, занятая ею, освобождается.
- `Focus()` — делает форму активной: свойства `Visible` и `Enabled` принимают значение `true` (форма становится видимой и доступной).

Листинг 10.2

```
Файл Form1.h

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
 //открытие Form2

System::Windows::Forms::DialogResult dr;
Form2 ^newDlg = gcnew Form2();
m: dr = newDlg->ShowDialog();

/*Вызывается Form2 в модальном режиме. В dr запоминается значение DialogResult.
Когда Form2 закроется, то значение ее свойства DialogResult будет сравниваться
с dr(там перед закрытием формы мы внесем значение OK в DialogResult):*/

if( dr == System::Windows::Forms::DialogResult::OK )
    return;
else
 {
  MessageBox::Show ("Ошибка закрытия Form2");
  goto m;
 }

}
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
{
 //Вызов Form3
 Form3 ^f3 = gcnew Form3();
 f3->Show();
}
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^ e)
{
  this->Close();
}


Файл Form2.h

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{

 this->DialogResult= System::Windows::Forms::DialogResult::OK;
 this->Close();  //закрытие Form2

}
```

```
Файл Form3.h

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{
 this->Close();
}
```

Рисование графиков в форме

В заключение покажем, как в форме можно рисовать графики. Вид формы с графиками функций `sin(x)` и `tan(x)` показан на рис. 10.34. Коды обработчиков кнопок (фрагменты h-файла приложения, выводящего графики функций) — в листинге 10.3.

Рис. 10.34. Графики синусоиды и тангенсоиды, построенные в форме

Листинг 10.3

```
#pragma once

namespace My1022011
{

        using namespace System;
        using namespace System::ComponentModel;
        using namespace System::Collections;
        using namespace System::Windows::Forms;
```

```
        using namespace System::Data;
        using namespace System::Drawing;

        /// <summary>
        /// Summary for Form1
        /// </summary>
        public ref class Form1 : public System::Windows::Forms::Form
        {
        public:
                Form1(void)
                {
                        InitializeComponent();
                        //
                        //TODO: Add the constructor code here
                        //
                }

        protected:
                /// <summary>
                /// Clean up any resources being used.
                /// </summary>
                ~Form1()
                {
                 if (components)
                   {
                        delete components;
                   }
                }
        private: System::Windows::Forms::Button^  button1;
        protected:
        private: System::Windows::Forms::Button^  button2;
        private: System::Windows::Forms::Button^  button4;

        private:
                /// <summary>
                /// Required designer variable.

                /// </summary>
                System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
                /// <summary>
                /// Required method for Designer support — do not modify
                /// the contents of this method with the code editor.
//Объявление моих функций
//--sin(x)-----------------------
//математические функции берутся из класса Math
```

```
 double fs(double x)
  {
     return(Math::Floor(Math::Sin(x)));
  }
//--tan(x)----------------------
double ft(double x)
  {
      return(Math::Floor(Math::Tan(x)));
  }


                /// </summary>
                void InitializeComponent(void)
                {
                        this->button1 = (gcnew System::Windows::Forms::Button());
                        this->button2 = (gcnew System::Windows::Forms::Button());
                        this->button4 = (gcnew System::Windows::Forms::Button());
                        this->SuspendLayout();
                        //
                        // button1
                        //
                        this->button1->Location = System::Drawing::Point(12,
436);
        this->button1->Name = L"button1";
        this->button1->Size = System::Drawing::Size(165, 42);
        this->button1->TabIndex = 0;
        this->button1->Text = L"Синусоида\r\n";
        this->button1->UseVisualStyleBackColor = true;
        this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
                        //
                        // button2
                        //
                        this->button2->Location = System::Drawing::Point(222, 436);
        this->button2->Name = L"button2";
        this->button2->Size = System::Drawing::Size(165, 42);
        this->button2->TabIndex = 1;
        this->button2->Text = L"Тангенсоида";
        this->button2->UseVisualStyleBackColor = true;
        this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
                        //
                        // button4
                        //
                        this->button4->Location = System::Drawing::Point(413, 436);
        this->button4->Name = L"button4";
        this->button4->Size = System::Drawing::Size(155, 42);
        this->button4->TabIndex = 3;
```

```
        this->button4->Text = L"Вы\r\nход";
        this->button4->UseVisualStyleBackColor = true;
        this->button4->Click += gcnew System::EventHandler(this,
&Form1::button4_Click);
                        //
                        // Form1
                        //
                        this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
        this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
        this->ClientSize = System::Drawing::Size(643, 481);
        this->Controls->Add(this->button4);
        this->Controls->Add(this->button2);
        this->Controls->Add(this->button1);
        this->Name = L"Form1";
        this->Text = L"Form1";
        this->ResumeLayout(false);

} //конец инициализации
#pragma endregion
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
   Color ^col = gcnew Color();
  Pen ^pen = gcnew Pen (col->Red) ;
 //Чтобы создать графический объект, надо получить ссылку на него,
 //выполнив метод CreateGraphics() компонента (формы)
 Graphics ^im = this->CreateGraphics();
 // im->DrawLine(pen,20, 10, 300, 100);
 /*рисует линию между 2-мя точками (x1,y1)и (x2,y2)*/

//вывод графика функции с использованием DrawLine()
 im->Clear(col->White); /*Закрасить предыдущее графическое
                          изображение белым цветом*/
 float x1,y1,x2,y2;  //текущие координаты в пикселах в форме
 x1=0; y1=0;   //начальные координаты графика функции f(x)

//вычисление функции y=f(x) в точках x1,x2,...
 x2=x1;
 while(x2<this->Width && y2<this->Height)
  {
    x2+=5; //следующая точка в пикселах по оси X
    y2=fs(x2)* 100;    /*эта функция определена в начале модуля
                        (*100 — для увеличения амплитуды, чтобы было
                        нагляднее)*/
   if(y2<0) y2*=-1;    //чтобы выводить отрицательные значения
   im->DrawLine(pen,x1,y1,x2,y2); // вывод линии между 2-мя точками
   /*точка 2 должна стать точкой 1,
   а точкой 2 должна стать следующая текущая точка: */
```

```
   x1=x2; y1=y2;
   continue;
  }

} //конец обработчика

 private: System::Void button2_Click(System::Object^  sender,
System::EventArgs^  e)
 {
  Color ^col = gcnew Color();
  Pen ^pen = gcnew Pen (col->Red) ;
 //Чтобы создать графический объект, надо получить ссылку на него,
 //выполнив метод CreateGraphics() компонента (формы)
 Graphics ^im = this->CreateGraphics();
 im->DrawLine(pen,20, 10, 300, 100);  //20, 10, 300, 100
 /*рисует линию между 2-мя точками (x1,y1) и (x2,y2)*/

//вывод графика функции с использованием DrawLine()
 im->Clear(col->White); /*закрасить предыдущее графическое
                         изображение белым цветом*/
 float x1,y1,x2,y2;  //текущие координаты в пикселах на форме
 x1=0; y1=0;   //начальные координаты графика функции f(x)

//вычисление функции y=f(x) в точках x1,x2,...
 x2=x1;
 while(x2<this->Width && y2<this->Height)
  {
    x2+=10; //следующая точка в пикселах по оси X
    y2=ft(x2)*50;    /*эта функция определена в начале модуля
                       (*50 — для увеличения амплитуды, чтобы было
                      нагляднее)*/
   if(y2<0) y2*=-1;    //чтобы выводить отрицательные значения
   im->DrawLine(pen,x1,y1,x2,y2); // вывод линии между 2-мя точками
   /*точка 2 должна стать точкой 1,
   а точкой 2 должна стать следующая текущая точка: */
   x1=x2; y1=y2;
   continue;
  }
}  //конец обработчика
 private: System::Void button4_Click(System::Object^  sender,
System::EventArgs^  e)
 {
   this->Close(); //завершение приложения
 } //конец обработчика
}; //конец класса Form1
}   //конец пространства
```

Пояснение

Символ `::` — это так называемый оператор контекстного разрешения (мы должны сказать компилятору, что будет использован глобальный идентификатор вместо локального, предваряя такой идентификатор символом `::`).

Примеры:

- `:: identifier` — это просто идентификатор;
- `class-name :: identifier` — это идентификатор, относящийся к классу;
- `namespace :: identifier` — это идентификатор, относящийся к пространству имен.

При выводе графика любой функции мы применили метод `DrawLine()` класса `Graphics`. Но напрямую сформировать указатель на этот класс, чтобы воспользоваться его членами (в частности, методом `DrawLine()`), нельзя. Надо сначала получить ссылку на этот графический объект через специальный метод формы, который называется `CreateGraphics()`.

Если написать оператор

```
Graphics ^im = this->CreateGraphics();
```

то получим желаемое (метод, по его определению, формирует ссылку на графический объект).

Теперь через определенную нами ссылку типа `Graphics` можем добраться до членов класса `Graphics` (в частности, до необходимого нам метода `DrawLine()`, который рисует прямую линию между двумя точками). Рисование происходит с помощью специального механизма, находящегося в классе `Pen` (перо или ручка с пером). Чтобы добраться до этого механизма, надо объявить ссылку на этот класс, а потом через нее добраться до нужного нам члена класса.

Формирование ссылки на класс происходит не само по себе, а с помощью утилиты gcnew, запускающей специальную программу-конструктор класса, который, в свою очередь, создает в выделенной утилитой памяти экземпляр класса, чтобы с ним можно было работать. Конструктор всегда имеет то же имя, что и класс, для которого он создается, только это обычная функция, выполняющая определенные действия по инициализации членов-данных класса (т. е. придает им некоторые начальные значения).

Конструктор для класса `Pen` имеет один параметр (цвет), потому что перо должно рисовать линии определенным цветом. Следовательно, прежде чем задавать работу конструктору класса `Pen`, нам нужно как-то определиться с цветом, который потом следует задать в качестве параметра этому конструктору. Цвета находятся в специальном классе `Color` (цвет). Чтобы добраться до нужного цвета в этом классе, надо сформировать ссылку на этот класс, а потом уже через нее достать нужный цвет. Отсюда имеем:

```
Color ^col = gcnew Color();
```

Утилита gcnew запускает конструктор класса `Color`, формирует экземпляр класса в памяти и выдает ссылку на этот экземпляр в переменную `col`. Теперь любой цвет из `Color` можно достать через полученную ссылку так:

```
"col->";
```

При этом откроется окно подсказчика, из которого остается только выбрать подходящий цвет. Когда вы начнете вводить начальные буквы нужного вам объекта, среда сразу установит подсветку на ближайший объект, имя которого начинается на вводимые вами символы, что ускоряет процесс выбора нужной строки. После выбора строки нажмите клавишу <Enter>, и она попадет в ваш оператор. Если вы не нашли подходящую строку, значит в объекте, на который вы сформировали ссылку, такого члена-данного нет.

После определения цвета можно выполнять конструктор пера:

```
Pen ^pen = gcnew Pen (col->Red);
```

Чтобы график выводился на поле формы, не занятое предыдущим графиком, следует графический объект, которым мы пользуемся для рисования (он фактически задает специальный холст для рисования, состоящий из точек-пикселов — это все более наглядно показано в Borland C++ Builder), закрасить нейтральным цветом, на фоне которого будет выводиться новый график.

Чтобы рисовать график, из непрерывной функции получают в цикле дискретные значения ее точек и между двумя соседними точками проводят прямую линию. Естественно, чем больше точек на данной поверхности, тем более точным будет график.

C/C++

Microsoft Visual

ГЛАВА 11

Компоненты, создающие интерфейс между пользователем и приложением

В этой главе мы рассмотрим некоторые компоненты, с помощью которых разработчик приложения может создать удобный интерфейс, позволяющий пользователю программного обеспечения легко общаться и управлять последним. Как известно из предыдущего материала, каждый компонент характеризуется тремя наборами данных, определяющими его функциональность: свойства, события и методы. Мы рассмотрим только компоненты первой необходимости, т. к. с течением времени разработчики среды пополняют ее все большим количеством компонентов, на описание которых потребуется не одна толстая книга. Владея принципами работы с основными компонентами, пользователь среды VC++ может самостоятельно осваивать новые, пользуясь справочной помощью (Help), поставляемой со средой системы.

В работе с компонентами используется механизм классов и пространств имен, определенный в библиотеке классов .NET Framework, которая включает в себя классы, интерфейсы, различные типы данных, обеспечивающие доступ к функциональным возможностям среды разработки. С целью достижения совместимости между различными языками, в библиотеке предусмотрен инструмент CLS (Common Language Specification). В данной версии среды разработки используется NET Framework версии 4.5 Beta. В ней много дополнений по сравнению с версией 4, применяемой в среде 2010. Но что любопытно для начинающего программиста: массивы могут занимать до двух гигабайт памяти. Это очень существенно. В библиотеке .NET Framework определены такие элементы:

- основные типы данных и аппарат исключений;
- структуры данных;
- элементы, обеспечивающие ввод/вывод данных;
- элементы, предоставляющие информацию о типах данных;
- элементы, обеспечивающие безопасность данных;
- элементы, обеспечивающие доступ к данным;
- богатый графический пользовательский интерфейс.

В библиотеке .NET Framework имеется достаточный набор как абстрактных (на их основе создаются конкретные классы), так и конкретных классов. Родственные типы в этой библиотеке объединены в пространства имен, чтобы легче было с ними работать. Первая часть полного имени — это имя пространства имен. Последняя часть имени — это тип имени. Например, `System.Collections.ArrayList` представляет тип `ArrayList`, который принадлежит пространству `System.Collections`. Типы данных из этого пространства используются для работы с наборами объектов.

Пространство имен *System*

Это пространство является базовым для фундаментальных типов данных в .NET Framework. Оно включает в себя классы, применяемые в базовых типах данных, которые используются всеми приложениями: `Object` (корневой класс в наследственной иерархии классов), `Byte`, `Char`, `Array`, `Int32`, `String` и т. д. Большинство из этих типов соответствуют первичным типам данных, используемым в языке программирования. В табл. 11.1 представлен список некоторых типов данных для C++.

Таблица 11.1. *Типы данных для C++*

Категория данного	Имя класса	Описание данного
`Integer`	`Byte`	8-битовое беззнаковое целое
	`SByte`	8-битовое целое со знаком
	`Int16`	16-битовое целое со знаком
	`Int32`	32-битовое целое со знаком
	`Int64`	64-битовое целое со знаком
	`UInt16`	16-битовое целое без знака
	`UInt32`	32-битовое целое без знака
	`UInt64`	64-битовое целое без знака
`Float`	`Single`	С обычной (32 бита) точностью число с плавающей точкой
	`Double`	С двойной (64 бита) точностью число с плавающей точкой
`Logical`	`Boolean`	Логическое (булево) число (`true` или `false`)
Другие типы	`Char`	Unicode (16-битовый) символ
	`Decimal`	Десятичное (128 бит) значение
	`IntPtr`	Целое со знаком, значение которого зависит от соответствующей платформы: 32-битовое значение на 32-битовой платформе и 64-битовое на 64-битовой платформе
	`UIntPtr`	Целое без знака, значение которого зависит от соответствующей платформы: 32-битовое значение на 32-битовой платформе и 64-битовое на 64-битовой платформе
Классы	`Object`	Корневой класс для иерархии классов
	`String`	Строка Unicode символов постоянной длины

Работа с переменными некоторых типов

Как работать с переменными, показано на примере обработчика обычной кнопки, где приведены способы работы со строковыми, числовыми и логическими переменными. Текст обработчика приведен в листинге 11.1.

Листинг 11.1

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{

//Проверка методов и свойств класса String
String ^s="123456789";
String ^s1;

//очистка строки
s1=s->Empty; //s1=""

//сравнение строк
s="12345";
s1="12345";
int i=s->Compare(s,s1);//i=0
s1="1234";
i=s->Compare(s,s1);  //i=1
s1="123458";
i=s->Compare(s,s1);  //i=-1

//сцепление строк:
s1=s->Concat(s,s1);  //s1="12345123458"
//копирование s в s1
s1=s->Copy(s);  //s1="12345"
//вставка подстроки, начиная с данного индекса
s=s1->Insert(2,"**"); //s=12**345

/*поиск и возврат индекса первого вхождения подстроки (45) в данную строку*/
 String ^a1="123";
 String ^a2="12345";
 i=a2->IndexOf("45");  //i=3 (индекс изменяется от 0)
 i=a2->IndexOf("45",2);  //i=3 (поиск идет от заданного(2) индекса)
 i=a2->IndexOf("12",1,3);  //i=-1
/*(вхождения не обнаружено.) Поиск идет от заданного индекса (1)
 и проверяется заданное количество символов (3)*/

/*поиск и возврат индекса последнего вхождения заданной
  подстроки в строку*/
a2="12121212";
i=a2->LastIndexOf("12"); //i=6
```

```
//определение длины строки
i=a2->Length; //i=8

/* Помещение строки (a1) в поле заданной ширины (6 символов),
позиции, оставшиеся незанятыми, слева (справа) заполняются пробелами
*/
a2=a1->PadLeft(6); //a2="   123"
a1=a2->PadLeft(3); //a1="123"
a2=a1->PadRight(6); //a2="   123"
//Удаление заданного количества символов из строки
a1="12345678";
a1=a1->Remove(3,4); /*a1="1238" (удаляет 4 символа,
                      начиная с 3-й позиции)*/

/*Замена всех встречающихся в 1-м параметре символов
  на символы из 2-го параметра*/
 a1="12345678";
 a2=a1;
 a2=a1->Replace(a1,"*//*"); // a2="*//*"
 a1="123";
 a2="456";
 a1=a1->Replace(a1,a2);// a1="456"

//Выделение подстроки
 a1="123456";
 a2=a1->Substring(4); /*a2="56" (выделение подстроки
                    с заданной позиции до конца строки)*/
 a1="123456";
 a2=a1->Substring(1,2); //a2="23"
/*(выделение символов с данной позиции (1-й параметр)
   в заданном количестве (2-й параметр))*/

//Вставка подстроки в строку, начиная с заданного индекса
 a1="123***"; a2="456";

 a2=a1->Insert(5, a2);//a2="123**456"

//Проверка методов и свойств данных категории Integer

//Преобразование числовой строки в 32-битовое целое
i=i.Parse("12345"); //i=12345

//целое — в строку
 a1=i.ToString();   //a1="12345"

//умножение/деление
i=i/10;  //i=1234
```

```
i=i/2;   //i=617
i=i*0.5;  //i=308

//Работа с float-данными
Single b,c;
b=10.100000;
c=b*10;  //c=101.00000
c=b/10;  //c=1.0100000
c=b+c;   //c=11.110001

double x,y;
x=10.1;
y=b*10;  //y=101.0.....
y=b/10; //y=1.01.......
//Работа с логическими данными
Boolean z,v;
z=01;  //z=true
v=z & 01; //v=true
z=0; //z=false
v=z | 01;  //v=true
```

В процессе разработки приложений постоянно приходится сталкиваться с необходимостью преобразования данных различных типов одно в другое. Например, дату в формате `DateTime` надо представить в виде строки, чтобы вывести ее в удобном для пользователя виде, и т. д.

Как преобразовывать данные различных типов? Есть несколько способов. Однако наиболее простой — применение методов специального класса `Convert`. Примеры преобразования типов данных показаны в листинге 11.2.

Листинг 11.2

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{
 Double dNumber = 23.15; //это число преобразуется в различные типы.
//Везде идет функция To+имя типа, в который преобразуются
//все методы из класса Convert

   // Returns 23
   Int32 iNumber = Convert::ToInt32( dNumber );  //double to int

   // Returns True
   Boolean bNumber = Convert::ToBoolean( dNumber ); /* double to Boolean*/

  // Returns "23.15"
   String^ strNumber = Convert::ToString( dNumber ); /* double to String ^ (так
же и int to String ^)*/
```

```
  // Returns '2'  //String ^ to Char
  Char chrNumber = Convert::ToChar(strNumber->Substring(0, 1));
//Сначала выделили один символ

    // DateTime to String ^
  String ^s="03/03/2007";
  DateTime d = Convert::ToDateTime(s);
  String ^s1 = Convert::ToString(d);

 }
```

Компонент *Button*

Компонент находится в группе компонентов **All Windows Forms** палитры компонентов. Этот компонент создает в форме, куда он помещен, элемент "кнопка", который надо нажимать щелчком мыши. Компонент `Button` обладает некоторыми свойствами, определяющими его поведение. Вид его в форме показан на рис. 11.1.

Рис. 11.1. Вид компонента `Button`

Свойства *Button*

Перечень свойств компонента представлен на рис. 11.2.

Многие свойства нам уже знакомы — мы с ними встречались при изучении формы. Рассмотрим некоторые из незнакомых свойств.

- `Anchor` — свойство, определяющее закрепленную позицию компонента. Если вы разрабатываете форму таким образом, чтобы пользователь мог изменять ее размеры в режиме исполнения приложения, то компоненты на вашей форме тоже должны соответственно изменяться в размерах. Когда компонент "зацеплен" за форму, а форма изменяется в размерах, то в случае "зацепления" компонент поддерживает дистанцию между собой и позицией анкера. Свойство `Anchor` взаимодействует со свойством `AutoSize`.

Рис. 11.2. Перечень свойств компонента `Button`

- `AutoEllipsis` — это свойство задает, будет ли появляться многоточие справа от текста, сообщая, что текст распространяется за пределы, отведенные размерами компонента. Если установить значение свойства в `true`, то когда пользователь проведет мышью над компонентом (в режиме исполнения приложения), текст, выходящий за пределы нижней части компонента, выведется в виде подсказки. Но следует помнить: чтобы установить свойство в `true`, надо свойство `AutoSize` установить в `false`. Если же `AutoSize` останется равным `true`, многоточие не появится. Все сказанное демонстрируется на рис. 11.3.
- `DialogResult` — свойство, значение которого выбирается из выпадающего списка. Служит для обеспечения закрытия формы, открытой как модальная (см. пояснение в *разд. "Некоторые методы формы" главы 10*).
- `Dock` — с помощью этого свойства обеспечивается причаливание (стыковка) компонентов (в частности, и кнопки) к различным сторонам формы или заполнение ими полностью какого-либо контейнера (компонента, который может содержать в себе другие компоненты, например, это может быть форма или панель). Чтобы обеспечить причаливание компонента, следует:

  1. Выбрать компонент, который требует причаливания.
  2. Раскрыть выпадающий список свойства компонента `Dock`. Откроется схема, состоящая из прямоугольников, имитирующих стороны формы и ее центр.
  3. Щелкнуть на прямоугольнике, обозначающем сторону формы, к которой мы хотим пристыковать наш компонент. Если требуется полное заполнение ком-

Рис. 11.3. Демонстрация работы свойства `AutoEllipsis`

понентом, то надо щелкнуть на центральном прямоугольнике. Если не требуется стыковка, надо выбрать значение `None`. После этого произойдет автоматическое пристыковывание к выбранной стороне формы или ее заполнение. Варианты причаливания к нижней кромке формы, а также и полное заполнение формы компонентом (с применением для наглядности свойства формы `Padding`) показаны на рис. 11.4.

- `FlatAppearance` — это свойство используется, если свойство кнопки `FlatStyle` равно `Flat`. С помощью этого свойства можно задать окантовку кнопки, а также проследить в момент исполнения приложения за состоянием кнопки. Если развернуть это свойство на несколько подсвойств (двойной щелчок на имени свойства: появятся подсвойства, которые можно устанавливать), то, задав каждое из

Рис. 11.4. Вариант причаливания

них, получим желаемое. Можно выбрать цвет окантовки кнопки (если щелкнуть в поле свойства `BorderColor`, то в конце поля появится кнопка, открывающая доступ к выбору цвета), можно выбрать цвет самой кнопки, в который она окрасится, когда на нее наведут курсор мыши, можно выбрать цвет, в который окрасится поле кнопки, если на нее нажать. Все сказанное показано на рис. 11.5.

Рис. 11.5. Демонстрация работы свойства `FlatAppearance`

Рис. 11.6. Виды стилей кнопок

- `FlatStyle` — задает стиль кнопки. Различные стили кнопок показаны на рис. 11.6. Стиль выбирается из выпадающего списка, который открывается кнопкой, появляющейся после щелчка мышью в поле этого свойства.
- `Image` — это свойство, позволяющее через кнопку с многоточием выбрать изображение, которое поместится в поле кнопки, но при условии, что значение `FaltStyle` не будет равно `System`.
- `ImageAlign` — с помощью этого свойства выравнивают изображение на кнопке: в поле свойства есть кнопка, которая открывает схему выравнивания, состоящую из ряда прямоугольников. На каком прямоугольнике мы щелкнем, в такое поле кнопки и сдвинется изображение.
- `ImageIndex` — это свойство связано со свойством `ImageList`, задающим ссылку на ряд изображений (пиктограмм), которые выбираются этим компонентом через диалоговое окно, открывающееся в поле этого свойства. Надо поместить `ImageList` в форму, тогда его имя станет видимым в поле свойства кнопки `ImageList`. Каждому изображению в списке присваивается свой порядковый номер (индекс), по которому впоследствии и можно будет выбирать нужное, или

ключ, в качестве которого служит название файла-изображения. Свойство кнопки `ImageIndex` выбирается из списка (при открытии списка становятся видны все индексы, сформированные компонентом `ImageList`). Свойство кнопки `ImageKey` тоже формируется открытием списка значений, в качестве которых выступают имена файлов пиктограмм, сформированные компонентом `ImageList`. В рассматриваемом свойстве как раз и задается индекс нужного изображения из списка изображений. По нему в кнопке должна появиться пиктограмма.

- `TabIndex` — в это свойство помещается сформированный средой программирования порядковый номер компонента в контейнере, например, в форме (мы пока не знаем другого контейнера, кроме формы). Какие номера имеют свойства `TabOrder`, в такой последовательности и станут активизироваться (получать фокус ввода) компоненты в форме после запуска приложения при последовательном нажатии клавиши <Tab>. Если вам надо изменить порядок активизации, то вы сами должны присвоить соответствующие значения свойствам компонентов `TabIndex`.
- `TabStop` — дает возможность отключать получение фокуса ввода данным компонентом с помощью клавиши <Tab> (надо просто установить это свойство в значение `false`).
- `TextImageRelation` — задает взаимоотношение между изображением и текстом (кто над кем будет выводиться и впереди кого, или кто кого будет замещать).
- `UseMnemonic` — если придать этому свойству значение `true`, то символ `&` будет показываться в текстовой строке, иначе (при `false`) его не будет видно. Значение по умолчанию `true`.
- `UseVisualStyleBackColor` — задает возможность использования визуальных стилей для фонового цвета компонента. Такие стили в системе специфицированы. Например, эти стили определяют цвет, размер или шрифт компонента. Свойство дает возможность координации этих параметров с интерфейсом вашего приложения.
- `Visible` — это свойство обеспечивает видимость или невидимость компонента в режиме исполнения приложения.

События *Button*

Перечень событий кнопки показан на рис. 11.7.

Большинство событий по-своему аналогичны одноименным событиям формы. Рассмотрим некоторые события:

- `Click` — возникает, когда на кнопке щелкают мышью;
- `Enter` — возникает, когда кнопка получает фокус ввода, т. е. становится активной (ее можно нажимать);
- `MouseHover` — возникает, когда курсор мыши находится над кнопкой. По этому событию можно, например, изменять свойства кнопки (как-то ее выпячивать,

чтобы пользователю было видно, что он сейчас держит мышь именно над той кнопкой, на которую ему следует нажать);

- `MouseLeave` — возникает, когда курсор мыши покидает кнопку. Здесь также можно воспользоваться наступлением этого события и, в противоположность предыдущему свойству, на основе которого изменены свойства кнопки, теперь их восстановить, чтобы кнопка приняла прежний вид.

Рис. 11.7. События кнопки

Методы *Button*

Кнопка имеет большое число методов, главным образом унаследованных от своих классов-предков. Рассмотрим только некоторые из них:

- `Hide()` — прячет кнопку (делает ее невидимой);
- `Focus()` — делает кнопку активной (ее можно нажимать);
- `Select()` — работает аналогично `Focus()`;
- `Show()` — показывает кнопку (присваивает ее свойству `Visible` значение `true`).

Компонент *Panel*

Этот компонент находится в списке **All Windows Forms** палитры компонентов. Панель — это компонент, который, как и форма, является контейнером, куда поме-

щаются другие компоненты. Панели обеспечивают общее (родовое) поведение для компонентов, помещенных в них: панельные компоненты могут содержать другие компоненты, объединяя их в единое целое. При перемещении панели такие компоненты перемещаются вместе с ней. Вид панели, помещенной в форму, и некоторые действия с ней показаны на рис. 11.8.

Рис. 11.8. Вид панели

Некоторые свойства *Panel*

Свойства панели, отображенные в окне **Properties**, показаны на рис. 11.9.

Ничего особенного по сравнению с кнопкой мы здесь не видим. Но так как панель — это контейнер, то в него могут помещаться другие компоненты. Поэтому возможна потеря их видимости в этом контейнере. Отсюда возникли свойства, связанные с введением скроллинга (прокрутки) для такого компонента. Это такие свойства, как `AutoScroll`, `AutoScrollMargin` и `AutoScrollMinSize`. Первое из них обеспечивает введение автоматического скроллинга по вертикали и по горизонтали в момент, когда очередной компонент, помещаемый на панель, становится видимым не полностью (не помещается весь на панель). Второе свойство определяет отступы от сторон панели при скроллинге, а третье свойство определяет минимальный размер полос прокрутки, создаваемых для скроллинга.

Некоторые события *Panel*

Перечень событий компонента, отображенный в окне **Properties**, показан на рис. 11.10.

Из событий панели отметим следующие:

- `HelpRequested` — возникает, когда пользователю требуется информация для компонента (нажимает клавишу <F1>);
- `Layout` — возникает, когда компоненты, расположенные на панели, меняют свои места;
- `PreviewKeyDown` — возникает, когда нажимается клавиша на клавиатуре, но перед событием `KeyDown`.

Рис. 11.9. Свойства панели

Рис. 11.10. Перечень событий компонента `Panel`

Компонент *Label*

Компонент `Label` (метка) находится в списке **All Windows Forms** палитры компонентов. Этот компонент выводит в свое поле тексты или изображения, которые пользователь в режиме исполнения приложения не может редактировать. Компонент используется, чтобы идентифицировать некоторый объект в форме или в другом контейнере (т. е. в качестве метки к другому компоненту), однако фокуса ввода получать не может.

Некоторые свойства *Label*

Свойства компонента, отображенные в окне **Properties**, показаны на рис. 11.11.

Рис. 11.11. Свойства компонента `Label`

Рассмотрим некоторые из них:

- `BorderStyle` — задает окантовку метки, которая выбирается из выпадающего списка. Там три значения: без окантовки, окантовка одной линией, окантовка под трехмерное пространство;
- `TextAlign` — задает способ расположения (выравнивания) текста, записываемого в поле свойства `Text` (будет ли текст выравниваться по левой, правой границе поля, или же по центру и т. п.). При нажатии кнопки выбора расположения тек-

ста открывается схема, по которой нужно установить место расположения текста, щелкая на необходимом прямоугольнике схемы (чтобы увидеть длинный текст в метке, надо отключить свойство `AutoSize`). После этого в метке появятся анкерные точки, за которые поле метки можно растягивать или сжимать.

Остальные свойства метки аналогичны ранее рассматриваемым свойствам в предыдущих компонентах.

События *Label*

События этого компонента показаны на рис. 11.12 (они в основном совпадают с событиями для ранее рассмотренных компонентов).

Рис. 11.12. События компонента `Label`

Компонент *TextBox*

Компонент находится в списке **All Windows Forms** палитры компонентов. Этот компонент задает в форме однострочное или многострочное редактируемое поле, через которое вводят/выводят строчные данные. Вид компонента в форме показан на рис. 11.13.

Рис. 11.13. Вид компонента `TextBox` в форме

Некоторые свойства *TextBox*

Свойства компонента, отображенные в окне **Properties**, показаны на рис. 11.14.

Рис. 11.14. Свойства компонента `TextBox`

Рассмотрим некоторые из них.

- `AcceptsReturn` — показывает, будет ли нажатие клавиши <Enter> в многострочном режиме этого компонента создавать новую строку текста (значение свойства равно `true`), или активизируется кнопка по умолчанию для формы (значение свойства равно `false`).

  Если значение свойства равно `false` (принято по умолчанию), то пользователь должен нажимать комбинацию клавиш <Ctrl>+<Enter>, чтобы организовывалась новая строка в многострочном режиме ввода.

Если для формы не существует кнопки по умолчанию, то нажатие клавиши <Enter> всегда будет создавать новую текстовую строку, независимо от значения свойства `AcceptsReturn`.

Кнопка по умолчанию для формы задается ее свойством `AcceptButton`, которое не отображается в окне **Properties**. Эта кнопка будет автоматически срабатывать, когда пользователь нажмет клавишу <Enter>. Наличие такого свойства позволяет конструировать действия по умолчанию. Если это свойство не определять, то такой кнопки не будет и тогда станет действовать правило: нажатие клавиши <Enter> всегда будет создавать новую текстовую строку, независимо от значения свойства `AcceptsReturn`.

- `AcceptsTab` — показывает, приведет ли нажатие клавиши <Tab> при многострочном режиме ввода к появлению кода этой клавиши в строках (вместо стандартной реакции на нажатие клавиши <Tab> — перемещение фокуса ввода на очередной по значению свойства `TabIndex` компонент).

  Если значение свойства `AcceptsTab` равно `false`, то нажатие клавиши <Tab> приведет к перемещению фокуса на очередной компонент, если же значение равно `true`, то, чтобы переместить фокус на очередной компонент, надо будет нажать комбинацию клавиш <Ctrl>+<Tab>, иначе нажатие <Tab> приведет к вставке символа табуляции в строку.

- `AutoCompleteCustomSource` — это свойство совместно со свойствами `AutoCompleteMode` и `AutoCompleteSource` обеспечивает автоматическое пополнение или подсказку с выбором из списка для вводимых строк, что служит контролю ввода, т. к. в `TextBox` могут вводиться различного рода данные: URL, адреса, имена файлов, команды.

  Свойства `AutoCompleteMode` (способ автоматического пополнения) и `AutoCompleteSource` (источник, т. е. откуда берутся данные автоматического пополнения) должны работать вместе. В частности, если из выпадающего списка свойства `AutoCompleteSource` выбрать значение `CustomSource`, то можно в качестве источника данных использовать свойство `AutoCompleteCustomSource` — оно позволяет открыть специальный редактор, с помощью которого можно задавать строки данных. Без значения `AutoCompleteSource`, равного `CustomSource`, это свойство использовать нельзя. Следует заметить, что `TextBox` должен быть обязательно в однострочном режиме. Пример использования этих трех свойств показан на рис. 11.15.

- `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).

- `Lines` — с помощью этого свойства можно задавать строки (через открываемое диалоговое окно), выводить строки и, естественно, вводить строки. Все, что задано в этом свойстве, попадает в свойство `Text` и наоборот. Пример работы со свойством `Lines` показан на рис. 11.16.

Рис. 11.15. Пример использования свойств `AutoCompleteCustomSource`, `AutoCompleteMode`, `AutoCompleteSource`

Рис. 11.16. Пример работы со свойством `Lines`

- `Multiline` — задает режим многострочного ввода (мы его задали другим способом — открыли вспомогательное меню на самом компоненте).
- `PasswordChar` — если мы хотим, чтобы вводимые в поле `TextBox` символы не высвечивались, а заменялись неким другим символом, как это происходит при вводе пароля, то в это свойство надо ввести значение такого символа (например, звездочку). Но при этом значение свойства `Multiline` должно быть `false`.
- `ReadOnly` — определяет, может ли пользователь менять текст в поле компонента: при значении этого свойства `true` — не может, при значении `false` — может.
- `ScrollBars` — этим свойством можно воспользоваться при многострочном режиме, когда строки не помещаются в отведенное пространство `TextBox` (можно ввести полосы прокрутки). Какие полосы вводить и надо ли их вводить, можно выбрать из выпадающего списка свойства. При этом необходимо следить за свойством `WordWrap` — горизонтальная полоса прокрутки может не появиться.
- `ShortcutsEnabled` — дает возможность применять установленные в среде клавишные комбинации быстрого вызова:
  - <Ctrl>+<Z>;
  - <Ctrl>+<E>;
  - <Ctrl>+<C>;
  - <Ctrl>+<Y>;
  - <Ctrl>+<X>;
  - <Ctrl>+<Backspace>;
  - <Ctrl>+<V>;
  - <Ctrl>+<Delete>;
  - <Ctrl>+<A>;
  - <Shift>+<Delete>;
  - <Ctrl>+<L>;
  - <Shift>+<Insert>;
  - <Ctrl>+<R>.
- `Text` — через это свойство вводится/выводится одна строка текста.
- `TextAlign` — свойство позволяет выравнивать текст в поле компонента, выбирая способ выравнивания из выпадающего списка (текст может располагаться слева, справа или по центру поля).
- `UseSystemPasswordChar` — задает возможность объявления вводимого символа в качестве парольного по умолчанию. Если такой символ введен, то любой символ, вводимый в свойство `PasswordChar`, будет проигнорирован.
- `WordWrap` — задает возможность автоматического переноса символов к началу следующей строки, т. е. строку, не умещающуюся в поле ввода, можно продолжить (не нажимая, естественно, при этом на клавишу <Enter>).

События *TextBox*

Перечень событий компонента, отображаемый в окне **Properties**, показан на рис. 11.17.

Из всех событий компонента рассмотрим интересное событие `KeyDown`. Оно возникает, когда пользователь приложения нажимает любую клавишу на клавиатуре, а

Рис. 11.17. События компонента `TextBox`

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 отсылает всякий раз, когда нажимаете любую клавишу при вводе, надо вставить проверку на нажатие клавиши <Enter>. Вид содержимого обработчика этого события:

```
if(e->KeyCode == Keys::Enter)
 {
  /* Здесь помещаются операторы, обрабатывающие введенную строку*/
 }
```

Пояснение:

`e` — это параметр обработчика события `KeyDown`.

Он описан так:

```
System::Windows::Forms::KeyEventArgs^  e
```

То есть это ссылка на класс `KeyEventArgs`, который обеспечивает данными обработка событий `KeyDown` и `KeyUp`.

Когда вы введете `e->`, то появится окно подсказки, из которого нужно выбрать элемент `KeyCode`, который создает код клавиатуры для событий `KeyDown` или `KeyUp`.

`KeyCode` описан как элемент типа `Keys`, где класс `Keys` — это перечислимый класс, содержащий значения различных кодов клавиатуры.

Когда введете `Keys::`, то откроется окно подсказчика, из которого сможете выбрать мнемонический код интересующей вас клавиши. В частности, вы там найдете кла-

вишу с именем <Enter> (<Enter> — вовсе не обязательная клавиша для фиксации окончания ввода — можно применять и другую, но эта клавиша общепринята и удобна).

Примечание

В рассматриваемой версии продукта (2011 beta) подсказчик отключен, поэтому следует набирать названия элементов без него.

Некоторые методы *TextBox*

Перечень некоторых методов компонента представлен в табл. 11.2.

Таблица 11.2. *Перечень некоторых методов компонента* `TextBox`

Имя метода	Описание метода
`AppendText`	Добавляет текст к текущему тексту в окне компонента
`Clear`	Удаляет весь текст из поля ввода/вывода
`Copy`	Копирует выбранные строки в буфер памяти
`CreateGraphics`	С помощью этого метода можно создать графический объект для рисования на компоненте
`Cut`	Вырезает отмеченное множество строк в поле ввода/вывода и помещает их в буфер памяти
`DeselectAll`	Снимает состояние выборки строк в компоненте
`Dispose`	Удаляет все ресурсы, занятые компонентом, из памяти
`Focus`	Устанавливает фокус ввода компоненту
`Hide`	Прячет компонент от пользователя
`Paste`	Заменяет текущую выборку в поле ввода/вывода содержимым буфера памяти
`Select`	Выбирает заданный текст внутри компонента
`SelectAll`	Выбирает весь текст внутри компонента
`Show`	Выводит компонент (делает его видимым)
`Undo`	Отменяет последнюю операцию редактирования в поле ввода

Структуру каждого метода компонента можно посмотреть в Help среды программирования. Вызов метода происходит, например, по форме:

```
this->TextBox1->Show();
```

или

```
String ^s="Добавка текста";
this->textBox1->AppendText(s);
```

Все зависит от структуры соответствующего метода.

Компонент *MenuStrip*

Компонент находится в списке **All Windows Forms** палитры компонентов. 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 `MainMenuStrip`, в окне которого и появляется имя компонента.

Когда меню сформировано, то после запуска приложения на выполнение в левой верхней 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). При переносе значка меню из палитры в форму значок располагается не в самой форме, а в нижней части рабочего стола (на специальной полосе под формой). В то же время в форме (в ее верхней части) появляется полоса для отображения будущих опций меню (рис. 11.18).

Рис. 11.18. Помещение компонента главного меню в форму

Рис. 11.19. Выпадающий список **MenuStrip Tasks**

На рис. 11.19 показан выпадающий список **MenuStrip Tasks**.

Это диалоговое окно обеспечивает доступ к типичным командам и свойствам.

- `Embed in ToolStripContainer` — позволяет (щелчком мыши) поместить меню в специальный контейнер (вместо расположения его в форме). Контейнер — это объект со своим набором свойств, установка которых позволяет создавать меню, более удобное для пользователя. Вид инструментального контейнера с помещенной в него заготовкой меню показан на рис. 11.20.
- `Insert Standard Items` — добавляет общепринятые опции меню (рис. 11.21).
- `RenderMode` — опция дает возможность выбора из выпадающего списка способа изображения меню: системного (`System`), профессионального (`Professional`) или

управляемого (`ManagerRenderMode`). Опция `System` обеспечивает темноватый фон по опциям меню опции `ManagerRenderMode` и `Professional` светлый.

- `Dock` — выводит (по щелчку на кнопке) в поле этой опции схему причаливания меню к той или иной стороне формы. На рис. 11.22 показано, что меню поместилось, прижатым к левой стороне формы.

Рис. 11.20. Меню, помещенное в инструментальный контейнер

Рис. 11.21. Добавка типизированных опций в меню

Рис. 11.22. Причаливание меню к левой стороне формы

- `GripStyle` — в этой опции существует выпадающий список, задающий элемент стиля полосы меню: невидима или видима будет специальная пунктирная канавка в верхней части полосы (рис. 11.23).

Рис. 11.23. Появление в верху меню пунктирной канавки

- `Edit Items` — с помощью этой опции и задаются опции самого меню. Если щелкнуть на этой опции, то откроется диалоговое окно для задания опций главного меню, причем в левой его части существует окно для добавления новых опций, а в правой части открывается окно для настройки свойств добавляемых опций (рис. 11.24). С помощью этого окна можно не только добавлять новые опции, но и удалять и реорганизовывать их.

Рис. 11.24. Диалоговое окно для задания опций главного меню

Рассмотрим некоторые свойства сформированной опции и посмотрим, как их устанавливать. Прежде всего опцию надо как-то назвать и сделать это так, чтобы поль-

зователю было легко ориентироваться в меню. Название опции задается, как и у ранее рассматриваемых компонентов, в свойстве `Text`. После задания главных опций, щелкая на них, можно задавать остальные подопции: сначала — подмножество в виде столбца, потом — к каждому элементу столбца — задается свое подменю и т. д. Когда мы вписываем названия элементов меню в появляющиеся пустые заготовки, то эти названия попадают в свойства `text` формируемых элементов. В конце концов в соответствии с алгоритмом задачи мы остановимся на некотором шаге, когда создан последний элемент некоторой ветки меню, который должен будет выходить на обработку данных. Как это сделать? Надо открыть свойства этого последнего в цепочке элемента, в свойствах выбрать колонку событий и дважды щелкнуть на событии `Click`, которое выведет нас на обработчик этого события в программном модуле. Там мы и запишем реакцию щелчка мыши на последнем элементе меню. Все это показано на рис. 11.25.

Рис. 11.25. Опция меню и ее окно **Properties**

Вид обработчика события:

```
private: System::Void toolStripMenuItem6_Click(System::Object^  sender,
System::EventArgs^  e)
{


/*Здесь помещаются операторы C++, которые реализуют алгоритм, соответствующий
функции этого элемента меню. В частности, здесь же можно устанавливать и
свойства этой опции программным путем, задавая операторы типа
this->toolStripMenuItem6->
*/
}
```

Таким образом, мы связали созданную нами опцию с возможностью обработки при ее нажатии (когда начнется исполняться приложение).

Аналогично создаются и другие опции (в глубину и ширину меню): когда вы щелкните на какой-то опции (т. е. активизируете ее), то сразу откроются два окна (рядом и справа) с надписью в них: **Type Here** (вводите сюда).

То окно, которое справа, позволяет определить новую горизонтальную подопцию, которое внизу — новую вертикальную.

Допустим, мы опцию сформировали. Но возникают два вопроса, которые опытный пользователь сразу вам задаст. Он скажет: "Я знаю, что при эксплуатации любой программы для обеспечения большей скорости работы оператора надо иметь возможность работать не только с помощью мыши, но и с помощью клавиатуры. Вы это предусмотрели? И второе — у меня в приложении намечается довольно сложное меню, которое отражает функцию моего предприятия. Поэтому с приложением будут работать многие операторы, и я бы хотел, чтобы оператор, работающий с одним разделом меню, не мог работать с другим, точнее, чтобы он не имел к нему доступа".

И вы должны будете его требования учесть. Как же этого добиться?

Что касается дублирования действий мыши работой с клавиатурой, то этот вопрос решается. Активизируйте любую созданную вами опцию и посмотрите ее свойства в окне **Properties**. Вы там найдете свойства `ShowShortcutKeys` и `ShortcutKeys`. Первое из них обеспечивает видимость клавиш быстрого вызова (`ShortcutKeys`) в наименовании опции (если эти клавиши будут определены). А второе свойство как раз и задает комбинацию клавиш, при нажатии которой опция станет выполняться так, как если бы вы на ней щелкнули мышью. Задание происходит в два этапа, как показано на рис. 11.26.

Какой вывод? Если пользователь запомнит комбинации клавиш у исполнительных опций, то ему нет необходимости двигаться по всему дереву меню (что делается с помощью мыши), а стоит только нажать нужную комбинацию и соответствующая опция исполнится.

Текст обработчика события для этой опции представляет собой следующее:

```
private: System::Void toolStripMenuItem3_Click(System::Object^  sender,
System::EventArgs^  e)
{
MessageBox::Show("Нажата комбинация <Ctrl+2>", "Приложение 11.01-2011",
MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
}
```

Результат показан на рис. 11.27.

Вернемся теперь ко второму вопросу, поставленному пользователем разработчику: каким образом ограничить доступ ко всему меню некоторым операторам?

Для этого делается следующее: главная опция или отдельные подопции либо лишаются возможности доступа к ним, либо делаются вообще невидимыми. Первое достигается за счет переброски свойства опции `Enabled` в значение `false`, а второе — за счет присвоения свойству `Visible` тоже значения `false`. Но это делается не "в лоб". Следует разработать механизм авторизации доступа к меню, который станет по регистрационным данным пользователя открывать одни опции, делать невидимыми другие и недоступными третьи.

Рис. 11.26. Вид меню с заданными клавишами быстрого вызова

Рис. 11.27. Выполнение опций меню с помощью горячих клавиш

Некоторые свойства *MenuStrip*

- `BackgroundImage` — задает (с помощью выбора через диалоговое окно) фоновое изображение, которое помещается в меню и на фоне которого будут видны его опции. Так же можно поступать и с опциями, т. к. у них тоже есть такое свойство (рис. 11.28). С помощью свойства `BackgroundImageLayout` изображение можно "подогнать" под соответствующий формат.
- `Items` — через диалоговое окно этого свойства формируются главные опции меню.
- `LayoutStyle` — стиль размещения меню. Выбирается из выпадающего списка.
- `Checked` — с помощью этого свойства можно контролировать, была ли выбрана данная команда меню. Это очень важно при эксплуатации приложения: если в вашем меню множество опций, а вы некоторые из них уже выполнили, то если не пометить выполненные, возможно по ошибке станете выполнять какую-нибудь опцию снова. Существует свойство `CheckOnClick`, которое (если установить его значение в `true`) обеспечит необходимую пометку выполненной опции (но при условии, что ее свойство `Checked` тоже имеет значение `true`). Пример работы с этим свойством показан на рис. 11.29. Обработчик события имеет вид:

```
if(this->toolStripMenuItem5->CheckOnClick==false)
 {
  this->toolStripMenuItem5->Checked=true;
  this->toolStripMenuItem5->CheckOnClick=true;
 }
```

```
else
 {
    this->toolStripMenuItem5->Checked=false;
  this->toolStripMenuItem5->CheckOnClick=false;
 }
```

Рис. 11.28. Формирование фонового изображения на полосе меню

Рис. 11.29. Пример работы со свойством `Checked`

События *MenuStrip*

Перечень событий компонента приведен на рис. 11.30.

Рис. 11.30. Перечень событий `MenuStrip`

Компонент *ContextMenuStrip*

Компонент находится в списке **All Windows Forms** палитры компонентов. Этот компонент придает другому компоненту, с которым он связывается, дополнительные функциональные возможности, он может быть связан с любым другим компо-

нентом (формой, кнопкой и т. д.), имеющим свойство `ContextMenuStrip`. Когда компонент помещается в форму, его имя будет видно в любом из компонентов формы, у которого есть свойство `ContextMenuStrip`. Это обычное меню, где 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. На рис. 11.31 показано действие контекстного меню. Меню появляется в том месте, где находится указатель мыши. Задание опций контекстного меню аналогично заданию опций в главном меню.

Рис. 11.31. Контекстное меню формы

Компонент *ListView*

Этот компонент находится в списке **All Windows Forms** палитры компонентов. С его помощью в форме выводится список элементов с пиктограммами, его можно использовать в пользовательском интерфейсе аналогично использованию правого окна в Windows Explorer. Вид компонента в форме показан на рис. 11.32.

Рис. 11.32. Вид `ListView` в форме

Начнем с того, что рассматриваемый компонент позволяет выводить информацию в виде таблицы, которая может иметь столбцы с заголовками, а внутри поля выво-

да — строки, сгруппированные в задаваемые группы. Например, можно задавать такие группы, как "Группа адресов Интернета", "Группа текстовых файлов" и т. д. В таком порядке и рассмотрим перечень задач из открывающегося окна, показанного на рис. 11.32.

Если выбрать опцию `Edit Columns` (редактирование столбцов), то для их задания и возможного редактирования откроется диалоговое окно, показанное на рис. 11.33.

Рис. 11.33. Диалоговое окно для задания столбцов вывода с помощью `ListView`

В этом окне в его левой половине кнопкой **Add** будем добавлять новую колонку к компоненту. При очередном добавлении в правом окне каждый раз станет открываться окно свойств, которые позволяют придавать добавленному элементу определенные свойства: задать его имя (это делается в свойстве `Text`), ширину текста и т. д. Но если вы, задав хотя бы имена и нажав кнопку **OK**, завершающую задание колонок, посмотрите на компонент `ListView`, вы заданных заголовков не увидите. Чтобы они появились в компоненте, надо в окне задач (рис. 11.33, справа) выбрать опцию `View`, открыть ее выпадающее меню и в нем выбрать опцию `Details`. Только тогда сформированные вами заголовки с их именами появятся в `ListView`. Эти заголовки будут разделены вертикальными линиями, за которые можно мышью зацепляться и перетягивать их влево-вправо, чтобы получше отрегулировать видимость заголовка, т. к. его наименование может не помещаться в отведенное ему по умолчанию место.

Сформируем теперь группы, в которые станут объединяться строки в момент их вывода в окно `ListView`. Для этого выполним опцию `Edit Groups` (редактирование групп) в меню задач компонента (рис. 11.34). Правила формирования групп очень схожи с правилами формирования колонок: диалоговое окно очень похоже.

Однако после формирования групп мы их не увидим в окне `ListView`. Они в нем появятся только после задания элементов вывода (`Items`), т. е. самих конкретных данных. При их задании в свойствах каждого элемента будет свойство `Group`, т. е. надо будет задать, к какой группе принадлежит создаваемый элемент. В поле этого свойства имеется выпадающий список, в котором будут видны все группы, которые

мы создали предварительно. Форма задания выводимых элементов аналогична заданию колонок и групп: то же диалоговое окно, так же открывается окно свойств заданного элемента и т. д.

Рис. 11.34. Формирование групп вывода информации через `ListView`

Здесь следует отметить, что когда мы задаем имя элемента в свойстве `Text`, это имя попадает в первую определенную ранее колонку. Содержимое остальных заданных колонок определяется заданием свойства `SubItems`. В его поле надо щелкнуть на кнопке с многоточием, в результате чего откроется диалоговое окно уже знакомого нам формата, в котором мы добавляем элемент за элементом: первый добавленный будет значением второй колонки `ListView`, второй добавленный — третьей и т. д.

Последовательность формирования элементов вывода через опции `Edit Items`, формирование субэлементов и общий результат использования компонента `ListView` для вывода показана на рис. 11.35—11.37.

В соответствии с рис. 11.32 компонент `ListView` имеет пять режимов просмотра (свойство `View`):

- `LargeIcon` — выводит большие пиктограммы рядом с текстом элемента. Элементы появляются в многоколоночном варианте, если компонент достаточно широк;
- `SmallIcon` — режим такой же, как и предыдущий, за исключением того, что пиктограммы появляются маленькими;
- `List` — выводит маленькие пиктограммы, но элементы всегда выводятся в одну колонку;
- `Details` — выводит элементы во многих колонках;

Рис. 11.35. Последовательность формирования элементов вывода через опцию `Edit Items`

Рис. 11.36. Последовательность формирования субэлементов вывода

Рис. 11.37. Общий результат вывода

- `Tile` — этот режим доступен только для Windows XP. Вывод можно подстроить, отрегулировав размер вывода по высоте и ширине, задав свойство `TileSize`, а также отрегулировав количество выводимых строк с помощью свойства `Columns`.

Пример вывода информации с помощью компонента `ListView` мы видели на рис. 11.37. Однако, глядя на него, мы ничего, кроме заданной информации, не имеем. А хотелось бы, чтобы, щелкая мышью на имени выводимого элемента, мы извлекали его содержимое для просмотра. Для достижения этой цели надо бы еще обработать как-то элемент наподобие того, как это делается с элементом обыкновенного меню. Об этом чуть позже.

Не все свойства `ListView` работают во всех режимах вывода. В табл. 11.3 приведены некоторые свойства `ListView` и соответствующие им режимы вывода.

Таблица 11.3. *Некоторые свойства `ListView` и соответствующие им режимы вывода*

Имя свойства	Описание	Режим вывода
`Columns`	С его помощью задают заголовки колонок	`Details` или `Tile`
`Groups`	Задает структуру, в которую можно объединить группу элементов	Все режимы, кроме `List`
`HeaderStyle`	Задает стиль заголовка колонки (могут ли сами заголовки служить кнопками выполнения каких-либо действий, будут ли показаны заголовки или нет)	`Details`

Некоторые свойства *ListView*

- `Items` — это ключевое свойство, которое содержит элементы, выводимые компонентом `ListView`. Задание выводимых компонентом элементов мы только что рассмотрели.

- `SelectedItems` — свойство содержит набор элементов, которые подверглись выборке.
- `MultiSelect` — если это свойство установлено в `true`, то пользователь может выбрать множество элементов.
- `CheckBoxes` — если это свойство установлено в `true`, то `ListView` может выводить окна контроля рядом с элементами (в режиме `Title` свойство не поддерживается).

  Включение `CheckBoxes` позволяет выводить список элементов или подэлементов (для режима `Details`), которые пользователь может затем выбирать, щелкая на окне контроля (после этого там появляется галочка). То есть с помощью этого свойства можно выбирать множество элементов, не используя при этом обычный способ (нажатие клавиши <Ctrl> и щелканье мышью на элементе, который требуется выбрать). Причем выборку можно производить и при отключенном свойстве `MultiSelect`. Чтобы определить, помечен ли галочкой элемент, надо создать обработчик события `ItemCheck`. Чтобы получить все помеченные галочкой элементы, надо использовать свойство `CheckedItems`, которое содержит текущий помеченный элемент. Свойство имеет ссылку на класс `CheckedListViewItemCollection`, который содержит все помеченные элементы. Чтобы получить индексы всех помеченных элементов, надо использовать свойство `CheckedIndices`.
- `StateImageList` — если значение этого свойства равно какому-то компоненту типа `ImageList`, то вместо окон контроля (check boxes) станут выводиться изображения из компонента `ImageList` с индексами 0 и 1. Причем изображение с индексом 0 будет выводиться вместо неотмеченного галочкой окна контроля, а вместо помеченного галочкой станет выводиться изображение, соответствующее индексу 1.
- `Activation` — это свойство показывает, какого типа действие (`Standard`, `OneClick` или `TwoClick`) должен совершить пользователь, чтобы активизировать элемент в списке. Действие типа `OneClick` требует одного щелчка мыши для активизации элемента, при этом один щелчок меняет цвет элемента. Действие типа `TwoClick` требует для активизации элемента двойного щелчка мыши, при этом в окнах контроля появляются галочки. Действие типа `Standard` также требует одного щелчка, но при этом элемент не меняет своего цвета и окно контроля галочкой не помечается.
- `HotTracking` — это свойство используется для показа элемента, являющегося гиперссылкой, когда над ним проходит курсор мыши. При этом элемент подсвечивается и подчеркивается (как гиперссылка). Это означает, что на такую ссылку можно нажимать как на кнопку. Форма курсора мыши при установке `HotTracking = true` меняется со стандартного на курсор типа `Hand` (рука).
- `GridLines` — с помощью этого свойства можно задавать решетку в поле вывода элементов (столбцы и строки будут разделены горизонтальными и вертикальными линиями).

- `LabelEdit` — если это свойство установлено в `true`, то пользователь имеет возможность модифицировать текст элемента (надо щелкнуть текст элемента, чтобы перевести его в состояние редактирования). Однако тексты подэлементов не могут таким образом редактироваться — для этого надо открыть диалоговое окно двойным щелчком на подэлементе.
- `Group` — с помощью этого свойства создаются структуры, объединяющие в себе элементы. Когда открывается диалоговое окно для задания элемента, в нем есть свойство `Group`, в котором (в выпадающем списке) имеются все созданные группы. Какую группу мы выберем из этого списка, к той группе и станет относиться формируемый элемент. Группы тоже создаются в специальном диалоговом окне, которое открывается, если щелкнуть в поле свойства `Group` в окне **Properties** для компонента `ListView`. Вид диалогового окна для создания группы был показан на рис. 11.34.

Примечание

Следует помнить, что если свойство `Activation` установлено в `OneClick` или в `TwoClick`, то указанное редактирование элементов будет проигнорировано, несмотря на значение свойства `LabelEdit`. Кроме того, надо отключить свойство `HotTracking`, т. к. по его определению щелчок на элементе будет запускать его на выполнение.

События *ListView*

Перечень событий компонента представлен на рис. 11.38.

Среди множества событий компонента мы видим немало знакомых, однако появились и чисто специфические (касающиеся работы с элементами компонента `ListView`).

Рис. 11.38. Перечень событий `ListView`

Ранее мы рассматривали свойство `HotTracking`, которое позволяло элементы `ListView` делать кнопками: при наведении на них курсора мыши сам курсор менял форму на курсор типа `Hand` (рука), что являлось признаком возможности щелчка на этом элементе.

Что значит щелкнуть на элементе? Это означает сделать его активным. Поищем среди событий `ListView` то, которое связано с активизацией элемента. Такое событие есть и его имя `ItemActivate` (оно возникает именно после активизации элемента). Проверим это на практике, сформировав обработчик этого события, куда поместим обычную функцию выдачи сообщения, которая выдаст информацию, что при щелчке на элементе мы попали именно в обработчик события `ItemActivate`. Итак, свойство `HotTracking` устанавливаем в `true`, а обработчик события `ItemActivate` сформируем в виде:

```
private: System::Void listView1_ItemActivate(System::Object^  sender,
System::EventArgs^  e)
{
MessageBox::Show("Реакция на курсор Hand", "Приложение 10.1-2011",
MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
}
```

Компилируем приложение и выполняем. Результат показан на рис. 11.39.

Рис. 11.39. Использование элементов вывода `ListView` в качестве кнопок

Какой вывод из этого? А вывод такой: мы можем использовать `ListView` для запуска различных приложений, имена которых сможем формировать как элементы это-

го компонента. В частности, например, записав в качестве элементов необходимые нам для работы адреса интернет-сайтов и умея в обработчике запускать Web-браузер, мы сможем создать себе удобный справочник сайтов. Кстати, такой компонент в палитре компонентов имеется. Его мы сейчас и рассмотрим.

Компонент *WebBrowser*

Компонент находится в списке **All Windows Forms**.

Вид компонента в форме показан на рис. 11.40.

Рис. 11.40. Вид `WebBrowser` в форме

Неопытный пользователь сразу и не найдет компонент на форме: он причалил ко всем сторонам формы и только маленькая стрелка в правом верхнем углу формы указывает, что форма не пуста. Если на этой стрелке щелкнуть мышью, откроется перечень опций компонента (всего одна опция). Из этой опции следует, что щелкнув на ней, можно отменить свойство `Dock` компонента, и он перестанет причаливать к форме, а станет иметь естественный вид.

Этот компонент позволяет выводить Web-страницы прямо в вашем приложении. Его можно использовать вместо Internet Explorer (у него имеется ряд свойств, методов и событий, которые позволяют выполнять функции Internet Explorer).

Например, можно использовать опцию **Navigated**, чтобы пользоваться списком адресов, можно использовать опции **GoBack** (дает возможность перейти к предыдущей Web-странице), **GoForward** (дает возможность перейти к последующей Web-странице), **Stop** (приостанавливает текущую навигацию и связанные с ней звуки и анимацию) и **Refresh** (перезагружает текущую Web-страницу) для создания навигационных кнопок на линейке инструментов. Можно обработать событие `Navigated` для обновления линейки адресов значением свойства `Url` (здесь задается или сюда записывается интернет-адрес текущей Web-страницы), можно обработать заголовочную линейку значением свойства `DocumentTitle` (дает заголовок текущей Web-страницы).

Если вы хотите сгенерировать свою собственную страницу внутри приложения, то должны установить свойство `DocumentText` (через него можно ввести или прочитать HTML-текст текущей Web-страницы). Если вы знакомы с моделью DOM (Document Object Model), то также можете манипулировать содержимым текущей Web-страницы, используя свойство `Document`. С помощью этого свойства вы можете сохранять и модифицировать документы непосредственно в памяти, минуя работу с файлами.

Теперь мы готовы к совместному применению компонентов `ListView` и `WebBrowser`, чтобы создать свой справочник интернет-адресов. Пример на этапе дизайна показан на рис. 11.41.

Рис. 11.41. Разработка совместного использования компонентов `ListView` и `WebBrowser`

Рис. 11.42. Созданный справочник интернет-адресов

Результат работы приложения показан на рис. 11.42, h-файл приложения приведен в листинге 11.3.

Листинг 11.3

```
#pragma once

namespace WindowsFormsApplication8 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
    private: System::Windows::Forms::WebBrowser^  webBrowser1;
    protected:
    private: System::Windows::Forms::ListView^  listView1;


    private: System::Windows::Forms::Button^  button6;
    private: System::Windows::Forms::Button^  button1;
    private: System::Windows::Forms::Button^  button2;
```

```
    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>

        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            System::Windows::Forms::ListViewGroup^  listViewGroup1 = (gcnew
System::Windows::Forms::ListViewGroup(L"Интернет-сайты",
System::Windows::Forms::HorizontalAlignment::Left));
            System::Windows::Forms::ListViewItem^  listViewItem1 = (gcnew
System::Windows::Forms::ListViewItem(L"www.boris-pahomov.narod2.ru"));
            System::Windows::Forms::ListViewItem^  listViewItem2 = (gcnew
System::Windows::Forms::ListViewItem(L"www.bhv.ru"));
            this->webBrowser1 = (gcnew System::Windows::Forms::WebBrowser());
            this->listView1 = (gcnew System::Windows::Forms::ListView());
            this->button6 = (gcnew System::Windows::Forms::Button());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->button2 = (gcnew System::Windows::Forms::Button());
            this->SuspendLayout();
            //
            // webBrowser1
            //
            this->webBrowser1->Location = System::Drawing::Point(222, 53);
            this->webBrowser1->MinimumSize = System::Drawing::Size(20, 20);
            this->webBrowser1->Name = L"webBrowser1";
            this->webBrowser1->Size = System::Drawing::Size(503, 250);
            this->webBrowser1->TabIndex = 0;
            //
            // listView1
            //
            this->listView1->Activation = System::Windows::Forms::
ItemActivation::OneClick;
            listViewGroup1->Header = L"Интернет-сайты";
            listViewGroup1->Name = L"Group1";
            this->listView1->Groups->AddRange(gcnew cli::array<
System::Windows::Forms::ListViewGroup^  >(1) {listViewGroup1});
            this->listView1->HotTracking = true;
            this->listView1->HoverSelection = true;
            listViewItem1->Group = listViewGroup1;
```

```
            listViewItem2->Group = listViewGroup1;
            this->listView1->Items->AddRange(gcnew cli::array<
System::Windows::Forms::ListViewItem^  >(2) {listViewItem1, listViewItem2});
            this->listView1->Location = System::Drawing::Point(12, 53);
            this->listView1->Name = L"listView1";
            this->listView1->Size = System::Drawing::Size(202, 169);
            this->listView1->TabIndex = 1;
            this->listView1->UseCompatibleStateImageBehavior = false;
            this->listView1->View = System::Windows::Forms::View::Tile;
            this->listView1->ItemActivate += gcnew System::EventHandler(this,
&Form1::listView1_ItemActivate);
            //
            // button6
            //
            this->button6->Location = System::Drawing::Point(59, 229);
            this->button6->Name = L"button6";
            this->button6->Size = System::Drawing::Size(75, 23);
            this->button6->TabIndex = 7;
            this->button6->Text = L"Выход\r\n";
            this->button6->UseVisualStyleBackColor = true;
            this->button6->Click += gcnew System::EventHandler(this,
&Form1::button6_Click);
            //
            // button1
            //
            this->button1->Location = System::Drawing::Point(222, 22);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(75, 23);
            this->button1->TabIndex = 8;
            this->button1->Text = L"Назад";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
          //
          // button2
          //
          this->button2->Location = System::Drawing::Point(337, 22);
          this->button2->Name = L"button2";
          this->button2->Size = System::Drawing::Size(75, 23);
          this->button2->TabIndex = 9;
          this->button2->Text = L"Вперед";
          this->button2->UseVisualStyleBackColor = true;
          this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
            //
            // Form1
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
```

```
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(726, 299);
            this->Controls->Add(this->button2);
            this->Controls->Add(this->button1);
            this->Controls->Add(this->button6);
            this->Controls->Add(this->listView1);
            this->Controls->Add(this->webBrowser1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->ResumeLayout(false);

        }
#pragma endregion
    private: System::Void listView1_ItemActivate(System::Object^  sender,
System::EventArgs^  e)
 {
    ListView::SelectedListViewItemCollection^ breakfast =
                this->listView1->SelectedItems;
    System::Collections::IEnumerator^ myEnum = breakfast->GetEnumerator();
 while ( myEnum->MoveNext() )// надо запустить перечисление, т. к.
                             // в выборке может быть много элементов
   {
    ListViewItem ^it = safe_cast<ListViewItem^>(myEnum->Current);
    String ^s;s=it->Text;
         //Uri ^Url;
    this->webBrowser1->Navigate(s);
    //this->webBrowser1->Url= dynamic_cast <Uri^>(s);//it->Text;

     }
  }//конец обработчика
private: System::Void button6_Click(System::Object^  sender, System::EventArgs^ e)
 {
  this->Close();
 }

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
   //GoBack
   this->webBrowser1->GoBack();
 }


private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
   //GoForward
   this->webBrowser1->GoForward();
 }
};
}
```

Компонент *ListBox*

Компонент находится в списке **All Windows Forms** палитры компонентов. Этот компонент выводит список элементов, из которых пользователь может выбрать как один, так и множество элементов.

Если множество элементов превосходит размеры окна, то в компоненте автоматически появляется полоса прокрутки.

Если свойство `MultiColumn` установлено в `true`, то компонент выводит элементы в несколько колонок, при этом появляется горизонтальная полоса прокрутки.

Если `MultiColumn` установлено в `false`, то вывод элементов идет в одну колонку и появляется вертикальная полоса прокрутки.

Если свойство `ScrollAlwaysVisible` установлено в `true`, то полоса прокрутки появляется независимо от количества элементов.

Свойство `SelectionMode` задает, сколько элементов может выбираться за один раз.

Как работать с *ListBox*

Свойство `SelectedIndex` возвращает целочисленное значение, которое соответствует первому элементу в списке выбранных. Если выборка оказалась пустой, то значение этого свойства устанавливается в –1. Значение индекса в списке изменяется от нуля. При многострочной выборке это свойство возвращает индекс первого элемента из списка выбранных. Вид компонента в форме показан на рис. 11.43.

Рис. 11.43. Вид `ListBox` в форме

Свойство `SelectedItem` возвращает выбранный элемент. Обычно, это текстовая строка.

Количество элементов в списке сообщается в свойстве `Count`, значение которого всегда на единицу больше индекса последней строки списка, потому что последний отсчитывается от нуля.

Чтобы добавить или удалить строки из списка, используют методы:

- `Add ()` — добавить элемент в конец списка;
- `Insert()` — вставить элемент внутрь списка;

- `Clear()` — удалить все элементы из списка (очищает список);
- `Remove()` — удалить заданный элемент из списка.

Кроме того, можно добавлять элементы в список, используя свойство `Items` в режиме дизайна.

Свойства *ListBox*

Перечень свойств компонента, отображенных в окне **Properties**, показан на рис. 11.44.

Рис. 11.44. Перечень свойств `ListBox`

Уточним смысл некоторых свойств.

- `ColumnWidth` — задает ширину колонки списка при многоколоночном списке.
- `DataSource` — задает источник данных, с помощью которого можно заполнять список. Существуют два способа:
  - использовать метод `Add()` — в этом случае свойство `DataSource` должно быть отключено;
  - подключаться к различным источникам данных, доступ к которым формируется через диалоговое окно, открывающееся кнопкой, расположенной в поле этого свойства. Пока в версии Beta в среде отсутствует возможность работы с источником данных.
- `Items` — это элементы компонента, которые можно не только просматривать, но и изменять. Для задания списка строк-элементов надо щелкнуть на кнопке

с многоточием в поле этого свойства, чтобы открылось окно редактора, позволяющего вводить и редактировать строки (рис. 11.45).

Извлечь строки из компонента можно так:

```
String ^it= ListBox1->Items[i]->ToString();
```

где `i` — это номер строки (начинается с нуля).

Обнаружить строку, на которой был щелчок мыши (обработка события `Click`), можно так:

```
String ^it = this-> listBox1-> Items[this-> listBox1-> SelectedIndex] ->
ToString();
```

где `SelectedIndex` — индекс выбранной строки.

- `MultiColumn` — обеспечивает компоненту работу в многоколоночном режиме (т. е. набор строк, не помещающийся в окно, будет размещаться в новых колонках). Вид `ListBox` в многоколоночном режиме показан на рис. 11.46.

Рис. 11.45. Окно редактора для ввода строк в компонент `ListBox`

Рис. 11.46. Вид компонента `ListBox` в многоколоночном режиме

Пример использования этого свойства приведен в листинге 11.4.

Листинг 11.4

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)

this->listBox1->Items->Clear(); //очистка компонента
 String ^it;
 listBox1->MultiColumn = false;  //одноколоночный режим

 for (int i = 0; i < 50; i++)
  {
   it= Convert::ToString(i); //так преобразуется int to String ^
   it=it->Concat("Items_",it);
   listBox1->Items->Add(it);
   }
  } //конец обработчика
```

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
      // Add items to the ListBox.
  this->listBox1->Items->Clear();
      for ( int x = 0; x < 50; x++ )
       {
        String ^it= Convert::ToString(x);
         //так преобразуется int to String ^
           it=it->Concat("Items_",it);
      listBox1->Items->Add(it);
 }
      listBox1->MultiColumn = true; //многоколоночный режим
      listBox1->ColumnWidth = 50; //это ширина колонки в пикселах

  /*Чтобы автоматически выводить любой текст в многоколоночном варианте,
    надо уметь рассчитать максимальную длину строки в пикселах,
    т. е. уметь переводить символы в пикселы.*/
```

Определение максимальной ширины строки в пикселах для нашего примера:

```
int width = (int)listBox1-> CreateGraphics()-> MeasureString(listBox1->
Items[listBox1-> Items->Count — 1]-> ToString(), listBox1-> Font).Width;
      listBox1-> ColumnWidth = width;
```

Здесь метод `CreateGraphics()` создает графический объект для `listBox1` (вывод строк происходит в графике).

У этого объекта есть метод `MeasureString(String,Font)`, который измеряет длину строки в пикселах, выводимую данным шрифтом (разные шрифты занимают на экране разное количество пикселов).

Параметр `String` — это:

```
listBox1->Items[listBox1->Items->Count — 1]->ToString()
```

для нашего примера это последний (самый длинный) элемент.

`i`-й элемент извлекается так:

```
String ^it=listBox1->Items[i]->ToString();
```

Параметр `Font` — это:

```
listBox1->Font
```

Метод `MeasureString(String,Font)` возвращает данные по структуре типа `SizeF`, которая состоит из двух элементов типа `float`. Это координаты прямоугольника (в пикселах), куда помещается изображение строки на экране. Нам надо знать ширину этого прямоугольника, т. к. она определяет размер строки в пикселах. Поэтому мы пишем:

```
MeasureString(String,Font).width
```

забирая из структуры только один элемент.

Но поскольку этот элемент имеет тип `float`, то его надо перевести в `int`, потому что ширина в `ListBox` задается в `int`.

Поэтому перед `ListBox1->CreateGraphics()` стоит `(int)` — принудительный перевод `float` в `int`.

В общем случае надо определять строку максимальной длины, образуя цикл:

```
String ^it;
String ^it_0; //здесь будет предыдущая строка
it_0=listBox1->Items[0]->ToString();
for(int i=0; i < listBox1->Items->Count; i++)
 {
  it=listBox1->Items[i]->ToString();
/*
и далее сравнивать текущую и предыдущую строки, применяя метод сравнения для
строк типа String ^ и выбирая из них большую
*/
 }
```

Как использовать *ListBox*

Компонент создает прямоугольную область, в которой отображается список текстовых строк. Эти текстовые строки можно добавлять в список, выбирать или удалять из него. 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`, а если необходимо ввести какую-либо должность в базу данных, то достаточно открыть список должностей и щелкнуть на требуемой должности, как соответствующее наименование "перекочует" в базу данных.

Как формировать список строк

На этапе разработки приложения можно сформировать так называемый отладочный список, который в дальнейшем (когда приложение будет эксплуатироваться) неудобно поддерживать в актуальном состоянии, т. к. требуется корректировка списка и перекомпиляция приложения. Но, тем не менее, для отладочных работ список надо сформировать. Это делается с помощью редактора текста, окно которого открывается, если щелкнуть на кнопке с многоточием в поле свойства компонента `Items`.

Для производственных же нужд требуется создать функцию, которая бы загружала текстовый файл в компонент (мы разбираем только случай, когда элементами `ListBox` являются текстовые строки). Воспользоваться возможностями данной среды разработки для работы с базами данных пока не представляется возможным: работа с источниками данных в данной версии среды отключена.

Поэтому для нашей задачи вернемся к первому способу: создадим функцию, которая станет загружать текстовые строки из обыкновенного текстового файла, подготовленного известным и простейшим инструментом — программой WordPad. После создания нашей программы применим ее для загрузки строк и попробуем вывести введенное множество строк в компонент `TextBox`. Вид приложения в режиме дизайна показан на рис. 11.47, текст приложения — в листинге 11.5. Результат работы приложения приведен на рис. 11.48.

Рис. 11.47. Вид приложения в режиме дизайна

Листинг 11.5

```
#pragma once

namespace Proverka_vvoda_v_CPP {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::IO;
    using namespace System::Text;

    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
```

```
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
    private: System::Windows::Forms::Button^  button1;
    private: System::Windows::Forms::ListBox^  listBox1;
    private: System::Windows::Forms::TextBox^  textBox1;
    private: System::Windows::Forms::Button^  button2;
    protected:

    private:
        // <summary>
        // Required designer variable.
        int fix;
        StreamWriter ^sw;
    private: System::Windows::Forms::Button^  button3;
              // </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        // <summary>
        // Required method for Designer support — do not modify
        // the contents of this method with the code editor.

        // </summary>
        void InitializeComponent(void)
        {
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->listBox1 = (gcnew System::Windows::Forms::ListBox());
            this->textBox1 = (gcnew System::Windows::Forms::TextBox());
            this->button2 = (gcnew System::Windows::Forms::Button());
            this->button3 = (gcnew System::Windows::Forms::Button());
this->SuspendLayout();
            //
            // button1
            //
```

```
this->button1->Location = System::Drawing::Point(24, 12);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(287, 23);
this->button1->TabIndex = 0;
this->button1->Text = L"Читать текстовый файл";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
            //
            // listBox1
            //
this->listBox1->FormatString = L"/";
this->listBox1->FormattingEnabled = true;
this->listBox1->ItemHeight = 16;
this->listBox1->Location = System::Drawing::Point(24, 55);
this->listBox1->Name = L"listBox1";
this->listBox1->Size = System::Drawing::Size(287, 100);
this->listBox1->TabIndex = 1;
            //
            // textBox1
            //
this->textBox1->Location = System::Drawing::Point(24, 161);
this->textBox1->Name = L"textBox1";
this->textBox1->Size = System::Drawing::Size(287, 22);
this->textBox1->TabIndex = 2;
            //
            // button2
            //
this->button2->Location = System::Drawing::Point(24, 190);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(287, 23);
this->button2->TabIndex = 3;
this->button2->Text = L"Запись данных в текстовый файл\r\n";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
            //
            // button3
            //
this->button3->Location = System::Drawing::Point(328, 55);
this->button3->Name = L"button3";
this->button3->Size = System::Drawing::Size(25, 128);
this->button3->TabIndex = 4;
this->button3->Text = L"Выход";
this->button3->UseVisualStyleBackColor = true;
this->button3->Click += gcnew System::EventHandler(this, &Form1::button3_Click);
            //
            // Form1
            //
```

```
this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
this->ClientSize = System::Drawing::Size(378, 234);
this->Controls->Add(this->button3);
this->Controls->Add(this->button2);
this->Controls->Add(this->textBox1);
this->Controls->Add(this->listBox1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->Shown += gcnew System::EventHandler(this, &Form1::Form1_Shown);
this->ResumeLayout(false);
this->PerformLayout();

         }
#pragma endregion
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
//файл должен быть записан WordPad'ом как текстовый в кодировке Юникод
   String^ path = "d:\\for_write_ListBox.txt";
   if (  !File::Exists( path ) )
    {
      // Create a file to write to
      sw = File::CreateText( path ); // StreamWriter^
      try
      {
         sw->WriteLine( "Hello" );  //Это данные для контроля ввода.
         sw->WriteLine( "And" );    //Если читаемый файл не найден,
         sw->WriteLine( "Welcome" );//эти данные выведутся
      }
      finally
      {
         if ( sw )
          delete (IDisposable^)(sw);
      }
   }

   // Open the file to read from
   TextReader ^ sr = File::OpenText( path );
   try
    {
      String^ s = "";
      while ( s = sr->ReadLine() )
       {
     this->listBox1->Items->Add(s);
       }
   }
```

```
   finally
   {
     if ( sr )
      delete (IDisposable^)(sr);
   }
  }
 private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
  {
    // Create a file to write to
    if(fix != 1)
     {
       String^ path = "d:\\for_write_ListBox.txt";
       sw = File::CreateText( path ); //StreamWriter ^
      }
    if(this->textBox1->Text != "#")
     {
         sw->WriteLine(this->textBox1->Text);
         textBox1->Text="";
      fix=1;
   this->textBox1->Focus();
   goto m;
  }
   else
   sw->Close();
m:;
  }

private: System::Void Form1_Shown(System::Object^  sender, System::EventArgs^ e)
{
    //При первом появлении формы фокус ввода передается на запись
 this->textBox1->Focus();
}
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^ e)
{
 this->Close();
}
};
}
```

Поясним листинг 11.5. В список имен пространств надо добавить пространства

```
using namespace System::IO;
using namespace System::Text;
```

для работы с файлами. В секцию для переменных дизайнера надо добавить объявление переменных

```
int fix;
StreamWriter ^sw;
```

Кроме того, чтобы было удобно вводить данные в файл, после появления формы курсор должен быть установлен на ввод строк, т. е. элемент `TextBox` должен получить фокус ввода. Это делается обработкой события `Shown`, которое возникает, когда форма впервые появляется на экране. Пояснения, в основном, даны по тексту программы. Признак конца ввода всех строк текста — символ "#". Отметим, что данный вариант программы далек от совершенства и может рассматриваться только в качестве примера: здесь надо было бы имена файлов вводить извне (с этим мы познакомимся, когда начнем изучать компоненты из группы **Dialogs**). Ввод строк для их записи в файл сделан, как раньше говорили, по рабоче-крестьянски: надо было бы ввод в строку заканчивать нажатием клавиши <Enter>, а не выдумывать признак конца строки ввода и специальную кнопку, которую надо нажимать после ввода строки в `TextBox`, чтобы строка записалась в файл, что очень неудобно. Но для этого надо уметь отлавливать код клавиши, которая была нажата в данный момент. В этой среде автору это не удалось: в заголовке обработчика события `OnClick` формируется параметр `System::EventArgs^ e`. Класс `EventArgs` в предыдущих версиях среды программирования как раз и содержал код текущей нажатой клавиши, которую сравнивали с клавишей <Enter> или любой другой клавишей. В используемой сегодня версии среды этот класс почти полностью выхолощен и содержит один элемент под названием `Empty`. Может в дальнейшем разработчики VC++ найдут возможность исправить эту и не только эту досадную ошибку.

Результат работы программы, приведенной в листинге, показан на рис. 11.48.

Рис. 11.48. Ввод и вывод с использованием `ListBox`

Компонент *ComboBox*

Компонент находится в списке **All Windows Forms** палитры компонентов. Этот компонент является комбинацией редактируемого поля и списка `ListBox`: в форме он представляется в виде редактируемого поля с треугольной кнопкой справа. Компонент `ComboBox` используется для вывода данных в виде выпадающего списка и последующей выборки их из этого списка. По умолчанию `ComboBox` появляется в виде окна для ввода/вывода текста (аналог однострокового `TextBox`), при этом

Рис. 11.49. Компонент `ComboBox` в форме

выпадающий список скрыт. Он является также аналогом компонента `ListBox`, из которого пользователь может выбирать элементы (рис. 11.49).

Свойства *ComboBox*

Перечень свойств компонента, отображенных в его окне **Properties**, показан на рис. 11.50.

Многие свойства нам уже знакомы. Особенно большое совпадение со свойствами компонента `ListBox`, что вполне естественно, т. к. `ComboBox` является комбинацией `ListBox` и `TextBox`. Однако, что тоже вполне естественно, имеются и чисто специфические свойства.

- `Items` — содержит набор строк `ComboBox`. Это свойство можно как задавать в режиме дизайна, открыв диалоговое окно редактора кнопкой с многоточием в поле

Рис. 11.50. Перечень свойств компонента `ComboBox`

этого свойства, чтобы ввести туда необходимые строки, так и программно формировать. Если некоторая строка отмечена в `ComboBox`, то ее индекс помещается в свойство `SelectedIndex`.

- `SelectedIndex` — это свойство не показано в окне **Properties**. Это целочисленная переменная, изменяющаяся от нуля (т. е. первая строка `ComboBox` будет иметь индекс, равный нулю, вторая — единице и т. д.). Если кому не нравится произносить слово "индекс (указатель)", то можно назвать его просто номером строки с учетом его отсчета от нуля. Можно программно изменять выбранный из `ComboBox` элемент, изменять значение `SelectedIndex`. При этом в списке будет отмечаться новый элемент, соответствующий новому значению `SelectedIndex`. Пока ни один элемент из `ComboBox` не выбран, значение `SelectedIndex` равно –1.
- `SelectedItem` — свойство, сходное со свойством `SelectedIndex`, только оно возвращает выбранный элемент (обычно это строка). Это свойство не показано в окне **Properties**.
- `Count` — свойство, содержащее количество элементов в списке `ComboBox`. Расчет количества ведется от 1 (если в списке 10 строк, то `Count` будет равен 10). Это свойство не показано в окне **Properties**.
- `DropDownStyle` — свойство, задающее стиль вывода данных компонентом. Может принимать значения:
  - `Simple` — в этом случае работает поле редактирования, а кнопка раскрытия списка спрятана (можно только вводить строку данных);
  - `DropDown` — в этом случае стрелка раскрытия списка видна, и с ее помощью можно раскрыть список, выбрать строку, которая попадет в поле редактирования, где ее можно отредактировать, прежде чем использовать далее;
  - `DropDownList` — выборку из списка можно делать, но выбранную строку уже редактировать нельзя.
- `Text` — свойство, содержащее значение поля редактирования компонента (т. е. из списка можно извлекать строку и редактировать ее).
- `DropDownWidth` и `DropDownHeight` — ширина и высота выпадающего списка. Если последнее свойство таково, что окно списка не вмещает весь список, то в окне появится полоса прокрутки. Если же окно по размеру больше списка, то при выводе окно примет размер списка.
- `FlatStyle` — стиль окна редактирования. Если задать это свойство в виде `PopUp`, то при наведении курсора мыши на окно, оно "всплывет", что весьма удобно при контроле за движением курсора мыши. Если свойству придать значение `System`, то при наведении курсора мыши на окно стрелка, раскрывающая список, изменит цвет.
- `FormatString` — с помощью этого свойства можно задавать форматы вывода некоторых типов данных (чтобы задать формат, надо посредством кнопки с многоточием открыть диалоговое окно и выбрать подходящий формат для выводимых

строк). При этом надо помнить, что элементы списка должны быть соответствующего типа (датами, данными по валюте и т. п.).

- `AutoCompleteCustomSource` — это свойство совместно со свойствами `AutoCompleteMode` и `AutoCompleteSource` обеспечивает подсказку с выбором значения из списка для вводимых строк.

События *ComboBox*

Перечень событий компонента, отображаемых в его окне **Properties**, показан на рис. 11.51.

Рис. 11.51. Перечень событий компонента `ComboBox`

Из всех событий компонента нас в первую очередь интересует то, которое наступает, если в выпадающем списке щелкнуть на выбранной строке. При этом мы должны будем попасть в обработчик этого события, чтобы в нем извлечь из списка нужную нам строку. Это событие — `DropDownClosed`. Действительно, событие возникает, когда закрывается выпадающий список. А он закрывается именно после щелчка на какой-либо его строке либо после щелчка на поле редактирования (в этом случае никакой выборки не произведено и поэтому свойство `SelectedIndex` устанавливается в –1, следовательно, эту ситуацию в обработчике тоже надо учитывать).

Некоторые методы *ComboBox*

Компонент имеет большое количество методов, которые можно посмотреть в справочной системе, нажав клавишу <F1> при подсвеченном в редакторе текста имени этого объекта. Отметим только часто применяемые, такие как:

- `Focus()` — передает фокус ввода компоненту: компонент становится активным и с ним можно работать;
- `Hide()` — делает объект невидимым (прячет его).

Чтобы в режиме проектирования добавлять элемент в список, надо воспользоваться свойством `Items` — при нажатии на кнопку с многоточием в поле этого свойства открывается диалоговое окно для задания множества строк (окно, аналогичное такому же окну для `ListBox`). Если же вам потребуется обеспечить автоматическую загрузку поля этого компонента, то следует воспользоваться методами свойства `Items`, которое само является классом и потому обладает своими методами.

После стрелки откроется окно подсказчика.

Основные методы, употребляемые для загрузки `ComboBox`, — следующие (фактически это методы класса `Items`, в `ComboBox` вы их не найдете):

- `Clear()` — очищает поле `ComboBox`;
- `Add()` — добавляет элемент в конец поля `ComboBox`;
- `IndexOf()` — выдает индекс строки в поле `ComboBox`. Строка задается в аргументе метода. Если строка не найдена, то выдается –1;
- `Insert()` — вставляет строку в поле `ComboBox` перед указанным в аргументе индексом.

  Например, следующие операторы, помещенные в обработчик кнопки,

  ```
  this->comboBox1->Items->Clear();
  this->comboBox1->Items->Add("Строка1");  //индекс этой строки равен 0
  this->comboBox1->Items->Add("Строка2");  //индекс этой строки равен 1
  this->comboBox1->Items->Insert(1,"Insert");
  ```

  дают такой результат:

  ```
  Строка1
  Insert
  Строка2
  ```

  Из примера видно, что индекс в `ComboBox` изменяется от нуля. Это можно проверить и методом `IndexOf()`, выполнив после указанных выше операторов оператор:

  ```
  int i=this->comboBox1->Items->IndexOf("Строка1");
  ```

  Значение `i` будет равно 0.

- `Remove()` — удаляет строку, указанную в аргументе, из поля `ComboBox`. Например, выполним оператор на множестве трех предыдущих строк:

  ```
  this->comboBox1->Items->Remove("Insert");
  ```

  Получим результат:

  ```
  Строка1
  Строка2
  ```

- `RemoveAt()` — удаляет строку из поля `ComboBox`, индекс которой указан в аргументе.

  Например, выполним оператор:

  ```
  this->comboBox1->Items->RemoveAt(1);
  ```

на множестве трех строк:

```
Строка1
Insert
Строка2
```

Получим такой результат:

```
Строка1
Строка2
```

Для работы с `ComboBox` широко применяется и свойство `Count` из `Items`. В нем всегда находится количество элементов `ComboBox`.

Например, если выполнить оператор:

```
int j=this->comboBox1->Items->Count;
```

на множестве строк:

```
Строка1
Строка2
```

то в переменной `j` получим 2.

Заметим, что в поле `ComboBox` можно располагать не только строки, но и другие объекты.

Примеры использования *ComboBox*

Компонент используется для просмотра и выборки элементов.

Пример 1

Допустим, что вам требуется каждый раз просматривать сведения об авторах книг в библиотеке. Воспользоваться источником данных, который связывается с базой данных, мы не можем в данной версии среды: работа с источником данных отключена. Воспользуемся возможностью работы с файлом данных, которую мы рассматривали при изучении компонента `ListBox`. На рис. 11.52 показан пример рабо-

Рис. 11.52. Работа `ComboBox` с текстовым файлом

ты ComboBox с внешним файлом, формируемым из вашего приложения, а потом просматриваемым в ComboBox.

Текст программы приведен в листинге 11.6. Пояснения — те же, что и для программы по работе ListBox.

Листинг 11.6

```
#pragma once

namespace ComboBox2011 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::IO;
    using namespace System::Text;


    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
```

```
    private: System::Windows::Forms::ComboBox^  comboBox1;
    private: System::Windows::Forms::TextBox^  textBox1;
    private: System::Windows::Forms::Button^  button1;
    protected:

    private:
        /// <summary>
        /// Required designer variable.

        int fix;
        StreamWriter ^sw;

        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->comboBox1 = (gcnew System::Windows::Forms::ComboBox());
            this->textBox1 = (gcnew System::Windows::Forms::TextBox());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->SuspendLayout();
            //
            // comboBox1
            //
            this->comboBox1->FormattingEnabled = true;
            this->comboBox1->Items->AddRange(gcnew cli::array< System::
                                    Object^  >(3) {L"111", L"222", L"333"});
            this->comboBox1->Location = System::Drawing::Point(13, 13);
            this->comboBox1->Name = L"comboBox1";
            this->comboBox1->Size = System::Drawing::Size(121, 24);
            this->comboBox1->TabIndex = 0;
            this->comboBox1->Click += gcnew System::EventHandler(this,
&Form1::comboBox1_Click);
            //
            // textBox1
            //
            this->textBox1->Location = System::Drawing::Point(13, 63);
            this->textBox1->Name = L"textBox1";
            this->textBox1->Size = System::Drawing::Size(159, 22);
            this->textBox1->TabIndex = 1;
            //
            // button1
            //
```

```
            this->button1->Location = System::Drawing::Point(13, 101);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(159, 30);
            this->button1->TabIndex = 2;
            this->button1->Text = L"Запись в файл";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
                                                    &Form1::button1_Click);
            //
            // Form1
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(229, 163);
            this->Controls->Add(this->button1);
            this->Controls->Add(this->textBox1);
            this->Controls->Add(this->comboBox1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->Shown += gcnew System::EventHandler(this,
&Form1::Form1_Shown);
            this->ResumeLayout(false);
            this->PerformLayout();

        }
#pragma endregion
    private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
     {
       // Create a file to write to
    if(fix != 1)
     {
       String^ path = "d:\\for_write_ListBox.txt";
       sw = File::CreateText( path ); //StreamWriter ^
      }
    if(this->textBox1->Text != "#")
     {
         sw->WriteLine(this->textBox1->Text);
         textBox1->Text="";
      fix=1;
   this->textBox1->Focus();
   goto m;
  }
   else
   sw->Close();
m:;

     }
```

```
    private: System::Void Form1_Shown(System::Object^  sender,
System::EventArgs^  e)
     {
       this->textBox1->Focus();
     }
private: System::Void comboBox1_Click(System::Object^  sender,
System::EventArgs^  e)
 {
   String^ path = "d:\\for_write_ListBox.txt";
   if (  !File::Exists( path ) )
    {
      // Create a file to write to
      sw = File::CreateText( path ); // StreamWriter^
      try
      {
         sw->WriteLine( "Hello" ); //Это данные для контроля ввода.
         sw->WriteLine( "And" );    //Если читаемый файл не найден,
         sw->WriteLine( "Welcome" );//эти данные выведутся
      }
      finally
      {
         if ( sw )
          delete (IDisposable^)(sw);
      }
   }
   //очистка comboBox1:
    this->comboBox1->Items->Clear();
   // Open the file to read from
   TextReader ^ sr = File::OpenText( path );
   try
    {
      String^ s = "";
      while ( s = sr->ReadLine() )
       {
     this->comboBox1->Items->Add(s);
       }
   }
   finally
   {
     if ( sr )
      delete (IDisposable^)(sr);
   }
 }
};
}
```

Пример 2

Допустим, что вы создаете приложение для обработки данных по рекрутингу — набору рабочей силы для предприятий города. Естественно, что нужно обрабатывать анкетные данные клиента, данные предприятий, которые запрашивают помощь в наборе работников. И везде требуется работа со справочными данными. Например, надо выбрать из перечня должностей требуемую должность. Конечно же, такой перечень следует ввести в `ComboBox` и из него делать выборку.

В примере 1 мы научились, как загружать строки из текстового файла (программное формирование свойства `Items`). Здесь мы покажем, как выполнять выборку строк и засылать выбранную строку в `TextBox`. Результат работы приложения показан на рис. 11.53, а текст программы приведен в листинге 11.7.

Рис. 11.53. Загрузка текстового файла в `ComboBox`, выборка строки из `ComboBox` и пересылка ее в `TextBox`

Листинг 11.7

```
#pragma once

namespace ComboBox22011 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::IO;
    using namespace System::Text;


    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
```

```
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
    private: System::Windows::Forms::ComboBox^  comboBox1;
    protected:
    private: System::Windows::Forms::TextBox^  textBox1;
    private: System::Windows::Forms::Button^  button1;

    private:
        /// <summary>
        /// Required designer variable.
        int fix;
        StreamWriter ^sw;

        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->comboBox1 = (gcnew System::Windows::Forms::ComboBox());
            this->textBox1 = (gcnew System::Windows::Forms::TextBox());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->SuspendLayout();
            //
            // comboBox1
            //
            this->comboBox1->FormattingEnabled = true;
```

```
            this->comboBox1->Location = System::Drawing::Point(13, 13);
            this->comboBox1->Name = L"comboBox1";
            this->comboBox1->Size = System::Drawing::Size(121, 24);
            this->comboBox1->TabIndex = 0;
            this->comboBox1->DropDownClosed += gcnew
System::EventHandler(this, &Form1::comboBox1_DropDownClosed);
            this->comboBox1->Click += gcnew System::EventHandler(this,
&Form1::comboBox1_Click);
            //
            // textBox1
            //
            this->textBox1->Location = System::Drawing::Point(13, 68);
            this->textBox1->Name = L"textBox1";
            this->textBox1->Size = System::Drawing::Size(121, 22);
            this->textBox1->TabIndex = 1;
            //
            // button1
            //
            this->button1->Location = System::Drawing::Point(35, 101);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(75, 23);
            this->button1->TabIndex = 2;
            this->button1->Text = L"Выход";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
            //
            // Form1
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(153, 136);
            this->Controls->Add(this->button1);
            this->Controls->Add(this->textBox1);
            this->Controls->Add(this->comboBox1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->ResumeLayout(false);
            this->PerformLayout();

        }
#pragma endregion
    private: System::Void comboBox1_Click(System::Object^  sender,
System::EventArgs^  e)
     {
         //Засылка строк из текстового файла в ComboBox
         String^ path = "d:\\for_write_ListBox.txt";
```

```
   if (  !File::Exists( path ) )
    {
      // Create a file to write to
      sw = File::CreateText( path ); // StreamWriter^
      try
      {
         sw->WriteLine( "Hello" );  //Это данные для контроля ввода.
         sw->WriteLine( "And" );    //Если читаемый файл не найден,
         sw->WriteLine( "Welcome" );//эти данные выведутся
      }
      finally
      {
         if ( sw )
          delete (IDisposable^)(sw);
      }
   }
   //очистка comboBox1:
    this->comboBox1->Items->Clear();
   // Open the file to read from
   TextReader ^ sr = File::OpenText( path );
   try
    {
      String^ s = "";
      while ( s = sr->ReadLine() )
       {
     this->comboBox1->Items->Add(s);
       }
   }
   finally
   {
     if ( sr )
      delete (IDisposable^)(sr);
   }
  }

 private: System::Void comboBox1_DropDownClosed(System::Object^  sender,
System::EventArgs^  e)
  {
      //Обработка щелчка на выбранной строке выпадающего меню
     //Выборка из списка и пересылка в TextBox
  if(this->comboBox1->SelectedIndex == -1)
    return;
  this->textBox1->Text=this->comboBox1->Items[this->comboBox1->SelectedIndex]-
>ToString();

  }
```

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
  this->Close();
 }
};
}
```

Пояснения — по тексту программы и остальные — из пояснений к ListBox.

Пример 3

Покажем, как использовать ComboBox для работы не со строками, а с графическими объектами. Предположим, что требуется создать список для выбора уровней опасности, скажем, обстановки в городе.

Нам надо будет в поле ComboBox расположить три цветных полосы: красную, желтую и зеленую, которые соответствуют высшему, среднему и низкому уровням опасности соответственно. Поэтому надо построить работу с ComboBox так, чтобы при выборе красной полосы в поле редактирования попадало сообщение "Высший уровень опасности", при выборе желтой полосы — "Средний уровень опасности" и т. д.

Сам компонент построим в программе, т. е. создадим его, а затем свойствам присвоим необходимые значения. Как строятся подобные списки? Все определяется значением свойства DrawMode (способ рисования элементов ComboBox). Если значение этого свойства равно Normal, то в качестве объектов рисования служат строки. Примеры работы с ними мы рассмотрели. В этом случае среда сама добавляемые элементы переводит в текстовые строки.

Если же значение DrawMode не равно Normal, то предполагается, что пользователь (т. е. вы) сам должен написать операторы построения объектов со всеми их характеристиками (например, с цветом, шрифтом и т. п.). Причем операторы должны быть помещены в обработчики событий DrawItem (рисование элемента) и MeasureItem (размеры элемента). Как это делается, показано в листинге 11.8.

На этапе дизайна компонент ComboBox был помещен в форму. Затем его свойствам были установлены значения, приведенные в табл. 11.4.

Таблица 11.4. *Значения свойств компонента* ComboBox

Свойства	Значения
Items	Высший уровень опасности Средний уровень опасности Низкий уровень опасности
Location	10;20
Size	200;500
DropDownWidth	150

Таблица 11.4 *(окончание)*

Свойства	Значения
DropDownHeight	300
DropDownStyle	DropDown

Затем были сформированы обработчики событий DrawItem и MeasureItem. Остальные пояснения — в комментариях листинга 11.8.

Листинг 11.8

```
#pragma once

namespace ComboBox32011 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
```

```
    private: System::Windows::Forms::ComboBox^  comboBox1;
    protected:

    protected:

    protected:

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->comboBox1 = (gcnew System::Windows::Forms::ComboBox());
            this->SuspendLayout();
            //
            // comboBox1
            //
            this->comboBox1->DrawMode =
System::Windows::Forms::DrawMode::OwnerDrawVariable;
            this->comboBox1->DropDownHeight = 300;
            this->comboBox1->DropDownWidth = 150;
            this->comboBox1->FormattingEnabled = true;
            this->comboBox1->IntegralHeight = false;
            this->comboBox1->Items->AddRange(gcnew cli::array< System::
Object^  >(3) {L" Высший уровень опасности", L" Средний уровень опасности",
              L" Низкий уровень опасности"});
            this->comboBox1->Location = System::Drawing::Point(10, 20);
            this->comboBox1->Name = L"comboBox1";
            this->comboBox1->Size = System::Drawing::Size(200, 23);
            this->comboBox1->TabIndex = 1;
            this->comboBox1->DrawItem += gcnew
System::Windows::Forms::DrawItemEventHandler(this,
&Form1::comboBox1_DrawItem_1);
            this->comboBox1->MeasureItem += gcnew
System::Windows::Forms::MeasureItemEventHandler(this,
&Form1::comboBox1_MeasureItem_1);
            //
            // Form1
            //
```

```
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(251, 178);
            this->Controls->Add(this->comboBox1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->ResumeLayout(false);

        }
#pragma endregion
//-----------------------------------------------------------------------------

private: System::Void comboBox1_MeasureItem_1(System::Object^  sender,
System::Windows::Forms::MeasureItemEventArgs^  e)
    {
/*
Если вы установили значение свойства DrawMode в OwnerDrawVariable,
то должны обработать событие MeasureItem. В его обработчике надо установить
размеры элемента ComboBox: его высоту (height) и ширину (width) перед тем, как
этот элемент будет прорисован
*/

switch ( e->Index )//выдается индекс элемента, для которого вычисляются
                   //высота и ширина
      {
         case 0:
            e->ItemHeight = 40;  //высота для элемента с индексом 0
            break;
         case 1:
            e->ItemHeight = 30;
            break;
         case 2:
            e->ItemHeight = 20;
            break;
      }
      e->ItemWidth = 100; //ширина
   } //метод
private: System::Void comboBox1_DrawItem_1(System::Object^  sender,
System::Windows::Forms::DrawItemEventArgs^  e)
 {
/*
Событие наступает, когда мы щелкаем на кнопке открытия списка.
В этом обработчике идет прорисовка элементов ComboBox. Для случая нашего
примера надо нарисовать прямоугольники (они будут служить элементами списка)
и закрасить их в разные цвета. Кроме того, надо будет задать характеристики
шрифта для вывода наименований выбранных элементов
*/
```

```
      float size = 0;
      System::Drawing::Font^ myFont;
      FontFamily^ family = nullptr;

      System::Drawing::Color DangerColor;
      switch ( e->Index )
      {
         case 0:
            size = 20;
            DangerColor = System::Drawing::Color::Red;
            family = FontFamily::GenericSansSerif;
            break;
         case 1:
            size = 30;
            DangerColor = System::Drawing::Color::Gold;
            family = FontFamily::GenericMonospace;
            break;
         case 2:
            size = 40;
                   DangerColor = System::Drawing::Color::LawnGreen;
            family = FontFamily::GenericSansSerif;
            break;
      }

      // Рисование фона элемента
      e->DrawBackground();

/* Создание прямоугольника и заполнение его цветами уровней опасности.
Изменение размеров прямоугольников, основанных на длинах имени каждого элемента
*/
      Rectangle rectangle = Rectangle( 2, e->Bounds.Top + 2,
         e->Bounds.Height, e->Bounds.Height — 4 );
      e->Graphics->FillRectangle( gcnew SolidBrush( DangerColor ), rectangle );

/*
Создание текста для каждого элемента с использованием размеров, цвета и шрифта
каждого элемента
*/
      myFont = gcnew System::Drawing::Font( family, size, FontStyle::Bold );

/*
 Создание фокуса ввода для каждого элемента (т. е. для прямоугольника).
Если курсор мыши появляется над элементом, то при передаче фокуса элементу, он
подсвечивается
*/
      e->DrawFocusRectangle();
   }
  };
}
```

Результат работы показан на рис. 11.54.

Рис. 11.54. Задание и обработка элементов `ComboBox` в виде цветных прямоугольников (полос)

Компонент *MaskedTextBox*

Компонент находится в списке **All Windows Forms** палитры компонентов. 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, то этот символ системой контроля будет отвергнут. Компонент использует специальный синтаксис для объявления маски. Маска задается в свойстве `Mask`. Существуют стандартные маски, их перечень открывается в диалоговом окне, в которое можно войти из свойства `Mask`. Но можно и самому задать маску в этом свойстве, пользуясь специальными символами.

В табл. 11.5 приведен перечень специальных символов, задающих маску.

Таблица 11.5. *Перечень специальных символов, задающих маску*

Маскирующий символ	Описание
`0`	Указывает, что на этом месте должна быть цифра, обязательная к вводу (любая цифра от нуля до девяти)
`9`	Указывает, что на этом месте может быть (но не обязательно) цифра или пробел
`#`	Указывает, что на этом месте может быть (но не обязательно) цифра или пробел. Если эта позиция в маске не будет заполнена, то в свойство `Text` выводится пробел. Допускаются знаки + и –
`L`	Указывает, что на этом месте должна быть (обязательно) буква и только из диапазона ASCII

Таблица 11.5 *(окончание)*

Маскирующий символ	**Описание**
?	Указывает, что на этом месте может быть (не обязательно) буква и только из диапазона ASCII
&	Указывает, что на этом месте должен быть (обязательно) символ. Если свойство AsciiOnly установлено в true, то этот элемент ведет себя как элемент L
C	Указывает, что на этом месте может быть (не обязательно) любой символ. Если свойство AsciiOnly установлено в true, то этот элемент ведет себя как элемент ?
A	Указывает, что на этом месте может быть (не обязательно) любая буква или цифра. Если свойство AsciiOnly установлено в true, то не отвергаются только символы кода ASCII (a-z и A-Z)
a	Указывает, что на этом месте может быть (не обязательно) любая буква или цифра. Если свойство AsciiOnly установлено в true, то не отвергаются только символы кода ASCII (a-z и A-Z)
. (точка)	Разделитель десятичный
, (запятая)	Разделитель тысяч
:	Разделитель времени
/	Разделитель даты
$	Символ валюты
<	Преобразует все символы к нижнему регистру
>	Преобразует все символы к верхнему регистру
|	Запрещает влияние элементов < или >
\	Теряется маска-символ. Он заменяется символом-литералом (например, маска 00_0\0 заменяется на маску ____0). Элемент \\ — это элемент \ для обратного слэша
Все остальные символы	Все немаскированные символы появляются в поле ввода. Символы-литералы всегда занимают в маске статическую позицию в режиме исполнения приложения и не могут быть передвинуты или удалены пользователем

Используя маску, можно без написания специальных участков программы в приложении добиться следующего:

- вводить символы заданного типа и никакие другие (например, только цифры);
- обеспечить обязательный ввод символов в поле ввода;
- за счет ввода в маску специальных символов-литералов, которые будут составлять неотъемлемую часть введенной строки, обеспечить ускорение ввода информации для пользователя (например, дефис в телефонном номере, точки в дате, знак валюты в цене и др.);

- задавать специальный ввод, когда будут автоматически преобразованы, например, символы нижнего регистра в верхний и т. д.

В маске содержится символ-подсказка (он задается в свойстве `PromptChar`), на место которого пользователь должен вводить символы-данные. Например, маска даты может иметь вид:

"__.__.____"

Здесь символом-подсказкой является подчеркивание, а символом-литералом — точка. Вместо символа-подсказки надо вводить цифры даты (например, 19.03.2012). Точку (символ-литерал) вводить не надо — она автоматически пропускается при наборе цифр.

Вид `MaskedTextBox` в форме и пример выбора маски показан на рис. 11.55.

Рис. 11.55. Вид `MaskedTextBox` в форме и пример выбора маски

Свойства *MaskedTextBox*

Перечень свойств показан на рис. 11.56, описания некоторых свойств приведены далее.

- `PromptChar` — предоставляет возможность задавать свой собственный символ маски. Вместо подчеркивания можно задать, например, символ *. Тогда маска даты станет выглядеть так: **.**.****. Это будет более наглядно, т. к. знак подчеркивания сливается в сплошную линию.
- `HidePromptOnLeave` — предоставляет возможность видеть символы-литералы даже в то время, когда сам компонент теряет фокус ввода.

Ввод по маске осуществляется так: действительно вводимые символы заменяют собой символы маски, а постоянно заданные символы (например, точки в дате) пропускаются, когда указатель ввода в поле доходит до них. Система просто передвигает указатель ввода на следующий символ-подсказку. Если же пользователь вводит запрещенный маской символ, то возникает событие `MaskInputRejected` (отвергнутый маской ввод), позволяющее в его обработчике осуществить какую-то пользовательскую реакцию.

Рис. 11.56. Свойства `MaskedTextBox`

- `MaskFull` — позволяет проверить, все ли требуемые символы маски введены (например, вы вводили дату и не заметили, что вместо восьми символов ввели только семь — при большой скорости ввода это всегда может произойти). Поэтому данный момент надо программно контролировать. Пример задания такого контроля показан на рис. 11.57, текст обработчика — в листинге 11.9.

Листинг 11.9

```
private: System::Void maskedTextBox1_KeyDown(System::Object^  sender,
System::Windows::Forms::KeyEventArgs^  e)
 {
  if(e->KeyCode == Keys::Enter)
   {
    if(this->maskedTextBox1->MaskFull == false)
     {
       MessageBox::Show("Дата введена не полностью", "Приложение
        56", MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
```

```
        this->textBox1->Clear();
        this->maskedTextBox1->Clear();

        return;
       }
    this->textBox1->Text=this->maskedTextBox1->Text;
   }
}
```

Рис. 11.57. Контроль на полноту ввода по маске

- `Text` — возвращает строку, введенную по маске.
- `TextMaskFormat` — определяет, как символы-литералы и подсказка взаимодействуют, когда генерируется форматная строка (т. е. строка, введенная по маске). Точнее, это свойство задает, будут ли они (оба вместе или по отдельности) включены в итоговую строку (в свойство `Text`). Если исключаются символы-подсказки, то они заменяются пробелами.
- `AsciiOnly` — используется для ограничения ввода (могут вводиться только символы a—z, A—Z и 0—9), хотя среда программирования поддерживает все символы Unicode. Их, как известно, намного больше, чем символов ASCII (последние кодируются на основе кода длиной в 8 битов, а первые — на основе 16 битов, т. е. ясно, что $2^{16} > 2^{8}$).

Компонент *CheckedListBox*

Компонент находится в списке **All Windows Forms** палитры компонентов. Компонент `CheckedListBox` является расширением `ListBox`. Он делает почти все, что дела-

ет `ListBox`, но дополнительно выводит окна контроля (флажки-переключатели), в которых можно делать отметку галочкой. Вид компонента показан на рис. 11.58.

Пользователь может помечать элементы списка, щелкая мышью на одной или нескольких позициях (устанавливать флажок). Повторный щелчок снимает включение флажка. Выбор элемента списка еще не означает, что флажок устанавливается/снимается.

Рис. 11.58. Вид компонента `CheckedListBox`, помещенного в форму

Существует свойство `CheckOnClick`, которое разрешает/запрещает делать пометку (устанавливать/снимать флажок). Если это свойство установлено в `false`, то при щелчке мышью на позиции галочка (флажок) в ней не появится. Но одновременно со щелчком на позиции идет подсветка строки. Если повторно щелкнуть на отмеченной строке, то галочка появится. Таким образом, при значении свойства `CheckOnClick`, установленным в `false`, для включения флажка надо сначала отметить элемент списка (щелчком на строке), потом сделать повторный щелчок.

А выключается флажок при щелчке на строке или на нем самом.

Если же `CheckOnClick` установлено в `true`, то флажок включается одновременно с выбором элемента (и выключается при повторном щелчке на нем).

Существует свойство `ThreeDCheckBoxes`, которое определяет стиль окна флажка (будет ли оно в стиле `Flat` или `Normal`). Если значение свойства равно `true`, то стиль `Flat`, иначе — `Normal`.

Перечень свойств компонента, отображенных в его окне **Properties**, показан на рис. 11.59.

Рис. 11.59. Свойства компонента `CheckedListBox`

Компонент `CheckedListBox` поддерживает три состояния флажка:

- `Checked` — флажок включен;
- `Unchecked` — флажок выключен;
- `Indeterminate` — состояние неопределенности (флажок закрашен в серый цвет). Такое состояние можно устанавливать только в режиме исполнения, т. е. программно (т. к. этого механизма выполнения в режиме дизайна нет).

Существуют методы `CheckedListBox`, с помощью которых можно определять и устанавливать состояние флажка:

- `GetItemCheckState (int index)`;
- `SetItemCheckState (int index, CheckState value)`.

Здесь:

- `index` — это индекс того элемента, состояние флажка которого определяется;
- `CheckState value` — значение состояния флажка. Состояние определено классом `CheckState` и имеет три значения (`Checked`, `Unchecked`, `Indeterminate` — им соответствуют числовые значения 1, 0 и 2).

Приведем пример задания состояния флажков и определения их состояния. Текст обработчика события `DoubleClick` компонента `CheckedListBox` приведен в листинге 11.10.

Листинг 11.10

```
private: System::Void checkedListBox1_DoubleClick(System::Object^  sender,
System::EventArgs^  e)
 {
  this->checkedListBox1->SetItemCheckState(0,
   CheckState::Checked);
 this->checkedListBox1->SetItemCheckState(1,
   CheckState::Unchecked);
 this->checkedListBox1->SetItemCheckState(2,
   CheckState::Indeterminate);
int i=(int)this->checkedListBox1->GetItemCheckState(0);
 }
```

Здесь задаются состояния для флажков 1, 2 и 3-й строк. Последний оператор определяет состояние 1-й строки. Так как тип результата, выдаваемого методом `GetItemCheckState()`, — это тип `CheckState`, то, чтобы увидеть состояние, надо этот тип привести к типу `int` принудительно. Поэтому после знака присвоения стоит `(int)`.

Чтобы узнать, помечен ли элемент списка, можно выполнить оператор:

```
bool  b=this->checkedListBox1->GetItemChecked(i);
```

Результат метода `GetItemChecked()` — логическая переменная, `i` — индекс элемента.

Можно и другим способом установить флажок. Для этого следует выполнить оператор:

```
this->checkedListBox1->SetItemChecked(3, 1);
```

Здесь:

- `3` — индекс элемента (4-я строка);
- `1` — булево значение `true` (второй параметр этого метода имеет булевый тип).

Обратим внимание на событие `ItemCheck`, которое возникает, когда состояние флажка меняется. Оно пригодится при обработке выборки из `CheckedListBox`.

В `CheckedListBox` можно также загружать текстовые строки из файла, как и для `ListBox`. Программа будет такая же, только в ней надо заменить `ListBox` на `CheckedListBox`.

Пример: домашний телефонный справочник

Приведем пример приложения для домашнего телефонного справочника, построенного с использованием компонента `CheckedListBox`. Справочник должен содержать номера телефонов и специальный ящик (горячий ящик), в котором будут храниться самые необходимые из всего списка телефоны (чтобы ими можно было бы-

стро воспользоваться). В ящик будут посылаться только те телефоны, которые в списке будут помечаться галочками (т. е. их флажки будут включены). Как только флажок выключается, номер телефона из горячего ящика удаляется. Однако строку из горячего ящика можно удалить и по щелчку мыши на ней.

Итак, приложение работает следующим образом: как только оно загружается для выполнения (фактически загружается форма), автоматически читаются два текстовых файла, в которых находятся сведения о предыдущей работе приложения (перед тем, как форме появиться на экране, срабатывает событие `Shown`, в обработчик которого помещены операторы загрузки файлов). Этими файлами заполняется содержимое двух контейнеров: компонента `ChekedListBox` и компонента `ComboBox`. Далее приложение работает в обычном режиме: строки можно добавлять в оба контейнера и удалять из каждого в отдельности. Как только приложение завершит свою работу (кнопка **Выход**), содержимое контейнеров сохранится в соответствующих файлах (чтобы оно при выгрузке не исчезало).

Отсюда возникает проблема первоначального запуска приложения: надо вручную сформировать указанные два файла (их имена указаны в тексте приложения, которое, надеюсь, читатель сам создаст у себя заново и выберет те имена файлов, которые ему подойдут). Надо взять обыкновенный WordPad (но не Блокнот: будут проблемы с кодировкой) и с его помощью создать пустые текстовые файлы... А далее в приложении вы сами введете те тексты, которые вам нужны.

Форма приложения в режиме дизайна приведена на рис. 11.60, а текст — в листинге 11.11 (приведен полностью h-файл, чтобы можно было видеть всю структуру приложения и места вставки пользовательских функций работы с файлами).

Результаты работы приложения показаны на рис. 11.61—11.63.

Рис. 11.60. Форма приложения "Домашний телефонный справочник"

Рис. 11.61. Работа с телефонным справочником. Часть 1

Рис. 11.62. Работа с телефонным справочником. Часть 2

Рис. 11.63. Работа с телефонным справочником. Часть 3

Листинг 11.11

```
#pragma once

namespace Tel_Sprav2011 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::IO;
    //using namespace System::IO::File;
    using namespace System::Text;
    //using namespace std;

    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
```

```
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
    private: System::Windows::Forms::Label^  label1;
    protected:
    private: System::Windows::Forms::ComboBox^  comboBox1;
    private: System::Windows::Forms::Label^  label2;
    private: System::Windows::Forms::Label^  label3;
    private: System::Windows::Forms::Button^  button1;
    private: System::Windows::Forms::CheckedListBox^  checkedListBox1;
    private: System::Windows::Forms::TextBox^  textBox1;
    private: System::Windows::Forms::TextBox^  textBox2;
    private: System::Windows::Forms::Button^  button2;
    private: System::Windows::Forms::Button^  button3;
    private: System::Windows::Forms::Label^  label4;
    private: System::Windows::Forms::Button^  button4;

    private:
        /// <summary>
        /// Required designer variable.

        int fix;
        StreamWriter ^sw;

        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->label1 = (gcnew System::Windows::Forms::Label());
```

```
            this->comboBox1 = (gcnew System::Windows::Forms::ComboBox());
            this->label2 = (gcnew System::Windows::Forms::Label());
            this->label3 = (gcnew System::Windows::Forms::Label());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->checkedListBox1 = (gcnew System::Windows::
                                                Forms::CheckedListBox());
            this->textBox1 = (gcnew System::Windows::Forms::TextBox());
            this->textBox2 = (gcnew System::Windows::Forms::TextBox());
            this->button2 = (gcnew System::Windows::Forms::Button());
            this->button3 = (gcnew System::Windows::Forms::Button());
            this->label4 = (gcnew System::Windows::Forms::Label());
            this->button4 = (gcnew System::Windows::Forms::Button());
            this->SuspendLayout();
            //
            // label1
            //
            this->label1->AutoSize = true;
            this->label1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans
                               Serif", 7.8F, System::Drawing::FontStyle::Bold,
                               System::Drawing::GraphicsUnit::Point,
              static_cast<System::Byte>(204)));
            this->label1->Location = System::Drawing::Point(15, 13);
            this->label1->Name = L"label1";
            this->label1->Size = System::Drawing::Size(378, 17);
            this->label1->TabIndex = 0;
            this->label1->Text = L"Десять самых нужных телефонов (горячий ящик)";
            //
            // comboBox1
            //
            this->comboBox1->FormattingEnabled = true;
            this->comboBox1->Location = System::Drawing::Point(44, 33);
            this->comboBox1->Name = L"comboBox1";
            this->comboBox1->Size = System::Drawing::Size(289, 24);
            this->comboBox1->TabIndex = 1;
            this->comboBox1->Click += gcnew System::EventHandler(this,
&Form1::comboBox1_Click);
            //
            // label2
            //
            this->label2->AutoSize = true;
            this->label2->Location = System::Drawing::Point(41, 72);
            this->label2->Name = L"label2";
            this->label2->Size = System::Drawing::Size(118, 17);
            this->label2->TabIndex = 2;
            this->label2->Text = L"Номер абонента";
            //
            // label3
            //
```

```
this->label3->AutoSize = true;
this->label3->Location = System::Drawing::Point(231, 72);
this->label3->Name = L"label3";
this->label3->Size = System::Drawing::Size(98, 17);
this->label3->TabIndex = 3;
this->label3->Text = L"Комментарий";
//
// button1
//
this->button1->Location = System::Drawing::Point(399, -1);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(116, 101);
this->button1->TabIndex = 4;
this->button1->Text = L"Выход";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this,
                                        &Form1::button1_Click);
//
// checkedListBox1
//
this->checkedListBox1->FormattingEnabled = true;
this->checkedListBox1->Location = System::Drawing::Point(18, 106);
this->checkedListBox1->Name = L"checkedListBox1";
this->checkedListBox1->Size = System::Drawing::Size(497, 89);
this->checkedListBox1->TabIndex = 5;
this->checkedListBox1->ItemCheck += gcnew System::Windows::Forms::
ItemCheckEventHandler(this, &Form1::checkedListBox1_ItemCheck);
//
// textBox1
//
this->textBox1->Location = System::Drawing::Point(18, 216);
this->textBox1->Name = L"textBox1";
this->textBox1->Size = System::Drawing::Size(100, 22);
this->textBox1->TabIndex = 6;
this->textBox1->KeyDown += gcnew System::Windows::Forms::
KeyEventHandler(this, &Form1::textBox1_KeyDown);
//
// textBox2
//
this->textBox2->Location = System::Drawing::Point(414, 215);
this->textBox2->Name = L"textBox2";
this->textBox2->Size = System::Drawing::Size(100, 22);
this->textBox2->TabIndex = 7;
this->textBox2->KeyDown += gcnew System::Windows::Forms::
                  KeyEventHandler(this, &Form1::textBox2_KeyDown);
//
// button2
//
```

```
            this->button2->Location = System::Drawing::Point(18, 255);
            this->button2->Name = L"button2";
            this->button2->Size = System::Drawing::Size(100, 48);
            this->button2->TabIndex = 8;
            this->button2->Text = L"Добавить в список";
            this->button2->UseVisualStyleBackColor = true;
            this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
            //
            // button3
            //
            this->button3->Location = System::Drawing::Point(423, 255);
            this->button3->Name = L"button3";
            this->button3->Size = System::Drawing::Size(91, 48);
            this->button3->TabIndex = 9;
            this->button3->Text = L"Удалить из списка";
            this->button3->UseVisualStyleBackColor = true;
            this->button3->Click += gcnew System::EventHandler(this,
&Form1::button3_Click);
            //
            // label4
            //
            this->label4->AutoSize = true;
            this->label4->Location = System::Drawing::Point(-147, 241);
            this->label4->Name = L"label4";
            this->label4->Size = System::Drawing::Size(16, 17);
            this->label4->TabIndex = 10;
            this->label4->Text = L">";
            //
            // button4
            //
            this->button4->Location = System::Drawing::Point(157, 216);
            this->button4->Name = L"button4";
            this->button4->Size = System::Drawing::Size(227, 112);
            this->button4->TabIndex = 11;
            this->button4->Text = L"Перед нажатием на кнопку \"Добавить\"
введите номер телефона и комментарий в левое"
              L" и правое поля ввода, заканчивая каждый раз ввод нажатием
<Enter>";
            this->button4->UseVisualStyleBackColor = true;
            //
            // Form1
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(522, 331);
            this->Controls->Add(this->button4);
```

```
            this->Controls->Add(this->label4);
            this->Controls->Add(this->button3);
            this->Controls->Add(this->button2);
            this->Controls->Add(this->textBox2);
            this->Controls->Add(this->textBox1);
            this->Controls->Add(this->checkedListBox1);
            this->Controls->Add(this->button1);
            this->Controls->Add(this->label3);
            this->Controls->Add(this->label2);
            this->Controls->Add(this->comboBox1);
            this->Controls->Add(this->label1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->Shown += gcnew System::EventHandler(this, &Form1::Form1_Shown);
            this->ResumeLayout(false);
            this->PerformLayout();

        }
#pragma endregion
private: System::Void comboBox1_Click(System::Object^  sender,
System::EventArgs^  e)
 {
  //Удаление строки из ComboBox по щелчку на ней
  int i=this->comboBox1->SelectedIndex;
  if(i== -1) return;
  this->comboBox1->Items->Remove(this->comboBox1->SelectedItem);
 }
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{
    /*Когда приложение завершается,
  надо сохранить данные ChekedListBox и ComboBox  в файлах*/

    //Сохранение checkedListBox

    // Create a file to write to

    String^ path = "d:\\for_tel_sprav_chb.txt";
    sw = File::CreateText( path ); //StreamWriter ^

    String ^s;
    int y=this->checkedListBox1->Items->Count;
    for ( this->checkedListBox1->SelectedIndex = 0; this->checkedListBox1->
SelectedIndex < y-1; this->checkedListBox1->SelectedIndex++ )
      {
       s=this->checkedListBox1->Items[this->checkedListBox1->SelectedIndex]->
ToString();
       sw->WriteLine(s);
      }
```

```
      s=this->checkedListBox1->Items[this->checkedListBox1->SelectedIndex]->
                                                                    ToString();
      sw->WriteLine(s);
      sw->Close();

      //Сохранение ComboBox
    path = "d:\\for_tel_sprav_cb.txt";
    sw = File::CreateText( path ); //StreamWriter ^

     y=this->comboBox1->Items->Count;
     if(y==0)
     {
       sw->Close();
       this->Close();
       return;
     }
    for ( this->comboBox1->SelectedIndex = 0; this->comboBox1->
                        SelectedIndex < y-1; this->comboBox1->SelectedIndex++ )
      {
       s=this->comboBox1->Items[this->comboBox1->SelectedIndex]->ToString();
       sw->WriteLine(s);
      }
      s=this->comboBox1->Items[this->comboBox1->SelectedIndex]->ToString();
      sw->WriteLine(s);
      sw->Close();

      this->Close();
}
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
     //Добавить в список
     String^ r = "
";

/*Формирование в строке r данных, введенных в поля ввода
  для телефона и комментария*/
  r=r->Concat(this->textBox1->Text, r);
  r=r->Insert(35, this->textBox2->Text);
  this->checkedListBox1->Items->Add(r,0);//состояние = 0  отключен
  this->textBox1->Text="";
  this->textBox2->Text="";
  this->textBox1->Focus();
 }
private: System::Void textBox1_KeyDown(System::Object^  sender,
System::Windows::Forms::KeyEventArgs^  e)
  {
    if(e->KeyCode == Keys::Enter)
```

```
    {
     this->textBox2->Focus();
    }
   }
private: System::Void textBox2_KeyDown(System::Object^  sender,
System::Windows::Forms::KeyEventArgs^  e)
   {
     if(e->KeyCode == Keys::Enter)
      {
       this->button2->Focus();
      }
     }
private: System::Void Form1_Shown(System::Object^  sender, System::EventArgs^ e)
 {
 //Загрузка компонентов из сохраненных от пред. сеанса файлов
 //Ввод первоначальных данных в ComboBox и CheckedListBox
 //Файл из одной строки, т. к. для ввода применен while()
 //Файл должен быть записан WordPad'ом как текстовый в кодировке Юникод

          //Очистка компонентов
          this->comboBox1->Items->Clear();
          this->checkedListBox1->Items->Clear();

          //Запись в checkedListBox

    String^ path = "d:\\for_tel_sprav_chb.txt";
    if (  !File::Exists( path ) )
     {
       // Create a file to write to
       sw = File::CreateText( path ); // StreamWriter^
       try
       {
          sw->WriteLine( "Hello" ); //Это данные для контроля ввода.
          sw->WriteLine( "And" );    //Если читаемый файл не найден,
          sw->WriteLine( "Welcome" );//эти данные выведутся
       }
       finally
       {
          if ( sw )
           delete (IDisposable^)(sw);
       }
    }

    // Open the file to read from
    TextReader ^ sr = File::OpenText( path );
    try
```

```
    {
      String^ s = "";
      while ( s = sr->ReadLine() )
       {
         this->checkedListBox1->Items->Add(s);
       }
   }
   finally
   {
     if ( sr )
      delete (IDisposable^)(sr);
   }

   //Чтение файла для ComboBox
   path = "d:\\for_tel_sprav_cb.txt";
   if (  !File::Exists( path ) )
    {
      // Create a file to write to
      sw = File::CreateText( path ); // StreamWriter^
      try
      {
         sw->WriteLine( "Hello" ); //Это данные для контроля ввода.
         sw->WriteLine( "And" );    //Если читаемый файл не найден,
         sw->WriteLine( "Welcome" );//эти данные выведутся
      }
      finally
      {
         if ( sw )
          delete (IDisposable^)(sw);
      }
   }

   // Open the file to read from
   sr = File::OpenText( path );
   try
    {
      String^ s = "";
      while ( s = sr->ReadLine() )
       {
         this->comboBox1->Items->Add(s);
       }
   }
   finally
   {
     if ( sr )
      delete (IDisposable^)(sr);
   }
}
```

```
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^ e)
 {
   // Обработка кнопки "Удалить из списка"
  if(this->checkedListBox1->SelectedIndex == -1)

 //строку не отметили для удаления
   {
    MessageBox::Show("Отметьте строку для удаления", "Приложение Пользователя",
MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
    return;
    }
   this->checkedListBox1->Items->Remove(this->checkedListBox1->SelectedItem);
 }
private: System::Void checkedListBox1_ItemCheck(System::Object^  sender,
System::Windows::Forms::ItemCheckEventArgs^  e)
 {
     /*Обработка выборки из списка.
       В зависимости от свойства CheckOnClick состояние флажка меняется
       либо от одного щелчка, либо от повторного.
       Здесь установлено, что от одного щелчка.
       Сюда попадаем, когда щелчком мыши выбираем строку из списка */

 String ^str,^tel,^str1;
 int i=this->checkedListBox1->SelectedIndex;

 /*Здесь будет индекс выбранной строки после клика (щелчка) на ней*/

 str=dynamic_cast <String ^>(this->checkedListBox1->SelectedItem);//перевод из
типа Object ^ в String ^

 //Здесь будет выбранная строка после клика на ней

  tel=str->Substring(0,str->Length); //выделили номер телефона

    //Добавка или удаление номера телефона в (из) ComboBox
    /*Поиск строки в ComboBox: если она найдена,
      то удаляется, если не найдена, то после этого блока она
      добавляется*/

  int k=0,j=this->comboBox1->Items->Count;
  for(int i=0; i < j; i++)
   {
    str1=dynamic_cast <String ^>(this->comboBox1->Items[i]);
    if(System::String::Compare(str1,tel) != 0)
     //Строки не сравнились
      continue;
```

```
      else
       {
        Object ^str2=dynamic_cast <Object ^> (str1);

        /*Метод Remove() требует типа Object ^,
         поэтому мы перевели тип String ^ в  тип Object ^ */

         this->comboBox1->Items->Remove(str2);
         k=1;
        break;
       }
      } //for()
      if(k == 1)  //Строку удалили
       return;

      /*Здесь ситуация, когда строки в ящике нет, поэтому ее надо
        в него добавить */

      if(this->comboBox1->Items->Count > 10)
       return;

      /*Если в ящике уже 10 строк, то вставлять не надо (так мы
       договорились, что ящик будет содержать не более 10-ти строк) */

       this->comboBox1->Items->Add(tel);
  }
};
}
```

Чтобы добавить новую строку в список, надо активизировать щелчком мыши поле ввода номера телефона (оно расположено над кнопкой **Добавить в список**). Затем нужно ввести в это поле текст и нажать клавишу <Enter>.

Фокус ввода перейдет к полю над кнопкой **Удалить из списка** (туда вводится второй текст, например комментарий к первому введенному тексту, и снова нажимается клавиша <Enter>). Фокус ввода перейдет к кнопке **Добавить в список**, после чего надо снова нажать клавишу <Enter>. Обе введенные строки перенесутся в поле `CheckedListBox`.

Чтобы удалить строку из `CheckedListBox`, ее надо пометить, щелкнув на ней мышью, а затем нажать кнопку **Удалить из списка**.

Чтобы удалить строку из `ComboBox`, надо открыть выпадающий список и щелкнуть на строке, которую требуется удалить.

Компоненты *CheckBox* и *RadioButton*

Компоненты расположены в группе **Common Controls** палитры компонентов. Оба используются как флажки-переключатели для предоставления возможного выбора после их работы — они выдают результат "включен/выключен".

Если компонент `CheckBox` находится в группе себе подобных, то, включив один `CheckBox`, можно включать и остальные и при этом ни один из них не выключится (т. е. не изменит своего состояния).

Компонент `RadioButton` ведет себя по-другому: когда он находится совместно с такими же компонентами в одной группе, то не допускает, чтобы был еще включен какой-то другой `RadioButton` — он тут же выключается. То есть в группе компонентов `RadioButton` может быть включен только один из них, а остальные будут автоматически выключены.

Оба этих компонента соответствуют математическим понятиям конъюнкции и дизъюнкции. Когда мы говорим "находится в группе", то имеется в виду, что у множества таких компонентов один родитель (например, одна панель). Если одно множество `RadioButton` находится на одной панели, а другое — на своей собственной, то включение такой кнопки на панели *A* не повлияет на состояние такой же кнопки на панели *B*, поскольку у них разные родители. Поэтому если необходимо, чтобы кнопки `RadioButton` обрабатывали несколько непересекающихся ситуаций, то такие кнопки нужно разместить на разных панелях или в других групповых контейнерах.

Перечень свойств `CheckBox`, отображенных в окне **Properties**, представлен на рис. 11.64, а свойств `RadioButton` — на рис. 11.65.

Рис. 11.64. Свойства `CheckBox`

Рис. 11.65. Свойства `RadioButton`

Рассмотрим некоторые свойства этих компонентов.

- `Appearance` — определяет форму появления компонента (в виде обычного флажка или в виде кнопки).
- `Checked` — по этому свойству в режиме исполнения приложения можно определить, включен или выключен флажок.
- `CheckState` — устанавливает трехвидовое состояние:
  - `Checked` — флажок включен;
  - `Unchecked` — флажок выключен;
  - `Indeterminate` — состояние неопределенности (флажок закрашен в серый цвет).
- `ThreeState` — задает поддержку двух (`ThreeState = false`) или трех (`ThreeState = true`) состояний. У `RadioButton` этого свойства, естественно, нет, поскольку этот компонент имеет по определению всего два состояния.

  Если значение `ThreeState` установлено в `true`, то свойство `Checked` всегда возвращает `true` для любого из состояний: `Checked` или `Indeterminate` (т. е. и при значении `CheckState = Checked`, и при значении `CheckState = Indeterminate` свойство `Checked` всегда возвращает `true`).
- `FlatStyle` — определяет стиль появления компонента (см. `CheckedListBox`).
- `CheckAlign` — свойство, позволяющее открыть выпадающий список, где можно выбрать схему размещения флажка в поле компонента (его можно разместить в девяти местах окна компонента, но при этом `Appearance` должно быть не кнопкой).

Рис. 11.66. Примеры работы переключателей

Примеры работы переключателей показаны на рис. 11.66, текст обработчиков событий — в листингах 11.12 и 11.13.

Листинг 11.12

```
private: System::Void checkBox1_CheckStateChanged(System::Object^  sender,
System::EventArgs^  e)
 {
  if(this->checkBox1->CheckState == CheckState::Checked)
     this->panel1->BackColor=Color::Aqua ;
  if(this->checkBox1->CheckState == CheckState::Unchecked)
     this->panel1->BackColor=Color::Blue;
  }
private: System::Void checkBox2_CheckStateChanged(System::Object^  sender,
System::EventArgs^  e)
 {
  if(this->checkBox2->CheckState == CheckState::Checked)
     this->panel2->BackColor=Color::Aquamarine ;
  if(this->checkBox2->CheckState == CheckState::Unchecked)
     this->panel2->BackColor=Color::Azure;
  }
private: System::Void radioButton1_CheckedChanged(System::Object^  sender,
System::EventArgs^  e)
 {
  this->pictureBox2->Visible=false;
  this->pictureBox1->Visible=true;
 }
private: System::Void radioButton2_CheckedChanged(System::Object^  sender,
System::EventArgs^  e)
 {
  this->pictureBox2->Visible=true;
  this->pictureBox1->Visible=false;
 }
```

Листинг 11.13

```
private: System::Void checkBox1_CheckedChanged(System::Object^  sender,
System::EventArgs^  e)
 {
  if(this->checkBox1->CheckState == CheckState::Checked)
     this->panel1->BackColor=Color::Red ;
  else
     this->panel1->BackColor=Color::Blue;

 }
private: System::Void radioButton1_CheckedChanged(System::Object^  sender,
System::EventArgs^  e)
```

```
  {
   this->pictureBox2->Visible=false;
   this->pictureBox1->Visible=true;
  }
private: System::Void checkBox2_CheckStateChanged_1(System::Object^  sender,
System::EventArgs^  e)
{
  if(this->checkBox2->CheckState == CheckState::Checked)
     this->panel2->BackColor=Color::Aquamarine ;
  else
     this->panel2->BackColor=Color::Beige;
}
private: System::Void radioButton2_CheckedChanged_1(System::Object^  sender,
System::EventArgs^  e)
{
  this->pictureBox2->Visible=true;
  this->pictureBox1->Visible=false;
}
```

Компонент *GroupBox*

Довольно часто вместо отдельных переключателей используют их групповой контейнер — компонент `GroupBox`, который расположен в группе **All Windows Forms** палитры компонентов.

Рис. 11.67. Свойства `GroupBox`

Вообще GroupBox-компоненты используют, чтобы обеспечить разделение компонентов на различные группы, которые становятся для них родителями, с тем, чтобы компоненты унаследовали некоторые свойства своих родителей. Обычно так делают, чтобы подразделить форму на несколько функций. Да и дизайн осуществлять удобнее: перемещая только GroupBox, мы одновременно перемещаем все компоненты, которые в нем находятся.

Перечень свойств компонента, отображенный в его окне **Properties**, показан на рис. 11.67 (никаких новых свойств по сравнению с ранее встречавшимися, мы тут не находим). Применение компонента вместе с флажками было показано на рис. 11.66, в котором в этот контейнер помещались компоненты RadioButton и CheckBox.

Компонент *LinkLabel*

Компонент расположен в группе **All Windows Forms** палитры компонентов, он позволяет добавлять к приложениям ссылки на Web-страницы, задавать адреса папок, файлов (на первом уровне), которые будут находиться и открываться. Наряду с этим главным качеством компонента, его можно использовать аналогично компоненту Label.

Как же работать с этим компонентом?

Самое простое — щелкнуть на нем дважды, в результате чего откроется заготовка обработчика. Параметром этого обработчика является ссылка на класс LinkLabelLinkClickedEventArgs, который содержит данные, необходимые для обработки строки типа String, содержащей гиперссылку — обычный адрес, по которому надо добраться до необходимых данных (Web-страницы, обычной папки вашего компьютера или просто файла). Ссылка будет обработана, и на ваших глазах появится открытый документ (сайт, содержимое папки или содержимое файла). Но сами по себе они не откроются — надо выполнить метод Start() из класса Process (класса, обеспечивающего запуск различных процессов внутри приложения).

Вид метода запуска выглядит так:

```
System::Diagnostics::Process::Start( str );
```

где str — это переменная типа String ^.

Итак, самый простой способ использования гиперссылки таков:

в обработчике события LinkClicked объявляем:

```
String ^str="адрес объекта";
```

затем выполняем оператор:

```
System::Diagnostics::Process::Start(str );
```

Примеры адреса объекта:

- C:\\a.txt — это файл, если этот путь задается в тексте программы как константа, а если, например, в поле компонента TextBox, то двойной BackSlash надо заменить на одинарный);

- D:\\Фото — это папка;
- **www.mail.ru** — это интернет-адрес.

Форма приложения, использующего `LinkLabel`, и результаты работы — на рис. 11.68, а тексты обработчиков событий — в листинге 11.14.

Рис. 11.68. Демонстрация работы `LinkLabel`

Листинг 11.14

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
    {
        this->Close();
    }
    private: System::Void linkLabel1_LinkClicked(System::Object^  sender,
System::Windows::Forms::LinkLabelLinkClickedEventArgs^  e)
    {
      this->linkLabel1->Links[ linkLabel1->Links->IndexOf( e->Link ) ]->
                                                          Visited = true;
      String ^str =this->textBox1->Text;
      System::Diagnostics::Process::Start( str );
  // класс, обеспечивающий запуск процессов внутри приложения

     }
private: System::Void Form1_Shown(System::Object^  sender, System::EventArgs^ e)
{
    this->textBox1->Focus();
}
```

Рассмотрим более подробно сам процесс создания гиперссылки. Во-первых, у нее есть наименование, которое помещается в свойство `Text`, и собственно сама ссылка, т. е. путь к объекту, на который идет ссылка.

Куда помещается сама ссылка? Она помещается в свойство `Links`, которое не отражается в окне **Properties**. Из самого названия свойства (связи, ссылки) следует, что ссылок можно задавать и больше, чем на один объект. Это свойство является указателем на элемент `LinkCollection` класса `LinkLabel`, который содержит множество `M{}` ссылок, задающихся в `LinkLabel`, поэтому можно воспользоваться методами этого класса, в частности — методом `Add()`, который добавляет в `M{}` новую гиперссылку (это можно сделать только в тексте программы, т. к. свойство `Links` не высвечивается в окне **Properties**). Поскольку ссылок бывает одна и более, то первая ссылка обязательно помещается в свойство `LinkArea` (нажав на кнопку с многоточием в поле этого свойства, можно такую ссылку задать прямо в режиме дизайна проекта). Остальные ссылки можно задать в тексте программы, помещая их в свойство `Links` методом `Add()`. В любом случае, зададите ли вы одну-единственную ссылку или множество, все они в режиме исполнения приложения попадут во множество M{}, т. е. в свойство `Links`. Следовательно, если вы задаете в тексте программы даже единственную ссылку, то можно поместить ее сразу в первый элемент из множества M{}, однако сведения о том, что это единственная ссылка, надо все же отразить в свойстве `LinkArea`.

Каким образом?

Дело в том, что сама ссылка непосредственно связана со своим наименованием, т. е. с текстом этого наименования, который хранится в свойстве `Text`. Это свойство содержит строку символов (точнее — слов). Слова следует пометить так, чтобы было понятно, что они отображают названия гиперссылок. Если слово отображает наименование гиперссылки, оно сразу оказывается непохожим на остальные части текста в строке свойства `Text`: такое слово становится подчеркнутым, цвет его шрифта изменяется в соответствии с цветом, установленным в свойстве `LinkColor` (цвет выбирается из всплывающей палитры цветов, если в поле свойства нажать кнопку со стрелкой). Например, свойство `Text` содержит текст "Это гиперссылка, а это просто текст". Если вы пометили часть текста "Это гиперссылка" таким образом, чтобы она относилась к наименованию гиперссылки, то когда в режиме исполнения приложения вы на такую помеченную часть текста наведете курсор мыши, его вид изменится: он станет изображаться в виде кисти руки. Это означает, что в этом месте можно щелкнуть кнопкой мыши и программа отошлет нас к объекту, адрес которого находится в свойстве `Links` в соответствующем его элементе, имеющим тот же порядковый номер от начала множества, что и помеченная ссылкой часть текста в свойстве `Text`. Сколько частей в свойстве `Text` мы пометим в качестве названий гиперссылок, столько фактических адресов объектов, на которые будет идти ссылка из помеченных частей текста в свойстве `Text`, надо будет задать в свойстве `Links`. Ровно столько же, и ни больше ни меньше, иначе возникнет исключительная ситуация, которая, если ее не обработать, приведет к аварийному останову программы.

Как же помечать части текстовой строки, чтобы они становились наименованиями гиперссылок? Пометки задаются в виде указания номера позиции (счет от нуля) символа, которым начинается наименование гиперссылки, и количества символов, относящихся к наименованию гиперссылки.

В самом свойстве `LinkArea` (а можно в его подствойствах — `Start` и `Length`) надо набрать соответственно номер начальной позиции (счет от нуля) и длину подстроки, набранной в редакторе `LinkArea` (то, что набирается в редакторе `LinkArea`, автоматически попадает в свойство `Text`), которая будет играть роль имени первой и, возможно, единственной, гиперссылки. Если вы набираете координаты имени гиперссылки в самом поле `LinkArea`, то начало и длину имени следует набирать через точку с запятой.

Например, в поле редактора вы набрали текст "Это гиперссылка, а это просто текст". В свойстве `LinkArea` автоматически установятся значения (35;0). Получатся данные: `Start=35` и `Length=0`. Это неправильно. И свидетельством тому явится исчезновение подсветки шрифта в наименовании, которое мы ввели. Вы должны за этим проследить и самостоятельно установить значения подсвойств `Start` и `Length`.

Если, например, зададите величины (0;23), то получите подсвеченный и подчеркнутый текст такой: "Это гиперссылка, а это просто текст" (остальные символы заданного текста не станут отражать название гиперссылки).

Если же зададите (0;15), то получите Это гиперссылка.

Для выделения остальных частей строки текста из свойства `Text` вы должны задавать начало и длину в программе и помещать их (как и сами адреса объектов) в элементы свойства `Links`. Если, например, вы станете добавлять данные из объекта `TextBox1`, то оператор добавки будет выглядеть так:

```
this->linkLabel1->Links->Add( (int)ArStart[i], (int)ArLen[i], this->textBox1-
>Lines[i]);
```

Здесь аргументы `(int)ArStart[i]`, `(int)ArLen[i]` означают `i`-е элементы массивов, где хранятся соответственно номера символов начальных позиций и длин текста из свойства `Text`, которые будут помечены в качестве наименования гиперссылки, добавляемой в качестве пути к объекту из `i`-й строки `TextBox1`. Начало и длина принудительно переводятся в тип `int`, как требует того формат метода `Add()`, потому что массивы `ArStart[]`, `ArLen[]` — это числовые (`Int32`) managed-массивы.

Когда у нас в проекте несколько гиперссылок, то требуется отмечать, что мы щелкали на какой-то конкретной гиперссылке (как говорят, "посетили" гиперссылку), иначе в работе может наступить полная путаница.

В классе `Links` имеется на этот счет специальное свойство `Visited`, устанавливаемое в значения `true` (посетили ссылку) и `false` (не посещали ссылку). Кроме того, к ссылке можно разрешить или запретить доступ с помощью установки в значения `true` и `false` другого свойства класса `Links` — свойства `Enable`.

В качестве примера приведем приложение, где задается строка в компоненте `TextBox`, из которой формируются два имени гиперссылок и адреса к объектам, за-

данные в другом компоненте `TextBox`. Текст приложения приведен в листинге 11.15, а результат его работы — на рис. 11.69 и 11.70.

Листинг 11.15

```
#pragma once
namespace My2008LinkLabel {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System;
    using namespace System::Runtime::InteropServices;


/// <summary>
/// Summary for Form1
///
/// WARNING: If you change the name of this class, you will need
/// to change the 'Resource File Name' property for the managed
/// resource compiler tool associated with all .resx files this
/// class depends on.  Otherwise, the designers will not be able
/// to interact properly with localized resources
/// associated with this form.
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
public:
Form1(void)
{
InitializeComponent();
//
//TODO: Add the constructor code here
//
}

protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
{
if (components)
{
```

```
delete components;
}
}

protected:


private: System::Windows::Forms::Button^  button1;
private: System::Windows::Forms::LinkLabel^  linkLabel1;
private: System::Windows::Forms::Button^  button2;
private: System::Windows::Forms::TextBox^  textBox1;
private: System::Windows::Forms::TextBox^  textBox2;
private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::Label^  label2;

private:
/// <summary>
/// Required designer variable.

//----------------- PasteTextString() ----------------------------------------
/*просматривает строку, удаляет из нее все "/", "*", одновременно
запоминая координаты слов в новой строке в массивах ArStart[],ArLen[]*/

//managed-массивы будут и выходными
void PasteTextString(String ^in, array <String ^> ^out,  array <int> ^ArStart,
array <int> ^ArLen, array <int> ^Ns)
  {
     in->ToCharArray(); //это будет уже массивом символов
         String ^p;
 int Ar=0; //индекс для ArStart и ArLen

//посимвольный просмотр строки
 String ^out1="";
 int len=in->Length;
  for(int i=0,j=0; i < len; i++)
  {
  wchar_t s=in->ToCharArray()[i];
             p= s.ToString();

  if(p != "/" && p != "*")
  {
               out1+=p;
   j++; //считает кол-во добавок
  }
  if (p == "/")
   //запомнить позицию разделителя (начало) в новой строке out1:
                ArStart[Ar]=j; //i;
```

```
  if(p == "*")
  {
                //запомнить позицию разделителя (длина):
                 ArLen[Ar]=i — ArStart[Ar] -1;
                 Ar++;
  }
  }
  Ns[0]=Ar;
  out[0]=out1;

}

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

/// </summary>
System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support — do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
this->button1 = (gcnew System::Windows::Forms::Button());
this->linkLabel1 = (gcnew System::Windows::Forms::LinkLabel());
this->button2 = (gcnew System::Windows::Forms::Button());
this->textBox1 = (gcnew System::Windows::Forms::TextBox());
this->textBox2 = (gcnew System::Windows::Forms::TextBox());
this->label1 = (gcnew System::Windows::Forms::Label());
this->label2 = (gcnew System::Windows::Forms::Label());
this->SuspendLayout();
//
// button1
//
this->button1->Location = System::Drawing::Point(19, 17);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(18, 75);
this->button1->TabIndex = 4;
this->button1->Text = L"Выход";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
//
// linkLabel1
//
this->linkLabel1->AutoSize = true;
this->linkLabel1->LinkArea = System::Windows::Forms::LinkArea(0, 8);
this->linkLabel1->Location = System::Drawing::Point(64, 17);
```

```
this->linkLabel1->Name = L"linkLabel1";
this->linkLabel1->Size = System::Drawing::Size(55, 17);
this->linkLabel1->TabIndex = 5;
this->linkLabel1->TabStop = true;
this->linkLabel1->Text = L"linkLabel1";
this->linkLabel1->UseCompatibleTextRendering = true;
this->linkLabel1->LinkClicked += gcnew
System::Windows::Forms::LinkLabelLinkClickedEventHandler(this,
&Form1::linkLabel1_LinkClicked_1);
//
// button2
//
this->button2->Location = System::Drawing::Point(106, 47);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(114, 45);
this->button2->TabIndex = 6;
this->button2->Text = L"Сформировать множество ссылок\r\n";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this,
                            &Form1::button2_Click);
//
// textBox1
//
this->textBox1->Location = System::Drawing::Point(19, 113);
this->textBox1->Multiline = true;
this->textBox1->Name = L"textBox1";
this->textBox1->Size = System::Drawing::Size(186, 81);
this->textBox1->TabIndex = 7;
//
// textBox2
//
this->textBox2->Location = System::Drawing::Point(227, 115);
this->textBox2->Multiline = true;
this->textBox2->Name = L"textBox2";
this->textBox2->Size = System::Drawing::Size(196, 79);
this->textBox2->TabIndex = 8;
//
// label1
//
this->label1->AutoSize = true;
this->label1->Location = System::Drawing::Point(42, 98);
this->label1->Name = L"label1";
this->label1->Size = System::Drawing::Size(136, 13);
this->label1->TabIndex = 9;
this->label1->Text = L"Задание адресов ссылок\r\n";
//
// label2
//
```

```
this->label2->AutoSize = true;
this->label2->Location = System::Drawing::Point(212, 98);
this->label2->Name = L"label2";
this->label2->Size = System::Drawing::Size(233, 13);
this->label2->TabIndex = 10;
this->label2->Text = L"Формирование свойства  Text гиперссылки\r\n";
//
// Form1
//
this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
this->ClientSize = System::Drawing::Size(454, 206);
this->Controls->Add(this->label2);
this->Controls->Add(this->label1);
this->Controls->Add(this->textBox2);
this->Controls->Add(this->textBox1);
this->Controls->Add(this->button2);
this->Controls->Add(this->linkLabel1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
this->PerformLayout();

}
#pragma endregion

//------------------- Обработчики событий --------------------------
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
 this->Close();
 }
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
/* Требуем, чтобы первая ссылка всегда посещалась (т. е. помечалась после
щелчка на ее имени)*/
      this->linkLabel1->Links[ 0 ]->Visited = true;

   //формирование свойства Text с помощью TextBox:
  //кол-во строк в обоих TextBox'ах должно быть одинаковым

  this->linkLabel1->Text="";

//-------------------------------------------
 /*  /Register* Folder    Call /MSN*
Эту строку будем помещать в свойство Text. Для выделения частей, которые
попадут в качестве имени гиперссылки, применяем разделители:
"/" - для обозначения начала текста, а "*" - для обозначения конца текста
*/
```

Form1

Выхо

linkLabel1

Сформировать множество ссылок

Задание адресов ссылок

Сюда пишутся в виде отдельных строк адреса, по которым станут работать ссылки

Формирование свойства Text гиперссылки

Сюда подряд пишутся элементы, из которых будет формироваться строка ссылок. То, что получит статус ссылки, должно быть между символами "/" и "*"

Вид формы приложения после запуска

Form1

Выхо

Register Call BHV

Сформировать множество ссылок

Задание адресов ссылок

D:\Register
www.bhv.ru

Формирование свойства Text гиперссылки

/Register* /Call BHV*

Сформировали строки в левом и правом textBox'е, нажали на кнопку "Сформировать множество ссылок" и выше кнопки появилась строка из правого textBox, части которой приняли вид гиперссылки

Рис. 11.69. Создание нескольких гиперссылок. Часть 1

Рис. 11.70. Создание нескольких гиперссылок. Часть 2

```
// Формирование имен гиперссылок в свойстве Text и начала с длинной для
// свойства Links:
  String ^in;
  array <int> ^Ns = gcnew  array <int> (1);
  array <String ^> ^out = gcnew  array <String ^> (1);
  int tbl=this->textBox2->Text->Length;
  in=this->textBox2->Text;
  array <int> ^ArStart = gcnew  array <int> (tbl);
  array <int> ^ArLen  = gcnew  array <int> (tbl);

  PasteTextString(in, out, ArStart,ArLen,Ns);
  this->linkLabel1->Text=out[0];

//--------- Формирование собственно гиперссылок в свойстве Links --------------
   int NumLines=Ns[0];  //количество элементов массива
   for(int i=0; i< NumLines; i++)
    {
     if(i==0)  //для первой гиперссылки
      {
       this->linkLabel1->Links[ i ]->LinkData = this->textBox1->Lines[i];
//надо задать еще Start и Length для LinkArea:
this->linkLabel1->LinkArea.Start=ArStart[0];
this->linkLabel1->LinkArea.Length=ArLen[0];
       this->linkLabel1->Links[ 0 ]->Visited = true;
       this->linkLabel1->Links[ 0 ]->Enabled = true;
continue;
      }
  /*координаты берутся из сформированных массивов ArStart[],ArLen[],
  а сами ссылки — из TextBox1: */
        this->linkLabel1->Links->Add( (int)ArStart[i], (int)ArLen[i], this-
>textBox1->Lines[i]);
    }

  }
//------------------ Обработчик щелчка на имени гиперссылки -------------------

private: System::Void linkLabel1_LinkClicked_1(System::Object^  sender,
System::Windows::Forms::LinkLabelLinkClickedEventArgs^  e)
 {
  //Гиперссылка, на которой был щелчок, помечается как посещенная:
      this->linkLabel1->Links[ linkLabel1->Links->IndexOf( e->Link ) ]->Visited
= true;

      // Display the appropriate link based on the value of the
      // LinkData property of the Link Object*.
/*Подготовка данных для команды Start (извлекается адрес объекта)*/
      String^ target = dynamic_cast<String^>(e->Link->LinkData);
```

```
/*объект отыскивается и выводится на экран:*/
      System::Diagnostics::Process::Start( target );

  } //обработчик
 }; //форма
}  //программный модуль
```

Компонент *PictureBox*

Компонент находится в списке **All Windows Forms** палитры компонентов. Через этот компонент в форму выводится графическое изображение.

Какое изображение надо выводить, указывается в свойстве `Image`. Если нажать кнопку с многоточием в поле этого свойства, то откроется диалоговое окно для выбора объекта в форматах bmp, jpeg, icon, gif, png метафайла. Можно также загрузить изображение в форму, воспользовавшись свойством `ImageLocation` и методами `Load()` и `LoadAsync()`.

Компонент содержит свойства, определяющие, как выводить изображение внутри границ самого этого объекта (в форме `PictureBox` отображается в виде пустого квадрата).

Вид компонента в форме с открытым диалоговым окном **PictureBox Tasks** показан на рис. 11.71, перечень свойств, отображенных в окне **Properties**, приведен на рис. 11.72.

Рис. 11.71. Вид компонента `PictureBox` в форме

Некоторые свойства компонента *PictureBox*

- `Image` — задает изображение, загружаемое в компонент (в поле этого свойства имеется кнопка с многоточием, с помощью которой открывается диалоговое окно для загрузки изображения).

  Можно загружать и сохранять изображение также и в режиме исполнения приложения с помощью методов класса `PictureBox`.

  Так, например, `Load()` позволяет загружать изображение из файла, путь к которому указан в свойстве `ImageLocation`. Если в этом свойстве не задавать пути, а указать его в переменной типа `String` (например, `String ^ Url`), то с помощью

Рис. 11.72. Свойства `PictureBox`

метода `Load (Url)` в форму также можно загрузить изображение. В этом случае метод `Load()` сам назначит свойству `ImageLocation` значение переменной `Url` и далее станет работать как этот же метод в своей первой форме (т. е. без параметра).

Пример работы этой функции в обеих формах показан на рис. 11.73 (там же можно увидеть, как работать с `TextBox`, свойство которого `Multiline` надо установить в `true`, чтобы поле компонента растягивалось). Когда информация не помещается в поле, надо его перевести в режим `Multiline` и растянуть до требуемого размера, что и сделано, поэтому весь путь к изображению виден полностью. Тексты обработчиков кнопок показаны в листинге 11.16.

Листинг 11.16

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
        {
      this->pictureBox1->ImageLocation=this->textBox1->Text;

      //загружает изображение, адрес которого указан в свойстве
      //ImageLocation:

      this->pictureBox1->Load();
        }
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^ e)
        {
              this->Close();
        }
```

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
{
        /*Эта форма Load() сама устанавливает свойство ImageLocation
    в значение, полученное от переменной Url*/

    String ^Url;
    Url=this->textBox1->Text;
    this->pictureBox1->Load(Url);
}
```

Рис. 11.73. Пример загрузки изображений разного типа различными формами `Load()`

- `Size Mode` — дает возможность регулирования изображения в отведенном пространстве компонента `PictureBox`. Это свойство имеет ряд значений, которые можно выбирать из выпадающего списка, открывающегося кнопкой в поле этого свойства:
  - `Normal` — размещает изображение в левом верхнем углу пространства `PictureBox`, а не вмещающаяся часть изображения отрезается;
  - `StretchImage` — изображение принимает размеры и форму компонента. Если компонент изменит размеры, то изображение тоже изменит размеры;
  - `AutoSize` — заставляет компонент изменить свои размеры и принять размеры самого изображения;
  - `CenterImage` — помещает изображение в центр пространства компонента, не меняя его размера (т. е. если размер изображения больше размера пространства компонента, то изображение обрезается);

- `Zoom` — это свойство обеспечивает соблюдение пропорций изображения при подгонке его к размерам пространства компонента.

Значение `StretchImage` просто втискивает изображение в размеры компонента и может исказить изображение, а `Zoom` тоже втискивает, но при этом не нарушает пропорций, т. е. не искажает изображение.

- `ErrorImage` — здесь можно задать изображение, которое станет выводиться вместо запрашиваемого, если последнее не удается загрузить из-за возникающих ошибок или по причине отмены загрузки.
- `InitialImage` — с помощью этого свойства можно задать вывод "успокаивающего" изображения, которое выводится в компонент на то время, пока основное изображение загружается.

Компонент *DateTimePicker*

Компонент (календарь) находится в списке **All Windows Forms** палитры компонентов и позволяет пользователю выбирать необходимую дату или время. При выборе даты или времени (это задается в свойстве `Format`) компонент может представляться в двух формах: в виде прямоугольного поля, в котором высвечивается дата или время, и в виде выпадающего списка с датами (рис. 11.74).

Рис. 11.74. Формы компонента `DateTimePicker` при выборе информации и сам выбор

Выборка данных в обеих формах организована с использованием механизма прокрутки.

В форме без выпадающего списка используют установку свойства `ShowUpDown` в значение `true`. Тогда вместо кнопки для раскрытия выпадающего списка появляется элемент прокрутки содержимого поля — прямоугольник со стрелками вверх/вниз (больше/меньше). С его помощью можно увеличивать или уменьшать значения отмеченного элемента данного, тем самым значительно ускоряя поиск. Если, например, в поле находится дата в формате **21 марта 2012 г.**, то при необходимости изменения месяца или года вы щелкаете на элементе **21** и начинаете нажимать кнопкой мыши на стрелки "вверх/вниз" прокручивающего механизма (таким способом вы подгоняете под нужные значения месяц и год).

Если же вы работаете с другой формой выборки даты — из выпадающего списка, то список открывается кнопкой с галочкой (флажком). В окне списка тоже имеются прокручивающие механизмы (по краям окна — стрелки "влево/вправо") не только в целом для даты, но и отдельно для выбора месяца и прокрутки величины года. Если включено свойство `ShowCheckBox`, то слева от выбранной даты появляется флажок (при условии, что свойство `Checked` тоже будет установлено в `true`). Если флажок включен, то выбранную дату (кроме наименования месяца) можно редактировать (щелчком мыши надо отметить соответствующий элемент, а затем ввести с клавиатуры свое значение). Если флажок выключить, дата становится недоступной.

Выбранная с помощью `DateTimePicker` дата помещается в его свойство `Value`, откуда ее можно брать в режиме исполнения приложения. Существуют свойства `MaxDate` и `MinDate`, которые задают диапазон изменения даты. Значения этих свойств можно установить тоже с помощью механизма календаря (календарь открывается для выборки из него необходимой даты, если нажать кнопку с галочкой в поле каждого из этих свойств).

Задание этих свойств — механизм контроля выборки (при осуществлении выборки система не позволит выбрать даты вне указанного диапазона). Например, если установить диапазон между 01.03.2012 и 08.03.2012, то вы не сможете набрать, например, дату 09.03.2012. При разработке конкретного приложения можно использовать этот механизм, задавая диапазон дат во время исполнения приложения, что будет определенной гарантией того, что оператор, вводя даты, не совершит ошибки.

Значения дат могут выводиться в 4-х форматах, что определяется свойством `Format`:

- `Long` — длинный формат. Если выбрать это значение, то дата станет выводиться так: 12 апреля 2012 г.;
- `Short` — короткий формат. Если выбрать это значение, то дата станет выводиться в виде: 12.04.2012;
- `Time` — формат времени;
- `Custom` — при выборе этого формата надо установить свойство `CustomFormat` в соответствующее значение, принятое в среде разработки (об этом чуть позже).

Рис. 11.75. Свойства компонента `DateTimePicker`

Перечень свойств `DateTimePicker`, отображенных в его окне **Properties**, показан на рис. 11.75.

Форматные строки даты и времени

Дата и время относятся к типу `DateTime`, который имеет специальный механизм, позволяющий представлять эти две величины в виде строки. Причем имеются две категории перевода даты в строку: стандартная и пользовательская. Там, где стандартный способ перевода даты в строку не подходит, применяется другой механизм — так называемый пользовательский формат.

Для чего нужен перевод даты в строку?

Например, вы получили из календаря некоторую дату и хотите ее вывести на экран в удобном для вас формате. Допустим, что вы хотите вывести дату в виде последовательности цифр, разделенных на группы (число, месяц, год), между которыми должны быть специальные символы-разделители (например, звездочки). К тому же время и дата должны быть отформатированы в соответствии с принятыми в данной стране правилами. Такое представление — это уже не число, а обыкновенная строка.

Форматирование дат происходит с помощью задания специальной форматной строки, а когда такую строку не задают, то форматирование происходит по правилам умолчания. Форматная строка содержит специальные символы форматирования, определяющие, как значение даты станет преобразовываться.

Стандартное и пользовательское форматирование

Стандартное форматирование состоит в применении набора символов форматирования, представленных в табл. 11.6. Если задано стандартное форматирование даты, а символа форматирования нет в таблице, которая хранится в соответствующем классе, то система выдаст ошибку. Вид результата форматирования (т. е. строка после форматирования) зависит от региональных установок вашего компьютера, задаваемых через панель управления. Компьютеры с различными региональными установками будут, естественно, выдавать разные результаты форматирования.

Таблица 11.6. *Символы форматирования*

Форматный символ	Описание
d	Так задают день месяца. Число, состоящее из одной цифры, не будет иметь впереди нуля
dd	Так задают день месяца, но в этом случае число, состоящее из одной цифры, будет иметь впереди нуль
ddd	При таком форматировании станут выдаваться не названия дней недели, а их аббревиатуры
dddd	При таком форматировании станут выдаваться полные названия дней недели
M	Месяц будет выдаваться в виде числа. Месяц, представленный одной цифрой, не будет иметь впереди нуля
MM	Месяц будет выдаваться в виде числа. Месяц, представленный одной цифрой, будет иметь впереди нуль
MMM	При таком форматировании станут выдаваться не названия месяцев, а их аббревиатуры
MMMM	При таком форматировании будет выдаваться полное название месяца
y	При таком форматировании будет выдаваться год, но без выдачи века. Если год представляется одной цифрой (например, 7-й год), то перед ним нуля не будет
yy	При таком форматировании будет выдаваться год, но без выдачи века. Если год представляется одной цифрой, то в выводе перед ним появится нуль
yyyy	При таком форматировании будет выдаваться год из четырех цифр, включая век
h	При таком форматировании выводится время в 12-часовом формате. Одноразрядное значение времени не будет иметь впереди нуля
hh	Такой же смысл, что и для h, но с нулем впереди числа
H	При таком форматировании выводится время в 24-часовом формате. Одноразрядное значение времени не будет иметь впереди нуля
HH	Такой же смысл, что и для H, но с нулем впереди числа
m	Так выводятся минуты. Одноразрядное значение не будет иметь впереди нуля
mm	Такой же смысл, что и для m, но с нулем впереди числа

Таблица 11.6 *(окончание)*

Форматный символ	Описание
s	Так выводятся секунды. Одноразрядное значение не будет иметь впереди нуля
ss	Такой же смысл, что и для s, но с нулем впереди числа
f	Так выводятся доли секунды. Если секунда — это одноразрядное число, то лишние цифры отсекаются
ff	Так выводятся доли секунды. Если секунда — это двухразрядное число, то лишние цифры отсекаются
\ c	Здесь "c" — это любой символ. Чтобы вывести обратную косую черту (бэкслэш), надо использовать символы "\\"

Примечание

В табл. 11.6 опущены некоторые специфические форматы.

Пример форматирования дат приведен в приложении, текст обработчиков которого показан в листинге 11.17, а форма в режиме дизайна и в режиме исполнения — на рис. 11.76.

Рис. 11.76. Пример форматирования дат

Листинг 11.17

```
#pragma once

namespace My117_2010Форматированиедат {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
```

```
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    using namespace System::Globalization;  //для даты

    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
    private: System::Windows::Forms::DateTimePicker^  dateTimePicker1;
    protected:
    private: System::Windows::Forms::Button^  button1;
    private: System::Windows::Forms::Button^  button2;
    private: System::Windows::Forms::Label^  label1;
    private: System::Windows::Forms::Label^  label2;
    private: System::Windows::Forms::ListBox^  listBox1;
    private: System::Windows::Forms::ListBox^  listBox2;

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
```

```
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->dateTimePicker1 = (gcnew
System::Windows::Forms::DateTimePicker());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->button2 = (gcnew System::Windows::Forms::Button());
            this->label1 = (gcnew System::Windows::Forms::Label());
            this->label2 = (gcnew System::Windows::Forms::Label());
            this->listBox1 = (gcnew System::Windows::Forms::ListBox());
            this->listBox2 = (gcnew System::Windows::Forms::ListBox());
            this->SuspendLayout();
            //
            // dateTimePicker1
            //
            this->dateTimePicker1->Location = System::Drawing::Point(34, 1);
            this->dateTimePicker1->Name = L"dateTimePicker1";
            this->dateTimePicker1->Size = System::Drawing::Size(127, 20);
            this->dateTimePicker1->TabIndex = 0;
            //
            // button1
            //
            this->button1->Location = System::Drawing::Point(3, 1);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(25, 99);
            this->button1->TabIndex = 1;
            this->button1->Text = L"Выход";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
            //
            // button2
            //
            this->button2->Location = System::Drawing::Point(34, 26);
            this->button2->Name = L"button2";
            this->button2->Size = System::Drawing::Size(65, 49);
            this->button2->TabIndex = 2;
            this->button2->Text = L"Форматировать дату";
            this->button2->UseVisualStyleBackColor = true;
            this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
            //
            // label1
            //
            this->label1->AutoSize = true;
            this->label1->Location = System::Drawing::Point(66, 108);
```

```
        this->label1->Name = L"label1";
        this->label1->Size = System::Drawing::Size(57, 13);
        this->label1->TabIndex = 3;
        this->label1->Text = L"Форматы\r\n";
        //
        // label2
        //
        this->label2->AutoSize = true;
        this->label2->Location = System::Drawing::Point(218, 108);
        this->label2->Name = L"label2";
        this->label2->Size = System::Drawing::Size(157, 13);
        this->label2->TabIndex = 4;
        this->label2->Text = L"Результаты форматирования";
        //
        // listBox1
        //
        this->listBox1->FormattingEnabled = true;
        this->listBox1->Location = System::Drawing::Point(12, 124);
        this->listBox1->Name = L"listBox1";
        this->listBox1->Size = System::Drawing::Size(185, 95);
        this->listBox1->TabIndex = 5;
        //
        // listBox2
        //
        this->listBox2->FormattingEnabled = true;
        this->listBox2->Location = System::Drawing::Point(203, 124);
        this->listBox2->Name = L"listBox2";
        this->listBox2->Size = System::Drawing::Size(199, 95);
        this->listBox2->TabIndex = 6;
        //
        // Form1
        //
        this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
        this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
        this->ClientSize = System::Drawing::Size(413, 223);
        this->Controls->Add(this->listBox2);
        this->Controls->Add(this->listBox1);
        this->Controls->Add(this->label2);
        this->Controls->Add(this->label1);
        this->Controls->Add(this->button2);
        this->Controls->Add(this->button1);
        this->Controls->Add(this->dateTimePicker1);
        this->Name = L"Form1";
        this->Text = L"Form1";
        this->ResumeLayout(false);
        this->PerformLayout();

    }
```

```
#pragma endregion
    private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
              {
               this->Close();
              }
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
          {
          DateTime ^dt=DateTime();
 String ^s;
//объявлен массив строк с именем format и инициализирован форматами
 array<String^>^format = {L"d/M/y",L"dd/MM/yy",L"dd/MM/yyyy",L"d MMM yyyy
\'г\', ddd",L"d  MMMM yyyy \'г\', dddd",L"dd/MM/yy HH::mm",L"ddd, MMM d
\"'\"yy"};
   dt=this->dateTimePicker1->Value; //здесь будет выбранная дата
   for ( int i = 0; i < format->Length; i++ )
   {
//CurrentInfo — учет национальных особенностей (см. региональные
//установки на панели управления)
   s=dt->ToString( format[ i ], DateTimeFormatInfo::CurrentInfo);   //перевод
даты в строку с одновременным форматированием
   this->listBox1->Items->Add(format[ i ]);
   this->listBox2->Items->Add(s);
   } //for
}//Обработчик
};
}
```

Если нужны региональные настройки, то в перечень используемых пространств h-файла надо добавить строку:

```
using namespace System::Globalization;  //для даты
```

Некоторые сведения о работе с датами

При разработке приложений часто приходится сталкиваться с необходимостью работы с датами, а не только с форматированным выводом дат. Приходится заниматься декомпозицией даты (разбиением ее на число, месяц, год) и последующей работой с каждой из ее частей, приходится добавлять к дате дни, месяцы и годы, сравнивать даты, складывать и вычитать их, переводить содержимое строки, в которой записана дата, в тип `DateTime` (последнее, кстати, выполняет метод `Parse(String)`). Работа с датами основывается на использовании элементов класса `DateTime`. Сожержимое данного класса легко найти в справочной системе среды, выйдя через клавишу <F1>, когда `DateTime` подсвечено, на документацию в Интернете (MSDN) и набрав в открывшемся окне в его поисковом поле: `DateTime`. Далее показаны несколько методов этого класса:

- `AddDays` — добавка заданного количества дней к значению даты;
- `AddHours` — добавка заданного количества часов к значению даты;

- `AddMilliseconds` — добавка заданного количества миллисекунд к значению даты;
- `AddMinutes` — добавка заданного количества минут к значению даты;
- `AddMonths` — добавка заданного количества месяцев к значению даты;
- `AddSeconds` — добавка заданного количества секунд к значению даты;
- `AddTicks` — добавка заданного количества тиков к значению даты;
- `AddYears` — добавка заданного количества лет к значению даты;
- `Compare` — сравнение двух объектов типа `DateTime` и возврат целого, которое указывает, меньше ли первая дата второй или нет;
- `DaysInMonth` — возвращает количество дней в месяце в указанном месяце года;
- `FromBinary` — переводит 64-разрядную величину в тип `DateTime`.

Остановимся более подробно на вычитании дат. На рис. 11.77 показаны результаты вычитания разных дат и вид формы с компонентами, обеспечивающими эти операции. В листинге 11.18 приведены тексты обработчиков событий этого приложения.

Рис. 11.77. Операции с датами

Листинг 11.18

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
 this->Close();
 }
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
 String ^s;
```

```
 System::DateTime date1 = System::DateTime( 2010, 6, 3, 22, 15, 0 );
System::DateTime date2 = System::DateTime( 2010, 12, 6, 13, 2, 0 );

/*Это один из конструкторов класса DateTime, инициализирующий
экземпляр класса значениями (год, месяц, число, часы, минуты, секунды)*/

System::DateTime date3 = System::DateTime( 2010, 10, 12, 8, 42, 0 );

// diff1 (разность) равна 185 дней, 14 часов, 47 минут.
// Класс TimeSpan задает интервал времени. У него есть свои события и методы
System::TimeSpan diff1 = date2.Subtract( date1 );
this->listBox1->Items->Add(date2.ToString());
this->listBox2->Items->Add(date1.ToString());
this->listBox3->Items->Add(diff1.ToString());

// date4 получается  17:55:00
System::DateTime date4 = date3.Subtract( diff1 );
this->listBox1->Items->Add(date3.ToString());
this->listBox2->Items->Add(diff1.ToString());
this->listBox3->Items->Add(date4.ToString());

// diff2 (разность) равна 55 дней, 4 часа,  20 минут.
System::TimeSpan diff2 = date2 — date3;
this->listBox1->Items->Add(date2.ToString());
this->listBox2->Items->Add(date3.ToString());
this->listBox3->Items->Add(diff2.ToString());

// date5 получается 17:55:00
System::DateTime date5 = date1 — diff2;
this->listBox1->Items->Add(date1.ToString());
this->listBox2->Items->Add(diff2.ToString());
this->listBox3->Items->Add(date5.ToString());
}
```

Теперь посмотрим, как работают некоторые методы работы с датами. На рис. 11.77 показаны результаты сложения дат и вид формы с компонентами, обеспечивающими эти операции. К первому слагаемому добавлялось построчно количество дней, месяцев и лет. Проверялись методы `Add()`, `AddDays()`, `AddMonths()`, `AddYears()`.

В листинге 11.19 приведен текст программы этого приложения.

Листинг 11.19

```
#pragma once

namespace AddDate2011 {

    using namespace System;
    using namespace System::ComponentModel;
```

```
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::Globalization;  //for FormatInfo

/// <summary>
/// Summary for Form1
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
public:
    Form1(void)
    {
        InitializeComponent();
        //
        //TODO: Add the constructor code here
        //
    }

protected:
    /// <summary>
    /// Clean up any resources being used.
    /// </summary>
    ~Form1()
    {
        if (components)
        {
          delete components;
        }
    }
private: System::Windows::Forms::Button^  button1;
protected:
private: System::Windows::Forms::Button^  button2;
private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::Label^  label3;
private: System::Windows::Forms::ListBox^  listBox1;
private: System::Windows::Forms::ListBox^  listBox2;
private: System::Windows::Forms::ListBox^  listBox3;

private:
    /// <summary>
    /// Required designer variable.
    /// </summary>
    System::ComponentModel::Container ^components;
```

```
#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->button2 = (gcnew System::Windows::Forms::Button());
            this->label1 = (gcnew System::Windows::Forms::Label());
            this->label2 = (gcnew System::Windows::Forms::Label());
            this->label3 = (gcnew System::Windows::Forms::Label());
            this->listBox1 = (gcnew System::Windows::Forms::ListBox());
            this->listBox2 = (gcnew System::Windows::Forms::ListBox());
            this->listBox3 = (gcnew System::Windows::Forms::ListBox());
            this->SuspendLayout();
            //
            // button1
            //
        this->button1->Location = System::Drawing::Point(13, 13);
        this->button1->Name = L"button1";
        this->button1->Size = System::Drawing::Size(75, 23);
        this->button1->TabIndex = 0;
        this->button1->Text = L"Пуск";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this,&Form1::button1_Click);
            //
            // button2
            //
        this->button2->Location = System::Drawing::Point(387, 12);
        this->button2->Name = L"button2";
        this->button2->Size = System::Drawing::Size(75, 23);
        this->button2->TabIndex = 1;
        this->button2->Text = L"Выход";
        this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
            //
            // label1
            //
            this->label1->AutoSize = true;
        this->label1->Location = System::Drawing::Point(13, 56);
            this->label1->Name = L"label1";
            this->label1->Size = System::Drawing::Size(102, 17);
            this->label1->TabIndex = 2;
            this->label1->Text = L"1-е слагаемое";
```

```
        //
        // label2
        //
        this->label2->AutoSize = true;
    this->label2->Location = System::Drawing::Point(193, 56);
        this->label2->Name = L"label2";
        this->label2->Size = System::Drawing::Size(102, 17);
        this->label2->TabIndex = 3;
        this->label2->Text = L"2-е слагаемое";
        //
        // label3
        //
        this->label3->AutoSize = true;
    this->label3->Location = System::Drawing::Point(355, 56);
        this->label3->Name = L"label3";
        this->label3->Size = System::Drawing::Size(76, 17);
        this->label3->TabIndex = 4;
        this->label3->Text = L"Результат";
        //
        // listBox1
        //
        this->listBox1->FormattingEnabled = true;
        this->listBox1->ItemHeight = 16;
    this->listBox1->Location = System::Drawing::Point(16, 85);
        this->listBox1->Name = L"listBox1";
        this->listBox1->Size = System::Drawing::Size(161, 84);
        this->listBox1->TabIndex = 5;
        //
        // listBox2
        //
        this->listBox2->FormattingEnabled = true;
        this->listBox2->ItemHeight = 16;
    this->listBox2->Location = System::Drawing::Point(196, 85);
        this->listBox2->Name = L"listBox2";
        this->listBox2->Size = System::Drawing::Size(156, 84);
        this->listBox2->TabIndex = 6;
        //
        // listBox3
        //
        this->listBox3->FormattingEnabled = true;
        this->listBox3->ItemHeight = 16;
    this->listBox3->Location = System::Drawing::Point(358, 85);
        this->listBox3->Name = L"listBox3";
        this->listBox3->Size = System::Drawing::Size(138, 84);
        this->listBox3->TabIndex = 7;
        //
        // Form1
        //
```

```
        this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(498, 174);
            this->Controls->Add(this->listBox3);
            this->Controls->Add(this->listBox2);
            this->Controls->Add(this->listBox1);
            this->Controls->Add(this->label3);
            this->Controls->Add(this->label2);
            this->Controls->Add(this->label1);
            this->Controls->Add(this->button2);
            this->Controls->Add(this->button1);
            this->Name = L"Form1";
            this->Text = L"Сложение дат";
            this->ResumeLayout(false);
            this->PerformLayout();

        }
#pragma endregion
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
    {
     this->Close();
    }
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{
String ^s;

/*Метод Add()- добавляет интервал времени к данной дате
               Интервал задается данными класса TimeSpan:
               запускают один из его конструкторов, который
               и задает добавку в днях к данному экземпляру даты*/
/*Здесь подсчитывается, какой день недели будет, если к текущей дате
 добавить 36 дней*/

System::DateTime today = System::DateTime::Now; //текущая дата
System::TimeSpan duration( 36, 0, 0, 0 );

/*Конструктор придает начальные значения членам класса:
  задаются количество дней, часы, минуты, секунды.
  Фактически это структура: 1-й элемент — интервал времени в днях,
  2-й элемент — интервал времени в часах и т. д.
  Мы станем использовать интервал "количество дней"*/

System::DateTime answer = today.Add( duration );

/*Эта функция добавляет данные структуры duration
  к данным структуры даты*/

s=answer.ToString("dddd",DateTimeFormatInfo::CurrentInfo);
```

```
//вывод даты в строку с учетом Региональных настроек

this->listBox3->Items->Add(s); //запись результата в ListBox
s=today.ToString("dd/MM/yyyy \'г\', dddd", DateTimeFormatInfo::CurrentInfo);
this->listBox1->Items->Add(s); //запись 1-го слагаемого в ListBox
s=duration.ToString();
this->listBox2->Items->Add(s); //запись 2-го слагаемого в ListBox
//==============================================================

//Проверка функции добавления дней к дате

double d=36;
answer = today.AddDays( d );
s=answer.ToString("dddd",DateTimeFormatInfo::CurrentInfo);

//вывод даты в строку с учетом региональных настроек

this->listBox3->Items->Add(s); //запись результата в ListBox
s=today.ToString("dd/MM/yyyy \'г\', dddd", DateTimeFormatInfo::CurrentInfo);
this->listBox1->Items->Add(s); //запись 1-го слагаемого в ListBox
s=d.ToString();
this->listBox2->Items->Add(s); //запись 2-го слагаемого в ListBox
//==============================================================

//Проверка функции добавления месяцев к дате

int d1=36;
answer = today.AddMonths( d1 );
s=answer.ToString("dddd",DateTimeFormatInfo::CurrentInfo);

//вывод даты в строку с учетом региональных настроек

this->listBox3->Items->Add(s); //запись результата в ListBox
s=today.ToString("dd/MM/yyyy \'г\', dddd", DateTimeFormatInfo::CurrentInfo);
this->listBox1->Items->Add(s); //запись 1-го слагаемого в ListBox
s=d1.ToString();
this->listBox2->Items->Add(s); //запись 2-го слагаемого в ListBox
//==============================================================

//Проверка функции добавления лет к дате

int d2=36;
answer = today.AddYears( d2 );
s=answer.ToString("dddd",DateTimeFormatInfo::CurrentInfo);

//вывод даты в строку с учетом региональных настроек

this->listBox3->Items->Add(s); //запись результата в ListBox
s=today.ToString("dd/MM/yyyy \'г\', dddd", DateTimeFormatInfo::CurrentInfo);
```

```
this->listBox1->Items->Add(s); //запись 1-го слагаемого в ListBox
s=d2.ToString();
this->listBox2->Items->Add(s); //запись 2-го слагаемого в ListBox
}
};
}
```

Примечание

На рис. 11.77 последние две строки результата получились странными, потому что второе слагаемое — это уже не дни, а месяцы и годы.

Компонент *TabControl*

Компонент находится в списке **All Windows Forms** палитры компонентов. Он позволяет построить набор страниц, которые друг друга перекрывают на экране и которые можно перелистывать. Но главная их ценность в том, что на эти страницы, находящиеся в рамках одной формы, где размещен сам компонент `TabControl`, можно помещать другие компоненты, тем самым расширяя возможности формы. Например, вы должны разработать приложение по управлению кадрами предприятия. Все компоненты, обеспечивающие решение этой проблемы, можно разместить в одной форме, но на разных страницах компонента `TabControl`, который достаточно поместить в форму. На каждой странице вы можете разместить по одному справочнику и организовать на ней ведение справочника, на других страницах можете разместить элементы управления для получения аналитических таблиц и т. д. Чтобы переключаться между страницами, достаточно щелкнуть мышью на любой из них. Вид компонента в форме с раскрытым диалоговым окном **TabControl Tasks** показан на рис. 11.78.

Рис. 11.78. Вид компонента `TabControl` в форме

Самым важным свойством `TabControl` является свойство `TabPages`, которое содержит отдельные страницы. Каждая вкладка — это отдельный объект `TabPage` со своими свойствами, методами и событиями. Когда вы переключаетесь с одной страницы на другую, щелкая мышью на вкладке, возникает событие `Click` для объекта `TabPage`.

Как задавать страницы

Поместите компонент `TabControl` в форму и откройте его диалоговое окно **TabControl Tasks** (см. рис. 11.78). Там есть две опции, позволяющие добавлять или удалять страницы. Перемещаться по уже сформированным страницам можно посредством щелчка мышью на вкладке каждой страницы. Если страницы не умещаются в поле компонента, то автоматически формируются кнопки прокрутки.

Если вам неудобно работать с кнопками прокрутки, и вы хотите видеть все страницы сразу, тогда задайте значение свойства `MultiLine` равным `true`, — получите желаемый результат. Если случится, что не все вкладки в многостраничном режиме появляются, то следует установить свойство `Width` (ширина) для компонента `TabControl` на такое значение, чтобы ширина самого компонента была больше, чем ширина всех его вкладок.

На вкладке можно поместить пиктограмму. Для чего это делается? При эксплуатации приложения пользователям бывает некомфортно каждый раз вчитываться в название страницы на вкладке. Легче всего запоминается пиктограмма, на которую следует нажимать в необходимых случаях.

Чтобы создать вкладки с пиктограммами, нужно в форму поместить компонент `ImageList`, задать в нем пиктограммы (их надо заранее заготовить в соответствующих файлах с расширением icon средствами Photoshop или другого инструмента), затем установить свойство `ImageList` компонента `TabControl` (привязать компонент `ImageList` через это свойство к компоненту `TabControl`). Тогда все пиктограммы, находящиеся в списке компонента `imageList`, будут видны и на вкладках компонента `TabControl`.

Теперь следует активизировать нужную вкладку и ее свойство `ImageIndex` установить на индекс подходящей пиктограммы (когда вы откроете выпадающий список свойства вкладки `ImageIndex`, в нем будут видны все пиктограммы со значениями их индексов (номеров)). При активизации вкладки могут возникнуть проблемы: не всегда удается (из-за неопытности начинающего) переключаться с активного компонента `TabControl` на соответствующую вкладку `TabPage`. Если вы просто щелкните на вкладке, станет активным весь компонент `TabControl`. Если вы щелкните в поле страницы, которую открывает данная вкладка, то станет активной именно страница, т. е. объект `TabPage`. Вот с его свойствами и надо будет работать для задания пиктограммы на вкладке.

Расположение вкладок в компоненте `TabControl` можно менять: помещать их сверху, снизу, слева и справа. Достаточно выбрать соответствующее значение свойства `Alignment` компонента `TabControl` из выпадающего списка. Надо твердо помнить, что `TabControl` и `TabPage` (его составная часть) — это все-таки два разных объекта, которые можно по-своему настраивать с помощью их собственных свойств и методов. Все, что относится ко всем страницам, находится в `TabControl`. Все, что относится к отдельной странице, регулируется через компонент `TabPage`.

Вы можете изменить форму вкладок (не надо путать вкладку, как заголовок страницы, и вкладку в более широком смысле — в смысле страницы блокнота

`TabControl`). Для этого нужно воспользоваться свойством `Appearance` компонента `TabControl` — все вкладки могут одновременно изменять форму, а для отдельной вкладки это не имеет места (поскольку изменяется значение свойства главного компонента — родителя). Заголовки могут иметь три формы: нормальную, в виде кнопок, в виде таких же кнопок, но разделенных между собой вертикальными полосами.

Перечень свойств `TabControl`, отображенных в окне **Properties**, показан на рис. 11.79.

Рис. 11.79. Перечень свойств `TabControl`

Мы задавали страницы компонента `TabControl`, используя его диалоговое окно, которое открывается кнопкой в правом верхнем углу активного компонента. Однако более удобный способ задания страниц — это воспользоваться его свойством `TabPages`. Если нажать кнопку с многоточием в поле этого свойства, то откроется знакомое нам по изучению предыдущих компонентов диалоговое окно для работы со страницами (рис. 11.80). Здесь можно не только добавлять, удалять, менять местами страницы, но также сразу устанавливать их основные свойства, не обращаясь к окну **Properties**.

Обратите внимание и на следующие два свойства:

- `SelectedIndex` — свойство, которое содержит номер активной страницы (если ему присвоить целое значение, то страница с соответствующим номером (отсчет

от нуля) станет активной (доступной)). Если в `TabControl` ни одна страница не выбрана, то значение `SelectedIndex` равно –1.

Если надо программно открыть страницу, то следует выполнить такую команду (например, для страницы 2):

```
this->tabControl1->SelectedIndex = 1;
```

- `SelectedTab` — представляет собой выбранную страницу (объект `TabPage`). Если требуется программно открыть страницу, то надо выполнить такую команду (например, для страницы 2):

```
this->tabControl1->SelectedTab = tabPage2;
```

Рис. 11.80. Диалоговое окно для работы со страницами

Некоторые методы *TabControl*

Из методов компонента отметим методы `Hide()` и `Show()`, позволяющие делать страницу невидимой и, наоборот, видимой.

Представляет интерес метод `SelectTab()`, имеющий три варианта реализации и позволяющий в режиме исполнения приложения активизировать нужную страницу. Здесь в качестве параметра задаются:

- номер страницы (отсчет от нуля);
- имя страницы, заданное в переменной типа `String ^`;
- имя страницы, заданное в свойстве `Name`.

Форма приложения, демонстрирующего работу метода, а также результаты его выполнения приведены на рис. 11.81, тексты обработчиков — в листинге 11.20.

Рис. 11.81. Демонстрация работы метода активизации страниц

Листинг 11.20

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
  if(i== 0)
   {
    this->tabControl1->SelectTab(i+1);
    i=1;
   }
  else
  {
   this->tabControl1->SelectTab(i-1);
   i=0;
  }
 }
```

Переменная `i` объявлена как `int i`:

```
private:
            /// <summary>
            /// Required designer variable.
            int i;
```

Некоторые свойства страницы *TabPage*

Каждая страница компонента `TabControl` представляет собой отдельный объект — экземпляр класса `TabPage` со своими свойствами, методами и событиями.

Перечень некоторых свойств страницы, отображенных в окне **Properties**, показан на рис. 11.82. Там же приводится измененный стиль окантовки страницы и появление полос прокрутки, которые явились следствием изменения свойств `BorderStyle` и `AutoScroll`.

Рис. 11.82. Свойства страницы компонента `TabControl`

Как защитить страницу от неавторизованного доступа

Бывают случаи, когда требуется защитить информацию, расположенную на странице, от неавторизованного доступа. Например, на странице хранятся сведения, доступ к которым ограничен и может происходить только по паролю. Поэтому разработчик приложения должен уметь отказывать в доступе к странице. Закрыть страницу можно программным способом (в обработчике события `SelectedIndexChanged` компонента `TabControl`). Это событие возникает, когда пользователь переключается от одной страницы к другой. Надо проверить права пользователя, принятые для данного приложения, и если прав недостаточно, то закрыть страницу, к которой должен быть переход, выдать пользователю соответствующее сообщение и вернуться на предыдущую страницу.

Рассмотрим такой простой вариан защиты доступа к странице: запишем в `ListBox` имена закрытых страниц (можно усложнить задание и добавить пароли доступа к странице, по которым они открываются). А в обработчике события `SelectedIndexChanged` проверять имя открываемой страницы на попадание ее в "черный" список. Если страница присутствует в этом списке, то следует уменьшить ее индекс на единицу и вернуться к предыдущей странице. Пример такого обработчика (без проверки на пароли, т. к. это уже сугубо индивидуальные структуры) приведен в листинге 11.21, а результаты работы — на рис. 11.83 и 11.84. Отметим, что для добавления страницы в компонент на этапе дизайна надо открыть его контекстное меню и выполнить опцию **Add Tab**. А чтобы установить курсор мыши на `TabControl`, надо установить его на полосу после последней вкладки.

Рис. 11.83. Закрытие доступа к некоторым страницам. Часть 1

Рис. 11.84. Закрытие доступа к некоторым страницам. Часть 2

Листинг 11.21

```
private: System::Void tabControl1_SelectedIndexChanged_1(System::Object^
sender, System::EventArgs^  e)
 {
  String ^s=this->tabControl1->SelectedTab->Name;
  for(int i=0; i < this->listBox1->Items->Count; i++)
   {
    if(!s->Compare(s,this->listBox1->Items[i]->ToString()))  //можно
                                        //закрыть страницу таким способом
     {
      MessageBox::Show("Эта страница для Вас закрыта", "Приложение 66",
        MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
```

```
                //вернуть индекс назад
        Int j=this->tabControl1->SelectedIndex -1;
        this->tabControl1->SelectTab(j);
        break;
        }
    }//for
  }
```

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. На основе уже знакомых нам компонентов рассмотрим простейшую задачу защиты — 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, то доступ к приложению открыт.

Рассмотрим приложение, в котором в качестве основного компонента используется `TabControl`. В нем должно разместиться все приложение, начиная со 2-й страницы (а на 1-й странице расположится программа регистрации). Если регистрация пройдет успешно, то откроется 2-я страница. Мы специально взяли всего две страницы для примера (в реальной ситуации надо открыть доступ к соответствующим страницам по паролю).

Уже на первой странице располагаются компоненты: кнопки, `TextBox` и `ListBox`. Здесь в режиме дизайна формируются три строки с паролями. Пароль состоит из трех частей, отделенных друг от друга косой чертой: имя пользователя, собственно пароль и дата, до которой действует пароль.

Свойство `Visuble ListBox` устанавливается в `false`, чтобы в режиме исполнения он не был виден. Кнопка, разрешающая администратору задачи открывать список паролей для просмотра, тоже заблокирована (становится недоступной для нажатия) — ее свойство `Enabled` на этапе проектирования установлено в `false`.

У администратора имеется свой пароль (admin), который для простоты "зашит" в программу. Если пользователь наберет имя, соответствующее паролю администратора, кнопка разблокируется и высветится список паролей. При повторном нажатии на эту кнопку список паролей исчезает, и кнопка снова блокируется.

Во время работы на первой вкладке остальные будут заблокированы. При вводе данных регистрации признаком окончания ввода является нажатие клавиши <Enter>.

Форма с компонентами, реализующими регистрацию, а также результаты ее выполнения показаны на рис. 11.85 и 11.86. Текст приложения приводится в листинге 11.22.

Рис. 11.85. Регистрация пользователя в приложении. Часть 1

Рис. 11.86. Регистрация пользователя в приложении. Часть 2

Листинг 11.22

```
#pragma once

namespace TabControl3_User_Reg2011 {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;

public ref class Form1 : public System::Windows::Forms::Form
{
public:
 Form1(void)
  {
   InitializeComponent();
    //
    //TODO: Add the constructor code here
    //
  }

protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
 {
  if (components)
   {
    delete components;
   }
  }
private: System::Windows::Forms::TabControl^  tabControl1;
protected:
private: System::Windows::Forms::TabPage^  tabPage1;
private: System::Windows::Forms::Label^  label1;
private: System::Windows::Forms::TabPage^  tabPage2;
private: System::Windows::Forms::Button^  button1;
private: System::Windows::Forms::TextBox^  textBox2;
private: System::Windows::Forms::TextBox^  textBox1;
private: System::Windows::Forms::Label^  label3;
private: System::Windows::Forms::Label^  label2;
private: System::Windows::Forms::ListBox^  listBox1;
private: System::Windows::Forms::Button^  button2;
```

```
private:
 /// <summary>
 /// Required designer variable.

//Здесь пользовательская информация
int pos,pos1;//глобальные
       String ^w,^us,^pas;
       String ^key; //ключ открытия 1-й страницы


private: System::Windows::Forms::Label^  label4;

      /// </summary>
System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support — do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
 {
     this->tabControl1 = (gcnew System::Windows::Forms::TabControl());
     this->tabPage1 = (gcnew System::Windows::Forms::TabPage());
     this->listBox1 = (gcnew System::Windows::Forms::ListBox());
     this->button2 = (gcnew System::Windows::Forms::Button());
     this->button1 = (gcnew System::Windows::Forms::Button());
     this->textBox2 = (gcnew System::Windows::Forms::TextBox());
     this->textBox1 = (gcnew System::Windows::Forms::TextBox());
     this->label3 = (gcnew System::Windows::Forms::Label());
     this->label2 = (gcnew System::Windows::Forms::Label());
     this->label1 = (gcnew System::Windows::Forms::Label());
     this->tabPage2 = (gcnew System::Windows::Forms::TabPage());
     this->label4 = (gcnew System::Windows::Forms::Label());
     this->tabControl1->SuspendLayout();
     this->tabPage1->SuspendLayout();
     this->tabPage2->SuspendLayout();
     this->SuspendLayout();
     //
     // tabControl1
     //
     this->tabControl1->Controls->Add(this->tabPage1);
     this->tabControl1->Controls->Add(this->tabPage2);
     this->tabControl1->Location = System::Drawing::Point(16, 15);
     this->tabControl1->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
     this->tabControl1->Name = L"tabControl1";
     this->tabControl1->SelectedIndex = 0;
```

```
    this->tabControl1->Size = System::Drawing::Size(537, 337);
    this->tabControl1->TabIndex = 0;
    this->tabControl1->SelectedIndexChanged += gcnew
System::EventHandler(this, &Form1::tabControl1_SelectedIndexChanged);
    //
    // tabPage1
    //
    this->tabPage1->Controls->Add(this->listBox1);
    this->tabPage1->Controls->Add(this->button2);
    this->tabPage1->Controls->Add(this->button1);
    this->tabPage1->Controls->Add(this->textBox2);
    this->tabPage1->Controls->Add(this->textBox1);
    this->tabPage1->Controls->Add(this->label3);
    this->tabPage1->Controls->Add(this->label2);
    this->tabPage1->Controls->Add(this->label1);
    this->tabPage1->Location = System::Drawing::Point(4, 25);
    this->tabPage1->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
    this->tabPage1->Name = L"tabPage1";
    this->tabPage1->Padding = System::Windows::Forms::Padding(4, 4, 4, 4);
    this->tabPage1->Size = System::Drawing::Size(529, 308);
    this->tabPage1->TabIndex = 0;
    this->tabPage1->Text = L"Регистрация пользователя";
    this->tabPage1->UseVisualStyleBackColor = true;
    //
    // listBox1
    //
    this->listBox1->FormattingEnabled = true;
    this->listBox1->ItemHeight = 16;
    this->listBox1->Items->AddRange(gcnew cli::array< System::Object^  >(3)
{L"us1/pas1/02.02.2012", L"us2/pas2/15.07.2012",
        L"us3/pas3/31.12.2012"});
    this->listBox1->Location = System::Drawing::Point(19, 186);
    this->listBox1->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
    this->listBox1->Name = L"listBox1";
    this->listBox1->Size = System::Drawing::Size(457, 116);
    this->listBox1->TabIndex = 7;
    this->listBox1->Visible = false;
    //
    // button2
    //
    this->button2->Enabled = false;
    this->button2->Location = System::Drawing::Point(441, 22);
    this->button2->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
    this->button2->Name = L"button2";
    this->button2->Size = System::Drawing::Size(36, 146);
    this->button2->TabIndex = 6;
    this->button2->Text = L"Администратор";
```

```
	this->button2->UseVisualStyleBackColor = true;
	this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
	//
	// button1
	//
	this->button1->Location = System::Drawing::Point(8, 22);
	this->button1->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
	this->button1->Name = L"button1";
	this->button1->Size = System::Drawing::Size(25, 146);
	this->button1->TabIndex = 5;
	this->button1->Text = L"Выход";
	this->button1->UseVisualStyleBackColor = true;
	this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
	//
	// textBox2
	//
	this->textBox2->Location = System::Drawing::Point(160, 118);
	this->textBox2->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
	this->textBox2->Name = L"textBox2";
	this->textBox2->PasswordChar = '*';
	this->textBox2->Size = System::Drawing::Size(132, 22);
	this->textBox2->TabIndex = 4;
	this->textBox2->KeyDown += gcnew
System::Windows::Forms::KeyEventHandler(this, &Form1::textBox2_KeyDown);
	//
	// textBox1
	//
	this->textBox1->Location = System::Drawing::Point(160, 73);
	this->textBox1->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
	this->textBox1->Name = L"textBox1";
	this->textBox1->PasswordChar = '*';
	this->textBox1->Size = System::Drawing::Size(132, 22);
	this->textBox1->TabIndex = 3;
	this->textBox1->KeyDown += gcnew
System::Windows::Forms::KeyEventHandler(this, &Form1::textBox1_KeyDown);
	//
	// label3
	//
	this->label3->Location = System::Drawing::Point(43, 122);
	this->label3->Margin = System::Windows::Forms::Padding(4, 0, 4, 0);
	this->label3->Name = L"label3";
	this->label3->Size = System::Drawing::Size(65, 28);
	this->label3->TabIndex = 2;
	this->label3->Text = L"Пароль";
	//
	// label2
	//
```

```
    this->label2->Location = System::Drawing::Point(43, 73);
    this->label2->Margin = System::Windows::Forms::Padding(4, 0, 4, 0);
    this->label2->Name = L"label2";
    this->label2->Size = System::Drawing::Size(109, 49);
    this->label2->TabIndex = 1;
    this->label2->Text = L"Имя пользователя";
    //
    // label1
    //
    this->label1->Font = (gcnew System::Drawing::Font(L"Microsoft Sans Serif",
15.75F, System::Drawing::FontStyle::Bold, System::Drawing::GraphicsUnit::Point,
        static_cast<System::Byte>(204)));
    this->label1->Location = System::Drawing::Point(47, 22);
    this->label1->Margin = System::Windows::Forms::Padding(4, 0, 4, 0);
    this->label1->Name = L"label1";
    this->label1->Size = System::Drawing::Size(411, 28);
    this->label1->TabIndex = 0;
    this->label1->Text = L"Т р у д о у с т р о й с т в о";
    //
    // tabPage2
    //
    this->tabPage2->Controls->Add(this->label4);
    this->tabPage2->Location = System::Drawing::Point(4, 25);
    this->tabPage2->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
    this->tabPage2->Name = L"tabPage2";
    this->tabPage2->Padding = System::Windows::Forms::Padding(4, 4, 4, 4);
    this->tabPage2->Size = System::Drawing::Size(529, 308);
    this->tabPage2->TabIndex = 1;
    this->tabPage2->Text = L"Приложение";
    this->tabPage2->UseVisualStyleBackColor = true;
    //
    // label4
    //
    this->label4->Location = System::Drawing::Point(155, 94);
    this->label4->Margin = System::Windows::Forms::Padding(4, 0, 4, 0);
    this->label4->Name = L"label4";
    this->label4->Size = System::Drawing::Size(197, 47);
    this->label4->TabIndex = 0;
    this->label4->Text = L"Здесь размещаются компоненты приложения";
    //
    // Form1
    //
    this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
    this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
    this->ClientSize = System::Drawing::Size(548, 348);
    this->Controls->Add(this->tabControl1);
    this->Margin = System::Windows::Forms::Padding(4, 4, 4, 4);
```

```
      this->Name = L"Form1";
      this->Tag = L"";
      this->Text = L"Городское агентство по трудоустройству";
      this->Shown += gcnew System::EventHandler(this, &Form1::Form1_Shown);
      this->Resize += gcnew System::EventHandler(this, &Form1::Form1_Resize);
      this->tabControl1->ResumeLayout(false);
      this->tabPage1->ResumeLayout(false);
      this->tabPage1->PerformLayout();
      this->tabPage2->ResumeLayout(false);
      this->ResumeLayout(false);

}
#pragma endregion
private: System::Void button1_Click(System::Object^  sender,
  System::EventArgs^  e)
{
 this->Close();
}

private: System::Void textBox1_KeyDown(System::Object^
sender,System::Windows::Forms::KeyEventArgs^  e)
 {
  if(e->KeyCode == Keys::Enter)
   {
    //проверка UserName
    String ^s;
    int jj;
    s=this->textBox1->Text;
    jj=s->Compare(s,"admin");
    if(!jj) //это имя администратора
     {
      this->button2->Enabled=true; //открывается его кнопка
      this->button2->Focus();
      this->textBox2->Text="";
      return;
     }

    int j=0,i=0;
    for(i=0;i<this->listBox1->Items->Count; i++)
     {
      w=this->listBox1->Items[i]->ToString();
      int pos=w->IndexOf("/");
      us=w->Substring(0,pos);
      jj=s->Compare(s,us);
      if(jj) continue;
      else
       {
        j++;
```

```
        int pos1=pos+1; //
        break;
       }
     } //for
     if(!j)
      {
       MessageBox::Show("Ошибка в UserName", "Приложение
67",MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
       this->textBox1->Text="";
       this->textBox1->Focus();
       return;
      }
//Здесь имя пользователя найдено, надо идти на ввод пароля
    this->textBox1->Text="";
    this->textBox2->Text="";
    this->textBox2->Focus();
    return;
  } //if  (== Enter)
} //Обработчик

private: System::Void textBox2_KeyDown(System::Object^
sender,System::Windows::Forms::KeyEventArgs^  e)
 {
  //обработка пароля
   if(e->KeyCode == Keys::Enter)
    {
     String ^s=this->textBox2->Text;
     int j=0;
     int pos=w->IndexOf("/"); //Подгонка под начало поиска 1-го
                              //разделителя после найденного.
                              //1-й всегда найдется, т. к. он нашелся
                             //при поиске UserName
     int pos1=w->IndexOf("/",pos+1); //ищется 2-й разделитель
     if(!pos1)
      {
       MessageBox::Show("Ошибка в строке паролей", "Приложение
",MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
       this->textBox2->Text="";
       this->textBox2->Focus();
       return;
      }
     pas=w->Substring(pos+1, (pos1 — pos -1));
     int jj=s->Compare(s,pas);
     if(jj)
      {
       MessageBox::Show("Ошибка в Password", "Приложение
",MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
```

```
        this->textBox2->Text="";
        this->textBox2->Focus();
        return;
       }
//Пароль сравнился. Проверяем дату
     pos1=pos1+1;
     pas=w->Substring(pos1);
     DateTime din,dtek;
     dtek=dtek.Today;  //текущая дата
     din=din.Parse(pas);// =StrToDate(pas);  дата из пароля
     if(dtek > din)
      {
       key=pas;
         MessageBox::Show("Срок действия пароля истек", "Приложение
",MessageBoxButtons::OK,MessageBoxIcon::Asterisk);
        this->textBox2->Text="";
        this->textBox2->Focus();
        return;
       }
        return;
     this->textBox1->Text="";
     this->textBox2->Text="";
     this->tabControl1->SelectedIndex=1;
   }//if  Key==
}//обработчик
//-----------------------------------------------------------------
private: System::Void tabControl1_SelectedIndexChanged(System::Object^
sender,System::EventArgs^  e)
 {
  int i=key->Compare(key,pas);
  if(!i)
   {
    this->tabControl1->SelectedIndex =0;
   }
  }

private: System::Void Form1_Resize(System::Object^  sender,
  System::EventArgs^  e)
 {
  this->textBox1->Focus();  //установка фокуса ввода на UserName
 }

private: System::Void Form1_Shown(System::Object^  sender,System::EventArgs^ e)
 {
  this->textBox1->Focus();  //установка фокуса ввода на UserName
 }
```

```
private: System::Void button2_Click(System::Object^  sender,System::EventArgs^ e)
 {
  if(this->listBox1->Visible==false)
   this->listBox1->Show();
  else
   {
    this->listBox1->Hide();
    this->button2->Enabled=false;
    this->textBox1->Text="";
    this->textBox1->Focus();
   }
 }
}; //класс
}//namespace
```

Компонент *Timer*

Компонент находится в списке **All Windows Forms** палитры компонентов. Он задает счетчик времени.

Свойство `Enabled` управляет запуском и остановкой таймера.

Свойство `Interval` задает промежуток времени, через который возникает единственное его событие `Tick`. При разработке обработчика события следует учитывать, что новое событие не возникает, пока не выполнятся все команды обработчика. Как только все команды обработчика будут завершены, новое событие возникает не позднее, чем через интервал времени, заданный в свойстве `Interval`. Основными методами компонента являются `Start()` и `Stop()`, которые запускают и останавливают таймер.

`Timer` — это удобное средство для организации процессов, автоматически повторяющихся через равные интервалы времени. Например, вы хотите, чтобы на экране компьютера происходило движение различных окрашенных линий. Вставьте в обработчик события `Tick` формирование таких линий и запустите это приложение. Пока ваш компьютер будет включен (или пока вы не отключите таймер с помощью кнопки), его экран будет светиться разноцветными линиями (рис. 11.87). Текст обработчиков событий приводится в листинге 11.23. Здесь есть один момент, который надо учесть: в файл stdafx.h надо добавить строку `include <stdlib.h>`, чтобы программа узнавала функцию `rand()` — функцию получения случайного числа. Почему именно в этот файл, подсказывает компилятор. Файл stdafx.h можно открыть прямо из окна **Solution Explorer** (папка Source Files) через контекстное меню файла.

Рис. 11.87. Автоматическая разрисовка экрана и ход этого процесса

Листинг 11.23

```
#pragma once

namespace Timer2011 {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

    /// <summary>
    /// Summary for Form1
    /// </summary>
    public ref class Form1 : public System::Windows::Forms::Form
    {
    public:
        Form1(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
```

```
        /// Clean up any resources being used.
        /// </summary>
        ~Form1()
        {
            if (components)
            {
              delete components;
            }
        }
    private: System::Windows::Forms::Button^  button1;
    private: System::Windows::Forms::Timer^  timer1;
    private: System::Windows::Forms::ProgressBar^  progressBar1;
    private: System::ComponentModel::IContainer^  components;
    protected:

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>


#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support — do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->components = (gcnew System::ComponentModel::Container());
            this->button1 = (gcnew System::Windows::Forms::Button());
            this->timer1 = (gcnew System::Windows::Forms::Timer(this-
>components));
            this->progressBar1 = (gcnew
System::Windows::Forms::ProgressBar());
            this->SuspendLayout();
            //
            // button1
            //
            this->button1->Location = System::Drawing::Point(22, 13);
            this->button1->Name = L"button1";
            this->button1->Size = System::Drawing::Size(113, 43);
            this->button1->TabIndex = 0;
            this->button1->Text = L"Пуск/останов таймера";
            this->button1->UseVisualStyleBackColor = true;
            this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click_1);
```

```
            //
            // timer1
            //
            this->timer1->Tick += gcnew System::EventHandler(this,
&Form1::timer1_Tick);
            //
            // progressBar1
            //
            this->progressBar1->Location = System::Drawing::Point(22, 62);
            this->progressBar1->Name = L"progressBar1";
            this->progressBar1->Size = System::Drawing::Size(113, 23);
            this->progressBar1->TabIndex = 1;
            //
            // Form1
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(282, 255);
            this->Controls->Add(this->progressBar1);
            this->Controls->Add(this->button1);
            this->Name = L"Form1";
            this->Text = L"Form1";
            this->ResumeLayout(false);

        }
#pragma endregion
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^ e)
{
  Color ^col = gcnew Color();
  Pen ^pen = gcnew Pen(col->Black);
  //чтобы создать графический объект, надо получить ссылку на него
  //выполнив метод CreateGraphics() компонента (формы)
 Graphics ^im = this->CreateGraphics();

int x1,x2,y1,y2;

  x1=rand(); //функция получения случайного числа
  x2=rand();
  y1=rand();
  y2=rand();
  pen->Width=5; //ширина пера для рисования линии

  /*надо привести интервалы случайных чисел,
  чтобы они попадали в форму*/

 if(x1 > 200)
 x1=200-(x1%200);
```

```
 pen->Color=Color::FromArgb(x1);
 if(x2 > 200)
 x2=200-(x2%200);
 if(y1 > 200)
 y1=200-(y1%200);
 if(y2 > 200)
 y2=200-(y2%200);
  pen->Color=Color::FromArgb(x1,x2,y1,y2);
  im->DrawLine(pen,x1,y1,x2,y2);
  /*рисует линию между 2-мя точками (x1,y1)и (x2,y2)*/

  this->progressBar1->Value++;

 }

private: System::Void button1_Click_1(System::Object^  sender,
System::EventArgs^  e)
 {
   this->progressBar1->Value=0;

  //включение/отключение таймера

  if(!timer1->Enabled)
   timer1->Enabled=true;
  else
   timer1->Enabled=false;
 }
};
}
```

Компонент *ProgressBar*

Этот компонент мы использовали в предыдущем примере. Компонент находится в списке **All Windows Forms** палитры компонентов. Этот компонент создает индикатор некоторого процесса, благодаря чему можно наблюдать ход процесса во времени. Прямоугольный индикатор при достаточно длительном процессе постепенно заполняется символом-заполнителем слева направо, причем заполнение завершается с окончанием самого процесса. Это заполнение организовано с помощью свойств и методов компонента `ProgressBar`.

Свойства `Min` и `Max` задают интервал значений индикатора.

Свойство `Value` (его надо изменять самому) определяет текущую позицию индикатора внутри интервала `Min` — `Max`.

Свойство `Step` задает начало отсчета для величины `Value`.

Метод `PerformStep()` вызывает изменение свойства `Value` на величину 1. Если требуется задать большую величину приращения, следует сначала выполнить метод `Perform(n)`, который задаст шаг приращения, равный `n`.

Чтобы организовать работу компонента `ProgressBar` по отображению хода процесса, надо использовать компонент `timer`: включить счетчик времени до начала процесса (`Timer1->Enabled=true;`), установить значение свойства `Value` компонента в ноль (`ProgressBar1->Value=0;`), а в обработчике события `Tick` наращивать значение `Value` (`ProgressBar1->Value++;`).

После окончания контролируемого процесса надо выключить таймер и скрыть сам индикатор (`ProgressBar1->Visible=false;`).

Пример совместной работы `ProgressBar` с компонентом `Timer` показан в *разд. "Компонент Timer" этой главы*.

Компонент *OpenFileDialog*

Компонент находится в списке **All Windows Forms** палитры компонентов. Он предназначен для выбора файлов, выводит на экран стандартное окно Windows для выбора и открытия файлов.

Чтобы начать диалог по поиску файла, надо использовать метод `ShowDialog()`. Если установить свойство `Multiselect` в `true`, то можно выбрать группу файлов.

Можно воспользоваться свойством `ShowReadOnly`, которое дает возможность появиться галочке рядом с файлом, если он имеет тип "только для чтения".

Свойство `Filter` задает условие фильтрации файлов (чтобы выбирались только те, которые указаны в фильтре).

Компонент при добавлении в форму не появляется в форме, а помещается на специальный поддон, расположенный в нижней части окна дизайнера форм (как и многие другие компонеты: `ImageList`, `Timer` и др.). Кстати, если требуется открыть папку вместо файла, то следует воспользоваться классом `FolderBrowserDialog`.

Диалоговое окно для выбора файла появляется в режиме исполнения приложения в момент выполнения метода `ShowDialog()`. Когда пользователь в диалоговом окне нажимает на кнопку **Открыть** (или **Open** — это зависит от установленной у вас версии Windows), метод `ShowDialog()` возвращает значение `DialogResult`, которое сравнивается со значением такого же свойства у формы. А в форме мы его устанавливаем равным `OK`. Это значит, что если возвращенное значение тоже будет `OK`, то метод сработал без ошибок, и окно открылось. Теперь если нажать в окне на кнопку **Открыть**, то окно закроется, и имя выбранного файла поместится в свойство компонента `fileName`, откуда его можно в дальнейшем брать и использовать по назначению. На рис. 11.88 показан перечень свойств компонента `OpenFileDialog` и результат работы компонента по выбору текстового файла и выводу его содержимого в `ListBox`.

Рис. 11.88. Свойства `OpenFileDialog` и результат работы компонента по выбору текстового файла и вывода его содержимого в `ListBox`

Строка фильтра в свойстве `Filter` задается по правилам, видным из следующего примера:

```
Text files (*.txt)|*.txt|All files (*.*)|*.*
```

Так задаются фильтры для выборки текстовых файлов или всех файлов. Когда откроется диалоговое окно, то в его поле **Тип файлов** (если раскрыть выпадающий список) увидим все заданные типы, и именно те, которые начинаются с наименования группы.

Например, если задать, чтобы высветились все текстовые файлы и все файлы Word, то в этом случае строка фильтра будет такой:

```
Text Files (*.txt)|*.txt|Word Files (*.doc)
```

а в поле типов файлов будет две строки:

```
Text Files *.txt
Word Files *.doc
```

Далее можно выбирать любой тип и открывать соответствующий файл (названия типа выводятся в диалоговом окне для информации).

Допустим, мы хотим выбрать файлы изображений.

В этом случае зададим такой фильтр:

```
Image Files(*.BMP;*.JPG;*.GIF)|*.BMP;*.JPG;*.GIF|All files (*.*)|*.*
```

В листинге 11.24 приведен пример работы с компонентом `OpenFileDialog`.

Листинг 11.24

```
#pragma once

namespace OpenDialog2011 {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::IO;
using namespace System::Text;

public ref class Form1 : public System::Windows::Forms::Form
{
 public:
  Form1(void)
   {
    InitializeComponent();
     //
     //TODO: Add the constructor code here
           //
   }

protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
 ~Form1()
  {
   if (components)
    {
     delete components;
    }
         }
private: System::Windows::Forms::Button^  button1;
private: System::Windows::Forms::OpenFileDialog^
        openFileDialog1;
```

```
private: System::Windows::Forms::Button^  button2;
protected:
private: System::Windows::Forms::ListBox^  listBox1;

private:
/// <summary>
/// Required designer variable.

       int fix;
       StreamWriter ^sw;

//=========================================================

//==============================================================
/// </summary>
System::ComponentModel::Container ^components;
#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support — do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
 {
  this->button1 = (gcnew System::Windows::Forms::Button());
  this->listBox1 = (gcnew System::Windows::Forms::ListBox());
  this->openFileDialog1 = (gcnew
    System::Windows::Forms::OpenFileDialog());
  this->button2 = (gcnew System::Windows::Forms::Button());
  this->SuspendLayout();
//
// button1
//
this->button1->Location = System::Drawing::Point(12, 1);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(156, 23);
this->button1->TabIndex = 0;
this->button1->Text = L"Выбрать текстовый файл";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this,
  &Form1::button1_Click);
//
// listBox1
//
this->listBox1->FormattingEnabled = true;
this->listBox1->Location = System::Drawing::Point(12, 30);
this->listBox1->Name = L"listBox1";
```

```
this->listBox1->Size = System::Drawing::Size(156, 30);
this->listBox1->TabIndex = 1;
//
// openFileDialog1
//
this->openFileDialog1->FileName = L"openFileDialog1";
this->openFileDialog1->Filter = L"Text files (*.txt)|*.txt|Word  files
(*.doc)|*.doc";
//
// button2
//
this->button2->Location = System::Drawing::Point(180, 1);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(25, 73);
this->button2->TabIndex = 2;
this->button2->Text = L"Выход";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this,
  &Form1::button2_Click);
//
// Form1
//
this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
this->ClientSize = System::Drawing::Size(218, 74);
this->Controls->Add(this->button2);
this->Controls->Add(this->listBox1);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
}
#pragma endregion
private: System::Void button1_Click(System::Object^  sender,
  System::EventArgs^  e)
  {
   /*Значения OK,Cancel... достаются как
     System::Windows::Forms::DialogResult::
     (мы разрабатываем приложения типа Windows Forms и данные берем
     из класса Forms)*/

    Stream^ s;

   //в Form1 задано свойство DialogResult=OK и с ним будет
   //сравниваться ShowDialog()

   this->DialogResult= System::Windows::Forms::DialogResult::OK;
```

```
  //ShowDialog() возвращает имя файла в FileName

  if(this->openFileDialog1->ShowDialog()==
    System::Windows::Forms::DialogResult::OK)

    /*ShowDialog() возвращает переменную типа DialogResult  */
   {
    if( (s = this->openFileDialog1->OpenFile()) != nullptr )
     {
//здесь идут операторы чтения файла из потока
      String ^path=this->openFileDialog1->FileName;

  //участок чтения содержимого открытого файла в ListBox
  //файл должен быть записан WordPad'ом как текстовый в кодировке Юникод

   if (  !File::Exists( path ) )
    {
      // Create a file to write to
      sw = File::CreateText( path ); // StreamWriter^
      try
      {
         sw->WriteLine( "Hello" );
         sw->WriteLine( "And" );
         sw->WriteLine( "Welcome" );
      }
      finally
      {
         if ( sw )
                  delete (IDisposable^)(sw);
      }
   }

   // Open the file to read from
   TextReader ^ sr = File::OpenText( path );
   try
    {
      String^ s = "";

      while ( s = sr->ReadLine() )
      {
              this->listBox1->Items->Add(s);
      }
   }
   finally
   {
    if ( sr )
      delete (IDisposable^)(sr);
   }
  }
```

```
        //конец участка чтения файла

  }
   else
     MessageBox::Show ("Ошибка открытия файла");
  }

private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
  this->Close();
 }
};
}
```

Перед компиляцией приложения надо включить в файл stdafx.cpp следующие дополнительные строки:

```
#include <io.h>
#include <stdio.h>
#include <fcntl.h>
#include <share.h>
```

При подготовке текстового файла с помощью программы Блокнот текст следует сохранять в кодировке UTF-8.

Компонент *SaveFileDialog*

Компонент находится в списке **All Windows Forms** палитры компонентов. С помощью этого компонента можно сохранять файл в нужном месте файловой структуры так же, как это делается в Windows. Но файл сам по себе не сохраняется, т. к. компонент дает только путь к будущему месту расположения файла. На пользователе лежит обязанность самому написать участок программы для сохранения файла. Как и в `OpenFileDialog`, в этом компоненте применяется метод `ShowDialog()` для открытия диалогового окна в режиме исполнения. Файл можно открыть в режиме чтения/записи, используя метод `OpenFile()`.

Компонент при добавлении его в форму появляется не в самой форме, а на поддоне, расположенном в нижней части окна дизайнера форм (рис. 11.89).

Перечень свойств `SaveFileDialog` приведен также на рис. 11.89.

Почти все свойства компонента совпадают со свойствами `OpenFileDialog`. Отметим только свойство `OverwritePrompt`, которое (если оно установлено в `true`), вызывает появление диалогового окна **Save As**.

Когда пользователь выбирает имя файла и нажимает на кнопку **Save** в диалоговом окне, метод `ShowDialog()` заносит в свойство `FileName` компонента имя файла и путь к нему. Никакой перезаписи файла при этом не происходит. Отсюда следует, что

для записи файла в необходимое место файловой структуры нужно применять методы сохранения файла. Приведем пример приложения, которое читает текстовый файл, подготовленный с помощью WordPad в Юникоде и переписывает его в другое место под именем, которое мы выбираем в диалоговом окне. Затем файл с целью проверки читается в другое окно. Вид подготовленного файла, формы приложения в режиме дизайна и исполнения показан на рис. 11.90. Текст приложения приводится в листинге 11.25.

Рис. 11.89. Свойства компонента `SaveFileDialog`

Рис. 11.90. Пример одновременного применения `OpenFileDialog` и `SaveFileDialog`

Листинг 11.25

```
#pragma once
namespace  My71SaveFileDialog {

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
using namespace System::IO;  //для ввода/вывода
using namespace System::Text;
public ref class Form1 : public System::Windows::Forms::Form
 {
  public:
   Form1(void)
    {
     InitializeComponent();
     //
     //TODO: Add the constructor code here
     //
    }

protected:
/// <summary>
/// Clean up any resources being used.
/// </summary>
~Form1()
 {
  if (components)
   {
    delete components;
   }
  }
private: System::Windows::Forms::SaveFileDialog^  saveFileDialog1;
private: System::Windows::Forms::OpenFileDialog^  openFileDialog1;
private: System::Windows::Forms::Button^  button1;
private: System::Windows::Forms::ListBox^  listBox1;
private: System::Windows::Forms::ListBox^  listBox2;
private: System::Windows::Forms::Button^  button2;
protected:
 private:
 /// <summary>
 /// Required designer variable.
//=================================================================
void LoadFromFile(String ^File, ListBox ^lb)
```

```
{
 /*
  Этот метод открывает текстовый файл, читает все его строки в
 строку String ^ и закрывает файл. Строки должны отделяться друг от
 друга разделителем "/"
*/
 String ^d, ^b = File::ReadAllText(File);
/*(надо будет выделять по разделителю "/")*/
 lb->Items->Clear();
 /*Разборка длинной строки на строки, отделенные друг от друга
   разделителями*/
 while(b->Length > 0)
  {
   int i=b->IndexOf("/");  //поиск 1-го вхождения подстроки в строку
   d=b->Substring(0,i);
   lb->Items->Add(d);
   b=b->Substring(i+1,b->Length — d->Length -1);
  }
}
//--------------------------------------------------
void SaveToFile(String ^File,ListBox ^lb)
 {
  String ^a,^b;
  int j=lb->Items->Count;
  File::Delete(File);
  for(int i=0; i < j; i++)
   {
/*Чтение строк ChekedListBox в a и формирование длинной строки в b*/
    a=lb->Items[i]->ToString();
    b+=a->Concat(a,"/");  //добавка разделителя строк
/*Этот метод открывает файл, добавляет к нему строку типа String ^,
  закрывает файл. Если файл не существует, он создается */
  } //for
 File::AppendAllText(File, b);
}
//==============================================================
/// </summary>
System::ComponentModel::Container ^components;
#pragma region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support — do not modify
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
 {
  this->saveFileDialog1 = (gcnew
    System::Windows::Forms::SaveFileDialog());
```

```
  this->openFileDialog1 = (gcnew
    System::Windows::Forms::OpenFileDialog());
  this->button1 = (gcnew System::Windows::Forms::Button());
  this->button2 = (gcnew System::Windows::Forms::Button());
  this->listBox1 = (gcnew System::Windows::Forms::ListBox());
  this->listBox2 = (gcnew System::Windows::Forms::ListBox());
  this->SuspendLayout();
//
// saveFileDialog1
//
this->saveFileDialog1->Filter = L"Text files (*.*)|*.txt";
//
// openFileDialog1
//
this->openFileDialog1->FileName = L"openFileDialog1";
this->openFileDialog1->Filter = L"Text files (*.*)|*.txt";
//
// button1
//
this->button1->Location = System::Drawing::Point(0, 9);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(90, 40);
this->button1->TabIndex = 0;
this->button1->Text = L"Копировать файл";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this,
  &Form1::button1_Click);
//
// button2
//
this->button2->Location = System::Drawing::Point(173, 9);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(33, 43);
this->button2->TabIndex = 1;
this->button2->Text = L"Выход";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this,
  &Form1::button2_Click);
//
// listBox1
//
this->listBox1->FormattingEnabled = true;
this->listBox1->Location = System::Drawing::Point(0, 58);
this->listBox1->Name = L"listBox1";
this->listBox1->Size = System::Drawing::Size(206, 43);
this->listBox1->TabIndex = 2;
```

```
//
// listBox2
//
this->listBox2->FormattingEnabled = true;
this->listBox2->Location = System::Drawing::Point(0, 109);
this->listBox2->Name = L"listBox2";
this->listBox2->Size = System::Drawing::Size(206, 43);
this->listBox2->TabIndex = 3;
//
// Form1
//
this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
this->ClientSize = System::Drawing::Size(207, 164);
this->Controls->Add(this->listBox2);
this->Controls->Add(this->listBox1);
this->Controls->Add(this->button2);
this->Controls->Add(this->button1);
this->Name = L"Form1";
this->Text = L"Form1";
this->ResumeLayout(false);
}
#pragma endregion
private: System::Void button2_Click(System::Object^  sender,
  System::EventArgs^  e)
 {
  this->Close();
 }
private: System::Void button1_Click(System::Object^  sender,
  System::EventArgs^  e)
 {
  //ShowDialog() возвращает имя файла в FileName
  this->openFileDialog1->ShowDialog();
  String ^a=this->openFileDialog1->FileName;
  LoadFromFile(a, this->listBox1);  //Загрузка файла в ListBox1

  /*Выбор имени и пути для выходного файла
   с помощью SaveFileDialog/ Имя файла будет в FileName*/
  this->saveFileDialog1->ShowDialog();
  a=this->saveFileDialog1->FileName;
  SaveToFile(a,this->listBox1);
  //сохранение файла из Listbox1
  //чтение сохраненного файла в ListBox2
  this->openFileDialog1->ShowDialog();
  a=this->openFileDialog1->FileName;
  LoadFromFile(a, this->listBox2);
 }
};
}
```

Компонент *ColorDialog*

Компонент находится в списке **All Windows Forms** палитры компонентов. Он делает возможным выбор цвета в диалоговом окне, работает точно так же, как и остальные диалоговые компоненты: выполняется метод `ShowDialog()`, открывается палитра цветов, из которой нужно выбрать необходимый цвет. Значение цвета помещается в свойство `Color` компонента, после чего цвет может использоваться в дальнейшем. Пример работы компонента показан на рис. 11.91.

Рис. 11.91. Пример использования компонента `ColorDialog`

Текст обработчика кнопки, изменяющего цвет шрифта формы, приводится в листинге 11.26.

Листинг 11.26

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
{
 this->colorDialog1->ShowDialog();
 this->ForeColor = this->colorDialog1->Color;
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
{
 this->Close();
}
```

Компонент *FontDialog*

Компонент находится в списке **Dialogs** палитры компонентов, обеспечивает выбор шрифта и его атрибутов (стиля, размера, цвета и т. п.) в диалоговом режиме.

После выбора нужного шрифта его название попадает в свойство компонента `Font`. Компонент работает, как и предыдущий. Вид обработчика кнопки, заставляющей изменять шрифт, показан в листинге 11.27.

Листинг 11.27

```
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
{
 this->Close();
}
private: System::Void button1_Click(System::Object^  sender,
 System::EventArgs^  e)
 {
  this->fontDialog1->ShowDialog();
  this->Font = this->fontDialog1->Font;
 }
```

Компонент *PrintDialog*

Компонент находится в списке **Printing** палитры компонентов. С помощью этого компонента можно открыть диалоговое окно настройки печати (рис. 11.92), где надо выбрать принтер, а также страницы, которые следует печатать, и установить некоторые свойства печати.

Рис. 11.92. Настройка печати с помощью компонента `PrintDialog`

Обработчик кнопки, показанной на рис. 11.92, имеет вид:

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
  this->printDialog1->ShowDialog();
 }
```

Компонент *ToolStrip*

Компонент находится в списке **All Windows Forms** палитры компонентов. Он позволяет создавать линейки инструментов с различными элементами пользовательского интерфейса. Например, вам требуется работать с такими элементами, как кнопка, метка, `TextBox`, `ProgressBar` и др. Все это можно собрать на одной линейке `ToolStrip` и иметь все перед глазами, как будто это элементы некоторого меню. Вид компонента в форме показан на рис. 11.93.

Если внимательно посмотреть, то работа с этим компонентом похожа на работу с компонентом `MenuStrip`, рассмотренным в начале этой главы.

Используя этот компонент, можно:

- создавать пользовательские линейки инструментов, применяя передовые интерфейсные и компоновочные возможности, такие как: причаливание, кнопки с текстом и изображением, выпадающие кнопки и другие элементы, реорганизуемые в процессе исполнения приложения;
- перетаскивать элементы с одной линейки инструментов на другую или внутри одной и той же линейки;
- создавать выпадающие элементы;
- подключать к линейке другие компоненты и передавать им функциональные возможности данной линейки;
- расширять функциональность и модифицировать появление и поведение линейки;
- создавать линейку, схожую с линейкой инструментов широко известного продукта Microsoft Word.

Рис. 11.93. Вид `ToolStrip` в форме

Некоторые свойства *ToolStrip*

Отметим следующие свойства: `Items` и `Dock`.

`Items` задает набор элементов, из которых можно формировать линейку инструментов. Набором можно пользоваться с помощью диалогового окна, открывающегося кнопкой с многоточием (в поле свойства `Items`) (рис. 11.94).

Из рисунка видно, что в левой части диалогового окна располагаются: меню выбора элементов для помещения их в линейку инструментов, окно для помещения в него выбранных из меню элементов, кнопки работы с выбранными элементами, позволяющие переупорядочивать элементы или удалять их из списка выбранных. Порядок, в котором элементы расположены в окне после их выбора, сохранится

Рис. 11.94. Задание элементов линейки инструментов

в сформированной затем линейке инструментов при расположении их слева направо: первый элемент из списка в окне поместится в линейку самым левым, второй — правее первого и т. д.

В правой части диалогового окна расположено окно свойств выбранного элемента. Для настройки свойств на элементе левого окна надо щелкнуть мышью (отметить его), тогда в правом окне появится набор свойств отмеченного элемента. Мы не станем рассматривать этот набор свойств, т. к. они уже описаны ранее — это метки, ссылочные метки, кнопки и т. д.

Следует отметить, что линейка инструментов из выбранных элементов формируется наподобие соответствующей линейки для продукта Microsoft Word. Например, если выбранные элементы не помещаются на линейке из-за ограниченных размеров формы, то такие элементы сворачиваются в специальный список, на который начинает указывать автоматически созданная кнопка. Но это произойдет только тогда, когда свойство компонента `CanOwerflow` (в переводе на русский язык "может переполняться") установлено в `true` (т. е. линейка может содержать большее количество элементов, чем занимаемое ею пространство в форме).

`Dock` задает, к какой стороне родительского контейнера причалит линейка инструментов. Это свойство рассматривалось в компоненте `Button`. И в данном свойстве требуется раскрыть выпадающий список. При этом откроется схема, состоящая из прямоугольников, имитирующих стороны формы и ее центр. Выбор того или иного прямоугольника определяет сторону причаливания линейки инструментов (без причаливания, слева, справа, снизу, сверху, в центре). Если значение свойства установлено в `None` (без причаливания), то на этапе проектирования линейку можно свободно перемещать в рамках формы при условии, что свойство `Locked` установлено в `false` (при значении `true` блокируется перемещение компонента, а в верхнем левом углу линейки появляется пиктограмма замка).

Использование *ToolStrip*

Содержимое линейки инструментов определяет разработчик приложения. Пользование элементами линейки зависит от типа самого элемента. Если, например, элемент — обычная кнопка (`Button`), то для ее использования надо создать обработчик события `Click`. Для этого необходимо открыть контекстное меню кнопки (щелкнуть на ней правой кнопкой мыши) и выполнить команду **Properties** (свойства). В открывшемся окне нужно щелкнуть на вкладке **Events** (события), а затем в открывшемся перечне событий дважды щелкнуть на событии `Click`. При этом создастся обработчик события, в который и следует записать команды реакции приложения на нажатие кнопки (это обычный знакомый путь обработки кнопки).

Если элементом линейки является кнопка выпадающего меню (`DropDownButton`), то с ней надо поступать как с компонентом, задающим меню. Можно создать опции прямо на линейке с использованием окна **Properties**, а можно открыть свойство линейки `Items`, отметить мышью обрабатываемую кнопку в левой части диалогового окна, а в правой использовать свойства кнопки для формирования элементов ее меню (свойство `DropDownItems`). Точно так же следует поступать и с элементом линейки `ComboBox`: через диалоговое окно свойства линейки `Items` открыть список

свойств элемента `ComboBox` и среди них выбрать свойство `Items`. Если затем нажать на кнопку с многоточием в поле этого свойства, то откроется окно редактора элементов, в котором можно задать перечень элементов `ComboBox`.

Если элементом линейки выступает метка-ссылка (для этого свойство метки `IsLink` должно быть равно `true`), то надо обработать ее событие `Click`, записав в его обработчике следующие строки:

```
String ^str =this->toolStripTextBox1->Text;
System::Diagnostics::Process::Start(str);
```

Из приведенного текста видно, что в форму надо поместить еще компонент `TextBox`, в котором и будет задаваться интернет-адрес. После задания этого адреса следует щелкнуть на метке-ссылке (предварительно откомпилировав приложение). Результат показан на рис. 11.95.

Примечание

В тексте главы приводились листинги приложений — копии кода конкретного приложения, взятого из вкладки **Form1.h**. По таким листингам можно восстановить само приложение, т. е. его вид в режиме дизайна (**Form1.h (Design)**). Делается это следующим образом. Создается новое приложение с пустой формой, открывается его вкладка **Form1.h**, весь первоначальный код из нее удаляется и вместо него вставляется код из листинга. После этого открывается вкладка **Form1.h (Design)**). Среда программирования сформирует форму и ее содержимое в соответствии с данными листинга.

Рис. 11.95. Работа элементов `TextBox` и `Label` на линейке `ToolStrip`

C/C++ Microsoft Visual

ГЛАВА 12

Работа с наборами данных. Общие сведения о базах данных

Что такое база данных? Это специальным образом организованное пространство памяти для хранения определенных групп данных, снабженное специальным программным обеспечением для поддержания их (этих данных) в активном состоянии, а также для создания возможности пользователю отсылать свою информацию в это пространство и получать из него любую другую информацию. Под группами данных мы понимаем следующие элементы:

- так называемые прямоугольные реляционные (т. е. связанные между собой некоторыми отношениями) таблицы данных;
- элементы, называемые вьюерами (от англ. *View* — осмотр, обозрение), с помощью которых можно просматривать таблицы баз данных;
- элементы, называемые хранимыми процедурами. Это специальные программы, расположенные на серверах, целью которых является обработка информации большого объема и пересылка небольшого по объему результата компьютеру-клиенту, запросившему у сервера такую информацию.

  Цель создания таких элементов — снизить до минимума временные затраты на перекачку информации от сервера клиенту (все, что можно сделать на сервере, а не у клиента, должно делаться на сервере);
- элементы, называемые триггерами (от англ. *Trigger* — запускать, инициировать). Триггер — это отдельная подпрограмма, связанная с таблицей или вьюером, которая автоматически запускается, чтобы выполнить некоторое действие, когда модифицируется таблица или вьюер на уровне строки (а именно: когда строка вставляется, удаляется или модифицируется).

  Триггер никогда не вызывается напрямую, а только при модификации таблиц или вьюеров. В чем польза от применения триггеров? С их помощью автоматически осуществляется принудительная обработка ограничений, что позволяет пользователю быть уверенным в правильности ввода достоверных данных, происходит автоматическая регистрация изменений в таблицах. Приложение может регистрировать с помощью триггера изменения в таблице, а также автоматически регистрировать изменения в базе данных с помощью обработчиков событий в триггерах.

Прямоугольные таблицы данных состоят из строк и столбцов данных.

Столбцы (их еще называют полями) являются определяющими в таких таблицах — они 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. Это такие данные, как фамилия, имя, отчество, год рождения, должность, категория работника, оклад, дата поступления на работу и т. д.

Строки таблицы (их еще называют записями) отражают данные по конкретному работнику.

Базы данных бывают *локальные* (когда они расположены на вашем же компьютере) и *удаленные* (когда они расположены на других компьютерах, соединенных одной сетью с вашим). Компьютеры, на которых располагаются такие базы данных, называются *серверами*, а ваш компьютер по отношению к таким базам выступает в роли их *клиента*.

Типы баз данных бывают самые разные, потому что они (базы данных) поддерживают разные структуры таблиц, которые, собственно, и содержат данные. Кроме того, каждая база имеет свой собственный механизм ведения базы данных (т. е. поддержки ее в активном состоянии). Примерами локальных баз данных являются базы данных типа MS Access, а примерами удаленных — InterBase, Informix, SyBase, Oracle и др.

Для создания баз данных и организации их взаимодействия с приложениями пользователя (клиентскими приложениями) существуют различные механизмы. Одним из таких механизмов является MS SQL Server, который поддерживает (т. е. обеспечивает своими средствами их создание и активное состояние) все вышеперечисленные элементы баз данных.

Прежде чем пользоваться базами данных, надо понять, как они создаются и как работают.

Проектирование баз данных

С помощью базы данных описывают не только структуру информации реальных предприятий и организаций, но и обеспечивают ее обработку, символически представляя реальные объекты своими специальными структурами: таблицами, вьюерами, хранимыми процедурами и другими элементами. Поскольку информация в базе данных организована и хранится в виде определенных объектов, то к таким объектам может быть организован доступ с помощью приложений, создаваемых в различных программных средах и пользовательских интерфейсах, формируемых с помощью средств базы данных (БД).

Одним из главных факторов в создании базы данных является ее правильное проектирование. Обычно перед помещением в базу данных информация представляется в виде прямоугольных таблиц, состоящих из строк и столбцов (колонок). На-

пример, процесс начисления и выдачи зарплаты работникам предприятия можно отобразить в виде такой прямоугольной таблицы, в которой в качестве столбцов будут характеристики работника, виды начислений и удержаний, даты оплаты и т. п., а в качестве строк — конкретные работники.

Логическое проектирование БД — это интерактивный процесс, состоящий из расчленения больших структур информации на мелкие, элементарные данные. Этот процесс носит название *нормализации* (в философском смысле — анализ). Цель нормализации состоит в определении природных связей между данными в будущей базе данных. Это делается путем расщепления конкретной таблицы на более мелкие, простые таблицы (с меньшим количеством столбцов). После такого расщепления лучше видно, какие элементы или группы элементов можно объединить в отдельные таблицы и каковы на самом деле связи между построенными таблицами (этот обратный процесс в философии носит название синтеза).

Модель базы данных

При проектировании базы данных важно представлять различие между описанием базы данных и самой базой данных. Описание БД называют моделью данных. Она формируется на этапе проектирования БД. *Модель* — это шаблон для создания таблиц (он описывает логическую структуру БД, включая данные и их суть, типы данных, пользовательские операции, связи между объектами-данными, ограничения на целостность данных). Логические структуры, описывающие базу данных, не подвержены воздействиям со стороны соответствующих физических структур, хранимых в БД. Этот факт обеспечивает межплатформенную переносимость данных.

Реляционную базу данных (т. е. базу, в которой определены некоторые отношения (relations) между данными) нетрудно переносить на различные технические платформы, потому что механизм доступа к ней определен моделью БД и остается неизменным независимо от места ее хранения.

Если пользоваться строительной терминологией, то логическая структура БД — это проект дома, а сама БД — дом, построенный по этому проекту. Поэтому, имея проект "дома", сам "дом" можно построить и на платформе Windows, и на платформе UNIX, и т. д.

При разработке БД ее нужно настраивать на различные информационные потребности пользователя. Это делается с помощью вьюеров (viewers) — программ просмотра данных, которые выводят подмножества данных по заказам конкретных пользователей или их групп. Вьюеры могут использоваться, чтобы спрятать необходимые данные или отфильтровать их.

Структура проектирования базы данных

При проектировании БД выполняются следующие конкретные действия:

- определение требований к информации (объем, частота использования, безопасность и т. д.) путем опроса будущих пользователей;

- анализ реальных объектов, которые требуется смоделировать в БД. Перевод управления объектами в управление элементами БД и формирование списка элементов БД (синтез БД);
- решение вопросов идентификации элементов в БД (т. е. как и по каким признакам находить ту или иную информацию);
- разработка набора правил доступа к каждой таблице (т. е. того, как каждая таблица станет наполняться и модифицироваться);
- установка отношений между объектами (таблицами и колонками);
- планирование безопасности БД.

Идентификация сущностей и атрибутов

Основываясь на требованиях заказчика (пользователя), собранных вами, можно определить объекты, которые должны быть в базе данных (т. е. сущности и их атрибуты).

Сущность — некий объект, который требуется описать в базе данных (это может быть автомобиль, работник, компания, должность, проект и многое другое).

Каждая сущность имеет свойства, называемые *атрибутами*.

Допустим, что проектируется база данных, которая будет содержать сведения о работниках компании, об иерархии подразделений, о текущих проектах, о клиентах (покупателях) и продажах. В табл. 12.1 приведен пример, показывающий, как создать список сущностей и атрибутов для организации данных.

Таблица 12.1. *Перечень объектов для создаваемой базы данных*

Сущность	Атрибуты сущности
EMPLOYEE (Работники)	**Emp_Num** (Табельный номер)
	Last_Name (Фамилия)
	First_Name (Имя)
	Dept_Num (Код подразделения)
	Job_Code (Код должности)
	Phone_Num (Телефон)
	Salary (Зарплата)
DEPARTMENT (Компания)	**Dept_Num** (Код подразделения)
	Dept_Name (Наименование)
	Dept_Head_Name (Имя руководителя)
	Budget (Бюджет)
	Location (Месторасположение)
	Phone_Num (Телефон)

Таблица 12.1 *(окончание)*

Сущность	Атрибуты сущности
PROJECT (Проекты)	**Project_ID** (Код проекта)
	Project_Name (Название проекта)
	Project_Description (Описание проекта)
	Team_Leader (Руководитель)
	Product (Изделие)

Составляя таким способом списки сущностей и соответствующих атрибутов, удается удалить избыточные записи.

Могут ли быть таблицами сущности из составленного списка? Могут ли быть перемещены колонки из одной группы в другую? Появляется ли один и тот же атрибут в некоторых сущностях? На такие вопросы надо дать ответ.

Каждый атрибут может появляться лишь однажды, и вы должны определить, какая сущность является первичным собственником этого атрибута. Например, атрибут **Dept_Head_Name** (Имя руководителя) должен быть удален, т. к. **Last_Name** (Фамилия), **First_Name** (Имя), а также **Emp_Num** (Табельный номер) уже существуют в сущности **EMPLOYEE** (Работники). А для доступа к руководителю подразделения можно воспользоваться табельным номером работника (руководитель в качестве работника имеет свой табельный номер).

Посмотрим теперь, как отобразить составленный список в объекты базы данных: сущности — в таблицы, а атрибуты — в колонки.

Проектирование таблиц

В реляционной базе данных объектом, представляющим отдельную сущность, является таблица, которая является двумерной матрицей строк и столбцов. Каждый столбец матрицы представляет один атрибут. Каждая строка представляет специфический экземпляр сущности.

Например, мы проектируем таблицу **Покупатели**. В таком случае таблица **Покупатели** — это сущность, а код покупателя, его адрес, контактный телефон и т. д. являются атрибутами данной сущности. Специфические экземпляры сущности — это 1-й покупатель, 2-й покупатель и т. д. В некоторых СУБД (системах управления базами данных), например MS Access, поставляемой с русским вариантом Windows, таблицы и их атрибуты можно задавать по-русски. В большинстве же СУБД — по-английски.

После определения сущностей и атрибутов создается модель, которая служит логическим проектом структуры вашей базы данных. Модель отображает сущности и атрибуты в таблицу, в которой колонки являются детальным описанием сущностей БД.

В табл. 12.2 показано, как сущность **EMPLOYEE** была преобразована из списка в таблицу.

Таблица 12.2. *Таблица для сущности* ***EMPLOYEE***

Emp_Num	Last_Name	First_Name	Dept_Num	Job_Code	Phone_Num	Salary
24	Smith	John	100	Eng	4968	64000
48	Carter	Catherine	900	Sales	4967	72500
36	Smith	Jane	600	Admin	4800	37500

Каждая строка таблицы представляет одного работника.

Колонки **Emp_Num**, **Last_Name**, **First_Name**, **Dept Num**, **Job_Code**, **Phone_Num**, **Salary** представляют атрибуты работника.

Когда таблица наполнена данными и строка добавляется к таблице, то каждый элемент строки — это ячейка, в которой сохраняется некая информация (например, "Smith").

Вся строка информации по одному работнику называется просто записью. Колонки таблицы, полученные из атрибутов сущности, называют полями таблицы.

Определение неповторяющихся атрибутов

Одной из задач проектирования базы данных является обеспечение уникального определения каждого экземпляра сущности, т. е. необходимо сделать так, чтобы система могла извлечь из таблицы единственную строку. Иными словами, должна быть однозначная идентификация строки таблицы.

Одну строку от другой отличают по значениям так называемого *первичного ключа*.

Значения, введенные в колонку первичного ключа или во множество колонок, составляющих первичный ключ (а он может состоять и из нескольких атрибутов), являются уникальными для каждой строки. Если вы попытаетесь ввести некоторое значение в первичный ключ, которое уже существует в другой строке той же колонки, то система контроля, предусмотренная в БД, прекратит операцию и выдаст ошибку.

Например, в приведенной таблице **EMPLOYEE** поле **Emp_Num** (Табельный номер) является уникальным атрибутом. Такой атрибут может использоваться как идентификатор каждого работника в базе данных (т. е. каждой строки таблицы). Следовательно, этот реквизит можно взять в качестве первичного ключа данной таблицы. Когда вы выбираете какой-то реквизит на роль первичного ключа, проверьте, действительно ли он уникален, т. е. не встречается ли более одного раза во всех строках таблицы.

Если не существует единственной колонки, обладающей свойством неповторяемости данных в ее строках (это свойство называют еще уникальностью), то разработ-

чику таблицы самому следует определить первичный ключ на основе двух и более колонок, которые вместе дают требуемую однозначность. Нужно просто подобрать такие поля. При задании таблицы обычно такие атрибуты помечаются, а физически (при образовании ключа) они сцепляются друг с другом, обеспечивая свою уникальность для каждой строки таблицы.

Вместе с первичным ключом таблицы (primary key) существует и так называемый *уникальный ключ* (unique key). Уникальный ключ — это не идентификатор строки, каким является первичный ключ. Основная цель уникального ключа — обеспечение ввода в колонку, названную уникальным ключом, уникального значения. У таблицы может быть только один первичный ключ и любое количество уникальных.

Первичный и уникальный ключи задаются при создании таблицы, когда соответствующие поля относят к первичному ключу или к уникальному. Если поле помечено как уникальное, то в него для каждой строки можно вводить *только различные данные*. Если в некоторую строку данного поля ввести значение, которое уже встречалось в какой-либо другой строке, то система выдаст сообщение об ошибке. Так должна работать любая БД.

Набор правил при разработке таблицы

При создании базы данных требуется разработать набор правил для каждой таблицы и колонок. Правила устанавливают и обеспечивают целостность данных (связанные данные должны и модифицироваться, и удаляться совместно).

Определение ограничений на целостность данных

Ограничения на целостность данных — это правила, управляющие связями: "колонка—таблица" и "таблица—таблица", а также учитывающие проверку данных на достоверность.

Эти правила должны охватывать все транзакции, которые имеют доступ к БД и автоматически поддерживаются системой.

Транзакция — это совокупность операций, обеспечивающих завершенность некоторого действия. Например, с вашего компьютера, установленного в банке, подана команда: "Перечислить такую-то сумму с такого-то счета на счет такого-то клиента". Операция (в данном случае это транзакция) будет считаться завершенной, если все детали будут выполнены, и до окончания перечисления не произойдет ни отказа компьютера, ни чего-либо другого, помешавшего завершению перечисления. Ограничения на целостность данных могут быть применены как в целом к таблице, так и к отдельной колонке. Мы уже знаем: ограничения, налагаемые на первичный или уникальный ключ (по их определению) гарантируют, что любые два значения в колонке или в наборе колонок не будут совпадать.

Значения данных, которые однозначно определяют строки (первичный ключ) в одной таблице, могут появляться в других таблицах. В этой связи вводится понятие *внешнего ключа* (foreign key). Внешний ключ — это колонка или набор колонок

таблицы, которые содержат значения, совпадающие с первичным ключом в другой таблице.

Принудительное обеспечение целостности данных

При задании внешнего ключа необходимо быть уверенным, что станет поддерживаться целостность данных, когда более одной таблицы ссылаются на одни и те же данные. Строки одной таблицы всегда должны соответствовать строкам таблиц, ссылающихся на эту таблицу.

Например, вы обрабатываете движение товаров на складе. В вашей БД имеются таблицы **Поставщики** и **Покупатели**, содержащие коды товаров, а также таблица **Ценник** (сегодня модно давать ей название **Прайс-лист**), в которой имеются характеристики товаров (цена, единица измерения и т. п.). Все три таблицы связаны между собой кодом товара. И, например, неаккуратное изменение какой-либо строки в таблице **Ценник** может разрушить данные в остальных таблицах. Поддержание режима целостности данных обеспечивает их нормальное функционирование. Поэтому в программном обеспечении баз данных предусматривают возможность принудительного обеспечения ссылочной целостности с помощью задания внешнего ключа. Но прежде чем задать внешний ключ, необходимо определить уникальный и первичные ключи, на которые ссылается этот внешний ключ. Тогда, если информацию изменить в одном месте, она автоматически будет соответственно изменена и во всех остальных местах, в которых уже существует.

Выбор индексов

Индекс (указатель) — это механизм, используемый для ускорения извлечения записей таблицы в ответ на некоторый запрос при определенных условиях поиска. Это как поиск по указателю в книге: индекс (указатель) содержит номера страниц, связанных с данным термином, что позволяет быстро найти нужную страницу. Индекс базы данных служит логическим указателем на физическое размещение (адрес) строки в таблице. Индекс хранит каждое значение колонки, по которой предполагается вести поиск в таблице, или значения колонок с указателями на все дисковые блоки, содержащие строки с таким значением. Извлечение данных является быстрым, потому что значения индекса упорядочены и сами они относительно невелики. Это позволяет системе быстро отыскивать ключевые значения.

Как только в индексе ключевое значение найдено, система выбирает связанный с этим значением указатель на физическое размещение данного, которое ищется. Использование индексов требует меньшего количества страниц выборки, чем последовательный просмотр каждой строки таблицы. Индекс может быть определен на одной или нескольких колонках таблицы.

Язык SQL

Каждая система управления реляционной БД предусматривает применение специального языка структурированных запросов SQL (Structured Query Language), при-

чем используется не весь язык, а некоторое его подмножество. Например, Microsoft SQL Server использует вариант SQL, названный Transact-SQL, который, по мнению разработчиков этого продукта, наиболее подходит для созданной ими структуры БД.

На языке SQL создаются элементы БД (структуры таблиц, хранимые процедуры и т. д.). На этом же языке пишутся запросы к БД на выборку или модификацию информации. Мы дадим лишь основные сведения о наиболее часто употребляемых операторах этого языка, потому что на их использовании построена работа с наборами данных.

Работа с наборами данных построена на принципе "запрос—ответ", а запросы формируются на языке SQL. Правда, среда программирования предоставляет возможность определенной автоматизации этого процесса, однако без понимания сути дела пользоваться услугами этой среды довольно затруднительно, ибо часто приходится самому вмешиваться в процесс построения запросов. Наиболее часто используемые операторы SQL приведены в табл. 12.3.

Таблица 12.3. *Наиболее используемые операторы SQL*

Оператор	Краткая суть
SELECT	С его помощью выбирают из таблиц БД определенные столбцы
WHERE	Это элемент оператора, в котором задаются ограничения на выборку данных
ORDER BY	Это элемент оператора, в котором задаются сведения об упорядочении выборки по тем или иным полям, он обеспечивает сортировку данных как по возрастанию, так и по убыванию значений отдельных полей
INSERT	Добавляет записи к таблице
DELETE	Удаляет записи из таблицы
UPDATE	Модифицирует отдельные поля записи

Чтобы иметь некоторое представление, например, об операторе **SELECT**, приведем сокращенный синтаксис этого оператора:

```
SELECT
   {* | <val> [, <val> ...]}
   [INTO :var [, :var ...]]
    FROM <tableref> [, <tableref> ...]
   [WHERE <search_condition>]
   [ORDER BY <order_list>]
```

Здесь:

- **INTO** — содержит список переменных (`var`), в которых станут запоминаться извлекаемые поля таблиц;
- **FROM** — в этом элементе перечисляются таблицы, из которых будет происходить выборка;

- **WHERE** — этот элемент задает условия поиска данных;
- **ORDER BY** — задает сортировку возвращаемых строк (по возрастанию или убыванию значений полей таблицы);
- `<val> = { col | <expr> | <function>}`
  - `col` — имя поля таблицы;
  - `expr` — некоторое SQL-выражение. Например, можно записать оператор **SELECT** в таком виде: **SELECT** `Salary * 0.13 from Employee` (что означает выбрать из таблицы `Employee` столбец `Salary`, умноженный на `0.13`). Результатом будет столбец суммы налога по каждому работнику. `Salary * 0.13` — это и есть SQL-выражение;
  -
```
<function> = COUNT (* | [ALL] <val> | DISTINCT <val>)
| SUM ([ALL] <val> | DISTINCT <val>)
| AVG ([ALL] <val> | DISTINCT <val>)
| MAX ([ALL] <val> | DISTINCT <val>)
| MIN ([ALL] <val> | DISTINCT <val>)
| CAST (<val> AS <datatype>)
| UPPER (<val>)
```

    это список функций, которые могут участвовать в операторе **SELECT**.
- `<tableref>` — это имена таблиц, из которых происходит выборка;
- `<search_condition>` — это условия отбора данных в выборку:

```
| <val> [NOT] BETWEEN <val> AND <val>
| <val> [NOT] LIKE <val> [ESCAPE <val>]
| <val> [NOT] IN (<val> [, <val> ...] | <select_list>)
| <val> IS [NOT] NULL
| <val> {>= | <=}
| <val> [NOT] {= | < | >}
| {ALL | SOME | ANY} (<select_list>)
```

Примеры оператора *SELECT*

- **SELECT** `JOB_GRADE, JOB_CODE, JOB_COUNTRY, MAX_SALARY FROM PROJECT;`

  Этот оператор выбирает из таблицы `PROJECT` четыре поля (т. е. все строки по этим полям).

  А куда, кстати, поместятся результаты выборки? Они будут в том объекте, который этот запрос выполняет: у такого компонента имеется командная строка (Command Text), где и должен находиться текст оператора выборки. Компонент выполнит этот оператор, а результат либо поместит в своем буфере, откуда его надо будет извлекать, либо передаст другому компоненту, с которым он связан. Все зависит от того, как построена среда программирования. В нашем случае будет иметь место последнее утверждение: к каждой таблице подключается такой компонент (он называется `Table Adapter`), который формирует запрос на выборку, выполняет эту выборку и передает ее результаты в связанную с ним таблицу.

- **SELECT** * **FROM** COUNTRIES;

  В этом операторе выбираются все строки из таблицы с именем COUNTRIES.

- **EXEC SQL**
  SELECT COUNT (*) **INTO** :cnt **FROM** COUNTRY
  WHERE POPULATION > 5000000;

  Этот оператор использует при выборке функцию для подсчета всех строк в таблице. Поиск должен удовлетворять условию, определенному в элементе **WHERE** (нужно выбрать из таблицы COUNTRY те записи, в которых значение поля POPULATION больше 5 000 000).

- **SELECT** CITY, STATE **FROM** CITIES **ORDER BY** STATE;

  Этот оператор извлекает из таблицы CITIES два столбца: CITY и STATE, а также упорядочивает строки по значению столбца STATE.

И еще об одном надо сказать: в данной среде программирования общение с базами данных происходит по так называемой ADO-технологии, когда, чтобы добраться до соответствующей БД, формируется специальная строка, называемая строкой соединения с БД (Connecting String). Она формируется с помощью средств самой среды через диалоговые окна общения с пользователем. Среда Visual C++ 2008 позволяет подсоединяться к двум типам баз данных: базам данных типа MS Access и базам данных типа MS SQL Server. С точки зрения технологии работы с ними, эти БД между собой ничем не отличаются. В момент формирования строки соединения следует просто выбрать из двух вариантов.

Наборы данных (компонент *DataSet*)

Примечание

В настоящей версии работа с наборами данных отключена. Здесь приводится работа с наборами данных для версии 2005 г. (см. пояснение во *введении*).

Наборы данных (НД) — это такие объекты DataSet (в нашем случае это компонент палитры компонентов), через которые происходит подключение приложения к базам данных. При этом информация из БД загружается в НД, который затем станет обеспечивать данными ваше приложение из локальной кэш-памяти. С таким НД можно работать даже тогда, когда ваше приложение отсоединено от базы данных. НД поддерживает информацию об изменениях таким образом, что обновленные данные могут быть снова отосланы БД, когда ваше приложение подсоединится к ней. Короче говоря, НД — это такой компонент, через который осуществляется связь приложения с выбранной (во время установления этой связи) БД соответствующего типа.

Наборы данных, предусмотренные в данной версии среды программирования обозначаются как typed и untyped. Работа с первым типом в данной версии отключена, хотя и осталась возможность встретиться с этим наименованием при соединении с базами данных. Это так называемые НД со схемами. Второй тип НД — без схем. НД со схемами предназначены для подключения к таблицам баз данных, находя-

щихся вне вашего приложения, т. е. извне. С помощью таких типизированных наборов извлекаются взаимосвязи между таблицами из специальных файлов БД и рисуются их схемы прямо в окне дизайнера набора данных. С использованием такого типа НД легче составлять программы. Нетипизированные наборы данных работают с таблицами, которые строятся вами на этапе дизайна приложения прямо в самом будущем приложении. Эта возможность в данной среде оставлена, что позволяет подсоединяться к внешним базам данных через таблицы, построенные в вашем приложении точно по структуре тех таблиц внешней базы данных, с которыми вы хотите установить связь и их обрабатывать в дальнейшем.

Когда вы пытаетесь поместить компонент `DataSet` в форму (а он находится в группе **Data** палитры компонентов), то открывается диалоговое окно для выбора типа НД (рис. 12.1).

Рис. 12.1. Выбор типа НД

Так как выше мы отметили, что работа в данной версии ведется с нетипизированными НД, то в появившемся окне надо выбрать переключатель **Untyped dataset**. В результате ничего не случится: компонент поместится на поддон формы. Останется только его настроить на получение данных из некоторой таблицы базы данных. Сам этот компонент не участвует в подсоединении к базе данных. Он лишь позволяет построить вам у себя таблицу данных по той структуре, какая имеется у таблицы базы данных, к которой вы хотите подсоединиться в вашем приложении.

Общая технология организации работы с базой данных в приложении

В дальнейшем мы станем работать с базами данных, средства создания которых у нас всегда под рукой: это MS Access, поставляемое вместе с операционной системой. Итак, технология такова.

1. Создается таблица-файл с помощью MS Access. Ее надо достать средствами VC++ 2011.

2. Для этого берется компонент `oleDbDataAdapter`, который настраивается на доступ к БД. Вы можете не обнаружить некоторые компоненты в списке **ToolBox**, т. к. в нем отражены только самые необходимые компоненты на взгляд разработчика среды. Остальные, предусмотренные средой программирования, вы можете подключить сами к **ToolBox**, воспользовавшись опцией главного меню **Choose ToolBox Items**. Откроется диалоговое окно со списком подключаемых компонентов, в котором в соответствующей папке (для начинающих — это компоненты из папки Framework...) следует найти нужный подключаемый компонент и слева от него поставить галочку. После закрытия окна компонент попадет в **ToolBox**.
3. После `oleDbDataAdapter` берется другой компонент: `dataSet`. В нем создается таблица по структуре, совпадающая с таблицей из БД.
4. Затем берется третий компонент: `dataGridView`. Он настраивается на высветку строк из таблицы, которая создается в предыдущем компоненте вручную на этапе проектирования. Тогда первого заставляют с помощью специальных команд достать из БД данные и записать в таблицу второго, а третий, связанный со вторым, должен на экране отразить то, что второй получил.

Кроме этих трех основных, необходимо работать еще с четырьмя:

- `oleDbConnection` — этот компонент появляется на поддоне формы, когда мы установили связь с БД через `oleDbDataAdapter`. Появляется как спутник. Дело в том, что у этого компонента есть свойство, через которое формируется так называемая строка соединения — путь от приложения к таблице БД. Поэтому через такой компонент можно самостоятельно подсоединиться к БД. Но в данном случае удобнее подсоединяться сразу через `oleDbDataAdapter`;
- `dataGridView` — через этот компонент на экран выводится таблица из БД, с которой мы намерены работать, которую достал `oleDbDataAdapter` и поместил в `dataSet`. Компонент позволяет легко корректировать таблицу и отсылать ее обратно в БД;
- `bindingSource` — компонент содержит связь с `dataSet`, чтобы через нее передать данные таблицы из БД компоненту `dataGridView`;
- `bindingNavigator` — компонент управляет действиями по работе пользователя с таблицей данных, выводимых на экран компонентом `dataGridView` (перемещение по элементам таблицы, добавление элементов, удаление элементов, другие действия, связанные с обработкой элементов, которые можно осуществлять, обрабатывая события этого компонента).

Пример работы с базой данных

С помощью средств MS Access создадим БД, состоящую из одной таблицы, которую назовем условно "Авторы". Как создавать такие таблицы в среде MS Access мы здесь не рассматриваем. Это не входит в нашу задачу. Отметим только один странный момент (возможно, что это недоработка разработчика среды), что если создать БД MS Access версии, поставляемой с Windows 7, то компоненты типа ole

(`oleDbConnection`, например) не работают с такой БД. В Интернете полно жалоб по этому поводу. Поэтому, если мы хотим воспользоваться инструментом MS Access, следует создать БД более ранней версии. Хорошо, что этот вариант предусмотрен в поставляемой новой версии MS Access. Таким образом, создадим таблицу "Авторы" MS Access версии 2000, например. Расширение ее файла — mdb. Расширение же файлов новейшей версии MS Access — accdb. Вид таблицы показан на рис. 12.2.

Рис. 12.2. Таблица "Авторы"

Отметим, что при создании таблицы первое ее поле — автоматический счетчик строк — является ключевым полем: у него одним из свойств является то, что это поле индексированное и не может иметь более одного значения, т. е. уникальное. Об этом надо помнить, т. к. при настройке компонентов работы с БД нам придется это учитывать.

Начнем создавать приложение по технологии, описанной выше. Создадим приложение обычным путем, выполнив опции **File | NewProject | Windows Forms Application**. На экране увидим пустую форму, которую нам предстоит заполнить. Поместим в форму компонент `oleDbDataAdapter` (в дальнейшем — адаптер). Тут же откроется диалоговое окно для подключения к БД. Если мы ранее подключались к какой-то БД, то ее имя высветится в окне. Если нас эта БД устраивает, то нажимаем на **OK** и идем дальше. Если нет, нажимаем на кнопку **New Connection** (рис. 12.3).

Отметим, что все связи с базами данных, устанавливаемыми приложениями, отражаются в окне **Server Explorer** (вкладка для открытия окна — рядом с вкладкой **ToolBox** в правом торце рабочего стола). Если вам мешают эти данные, то можно их удалить через это окно, воспользовавшись его контекстным меню. Тогда, например, в открывшемся окне, показанном на рис. 12.3, не появится последняя БД, с которой была установлена связь. Но можно воспользоваться и возможностью заранее установить связь с БД. Для этого следует выполнить опции главного меню **Tools | Connect to DataBase**. Далее необходимо выполнить шаги, описанные ниже, когда мы настраиваем адаптер и дошли до шага, когда нажимаем на кнопку **New Connection**.

После нажатия на кнопку **New Connection** открывается окно для организации непосредственной связи с базой данных (рис. 12.4).

Смотрим на поле **Data source** (источник данных). Источник **Microsoft SQL Server (OLE DB)** нас не устраивает, т. к. мы работаем с БД типа MS Access. Поэтому

Рис. 12.3. Выбор соединения с БД

Рис. 12.4. Окно организации связи с БД

Рис. 12.5. Выбор типа источника данных

нажимаем кнопку **Change** (изменить). Открывается диалоговое окно для выбора подходящего типа источника данных (рис. 12.5).

После нажатия на кнопку **OK**, появится окно, показанное на рис. 12.6. Кнопкой **Browse** находим нашу БД (рис. 12.7).

Остается проверить, действительно ли произошло соединение с БД. Для этого нажимаем на кнопку **Test Connection** (проверить соединение). Результат — на рис. 12.8.

Рис. 12.6. Окно для поиска БД типа выбранной на предыдущем шаге

Рис. 12.7. Путь к БД

Рис. 12.8. Проверка соединения с БД

Видим, что соединение произошло. Переходим к следующему шагу: после нажатия на кнопку **OK**, появится знакомое уже диалоговое окно, в котором отразилось найденное соединение (рис. 12.9).

Рис. 12.9. Окно соединения с источником данных

Нажимаем на кнопку **Next**, в результате чего получаем открытое окно, показанное на рис. 12.10.

Видим, что переключатель выбора работы с хранимыми процедурами заблокирован, и нам остается только нажать на кнопку **Next**. Откроется диалоговое окно, показанное на рис. 12.11, окно для определения оператора выборки данных из БД, с которой ранее было соединение.

Рис. 12.10. Окно выбора работы с SQL-операторами или с хранимыми процедурами

Рис. 12.11. Работа с SQL-операторами

Мы можем сами написать оператор `SELECT`, который должен выбирать данные из таблицы "Авторы". Но мы воспользуемся кнопкой **Query Builder** (построитель запроса), которая откроет окно для выбора конструкции оператора `SELECT` (рис. 12.12).

В данном примере мы работаем с таблицей "Авторы", поэтому выбираем вариант, показанный на рис. 12.12, и нажимаем на кнопку **Add** (добавить).

Рис. 12.12. Построение оператора `SELECT`

Нажимаем на кнопку **Close** (закрыть), окно закрывается, и мы видим окно, показанное на рис. 12.13.

Теперь мы можем галочками отметить поля в таблице окна "Авторы", которые станут выбираться из таблицы БД, а затем проверить правильность сформированного на основании отмеченных полей оператора `SELECT`, нажав на кнопку **Execute Query** (выполнить запрос). Результат этих действий — на рис. 12.14.

Из рисунка видно, что связь с БД построена верно, и выборка данных происходит. Нажмем на кнопку **OK** и попадем в окно, показанное на рис. 12.15.

Нажимаем на кнопку **Next**, попадаем в окно, показанное на рис. 12.16.

Нажимаем на кнопку **Finish**, попадаем снова в окно с формой: конфигурация (настройка) адаптера завершена. Теперь он может доставать таблицу из БД и обеспечивать ее ведение.

Займемся теперь компонентом `dataSet`. Откроем свойства компонента (рис. 12.17).

Обратим внимание на свойство `Tables` (таблицы). Справа от названия свойства — поле этого свойства, в котором написано `Collection` (множество). То есть имеется

Рис. 12.13. Формирование оператора `SELECT` на основании выбранной таблицы БД

в виду, что таблиц может быть много. Почему? База данных может содержать не одну таблицу, а множество. Но мы зададим (построим) с помощью этого свойства только одну таблицу: такую, структура которой совпадает по структуре с таблицей "Авторы".

Щелкаем на кнопке с многоточием в поле рассматриваемого свойства (сначала надо щелкнуть на свойстве — отметить его; тогда появится кнопка с многоточием). Открывается пустое диалоговое окно. В нем щелкаем на кнопке **Add**: в левое поле окна добавляется строка `Table1`, а в правом поле окна появляются свойства `Table1`, которые определяют создаваемый объект (рис. 12.18).

Одним из свойств, которое нас интересует в данный момент, является свойство `Columns` (столбцы). Раньше мы рассмотрели, что именно столбцы определяют суть таблицы. Щелкнем в поле этого свойства. Там же, в поле, появится кнопка с многоточием. Щелкнем на этой кнопке. Откроется диалоговое окно для формирования столбцов будущей формы. Окно такое же, как и для задания таблиц. Добавим кнопкой **Add** шесть элементов в левую часть окна. В правой части обратим внимание на свойства `Caption` (название столбца), `Data Type` (тип данных в столбце) и `Unique` (задается единственность, уникальность элементов в этом поле при переходе от одной строки таблицы к другой, точнее — неповторяемость). Если задать это свойство как `true`, то такое поле можно брать в качестве элемента ключа в таблице. В нашей

Рис. 12.14. Действие сформированного оператора `SELECT`

Рис. 12.15. Построенный оператор `SELECT`

Рис. 12.16. Окно с информацией,
какие SQL-операторы сформированы для работы с таблицей БД

Рис. 12.17. Свойства `dataSet`

Рис. 12.18. Диалоговое окно для задания (создания, построения) таблиц

таблице "Авторы" все поля имеют строковый тип, что совпадает с принятым типом по умолчанию для свойства `Data Type`. Но самое первое поле должно быть ключевым, потому что мы строим таблицу по структуре идентичную таблице "Авторы". Поэтому у столбца с именем `Column1` надо изменить тип данного на `int 64` (двойное целое) и задать свойство `Unique=true`. Названия пока оставим те, что сформировала среда (для простоты). Поэтому формирование таблицы можно считать законченным. Вид ее показан на рис. 12.19.

Остался еще один момент: уточнить свойство построенной таблицы, которое называется `PrimaryKey` (первичный ключ). Если щелкнуть в поле этого свойства, откроется кнопка, нажав на которую увидим список полей таблицы, которые могут служить первичным ключом (одно поле или совокупность полей). Надо проставить галочки у тех полей, которые образуют первичный ключ. В нашем случае первичный ключ образует поле `Column1`. Его и помечаем (рис. 12.20).

Таким образом, с компонентом `dataSet` разобрались. То есть сформировали адаптер, который свяжется с таблицей в БД MS Access и заполнит ее данными только что построенную таблицу `Table1` в `dataSet`.

Теперь надо обеспечить вывод `Table1` на экран для просмотра и корректировки ее данных, чтобы потом уже откорректированную таблицу снова вернуть в БД (ранее мы отмечали, что `dataSet` работает только в памяти, а чтобы его содержимое попало назад в БД, это содержимое надо принудительно туда записать. Это делает как раз метод `Update`, созданный ранее в адаптере).

Рассмотрим компонент `bindingSource`. Перечень его свойств и их настройка показаны на рис. 12.21.

Рис. 12.19. Таблица, сформированная в компоненте `dataSet`

Рис. 12.20. Выбор первичного ключа в построенной таблице

Рис. 12.21. Свойства `bindingSource` и их настройка

По настройкам видно, что этот компонент связывается с объектом `dataSet` и его членом `Table1`. Вот эти-то элементы и попадут как связующие в следующий компонент, который призван показать наполненную данными из базы данных таблицу `Table1` на экране. Этот компонент — `dataGridView`, его вид в форме показан на рис. 12.22.

Настройка свойств компонента показана на рис. 12.23.

Рис. 12.22. `dataGridView` в форме приложения

Рис. 12.23. Настройка свойств `dataGridView`

Итак, `dataGridView` станет высвечивать на экране таблицу `Table1`. Но управлять перемещением по этой таблице, ее корректировкой призван другой компонент. Это `bindingNavigator` или просто Навигатор. Связь его с `dataGridView` — через `bindingSource`. Эта связь устанавливается так, как показано на рис. 12.24.

Вид Навигатора в форме показан также на рис. 12.24.

Рис. 12.24. Установление связи Навигатора с `dataGridView` через `bindingSource`

Остался еще один не выполненный шаг: создание таблицы соответствия полей исходной БД и построенной. Такая таблица определяется как свойство компонента адаптер. Это свойство называется `TableMappings` (схема связей в таблицах). Эта схема показана на рис. 12.25.

Добавим теперь в форму приложения три кнопки. Первая будет вызывать команду запуска адаптера и заполнение им таблицы `Table1`, вторая — вызывать команду записи обновленной в памяти `Table1` обратно в БД, третья — вызывать команду завершения приложения. Обработчики этих кнопок приведены в листинге 12.1.

Листинг 12.1

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
                        // загрузить данные из БД
  this->dataSet1->Clear();  //Очистка буфера  перед чтением туда
                            //таблицы
  this->oleDbDataAdapter1->Fill(dataTable1);
 }

private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
{
              //Сохранить данные в БД
    this->oleDbDataAdapter1->Update(dataSet1->Tables["Авторы"]);
}
```

```
private: System::Void button3_Click(System::Object^  sender, System::EventArgs^ e)
 {
   //Выйти из приложения
   this->Close();
 }
```

Рис. 12.25. Таблица соответствий между элементами таблицы из БД и таблицы из `dataSet`

Вид формы приложения перед компиляцией показан на рис. 12.26.

Компилируем приложение, нажав на клавишу <F7>. Компиляция прошла успешно. Запускаем приложение на выполнение клавишей <F5>. Вид формы после компиляции и запуска на выполнение показан на рис. 12.27.

Нажимаем на кнопку **Загрузить данные из БД в таблицу**. Результат показан на рис. 12.28.

Проверим, происходит ли удаление строк, добавка строк, изменение значений некоторых полей строк и, наконец, пересылка откорректированной таблицы обратно в БД. Результаты этих действий показаны на рис. 12.29 и 12.30.

Когда происходило испытание приложения в режиме добавления строки (это делалось кнопкой "Плюс" Навигатора, по которой создается пустая строка, куда вписываются данные полей), среда программирования требовала, чтобы задавался ключ

Рис. 12.26. Вид формы приложения перед компиляцией

Рис. 12.27. Вид формы после компиляции и запуска на выполнение

Рис. 12.28. Загрузка таблицы "Авторы" из БД

Рис. 12.29. Откорректированная в памяти (на экране) таблица "Авторы" и отосланная назад в БД

Рис. 12.30. Прочитанная снова таблица "Авторы"

строки, т. е. значение поля в колонке `Column1`. Это очень неудобно, т. к. значение ключевого поля трудно выбрать уникальным, когда в таблице много строк. Оказывается, из этой неприятности есть выход. Надо открыть свойства колонок таблицы в `dataSet` и у ключевого поля `Column1` изменить свойство `AutoIncrement` (автоматическое наращивание по единице) с `false` на `true` (рис. 12.31).

Приложение, реализующее рассмотренное выше решение, приводится в листинге 12.2.

Листинг 12.2

```
#pragma once

namespace db2011длягл122 {

   using namespace System;
   using namespace System::ComponentModel;
```

Рис. 12.31. Придание ключевому полю таблицы свойства автоматического наращивания значения

```
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;

/// <summary>
/// Summary for Form1
/// </summary>
public ref class Form1 : public System::Windows::Forms::Form
{
public:
   Form1(void)
   {
      InitializeComponent();
      //
      //TODO: Add the constructor code here
      //
   }

protected:
   /// <summary>
   /// Clean up any resources being used.
   /// </summary>
   ~Form1()
```

```
      {
         if (components)
         {
               delete components;
         }
      }
   private: System::Data::OleDb::OleDbCommand^  oleDbSelectCommand1;
   protected:
   private: System::Data::OleDb::OleDbCommand^  oleDbInsertCommand1;
   private: System::Data::OleDb::OleDbCommand^  oleDbUpdateCommand1;
   private: System::Data::OleDb::OleDbCommand^  oleDbDeleteCommand1;
   private: System::Data::OleDb::OleDbDataAdapter^  oleDbDataAdapter1;
   private: System::Data::OleDb::OleDbConnection^  oleDbConnection1;
   private: System::Windows::Forms::BindingSource^  bindingSource1;
   private: System::Data::DataSet^  dataSet1;
   private: System::Data::DataTable^  dataTable1;
   private: System::Data::DataColumn^  dataColumn1;
   private: System::Data::DataColumn^  dataColumn2;
   private: System::Data::DataColumn^  dataColumn3;
   private: System::Data::DataColumn^  dataColumn4;
   private: System::Data::DataColumn^  dataColumn5;
   private: System::Data::DataColumn^  dataColumn6;
   private: System::Windows::Forms::BindingNavigator^  bindingNavigator1;
   private: System::Windows::Forms::ToolStripButton^
bindingNavigatorAddNewItem;
   private: System::Windows::Forms::ToolStripLabel^  bindingNavigatorCountItem;
   private: System::Windows::Forms::ToolStripButton^
bindingNavigatorDeleteItem;
   private: System::Windows::Forms::ToolStripButton^
bindingNavigatorMoveFirstItem;
   private: System::Windows::Forms::ToolStripButton^
bindingNavigatorMovePreviousItem;
   private: System::Windows::Forms::ToolStripSeparator^
bindingNavigatorSeparator;
   private: System::Windows::Forms::ToolStripTextBox^
bindingNavigatorPositionItem;
   private: System::Windows::Forms::ToolStripSeparator^
bindingNavigatorSeparator1;
   private: System::Windows::Forms::ToolStripButton^
bindingNavigatorMoveNextItem;
   private: System::Windows::Forms::ToolStripButton^
bindingNavigatorMoveLastItem;
   private: System::Windows::Forms::ToolStripSeparator^
bindingNavigatorSeparator2;
   private: System::Windows::Forms::Button^  button1;
   private: System::Windows::Forms::DataGridView^  dataGridView1;
   private: System::Windows::Forms::Button^  button2;
   private: System::Windows::Forms::Button^  button3;
```

```
   private: System::Windows::Forms::DataGridViewTextBoxColumn^  Column1;
   private: System::Windows::Forms::DataGridViewTextBoxColumn^
column2DataGridViewTextBoxColumn;
   private: System::Windows::Forms::DataGridViewTextBoxColumn^
column3DataGridViewTextBoxColumn;
   private: System::Windows::Forms::DataGridViewTextBoxColumn^
column4DataGridViewTextBoxColumn;
   private: System::Windows::Forms::DataGridViewTextBoxColumn^
column5DataGridViewTextBoxColumn;
   private: System::Windows::Forms::DataGridViewTextBoxColumn^
column6DataGridViewTextBoxColumn;
   private: System::ComponentModel::IContainer^  components;

   private:
      /// <summary>
      /// Required designer variable.
      /// </summary>


#pragma region Windows Form Designer generated code
      /// <summary>
      /// Required method for Designer support — do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      void InitializeComponent(void)
      {
         this->components = (gcnew System::ComponentModel::Container());
         System::ComponentModel::ComponentResourceManager^  resources = (gcnew
System::ComponentModel::ComponentResourceManager(Form1::typeid));
         this->oleDbSelectCommand1 = (gcnew
System::Data::OleDb::OleDbCommand());
         this->oleDbConnection1 = (gcnew
System::Data::OleDb::OleDbConnection());
         this->oleDbInsertCommand1 = (gcnew
System::Data::OleDb::OleDbCommand());
         this->oleDbUpdateCommand1 = (gcnew
System::Data::OleDb::OleDbCommand());
         this->oleDbDeleteCommand1 = (gcnew
System::Data::OleDb::OleDbCommand());
         this->oleDbDataAdapter1 = (gcnew
System::Data::OleDb::OleDbDataAdapter());
         this->bindingSource1 = (gcnew
System::Windows::Forms::BindingSource(this->components));
         this->dataSet1 = (gcnew System::Data::DataSet());
         this->dataTable1 = (gcnew System::Data::DataTable());
         this->dataColumn1 = (gcnew System::Data::DataColumn());
         this->dataColumn2 = (gcnew System::Data::DataColumn());
         this->dataColumn3 = (gcnew System::Data::DataColumn());
         this->dataColumn4 = (gcnew System::Data::DataColumn());
```

```
        this->dataColumn5 = (gcnew System::Data::DataColumn());
        this->dataColumn6 = (gcnew System::Data::DataColumn());
        this->bindingNavigator1 = (gcnew
System::Windows::Forms::BindingNavigator(this->components));
        this->bindingNavigatorAddNewItem = (gcnew
System::Windows::Forms::ToolStripButton());
        this->bindingNavigatorCountItem = (gcnew
System::Windows::Forms::ToolStripLabel());
        this->bindingNavigatorDeleteItem = (gcnew
System::Windows::Forms::ToolStripButton());
        this->bindingNavigatorMoveFirstItem = (gcnew
System::Windows::Forms::ToolStripButton());
        this->bindingNavigatorMovePreviousItem = (gcnew
System::Windows::Forms::ToolStripButton());
        this->bindingNavigatorSeparator = (gcnew
System::Windows::Forms::ToolStripSeparator());
        this->bindingNavigatorPositionItem = (gcnew
System::Windows::Forms::ToolStripTextBox());
        this->bindingNavigatorSeparator1 = (gcnew
System::Windows::Forms::ToolStripSeparator());
        this->bindingNavigatorMoveNextItem = (gcnew
System::Windows::Forms::ToolStripButton());
        this->bindingNavigatorMoveLastItem = (gcnew
System::Windows::Forms::ToolStripButton());
        this->bindingNavigatorSeparator2 = (gcnew
System::Windows::Forms::ToolStripSeparator());
        this->button1 = (gcnew System::Windows::Forms::Button());
        this->dataGridView1 = (gcnew System::Windows::Forms::DataGridView());
        this->button2 = (gcnew System::Windows::Forms::Button());
        this->button3 = (gcnew System::Windows::Forms::Button());
        this->Column1 = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());
        this->column2DataGridViewTextBoxColumn = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());
        this->column3DataGridViewTextBoxColumn = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());
        this->column4DataGridViewTextBoxColumn = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());
        this->column5DataGridViewTextBoxColumn = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());
        this->column6DataGridViewTextBoxColumn = (gcnew
System::Windows::Forms::DataGridViewTextBoxColumn());
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this-
>bindingSource1))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this-
>dataSet1))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this-
>dataTable1))->BeginInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this-
>bindingNavigator1))->BeginInit();
```

```
        this->bindingNavigator1->SuspendLayout();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(this-
>dataGridView1))->BeginInit();
        this->SuspendLayout();
        //
        // oleDbSelectCommand1
        //
        this->oleDbSelectCommand1->CommandText = L"SELECT Авторы.*\r\nFROM
Авторы";
        this->oleDbSelectCommand1->Connection = this->oleDbConnection1;
        //
        // oleDbConnection1
        //
        this->oleDbConnection1->ConnectionString =
L"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=\"D:\\База данных
MSAccess\\MSAccess-"
              L"2000\\Database1.mdb\"";
        //
        // oleDbInsertCommand1
        //
        this->oleDbInsertCommand1->CommandText = L"INSERT INTO `Авторы`
(`Поле1`, `Поле2`, `Поле3`, `Поле4`, `Поле5`) VALUES (\?, \?, "
              L"\?, \?, \?)";
        this->oleDbInsertCommand1->Connection = this->oleDbConnection1;
        this->oleDbInsertCommand1->Parameters->AddRange(gcnew cli::array<
System::Data::OleDb::OleDbParameter^  >(5) {(gcnew
System::Data::OleDb::OleDbParameter(L"Поле1",
              System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле1")), (gcnew
System::Data::OleDb::OleDbParameter(L"Поле2",
System::Data::OleDb::OleDbType::VarWChar,
              0, L"Поле2")), (gcnew
System::Data::OleDb::OleDbParameter(L"Поле3",
System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле3")),
              (gcnew System::Data::OleDb::OleDbParameter(L"Поле4",
System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле4")), (gcnew
System::Data::OleDb::OleDbParameter(L"Поле5",
              System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле5"))});
        //
        // oleDbUpdateCommand1
        //
        this->oleDbUpdateCommand1->CommandText = resources-
>GetString(L"oleDbUpdateCommand1.CommandText");
        this->oleDbUpdateCommand1->Connection = this->oleDbConnection1;
        this->oleDbUpdateCommand1->Parameters->AddRange(gcnew cli::array<
System::Data::OleDb::OleDbParameter^  >(16) {(gcnew
System::Data::OleDb::OleDbParameter(L"Поле1",
              System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле1")), (gcnew
System::Data::OleDb::OleDbParameter(L"Поле2",
System::Data::OleDb::OleDbType::VarWChar,
```

```
                0, L"Поле2")), (gcnew
System::Data::OleDb::OleDbParameter(L"Поле3",
System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле3")),
                (gcnew System::Data::OleDb::OleDbParameter(L"Поле4",
System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле4")), (gcnew
System::Data::OleDb::OleDbParameter(L"Поле5",
                System::Data::OleDb::OleDbType::VarWChar, 0, L"Поле5")), (gcnew
System::Data::OleDb::OleDbParameter(L"Original_Код",
System::Data::OleDb::OleDbType::Integer,
                0, System::Data::ParameterDirection::Input, false,
static_cast<System::Byte>(0), static_cast<System::Byte>(0), L"Код",
System::Data::DataRowVersion::Original,
                nullptr)), (gcnew
System::Data::OleDb::OleDbParameter(L"IsNull_Поле1",
System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input,
                static_cast<System::Byte>(0), static_cast<System::Byte>(0),
L"Поле1", System::Data::DataRowVersion::Original, true, nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле1",
System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input,
                false, static_cast<System::Byte>(0),
static_cast<System::Byte>(0), L"Поле1", System::Data::DataRowVersion::Original,
nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"IsNull_Поле2",
System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input,
                static_cast<System::Byte>(0), static_cast<System::Byte>(0),
L"Поле2", System::Data::DataRowVersion::Original, true, nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле2",
System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input,
                false, static_cast<System::Byte>(0),
static_cast<System::Byte>(0), L"Поле2", System::Data::DataRowVersion::Original,
nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"IsNull_Поле3",
System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input,
                static_cast<System::Byte>(0), static_cast<System::Byte>(0),
L"Поле3", System::Data::DataRowVersion::Original, true, nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле3",
System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input,
                false, static_cast<System::Byte>(0),
static_cast<System::Byte>(0), L"Поле3", System::Data::DataRowVersion::Original,
nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"IsNull_Поле4",
System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input,
                static_cast<System::Byte>(0), static_cast<System::Byte>(0),
L"Поле4", System::Data::DataRowVersion::Original, true, nullptr)),
```

```
                (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле4",
System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input,
                false, static_cast<System::Byte>(0),
static_cast<System::Byte>(0), L"Поле4", System::Data::DataRowVersion::Original,
nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"IsNull_Поле5",
System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input,
                static_cast<System::Byte>(0), static_cast<System::Byte>(0),
L"Поле5", System::Data::DataRowVersion::Original, true, nullptr)),
                (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле5",
System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input,
                false, static_cast<System::Byte>(0),
static_cast<System::Byte>(0), L"Поле5", System::Data::DataRowVersion::Original,
nullptr))});
        //
        // oleDbDeleteCommand1
        //
        this->oleDbDeleteCommand1->CommandText = resources-
>GetString(L"oleDbDeleteCommand1.CommandText");
        this->oleDbDeleteCommand1->Connection = this->oleDbConnection1;
        this->oleDbDeleteCommand1->Parameters->AddRange(gcnew cli::array<
System::Data::OleDb::OleDbParameter^  >(11) {(gcnew
System::Data::OleDb::OleDbParameter(L"Original_Код",
                System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input, false, static_cast<System::Byte>(0),
                static_cast<System::Byte>(0), L"Код",
System::Data::DataRowVersion::Original, nullptr)), (gcnew
System::Data::OleDb::OleDbParameter(L"IsNull_Поле1",
                System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input, static_cast<System::Byte>(0),
static_cast<System::Byte>(0),
                L"Поле1", System::Data::DataRowVersion::Original, true,
nullptr)), (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле1",
                System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input, false, static_cast<System::Byte>(0),
                static_cast<System::Byte>(0), L"Поле1",
System::Data::DataRowVersion::Original, nullptr)), (gcnew
System::Data::OleDb::OleDbParameter(L"IsNull_Поле2",
                System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input, static_cast<System::Byte>(0),
static_cast<System::Byte>(0),
                L"Поле2", System::Data::DataRowVersion::Original, true,
nullptr)), (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле2",
                System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input, false, static_cast<System::Byte>(0),
                static_cast<System::Byte>(0), L"Поле2",
System::Data::DataRowVersion::Original, nullptr)), (gcnew
System::Data::OleDb::OleDbParameter(L"IsNull_Поле3",
```

```
                System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input, static_cast<System::Byte>(0),
static_cast<System::Byte>(0),
                L"Поле3", System::Data::DataRowVersion::Original, true,
nullptr)), (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле3",
                System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input, false, static_cast<System::Byte>(0),
                static_cast<System::Byte>(0), L"Поле3",
System::Data::DataRowVersion::Original, nullptr)), (gcnew
System::Data::OleDb::OleDbParameter(L"IsNull_Поле4",
                System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input, static_cast<System::Byte>(0),
static_cast<System::Byte>(0),
                L"Поле4", System::Data::DataRowVersion::Original, true,
nullptr)), (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле4",
                System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input, false, static_cast<System::Byte>(0),
                static_cast<System::Byte>(0), L"Поле4",
System::Data::DataRowVersion::Original, nullptr)), (gcnew
System::Data::OleDb::OleDbParameter(L"IsNull_Поле5",
                System::Data::OleDb::OleDbType::Integer, 0,
System::Data::ParameterDirection::Input, static_cast<System::Byte>(0),
static_cast<System::Byte>(0),
                L"Поле5", System::Data::DataRowVersion::Original, true,
nullptr)), (gcnew System::Data::OleDb::OleDbParameter(L"Original_Поле5",
                System::Data::OleDb::OleDbType::VarWChar, 0,
System::Data::ParameterDirection::Input, false, static_cast<System::Byte>(0),
                static_cast<System::Byte>(0), L"Поле5",
System::Data::DataRowVersion::Original, nullptr))});
        //
        // oleDbDataAdapter1
        //
        this->oleDbDataAdapter1->DeleteCommand = this->oleDbDeleteCommand1;
        this->oleDbDataAdapter1->InsertCommand = this->oleDbInsertCommand1;
        this->oleDbDataAdapter1->SelectCommand = this->oleDbSelectCommand1;
        cli::array< System::Data::Common::DataColumnMapping^ >^ __mcTemp__1 =
gcnew cli::array< System::Data::Common::DataColumnMapping^  >(6) {(gcnew
System::Data::Common::DataColumnMapping(L"Код",
                L"Column1")), (gcnew
System::Data::Common::DataColumnMapping(L"Поле1", L"Column2")), (gcnew
System::Data::Common::DataColumnMapping(L"Поле2",
                L"Column3")), (gcnew
System::Data::Common::DataColumnMapping(L"Поле3", L"Column4")), (gcnew
System::Data::Common::DataColumnMapping(L"Поле4",
                L"Column5")), (gcnew
System::Data::Common::DataColumnMapping(L"Поле5", L"Column6"))};
        this->oleDbDataAdapter1->TableMappings->AddRange(gcnew cli::array<
System::Data::Common::DataTableMapping^  >(1) {(gcnew
System::Data::Common::DataTableMapping(L"Table",
                L"Авторы", __mcTemp__1))});
        this->oleDbDataAdapter1->UpdateCommand = this->oleDbUpdateCommand1;
```

```
        //
        // bindingSource1
        //
        this->bindingSource1->DataMember = L"Авторы";
        this->bindingSource1->DataSource = this->dataSet1;
        this->bindingSource1->Sort = L"Column2";
        //
        // dataSet1
        //
        this->dataSet1->DataSetName = L"NewDataSet";
        this->dataSet1->Tables->AddRange(gcnew cli::array<
System::Data::DataTable^  >(1) {this->dataTable1});
        //
        // dataTable1
        //
        this->dataTable1->Columns->AddRange(gcnew cli::array<
System::Data::DataColumn^  >(6) {this->dataColumn1, this->dataColumn2,
              this->dataColumn3, this->dataColumn4, this->dataColumn5, this-
>dataColumn6});
        cli::array< System::String^ >^ __mcTemp__2 = gcnew cli::array<
System::String^  >(1) {L"Column1"};
        this->dataTable1->Constraints->AddRange(gcnew cli::array<
System::Data::Constraint^  >(1) {(gcnew
System::Data::UniqueConstraint(L"Constraint1",
              __mcTemp__2, true))});
        this->dataTable1->PrimaryKey = gcnew cli::array<
System::Data::DataColumn^  >(1) {this->dataColumn1};
        this->dataTable1->TableName = L"Авторы";
        //
        // dataColumn1
        //
        this->dataColumn1->AllowDBNull = false;
        this->dataColumn1->AutoIncrement = true;
        this->dataColumn1->ColumnName = L"Column1";
        this->dataColumn1->DataType = System::Int64::typeid;
        //
        // dataColumn2
        //
        this->dataColumn2->ColumnName = L"Column2";
        //
        // dataColumn3
        //
        this->dataColumn3->ColumnName = L"Column3";
        //
        // dataColumn4
        //
        this->dataColumn4->ColumnName = L"Column4";
        //
        // dataColumn5
        //
```

```
        this->dataColumn5->ColumnName = L"Column5";
        //
        // dataColumn6
        //
        this->dataColumn6->ColumnName = L"Column6";
        //
        // bindingNavigator1
        //
        this->bindingNavigator1->AddNewItem = this-
>bindingNavigatorAddNewItem;
        this->bindingNavigator1->BindingSource = this->bindingSource1;
        this->bindingNavigator1->CountItem = this->bindingNavigatorCountItem;
        this->bindingNavigator1->DeleteItem = this-
>bindingNavigatorDeleteItem;
        this->bindingNavigator1->Items->AddRange(gcnew cli::array<
System::Windows::Forms::ToolStripItem^  >(11) {this-
>bindingNavigatorMoveFirstItem,
              this->bindingNavigatorMovePreviousItem, this-
>bindingNavigatorSeparator, this->bindingNavigatorPositionItem, this-
>bindingNavigatorCountItem,
              this->bindingNavigatorSeparator1, this-
>bindingNavigatorMoveNextItem, this->bindingNavigatorMoveLastItem, this-
>bindingNavigatorSeparator2,
              this->bindingNavigatorAddNewItem, this-
>bindingNavigatorDeleteItem});
        this->bindingNavigator1->Location = System::Drawing::Point(0, 0);
        this->bindingNavigator1->MoveFirstItem = this-
>bindingNavigatorMoveFirstItem;
        this->bindingNavigator1->MoveLastItem = this-
>bindingNavigatorMoveLastItem;
        this->bindingNavigator1->MoveNextItem = this-
>bindingNavigatorMoveNextItem;
        this->bindingNavigator1->MovePreviousItem = this-
>bindingNavigatorMovePreviousItem;
        this->bindingNavigator1->Name = L"bindingNavigator1";
        this->bindingNavigator1->PositionItem = this-
>bindingNavigatorPositionItem;
        this->bindingNavigator1->Size = System::Drawing::Size(654, 27);
        this->bindingNavigator1->TabIndex = 1;
        this->bindingNavigator1->Text = L"bindingNavigator1";
        //
        // bindingNavigatorAddNewItem
        //
        this->bindingNavigatorAddNewItem->DisplayStyle =
System::Windows::Forms::ToolStripItemDisplayStyle::Image;
        this->bindingNavigatorAddNewItem->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"bindingNavigatorAddNewItem.Image")));
        this->bindingNavigatorAddNewItem->Name =
L"bindingNavigatorAddNewItem";
```

```
        this->bindingNavigatorAddNewItem->RightToLeftAutoMirrorImage = true;
        this->bindingNavigatorAddNewItem->Size = System::Drawing::Size(23, 24);
        this->bindingNavigatorAddNewItem->Text = L"Add new";
        //
        // bindingNavigatorCountItem
        //
        this->bindingNavigatorCountItem->Name = L"bindingNavigatorCountItem";
        this->bindingNavigatorCountItem->Size = System::Drawing::Size(45, 24);
        this->bindingNavigatorCountItem->Text = L"of {0}";
        this->bindingNavigatorCountItem->ToolTipText = L"Total number of items";
        //
        // bindingNavigatorDeleteItem
        //
        this->bindingNavigatorDeleteItem->DisplayStyle =
System::Windows::Forms::ToolStripItemDisplayStyle::Image;
        this->bindingNavigatorDeleteItem->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"bindingNavigatorDeleteItem.Image")));
        this->bindingNavigatorDeleteItem->Name =
L"bindingNavigatorDeleteItem";
        this->bindingNavigatorDeleteItem->RightToLeftAutoMirrorImage = true;
        this->bindingNavigatorDeleteItem->Size = System::Drawing::Size(23, 24);
        this->bindingNavigatorDeleteItem->Text = L"Delete";
        //
        // bindingNavigatorMoveFirstItem
        //
        this->bindingNavigatorMoveFirstItem->DisplayStyle =
System::Windows::Forms::ToolStripItemDisplayStyle::Image;
        this->bindingNavigatorMoveFirstItem->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"bindingNavigatorMoveFirstItem.Image")));
        this->bindingNavigatorMoveFirstItem->Name =
L"bindingNavigatorMoveFirstItem";
        this->bindingNavigatorMoveFirstItem->RightToLeftAutoMirrorImage =
true;
        this->bindingNavigatorMoveFirstItem->Size = System::Drawing::
Size(23, 24);
        this->bindingNavigatorMoveFirstItem->Text = L"Move first";
        //
        // bindingNavigatorMovePreviousItem
        //
        this->bindingNavigatorMovePreviousItem->DisplayStyle =
System::Windows::Forms::ToolStripItemDisplayStyle::Image;
        this->bindingNavigatorMovePreviousItem->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"bindingNavigatorMovePreviousItem.Image")));
        this->bindingNavigatorMovePreviousItem->Name =
L"bindingNavigatorMovePreviousItem";
```

```
        this->bindingNavigatorMovePreviousItem->RightToLeftAutoMirrorImage =
true;
        this->bindingNavigatorMovePreviousItem->Size =
System::Drawing::Size(23, 24);
        this->bindingNavigatorMovePreviousItem->Text = L"Move previous";
        //
        // bindingNavigatorSeparator
        //
        this->bindingNavigatorSeparator->Name = L"bindingNavigatorSeparator";
        this->bindingNavigatorSeparator->Size = System::Drawing::Size(6, 27);
        //
        // bindingNavigatorPositionItem
        //
        this->bindingNavigatorPositionItem->AccessibleName = L"Position";
        this->bindingNavigatorPositionItem->AutoSize = false;
        this->bindingNavigatorPositionItem->Name =
L"bindingNavigatorPositionItem";
        this->bindingNavigatorPositionItem->Size = System::Drawing::Size(50, 27);
        this->bindingNavigatorPositionItem->Text = L"0";
        this->bindingNavigatorPositionItem->ToolTipText = L"Current position";
        //
        // bindingNavigatorSeparator1
        //
        this->bindingNavigatorSeparator1->Name =
L"bindingNavigatorSeparator1";
        this->bindingNavigatorSeparator1->Size = System::Drawing::Size(6, 27);
        //
        // bindingNavigatorMoveNextItem
        //
        this->bindingNavigatorMoveNextItem->DisplayStyle =
System::Windows::Forms::ToolStripItemDisplayStyle::Image;
        this->bindingNavigatorMoveNextItem->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"bindingNavigatorMoveNextItem.Image")));
        this->bindingNavigatorMoveNextItem->Name =
L"bindingNavigatorMoveNextItem";
        this->bindingNavigatorMoveNextItem->RightToLeftAutoMirrorImage = true;
        this->bindingNavigatorMoveNextItem->Size = System::Drawing::Size(23, 24);
        this->bindingNavigatorMoveNextItem->Text = L"Move next";
        //
        // bindingNavigatorMoveLastItem
        //
        this->bindingNavigatorMoveLastItem->DisplayStyle =
System::Windows::Forms::ToolStripItemDisplayStyle::Image;
        this->bindingNavigatorMoveLastItem->Image =
(cli::safe_cast<System::Drawing::Image^  >(resources-
>GetObject(L"bindingNavigatorMoveLastItem.Image")));
        this->bindingNavigatorMoveLastItem->Name =
L"bindingNavigatorMoveLastItem";
```

```
        this->bindingNavigatorMoveLastItem->RightToLeftAutoMirrorImage = true;
        this->bindingNavigatorMoveLastItem->Size = System::Drawing::Size(23, 24);
        this->bindingNavigatorMoveLastItem->Text = L"Move last";
        //
        // bindingNavigatorSeparator2
        //
        this->bindingNavigatorSeparator2->Name =
L"bindingNavigatorSeparator2";
        this->bindingNavigatorSeparator2->Size = System::Drawing::Size(6, 27);
        //
        // button1
        //
        this->button1->Location = System::Drawing::Point(94, 206);
        this->button1->Name = L"button1";
        this->button1->Size = System::Drawing::Size(122, 62);
        this->button1->TabIndex = 2;
        this->button1->Text = L"Загрузить данные из БД в таблицу";
        this->button1->UseVisualStyleBackColor = true;
        this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
        //
        // dataGridView1
        //
        this->dataGridView1->AutoGenerateColumns = false;
        this->dataGridView1->ColumnHeadersHeightSizeMode =
System::Windows::Forms::DataGridViewColumnHeadersHeightSizeMode::AutoSize;
        this->dataGridView1->Columns->AddRange(gcnew cli::array<
System::Windows::Forms::DataGridViewColumn^  >(6) {this->Column1,
              this->column2DataGridViewTextBoxColumn, this-
>column3DataGridViewTextBoxColumn, this->column4DataGridViewTextBoxColumn,
this->column5DataGridViewTextBoxColumn,
              this->column6DataGridViewTextBoxColumn});
        this->dataGridView1->DataSource = this->bindingSource1;
        this->dataGridView1->Location = System::Drawing::Point(12, 30);
        this->dataGridView1->Name = L"dataGridView1";
        this->dataGridView1->RowTemplate->Height = 24;
        this->dataGridView1->Size = System::Drawing::Size(614, 170);
        this->dataGridView1->TabIndex = 3;
        //
        // button2
        //
        this->button2->Location = System::Drawing::Point(266, 206);
        this->button2->Name = L"button2";
        this->button2->Size = System::Drawing::Size(108, 62);
        this->button2->TabIndex = 4;
        this->button2->Text = L"Вернуть  данные в БД";
        this->button2->UseVisualStyleBackColor = true;
        this->button2->Click += gcnew System::EventHandler(this,
&Form1::button2_Click);
```

```
        //
        // button3
        //
        this->button3->Location = System::Drawing::Point(451, 206);
        this->button3->Name = L"button3";
        this->button3->Size = System::Drawing::Size(75, 62);
        this->button3->TabIndex = 5;
        this->button3->Text = L"Выход";
        this->button3->UseVisualStyleBackColor = true;
        this->button3->Click += gcnew System::EventHandler(this,
&Form1::button3_Click);
        //
        // Column1
        //
        this->Column1->DataPropertyName = L"Column1";
        this->Column1->HeaderText = L"Column1";
        this->Column1->Name = L"Column1";
        //
        // column2DataGridViewTextBoxColumn
        //
        this->column2DataGridViewTextBoxColumn->DataPropertyName = L"Column2";
        this->column2DataGridViewTextBoxColumn->HeaderText = L"Column2";
        this->column2DataGridViewTextBoxColumn->Name =
L"column2DataGridViewTextBoxColumn";
        //
        // column3DataGridViewTextBoxColumn
        //
        this->column3DataGridViewTextBoxColumn->DataPropertyName = L"Column3";
        this->column3DataGridViewTextBoxColumn->HeaderText = L"Column3";
        this->column3DataGridViewTextBoxColumn->Name =
L"column3DataGridViewTextBoxColumn";
        //
        // column4DataGridViewTextBoxColumn
        //
        this->column4DataGridViewTextBoxColumn->DataPropertyName = L"Column4";
        this->column4DataGridViewTextBoxColumn->HeaderText = L"Column4";
        this->column4DataGridViewTextBoxColumn->Name =
L"column4DataGridViewTextBoxColumn";
        //
        // column5DataGridViewTextBoxColumn
        //
        this->column5DataGridViewTextBoxColumn->DataPropertyName = L"Column5";
        this->column5DataGridViewTextBoxColumn->HeaderText = L"Column5";
        this->column5DataGridViewTextBoxColumn->Name =
L"column5DataGridViewTextBoxColumn";
        //
        // column6DataGridViewTextBoxColumn
        //
        this->column6DataGridViewTextBoxColumn->DataPropertyName = L"Column6";
        this->column6DataGridViewTextBoxColumn->HeaderText = L"Column6";
```

```
        this->column6DataGridViewTextBoxColumn->Name =
L"column6DataGridViewTextBoxColumn";
        //
        // Form1
        //
        this->AutoScaleDimensions = System::Drawing::SizeF(8, 16);
        this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
        this->ClientSize = System::Drawing::Size(654, 280);
        this->Controls->Add(this->button3);
        this->Controls->Add(this->button2);
        this->Controls->Add(this->dataGridView1);
        this->Controls->Add(this->button1);
        this->Controls->Add(this->bindingNavigator1);
        this->Name = L"Form1";
        this->Text = L"Form1";
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>(this->bindingSource1))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>(this->dataSet1))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>(this->dataTable1))->EndInit();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>(this->bindingNavigator1))->EndInit();
        this->bindingNavigator1->ResumeLayout(false);
        this->bindingNavigator1->PerformLayout();
        (cli::safe_cast<System::ComponentModel::ISupportInitialize^
>(this->dataGridView1))->EndInit();
        this->ResumeLayout(false);
        this->PerformLayout();

      }
#pragma endregion
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
   this->dataSet1->Clear();  //Очистка буфера перед чтением в него
                             //таблицы
   this->oleDbDataAdapter1->Fill(dataTable1);
 }
private: System::Void button2_Click(System::Object^  sender, System::EventArgs^ e)
 {
           //Сохранить данные в БД
    this->oleDbDataAdapter1->Update(dataSet1->Tables["Авторы"]);

 }

private: System::Void button3_Click(System::Object^  sender, System::EventArgs^ e)
 {
    this->Close();
 }
};
}
```

C/C++ Microsoft Visual

ГЛАВА 13

Управление исключительными ситуациями

В процессе исполнения приложений могут возникнуть так называемые исключения (исключительные ситуации) — это аномальные явления (например, деление на ноль, превышение размерности массивов и т. п.), которые требуют от программы определенной реакции. И в этом случае либо вы в программе как-то реагируете на отклонения от нормального режима, либо система сама отреагирует: обработает по-своему такую ситуацию, выдаст вам сообщение и, как правило, завершит выполнение программы. В таких случаях желательно не дожидаться реакции системы и самому предпринимать необходимые меры, т. е. управлять возникающими в процессе решения вашей задачи проблемами.

Язык C++ обеспечивает в этом смысле встроенную поддержку для обработки возникающих исключений. Управление обработкой исключений состоит в реакции программы на возникновение неожидаемых ею событий. Кроме специальных операторов, перехватывающих исключения, существует особый класс `System::Exceptions`, содержащий необходимые свойства и методы, помогающие обрабатывать исключительные ситуации. Например, в классе имеется свойство `Message`, содержащее описание возникшего исключения, или свойство `Source`, содержащее имя источника (приложения или объекта), где возникла ошибка. Можно воспользоваться свойством `HelpLink`, которое выдаст вам ссылку на соответствующий Help-файл, описывающий возникшую ситуацию, или свойством `HResult`, в котором находится числовой код возникшей ошибки.

В C++ процесс обработки исключительных ситуаций состоит в возникновении (говорят "в выбрасывании" — throwing) ситуаций и в последующем их захвате (catching) для обработки.

Операторы *try*, *catch* и *throw*

С помощью этих операторов и обрабатываются исключительные ситуации. Синтаксис объявления операторов таков:

```
try {
   /*
    здесь находится участок программы, в котором могут
    возникнуть исключительные ситуации
  */
    }
```

```
[ catch (объявление исключения)
 {
   /*
    операторы для обработки возникшего в try-блоке
    исключения
   */
  }
[catch (объявление исключения)
 {
   // операторы, обрабатывающие другой тип исключения
 } ] ... ]
```

`throw выражение`

Оператор `throw` выбрасывает для последующей обработки возникающее исключение.

Тело оператора `catch` — это фактически и есть обработчик исключения. Исключение выбрасывается оператором `throw`, захватывается оператором `catch` и в его теле обрабатывается. Объявление исключения в операторе `catch` определяет тип исключения, которое станет обрабатываться в `catch`. Этим типом может служить конкретный тип данного, определенный в языке, в том числе и класс. Если в объявлении исключения задано многоточие (`...`), то это означает, что оператор `catch` станет обрабатывать любой тип возникающего исключения. Такой `catch` должен быть последним в `try`-блоке. По синтаксису вы заметили, что блоков `catch` может быть более одного: если вы хотите отлавливать конкретные исключения, то сформируйте для каждого из них свой `catch`, в котором задайте тип обрабатываемого исключения.

Необязательный операнд (выражение) в `throw` имеет смысл операнда в операторе `return`: тоже возвращает нечто. Более детально процесс обработки исключительной ситуации состоит в следующем:

1. Система исполняет операторы программы в обычном режиме, не обращая внимания на наличие операторов обработки исключительных ситуаций.
2. Как только в момент исполнения участка программы в теле `try` возникает исключительная ситуация, из выражения, находящегося в операторе `throw` (т. е. выражение в этом операторе должно всегда присутствовать), создается объект класса `Exceptions` (просто вызывается конструктор этого класса) и тут же отыскивается наивысший по иерархии оператор `catch`, который может обработать исключение возникшего типа (или любого типа). Если подходящего по возникшему типу исключения оператора `catch` не находится, то отыскивается ближайший следующий `try`-блок (как вы уже, наверно, догадались, `try`-блоков может быть много, потому что исключительные ситуации могут возникать в различных участках вашей программы и каждый из таких подозрительных участков надо охватить `try`-блоком). Этот процесс продолжается, пока самый крайний `try`-блок не проверится.

3. Если все же подходящий `try`-блок не будет обнаружен или в момент такого поиска возникнет новое исключение, вызывается специальная функция среды, которая завершает выполнение приложения. Если же подходящий `try`-блок найдется, то начнется обработка исключения соответствующим оператором `catch`.

Пример 1

В этом примере обнаруживается ошибка превышения длины вводимой с клавиатуры строки: для ее размещения выделено всего 10 байтов, а мы вводим более 10 байтов. Хотя функция `getline()` контролирует количество вводимых символов с помощью своего 2-го параметра `lim`, но он задан большим, чем 10, и поэтому данный контроль не срабатывает. Создадим консольную программу, текст которой приведен в листинге 13.1, результат работы программы показан на рис. 13.1.

Листинг 13.1

```
// 13.1_2011_ Исключения.cpp : main project file.

#include "stdafx.h"

using namespace System;
#include "stdafx.h"
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>      //for getch()
#include <stdlib.h>     //atoi(),atof()


#define eof -1          //Ctrl+z
#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*/

 int getline(char s[],int lim) throw(...)
/*
 Внутри функции могут возникать любые исключения:
 еще один вариант задания выброски исключения.
*/
  {
    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()
{
 char *buf;
 try
    {
      buf = new char[10]; //отвели место для строки из 10 символов
      getline(buf,200); //"Ошиблись": макс. длина строки задана
                        //в 200 символов
      throw 1;  //Этот оператор выбросит исключение, если оно
                //возникнет

     }
   catch(...) //обработчик исключения любого типа
    {
Console::WriteLine(L"Исключение!");
Console::ReadLine();  //Задержка экрана
    }
}
```

Рис. 13.1. Обработка любой исключительной ситуации

Пример 2

Этот пример развивает обработку исключительных ситуаций. Дополним предыдущую программу обработкой введенных символов. В частности, зададим оператор перевода 1-го символа введенной строки в число. Если этот символ будет буквой, должна возникнуть исключительная ситуация. Чтобы отловить это исключение, надо новый оператор поместить в свой блок `try`, для которого определить свой блок `catch` (получаем вложенность блоков обработки исключений). Текст программы показан в листинге 13.2, а результат — на рис. 13.2.

Листинг 13.2

```
// 13.2_2011_ Исключения.cpp : main project file.
#include "stdafx.h"
#include <stdio.h>      //for getchar(),putchar()
#include <conio.h>      //for getch()
```

```
#include <stdlib.h>     //atoi(),atof()
#define eof -1          //Ctrl+z
#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*/
 int 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()
{
 char *buf;
 try
    {
      buf = new char[10]; //отвели место для строки из 10 символов
      getline(buf,200); //"Ошиблись": макс. длина строки задана
                        //в 200 символов
      throw 1;  //Этот оператор выбросит исключение, если оно
                //возникнет

  try
       {
    int i=buf[0];
       }
        catch(int)
         {
           printf("Convertion Exception!\n");
         }
     }

   catch(...) //обработчик исключения любого типа (должен быть последним)
    {
printf("Exception!\n");
_getch();//задержка экрана
 }
}
```

Заметим, что первым сработал самый внутренний обработчик. Если бы преобразование в число было правильным, то сработал бы уже внешний обработчик (если бы было превышение количества введенных символов).

Рис. 13.2. Обработка преобразования данных

Классы типов исключений

При обработке исключений можно задавать классы типов исключений. Например, обрабатывать все исключительные ситуации из класса исключений, возникающих при выполнении арифметических операций и т. п. Все типы таких классов расположены в пространстве `System`. В табл. 13.1 представлены некоторые классы.

Таблица 13.1. *Классы типов исключений*

Класс	**Исключительная ситуация возникает**
`AccessViolationException`	Когда идет попытка чтения/записи в защищенной памяти
`AppDomainUnloadedException`	Когда идет попытка доступа к незагружаемой области приложения
`ApplicationException`	Когда в приложении возникает нефатальная ошибка
`ArgumentException`	Когда один из аргументов метода недействителен
`ArgumentNullException`	Когда методу передается нулевая ссылка, а метод ее не распознает как действительный аргумент
`ArgumentOutOfRangeException`	Когда значение аргумента выходит за границы, определенные методом
`ArithmeticException`	При ошибках в арифметических операциях и при преобразовании данных
`ArrayTypeMismatchException`	При попытке поместить элемент в массив, тип которого не совпадает с типом элемента
`DivideByZeroException`	Когда идет попытка деления числа на ноль
`EntryPointNotFoundException`	При попытке загрузить класс в случае отсутствия точки входа (например, в приложении отсутствует метод `main()`)
`Exception`	В момент выполнения приложения
`FieldAccessException`	При попытке доступа внутри класса к членам с атрибутами `private` или `protected`
`FormatException`	Когда обнаруживается нарушение формата аргумента, объявленного в методе

Таблица 13.1 *(окончание)*

Класс	Исключительная ситуация возникает
IndexOutOfRangeException	При выходе индекса массива за границы массива
InsufficientMemoryException	Из-за нехватки памяти
InvalidCastException	При неправильном преобразовании данных
InvalidOperationException	Когда для текущего состояния объекта вызов метода неверен
MemberAccessException	При неудачной попытке доступа к члену класса
MethodAccessException	При неудачной попытке доступа к private- или protected-методу внутри класса
MissingFieldException	При попытке динамического доступа к несуществующему полю класса
MissingMemberException	При попытке динамического доступа к несуществующему члену класса
MissingMethodException	При попытке динамического доступа к несуществующему методу класса
NotFiniteNumberException	При обработке несуществующего или бесконечного числа с плавающей точкой
NotSupportedException	При вызове метода, который не поддерживается, или при работе с потоком данных, когда требуемая функциональность не поддерживается
NullReferenceException	При попытке разыменовать нулевую ссылку
OutOfMemoryException	При нехватке памяти для продолжения работы программы
OverflowException	При превышении значений арифметических операций или операций преобразования в контролируемом контексте
PlatformNotSupportedException	При попытке запустить приложение на неподдерживаемой платформе
RankException	При попытке передачи массива с неверной размерностью в метод
StackOverflowException	При переполнении стека (слишком много вызовов методов)
TimeoutException	По истечении времени, отведенному для процесса или операции
TypeLoadException	При сбое в момент загрузки некоторого типа
TypeUnloadedException	При попытке доступа к незагруженному классу
UnauthorizedAccessException	При попытке неавторизованного доступа (операционная система отвергает доступ из-за ошибки ввода/вывода или из-за нарушения специального типа безопасности)
UriFormatException	При обнаружении неверного Uniform Resource Identifier (URI)

Пример 3

В этом примере мы покажем, как использовать класс, задающий определенный тип исключения. Для этого воспользуемся классом из только что приведенной таблицы — классом обработки исключительных ситуаций, возникающих при выполнении арифметических действий и преобразований — классом `ArithmeticException`.

Зададим такой алгоритм: возьмем массив целых чисел, датчик случайных чисел в интервале [1, 10] и организуем деление каждого элемента массива на полученное случайное число следующим образом: 1-й элемент массива делим на 1-е случайное число, 2-й — на 2-е и т. д. Для простоты мы взяли массив из трех элементов. Поместим в форму две кнопки, одна из которых будет запускать на выполнение наш алгоритм, а другая обеспечит выход из приложения. Текст обработчиков событий этих кнопок приведен в листинге 13.3, а результат расчетов — на рис. 13.3.

Листинг 13.3

```
private: System::Void button1_Click(System::Object^  sender, System::EventArgs^ e)
 {
   array <int ^> ^m = gcnew  array <int ^> (3) {1,2,3};
   array <int ^> ^k1 = gcnew  array <int ^> (1) {0}; //Для вывода
                                    //счетчика while в виде строки
/* разделим 1-й элемент массива m[] на 1-е случайное число,
   2-й — на 2-е и т. д.*/

 int j1;
 int i;
        int k=0;
  try
   {
     while(k < 1000)
 {
   for(i=0; i < m->Length; i++)
    {
      Random ^r =  gcnew Random(); //Формирование обращения
                                    //к датчику случайных чисел
      int j=r->Next(0,10);//выдает случайное число в интервале [0,10]

    switch(i)
     {
       case 0: j1= (int) m[i] / j; break;
       case 1: j1= (int) m[i] / j; break;
       case 2: j1= (int) m[i] / j; break;
       default: break;

      } //switch

   } //for
```

```
        k++;
 } //while

   return; //см. пояснение после листинга
     throw 1;
   } //try
    catch (ArithmeticException ^e) // Задаем тип исключений,
                                   //которые будут обрабатываться
   {
      k1[0]=k; //для легкого преобразования int в String ^
    this->textBox1->Text="Системное сообщение:" +
    e->Message +
    " Шаг цикла=" +
    Convert::ToString(k1[0]);

     } //catch
   } //Button1_click

private: System::Void button2_Click(System::Object^  sender, System::EventArgs^
e)
 {
 this->Close();
 }
```

Рис. 13.3. Исключение при делении на ноль

Пояснение

Кнопку **Пуск** надо нажимать, пока не поймается случайная ситуация деления на ноль. Основные пояснения сделаны в тексте программы.

А сейчас обратим внимание на два момента:

- как формировать `try`-блок, где находится оператор `throw`, который выдает исключение;
- как избежать неприятностей преобразования числовых типов данных в строку при работе в режиме CLR, в котором мы создали приложение.

Что касается первого вопроса, то тут надо иметь в виду следующее: оператор `throw` кое-куда ставить нельзя, ибо он выбрасывает системе исключение не только когда наступает действительная причина его возникновения, но и тогда, когда он сам выполнится. В обычном режиме работы программы этот оператор не должен попадаться для исполнения по ходу выполнения вашей программы, иначе он сам прервет выполнение программы системным образом (на `catch` ваша программа в этом случае не попадет и сработает системный вариант). Поэтому `try`-блок надо формировать так, чтобы содержащийся в нем `throw` не выполнялся (тогда он сработает в момент появления в вашей программе исключительной ситуации, и управление будет передано на соответствующий `try`-блоку оператор `catch`).

Что касается второго вопроса, то здесь правило такое: среда CLR — среда специфическая и работает со своими типами данных (с объектами), а не с теми, к которым мы привыкли (`int`, `float`, `char` и т. п.). Одними из объектов являются managed-массивы (`array`-типы) и их элементы. Объекты имеют метод `ToString()`, переводящий объект в строку символов. Числовому объекту легко присвоить просто обычное число, а объект уже можно перевести в строку и потом вывести эту строку.

Функции, выдающие исключения

В соответствии с моделью создания и обработки исключительных ситуаций, принятой в данной версии C++, компилятор предполагает, что исключения могут выдаваться только от выполнения оператора `throw` или от вызова функции. Функция будет выдавать исключения, если ее заголовок дополнен одной из следующих спецификаций:

- `throw()` — функция выдает пустое исключение (т. е. не выдает никакого исключения);
- `throw(...)` — функция выдает любое исключение, которое возникнет при ее вызове;
- `throw(type)` — функция выдает исключение заданного типа.

В листинге 13.4 приводится функция `f1()`, выдающая исключение, когда ее аргумент четное число. Результат работы представлен на рис. 13.4.

Листинг 13.4

```
// 13.4_2011 Исключения.cpp : main project file.

#include "stdafx.h"
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>

void f1(int i)throw(int)
{
```

```
  if(i%2 == 0)//четное число (остаток от деления равен 0)
  throw 1;
}
void f2(int j) throw(...)
{
 f1(j);
}
//-----------------------------
int main()
{
 for(int j=0; j < 10; j++)
  {
try
{
f2(j);
}
catch(...)
{
 printf("j=%d\n",j);
 _getch(); //задержка экрана
}
  } //for
}
```

Рис. 13.4. Числа, в интервале [0.10), на которых возникают исключения

C/C++ Microsoft Visual

ГЛАВА 14

Преобразование между нерегулируемыми и регулируемыми (режим CLR) указателями

На практике часто встречаются случаи, когда требуется переходить от обычных указателей к регулируемым и наоборот (еще говорят "переход от native-среды к managed-среде и наоборот"). Этот процесс называют *маршалингом*. Среда VC++ содержит специальную библиотеку, предназначенную для этих целей. Эту библиотеку можно использовать без так называемого `marshal_context Class`, однако некоторые преобразования требуют наличия этого класса. Другие преобразования используют функцию `marshal_as()`. В табл. 14.1 представлены поддерживаемые преобразования с учетом требования контекста, задаваемого классом `marshal_context Class`.

Таблица 14.1. *Преобразование между нерегулируемыми и регулируемыми указателями*

Из типа	**В тип**	**Marshal-метод**	**Какой файл надо подключать (Include file)**
`System::String^`	`const char *`	`marshal_context`	marshal.h
`const char *`	`System::String^`	`marshal_as`	marshal.h
`char *`	`System::String^`	`marshal_as`	marshal.h
`System::String^`	`const wchar_t*`	`marshal_context`	marshal.h
`const wchar_t *`	`System::String^`	`marshal_as`	marshal.h
`wchar_t *`	`System::String^`	`marshal_as`	marshal.h
`System::IntPtr`	`HANDLE`	`marshal_as`	marshal_windows.h
`HANDLE`	`System::IntPtr`	`marshal_as`	marshal_windows.h
`System::String^`	`BSTR`	`marshal_context`	marshal_windows.h
`BSTR`	`System::String^`	`marshal_as`	marshal.h
`System::String^`	`bstr_t`	`marshal_as`	marshal_windows.h
`bstr_t`	`System::String^`	`marshal_as`	marshal_windows.h

Таблица 14.1 *(окончание)*

Из типа	В тип	Marshal-метод	Какой файл надо подключать (Include file)
System::String^	std::string	marshal_as	marshal_cppstd.h
std::string	System::String^	marshal_as	marshal_cppstd.h
System::String^	std::wstring	marshal_as	marshal_cppstd.h
std::wstring	System::String^	marshal_as	marshal_cppstd.h
System::String^	CStringT<char>	marshal_as	marshal_atl.h
CStringT<char>	System::String^	marshal_as	marshal_atl.h
System::String^	CStringT<wchar_t>	marshal_as	marshal_atl.h
CStringT<wchar_t>	System::String^	marshal_as	marshal_atl.h
System::String^	CComBSTR	marshal_as	marshal_atl.h
CComBSTR	System::String^	marshal_as	marshal_atl.h

Маршалинг требует контекстного файла только в том случае, когда выполняется преобразование из управляемого типа в родной (native), нерегулируемый тип, и при этом native-тип, в который идет преобразование, не имеет деструктора для автоматического освобождения памяти от объекта. Вот тогда маршалинг-контекст разрушает размещенный объект native-типа своим деструктором. Поэтому преобразования, требующие маршалинг-контекста, действительны только до момента удаления контекста. Чтобы сохранить любое маршалинг-значение, вы должны скопировать его в свою собственную переменную. Если строке присвоено значение `NULL`, результат преобразования такой строки не гарантируется.

На практике для использования маршалинга достаточно подключить к вашей программе пространство имен:

```
using namespace System::Runtime::InteropServices
```

Пример 1. Перевод строки *String ^* в ASCII-строку

Текст программы показан в листинге 14.1, а результат — на рис. 14.1.

Рис. 14.1. Результат перевода native-строки в managed-строку

Листинг 14.1

```
// 14.1_2011 Нерег в Рег.cpp : main project file.

#include "stdafx.h"
#include <stdio.h>

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged
/*сигнал компилятору, что далее следуют unmanaged-функции, работающие с
обычными (native) переменными, которые станут размещаться в неуправляемой куче
(памяти), за переполнением которой надо следить самому — вовремя освобождать
от занимающих ее объектов
*/

void NativeTakesAString(const char* p)
{
   printf("(native) received '%s'\n", p);
}

#pragma managed
/*сигнал компилятору, что далее следуют  managed-функции, работающие со средой
CLR, в которой переменные станут размещаться в управляемой куче (памяти), за
переполнением которой не надо следить самому — среда CLR сама позаботится об ее
освобождении от занимаемых объектов
*/

int main() //array<System::String ^> ^args
{
  String^ s = gcnew String("sample string");
  IntPtr ip = Marshal::StringToHGlobalAnsi(s);
/*этот метод выделяет для s место в неуправляемой куче и указатель на это место
передает в структуру IntPtr. Указатель имеет тип void, т. е. его еще надо
настроить на конкретный тип данного*/

  const char* str = static_cast<const char*>(ip.ToPointer());
 /*кастинг указателя: настройка его на native-тип char* с помощью оператора
static_cast <type-id> (expression). Теперь указатель str типа char будет
настроен в неуправляемой куче на начало переменной s. Так как s – константа, то
у указателя str взят квалификатор const, говорящий о том, что значение, на
которое указывает указатель, изменять нельзя
 */

   Console::WriteLine("(managed) passing string...");
/*т. к. мы находимся в managed-методе main(), то и вывод идет по WriteLine())*/

   NativeTakesAString( str ); //вызов native-метода внутри managed-метода
```

```
   Marshal::FreeHGlobal( ip );
 /*этот метод освобождает память, выделенную StringToHGlobalAnsi(s): т. к. это
уже неуправляемая куча, то память надо освобождать самому */
  Console::ReadLine();
 }
```

Пример 2. Перевод ASCII-строки в строку *String* ^

Текст программы показан в листинге 14.2, а результат — на рис. 14.2.

Рис. 14.2. Результат преобразования managed-строки в native-строку

Листинг 14.2

```
// 14.2_2011 Прим 2.cpp : main project file.

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

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma managed

void ManagedStringFunc(char* s) /*здесь managed-функция получает
native-аргумент*/
{
   String^ ms = Marshal::PtrToStringAnsi(static_cast<IntPtr>(s));

/*метод PtrToStringAnsi() копирует native-строку s в managed-пространство,
преобразуя каждый символ ASCII в Юникод, и выдает указатель на
месторасположение новой строки в переменную s, которая преобразуется в тип
IntPtr. Затем этот тип преобразуется оператором  static_cast в тип String ^ */

   Console::WriteLine("(managed): received '{0}'", ms);
}

#pragma unmanaged
```

```
void NativeProvidesAString()  //это native-функция, которая вызывает
                              //managed-функцию
 {
   printf("(native) calling managed func...\n");
   ManagedStringFunc("test string");
}
#pragma managed

int main()//это managed-функция, которая вызывает native-функцию

{
   NativeProvidesAString();
   _getch();
}
```

Пример 3. Преобразование строки *String ^* в строку *wchar_t*

Тип `wchar_t` — это native-тип символа по таблице Юникода. Этот тип данного отличается от типа `char` тем, что символ кодируется двумя байтами, а не одним. Класс `String` тоже создает строку из Юникод-символов, однако по своему определению относится к managed-типу. Преобразование, которое мы рассматриваем, фактически переводит Юникод-строку из состояния `managed` в состояние `native`. А в этом состоянии со строкой уже можно работать, применяя обычный указатель `*`. Юникод-тексты можно вводить в файлы и читать из них соответственно функциями `fputws` и `fgetws`. Текст программы показан в листинге 14.3, а результат — на рис. 14.3.

Рис. 14.3. Результат преобразования Юникод-строки в native-строку

Листинг 14.3

```
// 14.3_2011 Прим 3.cpp : main project file.

#include "stdafx.h"
#include <stdio.h> //для printf()
#include <conio.h>  //для _getch()
#include <vcclr.h> //для PtrToStringChars()
```

```
using namespace System;
using namespace System::Runtime::InteropServices;

#pragma unmanaged

void NativeTakesAString(const wchar_t* p)
{
   printf("(native) recieved '%S'\n", p);
}

#pragma managed

int main()
{
   String^ s = gcnew String("test string");
   pin_ptr<const wchar_t> str = PtrToStringChars(s);
/*
   метод размещает s в управляемой куче и выдает native-указатель
   типа _const_Char_ptr на размещенный объект. Этот указатель
   с помощью оператора
   [cli::]pin_ptr<cv_qualifier type> var = &initializer;
   преобразуется в тип const wchar_t.
   Здесь:
   cv_qualifier — квалификатор const или квалификатор volatile.
   Квалификатор volatile показывает, что поле может модифицироваться
   многочисленными средами, выполняющимися в данный момент,
   и все изменения данного поля будут присутствовать в этом поле.
   Указатель типа pin_ptr по умолчанию имеет квалификатор volatile,
   поэтому в операторе применен квалификатор const,
   чтобы объект в куче не изменял своего значения, а не только
   место расположения.
   initializer — это ссылочный тип данного: элемент managed-массива
   или любого другого объекта, которому вы назначаете native-указатель.
   Метод PtrToStringChars() как раз и выдает такой указатель:
   native-указатель в управляемой (managed) куче.
   type — тип initializer'a. В нашем случае это wchar_t
   var — имя pin_ptr указателя. В нашем случае это str.
   То есть str — это уже native-указатель строки String ^
   в управляемой куче.
*/

   Console::WriteLine("(managed) passing string to native func...");
   /*вывод сообщения из managed-функции*/
   NativeTakesAString( str );
   /*вызов native-функции, которая выдаст native-сообщение*/
   _getch();
}
```

Здесь применен указатель `pin_ptr` (указатель от зашкаливания, как определяет его автор). Это внутренний указатель, который предотвращает объект (на который он указывает) от какого-либо перемещения в управляемой куче (памяти, с которой работает режим CLR): значение указателя не изменяется средой CLR. Такое условие необходимо при передаче адреса managed-объекта native-функции, потому что этот адрес не должен меняться во время вызова native-функции.

Пример 4. Преобразование строки *wchar_t* в строку *String* ^

Это преобразование — обратное приведенному в примере 3. Преобразование, которое мы рассматриваем, фактически переводит `native` Юникод-строку в состояние `managed`. А в этом состоянии со строкой уже можно работать, применяя managed-указатель `^`. Текст программы показан в листинге 14.4, а результат — на рис. 14.4.

Рис. 14.4. Результат преобразования `wchar_t` в `String ^`

Листинг 14.4

```
// 14.4_2011 Прим 4.cpp : main project file.

// 2008-Marshal  wchar_t to Unicod.cpp : main project file.

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

using namespace System;
using namespace System::Runtime::InteropServices;

#pragma managed

void ManagedStringFunc(wchar_t* s)
 {
   String^ ms = Marshal::PtrToStringUni((IntPtr)s);
/*
   копирует native-символы строки s в Юникоде в native-кучу,
   преобразовывая их в managed-строку, и выдает managed-указатель
   на место расположения строки.
```

```
   Аргумент метода по его определению должен иметь
   тип IntPtr (внутренний указатель), поэтому s преобразуется
   к этому типу.
*/
   Console::WriteLine("(managed) recieved '{0}'", ms);
}

#pragma unmanaged

void NativeProvidesAString()
{
  printf("(unmanaged) calling managed func...\n");
   ManagedStringFunc(L"test string");
}

#pragma managed

int main() {
   NativeProvidesAString();
   _getch();
}
```

Пример 5. Маршалинг native-структуры

В данном примере показана работа с native-структурой в managed-функции. Текст программы показан в листинге 14.5, а результат — на рис. 14.5.

Рис. 14.5. Маршалинг структуры

Листинг 14.5

```
// 14.5_2011 Прим 5.cpp : main project file.

#include "stdafx.h"
#include <stdio.h>
#include <math.h>

using namespace System;
using namespace System::Runtime::InteropServices;
```

```
#pragma unmanaged

//native-структура:
struct Location
{
   int x;
   int y;
};
//параметры функции — два экземпляра структуры:
double GetDistance(Location loc1, Location loc2)
 {
   printf("[unmanaged] loc1(%d,%d)", loc1.x, loc1.y);
   printf(" loc2(%d,%d)\n", loc2.x, loc2.y);

   double h = loc1.x — loc2.x;
   double v = loc1.y = loc2.y;
//квадратный корень из h2 + v2:
   double dist = sqrt( pow(h,2) + pow(v,2) );

   return dist;
 }
//параметр функции — указатель на структуру
void InitLocation(Location* lp)
 {
   printf("[unmanaged] Initializing location...\n");
   lp->x = 50;
   lp->y = 50;
}

#pragma managed
//работа со структурой в managed-функции:
int main()
 {
   Location loc1;
   loc1.x = 0;
   loc1.y = 0;

   Location loc2;
   loc2.x = 100;
   loc2.y = 100;
//вызов native-функции в managed-функции:
   double dist = GetDistance(loc1, loc2);
   Console::WriteLine("[managed] distance = {0}", dist);

   Location loc3;
//вызов native-функции в managed-функции
//в ее параметре указатель, поэтому берется адрес экземпляра структуры:
```

```
    InitLocation(&loc3);
    Console::WriteLine("[managed] x={0} y={1}", loc3.x, loc3.y);
    Console::ReadLine();

}
```

Пример 6. Работа с массивом элементов native-структуры в managed-функции

Текст программы показан в листинге 14.6, а результат — на рис. 14.6.

Рис. 14.6. Результат работы с массивом элементов native-структуры в managed-функции

Листинг 14.6

```
// 14.6_2011 Прим 6.cpp : main project file.

#include "stdafx.h"
#include <stdio.h>

using namespace System;
using namespace System::Runtime::InteropServices;

// unmanaged struct
struct ListStruct
```

```
{
   int count;
   double* item;
};

#pragma unmanaged

void UnmanagedTakesListStruct(ListStruct list)
{
   printf_s("[unmanaged] count = %d\n", list.count);
   for (int i=0; i<list.count; i++)
      printf_s("array[%d] = %.2f\n", i, list.item[i]);
}

#pragma managed

int main()
 {
   ListStruct list;
   list.count = 10;
   list.item =new double[list.count];
/*
   оператор new размещает объект в native-куче
   и возвращает указатель на этот объект.
   В данном случае в куче формируется одномерный массив
   из list.count элементов.
*/

   Console::WriteLine("[managed] count = {0}", list.count);
   //создается указатель на объект, формирующий случайные числа:
   Random^ r = gcnew Random(0);

/*
   инициализация массива list.item[] случайными числами
   из интервала [0,1], умноженными на 100
*/

   for (int i=0; i<list.count; i++)
    {
      list.item[i] = r->NextDouble() * 100.0;
      Console::WriteLine("array[{0}] = {1}", i, list.item[i]);
    }
   //вызов native-функции в managed-функции
   UnmanagedTakesListStruct(list);
   Console::ReadLine();
}
```

Пример 7. Доступ к символам в классе *System::String*

Это частный случай примера 3. Здесь интересна работа с native-указателем, показывающим на native-строку (строку "обычных" символов) в управляемой куче. Кавычки поставлены для того, чтобы указать на необычность ситуации: каждый символ занимает (в отличие от обычной строки символов типа `char`) два, а не один байт, т. к. находится в управляемой куче. Поэтому и признак конца — это не обычный символ '\0', а L'\0'.

Текст программы показан в листинге 14.7, а результат — на рис. 14.7.

Рис. 14.7. Доступ к символам `System::String`-строки

Листинг 14.7

```
// 14.7_2011 Прим 7.cpp : main project file.

#include "stdafx.h"
#include<vcclr.h>

using namespace System;
int main()
{
   String ^ mystring = "abcdefg";
/*Описание метода показано в примере 3 */
   interior_ptr<const Char> ppchar = PtrToStringChars( mystring );

/*цикл по строке символов: от нулевого (*ppchar) до признака конца строки
(L'\0')*/
   for ( ; *ppchar != L'\0'; ++ppchar )
      Console::Write(*ppchar);
   Console::Write(L'\n'); //переход на новую, после выданной, строку
   Console::ReadLine();
}
```

Пример 8. Преобразование *char ** в массив *System::Byte*

Текст программы показан в листинге 14.8, а результат — на рис. 14.8.

Рис. 14.8. Результат преобразования строки обычных символов в массив `System::Byte`

Листинг 14.8

```
// 14.8_2011 Прим 8.cpp : main project file.

#include "stdafx.h"
#include <string.h>  //for strlen()

using namespace System;
using namespace System::Runtime::InteropServices;

int main() {
   char buf[] = "Native String";
   int len = strlen(buf);  //длина строки

   array< Byte >^ byteArray = gcnew array< Byte >(len + 2);

   /*копирование символов char с  преобразованием указателя на них
   во внутренний указатель на них в управляемой куче.
   Преобразование идет в обычном С-стиле:*/
   Marshal::Copy((IntPtr)buf,byteArray, 0, len);

/*вывод (для проверки преобразования) элементов массива
 с использованием методов класса array:*/
   for ( int i = byteArray->GetLowerBound(0); i <= byteArray->GetUpperBound(0);
i++ )
   {
/*byteArray->GetValue(i) выдает указатель на объект а объектами
  являются элементы массива типа Byte. Поэтому идет преобразование типа.
  Но внутренний указатель в управляемой куче обладает свойствами
  обычного С++-указателя, поэтому можно применить операцию
  разыменования (*), чтобы получить обычный символ:*/
   char dc =  *(Byte^) byteArray->GetValue(i);
```

```
      /*преобразование (Char) обычного символа в Юникод-символ,
   чтобы воспользоваться выводом из класса Console:*/
      Console::Write((Char)dc);
   }
   //выдает стандартный символ окончания строки в стандартный
   //выходной поток:
   Console::WriteLine();
   Console::ReadLine();
}
```

Пример 9. Преобразование *System::String* в *wchar_t* * или *char* *

Метод `PtrToStringChars()` из Vcclr.h можно использовать для преобразования строки `System::String` в native-строку типа `wchar_t *` или `char *`. Метод возвращает указатель на Юникод-строку, т. к. строки CLR — это строки Юникода. Затем, как показано в примере, вы можете конвертировать "широкую" строку в обычную.

Текст программы показан в листинге 14.9, а результат — на рис. 14.9.

Рис. 14.9. Результат преобразования `System::String`-строки в `wchar_t *` и `char *`

Листинг 14.9

```
// 14.9_2011 Прим 9.cpp : main project file.

#include "stdafx.h"
#include < stdio.h >
#include < conio.h >
#include < stdlib.h >
#include < vcclr.h >  //for PtrToStringChars()

using namespace System;

int main()
{
   String ^str = "Hello from our friends!";
```

```
   /*преобразование String-строки в тип wchar_t в управляемой куче
   и формирование pin-указателя на новую строку (указателя-фиксатора),
   чтобы строка не переместилась в куче, пока будет идти
   вызов native-функции printf():*/
   pin_ptr<const wchar_t> wch = PtrToStringChars(str);
   printf_s("%S\n", wch);

   /*преобразование к типу char *:
   можно сразу перевести wchar_t * в char *, используя одну
   из функций-преобразователей WideCharToMultiByte() и wcstombs_s():*/

   size_t convertedChars = 0;
   size_t  sizeInBytes = ((str->Length + 1) * 2);
   errno_t err = 0;
   /*выделение памяти в неуправляемой куче
     под размер managed-строки ()
     ch будет на нее указывать:*/
   char    *ch = (char *)malloc(sizeInBytes);

  err = wcstombs_s(&convertedChars, //кол-во преобразуемых символов
  ch, sizeInBytes, //адрес, куда они станут записываться
  wch,             //адрес, откуда они станут записываться
sizeInBytes);    //количество переписываемых байтов
   if (err != 0)
      printf_s("wcstombs_s  failed!\n");

    printf_s("%s\n", ch); //вывод переписанной строки
       _getch();
}
```

Пример 10. Преобразование *String* в *string*

Фирма Microsoft ввела в C++ типы строк `string` и `wstring`. На самом деле — это синонимы класса `basic_string`, введенные через `typedef`. Первый тип относится к классу, строки которого (как элементы класса) относятся к типу `char`, а второй тип — к классу, строки которого (как элементы класса) относятся к типу `wchar_t`. В табл. 14.2 приведены операторы для работы со строками указанных типов.

Таблица 14.2. *Операторы для работы со строками типов* `string` *и* `wstring`

Оператор	Описание
+	Сцепляет две строки
!=	Проверяет, не равны ли две строки-операнда
==	Проверяет, равны ли две строки-операнда

Таблица 14.2 *(окончание)*

Оператор	Описание
<	Проверяет, меньше ли строка слева от знака оператора строки справа от знака оператора
<=	Проверяет, меньше или равна строка слева от знака оператора строки справа от знака оператора
<<	Вставляет строку в выходной поток (аналог `cout` при C++ вводе/выводе)
>	Проверяет, больше ли строка слева от знака оператора строки справа от знака оператора
>=	Проверяет, больше или равна ли строка слева от знака оператора строки справа от знака оператора
>>	Извлекает строку из выходного потока (аналог `cin` при C++ вводе/выводе)
`swap`	Если выполнить оператор `swap(m1, m2);` с этой функцией, то в `m1` будет содержимое `m2`, а в `m2` — содержимое `m1`
`getline`	Извлекает строку из входного потока
`c_str`	Преобразует `string`-строку в `C`-строку

Для работы со строками в программе надо выполнить операторы:

```
#include <string>
```

и

```
using namespace std;
```

Пример программы, использующей оператор `!=`, приведен в листинге 14.10, а результат показан на рис. 14.10.

Листинг 14.10

```
// 14.10_2011 Прим 10.cpp : main project file.

#include "stdafx.h"
#include <string>
#include <conio.h>
#include <iostream>

using namespace std ;
void trueFalse(int x)
{
  cout << (x? "True": "False") << endl;
}

int main()
{
   string S1="ABC";
```

```
    char CP1[]="ABC";
    char CP2[]="DEF";
    char CP3[]="abc";

    cout << "S1 is " << S1 << endl;
    cout << "CP1 is " << CP1 << endl;
    cout << "CP2 is " << CP2 << endl;
    cout << "CP3 is " << CP3 << endl;

    cout << "S1!=CP1 returned ";
    trueFalse(S1!=CP1);  // False

    cout << "S1!=CP2 returned ";
    trueFalse(S1!=CP2);  // True

    cout << "S1!=CP3 returned ";
    trueFalse(S1!=CP3);  // True

    cout << "CP1!=S1 returned ";
    trueFalse(CP1!=S1);  // False

    cout << "CP2!=S1 returned ";
    trueFalse(CP2!=S1);  // True

    cout << "CP3!=S1 returned ";
    trueFalse(CP3!=S1);  // True

    _getch();
}
```

Строки `String` — это последовательности Юникод-символов.

Программа, использующая маршалинг для преобразования строк типа `String` в строки типа `string` и `wstring`, приведена в листинге 14.11, а результат ее работы показан на рис. 14.11.

Рис. 14.10. Результат работы программы, указанной в листинге 14.10

Рис. 14.11. Результат работы со строками `String`, `string` и `wstring`

Листинг 14.11

```
// 14.11_2011 Прим 11.cpp : main project file.

#include "stdafx.h"
#include <string>   //здесь операторы для string и wstring
#include <conio.h>
#include <iostream> //для C++ вывода

using namespace System;
using namespace std;  //здесь находятся данные о типах string и wstring
using namespace Runtime::InteropServices;

//----------------- функция копирует s в os типа string --------------------
void MarshalString ( String ^ s, string & os )
 {
   /*Маршал-метод копирует содержимое managed-строки
    в неуправляемую память (кучу) и выдает указатель типа void
    для его последующего преобразования в необходимый тип
   (в данном случае в const char*): */
   const char* chars =
      (const char*)(Marshal::StringToHGlobalAnsi(s)).ToPointer();
   /*os — указатель на строку с символами char:*/
   os = chars;

   /*освобождение памяти в неуправляемой куче, которую занимает строка:*/
   Marshal::FreeHGlobal(IntPtr((void*)chars));
}
//----------------- функция копирует s в os типа wstring --------------
void MarshalString ( String ^ s, wstring & os )
{
/*Маршал-метод копирует содержимое managed-строки
  в неуправляемую память (кучу) и выдает указатель типа void
  для его последующего преобразования в необходимый тип
  (в данном случае в const wchar_t*): */
   const wchar_t* chars =
      (const wchar_t*)(Marshal::StringToHGlobalUni(s)).ToPointer();
   os = chars; //теперь можно присвоить, т.к. переменные одного типа

   /*освобождение памяти в неуправляемой куче, которую занимает строка:*/
   Marshal::FreeHGlobal(IntPtr((void*)chars));
}

int main()
 {
   string a = "test";
   wstring b = L"test2";  //"широкая" строка (по 2 байта на символ)
   String ^ c = gcnew String("abcd");
```

```
   cout << a << endl; //вывод строки a
   MarshalString(c, a);  //перезапись c в a
   c = "efgh";
   MarshalString(c, b); //перезапись c в b
   cout << a << endl;   //вывод a
   wcout << b << endl;  //вывод b
   _getch();
}
```

Пример 11. Преобразование *string*-строки в *String*-строку

Текст программы показан в листинге 14.12, а результат — на рис. 14.12.

Рис. 14.12. Результат преобразования `string` в `String`

Листинг 14.12

```
// 14.12_2011 Прим 11.cpp : main project file.

#include "stdafx.h"
#include <string>
#include <iostream>

using namespace System;
using namespace std;

int main()
 {
   string str = "test";
   //можно записать и так, используя конструктор:
   //string str = string("test");
   cout << str << endl;
   /*метод c_str() класса string преобразует string-строку
   в обычную C-строку с символом '\0'  в качестве признака конца строки.
   string-строка такого признака конца не имеет, и этот символ может быть
   обычным ее символом. string-строка — это аналог AnsiString-строки
   в C++ Builder*/
```

```
  String^ str2 = gcnew String(str.c_str());
  Console::WriteLine(str2);
  Console::ReadLine();
 }
```

Пример 12. Объявление дескрипторов в native-типах

Дескрипторами называют указатели в среде CLR. Именно они указывают на объект в управляемой куче. Напрямую нельзя объявить дескриптор в native-типе. Например, в native-функции вы не можете сделать объявление типа такого:

```
String ^s;
```

Компилятор вам выдаст ошибку. Файл vcclr.h содержит специальный настраиваемый шаблон `gcroot`, позволяющий ссылаться на CLR-объекты из C++ кучи, т. е. объекты из неуправляемой кучи могут ссылать на объекты из управляемой кучи. Тем самым устанавливается связь между различными средами. При этом вам позволяется использовать дескриптор в native-типы (например, в функции) и трактовать его как основной тип.

Шаблон `gcroot` создан на основе класса:

```
System::Runtime::InteropServices::GCHandle,
```

который обеспечивает дескрипторами объекты в управляемой куче.

Отметим, что сами дескрипторы автоматически удаляются деструктором класса `gcroot` только тогда, когда они больше не используются. Их нельзя удалять вручную. Если же вы создаете gcroot-объект в native-куче (т. е. в неуправляемой), то должны сами вызвать оператор `delete` для освобождения ресурса. В режиме исполнения программы поддерживается постоянная связь между дескриптором и CLR-объектом, на который он указывает. Если объект по тем или иным причинам перемещается в куче, дескриптор всегда возвращает новый адрес объекта. Переменная не может получить pin-указатель (предохраняющий объект от перемещения в такой куче), пока она назначена шаблону `gcroot`.

Текст программы показан в листинге 14.13, а результат — на рис. 14.13.

Рис. 14.13. Работа с дескриптором в native-памяти

Листинг 14.13

```
// 14.13_2011 Прим 12.cpp : main project file.

#include "stdafx.h"
#include <vcclr.h>

using namespace System;

// reference_to_value_in_native.cpp
// compile with: /clr

public value struct V //CLR-структура
{
   String ^str;
};

class Native  //native-класс
{
public:
 /*член native-класса — дескриптор v_handle:*/
   gcroot< V^ > v_handle;
};

int main() //managed-функция
{
   Native native; //native-переменная, объявленная в managed-функции
   V v;           //managed-переменная

/*дескриптору присваивается значение v,
  т. е. формируется ссылка на managed-структуру,
  из которой теперь можно извлекать ее элементы:*/
   native.v_handle = v;
   native.v_handle->str = "Hello to all";
   Console::WriteLine("String in V: {0}", native.v_handle->str);
   Console::ReadLine();
}
```

Пример 13. Работа с дескриптором в native-функции

Текст программы показан в листинге 14.14, а результат — на рис. 14.14.

Рис. 14.14. Результат работы managed-типа в native-функции

Листинг 14.14

```
// 14.14_2011 Прим 13.cpp : main project file.

#include "stdafx.h"
#include "gcroot.h"
#include <conio.h>

using namespace System;

#pragma managed
class StringWrapper //managed-класс
{
private:
   gcroot<String ^ > x;

public:
/*метод присваивает private-члену x класса
  (т. е. дескриптору) адрес строки "ManagedString"
  в управляемой куче:*/
   StringWrapper()
    {
      String ^ str = gcnew String("ManagedString");
      x = str;
   }
/*метод-член класса присваивает переменной targetStr
  значение дескриптора x, который указывает
  на строку  "ManagedString", и выводит эту строку:*/
   void PrintString()
   {
      String ^ targetStr = x;
      Console::WriteLine("StringWrapper::x == {0}", targetStr);
   }
};
#pragma unmanaged
int main() //native-функция
{
   StringWrapper s;  //s-переменная managed-типа
   s.PrintString();  //печать строки "ManagedString"
  _getch();
}
```

Предметный указатель

#

#include 25

A

ASCII 28, 37, 80, 89
Atof 83
Atoi 83

B

Bool 87
Break 102

C

Case 102
Cerr 189
Char и Int 40
CheckBox 338
CheckedkListBox 321
Cin 189
CLR 16, 107
CLS 245
ColorDialog 407
ComboBox 298
Continue 103
Cout 189

D

DataSet 423
DateTimePicker 357
DialogResult 251
Double 45

E

Else 50
Else-If 96
Exit 65, 170

F

Fclose 166
Feof 169
Fgetc 166
Fgets 166
FILE 165
FontDialog 407
Fopen 165
Foreign key 419
Form 201
Fprintf 168
Fputc 166
Fputs 166
Friend 147
Fscanf 168
Fseek 167
Fstream 178
Ftell 168

G

Gcnew 108
Getch 23
Getchar 37
Gets 178
Goto 104
GroupBox 342

H

Header file 25
Hide 208

I

If 47

If-Else 95
Ifstream 178
ImageList 253
Ios 178
Istream 178

K

KeyDown 263

L

Label 258
LinkLabel 343
ListBox 288
ListView 274
Long 43

M

Main() 17, 23, 25, 28, 33, 35
Malloc 106
MaskedTextBox 317
MenuStrip 266
MS SQL Server 414

N

NET Framework, 245
NULL 106

O

OpenFileDialog 395
Ostream 178
Ofstream 178

P

PictureBox 354
PrintDialog 408
Printf 28, 33, 173
ProgressBar 394
Properties 201
Putchar 37
Puts 178

R

RadioButton 338
ReadMe.txt 23
Rewind 168
Recent 20

S

Save All 28
SaveFileDialog 401
Scanf 176
ShortcutKeys 270
Sizeof 189
Solution 19
Solution (решение) 19
Solution Explorer 20
Sprintf 79, 177
Sscanf 178
Static 70
Stdafx.cpp 23
Stdafx.h 23
Stderr 172
Stdin 172
Stdout 172
Strcat 80
Strcmp 80
Strcmpi 80
Strcpy 79
Strlen 80
Switch 100

T

TabControl 373
TButton 250
TextBox 259
Timer 390
TLabel 258
TPanel 255
Trigger 413
Typedef 121

V

Virtual 147

W

WebBrowser 282
While 29, 32
Windows Form 1, 197

А

Абстрактные классы 158
Адрес первого элемента массива 61
Арифметические операции 33
Атрибут unsigned 43

Б

Битовые поля 138
База данных 413

В

Ввод и вывод
◊ в С 165
◊ в С++ 178
Ввод строки символов с клавиатуры 59
Вкладка
◊ Property Pages 207
◊ Events 205
◊ Toolbox 216
Включение блока на языке ассемблера 75
Внешний ключ 419, 420
Внешняя переменная 66
Возврат к структуре стола по умолчанию 10
Выбор индексов 420
Вызов функции 72
Выпадающие меню 8
Выражение 35
Вьюеры 415

Г

Главная функция 19
Главное окно 7
Группировка набора окон 9

Д

Движение окрашенных линий 390
Двоичное дерево 133
Деструктор 144
◊ класса 156
Дизайнер форм 216
Добавление новых форм к проекту 220
Домашний телефонный справочник 324

З

Заголовок функции 72
Заголовочная часть оператора For 103
Запуск компилятора 23
Знак "присвоить" 31
Значения параметров функции по умолчанию 73

И

Имена и типы переменных 30
Индекс 420
◊ массива 52
Инициализация массива 55
Инкапсуляция 144
Интерфейсные функции 163
Использование шаблонов функций 76

К

Как создать свой внешний файл 69
Категории памяти 139
Класс 141
◊ Form 216
Классы 143
◊ и структуры 156
◊ -компоненты 144
Клиенты 414
Ключевое слово inline 74
Кнопки быстрого вызова 8
Кодирование по Unicode 37
Компиляция и сборка проекта 15
Компоненты, форма и ее модули 146
Консольное приложение 15, 17
Константы 88
◊ символьного типа 88
Конструктор 144
◊ класса 153
Контекстное меню редактора кода 210
Контекстное меню формы 219
Концепция объектно-ориентированного программирования 141
Куча 149

Л

Логическая операция && 50
Логическая операция || 50
Логические операции 90
Локальные БД 414
Локальные переменные 66

М

Манипуляторы и функции стандартного ввода/вывода в С++ 190

Маршалинг 471
Маршаллизация 108
Маска 317
Массив 51
Массивы указателей 117
Место описания функции в программе 59
Метка 104
Метод
◊ двоичного поиска 127
◊ половинного деления 96
Методы класса 142
Многомерные массивы 54
Модель базы данных 415

Н

Наборы данных 423
Наследование 145
Настройка редактора кода 212
Некоторые методы формы 235
Некоторые файлы проекта 202
Нерегулируемые указатели 107
Нормализации 415
Нумерация битов в числе 85

О

О справочной системе Help 13
Область действия
◊ идентификатора 74
◊ переменной 69
Обращение к элементам структур 122
Объект 141
Объектно-ориентированное программирование 141
Объявление и определение функции 73
Ограничение доступа к элементам меню 270
Одномерные массивы 51
Окно
◊ Properties 204
◊ сведений об объекте 204
Оператор
◊ #define 35
◊ -- 44
◊ ++ 44
◊ "Присвоить" (особенности) 50
◊ Do…While 83
◊ New и Delete 149
◊ For 34
◊ Return 58, 61
◊ Switch 102
Оператор в C/C++ 23
Операторы SQL 421
Операторы и блоки 95
Операция
◊ \\ 47
◊ << и >> 179
◊ отношения 89, 90
◊ "запятая" 63
◊ деления 33
◊ отрицания 90
Операции и выражения присваивания 93
Операции над указателями 111
Определение функции 72
Опции окна 10
Организация работы с множеством форм 221
Основные функции для работы с файлами в C 166

П

Палитра компонентов 217
Первичный ключ 418, 419
Перегрузка функций 75
Передача функции адресов 59
Перечислимый тип данных 86
Побитовые логические операции 92
Подключение библиотечного и собственного файла к программе 67
Подсчет
◊ количества слов в файле 48
◊ количества строк в файле 47
◊ символов в файле 42, 44
Полиморфизм 146
Поля класса 142
Преобразование числовых данных 249
Преобразования типов данных 91
Признак
◊ комментария 39
◊ конца строки 52, 60
◊ конца строки символов 63
◊ конца файла 38
Примеры создания классов 147
Принудительное преобразование к типу 91
Принципы построения классов 144
Программа
◊ копирования символьного файла 39, 42
◊ на C/C++ 28
◊ подсчета количества встречающихся в тексте слов 133
◊ с аргументами-указателями 112

◊ с аргументами-функциями 118
◊ где функция — член структуры 123
◊ -отладчик 61
Программы
◊ с использованием классов 147
◊ со структурами 125
Проектирование
◊ баз данных 414
◊ таблиц 417
Пространство имен 180
◊ System 246
Прототип функции 73
Прямоугольные таблицы 414

Р

Работа с
◊ бинарным файлом 187
◊ датами 365
◊ классом fstream 181
◊ классом ifstream 185
◊ классом ofstream 184
◊ логическими данными 249
◊ переменными типа String 247
◊ целыми и вещественными переменными 248
Рабочий стол среды C++ 7
Размерность массива 52
Регистрация пользователя в приложении 380
Регулируемый указатель 16, 107
Редактор кода 208
Рекурсивные функции 72
Рекурсия в структурах 133
Рисование графиков в форме 237

С

Свойства
◊ класса 142
◊ формы 221
Серверы 414
Символические константы 35
Символьные данные 37
Смысл квалификатора unsigned 86
Структура FILE 165
События формы 234
Создание
◊ проекта приложения 197
◊ простого шаблона функции 76
◊ функций 57
Специальный тип void(), 57
Ссылочный тип классов 156
Стандартный ввод cin 193
Стандартный ввод/вывод в C 172
Стандартный ввод/вывод в C++ 189
Стандартный вывод cout 189
Статическая переменная 74
Статические функции 158
Стековая память 58
Структура программ в VC++ 15
Структуры 120
◊ и функции 125
Суфлер кода 212

Т

Таблица кодов ASCII, структура 52
Тело функции 72
Тип auto 32
Типы данных 23
Точка с запятой 31
Транзакция 419
Триггер 413

У

Удаленные БД 414
Указатели 105
◊ * и ^ 16, 108
◊ this 208
◊ и массивы 109
◊ на функции 118
◊ символов и функций 113
Унарная операция 90
Уникальный ключ 419
Управляющий символ 28
Условие окончания цикла 32, 33, 35
Условное выражение 95

Ф

Формальные параметры 58
Форматные строки даты и времени 359
Функция
◊ стандартного ввода/вывода в C 172
◊ char 84
◊ malloc() 92
Функция
◊ выделения подстроки из строки 62
◊ копирования строки в строку 63

Х

Хранимые процедуры 413

Ч

Числа с плавающей точкой 31

Э

Экземпляр структуры 138
Эхо-сопровождение 40

Я

Язык SQL 420