004
М294

004.4

ИНФОРМАТИКА

МАРТЫНОВ Н. Н.

C# для начинающих

2035446

TATU KUTUBXONASI
368876-SONLI

КУДИЦ-ПРЕСС
Москва • 2007

004.43

ББК 32.973.26-018.1

Мартынов Н. Н.
С# для начинающих. – М.: КУДИЦ-ПРЕСС, 2007. – 272 с.

ISBN 5-91136-023-3

Книга является общедоступным учебником начального уровня по основам информатики и программированию на языке С#. Она может быть рекомендована студентам и преподавателям вузов, слушателям курсов повышения квалификации, школьникам и учителям средних школ. Кроме того, она будет полезна всем, кто интересуется созданием компьютерных программ с богатым графическим интерфейсом пользователя для решения задач математики, физики, химии, биологии и других дисциплин, в том числе гуманитарных, а также разработчикам в области информационных технологий, желающим освоить программирование на платформе Microsoft NET Framework.

От читателей требуется наличие минимальных знаний по основам информатики и программированию на языке С. Идеальным введением в данную книгу послужит предыдущий труд автора – Мартынов Н. Н. "Информатика. С для начинающих", Издательство "КУДИЦ-ОБРАЗ", Москва, 2006.

Книга прекрасно иллюстрируют роль языка С# как наиболее универсального языка программирования для платформы Microsoft NET Framework операционной системы Windows. Рассматриваются основы построения приложений Windows с графическим интерфейсом пользователя.

Для практической работы с пособием можно использовать компиляторы Microsoft Visual C# NET (2003) или Microsoft Visual C# NET (2005), приемы работы с которыми подробно описаны в Приложении к настоящему пособию.

Мартынов Николай Николаевич
С# для начинающих
Учебно-справочное издание

Макет В. Г. Клименко

ООО «КУДИЦ-ПРЕСС»
190068, С.-Петербург, Вознесенский пр-т, д. 55,
литер А, пом. 44
тел. (495) 333-82-11. ok@kudits.ru. http://books.kudits.ru

Подписано в печать 1.08.2007 г.
Формат 70x90/16. Бум. офс. Печать офс.
Усл. печ. л. 19,89. Тираж 2000. Заказ 1428

Отпечатано в ОАО «Щербинская типография»
117623, Москва, ул. Типографская, д. 10
Т. 659-23-27.

ISBN 5-91136-023-3

© Макет, обложка ООО «КУДИЦ-ПРЕСС», 2007
© Мартынов Н. Н., 2007

Содержание

Содержание

Часть I

Алгоритмы и программирование на языке C#

Глава 1. Простейшие алгоритмы и программы на языке C#

1.1. Формулировка алгоритмов и написание программ на языке C#

Как и ранее в книге [1], посвященной языку C, мы выберем для начала несколько простейших задач, вроде задачи о вычислении среднего арифметического трех чисел, или корня квадратного из суммы квадратов двух чисел и тому подобных задач. Для перечисленных задач *алгоритм решения* сводится просто к *применению математических формул* для вычисления результата. В результате можно сосредоточиться на компьютерных аспектах (основы языка C#, работа с программой-компилятором Microsoft Visual C# в среде Windows), а не на математике.

Итак, начинаем. Даны три действительных (вещественных) числа `x1`, `x2` и `x3`. Нужно вычислить их среднее арифметическое. Для этого, как мы знаем, нужно сначала сложить эти числа, а результат сложения поделить на три. Вот и весь алгоритм решения задачи. Мы его сформулировали на естественном языке, но можно также написать и математическую формулу:

```
res = ( x1 + x2 + x3 ) / 3
```

Здесь мы ввели *имя переменной* (*идентификатор*) для хранения результата вычислений в виде res. В общем, обычная алгебра.

Теперь эту формулу нужно записать по правилам языка программирования C#:

```
double res, x1, x2, x3;
x1 = 4.5; x2 = 5.6; x3 = 7.8;
res = ( x1 + x2 + x3 ) / 3;
```

причем собственно формула для вычисления результата расположена в третьей строке представленного фрагмента и незначительно отличается от записанной ранее математической формулы лишь наличием обязательной точки с запятой.

Точка с запятой в языке C#, как и в языках C/C++, служит обозначением конца *оператора*. Оператор является минимальной законченной смысловой конструкцией, являющейся аналогом предложения естественного языка. Именно так ее воспринимает *компилятор языка C#* – программа, в задачу которой входит перевод текста с языка C# на внутренний язык некоторого абстрактного универсального процессора (так называемый *промежуточный язык* – IL, *Intermediate Language*), команды которого пусть и не соответствуют в точности набору команд реальных процессоров, но весьма близки к ним (некоторые совпадают полностью, часть команд не используются, а некоторые являются приблизительными аналогами). Этот процесс, называемый *компиляцией*, завершается построением загрузочного exe-файла PE-формата, выполнение которого автоматически осуществляется специальной средой CLR (Common Language Runtime) платформы Microsoft NET Framework. Об этой платформе мы уже упоминали и будем ее подробнее рассматривать в последующих разделах. А сейчас вернемся к операторам.

В третьей и второй строках нашего фрагмента стоят *операторы-выражения*, то есть выражения языка C#, в конце которых проставлена точка с запятой. Выражения состоят из знаков *операций* и *операндов* (переменных, констант, подвыражений). Операции – это действия: сложение, деление и так далее. Операнды – это то, над чем операции выполняются. Знак «+» – это знак сложения (для наглядности кавычками знак операции отделяется от остального текста), знак «/» – это знак деления, знак «=» – это знак *операции присваивания*.

Из представленных пояснений следует, что во второй строке нашего фрагмента расположены (один за другим) три оператора-выражения, в каждом из которых осуществляется присваивание числовой константы некоторой переменной (переменные заданы своими именами). В первом из них переменной `x1` присваивается значение `4.5` (*точка* применяется для обозначения *десятичной запятой*), во втором – переменной `x2` присваивается число `5.6` и т. д. Числа выбраны нами произвольно и никакого специального смысла не несут.

Внимание: Язык C#, в отличие от своих знаменитых, но «слишком вольных» предшественников – языков C/C++, не допускает применения синтаксически корректных, но логически незавершенных операторов-выражений типа

x1 + x2, и тому подобных «изысков», а также требует обязательной инициализации переменных перед их использованием в вычислениях.

В третьей строке фрагмента наша формула, собственно, и вычисляется. С точки зрения языка C# здесь присутствует операция присваивания наряду с операциями сложения и деления. Работа любой операции присваивания заключается в том, что сначала вычисляется выражение справа от знака «=», а затем вычисленное значение присваивается переменной, стоящей слева от знака присваивания. Именно этот порядок вычислений мы и ожидаем от записи в третьей строке нашего фрагмента, с учетом понимания группировки операций круглыми скобками и понимания в алгебраическом смысле *приоритетов операций*. У операции деления приоритет выше, чем у операции сложения, поэтому нам и нужны здесь группирующие круглые скобки.

Наконец, в первой строке фрагмента присутствуют так называемые *операторы-определений* переменных x1, x2 и x3 *типа double*. Тип double – это тип вещественных переменных, которые на внутреннем компьютерном уровне устроены как конечные дроби. *Определить переменную* – значит *объявить ее имя* (строится из английских букв, знака подчеркивания и цифр, которые не могут стоять на первой позиции) и *указать тип*, а в итоге под эту переменную будет зарезервирован участок компьютерной памяти соответствующего размера. В этом участке и будет храниться числовое значение переменной. Как и в языках C/C++, в одном операторе определения языка C# можно перечислить *через запятую* сразу несколько идентификаторов определяемых переменных. Типы переменных в нашей программе объявлены с помощью *ключевого* (*зарезервированного*) *слова double*. Ключевые слова языка C# (их нельзя использовать в качестве имен переменных) мы будем постепенно изучать на протяжении книги.

А сейчас зададим нетерпеливый вопрос: можно ли уже пытаться компилировать наш фрагмент из трех строк? К сожалению, пока еще нельзя, так как нужно выполнить два обязательных требования языка C#.

Во-первых, минимальная вычислительная активность программы на языке C# должна быть сосредоточена в *функции* с именем *Main* (точка входа в программу), так что наш фрагмент должен оказаться «в составе определения этой функции»:

```
int Main( )
{
  double res, x1, x2, x3;

  x1 = 4.5; x2 = 5.6; x3 = 7.8;
  res = ( x1 + x2 + x3 ) / 3;

  return 0;
}
```

где первая строка – это строка *заголовка функции* Main, после которого следует *тело функции*, заключенное в фигурные скобки. Наш трехстрочный фрагмент теперь стал телом функции Main, и к которому еще добавился *оператор возврата из функции*, состоящий из ключевого слова return и величины возвращаемого значения (в данном случае для возвращаемого значения произвольно выбран нуль). Заголовок функции Main состоит из *ключевого слова int* (основной *целочисленный тип данных* языка C#), означающего *тип возвращаемого функцией значения*, с последующим именем функции и ее параметрами (аргументами), перечисленными через запятую внутри круглых скобок, но так как мы не предусматриваем для функции Main никаких параметров, то внутри круглых скобок не пишется ничего.

Внимание: В языке C#, в отличие от языков C/C++, использовать ключевое слово *void* внутри круглых скобок в заголовке определения функции нельзя.

Удовлетворив первое обязательное требование, переходим ко второму: программа на языке C# должна заключать все функции в *классы*, представляющие собой в простейшем случае сплав функций (функциональных классовых методов или просто методов – все это фактически синонимы) и полей данных, но последних может и не быть вовсе:

Листинг 1.1.

```
class main
{
  public static int Main( )
  {
    double res, x1, x2, x3;

    x1 = 4.5; x2 = 5.6; x3 = 7.8;
    res = ( x1 + x2 + x3 ) / 3;

    return 0;
  }
}
```

Имя класса, заключающего в себе стартовую функцию Main можно выбирать произвольно, так что мы, не мудрствуя лукаво, выбрали имя main.

Более существенным является дополнение заголовка определения функции Main *модификаторами* (ключевыми словами) *public* и *static*. Первый означает, что функцию можно вызывать извне (стартовая функция Main должна быть доступна внешнему программному загрузчику), а второй – что функция существует сама по себе, то есть без предварительного создания так называемых классовых объектов, о которых мы будем говорить позже.

Листинг 1.1 представляет собой текст законченной программы на языке C#. Это, конечно, сверхпростейшая программа, однако первый шаг всегда требует повышенного внимания, так что примите поздравления!

Текст с Листинга 1.1 уже можно подавать на вход программы-компилятора, но прежде, чем делать это, чуть задержимся на *стиле оформления* программного текста. Для компилятора стиль оформления не имеет никакого значения. Другое дело человек. Он и пишет программы, он же и читает их, причем не только свои собственные программы. Поэтому для человека стиль оформления имеет немалое значение. Одну и ту же программу можно написать в ясном, наглядном для зрительного восприятия стиле (что-то вроде варианта Листинга 1.1), а можно все так накрутить, что сил не хватит разобраться в диком лабиринте записей. Язык C# разрешает писать программу в свободном формате, лишь бы ключевые слова «не склеивались». Вот нарочитый *отрицательный пример* на эту тему:

```
class main{public static int Main(){double
res, x1, x2,
x3;x1
=4.5;x2=5.6;x3=7.8;res =
(
x1
 + x2 + x3 ) / 3;return
0;      }}
```

где мы в учебных целях намеренно исковеркали программу с Листинга 1.1, не добавив ни одной синтаксической ошибки. Но читать такую абракадабру вряд ли кто захочет (включая автора). *Отсюда мораль:* если что-то можно зрительно улучшить, то это следует сделать: лишний пробел, выравнивание отдельных элементов записей друг относительно друга, сдвиг записей операторов тела функции относительно левого края на заданное число позиций и так далее. Но не следует и переусердствовать: «лучшее – враг хорошего».

Имеется возможность вносить *комментарии* в текст программы на языке C#, то есть текст, который компилятором игнорируется, а нужен он лишь программисту для пояснений к программе или для улучшения ее зрительного вида (улучшения читаемости). Однострочный комментарий располагается в строке исходного кода программы после символов `//`, а многострочный комментарий (может занимать одну или более физических строк исходной программы) располагается после открывающих символов `/*` и до закрывающих символов `*/`:

```
class main
{
  public static int Main( )
```

```
  {
    // Определение переменных:
    double res, x1, x2, x3;

    /*- Инициализация и вычисления -*/
    x1 = 4.5; x2 = 5.6; x3 = 7.8;
    res = ( x1 + x2 + x3 ) / 3;
    /*----------------------------*/

    return 0;
  }
}
```

Здесь представлены оба вида комментариев, причем комментарии в виде набора знаков тире предназначаются якобы для улучшения зрительного выделения фрагмента кода из двух строк, в котором производится вычислительная работа с переменными x1, x2 и x3. На самом деле, это вряд ли удачное решение – в Листинге 1.1 и так все прекрасно сгруппировано и все прекрасно видно. В других, более сложных программах, показанное применение комментариев может и помочь в какой-то степени. Но в любом случае, лучше ограничиваться минимумом вспомогательных средств, а то ведь можно вместо улучшения читаемости получить ее ухудшение.

В отличие от книги [1], где для языка программирования C в составе комментариев русский язык не применялся, здесь мы сразу же допустили такую возможность, поскольку платформа Microsoft NET Framework скорее всего будет развертываться поверх операционных систем Windows 2000, Windows XP (или более поздних версий Windows), которые сами прекрасно работают с различными национальными языками, и позволяют это легко делать прикладным программам, в том числе и компилятору Microsoft Visual C#.

1.2. Применение компилятора Micorosft Visual C# NET (2003) для построения исполняемой программы в среде Windows

Более подробно про компилятор Microsoft Visual C# NET (2003) рассказано в Приложении к нашему учебному пособию. А сейчас мы просто начинаем работать, для чего запустим компилятор и с помощью команды меню File | New | Project... вызовем диалоговое окно New Project (см. рис. 1.1).

Здесь в левой части окна следует выделить строку *Visual C# Project*, а в правой части окна – строку *Empty Project*. В поле ввода Name нужно ввести с клавиатуры *название проекта* (у нас здесь это Proj1), а перед этим с помощью поля с на-

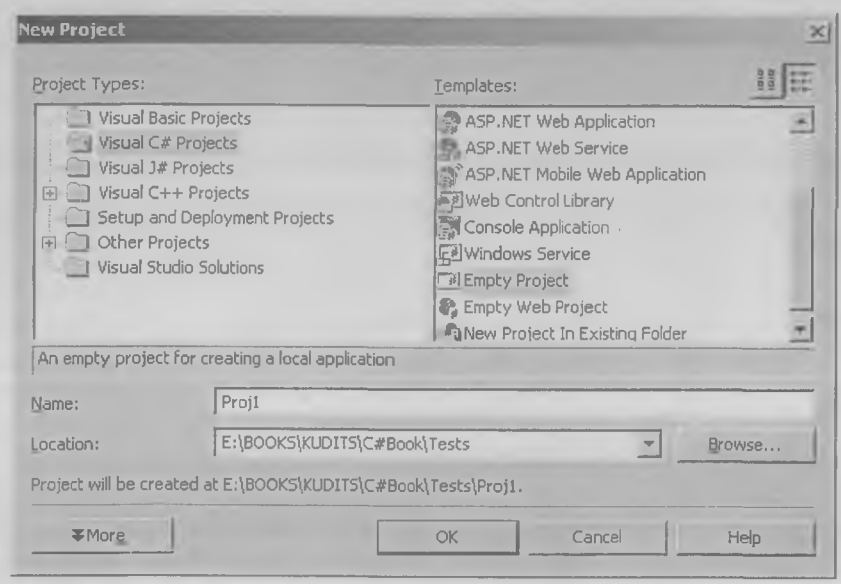


Рис. 1.1.

званием Location (и, возможно, кнопки Browse...) следует выбрать *дисковый каталог*, в котором все учебные проекты будут располагаться.

В конце концов нажимаем кнопку OK – и проект создан. Графической средой компилятора автоматически создается одноименный проекту дисковый каталог Proj1, в котором располагаются все *вспомогательные файлы*, обслуживающие проект, что и показано на рис. 1.2.

Теперь мы должны «внедрить в проект» файл с текстом Листинга 1.1. Для этого выполняем команду меню Project | Add New Item... графической среды компилятора Microsoft Visual C#, и в результате появляется одноименное диалоговое окно (см. рис. 1.3).

Выбираем строку Code File, набираем в поле Name имя файла main.cs (буквосочетание cs расшифровывается как C Sharp – официальное «звучащее название» языка C# от фирмы Microsoft), нажимаем кнопку Open и такой файл в каталоге нашего проекта автоматически создается (см. рис. 1.4).

Действительно, из рис. 1.4 видно, что в каталоге проекта Proj1 помимо четырех служебных файлов (из которых самыми важными являются файлы Proj1.sln и Proj1.csproj) появился предназначенный для хранения исходного текста нашей программы файл main.cs. Текст Листинга 1.1 в этот файл можно ввести

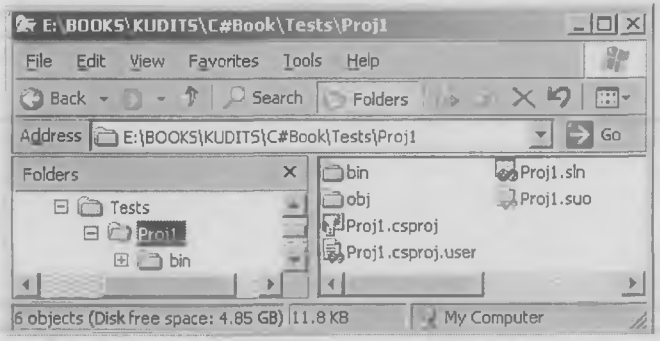


Рис. 1.2.

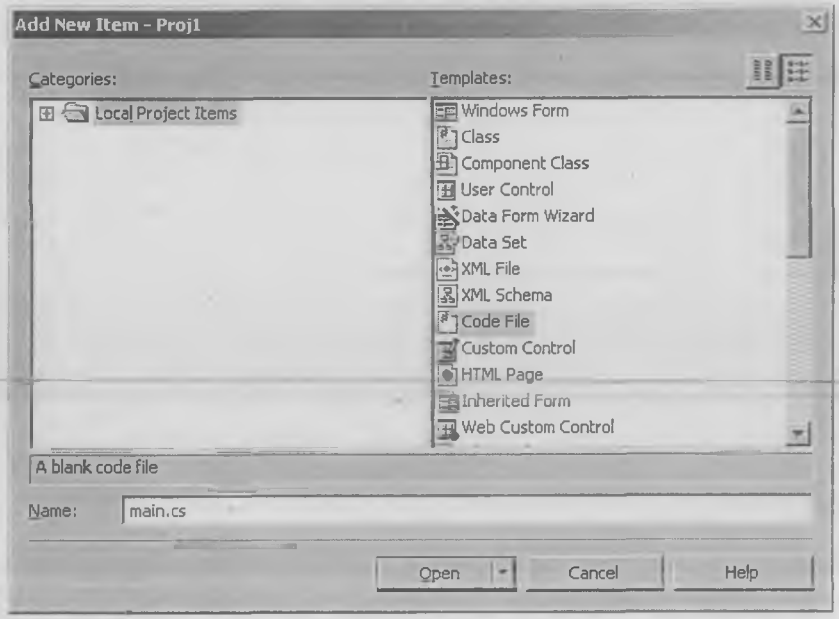


Рис. 1.3.

с клавиатуры в рамках графической среды компилятора Microsoft Visual C#, пользуясь правой частью его окна как текстовым редактором (см. рис. 1.5).

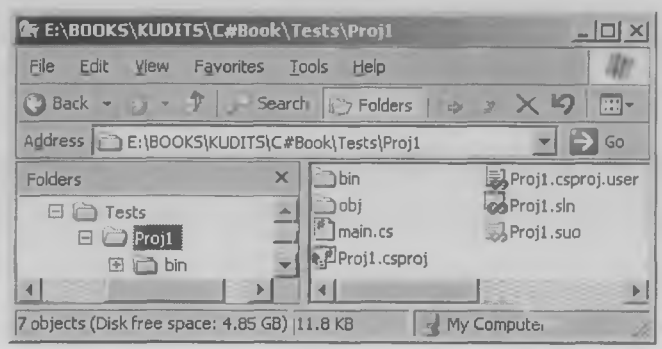


Рис. 1.4.

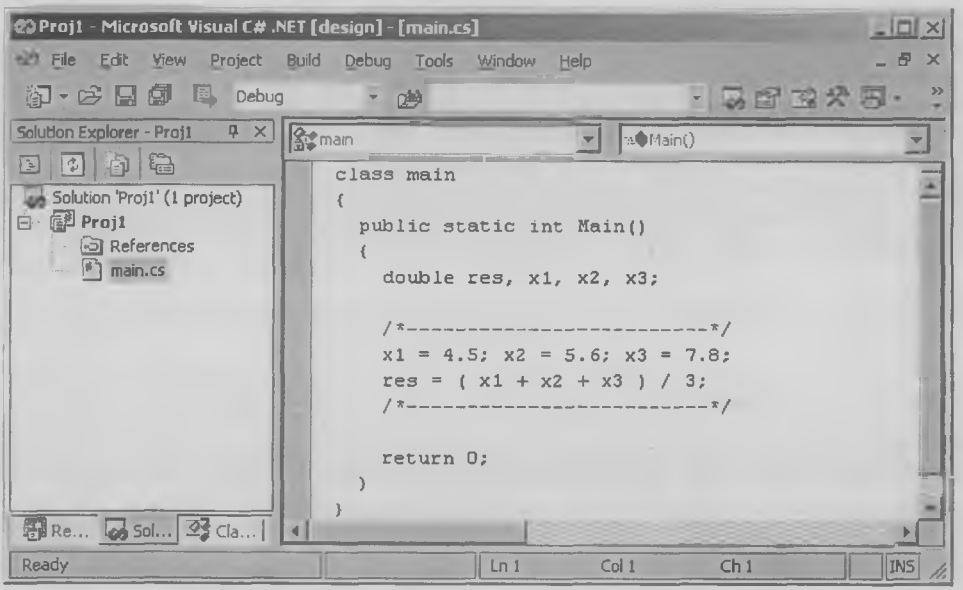


Рис. 1.5.

Правая часть главного окна компилятора Microsoft Visual C# NET служит *текстовым редактором*, причем этот редактор знает ключевые слова языка C# и выделяет их синим цветом.

Вот теперь *все готово к процессу компиляции*. Сделать это проще простого – нужно лишь выполнить команду меню Build | Build Solution. По истечении край-

не малого времени (столь простая программа, как наша начальная учебная программа, компилируется практически мгновенно) процесс компиляции завершается, а итоговый отчет о его результате можно наблюдать в *нижней части окна компилятора* (см. рис. 1.6).

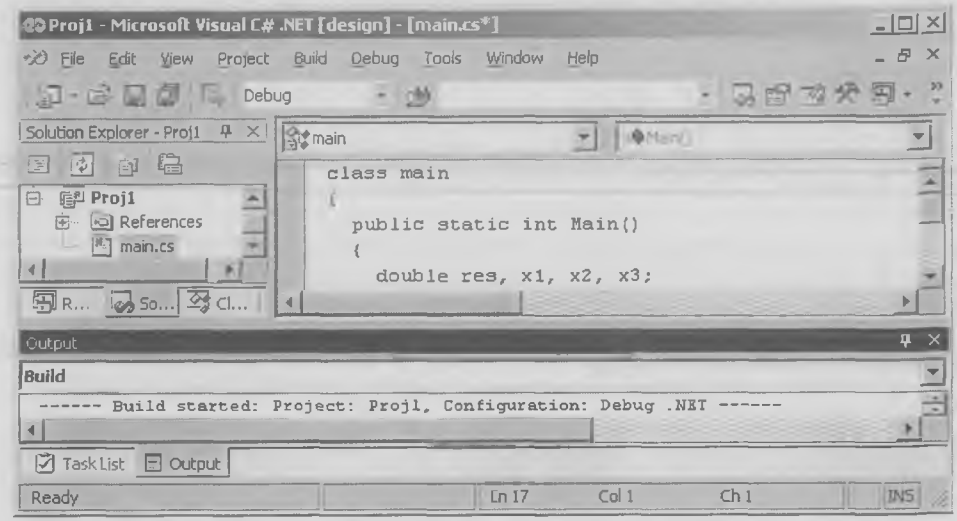


Рис. 1.6.

Самыми важными в списке сообщений из нижней части окна компилятора (при выделенной закладке Output) являются *сообщения об отсутствии ошибок в процессе компиляции* (0 errors), и даже об *отсутствии предупреждений* (0 warnings). Компилятор счел, что мы написали программу на языке C# абсолютно правильно и создал *загрузочный файл* Proj1.exe.

Этот файл можно найти в подкаталоге Debug рабочего каталога нашего проекта и запустить его оттуда на выполнение любым известным в рамках операционной системы Windows способом. Но лучше всего в процессе обучения языку C# запускать исполняемый вариант программы прямо из графической среды компилятора Microsoft Visual C# *клавишей F5*. Дело в том, что таким образом запускается на выполнение *отладочный вариант* скомпилированной программы (именно он располагается в подкаталоге Debug), и именно его создание промаркировано в нижней части окна компилятора (см. рис. 1.6) надписью

```
Project: Proj1, Configuration: Debug .NET
```

где важнейшим моментом является наличие слова *Debug*, которое и переводится как *отладка*.

Под отладкой понимается тестовый прогон полученной программы с целью выявления ее ошибочного или неадекватного первоначальным планам поведения. Ясно, что этот режим является желательным с практической точки зрения на всем пути разработки программы (от него отказываются лишь под самый конец, когда компилируют *окончательный Release вариант программы* для ее продажи или просто передачи в эксплуатацию).

При запуске клавишей F5 отладочного (Debug) варианта программы имеются дополнительные возможности по нахождению ошибок и по *динамическому наблюдению за значениями переменных*.

Например, можно *клавишей F9* поставить *маркер останова* в некоторую строку программы (там должен в этот момент располагаться мигающий текстовый курсор), и тогда выполнение программы приостановится на коде, соответствующем этой программной строке (см. рис. 1.7).

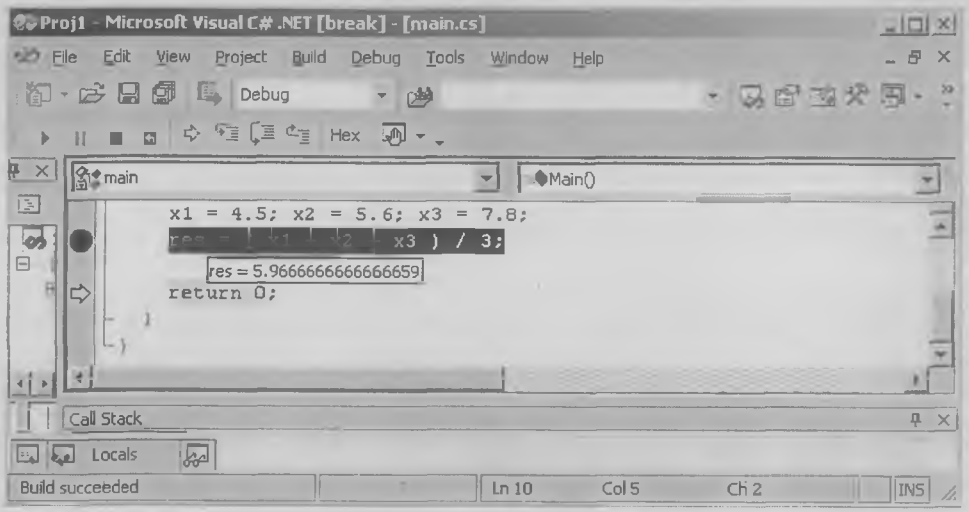


Рис. 1.7.

Наводя курсор мыши на имена переменных, можно во *всплывающем окне* наблюдать их текущие значения, полученные в процессе выполнения программы вплоть до точки останова. Например, на рис. 1.7 показано текущее значение переменной `res`. Такие возможности визуализации очень удобны для отладки программ.

TATU KUTUBXONASI
368876

Чтобы перейти к следующей строке программы на языке C#, достаточно нажать F10, после чего программа снова приостанавливает свое выполнение, и вы можете описанным выше способом наблюдать новые значения переменных.

Возобновление выполнения программы в автоматическом безостановочном режиме происходит по нажатию на клавишу F5.

1.3. Реализация вычислительных алгоритмов выделенными для этой цели функциями. Операторы ветвления и цикла.

Функции являются краеугольным камнем программирования на языке C (см. книгу [1]). Выделение в отдельную функцию программной реализации, например, вычислительных алгоритмов, улучшает структурированность всей программы, позволяет вести разработку и ее модификацию по частям. Кроме того, отдельные функции можно повторно использовать в составе иных программных проектов.

Естественно, что язык C#, являясь «разумным наследником» славных традиций языков C/C++, ничему из сказанного не препятствует. В качестве иллюстрации осуществим изъятие формулы для вычисления среднего арифметического трех действительных чисел из тела стартовой функции `Main` с Листинга 1.1 и перенесем ее в отдельную функцию с именем `Aver`:

Листинг 1.2.

```
class main
{
  // Стартовая функция:
  public static int Main()
  {
    double res, x1, x2, x3;
    x1 = 4.5; x2 = 5.6; x3 = 7.8;

    res = Aver( x1, x2, x3 );
    return 0;
  }

  // Функция для вычисления среднего
  // арифметического трех действительных чисел:
  static double Aver( double x, double y, double z )
  {
    return ( x + y + z ) / 3;
  }
}
```

В Листинге 1.2 помимо определения стартовой функции `Main` присутствует еще и *определение функции* `Aver`, предназначенной исключительно для вычисления среднего арифметического из трех действительных чисел, передаваемых ей в качестве параметров.

Фактические вычисления среднего арифметического значения из трех заданных чисел осуществляются из тела стартовой функции `Main` с помощью *вызова функции* `Aver`:

```
res = Aver( x1, x2, x3 );
```

в рамках которого этой функции передаются в качестве *фактических параметров* переменные `x1`, `x2` и `x3` типа `double`. Соответственно, в определении функции `Aver` идентификаторы `x`, `y` и `z` называются *формальными параметрами* (и им не соответствуют никакие конкретные числовые значения).

Кода Листинга 1.2 и представленных кратких пояснений достаточно, чтобы понять полную идентичность самой сути функциональных определений и вызовов в языке C (см. [1]) и языке C#. Поэтому мы, как и ранее в книге [1], будем ссылаться на функции, используя одновременно их имена и круглые скобки, так что всюду далее мы будем говорить о функциях `Main()`, `Aver()` и так далее.

В то же время, иной контекст существования функций в языке C#, – а именно в составе классов, диктует и неминуемые различия (в основном – в сторону дополнительных черт, расширяющих базовые возможности функций языка C): например, в Листинге 1.2 определение функции `Aver()` снабжается модификатором `static`, чтобы ее можно было вызывать просто так (как в языке C), то есть независимо от создания классовых объектов (об этом см. гл. 2).

Кроме того, классовый аспект определения функций в языке C# диктует необходимость компилятору полностью изучить устройство всех классовых методов до того, как принимать решение о правильности вызова некоторых из них. В результате *отпадает необходимость в предварительных объявлениях функций* (*прототипах*) в тех случаях, когда их определение представлено ниже по тексту программы, чем их вызов (как в Листинге 1.2). А это уже, наряду с признанием недостаточно удачной широко существующей в языке C практики макроопределений (см. гл. 7 из книги [1]), вообще почти полностью «ставит крест» на *препроцессоре* – в языке C# от него остались лишь *директивы условной компиляции*, но о них мы в этой книге ничего говорить не будем.

«Сухой остаток» от сказанного – определения функций в классах языка C# можно располагать в произвольном порядке, но все же, какой-то логический порядок, улучшающий «читабельность» исходного кода, лучше полного хаоса.

А теперь перейдем к наращиванию наших знаний про операторы языка C#, ибо без *операторов ветвления* и *операторов цикла* невозможно запрограммировать мало-мальски интересные и полезные вычислительные алгоритмы. Хорошая новость в отношении этих операторов заключается в том, что в языке C# они практически (правда, не полностью) идентичны таковым из языка C (см. гл. 2 из книги [1]). Во всяком случае, функцию `MySum()` из Листинга 2.9 разд. 2.6 книги [1], да и сам этот Листинг, мы воспроизводим здесь в C#-варианте с минимальными отклонениями от оригинала:

Листинг 1.3.

```
class main
{
  /*---------------------------------------*/
  static int MySum( int M, int N ) /*1*/
  {                                /*2*/
    int i, Sum;                    /*3*/
    i = M; Sum = 0;                /*4*/
                                   /*5*/
    while( i <= N )                /*6*/
    {                              /*7*/
      Sum = Sum + i;               /*8*/
      i++;                         /*9*/
    }                              /*10*/
                                   /*11*/
    return Sum;                    /*12*/
  }                                /*13*/
  /*---------------------------------------*/
  public static int Main( )        /*14*/
  {                                /*15*/
    int i, N, Sum;                 /*16*/
    i = 137; N = 895;              /*17*/
                                   /*18*/
    Sum = MySum( i, N );           /*19*/
    Sum = MySum( 5, 555 );         /*20*/
    Sum = MySum( Sum, Sum + 1 );   /*21*/
    Sum = MySum( N, N+3 ) + 4;     /*22*/
    MySum( 11, 123 );              /*23*/
                                   /*24*/
    return 0;                      /*27*/
  }                                /*28*/
  /*---------------------------------------*/
}
```

Оператор цикла, который часто называют *оператором while*, в только что представленном Листинге 1.3 располагается в строках /*6*/-/*10*/.

Если бы нам захотелось применить здесь иной вариант оператора цикла (с эквивалентным результатом работы), то текст функции MySum() принял бы следующий, более краткий вид:

```
static int MySum( int M, int N )          /*1*/
{                                          /*2*/
  int i, Sum;                              /*3*/
                                           /*4*/
  for( i = M, Sum = 0; i <= N; i++ )       /*5*/
  {                                        /*6*/
    Sum = Sum + i;                         /*7*/
  }                                        /*8*/
                                           /*9*/
  return Sum;                              /*10*/
}                                          /*11*/
```

Здесь в строках /*3*/-/*6*/ стоит оператор цикла на базе ключевого слова for, или как его часто для краткости называют – *оператор for*.

Итак, на первый взгляд, операторы цикла в языке C# практически неотличимы от таковых из языка C. То же самое имеет место и по отношению к операторам ветвления, использующим ключевые слова if и else (см. гл. 2 [1]).

Тем не менее, имеется и ряд отличий. Первое отличие заключается в том, что в языке C#, в отличие от языков C/C++, нет такой экзотической операции, как операция «запятая» (из-за этого не проходит, например, оператор i++, Sum++), и которая позволяла писать в операторе for внутри круглых скобок до первой точки с запятой (вместе с ней получался оператор языка C) совсем уж «накрученные выражения». В языке C# «сверхвольности» языков C/C++ отметаются довольно последовательно, так что в операторе for языка C# до первой точки с запятой в круглых скобках допускается писать лишь строго ограниченный набор выражений, в первую очередь – присваиваний, отделенных друг от друга запятой (запятая здесь выступает в качестве разделителя, а не операции).

Второе отличие несколько более существенное, хотя и оно связано с вольностями языков C/C++. Это отличие касается записей после первой точки с запятой в круглых скобках у оператора for или просто записей в круглых скобках после ключевого слова while. Это так называемые *условия цикла*, то есть условия, проверка выполнимости (истинности) которых влияет на принятие решения о продолжении/прекращении циклических итераций.

Вольность языков C/C++ в отношении условий цикла заключается в допущении использования в них арифметических выражений с трактовкой ненулевых

числовых значений как истинных. Эта вольность в языке C# отметается, а для условий цикла допускаются лишь *логические выражения*, использующие операции сравнения и логические операции (можно применять и *логические переменные*, определенные с *ключевым словом bool* – их возможные значения есть *true* или *false*).

Операции сравнения в языке C# те же самые, что и в языках C/C++ (то есть операции «==», «!=», «<», «<=», «>», «>=»). Логические операции языков C/C++, то есть операции «!», «&&» и «||» (см. разд. 2.3 из книги [1]) сохраняют свое поведение и в языке C#, но дополняются в нем новыми логическими операциями «&», «|» и «^», острой необходимости в которых нет и мы их рассматривать не будем.

Заканчивая данный раздел на «высокой ноте» операций языка C#, не забудем сообщить, что операции присваивания языков C/C++ не претерпели в C# изменений, так что по-прежнему можно использовать так называемые комбинированные (или составные) операции присваивания, позволяющие, в частности, вместо `i = i + 1` писать короче: `i += 1`, или еще короче: `i++` (см. выше строку `/*9*/` из Листинга 1.3).

1.4. Группировка функций в классы и пространства имен. Построение библиотек

Если бы мы разработали не одну функцию для определенного рода вычислений, как это имеет место в Листинге 1.3 с единственной вычислительной функцией `MySum()`, а целый набор таких функций, то было бы целесообразно для наглядности сгруппировать их в рамках отдельного, специально предназначенного для этого класса, поименованного, например, `MyCalc`.

Выполним преобразование Листинга 1.3 в этом направлении (несмотря на наличие единственной вычислительной функции):

Листинг 1.4.

```
/*-----  File main.cs  -----*/
class main
{
  public static int Main( )
  {
    int i, N, Sum;
    i = 137; N = 895;

    Sum = MyCalc.MySum( i, N );
    Sum = MyCalc.MySum( 5, 555 );
```

```
    Sum = MyCalc.MySum( Sum, Sum + 1 );
    Sum = MyCalc.MySum( N, N+3 ) + 4;
    MyCalc.MySum( 11, 123 );

    return 0;
  }
}

class MyCalc
{
  public static int MySum( int M, int N )
  {
    int i, Sum;
    i = M; Sum = 0;

    while( i <= N )
    {
      Sum = Sum + i;
      i++;
    }

    return Sum;
  }
}
```

По сравнению с Листингом 1.3 здесь пришлось в отдельном классе `MyCalc` определить функциональный метод `MySum()` с модификатором `public`, иначе его нельзя было бы вызвать извне класса `MyCalc`, например из метода `Main()` класса `main`.

Далее, поскольку классов в программе может быть определено много, то целесообразно осуществить какую-то «изоляцию» имен их методов друг от друга. Это в языке C# (как и ранее в языке C++) достигается за счет того, что вне классов имена методов (речь идет о методах с модификатором static) нужно *квалифицировать именем содержащего их класса*, как например `MyCalc.MySum()` в Листинге 1.4, и где имя класса отделяется от имени функционального метода точкой.

Теперь обратим внимание на то обстоятельство, что у нас оба определения классов по-прежнему сосредоточены в единственном файле main.cs. Такое решение не всегда удобно на практике, когда в едином проекте могут сосуществовать десятки и сотни классов. В этом случае целесообразно «раскидать» их по разным исходным файлам (каждый файл дожен быть, естественно, включен в проект), сосредотачивая в рамках одного и того же исходного текстового файла определения логически близких или связанных классов. Над каждым из таких файлов могут работать разные группы программистов.

Мы сейчас выполним это упражнение как чисто учебное (у нас-то всего-навсего два класса в Листинге 1.4):

Листинг 1.5.

```
/*-----  File main.cs  -----*/
class main
{
  public static int Main( )
  {
    int i, N, Sum;
    i = 137; N = 895;

    Sum = MyCalc.MySum( i, N );
    Sum = MyCalc.MySum( 5, 555 );
    Sum = MyCalc.MySum( Sum, Sum + 1 );
    Sum = MyCalc.MySum( N, N+3 ) + 4;
    MyCalc.MySum( 11, 123 );

    return 0;
  }
}
/*-----  File MyCalc.cs  -----*/
class MyCalc
{
  public static int MySum( int M, int N )
  {
    int i, Sum;
    i = M; Sum = 0;

    while( i <= N )
    {
      Sum = Sum + i;
      i++;
    }

    return Sum;
  }
}
```

Мы добавили в наш проект командой меню Project | Add New Item... графической среды компилятора Microsoft Visual C# новый файл MyCalc.cs, и перенесли в него определение класса `MyCalc` (и поэтому для наглядности назвали новый исходный файл проекта именем MyCalc.cs). В Листинге 1.5 раскидка исходного кода по разным файлам проекта отражается с помощью комментариев

/*--- File main.cs ---*/ и /*--- File MyCalc.cs ---*/. Но намного нагляднее все это выглядит в окне графической среды компилятора Microsoft Visual C# (см. рис. 1.8).

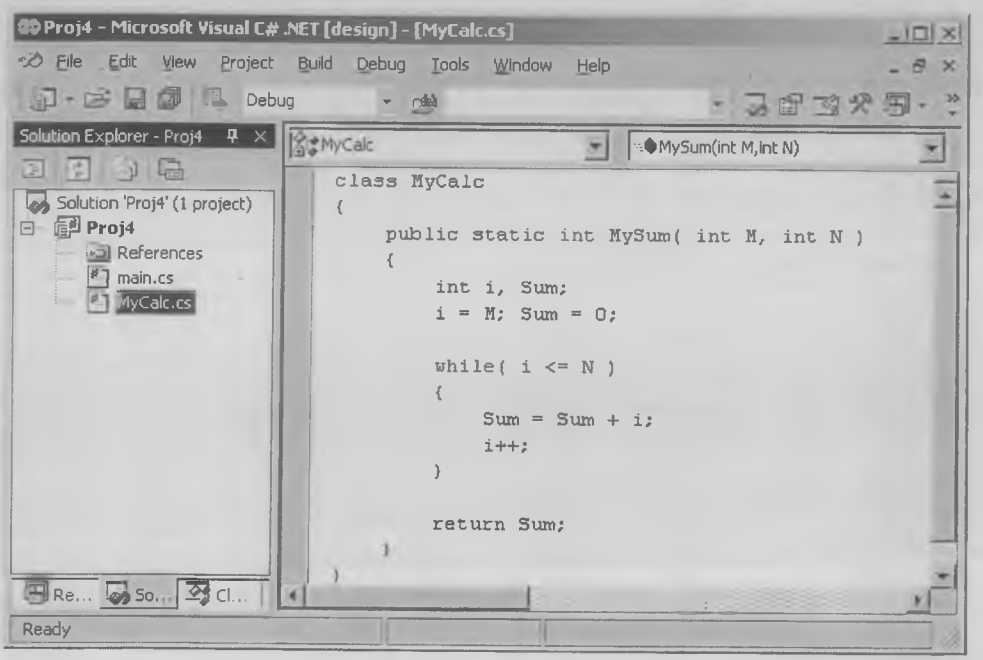


Рис. 1.8.

Из рис. 1.8 прекрасно видно, что в проект Proj4 внедрены два исходных файла – main.cs и MyCalc.cs. Содержимое файла MyCalc.cs видно в правой части окна компилятора на рис. 1.8, поскольку именно этот файл отселектирован (подсвечен) в списке файлов проекта в левой части окна.

Независимо от количества исходных файлов проекта, он по-прежнему компилируется единственной командой меню Build | Build Solution, в результате чего порождается загрузочный файл программы Proj4.exe.

Итак, логическая группировка классов по исходным файлам проекта нами рассмотрена. Но имеется еще возможность логической группировки имен классов в некоторые специальные наборы таким образом, чтобы одинаковые имена классов из разных наборов не конфликтовали друг с другом в рамках одного и того же программного проекта. Такие специальные наборы классов называются

пространствами имен и задаются с помощью *ключевого слова namespace*. Отметим, что логическая группировка классов в пространства имен не имеет прямого отношения к их раскидке по разным исходным файлам, хотя для простоты можно придерживаться рекомендации сосредотачивать все классы некоторого пространства имен в одном исходном файле.

Листинг 1.6.

```
/*-----  File main.cs  -----*/
using MyCalcClasses;

class main
{
  public static int Main( )
  {
    int i, N, Sum;
    i = 137; N = 895;

    Sum = MyCalc.MySum( i, N );
    Sum = MyCalc.MySum( 5, 555 );
    Sum = MyCalc.MySum( Sum, Sum + 1 );
    Sum = MyCalc.MySum( N, N+3 ) + 4;
    MyCalc.MySum( 11, 123 );

    return 0;
  }
}

/*-----  File MyCalc.cs  -----*/
namespace MyCalcClasses
{
  class MyCalc
  {
    public static int MySum( int M, int N )
    {
      int i, Sum;
      i = M; Sum = 0;

      while( i <= N )
      {
        Sum = Sum + i;
        i++;
      }

      return Sum;
    }
  }
}
```

Из Листинга 1.6 видно, что определение класса `MyCalc` заключено в пространство имен `MyCalcClasses`. Теперь вне пространства имен `MyCalcClasses` этот класс виден как `MyCalcClasses.MyCalc`, а его функциональный метод `MySum()` виден как `MyCalcClasses.MyCalc.MySum()`.

Итак, конечно неплохо, что нам удалось так четко сгруппировать и изолировать друг от друга потенциально конфликные имена классов и их функциональных методов, но для пользователей этих классов (часто их называют клиентами классов) и их методов теперь требуется при каждом вызове функции (метода) выполнять двойную квалификацию имени.

Если это утомительно, то можно воспользоваться *директивой*

```
using MyCalcClasses;
```

вносящей имена всех классов из указанного пространства имен в текущую область видимости, так что явного указания пространства имен уже не требуется. Именно это и было сделано в файле main.cs из Листинга 1.6.

Если бы мы упрятали определение класса `MyCalc` внутрь пространства имен, которое само определяется внутри другого пространства имен, как например в следующем коде

```
/*-----  File MyCalc.cs  -----*/
namespace MyCalcClassesOut
{
  namespace MyCalcClassesInner
  {
    class MyCalc
    {
      public static int MySum( int M, int N )
      {
        int i, Sum;
        i = M; Sum = 0;

        while( i <= N )
        {
          Sum = Sum + i;
          i++;
        }

        return Sum;
      }
    }
  }
}
```

то в файле main.cs (то есть в клиентском коде) метод `MySum()` пришлось бы квалифицировать как `MyCalcClassesOut.MyCalcClassesInner.MyCalc.MySum()`, или применить директиву

```
using MyCalcClassesOut.MyCalcClassesInner;
```

и обращаться к указанному методу как `MyCalc.MySum()`.

Пространства имен, в том числе и только что рассмотренные *вложенные пространства имен*, нам сразу же потребуются в следующем разделе, так как мы собираемся начать использовать классы из библиотеки Microsoft NET Framework.

Кроме того, из практических соображений полезно все собственные классы, сгруппированные по логическому принципу в один исходный cs-файл, дополнительно заключать в пространство имен с умело выбранным именем (часто в это имя включают название фирмы-разработчика кода), ибо этот файл можно одновременно использовать в разных программных проектах, в том числе и в проектах сторонних разработчиков. Для этого указанный исходный файл нужно скомпилировать в отдельном *библиотечном проекте* и получить *файл динамической библиотеки* (файл с расширением dll).

Проиллюстрируем сказанное о возможности практически удобного повторного использования содержимого файла MyCalc.cs из Листинга 1.6 в разных программных проектах. Ранее этот исходный файл был разработан в рамках проекта Proj4 (см. рис. 1.8) и вместе с файлом main.cs скомпилирован в загрузочный файл Proj4.exe (файл исполняемой под управлением среды CLR программы). Теперь создадим проект MyCalcClasses, имеющий тип Class Library (см. рис. 1.9).

При этом окно компилятора Microsoft Visual C# сразу же демонстрирует плоды непрошенной помощи (часто такие услуги являются «медвежьими») (см. рис. 1.10).

Удаляем из проекта все, что туда успело попасть помимо нашей воли, для чего подсвечиваем (селектируем) отдельные строки в левом окне графической среды компилятора, показанные на рис. 1.10, и нажимаем клавишу Delete. Но, все же, мы успеваем заметить очень важную подсказку от компилятора, которую специально пометим: для привлечения внимания.

Внимание: Чтобы библиотечный класс был доступен другим проектам, его определение нужно предварить ключевым словом `public`.

Теперь внедряем в наш библиотечный проект файл MyCalc.cs из Листинга 1.6, вносим в него отмеченное только что исправление

```
public class MyCalc
```

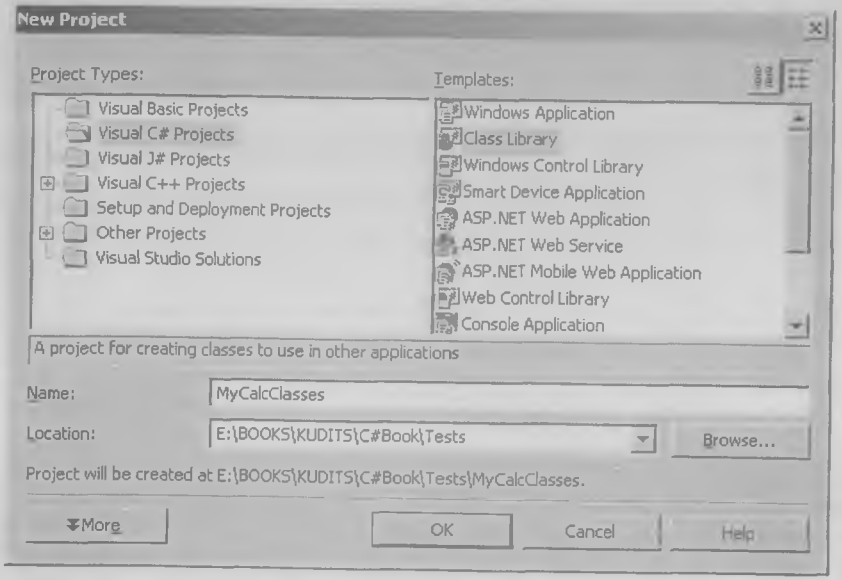


Рис. 1.9.

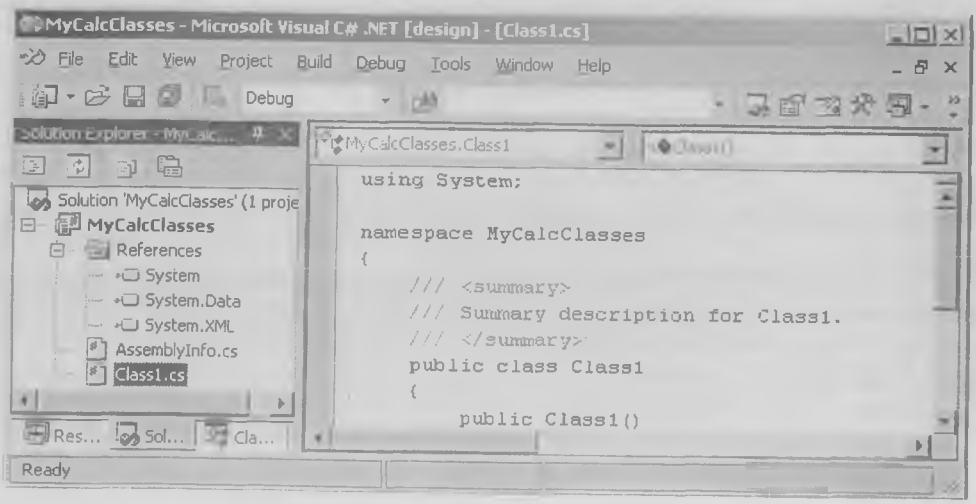


Рис. 1.10.

то есть *предваряем определение класса* MyCalc *ключевым словом public*, и после компиляции командой меню Build | Build Solution получаем *библиотечный файл MyCalcClasses.dll*.

Класс `MyCalc` доступен из любого проекта без необходимости прямого включения в него файла MyCalc.cs. Для иллюстрации создадим проект Proj5 типа Empty Project и внедрим в него единственный файл main.cs следующего содержания:

```
/*-----  File main.cs  -----*/
using MyCalcClasses;

class main
{
  public static int Main( )
  {
    int i = 137, N = 895, Sum;
    Sum = MyCalc.MySum( i, N );
    return 0;
  }
}
```

Попытка скомпилировать проект в таком состоянии порождает сообщение об ошибке:

```
E:\BOOKS\KUDITS\C#Book\Tests\Proj5\main.cs(1): The type or
namespace name 'MyCalcClasses' could not be found (are you
missing a using directive or an assembly reference?)
```

означающее, что пространство имен `MyCalcClasses` для проекта Proj5 недоступно.

Исправляется эта ситуация следующим образом: щелкаем правой клавишей мыши по строке References в левой части окна графической среды компилятора Microsoft Visual C# и из всплывающего контекстного меню выбираем команду Add Reference..., после чего появляется одноименное диалоговое окно (см. рис. 1.11).

С помощью кнопки Browse... находим на диске компьютера и выбираем ранее полученный библиотечный файл MyCalcClasses.dll, имя которого попадает при этом в список в нижней части диалогового окна так, как это показано на рис. 1.11. Нажимаем далее кнопку OK и проект Proj5 получает ссылку на ранее недоступное ему пространство имен `MyCalcClasses` (см. рис. 1.12).

После этого проект Proj5 успешно компилируется. Отметим, что указанная техника ссылки (reference) на динамическую библиотеку приводит не только к автоматическому копированию библиотечного файла MyCalcClasses.dll в рабочий каталог клиентского проекта (то есть использующего эту библиотеку),

но и заставляет графическую среду компилятора Microsoft Visual C# автоматически загружать более свежие версии библиотеки в момент открытия проекта.

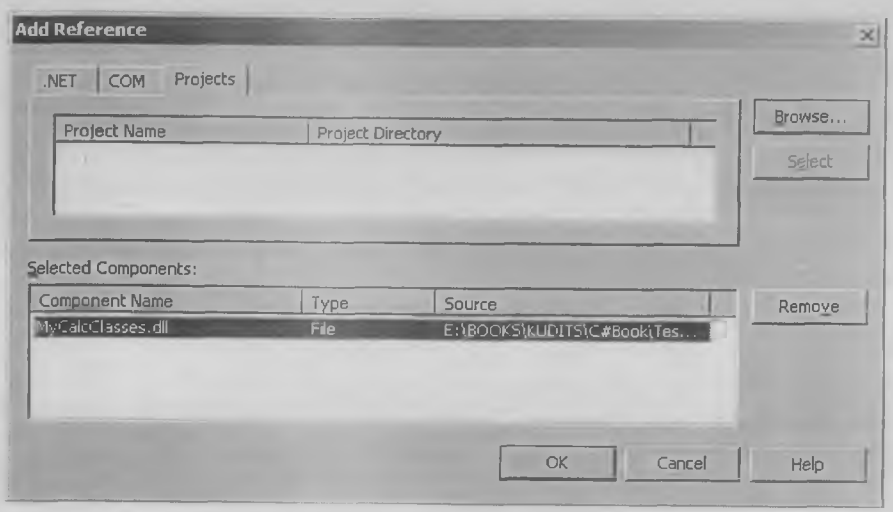


Рис. 1.11.

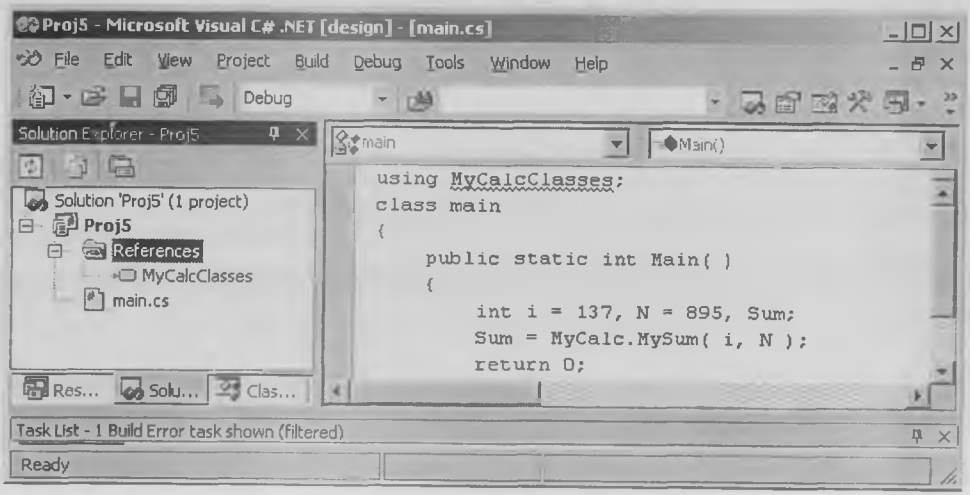


Рис. 1.12.

1.5. Библиотечные классы для математических вычислений и ввода/вывода

Как бы ни были хороши отладочные возможности графической среды компилятора Microsoft Visual C# по наблюдению за величинами переменных программы, все равно в программе нужно предусмотреть вывод основных результатов в ее консольное окно вместе с пояснительными надписями.

Операторов вывода в языке C# нет совсем. Вместо них используются функциональные методы класса Console, входящего в библиотеку классов программной платформы Microsoft NET Framework. Эту библиотеку называют *Framework Class Library*, или сокращенно *FCL*. Всюду далее мы будем ссылаться на эту библиотеку по указанному сокращению. Физически библиотека FCL состоит из совокупности нескольких dll-файлов, расположенных внутри системного каталога \Windows\Microsoft.NET\Framework.

Подчеркнем, что библиотека FCL является одним из важнейших компонентов всей платформы Microsoft NET Framework. Именно богатство классов этой библиотеки и удобство ее использования в программных разработках в значительной степени определяют огромную популярность этой новейшей платформы разработки и исполнения приложений Windows.

Начнем знакомство с библиотекой FCL с класса Console, статические методы (определенные с модификатором `static`) `ReadLine()` и `WriteLine()` которого обеспечивают простейшие способы ввода данных с клавиатуры и вывод результатов на дисплей.

Оба перечисленных метода библиотечного класса Console проще в использовании, чем аналогичные по назначению библиотечные функции `scanf()` и `printf()` из стандартной библиотеки языка C (см. гл. 2 из книги [1]).

Для примера переделаем Листинг 1.1 под ввод значений переменных `x1`, `x2` и `x3` с клавиатуры и под вывод величины их среднего арифметического, то есть переменной `res`, в консольное окно работающей программы на дисплее компьютера. Учтем, что класс `Console` определен в пространстве имен `System`.

Листинг 1.7.

```
using System;

class main
{
  public static int Main()
  {
    double res, x1, x2, x3;
```

```
        // Вывод предупредительного сообщения
        // и ввод трех чисел:
        Console.WriteLine( "Enter 3 numbers:" );
        x1 = Convert.ToDouble( Console.ReadLine() );
        x2 = Convert.ToDouble( Console.ReadLine() );
        x3 = Convert.ToDouble( Console.ReadLine() );

        // Вычисление среднего арифметического
        // и вывод результата на дисплей:
        res = ( x1 + x2 + x3 ) / 3;
        Console.WriteLine( "Average value = {0}", res );

        // Задержка консольного окна на дисплее:
        Console.WriteLine( "Press Enter to exit" );
        Console.ReadLine();
        return 0;
    }
}
```

Поясним код Листинга 1.7 чуть позже, а сейчас сообщим, что он компилируется безошибочно, получаемый при этом загрузочный файл успешно запускается на выполнение, и на рис. 1.13 показан сеанс работы с этой программой, когда пользователь в ответ на приглашение «Enter 3 numbers:» вводит с клавиатуры числа 1, 2 и 3 (заканчивая ввод каждого из перечисленных чисел нажатием клавиши Enter). Затем программа вычисляет среднее арифметическое введенных чисел и выводит результат в консольное окно. После этого программа ждет от пользователя нажатия клавиши Enter на клавиатуре компьютера для завершения своей работы, о чем и предупреждает его сообщением «Press Enter to exit». За счет такого ожидания окно программы не исчезает мгновенно с дисплея компьютера и пользователь может спокойно и сколь угодно долго лицезреть результаты ее работы.

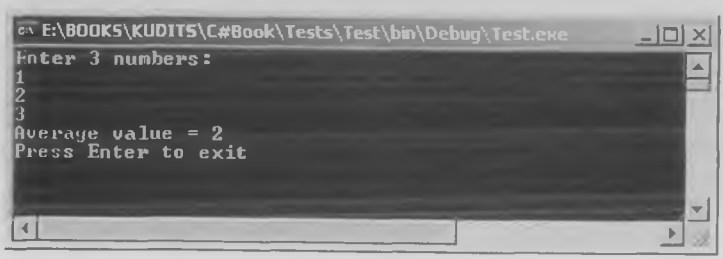


Рис. 1.13.

Теперь вернемся к коду Листинга 1.7. Чтобы иметь доступ к классам из пространства имен `System`, была применена директива

```
using System;
```

Ввод/вывод выполняется статическими методами библиотечного класса Console из указанного пространства имен. Функциональный метод `WriteLine()` предназначен для вывода информации в консольное окно работающей программы. Например, вызов этого метода

```
Console.WriteLine( "Enter 3 numbers:" );
```

с единственным текстовым параметром (буквально заданный текст, как и в языке C, нужно заключать с двух сторон в двойные кавычки) осуществляет вывод текста. А вызов метода `WriteLine()`

```
Console.WriteLine( "Average value = {0}", res );
```

помимо вывода текста, представленного его первым параметром, выводит еще и значение переменной res, причем последнее визуально позиционируется на дисплее точно по месту *поля подстановки* (*placeholder*), обозначенного как `{0}`.

Итак, фигурными скобками внутри строки форматирования (первый параметр метода `WriteLine()`) маркируются поля подстановки, целые числа которых означают порядковый номер, в котором перечислен идентификатор переменной в списке параметров вызова метода `WriteLine()`. Например, следующий фрагмент кода

```
double y0 = 0, y1 = 1, y2 = 2;
Console.WriteLine("First {2} Second {1} Third {0}", y0, y1, y2);
```

выводит в консольное окно программы следующую строку:

```
First 2 Second 1 Third 0
```

то есть на место поля подстановки `{2}` помещается значение переменной `y2`, стоящее вторым (если вести отсчет с нуля) в списке параметров вызова метода `WriteLine()` после строки форматирования.

Нетрудно заметить, что во многих чертах статический метод WriteLine() класса `Console` перекликается с функцией `printf()` из стандартной библиотеки языка C (см. гл. 1 из книги [1]), но он проще, так как не надо явно применять форматирующие символы, такие как `%Lf`, `%d` и так далее. Кроме того, он сам обеспечивает переход на новую экранную строку, так что нет необходимости лишний раз использовать *управляющий символ* `\n`, называемый *символом перевода строки*.

В то же время, предназначенный для ввода с клавиатуры функциональный метод `ReadLine()`, вызываемый без параметров, не имеет ничего общего с аналогичной библиотечной функции языка C – функции `scanf()`, и намного проще ее. Он возвращает введенное с клавиатуры число в так называемой строковой (текстовой) форме, так что нужно еще преобразовать такую строку в число типа `double`. Но это чрезвычайно легко сделать, так как в пространстве имен `System` имеется класс `Convert`, обладающий многочисленными статическими методами для преобразований одних встроенных типов данных языка C# в другие. В нашем случае

```
x1 = Convert.ToDouble( Console.ReadLine() );
```

метод `ToDouble()` класса `Convert` принимает на вход (в качестве фактического параметра вызова) строковое представление введенного с клавиатуры числа (это возвращаемое методом `ReadLine()` класса `Console` значение) и преобразует его в число типа `double`. Последнее и присваивается в данном случае переменной `x1`.

Поскольку окончание ввода с клавиатуры требует нажатия на клавишу Enter, то вызов из Листинга 1.7

```
// Задержка консольного окна на дисплее:
Console.WriteLine( "Press Enter to exit" );
```

предназначен лишь для того, чтобы задержать консольное окно нашей программы на дисплее компьютера, предотвратив его мгновенное исчезновение при выполнении оператора `return 0;` из функции `Main()`.

Ну и последнее, что можно с удовольствием отметить в отношении использования библиотечного класса Console и его методов ввода/вывода. Во-первых, приятным моментом является отсутствие необходимости явным образом организовывать ссылку (строка References из левой части окна компилятора) на библиотечный файл, содержащий классы пространства имен System (см. предыдущий раздел), так как компилятор располагает такой ссылкой по умолчанию. А, во-вторых, dll-файлы с библиотечными классами не сопровождаются параллельными заголовочными файлами (header files), как это имеет место в случае стандартной библиотеки языка C, и где в клиентский код нужно было в обязательном порядке вставлять директиву препроцессора `#include` с именем нужного заголовочного файла – в языке C# отсутствует необходимость в прототипах методов (функций) и указанной директивы препроцессора нет тоже.

Переходим к библиотечным математическим методам, сосредоточенным в качестве статических методов в классе Math из того же самого пространства имен System. В качестве иллюстрации приведем текст программы на C#, вычис-

ляющей корни квадратного уравнения, и являющейся полным аналогом Листинга 2.4 из гл. 2 книги [1] по языку C:

Листинг 1.8.

```
using System;

class main
{
  public static int Main( )
  {
    double A, B, C, D, x1, x2;

    Console.WriteLine( "Enter A, B and C: " );
    A = Convert.ToDouble( Console.ReadLine() );
    B = Convert.ToDouble( Console.ReadLine() );
    C = Convert.ToDouble( Console.ReadLine() );

    D = B * B - 4 * A * C;

    if( D >= 0 && A != 0 )
    {
      x1 = ( -B + Math.Sqrt(D) )/(2*A);
      x2 = ( -B - Math.Sqrt(D) )/(2*A);
      Console.WriteLine( "x1 = {0} x2 = {1}", x1, x2 );
    }
    else
    {
      Console.WriteLine( "Can't solve equation" );
    }

    Console.ReadLine();
    return 0;
  }
}
```

Из Листинга 1.8 видно, что вместо вызова функции `sqrt()` из стандартной библиотеки языка C, мы теперь обращаемся к статическому методу `Sqrt()` класса `Math`.

В процессе работы этой программы мы получим в консольном окне практически ту же «картинку» (см. рис. 1.14), что и показанная на рис. 2.3 в гл. 2 книги [1].

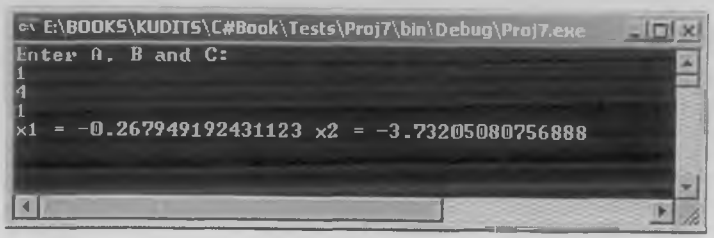


Рис. 1.14.

А теперь повторим в виде программы на языке C# целый *проект по вычислению интегралов методом трапеций*, который в книге [1] по языку C представлен Листингом 2.17 из гл. 2.

Листинг 1.9.

```
using System;

class main
{
  public static int Main( )
  {
    double a, b, eps, S;

    Console.WriteLine( "Enter a, b and eps: " );
    a   = Convert.ToDouble( Console.ReadLine() );
    b   = Convert.ToDouble( Console.ReadLine() );
    eps = Convert.ToDouble( Console.ReadLine() );

    /*-- Testing a, b, eps: --*/
    if( (b-a) <= 0 || eps < 0 )
    {

       Console.WriteLine( "Bad input values" );
       Console.ReadLine();
       return -1;
    }

    S = TrapezSquare( a, b, eps );
    Console.WriteLine( "\nFor a = {0}, b = {1}, eps = {2},"+
                       "S = {3}", a, b, eps, S );
```

```
    Console.ReadLine();
    return 0;
  }
  /*-------------------------------------------------------------*/
  static double TrapezSquare( double a, double b, double eps )
  {
    double res1, res2;
    int N;

    N = 2;
    res1 = MyTrapezSquare( a, b, N );
    N= N * 2;
    res2 = MyTrapezSquare( a, b, N );

    while( Math.Abs( res2 - res1 ) >= eps )
    {
      res1 = res2;
      N = N * 2;
      res2 = MyTrapezSquare( a, b, N );
    }

    return res2;
  }
  /*-------------------------------------------------------------*/
  static double MyTrapezSquare( double a, double b, int N )
  {
    double x, h, S1, S2;

    h = ( b - a ) / N;
    S1 = ( Math.Sin(a) + Math.Sin(b) ) / 2;
    S2 = 0;

    for( x=a+h; x < b; x=x+h )
    {
      S2 = S2 + Math.Sin(x);
    }

    return h * ( S1 + S2 );
  }
}
```

Кроме мелких отличий от параллельного кода на языке C, с большинством из которых мы уже знакомы, можно обратить внимание еще лишь на то, что в языке C# нельзя выполнять запись управляющей строки на разных физических строках дисплея так, как мы это делали в программах на языке C (см. гл. 2 из книги [1]):

```
printf( "\nFor a = %Lf, b = %Lf,"
        " N = %d, S = %Lf",
        a, b, N, S );
```

а нужно в обязательном порядке использовать *знак плюс*:

```
Console.WriteLine( "\nFor a = {0}, b = {1}, eps = {2},"+
                   "S = {3}", a, b, eps, S );
```

Внимание: В языке C#, в отличие от языка C, стоящие рядом друг с другом строки (разделенные лишь пробельными символами, включая перевод строки) автоматически не объединяются (то есть "Some""Text" это не то же самое, что "SomeText")

Указанное отличие от языка C диктуется наличием в языке C# развитого строкового типа данных string, изучению которого будет полностью посвящена гл. 4. Там мы и рассмотрим применение знака плюс для объединения (конкатенации) строк.

А сейчас констатируем, что программное решение на языке C#, представленное в Листинге 1.9, по-прежнему по сути дела *процедурное*, несмотря на классовый антураж, неизбежный при программировании на языке C#.

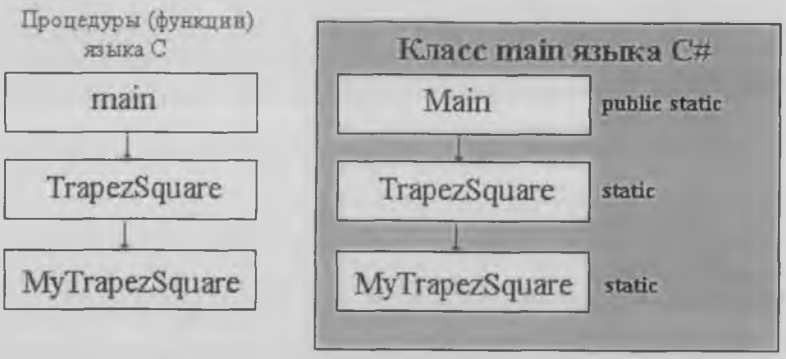


Рис. 1.15.

Рис. 1.15 прекрасно иллюстрирует сделанное выше утверждение о принципиальной идентичности чисто процедурного программного решения на языке C (Листинг 2.17 из книги [1]) и формально классового решения на языке C# из Листинга 1.9. Отсюда следует полезный практический вывод, что процедурные наработки на языке C могут быть относительно легко «переведены на язык C#».

1.6. Перегрузка функций (функциональных классовых методов) и передача параметров с модификаторами ref и out

При всей схожести языков C и C# с точки зрения процедурного программирования, у последнего имеется дополнительная возможность *перегрузки имен функций*, когда разные функции, входящие в одну и ту же область видимости, имеют одинаковое имя и не противоречат друг другу. Функции с одинаковым именем должны иметь разные параметры (в смысле количества параметров и/или их типов), а типы возвращаемых функциями значений в расчет при этом не берутся.

Листинг 1.10.

```
class main
{
  /*--------------------------------------*/
  public static int Main()
  {
    int     x1 = 1, x2 = 2, iRes;
    double  y1 = 1, y2 = 2, dRes;

    iRes = MyMin( x1, x2 );
    dRes = MyMin( y1, y2 );

    return 0;
  }
  /*--------------------------------------*/
  static int MyMin( int x, int y )
  { return ( x < y )? x : y; }
  /*--------------------------------------*/
  static double MyMin( double x, double y )
  { return ( x < y )? x : y; }
  /*--------------------------------------*/
}
```

Ясно, что возможность *перегрузки функций* (*function overloading*) является полезной с практической точки зрения, так как функциям, близким по сути выполняемых ими работ, удобно давать одинаковые «говорящие имена» типа `Find()`, `Sort()`, `Min()` и так далее, вместо того, чтобы выдумывать имена, не имеющие такой прозрачной самоочевидности.

В Листинге 1.10 присутствуют (не мешая друг другу) два перегруженных функциональных метода с одним и тем же именем – `MyMin`. Один из этих мето-

дов ищет минимум из двух целых чисел, а второй – из двух чисел типа `double`. Кроме того, в Листинге 1.10 мы нарочито применили так называемую *тернарную условную операцию* языков C/C++ (см. разд. 4.6 книги [2]), чтобы продемонстрировать ее прекрасную применимость и в программах на языке C#.

В библиотеке FCL перегрузка функциональных классовых методов применяется весьма интенсивно. Например, широчайше применяемый на практике функциональный метод `WriteLine()` класса `Console` – это вовсе не единственный метод, а довольно большой набор перегруженных методов, о чем можно узнать подробнее из встроенной в компилятор Microsoft Visual C# электронной справочной системы. Для этого нужно поместить текстовый курсор внутрь имени метода и нажать клавишу F1, и в результате появится окно справочной системы с ответом на наш вопрос (см. рис. 1.16).

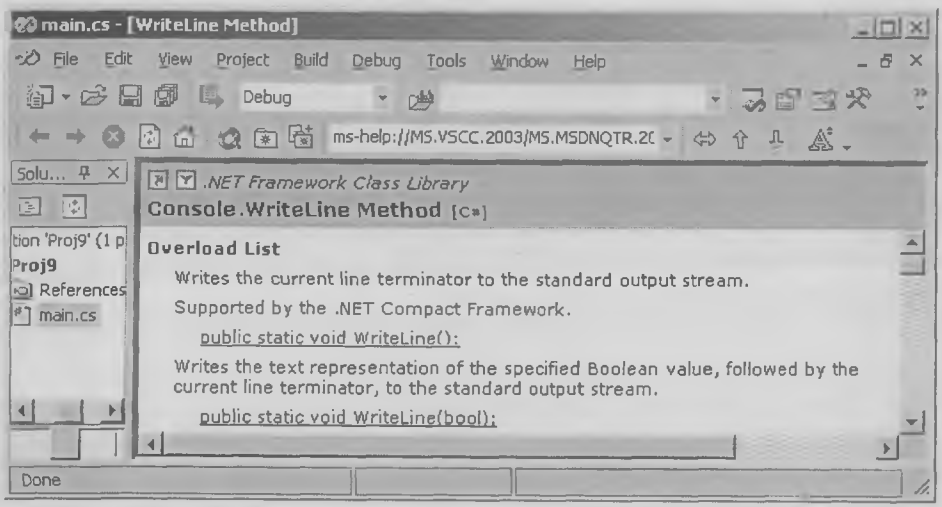


Рис. 1.16.

Из рис. 1.16 видно, что справочная информация приводится по всему списку перегруженных методов `WriteLine()` (*Overload List* – список перегрузок).

Завершая первую главу пособия отметим, что выявленная в предыдущем разделе похожесть синтаксиса и роли функций в языках C и C# не оставляет сомнений в том, что параметры функциональным методам в языке C# также *передаются по значению* (см. разд. 2.6 из книги [1]), то есть во время своей работы функции имеют дело с *копиями* фактических параметров. Это иногда порождает

проблемы, если нам, например, надо написать функцию MySwap(), которая должна обменять значения двух своих параметров.

Можно, конечно, написать иной вариант функции, которая вообще не принимает параметров, а обменивает значения у двух глобальных (по отношению к определению функции) переменных, как это ранее было показано в разд. 2.6 из книги [1].

Листинг 1.11.

```
using System;

class main
{
  static int M = 1, N = 5;

  /*-------------------------------------------*/
  public static int Main()
  {
    MySwap();
    Console.WriteLine( "M = {0} N = {1}", M, N );

    Console.ReadLine();
    return 0;
  }
  /*-------------------------------------------*/
  static void MySwap( )
  {
    int temp;
    temp = M;
    M = N;
    N = temp;
  }
  /*-------------------------------------------*/
}
```

В Листинге 1.11 целые переменные M и N объявлены с модификатором static вне определений классовых методов Main() и MySwap(), то есть глобально по отношению к этим методам. В теле функционального метода MySwap() значения этих переменных обмениваются местами, и в этом можно убедиться воочию по «картинке» консольного окна данной программы (см. рис. 1.17).

Однако, даже в языке C решение проблемы с помощью глобальных переменных не приветствуется по многим причинам, а в языке C#, можно сказать, что оно и того хуже.

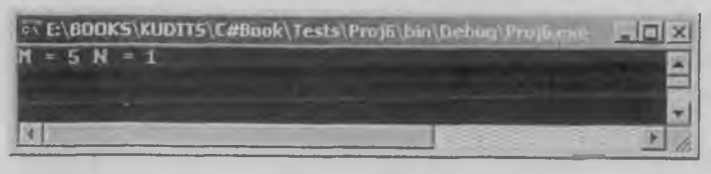


Рис. 1.17.

Более типичным и стандартным вариантом решения проблемы в языке C является применение *указателей* (см. разд. 6.5 из книги [1]). Но в языке C# указатели оставлены как самое последнее средство, к которому стоит прибегать в самом что ни на есть крайнем случае, «практически на грани фола». Мы, тем не менее, покажем, как это делается.

Листинг 1.12.

```
using System;

class main
{
  public static unsafe int Main()
  {
    int M = 1, N = 5;

    // Передаем адреса переменных M и N:
    MySwap( &M, &N );
    Console.WriteLine( "M = {0} N = {1}", M, N );

    Console.ReadLine();
    return 0;
  }
  /*-----------------------------------------*/
  static unsafe void MySwap( int* pM, int* pN )
  {
    int temp;
    temp = *pM;
    *pM = *pN;
    *pN = temp;
  }
}
```

В Листинге 1.12 стоят «пугающие модификаторы» *unsafe* (*ненадежный, опасный*), которыми в программах на языке C# требуется в обязательном порядке маркировать все методы, вознамерившиеся поработать с указательными типами или прямыми адресами (напомним, что операция & – это операция взятия адреса).

Более того, даже этого недостаточно – нужно еще переопределить умолчательные настройки компилятора, для чего командой меню Project | Properties... вызывается диалоговое окно Property Pages, в котором при активной (подсвеченной) строке Build из левой его части нужно выставить свойство Allow Unsafe Code Blocks в True (см. рис. 1.18).

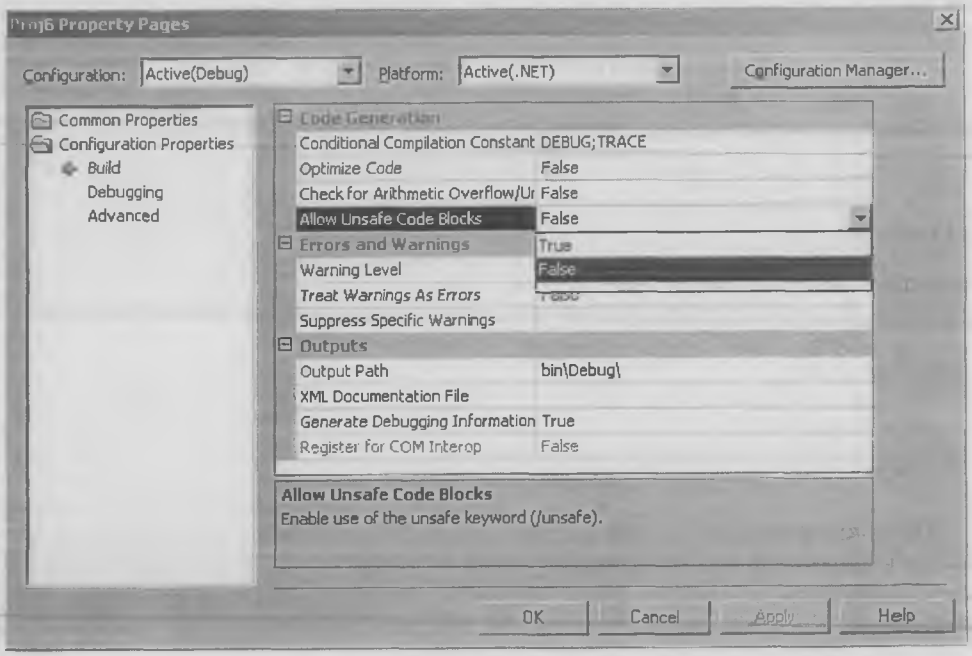


Рис. 1.18.

В общем ясно, что решение с указателями в языке C# практически «вне закона».

Наиболее приемлемое решение проблемы в языке C# ближе всего стоит к *передаче параметров по ссылке* в стиле языка C++ (см. [2]). Но по сравнению с языком C++ этот механизм в языке C# выполнен изящнее, нагляднее и проще.

Если мы хотим, чтобы функция работала не с копиями передаваемых ей во время вызова фактических переменных, а с самими этими переменными, нужно передавать ей адреса этих переменных, но не напрямую, как это было только что показано для случая указателей, а неявно, применяя ключевое слово *ref*.

Листинг 1.13.

```
using System;

class main
{
  public static int Main()
  {
    int M = 1, N = 5;

    // Передаем переменные M и N по ссылке:
    MySwap( ref M, ref N );
    Console.WriteLine( "M = {0} N = {1}", M, N );

    Console.ReadLine();
    return 0;
  }
  /*-----------------------------------------*/
  static void MySwap( ref int M, ref int N )
  {
    int temp;
    temp = M;
    M = N;
    N = temp;
  }
}
```

Промаркировав ключевым словом `ref` как формальные параметры в определении функционального метода `MySwap()`, так и фактические параметры его вызова, мы добились нужного нам эффекта, причем самым штатным для языка C# образом.

Глава 2. Классы языка C# как типы и объекты этих типов

2.1. Классовые поля данных и функциональные методы для их обработки

В гл. 1 классы языка C# выступали в роли своего рода оберток (wrappers) для статических функциональных методов. Использование таких классов не противоречит основным принципам процедурного программирования, поэтому многие вычислительные процедурные решения, ранее разработанные на языке C и приведенные в книге [1], были легко с практической точки зрения и вполне естественно «переведены на язык C#».

Однако даже в этих рамках было видно, что классы языка C# позволяют обеспечить их пользователю (то есть клиентскому коду) дополнительные удобства, что и было продемонстрировано на примере класса `Console`, статические методы `WriteLine()` и `ReadLine()` которого действительно удобнее в применении, чем аналогичные по назначению функции `printf()` и `scanf()` из стандартной библиотеки языка C.

Первопричина дополнительных возможностей классовых методов языка C#, особенно нестатических, кроется в наличии у класса полей данных, обеспечивающих объективно существующую, но скрытую от пользователю сложную внутреннюю структуру, которая и поддерживает дополнительную богатую функциональность классовых методов. Фактически, с помощью классов языка C# строятся высокоуровневые (далекие от устройства компьютерных микропроцессоров) *абстрактные типы данных*, в наилучшей степени приспособленные для решения задач конкретных предметных областей.

Настоящая глава посвящена изучению базовых синтаксических конструкций языка C#, позволяющих строить абстрактные типы данных и создавать и использовать объекты (переменные) этих типов. В ней мы не будем стремиться изучить классы языка C# во всей их мощи, включая довольно сложные синтаксические элементы и архитектурные концепции, а сосредоточимся лишь на самых важных основах, которые, конечно, «всему голова». Продолжим же мы эту тему в Ч. II настоящего пособия, где столкнемся с полным джентльменским набором средств так называемого объектно-ориентированного программирования.

Начнем с *классовых полей данных*. Для иллюстрации спроектируем класс `MyComplex`, имеющий три поля данных типа `double` с именами `R`, `I` и `Length`, и несколько методов для их обработки, имитирующих поведение *комплексных чисел* (с комплексными числами в минимальной степени знакомятся в средней

школе, а в технических ВУЗах их используют весьма интенсивно – см., например, учебник [4]).

Листинг 2.1.

```
class MyComplex
{
  // Methods:
  public void SetR( double x )
  { R = x; Length = Math.Sqrt( R*R + I*I ); }
  public void SetI( double y )
  { I = y; Length = Math.Sqrt( R*R + I*I ); }

  public void Print( )
  { Console.WriteLine( "Complex number = {0}+{1}i", R, I ); }

  static public void Sum(MyComplex z1,MyComplex z2,MyComplex z3)
  {
    z3.R = z1.R + z2.R; z3.I = z1.I + z2.I;
    z3.Length = Math.Sqrt( z3.R * z3.R + z3.I * z3.I );
  }

  // Data fields:
  private double  R;
  private double  I;
  public  double  Length;
}
```

У класса `MyComplex` объявлены три поля данных типа `double` с именами `R`, `I` и `Length`, соответственно. Они находятся в конце определения класса `MyComplex` и для наглядности предваряются комментарием `//Data fields`. *Местоположение объявления полей данных* в рамках определения содержащего их класса *не имеет принципиального значения*, однако лучше придерживаться некоторого порядка (хаос здесь скорее повредит, чем поможет).

Поле данных `R` предназначено для хранения величины действительной части комплексного числа (напоминаем, что `y` комплексного числа `x+iy` значение `x` называется его действительной частью, а значение `y` – мнимой частью), а поле `I` – для хранения мнимой части. Поле `Length` спроектировано под хранение длины вектора в геометрической интерпретации комплексных чисел [4], и которая вычисляется по формуле «корень квадратный из суммы квадратов `x` и `y`» (вычисления по этой формуле запрограммированы в теле методов `SetR()`, `SetI()` и `Sum()` из Листинга 2.1).

Поля `R` и `I` снабжены модификатором private, что означает возможность прямого обращения к этим полям *лишь из методов класса MyComplex*. Поле же

`Length` *доступно отовсюду*, так как имеет *режим доступа public*. Если у некоторого члена класса языка C# модификатор режима доступа `private` или `public` явно не указан, то по умолчанию предполагается `private`.

Внимание: В отличие от языка C++ (см. [2]) в языке C# модификаторы режима доступа public/private маркируют собой в точности один член класса (метод или поле данных), а не целую секцию вплоть до появления нового маркера. Кроме того, в языке C# отсутствует понятие друзей класса (и ключевого слова friend тоже нет совсем).

Дальнейшие пояснения по содержимому Листинга 2.1 в первую очередь предназначены читателям, не знакомым с языком C++, а для программистов на C++ здесь, пожалуй, все и так понятно. Начнем дополнительные пояснения с констатации факта, что объявления классовых полей данных выглядят точно также, как операторы определения локальных в функции переменных, но расположены они совсем в иных местах, а именно внутри тела определения класса, но вне тел его функциональных методов. При компиляции класса программой-компилятором под поля данных не резервируются области памяти, как это делается в отношении локальных переменных функциональных методов. Объявление *классовых полей данных* (*без модификатора static*) означает передачу компилятору сведений об устройстве *объектов класса* – в данном случае каждый объект класса MyComplex после своего создания (об этом см. следующий разд.) располагается в участке компьютерной памяти объемом, достаточным для размещения трех чисел типа double. А доступ к отдельным *полям объекта* по именам полей класса продемонстрирован в теле статического метода `Sum()` из Листинга 2.1.

До сих пор в гл. 1 мы лишь единыжды определяли поля данных в классе (см. Листинг 1.11), причем именно *статические поля данных*, которые были предназначены для имитации глобальных переменных языка C, и которые не имеют отношения к устройству объектов класса, так как существуют в единственном экземпляре и, естественно, вне классовых объектов.

Статические поля данных могут обрабатываться *только статическими классовыми методами*. Кроме статических полей данных статические методы получают еще информацию в виде своих параметров, например, в Листинге 2.1 статических полей данных нет, но статический метод `Sum()` работает над своими входными параметрами.

В свою очередь, *поля данных без модификатора static могут обрабатываться только нестатическими классовыми методами*, такими как `SetR()`, `SetI()` и `Print()` из Листинга 2.1. В теле этих методов ссылка на нестатические классовые поля данных происходит просто по их имени, то есть `R`, `I` или `Length`. Во время выполнения нестатических методов перечисленные идентификаторы представляют собой значения составляющих классовый объект полей (участков памяти).

2.2. Создание клиентским кодом классовых объектов и их использование

Как мы поняли из предыдущего раздела, определение класса с нестатическими полями данных и, соответственно, с нестатическими функциональными методами для их обработки представляет собой точный план устройства объектов класса и их функциональности, в совокупности составляющей суть нового, *пользовательского абстрактного типа данных*. Использование абстрактных типов данных, как и встроенных типов, предполагает создание *переменных* этого типа, которые в случае пользовательских абстрактных типов принято называть *объектами* (или *экземплярами класса*).

Для иллюстрации будем создавать объекты пользовательского типа `MyComplex` в теле функции `Main()`, которая и выступит, тем самым, в роли *клиента класса MyComplex*. Приведем сначала лишь фрагмент этой функции, имеющий отношение к созданию объектов абстрактных типов данных:

```
static public int Main( )
{
  double    x   =  0;       /*1*/
  MyComplex rZ  =  null;     /*2*/
```

Данный фрагмент прекрасно иллюстрирует как сходство, так и различие в создании переменных встроенного типа (здесь это тип `double`) и объектов пользовательского типа (здесь это класс `MyComplex`).

Оператор создания переменной x типа double, стоящий в строке `/*1*/`, заставляет компилятор выделить в стеке (это часть компьютерной памяти для доступа из прикладной программы по стековым командам микропроцессора) кусок памяти размера, достаточного для хранения значений типа `double`. Про такой тип выделения памяти говорят как о *статическом* выделении, или о выделении *на этапе компиляции*.

Стековые переменные являются «короткоживущими», так как по выходе из функции содержимое ее стека теряется (перезаписывается другими функциями, вызываемыми на выполнение). Для сохранения значений переменных встроенных типов между вызовами функций их приходится передавать «по значению» в качестве параметров функций (то есть копировать с места на место). Часть из рассмотренных особенностей существования переменных встроенных типов легла в основу решения об ином варианте создания в языке C# объектов классовых типов.

В языке C# объекты классовых типов создаются *только динамически* и только в так называемой *свободной памяти* (свободной от машинного кода програм-

мы, ее данных и стека; по-другому говорят – в *куче*). Из сказанного вытекает, что в строке `/*2*/` никакого объекта типа `MyComplex` еще не создано, а создана лишь стековая переменная `rZ`, предназначенная для хранения адреса будущего объекта типа `MyComplex`. Таким образом, строка `/*2*/` заставляет компилятор статически выделить в стеке функции память размера, достаточного для хранения любого компьютерного адреса (на машинном уровне для микропроцессоров архитектуры Intel IA-32, известных как Pentium, для хранения адресов отводится ровно 4 байта) и прописать эту память условным значением *null* (ключевое слово языка C#), которое исполняющая среда CLR трактует как *несуществующий адрес*. Можно предположить, что в качестве такого магического значения в глубине среды CLR выступает просто число 0, но опираться на такие предположения мы в любом случае не намерены. Просят нас писать `null` – будем писать `null` (нет проблем). Принято называть созданную таким образом переменную `rZ` *ссылкой на классовые объекты* (*object reference*) типа `MyComplex`.

После данных объяснений строка `/*2*/` уже не кажется сильно отличающейся от строки `/*1*/`.

Динамическое создание в свободной компьютерной памяти реального объекта типа `MyComplex` (то есть выделение куска памяти размером, равным *суммарному объему классовых нестатических полей данных*) выполняется по *операции new* (ключевое слово языка C#) во время исполнения скомпилированной компьютерной программы. Возвращаемое операцией new значение является начальным адресом выделенного блока памяти и подлежит сохранению в созданной ранее объектной ссылке:

```
static public int Main( )
{
  double    x   =  0;        /*1*/
  MyComplex rZ  =  null;     /*2*/
  rZ = new MyComplex();      /*3*/
```

Итак, динамическое создание классового объекта типа `MyComplex` и прикрепление его к объектной ссылке `rZ` производится именно в строке `/*3*/` представленного учебного фрагмента.

Для компактизации исходного кода программы создание объектной (классовой) ссылки и операцию выделения реальной памяти под объект можно соединить в рамках одного оператора языка C#:

```
MyComplex rZ  =  new MyClass();
```

К объектной ссылке можно применять единственную операцию – операцию «точка» для доступа к полям адресуемого ссылкой объекта, что уже было ранее в Листинге 2.1 продемонстрировано в коде метода Sum().

Внимание: В отличие от языка C++ (см. [2,3]), где динамически создаваемые классовые объекты адресуются *указателями*, над которыми можно выполнять разные операции, в том числе арифметические, в языке C# таких вольностей нет: динамические классовые объекты адресуются *ссылками языка C#* (близкими, но не идентичными ссылкам языка C++), над которыми никакие арифметические и иные операции, отличные от операции «точка», недопустимы.

Ближе к концу настоящего раздела мы еще раз вернемся к синтаксису создания классовых объектов в языке C# и особенностей работы с ними, а сейчас приведем полный и законченный рабочий пример, для чего создадим новый проект типа Empty Project и внедрим в него два исходных файла – MyComplex.cs и main.cs.

Файл MyComplex.cs будет содержать определение класса `MyComplex` из Листинга 2.1, дополненное необходимой директивой using System. А в файле main.cs будет располагаться код класса `main` со стартовой функцией `Main()`, то есть клиентский по отношению к классу `MyComplex` код:

Листинг 2.2.

```
/*-----  File MyComplex.cs  -----*/
using System;

class MyComplex
{
  // Methods:
  public void SetR( double x )
  { R = x; Length = Math.Sqrt( R*R + I*I ); }
  public void SetI( double y )
  { I = y; Length = Math.Sqrt( R*R + I*I ); }

  public void Print( )
  { Console.WriteLine( "Complex number = {0}+{1}i", R, I ); }

  static public void Sum(MyComplex z1,MyComplex z2,MyComplex z3)
  {
    z3.R = z1.R + z2.R; z3.I = z1.I + z2.I;
    z3.Length = Math.Sqrt( z3.R * z3.R + z3.I * z3.I );
  }

  // Data fields:
  private double  R;
  private double  I;
  public  double  Length;
}
```

```
/*-----  File main.cs  -----*/
using System;

class main
{
  static public int Main( )                                    /*1*/
  {                                                            /*2*/
    // Create 3 Complex objects:                               /*3*/
    MyComplex z1 = new MyComplex();                            /*4*/
    MyComplex z2 = new MyComplex();                            /*5*/
    MyComplex z3 = new MyComplex();                            /*6*/
                                                               /*7*/
    // Process "complex" numbers:                              /*8*/
    z1.SetR( 1 ); z1.SetI( 2 );                                /*9*/
    z2.SetR( 3 ); z2.SetI( 4 );                                /*10*/
                                                               /*11*/
    MyComplex.Sum( z1, z2, z3 );                               /*12*/
                                                               /*13*/
    z3.Print();                                                /*14*/
    Console.WriteLine( "Length = {0}", z3.Length );            /*15*/
                                                               /*16*/
    // Screen delay:                                           /*17*/
    Console.ReadLine();                                        /*18*/
    return 0;                                                  /*19*/
  }                                                            /*20*/
}
```

Итак, в клиентском коде в строках `/*4*/–/*6*/` создаются три классовых объекта типа `MyComplex`, адресуемые объектными ссылками `z1`, `z2` и `z3`.

От имени этих объектных ссылок можно вызывать классовые нестатические методы (строки `/*9*/–/*10*/` и `/*14*/–/*15*/`), а также обращаться к открытому нестатическому полю данных `Length` (строка `/*15*/`).

Обращение к статическому методу `Sum()` производится от имени класса, а не от имени объектных ссылок.

В целом, можно сказать, что в функции `Main()` мы осуществляем с помощью разработанного пользовательского (а не библиотечного) класса `MyComplex` относительно удобные манипуляции с комплексными числами. Для первого комплексного числа (объектная ссылка `z1`) мы методами `SetR()` и `SetI()` устанавливаем вещественную и мнимую части равными 1 и 2, соответственно. Для второго комплексного числа – 2 и 4, соответственно. Затем статическим методом `Sum()` мы складываем эти числа, и число `z3` хранит результат суммирования. Этот результата выводится на дисплей методом `Print()`, а также осуществляется доступ к полю `Length` результата.

Исходя из наших знаний комплексных чисел (см. [4]), мы ожидаем получить комплексное число `4+6i`. Компилируя Листинг 2.2 и запуская полученное NET-приложение на выполнение, в его консольном окне получаем следующую «картинку» (см. рис. 2.1).

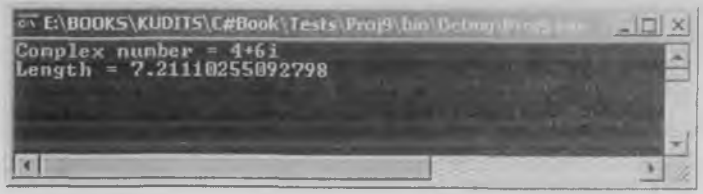


Рис. 2.1.

Полученный результат полностью совпадает с нашими ожиданиями, значит код работает правильно. В частности, метод Sum(), получая в качестве параметров ссылки на классовые объекты самым обычным образом, то есть «по значению» (ведь модификатор `ref` не применяется), через доступную ему копию ссылки `z3` преспокойно меняет сам этот объект. Именно поэтому мы и получаем после отработки метода `Sum()` новое значение объекта, адресуемого ссылкой z3.

Как мы и обещали выше, возвращаемся к рассмотрению тонкостей создания классовых объектов в языке C# и работы с ними. Начнем с того, что знатоки языка C++ после знакомства с клиентским кодом, то есть функцией `Main()` из Листинга 2.2, наверняка решат, что мы забыли в конце работы освободить память, выделенную под классовые объекты (комплексные числа). Однако на самом деле мы ничего не забыли. Дело в том, что в рамках исполняющей среды CLR память выделяется динамически в так называемой *управляемой куче* (*managed heap*), находящейся под управлением самой CLR. Именно эта среда и отвечает за освобождение памяти, выполняя автоматически так называемую *сборку мусора* (*garbage collection*).

Внимание: В отличие от языка C++, где операция new должна иметь парную ей операцию delete, в языке C# нет нужды в явном освобождении динамически выделенной памяти, и операции (ключевого слова) delete в этом языке нет вообще.

Пожалуй, при переходе от языка C++ к языку C# к этому привыкнуть труднее всего, также как и к коду следующего вида

```
MyComplex z1 = new MyComplex(); MyComplex z2 = new MyComplex();
z1 = z2;  /*1*/
```

где в строке /*1*/ друг другу присваиваются значения объектных ссылок, а не самих классовых объектов (и после чего обе ссылки z1 и z2 «глядят» на второй объект, а первый становится «бессылочным» и его дальнейшая судьба фактически препоручается сборщику мусора среды CLR).

Зато программистам на языке C++ понятно использование круглых скобок в строках с операцией new. Они-то знают, что здесь неявным образом вызывается так называемый *конструктор по умолчанию* (*default constructor*). В языке C++ в тех случаях, когда в классе явным образом конструктор «по умолчанию» (то есть без входных параметров) не определяется, его код автоматически генерируется компилятором.

Конструкторы, то есть специальные методы классов, есть и в языке C#. И в нем так же, как и в языке C++, предусмотрено автоматическое генерирование компилятором отсутствующего явным образом в исходном коде конструктора «по умолчанию». Более того, он даже более функционален, чем его аналог из языка C++, так как в нем автоматически поля данных класса инициализируются нулями. К подробному изучению классовых конструкторов мы перейдем прямо в следующем разделе.

Подытожим сделанное в настоящем разделе. Класс MyComplex сформировал абстрактный тип данных «комплексные числа» со скрытым внутренним устройством (поля R и I объявлены с модификатором доступа private) и простой работой клиента с объектами этого класса (то есть с комплексными числами) через набор открытых классовых методов и через одно открытое поле данных (поле Length объявлено с модификатором public). Пользователь (программист функции Main() в данном случае) может даже не читать содержимого файла MyComplex.cs – ему достаточно общего математического представления о комплексных числах [4] и справочных сведений об открытых функциональных методах и открытых полях данных этого класса.

В общем, абстрактный тип данных с самой первой попытки получился приемлемым, но есть и пожелания дальнейших улучшений. Во-первых, неплохо обеспечить возможность сразу создавать комплексные числа на основе значений их вещественных и мнимых частей. Это будет нами выполнено в следующем разделе путем явного программирования классовых конструкторов. Кроме того, не очень-то удобен пока что механизм сложения комплексных чисел – через статический метод Sum(). Лучше бы иметь возможность складывать комплексные числа операционно, то есть с помощью значка плюс. Доработка класса MyComplex в этом направлении касается перегрузки операций в классах языка C#, и будет выполнена нами ниже в разд. 2.4. После всех указанных доработок в конце

настоящей главы мы будем располагать учебной моделью «настоящего» класса – самоочевидного, элегантного и мощного.

А сейчас заметим, что классы, содержащие одни лишь статические методы и статические поля данных, не столько формируют новые абстрактные типы данных, сколько просто группируют функции по принципу их логического сходства и одинаковой области применения.

Класс, содержащий стартовую функцию `Main()` – это вообще вспомогательный инфраструктурный элемент программы на языке C#. Поэтому вряд ли стоит сильно «накручивать» этот класс, нашпиговывая его нестатическими полями данных и аналогичными методами (хотя, все может быть: как в известном фильме – «никогда не говори *никогда*»). Условно его можно назвать классом, в целом олицетворяющим все программное приложение.

2.3. Специальные методы классов: конструкторы

К специальным относятся классовые методы, которые и определяются, и вызываются с некоторыми особенностями. Например, особенность вызова *конструкторов* (*constructors*), то есть специальных классовых методов, предназначенных для удобной инициализации классовых объектов в момент их создания операцией new, мы уже наблюдали выше в Листинге 2.2.

Особенностью определения конструкторов является полное *отсутствие* в заголовках их определения *возвращаемых значений* (нельзя указывать даже void), а также их *имя*, стопроцентно *совпадающее с именем содержащего их класса*.

Конструкторов в рамках одного и того же класса может быть определено сколь угодно много (то есть их можно перегружать), а отличаться они должны по типу и количеству параметров. И еще очень важно отметить, что если среди явно определенных в классе конструкторов нет умолчательного (без параметров), то его уже и компилятор не предоставляет.

Изменим класс `MyComplex` из Листинга 2.2 в том отношении, что добавим определения двух конструкторов:

Листинг 2.3.

```
/*-----  File MyComplex.cs  -----*/
using System;

class MyComplex
{
  // Constructors:
```

```
  public MyComplex( ){ R = I = Length = 0; }
  public MyComplex( double r, double i )
  { R = r; I = i; Length = Math.Sqrt( R*R + I*I ); }

  // Methods:
  public void SetR( double x )
  { R = x; Length = Math.Sqrt( R*R + I*I ); }
  public void SetI( double y )
  { I = y; Length = Math.Sqrt( R*R + I*I ); }

  public void Print( )
  { Console.WriteLine( "Complex number = {0}+{1}i", R, I ); }

  static public void Sum(MyComplex z1,MyComplex z2,MyComplex z3)
  {
    z3.R = z1.R + z2.R; z3.I = z1.I + z2.I;
    z3.Length = Math.Sqrt( z3.R * z3.R + z3.I * z3.I );
  }

  // Data fields:
  private double  R;
  private double  I;
  public  double  Length;
}

/*-----  File main.cs  -----*/
using System;

class main
{
  static public int Main( )
  {
    // Create 3 Complex objects:
    MyComplex z1 = new MyComplex( 1, 2 );
    MyComplex z2 = new MyComplex( 3, 4 );
    MyComplex z3 = new MyComplex();

    // Process "complex" numbers:
    //z1.SetR( 1 ); z1.SetI( 2 );
    //z2.SetR( 3 ); z2.SetI( 4 );

    MyComplex.Sum( z1, z2, z3 );

    z3.Print();
    Console.WriteLine( "Length = {0}", z3.Length );
```

```
    // Screen delay:
    Console.ReadLine();
    return 0;
  }
}
```

Соответственно, клиентский код мы изменили так, что он для создания комплексных чисел z1 и z2 воспользовался классовым конструктором с двумя входными параметрами типа double, в результате чего отпала необходимость для инициализации вещественной и мнимой частей создаваемых комплексных чисел вызывать методы SetR() и SetI() (в Листинге 2.3 мы просто закомментировали их вызов).

В результате клиентский код стал компактнее, нагляднее и элегантнее, а пользовательский тип данных «комплексные числа», представленный классом (типом) MyComplex, стал прозрачнее и ближе ко встроенным типам, ибо теперь его объекты инициализируются прямо в месте их создания, как это можно и часто нужно делать в отношении переменных встроенных типов. Дальнейшее совершенствование класса MyComplex мы продолжим в следующем разделе, где введем перегруженную в этом классе операцию «+» (операция суммирования).

В отношении методов SetR() и SetI() теперь можно сказать, что они нужны лишь для изменения вещественных и мнимых частей комплексных чисел при возникновении такой необходимости в процессе обработки этих чисел.

Вернемся к конструкторам и введем понятие *копирующего конструктора* (*copy constructor*)– конструктора, который строит объект класса по уже имеющемуся объекту класса, и который он получает в качестве параметра (напоминаем, что реально в качестве фактического параметра вызова передается ссылка на объект):

```
public MyComplex( MyComplex z )
{ R = z.R; I = z.I; Length = z.Length; }
```

Для класса MyComplex столь тривиального копирующего конструктора, когда все поля создаваемого объекта копируются из аналогичных полей ранее созданного объекта, абсолютно достаточно ввиду простого внутреннего устройства комплексных чисел (объектов класса MyComplex). Вот доказательство сего очевидного факта:

```
/*-----  File main.cs  -----*/
using System;
class main
{
  static public int Main( )
```

```
  {
    MyComplex z1 = new MyComplex( 1, 2 );   /*1*/
    MyComplex z2 = new MyComplex( z1 );     /*2*/
    z2.Print();                             /*3*/
    Console.ReadLine();                     /*4*/
    return 0;                               /*5*/
  }
}
```

Представленный клиентский код создает в строке /*2*/ комплексное число z2 «по образу и подобию» ранее созданного комплексного числа z1 (то есть, применяя копирующий конструктор класса MyComplex) и выдает вполне ожидаемые (правильные) результаты в свое консольное окно (см. рис. 2.2).

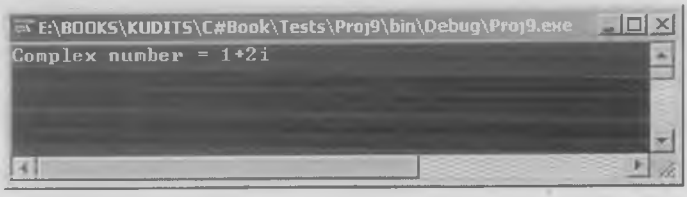


Рис. 2.2.

Столь простой копирующий конструктор не всегда хорош. Он чаще всего не подходит в ситуациях, когда класс содержит поля данных, имеющие тип некоторого иного класса (совсем точно – тип объектных ссылок на иной класс). О таких более сложных случаях у нас речь пойдет в последующих главах.

Закончим настоящий раздел двумя короткими замечаниями. Во-первых, в языке C# (в отличие от языка C++) помимо конструкторов допускается и *иная форма инициализации полей классовых объектов* при их создании, в чем-то менее последовательная, но более короткая с точки зрения объема требуемых от программиста записей:

```
class Test
{ public Test()
  { x = 7; }
  private int x = 6;     /*1*/
}
class main
{ static public int Main( )
  { Test obj = new Test();
```

```
    return 0;
  }
}
```

Эта форма инициализации использует знак операции «=» и показана в представленном учебном фрагменте в строке `/*1*/`. Поставим точки останова (breakpoints) в тексте рассматриваемой программы так, как показано на рис. 2.3.

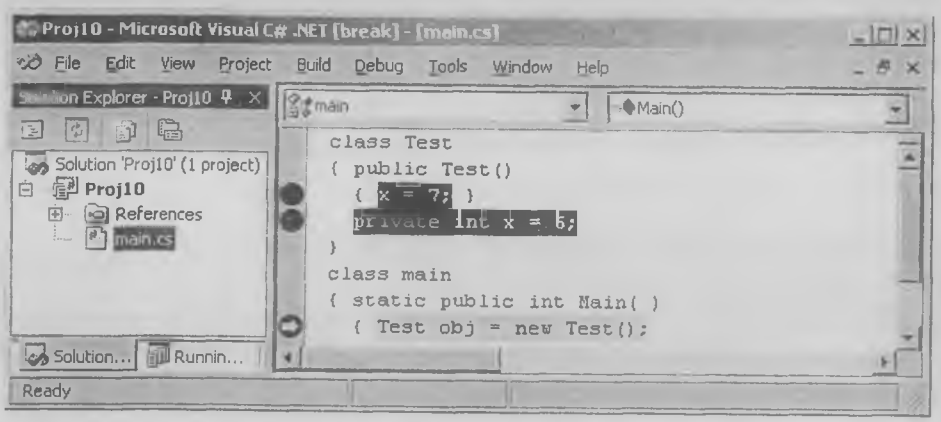


Рис. 2.3.

Пошагово выполняя под отладчиком эту программу, убеждаемся, что динамическое создание объекта типа `Test` в клиентском коде приводит сначала к исполнению инициализации из строки `/*1*/`, а затем осуществляется вызов традиционного конструктора. Так что эти способы сосуществует параллельно, конкурируют между собой и мы их в такой связке использовать в нашем пособии не будем (однако в стороннем опубликованном коде встречается использование как одного из двух этих вариантов, так и их совместное использование).

Второе замечание касается инициализации статических полей данных. Для этой цели в языке C# (в отличие от языка C++) предусмотрены отдельные *статические конструкторы*:

```
class Test
{ static Test()
  { y = 5; }
  public Test(){ x = 7; }
  private int x;
  static private int y;
}
```

```
class main
{ static public int Main( )
  { Test obj = new Test();
    return 0;
  }
}
```

Для статических конструкторов нельзя применять модификаторы режима доступа (`private` или `public`). С помощью прохода представленного учебного фрагмента под отладчиком убеждаемся, что *статический конструктор всегда выполняется раньше, чем нестатический конструктор*. При этом в клиентском коде нет даже намека на вызов статических конструкторов класса. За вызов статических конструкторов отвечает среда CLR, под управлением которой, как мы знаем, NET-приложения и выполняются.

2.4. Специальные методы классов: перегруженные операции

Сейчас мы выполним дальнейшее усовершенствование класса `MyComplex`, заменив в нем статический метод `Sum()` на так называемую перегруженную операцию, конкретно – операцию «+».

Перегруженные в классах операции относятся к специальным методам, которые и определяются и вызываются с некоторыми особенностями по сравнению со стандартными случаями. Во-первых, эти метода *обязаны быть статическими*. Другой особенностью определения перегруженных методов-операций является нетрадиционная схема формирования их имен – нужно после *ключевого слова operator* писать стандартный знак операции. Для двухоперандных операций (к которым операция суммирования и относится) перегружаемые операции-методы должны иметь два входных параметра.

Вот определение метода-операции «+» для класса `MyComplex`, где идентификатором `lOp` обозначен левый операнд этой операции, а идентификатором `rOp` – ее правый операнд:

```
static public MyComplex operator+( MyComplex lOp, MyComplex rOp )
{
  return new MyComplex( lOp.R + rOp.R, lOp.I + rOp.I );
}
```

Этот метод создает временный объект типа `MyComplex`, используя конструктор с двумя параметрами, которому передаются начальные значения веще-

ственной и мнимой частей, сформированные из сумм аналогичных частей слагаемых, и возвращает обычную объектную ссылку на него.

Тогда в клиентском коде можно использовать чрезвычайно удобную и наглядную «операционную форму» сложения комплексных чисел:

```
Z3 = z1 + z2;
```

Метод-операция `operator+()` принимает на вход объекты `z1` и `z2`, строит временный объект-сумму и возвращает ссылку на него, которая и запоминается в объектной ссылке `z3`. Главное, что при этом `z3` и выглядит, и на самом деле является суммой комплексных чисел `z1` и `z2`.

Можно сказать, что наш абстрактный тип `MyComplex` начал функционировать почти что как тип встроенный – предельно ясно и наглядно до самоочевидности, крайне лаконично и предсказуемо. Он заслуживает того, чтобы привести весь код этого типа (класса) полностью и порадоваться за его клиентский код:

Листинг 2.4.

```
/*-----  File MyComplex.cs  -----*/
using System;
class MyComplex
{
  // Constructors:
  public MyComplex( ){ R = I = Length = 0; }
  public MyComplex( double r, double i )
  { R = r; I = i; Length = Math.Sqrt( R*R + I*I ); }
  public MyComplex( MyComplex z )
  { R = z.R; I = z.I; Length = z.Length; }

  // Methods:
  public void SetR( double x )
  { R = x; Length = Math.Sqrt( R*R + I*I ); }
  public void SetI( double y )
  { I = y; Length = Math.Sqrt( R*R + I*I ); }
  public void Print( )
  { Console.WriteLine( "Complex number = {0}+{1}i", R, I ); }

  // Overloaded operator "+":
  static public MyComplex operator+(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R + rOp.R, lOp.I + rOp.I ); }
```

```
  // Data fields:
  private double R;
  private double I;
  public double Length;
}

/*-----  File main.cs  -----*/
using System;
class main
{
  static public int Main( )
  {
    MyComplex z1 = new MyComplex( 1, 2 );
    MyComplex z2 = new MyComplex( 3, 4 );
    MyComplex z3 = null;

    z3 = z1 + z2;   /*1*/
    z3.Print();
    Console.WriteLine( "Length = {0}", z3.Length );

    Console.ReadLine();
    return 0;
  }
}
```

В данном клиентском коде, то есть в теле функции `Main()`, мы создаем два комплексных числа `z1` и `z2`, а также пустую объектную ссылку `z3` типа `MyComplex`, которая и примет на себя сумму чисел `z1+z2` в результате неявного вызова метода `operator+()` в строке `/*1*/` из Листинга 2.4.

Код Листинга 2.4 безошибочно (в рамках проекта типа Empty Project) компилируется, запускается на выполнение и выдает в консольное окно ту же картинку, что показана выше на рис. 2.1.

Чтобы придать нашему классу `MyComplex` полный набор арифметических свойств комплексных чисел, требуется помимо операции «+» переопределить в этом классе еще и операции «-» и «*» (деление для комплексных чисел не предусмотрено) [4], а также определить статический метод `Abs()`, который призван возвращать абсолютное значение (модуль) комплексного числа. Пусть модуль комплексного числа и совпадает с длиной `Length` вектора геометрической интерпретации этого числа, но для наглядности указанный метод в проектируемом нами типе лучше все-таки определить:

Листинг 2.5.

```
/*-----  File MyComplex.cs  -----*/
using System;
class MyComplex
{
  // Constructors:
  public MyComplex( ){ R = I = Length = 0; }
  public MyComplex( double r, double i )
  { R = r; I = i; Length = Math.Sqrt( R*R + I*I ); }
  public MyComplex( MyComplex z )
  { R = z.R; I = z.I; Length = z.Length; }

  // Methods:
  public void SetR( double x )
  { R = x; Length = Math.Sqrt( R*R + I*I ); }
  public void SetI( double y )
  { I = y; Length = Math.Sqrt( R*R + I*I ); }
  public void Print( )
  { Console.WriteLine( "Complex number = {0}+{1}i", R, I ); }
  static public double Abs( MyComplex z )
  { return z.Length; }

  // Overloaded operators:
  static public MyComplex operator+(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R + rOp.R, lOp.I + rOp.I ); }
  static public MyComplex operator-(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R - rOp.R, lOp.I - rOp.I ); }
  static public MyComplex operator*(MyComplex lOp,MyComplex rOp)
  {
    double Re = lOp.R * rOp.R - lOp.I * rOp.I;
    double Im = lOp.R * rOp.I + lOp.I * rOp.R;
    return new MyComplex( Re, Im );
  }

  // Data fields:
  private double R;
  private double I;
  public double Length;
}

/*-----  File main.cs  -----*/
using System;
class main
{
  static public int Main( )
  {
```

```
        MyComplex z1 = new MyComplex( 1, 2 );
        MyComplex z2 = new MyComplex( 3, 4 );
        MyComplex z3 = null;

        z3 = z1 * z2;
        z3.Print();
        Console.WriteLine( "Abs value = {0}", MyComplex.Abs(z3) );

        Console.ReadLine();
        return 0;
    }
}
```

Компилируем Листинг 2.5, запускаем загрузочный exe-файл на выполнение и по картинке из его консольного окна (см. рис. 2.4) убеждаемся в «математической профпригодности» последнего варианта нашего типа `MyComplex`.

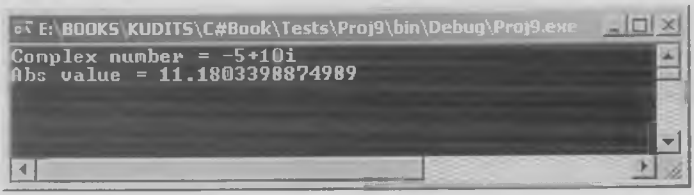


Рис. 2.4.

Легко видеть, что произведение комплексных чисел `(1+2i)(3+4i)` действительно равно `(-5+10i)`.

Итак, разработанный нами абстрактный тип `MyComplex` уже весьма хорош. Но, как говорится, нет предела совершенству. Например, не очень-то приятно для вывода значения комплексного числа применять специальный классовый метод `Print()`:

```
z3.Print();
```

вместо того, чтобы стандартным образом пользоваться методом WriteLine() библиотечного класса `Console`. Так что у нас еще остаются нереализованными некоторые планы по улучшению типа `MyComplex`, и мы с этим классом еще не раз столкнемся далее на страницах нашей книги.

А сейчас, под конец раздела, нам осталось сообщить, что *не все операции могут перегружаться в пользовательских классах*. Например, в языке C#, в отличие от языка C++ (см. [3]), перегрузка операций присваивания, а также перегрузка операций «`()`», «`[]`» и `new` (операция вызова функции, операция индексации и операция динамического создания классовых объектов) запрещена.

Глава 3. Массивы в языке C# и обработка числовых данных

3.1. Простейшее определение и инициализация одномерных числовых массивов, доступ к элементам

Часто в программах недостаточно иметь возможность работы с одиночными переменными целого и дробного типов данных, или с одиночными объектами пользовательских абстрактных типов. Встречаются ситуации, когда нужно работать с наборами однотипных данных, в совокупности составляющих единое целое. Например, речь может идти о совокупности показаний научных приборов в ходе некоторого эксперимента, или о наборе сведений о курсах акций и так далее. Подобного рода наборам данных желательно давать единственное имя, так как, во-первых, при большом количестве данных дать им индивидуальные имена практически невозможно, а во-вторых, единственное имя четче отражает факт их логического группирования в единый набор. В языке С наборам данных соответствует понятие *массива*. Массивы определяются в языке С операторами определения с дополнительным применением *квадратных скобок*.

Фактически, мы только что привели краткую выдержку из книги [1] про массивы языка С. Как и во многих других вопросах программирования и алгоритмистики, осуществить первое знакомство с массивами целесообразнее на базе языка программирования, более простого и не такого изощренного и масштабного, как язык C#. Для начального знакомства с предметом лучше всего подходит язык С, и придумать что-то иное (но столь же подходящее) сложно. Поэтому мы твердо рассчитываем, что читатели изучили программирование на С хотя бы в минимальной степени, например, как в книге [1]. Поэтому с массивами языка С они уже знакомы.

Про массивы языка C# сразу же можно высказать два противоположных тезиса: они весьма близки к массивам языка С; они сильно отличаются от массивов языка С. Оба тезиса, как это ни покажется странным на первый взгляд, верны.

Начнем с иллюстрации первого из этих тезисов (о близости массивов), и воспользуемся простой учебной программой на языке С из Листинга 3.9 книги [1], в которой создается одномерный массив целых чисел, после чего находится максимальный элемент этого массива.

Вот практически 100%-ый аналог указанной программы, но уже на языке C#:

Листинг 3.1.

```
class main
{
 public static int Main()
  {
    int size=73, i, Max;
    int[] Arr = {1,6,3,2,5,1,77,-18,99,16,45,12,-45,54,
                 567,-123,5,9,-4,67,-17,44,2,3,9,5,34,
                 11,-11,234,67,82,91,3,7,15,-32,-56,-77,
                 678, -987,456,-123,555,3,78,1,90,-4,37,
                 2,33,6,543,-88,37,66,-66,777,12,-45,567,
                 999,888,3,-765,34,89,-37,41,333,56,-987};

    //size = sizeof( Arr )/ sizeof( int );

    Max = Arr[0];
    for( i = 1; i < size; i++ )
    {
      if( Arr[i] > Max )
        Max = Arr[i];
    }
    return 0;
  }
}
```

Эта программа компилируется, запускается на выполнение и находит тот же самый максимальный элемент, равный 999, что и исходная программа на языке C. По сравнению с C-оригиналом в нее внесено три мелких изменения, причем одно из этих изменений является прямым следствием другого. Из-за того, что операция `sizeof` в языке C# к массивам неприменима в том же смысле, что и в языке C (и, вообще, эта операция может применяться только в unsafe методах – см. выше разд. 1.6), нам пришлось не вычислять значение переменной size, как ранее, а задать его явным образом. Другим изменением является положение квадратных скобок в операторе определения массива – не после имени массива, а после ключевого слова int, означающего тип элементов массива.

Несмотря на наличие мелких изменений, тезис о близости массивов языков C и C# (хотя бы внешней близости кода) Листингом 3.1 подтверждается. В книге [1] также имеется Листинг 3.10, в котором массив передается в качестве параметра функции `ArrAver()` для вычисления среднего арифметического значения его элементов. Мы здесь можем и Листинг 3.10 перевести на язык C# практически «один в один»:

Листинг 3.2.

```
class main
{
  static public int Main( )
  {
    int size = 73; double Aver;
    int[] Arr = {1,6,3,2,5,1,77,-18,99,16,45,12,-45,54,
                 567,-123,5,9,-4,67,-17,44,2,3,9,5,34,
                 11,-11,234,67,82,91,3,7,15,-32,-56,-77,
                 678, -987,456,-123,555,3,78,1,90,-4,37,
                 2,33,6,543,-88,37,66,-66,777,12,-45,567,
                 999,888,3,-765,34,89,-37,41,333,56,-987};

    //size = sizeof( Arr )/ sizeof( int );

    /*------------------------*/
    Aver = ArrAver( Arr, size );
    /*------------------------*/

    return 0;
  }

  static public double ArrAver( int[] ar, int size )
  {
    int i; double Sum = 0;

    for( i = 0; i < size; i++ )
    {
      Sum = Sum + ar[i];
    }

    return Sum / size;
  }
}
```

Здесь также следует отметить перемещение квадратных скобок в определении формального параметра функционального метода `ArrAver()` от идентификатора параметра к ключевому слову `int`, указывающему тип элементов массива, а все остальное практически одинаковое.

Это означает, что мы многое знаем о массивах языка C#. Например, как их можно инициализировать и как осуществляется с помощью операции индексации (операция «[]») доступ к индивидуальным элементам массива. И что начальный индекс массива (индекс для доступа к самому первому элементу) равен нулю. И что при передаче массива в функцию в качестве фактического параметра указывается имя массива. Это солидный объем знаний, доставшийся нам «в наследство» от языка C.

И тем не менее, это всего лишь внешнее сходство в написании синтаксических конструкций. Дело в том, что в языке C массивы строятся поверх указателей (см. разд. 6.2 из книги [1]), а в языке C# – поверх библиотечного класса Array из пространства имен System.

Так как *массивы являются встроенным типом данных* (компилятор знает о них абсолютно все заранее, и от нас не требуется никаких дополнительных подсказок), то с ними можно работать по специальному синтаксису, жестко закрепленному в языке и не требующему явного создания объектов класса `Array`. Тем не менее, применять операцию new, пусть и в измененном виде, в общем случае нужно. Только один частный случай создания массива с одновременной инициализацией, показанный выше на примерах в Листингах 3.1 и 3.2, допускает (в виде исключения) отсутствие явной записи операции new (можно писать, а можно и не писать). В настоящем разделе мы воспользовались этим исключением для демонстрации многих совпадений в синтаксисе массивов языков C и C#. О различиях мы узнали только сейчас, причем речь идет о весьма заметных и принципиальных отличиях.

О большей части этих различий мы расскажем в следующем разделе, а сейчас срочно «заделаем брешь» в автоматическом вычислении размера созданного массива (нам выше приходилось этот размер указывать явным образом). Для этого нужно воспользоваться классовым полем `Length`, достающимся массивам языка C# от класса Array. Наличие такого поля позволяет не только отказаться от ручного задания размера массива, но и *упростить механизм передачи массивов в функции* – теперь ведь не требуется одновременно передавать в функцию и размер массива. Вот улучшенный вариант программы с Листинга 3.2:

Листинг 3.3

```
class main
{
  static public int Main( )
  {
    double Aver;
    int[] Arr = {1,6,3,2,5,1,77,-18,99,16,45,12,-45,54,
                 567,-123,5,9,-4,67,-17,44,2,3,9,5,34,
                 11,-11,234,67,82,91,3,7,15,-32,-56,-77,
                 678, -987,456,-123,555,3,78,1,90,-4,37,
                 2,33,6,543,-88,37,66,-66,777,12,-45,567,
                 999,888,3,-765,34,89,-37,41,333,56,-987};
```

```
    /*------------------------*/
    Aver = ArrAver( Arr );
    /*------------------------*/

    return 0;
  }

  static public double ArrAver( int[] ar )
  {
    int i, size = ar.Length; double Sum = 0;

    for( i = 0; i < size; i++ )
    {
      Sum = Sum + ar[i];
    }

    return Sum / size;
  }
}
```

Функция `ArrAver()`, получив массив ar в качестве параметра, легко находит его размер с помощью выражения `ar.Length`.

3.2. Одномерные и многомерные массивы языка C#

Как мы выяснили в предыдущем разделе, в языке C# *для создания массивов* в общем случае *требуется применять операцию new*, динамически выделяющую память под заданное число элементов массива (этот кусок памяти и является фактически объектом типа `Array`, о котором говорилось выше в предыдущем разделе):

```
int[] ar = null;
ar = new int[6];
```

В первой строке определяется лишь объектная ссылка, инициализируемая типовым значением `null` (ссылка «смотрит в никуда»). Во второй строке выделяется память под массив, достаточная для хранения 6 элементов типа `int`. При этом все 6 элементов *автоматически* (*неявно*) *инициализируются нулями*.

Показанные выше строки кода можно записать короче в виде одной строки:

```
int[] ar = new int[6];
```

Для элементов массива можно указать инициализирующие значения явно:

```
int[] ar = new int[6] {1,2,3,4,5,6};
```

и в таком случае в квадратных скобках можно опустить число элементов массива:

```
int[] ar = new int[] {1,2,3,4,5,6};
```

В языке C# очень легко создавать *многомерные массивы*, из которых на практике чаще всего используются *двумерные прямоугольные массивы*:

```
double[,] twodAr = new double[2,3];
```

В отличие от языков C/C++, где доступ к отдельным элементам двумерных массивов осуществляется двумя последовательно применяемыми операциями индексации (см. [2]), в языке C# он выполняется лаконичнее – *одиночной индексацией* (в единственных квадратных скобках через запятую перечисляются индексы по первому и по второму измерениям):

```
for( int i = 0; i < twodAr.GetLength( 0 ); ++i )
 for( int j = 0; j < twodAr.GetLength( 1 ); ++j )
  twodAr[i,j] = i + j;
```

Метод `GetLength()`, достающийся массивам языка C# от класса `Array`, принимает на вход номер измерения (отсчитываются с нуля). Таким образом, вызов `twodAr.GetLength(0)` возвращает размер массива по первому измерению, а вызов `twodAr.GetLength(1)` – размер по второму измерению. Заодно сообщим, что выражение

```
twodAr.Rank
```

вырабатывает количество измерений многомерного массива, то есть 2 в данном конкретном случае. Значение выражения `twodAr.Length` равно общему количеству элементов в двумерном массиве, то есть 6 в данном случае.

Чтобы закрепить только что представленную информацию о массивах языка C#, построим демонстрационную программу по вычислению дисперсии элементов двумерного массива, формируемых случайным образом (с помощью объекта библиотечного класса Random). Все сведения по вычислению дисперсии приведены в книге [1], из которой мы за основу нашей программы берем Листинг 3.11:

Листинг 3.4.

```
using System;

class main
{
  /*-----------------------------------------------*/
  static public int Main( )
  {
    // Объект для генерации случайных чисел:
    Random rand = new Random();
```

```
    // Создание двумерного массива:
    double[,] twodAr = new double[2,3];

    // Формирование элементов массива
    // случайным образом:
    for( int i = 0; i < twodAr.GetLength( 0 ); ++i )
     for( int j = 0; j < twodAr.GetLength( 1 ); ++j )
      twodAr[i,j] = rand.NextDouble();

    // Вызов метода для вычисления дисперсии:
    double Disp = ArrDisp( twodAr );

    Console.WriteLine( "Disp = {0}", Disp );
    Console.ReadLine();
    return 0;
  }
  /*-----------------------------------------------*/
  static double ArrAver( double[,] ar )
  {
    double Sum = 0;

    for( int i = 0; i < ar.GetLength( 0 ); ++i )
     for( int j = 0; j < ar.GetLength( 1 ); ++j )
      Sum = Sum + ar[i,j];

    return Sum / ( ar.Length );
  }
  /*-----------------------------------------------*/
  static double ArrDisp( double[,] ar )
  {
    double Aver, dSum = 0;
    Aver = ArrAver( ar );

    for( int i = 0; i < ar.GetLength( 0 ); ++i )
     for( int j = 0; j < ar.GetLength( 1 ); ++j )
      dSum = dSum + ( ar[i,j] - Aver ) * ( ar[i,j] - Aver );

    return Math.Sqrt( dSum / ( ar.Length ) );
  }
  /*-----------------------------------------------*/
}
```

Метод `NextDouble()` библиотечного класса `Random` генерирует случайные числа от 0 до 1. Поэтому результат работы программы с Листинга 3.4 также является случайным и изменяется от запуска к запуску. На рис. 3.1 показано значение дисперсии элементов случайного двумерного массива вещественных

чисел, вычисленное и выведенное в консольное окно в некотором конкретном сеансе работы этой программы.

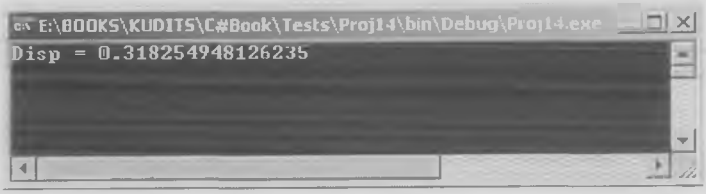


Рис. 3.1.

В работе с массивами легко допустить ошибку, связанную с неправильным значением индекса элемента. Например, типичен следующий вариант выхода за границу одномерного массива:

```
int[] ar = new int[6];
for( int i = 0; i <= ar.Length; ++i) ar[i]++;
```

когда в операторе цикла вместо знака сравнения «<» применяется знак «<=». В результате в последней итерации цикла происходит чтение памяти за границей массива. В языке C подобного рода ошибки чреваты неприятными последствиями, когда не только программа работает неправильно, но и конкретный характер последствий подобного рода ошибки абсолютно непредсказуем, очень труден для распознавания во время отладки. В книге [1] мы даже назвали выход за границу массивов «ахиллесовой пятой» этого типа данных языка C.

Спешим обрадовать читателей, что в языке C# эта проблема в самой своей неприятной ипостаси, когда во время работы программа демонстрирует непрогнозируемое поведение, устранена. На платформе Microsoft NET Framework приложения выполняются под управлением среды CLR, которая при попытке выхода за границу массива легко идентифицирует ситуацию из-за того, что массивы языка C# сообщают ей все о своем точном устройстве и размере (как мы знаем, массивы C# это объекты класса Array, а не «голые указатели» языка C). В результате среда CLR генерирует так называемое *исключение* (*exception*) и выдает в диалоговое окно диагностическую информацию (см. рис. 3.2).

В принципе, по нажатию кнопки Continue, можно продолжить работу приложения. Но лучше нажать кнопку Break, прекратить штатное исполнение приложения и перейти в режим отладки (debug), обратив внимание на диагностическое сообщение System.IndexOutOfRangeException, означающим выход за допустимый диапазон индексов.

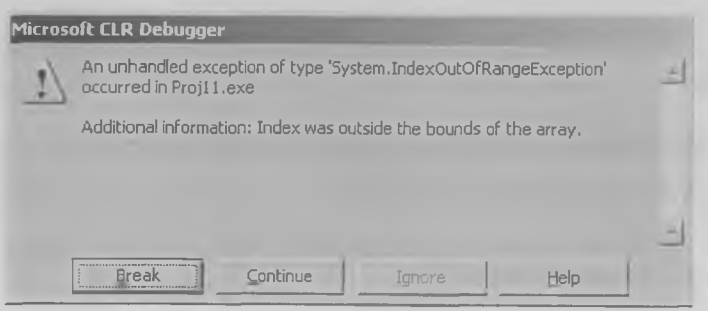


Рис. 3.2.

Самый главное достижение здесь состоит в том, что приложению не дают натворить чего-нибудь страшного. И еще, конечно, чрезвычайно полезна мгновенная диагностика для исправления недочетов в программе.

В языке C# существует возможность заранее программировать ответы на потенциально возможные исключительные ситуации – это так называемые обработчики исключений разных типов, сопоставленные проверяемому блоку кода (try-блоку). В нашей книге для начинающих мы не будем изучать автоматическую обработку исключений, отослав читателя к справочнику по языку C#, библиотеке FCL и программированию различных типов NET-приложений [5].

Ну и, наконец, в языке C# имеется *специальный вариант операторов цикла* (помимо циклов while и for), применяющий ключевые слова *foreach* и *in*, идеально подходящий для последовательного доступа ко всем элементам массива «по чтению» (то есть нельзя изменять значения элементов):

```
int[] ar = new int[6] {1,2,3,4,5,6};
foreach( int el in ar)
 Console.WriteLine( "Element in ar = {0}", el );
```

Этот лаконичный фрагмент кода последовательно выводит в консольное окно значения всех элементов массива ar (см. рис. 3.3).

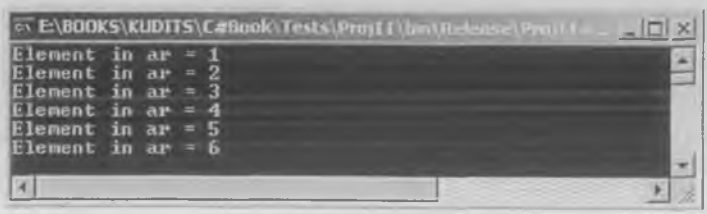


Рис. 3.3.

Можно сказать, что синтаксис циклов `foreach` говорит сам за себя. В данном случае здесь сказано буквально следующее: *для каждого* (*foreach*) элемента `el` из массива `ar`, имеющего тип int, выводить его значение в консольное окно методом `WriteLine()`. Добавить нечего.

Данный вариант цикла удобен и лаконичен, к тому же напрочь лишен возможности неправильного использования индексов (так как они вообще здесь не используются).

3.3. Массивы классовых объектов

Из материала гл. 2 мы знаем, что в языке C# с классовыми объектами работа ведется через так называемые объектные ссылки. Отсюда с неизбежностью следует, что создание массива следующего вида:

```
MyComplex[] mcAr = new MyComplex[4];
```

где `MyComplex` – это класс «комплексных чисел» из Листинга 2.5, означает лишь создание массива с элементами, каждый из которых является инициализированной значением `null` ссылкой на объекты типа `MyComplex`.

В сказанном легко убедиться на практике, для чего нужно в Debug-версии программы поставить в графической среде компилятора Microsoft Visual C# на данной строке кода клавишей F9 точку останова, выполнить эту строку (клавиша F10), и при положении текстового курса на идентификаторе mcAr клавиатурной комбинацией Shift-F9 вызвать диалоговое окно QuickWatch (см. рис. 3.4).

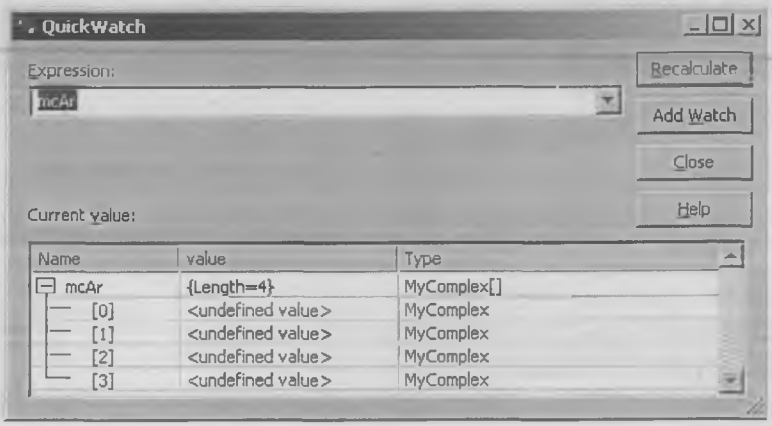


Рис. 3.4.

Из рис. 3.4 видно, что ссылка null в данном окне трактуется как undefined value (это действительно неопределенная ссылка, или как мы ее образно называем «глядящей в никуда» ссылкой).

Часть из этих ссылок можно сделать действительными, если операцией new выделить по ним из управляемой кучи среды CLR инициализированную (конструкторами) память под безымянные объекты класса MyComplex:

```
mcAr[0] = new MyComplex( 1, 2 );
mcAr[1] = new MyComplex( 3, 4 );
```

Теперь уже объектные ссылки с индексами 0 и 1 из массива mcAr представляют собой адреса реально созданных в памяти компьютера безымянных объектов класса MyComplex. В этом можно убедиться на практике с помощью диалогового окна QuickWatch, вызванного для идентификатора mcAr после исполнения двух представленных только что строк кода (см. рис. 3.5).

Рис. 3.5.

На рис. 3.5 надпись {MyComplex} символизирует действительное значение ссылки на объект типа MyComplex. Если щелкнуть левой клавишей мыши по значку плюс рядом с элементом массива, то лаконичная картинка развернется, и нам представится возможность лицезреть значения полей I, Length и R классового объекта типа MyComplex (см. рис. 3.6).

На рис. 3.6 видны значения полей R и I для элемента mcAr[0] массива mcAr, установленные конструктором с двумя параметрами (см. выше Листинг 2.5) во время создания элемента типа MyComplex операцией new.

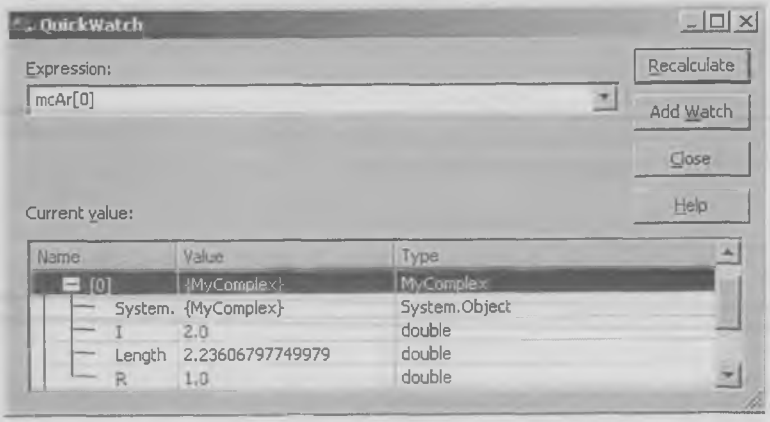


Рис. 3.6.

Чтобы закрепить полученные знания, приведем полный текст программы, в которой создается массив «комплексных чисел» (элементов типа `MyComplex`), после чего этот массив передается в функцию AverLen(), вычисляющую среднее арифметическое абсолютных значений комплексных чисел:

Листинг 3.5.

```
class main
{
  static public int Main( )
  {
    MyComplex[] mcAr = new MyComplex[4];

    mcAr[0] = new MyComplex( 1, 2 );
    mcAr[1] = new MyComplex( 3, 4 );
    mcAr[2] = new MyComplex( 5, 6 );
    mcAr[3] = new MyComplex( 7, 8 );

    /*-----------------------------*/
    double averLen = AverLen( mcAr );
    /*-----------------------------*/

    return 0;
  }
  /*-----------------------------------------*/
  static public double AverLen( MyComplex[] mcAr )
```

```
  {
    int i, size = mcAr.Length;
    double Sum = 0;

    for( i = 0; i < size; i++ )
    {
      Sum = Sum + mcAr[i].Length;
    }

    return Sum / size;
  }
}
```

Помимо Листинга 3.5, для большей наглядности приведем пару рисунков. Первый из них демонстрирует схематическое распределение памяти под массивы типа `int` и типа `MyComplex`, созданные операторами без одновременной инициализации элементов (см. рис. 3.7).

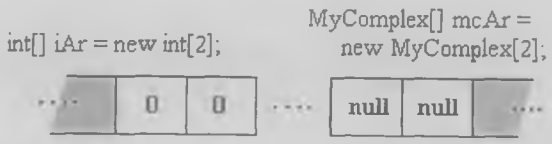


Рис. 3.7.

На втором из рисунков показано схематическое распределение памяти в случае применения инициализации (см. рис. 3.8).

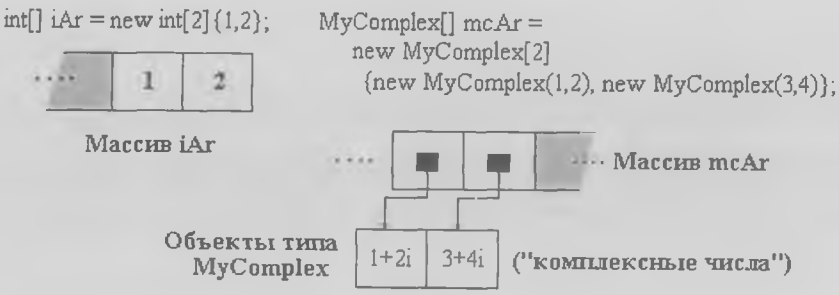


Рис. 3.8.

На рис. 3.8 помимо схемы взаимосвязи участков памяти, выделенных под массив и под «комплексные числа» (объекты класса `MyComplex`), стоит еще

обратить внимание на краткую форму записи оператора, создающего массив классовых объектов с одновременной инициализацией элементов.

3.4. Массивы в качестве полей данных классов языка C#

До сих пор мы создавали массивы в качестве переменных, локальных по отношению к классовым методам (то есть внутри тел функциональных методов). Применим теперь массивы в качестве полей данных классов языка C#.

Вот начальный пример учебного класса на эту тему:

Листинг 3.6.

```
using System;
class Test
{
  // Constructor:
  public Test( int Num ){ data = new int[Num]; }

  // Other methods:
  public int GetElem( int i ){ return data[i]; }
  public void SetElem( int i, int val ){ data[i] = val; }
  public int[] GetInnerAr( ){ return data; }

  // Data field:
  private int[] data;
}

class main
{
  static public int Main( )
  {
    Test obj = new Test(3);

    // Поэлементный доступ к внутреннему массиву:
    for( int i = 0; i < obj.GetInnerAr().Length; ++i )
     obj.SetElem( i, i*i );                          /*1*/

    // Доступ к самому внутреннему массиву "для чтения":
    foreach( int el in obj.GetInnerAr() )            /*2*/
     Console.WriteLine( "Elem = {0}", el );          /*3*/

    Console.ReadLine();
```

```
    return 0;
  }
}
```

Учебный класс `Test` имеет в своем составе закрытое поле, имеющее тип массива целых значений. Конструктор с одним параметром инициализирует это поле под заданное число элементов. С помощью открытого метода `SetElem()` устанавливаются значения индивидуальных элементов внутреннего массива.

Открытый метод `GetInnerAr()` возвращает ссылку на закрытое поле данных `data` (то есть на внутренний массив целых чисел). Этот метод удобно использовать для определения размера внутреннего массива, что мы и делаем в цикле, в котором прописываются значения индивидуальных элементов массива.

На рис. 3.9 показан результат работы этой программы, из которого видно, что элементы действительно проинициализированы квадратами их индексов (строка `/*1*/` из Листинга 3.6).

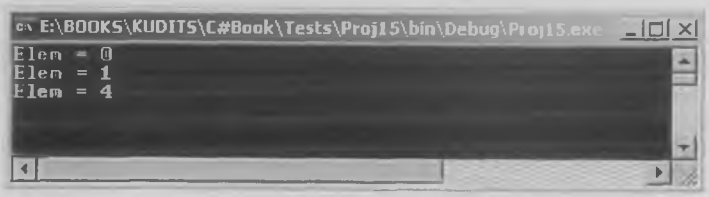


Рис. 3.9.

Непосредственно вывод в консольное окно осуществлен в Листинге 3.6 с помощью цикла foreach (строки `/*2*/-/*3*/`), автоматически последовательно перебирающим все элементы массива, предоставленные в его распоряжение методом `GetInnerAr()` учебного класса `Test`.

Сейчас самое время вспомнить (если кто забыл) о том, что любой массив в языке C# сам по себе неявным образом представляет собой объект класса Array, а значит адресуется объектной ссылкой. С точки зрения классов языка C# с подобного рода особенностью их устройства, когда поля данных являются массивами, мы сталкиваемся первый раз, и это имеет определенные последствия для программирования копирующих конструкторов (про копирующие конструкторы см. разд. 2.3).

Дело в том, что в случае полей данных, являющихся массивами, тривиальное копирование в теле копирующего конструктора всех классовых полей создает проблему, которую мы проиллюстрируем на следущем примере: добавим

в наш текущий учебный класс Test из Листинга 3.6 тривиальный копирующий конструктор:

```
public Test( Test ob ){ data = ob.data; }
```

а в коде клиента вместо одного, как ранее, создадим два объекта:

```
Test obj = new Test(3); Test obj2 = new Test(obj);
```

Второй объект создается из первого объекта тривиальным копирующим конструктором, и в итоге мы получаем два зависимых друг от друга объекта типа Test, что графически иллюстрируется с помощью рис. 3.10.

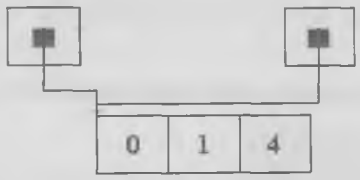


Рис. 3.10.

Зависимость объектов obj и obj2 заключается в том, что их поля данных data, являющиеся ссылками на массив целых чисел, глядят на один и тот же массив в памяти. Если в конце клиентского кода из Листинга 3.6, в котором прописываются элементы внутреннего массива объекта obj, выводить в консольное окно значения элементов внутреннего массива объекта obj2, то мы получим тот же самый результата, что и на рис. 3.9.

В общем, совершенно понятно, что в данном случае код копирующего конструктора класса Test нужно менять, отказываясь от схемы *тривиального копирования полей данных* (*shallow copy*). Вот программное решение, которое позволяет создавать при помощи нового варианта копирующего конструктора независимые друг от друга объекты:

```
public Test( Test ob )
{
  int len = ob.GetInnerAr().Length;
  data = new int[len];

  for( int i = 0; i < len; ++i )
   data[i] = ob.GetElem( i );
}
```

Закончим данный раздел необычно (для программистов на языке C++) выглядящим кодом, инициализирующим классовое поле данных типа статического массива:

```
class Test2
{
  // Constructor:
  public Test( int Num ){ data = new int[Num]; }

  // Other methods:
  public int GetElem( int i ){ return data[i]; }
  public void SetElem( int i, int val ){ data[i] = val; }
  public int[] GetInnerAr( ){ return data; }

  // Data field:
  private int[] data;
  // Static array:
  static public int ARR[] = new int[] {1,2,3,4,5};
}
```

Про синтаксис инициализации классовых полей данных с помощью операции «=» мы уже говорили выше (см. разд. 2.3) и решили его в нашем пособии не применять. Для поля данных, имеющего тип массива, этот синтаксис имеет еще более экзотический вид. И, тем не менее, конкретно для статических массивов он на практике используется довольно часто.

3.5. Сортировка массивов и поиск в массивах

Сравнивая массивы языков C и C#, можно сказать, что массивы языка C# устроены сложнее, откуда с неизбежностью вытекает, что их несколько сложнее изучать. Что мы получаем взамен? Уже известно (см. выше разд. 3.2), что становится более надежным исполняемый код, так как среда CLR контролирует его выполнение и следит за выходом за границы массивов. Это очень важное преимущество, и более всего оно будет понятно опытным программистам на C/C++, участвовавшим в крупных коммерческих разработках, и которые почти наверняка сильно обжигались на «простоте массивов языков C/C++».

А что могут получить в качестве премии за «сложность массивов языка C#» начинающие программисты? Оказывается, что очень много: ведь массивы языка C# суть объекты библиотечного класса `Array`, а последний обладает набором чрезвычайно полезных методов, самостоятельное программирование которых «вылетает в копеечку». Например, различные алгоритмы сортировки массивов обычно рассматриваются в специальных математических курсах. А читатели

книги [1] по собственному опыту знают, что в языке C даже сортировка массивов «методом пузырька», при всей своей принципиальной несложности, требует и внимания, и немалого объема работы (см. Листинг 3.18 из книги [1]). Так вот, в языке C# все это достается почти даром (только язык изучить нужно).

Например, в библиотечном классе `Array` определен статический метод `Sort()`, выполняющий эффективный алгоритм сортировки массивов:

Листинг 3.7.

```
using System;
class main
{
  static public int Main( )
  {
    Random rand = new Random();
    double[] Ar = new double[50000];

    for( int i = 0; i < Ar.Length; ++i )
     Ar[i] = rand.NextDouble();

    /*--------------*/
    Array.Sort( Ar );
    /*--------------*/

    foreach( double el in Ar )
     Console.WriteLine( "El = {0}", el );

    Console.ReadLine();
    return 0;
  }
}
```

Вызов сортировочного метода от лица имени класса (это ведь статический метод) столь прост, что даже и комментировать нечего. Единственно, на что стоит обратить внимание в Листинге 3.6, так это число `50000` для элементов массива типа `double`. Это весьма немалое число, так что для случая собственных реализаций алгоритмов сортировки (как мы это делали в книге [1]) не стоит сразу же устанавливать такую высокую планку. Здесь же все просто – библиотечный алгоритм реализован высокоэффективно, и надежно. Так что внедряем код Листинга 3.6 в проект типа Empty Project, компилируем и запускаем на выполнение. Даже на далеко не самых высокоскоростных компьютерах вся работа много времени не займет. Ее результат, то есть элементы массива `Ar`, расположенные в отсортированном порядке, показан на рис. 3.11.

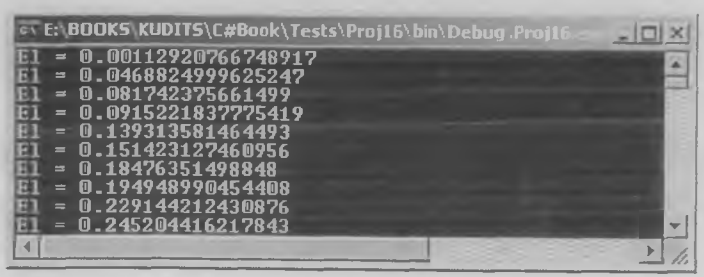


Рис. 3.11.

Из рис. 3.11 видно, что по умолчанию метод `Sort()` выполняет *сортировку элементов массива по возрастанию*. Чтобы развернуть направление сортировки и получить массив, отсортированный по убыванию, требуется привлечь синтаксические конструкции языка C#, которые мы пока что не изучали. Более сложным элементам языка C#, позволяющим строить мощнейшие обобщенные решения (например, метод `Sort()` работает не только с массивами типов `int`, `double`, но и с массивами пользовательских типов данных, а также сортировку по многочисленным пользовательским критериям).

Не менее проблемной задачей, чем сортировка больших массивов, является задача о поиске заданного значения в большом массиве данных. Эта задача также решена в рамках библиотечного класса `Array`, и представлена она статическим методом с «говорящим именем» `BinarySearch()`:

Листинг 3.8.

```
using System;
class main
{
  static public int Main( )
  {
    Random rand = new Random();
    int[] Ar = new int[50000];

    for( int i = 0; i < Ar.Length; ++i )
     Ar[i] = rand.Next( 0, 10000 );

    Array.Sort( Ar );
    int ind = Array.BinarySearch( Ar, 5 );
```

```
    if( ind >= 0 ) // success
     Console.WriteLine( "El = {0} for index = {1}",
                        Ar[ind], ind );
    else           // failure
     Console.WriteLine( "No any 5s in the array" );

    Console.ReadLine();
    return 0;
  }
}
```

Как видно из Листинга 3.8, в программах на языке C# применять функциональный метод `BinarySearch()` библиотечного класса `Array` (определен в пространстве имен `System`) намного проще, чем аналогичную по назначению функцию `bsearch()` из стандартной библиотеки языка C (см. разд. 9.2 книги [1]). В качестве параметров методу `BinarySearch()` передаются массив и искомое значение, а возвращает этот метод индекс первого найденного элемента, или отрицательное значение в случае неуспеха поиска. Нужно, правда, не забыть предварительно отсортировать массив методом `Sort()`.

Так как целочисленные элементы массива `Ar`, в котором ведется поиск, формируются случайным образом (их значения формируются параметрами вызова метода `Next()` и лежат в в данном случае в диапазоне от `0` до `10000`), то результаты работы программы с Листинга 3.8 в разных сеансах разные. Результат одного из сеансов, когда удается найти искомый элемент массива (со значением 5), показан на рис. 3.12.

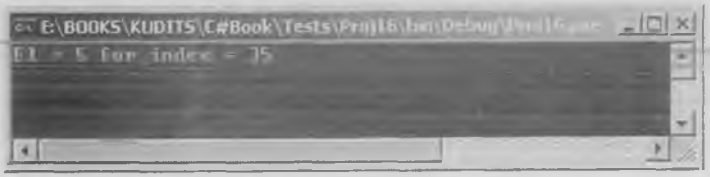


Рис. 3.12.

Из-за случайного характера формирования элементов массива `Ar` время от времени (крайне редко) возникают ситуации, когда найти элемент со значением 5 не удается (такой элемент в массиве `Ar` отсутствует), и в итоге консольное окно программы с Листинга 3.8 выглядит так, как показано на рис. 3.13.

В классе `Array` предусмотрены не только такие «тяжеловесные методы», как `Sort()` и `BinarySearch()`, эффективная реализация которых требует, вообще говоря, фундаментальных знаний по теории алгоритмов обработки данных,

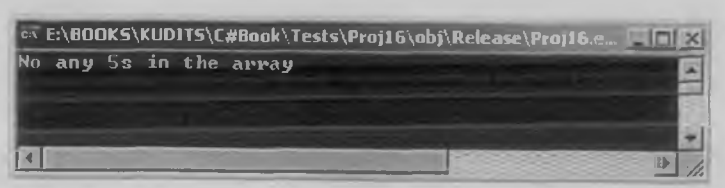


Рис. 3.13.

но и совсем несложные, но практически полезные операции, такие как «реверсирование элементов массива».

В книге [1] мы эту задачу решали самостоятельно (см. Листинг 3.5). А здесь с удовольствием воспользуемся готовыми услугами:

Листинг 3.9.

```
using System;
class main
{
  static public int Main( )
  {
    Random rand = new Random();
    double[] Ar = new double[5];

    for( int i = 0; i < Ar.Length; ++i )
     Ar[i] = rand.NextDouble();

    foreach( double el in Ar )
     Console.WriteLine( "Before reverse: El = \t{0}", el );

    /*-----------------*/
    Array.Reverse( Ar );
    /*-----------------*/

    Console.WriteLine();
    foreach( double el in Ar )
     Console.WriteLine( "After reverse: El = \t{0}", el );

    Console.ReadLine();
    return 0;
  }
}
```

Комментировать вызов метода `Reverse()` не имеет смысла – все ясно с первого взгляда. На рис. 3.14 показано консольное окно этой программы в одном из сеансов ее работы.

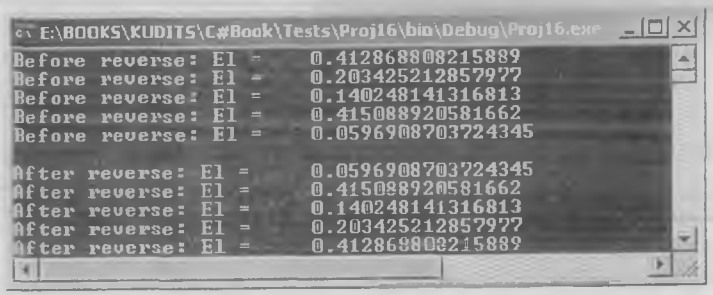


Рис. 3.14.

Из рис. 3.14 прекрасно видно, что метод `Reverse()` действительно переставляет элементы массива так, чтобы они шли в обратном (первоначальному) порядке.

Подводя окончательный итог главы, резюмируем, что массивы языка C# являются существенно более мощным и «интеллектуальным» средством программирования, чем простые и незатейливые (только что изучать проще) массивы языка C.

Глава 4. Строки языка C# и алгоритмы обработки текстов

4.1. Тип данных char и символьные константы в UNICODE-кодировке

Как мы знаем, для представления одиночных символов естественных языков в языке C предназначен тип данных char (см. разд. 4.1 из книги [1]). В языке C# тоже есть тип char и применяется он с той же целью. Только тип char в языке C# кодирует символы в так называемой *кодировке UNICODE* (2 байта на один символ, вместо 1 байта по стандартным для языка C кодировкам ASCII/ANSI). Это сразу и радикально решает все проблемы с национальными языками, отличными от английского (и от западноевропейских языков). Нас в первую очередь интересует русский язык. Так вот, кодировка UNICODE позволяет нам программно работать с русским языком без проблем, и поэтому в данной главе мы для подчеркивания этого факта (и для привыкания к нему) будем использовать в качестве примера обрабатываемых символов только символы русского языка.

В языке C#, как и в языке C, *символьные константы* представляют из себя запись символа в окружении из *апострофов* (одиночных кавычек).

Знать и помнить наизусть числовые значения UNICODE-кодов для тех или иных конкретных символов не нужно, ведь для этого можно воспользоваться следующей простейшей программой, которая выведет всю необходимую нам информацию в свое консольное окно:

Листинг 4.1.

```
using System;
class main
{
  static public int Main( )
  {
    char[] chAr = {'A','\\','Ф','Ж'};
    int[]  iCode = new int[chAr.Length];

    for( int i = 0; i < chAr.Length; ++i )
    {
      iCode[i] = (int)chAr[i];
      Console.WriteLine( "Для {0} код равен {1}",
                         chAr[i], iCode[i] );
    }
```

```
    Console.ReadLine();
    return 0;
  }
}
```

Здесь в массив chAr типа char[] помещены два русских символа – 'Ф' и 'Ж', английский символ 'A', а также символ обратной наклонной черты. С последним символом в языке C# ситуация та же самая, что и в языке C (см. разд. 4.1 из книги [1]) – он выполняет роль маркера начала управляющих символов (\n – перевод строки, \t – табуляция и т. д.), так что его самого приходится изображать в виде набора из двух символов наклонной черты.

Символы из массива chAr[] легко преобразуются в обычные целочисленные значения типа int с помощью *операции явного приведения типов*:

```
iCode[i] = (int)chAr[i];
```

Метод WriteLine() выводит в консольное окно элементы массива chAr в форме символов, а элементы массива iCode[i] – в форме числовых кодов этих символов.

Запускаем программу с Листинга 4.1 на выполнение и видим нужную нам информацию по UNICODE-кодам (см. рис. 4.1).

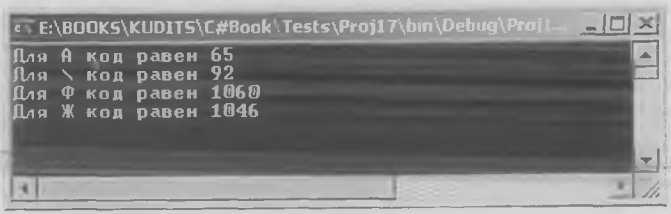


Рис. 4.1.

Из рис. 4.1 видно, что в UNICODE-кодировке английскому символу A соответствует числовой код 65, символу обратной наклонной чертой – код 92, русской букве Ф – числовой код 1060, русской букве Ж – код 1046.

Ясно, что UNOCODE-коды английских букв, цифр, знаков препинания и тому подобных символов те же самые, что и в ASCII-кодировке. Следующая программа показывает коды управляющих символов \n и \t:

Листинг 4.2.

```
using System;
class main
{
  static public int Main( )
  {
    char x = '\n', y = '\t';
    int ix = (int)x, iy = (int)y;

    Console.WriteLine( "Symbol \\n has code {0}", ix );
    Console.WriteLine( "Symbol \\t has code {0}", iy );

    Console.ReadLine();
    return 0;
  }
}
```

Компилируем программу, запускаем ее на выполнение и получаем ответ на наш вопрос (см. рис. 4.2).

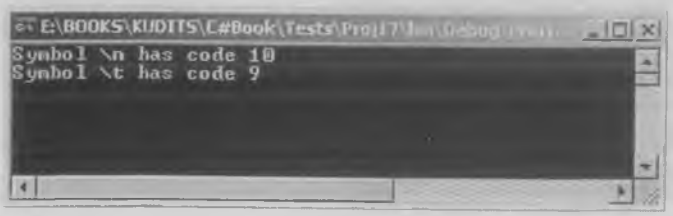


Рис. 4.2.

Рис. 4.2 показывает, что управляющему символу `\n` соответствует числовой UNICODE-код 10 (в десятичной записи), а управляющему символу табуляции `\t` соответствует числовой код 9 (то есть, все как в ASCII-кодировке – см. Листинг 4.4 и рис. 4.3 из книги [1]).

Другое дело, это коды символов некоторых национальных языков. Из рис. 4.1 видно, что русским буквам соответствует совсем иной диапазон значений UNICODE-кодов (больше 1000), для наглядной демонстрации которого предназначена следующая программа:

Листинг 4.3.

```
using System;
class main
{
```

```
  static public int Main( )
  {
    char x = (char)1040;

    Console.WriteLine( "UNICODE\t\tSymbol" );
    Console.WriteLine( "-----------------------------" );
    for( int i =0; i < 66; ++i, ++x )
    {
      Console.WriteLine( "{0}\t\t{1}", x, 1040+i );
    }

    Console.ReadLine();
    return 0;
  }
}
```

Данная программа, используя управляющие символы табуляции, порождает наглядный табличный вывод в консольное окно информации о UNICODE-кодах русских букв (см. рис. 4.3).

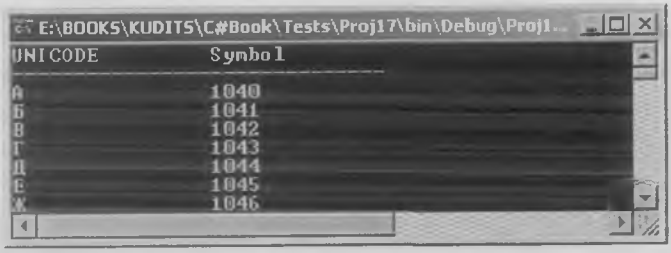


Рис. 4.3.

Заметим, что в языке C#, в отличие от языков C/C++, переменной типа char нельзя напрямую присвоить числовое значение, так как возникает ошибка компиляции. Поэтому в Листинге 4.3 мы используем операцию явного приведения типов данных:

```
char x = (char)1040;
```

То же самое присваивание можно выполнить, используя апострофы и шестнадцатеричную форму записи для десятичного числа 1040:

```
char x = '\u0410';
```

И, наконец, самое большое отличие типа char языка C# от одноименного типа данных языков C/C++ заключается в том, что этот встроенный тип данных

строится на базе библиотечного класса с именем Char (определен в пространстве имен System). Ранее мы столкнулись с подобным явлением на примере массивов языка C#, и здесь все то же самое (маленькое отличие – данный класс, то есть абстрактный тип данных Char, определяется не ключевым словом class, а другим ключевым словом – struct, что влечет за собой определенные отличия, но они в нашем пособии для простоты не рассматриваются).

Тип Char содержит набор чрезвычайно практически полезных статических методов, позволяющих запросить тип символа – буква или нет, большая буква (заглавная) или строчная, знак ли это пунктуации и так далее. Вот иллюстрирующий пример на эту тему:

Листинг 4.4.

```
using System;
class main
{
  static public int Main( )
  {
    char x1 = '\n', x2 = 'ж', x3 = '7';

    if( Char.IsWhiteSpace( x1 ) )
      Console.WriteLine( "\\n is whitespace" );
    if( Char.IsLetter( x2 ) )
      Console.WriteLine( "ж is letter" );
    if( Char.IsNumber( x3 ) )
      Console.WriteLine( "7 is number" );

    Console.ReadLine();
    return 0;
  }
}
```

Запускаем программу и видим, кого «признают за своих» статические методы IsWhiteSpace(), IsLetter() и IsNumber() (см. рис. 4.4).

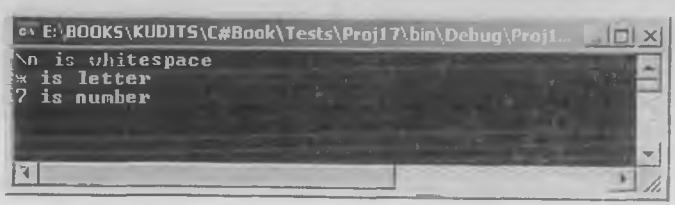


Рис. 4.4.

Итак, служебный управляющий символ \n трактуется как *пробельный* (*whitespace*), ну а символы 'ж' и '7' трактуются, естественно, как буква и цифра, соответственно. Имеется также и метод `IsLetterOrDigit()`, одинаково «приветствующий» и буквы, и цифры.

4.2. Массивы типа char и строки языка C#

Поскольку одиночные переменные типа char удобны для хранения символов (в смысле, UNICODE-кодов символов), то массивы элементов типа char пригодны для хранения наборов символов, в роли которых могут выступать слова естественного языка (иллюстрируем все для русского языка) и сочетания слов и знаков препинания:

Листинг 4.5.

```
using System;
class main
{
  static public int Main( )
  {
    char[] myT = { 'П', 'р', 'и', 'в', 'е', 'т', ',',
                   ' ', 'М', 'и', 'р', '!' };

    Console.WriteLine( "Words in char-array" );
    Console.WriteLine( "-------------------" );

    foreach( char el in myT )
     Console.Write( "{0}", el );

    Console.ReadLine();
    return 0;
  }
}
```

Данная программа, создав массив myT с элементами типа char, хранящими символы 'П', 'р', 'и', 'в', 'е', 'т', ',', ' ', 'М', 'и', 'р', '!', затем в цикле с помощью метода `Write()` библиотечного класса `Console` выводит в консольное окно фразу "Привет, Мир!", что и показано на рис. 4.5.

Использованный в Листинге 4.5 метод `Write()` библиотечного класса `Console`, в отличие от постоянно нами применяющегося метода `WriteLine()`, автоматического перевода строки не осуществляет, в результате чего буквы из фразы "Привет, Мир!" располагаются в одной и той же физической строке консольного окна (что и требовалось получить).

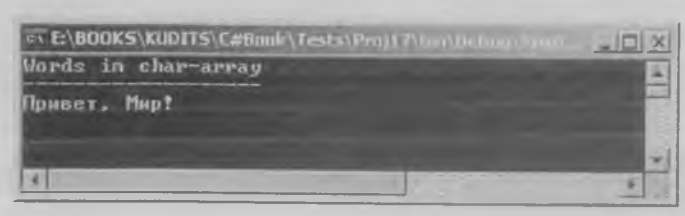


Рис. 4.5.

Хранение фраз естественного языка в массивах типа `char[]` открывает потенциально неограниченные возможности их обработки, из которых мы сейчас, не мудрствуя лукаво, выберем простейшую задачу о реверсировании букв в тексте:

Листинг 4.6.

```
using System;
class main
{
  static public int Main( )
  {
    char[] myT = { 'П', 'р', 'и', 'в', 'е', 'т', ',',
                   ' ', 'М', 'и', 'р', '!' };

    Console.WriteLine( "Words in reverse symbol order" );
    Console.WriteLine( "-----------------------------" );

    /*-----------------*/
    Array.Reverse( myT );
    /*-----------------*/

    foreach( char el in myT )
      Console.Write( "{0}", el );

    Console.ReadLine();
    return 0;
  }
}
```

Реверсирование фразы, содержащейся в массиве символов myT, в языке C# легко выполняется с помощью известного нам метода `Reverse()` библиотечного класса `Array` (см. выше Листинг 3.9), в то время как в языке C нам пришлось для этого писать собственную функция `ReversePhrase()` (см. Листинг 4.6 из книги [1]).

На рис. 4.6 показан результат трудов программы с Листинга 4.6.

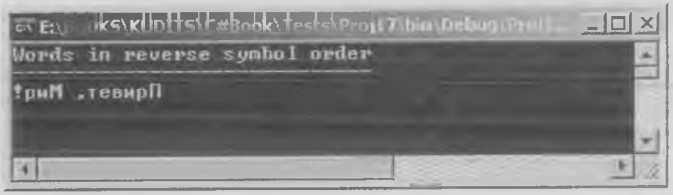


Рис. 4.6.

Как видно из рис. 4.6 метод `Reverse()` из фразы "Привет, Мир!" сделал анаграмму "!риМ ,тевирП".

В целом можно констатировать, что имитировать работу с фразами естественных языков при помощи символьных массивов (массивов элементов типа `char`) вполне возможно. Однако, как и в языке C, имеется принципиальная возможность упростить такого рода работу. Вспоминаем, что в языке C для этого вводится соглашение о терминальном нуле, после чего такие специфические массивы элементов типа `char` называются строками языка C, и еще дополнительно предоставляется богатый набор строковых библиотечных функций. Кроме того, строки языка C «подкрепляются» специальными *строковыми константами*, представляющими из себя текст, заключенный с двух сторон в двойные кавычки:

```
"Hello, world" или "Something else"
```

и так далее.

В языке C# пошли намного дальше и помимо строковых констант подкрепили строковый тип данных этого языка базированием на библиотечном классе String, определенном в пространстве имен System. Итак, *строки языка C#* – это фактически *объекты класса String*. Являясь встроенным типом языка C#, строки имеют и свой индивидуальный синтаксис, жестко закрепленный в языке. Например, вместо имени библиотечного класса String можно (и желательно) использовать *ключевое слово string* (фактически, это *синоним для имени String*).

Вот пример того, как в языке C# можно создать строку на базе существующего массива типа `char[]`:

```
char[] myT = { 'П', 'р', 'и', 'в', 'е', 'т' };
string str = new string( myT );
```

Как и в случае любых классов языка C# здесь идентификатором `str` обозначена ссылка на объект типа String, память под который выделяется операцией

new. Допустимость показанной формы создания строк обуславливается наличием в классе String конструктора, принимающего в качестве параметра массив типа `char[]`.

Обязательно стоит отметить, что в классе String *умолчательный конструктор не определен*, и поэтому в языке C# при попытке создать строку следующим образом:

```
string str = new string( ); // error
```

возникает ошибка компиляции.

Строки языка C# являются чрезвычайно важным, можно сказать инфраструктурным типом данных, и поэтому для них в синтаксисе языка предусмотрены всякого рода упрощающие исключения (это даже для строк языков C/C++ делается), являющиеся отклонением от синтаксиса создания классовых объектов в языке C#.

Самым важным таким исключением (упрощением), чаще всего применяемым на практике, является следующая форма создания строк

```
string str = "Привет";
```

использующая строковую константу, и не использующая операцию new. При этом `str` по-прежнему имеет смысл ссылки на объект типа String в памяти компьютера. Всюду далее в нашей книге мы для упрощения будем говорить о переменной `str` как о строке, так как точный смысл вам теперь должен быть уже полностью ясен. Например, вам наверняка понятно, что оператор

```
string str;
```

никакой реальной строки текста не создает, а лишь ссылку на тип `string`. И только в результате последующего оператора

```
str = "Привет";
```

(без явного использования операции new) ссылка `str` принимает и хранит адрес объекта типа String в памяти компьютера (этот объект и содержит в себе текст "Привет").

Применение строковых констант и операции «=» для строк языка C# более чем характерно:

```
string str = "Привет";
str = str + ", " + "Мир" + "!";
```

Из этого примера, формирующего для строки `str` значение "Привет, Мир!", прекрасно видно, что для класса String перегружаются операции «=» и «+», так

что работа со строками языка C# становится простой и удобной. Можно даже еще укоротить запись представленного фрагмента, если воспользоваться перегруженной операцией комбинированного (составного) присваивания «+=»:

```
string str = "Привет";
str += ", " + "Мир" + "!";
```

Но самое главное, конечно, что мы при этом вообще не думаем про размеры буферов в памяти, что в языке C является серьезнейшей проблемой в работе со строками, чреватой, помимо прочего, возникновением трудноуловимых ошибок, связанных с выходом за границу буферов и тяжелыми последствиями во время работы программы.

Уже во время изучения массивов языка C# мы узнали, что выход за границу массива не дает столь же тяжелых последствий во время работы программы, как это бывает в программах на языке C, ибо здесь классовая природа массивов обеспечивает генерацию исключений. Но классовая природа строк языка C# позволяет продвинуться еще дальше – здесь даже проблемы выхода за границы буферов не возникает, так как объекты класса String динамически управляют своими внутренними буферами, осуществляя во время работы программы их увеличение в случае необходимости.

При этом всегда можно запросить строку `str` на предмет ее длины (количества содержащихся в ней символов в UNICODE-кодировке) с помощью выражения `str.Length`. Это может пригодиться, например, в программе, где уже сформированную строку «разбирают» на отдельные символы:

Листинг 4.7.

```
using System;
class main
{
  static public int Main( )
  {
    string str1 = "Привет";
    string str2 = new string( '3', 2 );
    str1 += ", " + str2;

    int len = str1.Length;
    for( int i = 0; i < len; ++i )
    {
      char ch = str1[i];
      Console.WriteLine( "Символ на {0} позиции есть {1 }",
                         i, ch );
    }
```

```
    Console.ReadLine();
    return 0;
  }
}
```

Результат разбора строки `str1` на составляющие ее символы показан на рис. 4.7.

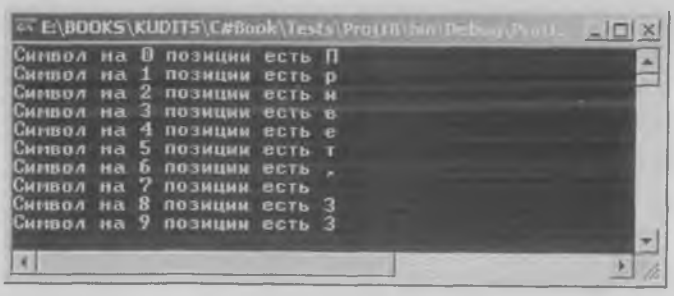


Рис. 4.7.

В Листинге 4.7 можно обратить внимание на следующие два момента. Во-первых, доступ к отдельным символам строки с помощью операции индексации (операции «[]») осуществляется в режиме «по чтению», то есть с его помощью нельзя заменять символы строки.

Во-вторых, из содержимого строки `str1`, показанного на рис. 4.7, становится понятно, как работает конструктор

```
string str2 = new string( '3', 2 );
```

Этот конструктор формирует строку, повторяя заданный символ (первый параметр) заданное число раз (второй параметр).

Ну и, наконец, строки можно формировать не только конструкторами класса `String` или на основе строковых констант, но и с помощью их ввода с клавиатуры. В принципе, мы с этим уже знакомы, так как еще в разд. 1.5 начали использовать метод `ReadLine()` библиотечного класса Console, который, как мы выяснили, возвращает в виде строки набор символов, соответствующих нажатым пользователем клавишам (за исключением клавиши Enter, сигнализирующей об окончании ввода). Но лишний наглядный пример не помешает:

Листинг 4.8.

```
using System;
class main
{
```

```
  static public int Main( )
  {
    string str1 = Console.ReadLine();    /*1*/
    string str2 = Console.ReadLine();    /*2*/
    string str3 = str1 + '\n' + str2;    /*3*/

    Console.WriteLine( str3 );
    Console.ReadLine();
    return 0;
  }
}
```

Здесь показаны три новых момента. Во-первых, это анонсированный выше ввод значений строк языка C# с клавиатуры (физические строки `/*1*/`-`/*2*/` Листинга 4.8). Во-вторых, показано, что «складывать» можно не только одни лишь строки между собой, но и строки с отдельными символами (строка `/*3*/` кода программы).

А в-третьих, в Листинге 4.8 продемонстрирован вызов метода `WriteLine()` с единственным фактическим параметром, в качестве которого использована сформированная нами ранее строка `str3`. Результат работы некоторого сеанса работы этой программы показан на рис. 4.8.

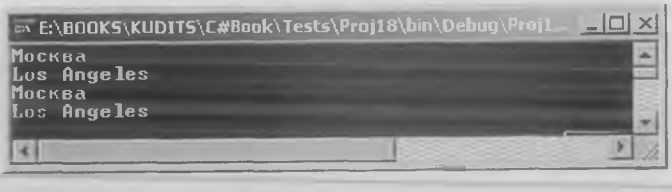


Рис. 4.8.

Как видно из рис. 4.8, в этом сеансе работы мы сначала с клавиатуры ввели строку "Москва" и нажали клавишу Enter (тем самым это строковое значение стало содержимым строкового объекта, адресуемого ссылкой `str1`), затем ввели строку "Los Angeles". После этого метод `WriteLine()` использует свой единственный параметр как «форматирующую строку», то есть выводит в консольное окно весь содержащийся в ней текст, кроме управляющего символа `\n`, который работает как сигнал, требующий выводить вторую часть строки `str3` в новую физическую строку консольного окна программы. Это мы и наблюдаем на рис. 4.8.

4.3. Обработка естественноязыковых текстов

В предыдущем разделе мы уже имели возможность убедиться в том, что со строками языка C# работать несравненно легче и приятнее (и менее опасно), чем со строками языка C. Действительно, в книге [1] целый раздел специально отводился под тренировку техники выполнения таких стандартных действий, как перезапись строк и конкатенация строк (см. разд. 4.4 из книги [1]). В языке C# отпадает необходимость в специальной отработке техники таких действий, так все это выполняется практически без усилий. Действительно, перезапись строк выполняется операцией присваивания

```
string st1 = "abc"; st1 = "новое содержимое"; st1 = "";
```

а конкатенация – операцией сложения (в том числе – комбинированной операцией):

```
string st2 = "abc"; st2 += "добавок";
```

При перезаписи строк старое их содержимое для нашей программы теряется (переходит под управление сборщика мусора). Еще стоит обратить внимание на строку "", формируемую двумя подряд записанными двойными кавычками – это так называемая *пустая строка*, длина которой равна нулю.

Конкатенацию строк часто применяют на практике в случае слишком длинной форматирующей строки метода `WriteLine()`:

```
Console.WriteLine( "Слишком длинная форматирующая"+
                   " строка" );
```

Итак, технические проблемы работы со строками в языке C# решаются, можно сказать, сами собой. Но помимо технических задач, существуют еще и практически важные задачи по обработке текстов естественных языков: проверка правописания, автоматический поиск фрагментов текста по образцу, выявление особенностей стиля того или иного писателя и так далее и тому подобное. Все это образцы работ с текстами, которые требуют разработки сложных алгоритмов, базирующихся на учете текстового синтаксиса и морфологии, а также на алгоритмах искусственного интеллекта, когда формулируются приблизительные правила, которыми в своей деятельности руководствуется человек (пусть, даже, и неосознанно). Перечисленные задачи являются весьма сложными и выполняются в общем случае коллективами профессиональных разработчиков предметных тем и программистов. Мы же сейчас рассмотрим небольшой спектр вспомогательных задач, лежащих в основе более сложных задач по обработке текстов.

Для решения технических задач обработки текстов в библиотечном классе String предусмотрено изрядное число методов. Начнем наше рассмотрение технических задач со следующих проблем: выделить из текста первые N символов, выделить из текста последние N символов, выделить из текста N символов, начиная с позиции m-го символа. Все перечисленные проблемы требуют применения метода `Substring()`:

```
str.Substring( StartIndex, length );
```

где `StartIndex` – это начальная позиция в исходной строке для выделяемой подстроки, а `length` – длина подстроки. Вот иллюстрирующий пример на эту тему:

Листинг 4.9.

```
using System;
class main
{
  static public int Main( )
  {
    string str1 = "Moscow";
    string str2 = null;

    // 1.
    str2 = str1.Substring( 0, 3 );
    Console.WriteLine( str2 + '\n' );

    // 2.
    str2 = str1.Substring( 1, 2 );
    Console.WriteLine( str2 + '\n' );

    // 3.
    str2 = str1.Substring( str1.Length - 3, 3 );
    Console.WriteLine( str2 );

    Console.ReadLine();
    return 0;
  }
}
```

В результате работы программы с Листинга 4.9 строка-приемник `str2` покажет в консольном окне вполне ожидаемое содержимое (см. рис. 4.9).

Действительно, рис. 4.9 показывает, что сначала в буферную строку `str2` были успешно скопированы первые три символа исходной строки `str1`, затем туда же были скопированы два символа, начиная с индекса 1 (получилось значение "os"). Наконец, из исходной строки `str1` были извлечены вызовом

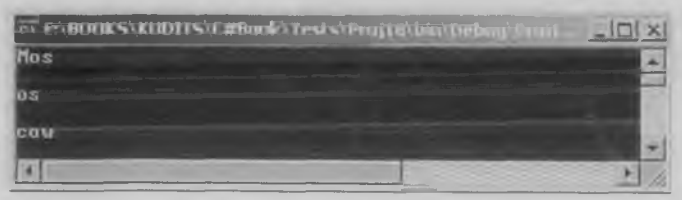


Рис. 4.9.

```
str1.Substring( str1.Length - 3, 3 )
```

три последние символа (получилось строковое значение "cow").

Кроме технической задачи об извлечении подстрок очень распространена задача о поиске внутри строки заданной подстроки с тем, чтобы изъять эту подстроку или заменить ее на другую подстроку, или просто идентифицировать текст по входящему в него фрагменту.

Для решения этой задачи следует применить метод `IndexOf()`:

```
ind = str.IndexOf( fragment, StartIndex );
```

который ищет местоположение начала подстроки `fragment` внутри строки `str`, начиная поиск с позиции `StartIndex`, и возвращает индекс первого символа искомого фрагмента. Если фрагмент внутри текстового отрезка не находится, то возвращается значение `-1`.

В связи с методом `IndexOf()` возникает вопрос о том, как с его помощью найти все вхождения подстроки в одну и ту же строку, а не только первое из них? Ответ на этот вопрос дает следующая модельная учебная программа:

Листинг 4.10.

```
using System;
class main
{
  static public int Main( )
  {
    string Text = "When you say \"yes\", I say \"yes\" too";
    string Frag = "yes"; string NewFrag = "no";
    int lenOld = Frag.Length, lenNew = NewFrag.Length, ind;

    Console.WriteLine( "Оригинальный текст: " + Text );

    ind = Text.IndexOf( Frag, 0 );
    while( ind >= 0 )
    {
      ind = Text.IndexOf( Frag, ind );
```

```
      if( ind >= 0 )
      {
        Text = Text.Remove( ind, lenOld );
        Text = Text.Insert( ind, NewFrag );
        ind += lenNew;
      }
    }
    Console.WriteLine( "Обработанный текст: " + Text );
    Console.ReadLine();
    return 0;
  }
}
```

Программа с Листинга 4.10 призвана найти все вхождения слова "yes" в исходном тексте "When you say \"yes\", I say \"yes\" too" (здесь используются комбинации \" для внесения внутрь текстовой константы двойных кавычек в качестве содержательных, а не ограничительных символов) и заменить его на слово "no".

Дисплейный вывод этой программы показан на рис. 4.10.

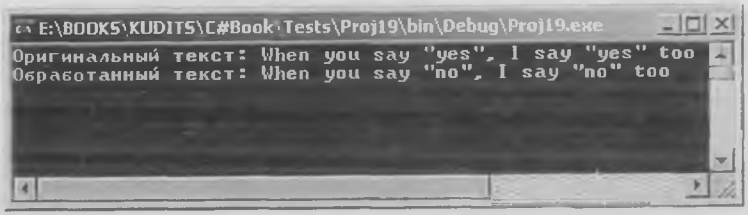


Рис. 4.10.

Из рис. 4.10 видно, что программа с Листинга 4.10 справляется с поставленной задачей. Разберем теперь детали ее работы.

Собственно, вся работа заключена в следующих трех строчках кода:

```
Text = Text.Remove( ind, lenOld );
Text = Text.Insert( ind, NewFrag );
ind += lenNew;
```

Поясним этот фрагмент: определив с помощью метода `IndexOf()` индекс `ind` искомого фрагмента, удаляем его методом `Remove()`, внедряем в то же место методом `Insert()` новый фрагмент `NewFrag` и поправляем текущий индекс `ind` для следующего поиска.

Все очень просто, но нужно отметить, что методы *Remove()* и *Insert()* формируют *новые строки* в качестве своих *возвратов*, поэтому мы их и запоми-

наем в переменной `Text` (то есть ссылке на объекты типа string), а иначе они для нашей программы будут потеряны (ими займется сборщик мусора).

Если нужно искать в тексте не фрагмент, а одиночный символ, то можно воспользоваться *перегруженным вариантом* метода `IndexOf()`, принимающим символ в качестве первого параметра:

```
string Text = "Moscow"; char ch = 's'; int ind;
ind = Text.IndexOf( ch, 0 );
```

В результате выполнения данного фрагмента кода переменная `ind` станет равной 2, так как символ 's' входит в строку "Moscow" на позиции с индексом 2.

Метод `IndexOf()` выполняет поиск с учетом регистра символов. Если нужно выполнить поиск без учета регистра символов, то можно исходный текст и искомый фрагмент привести сначала к одному и тому же регистру с помощью методов `ToLower()` или `ToUpper()` класса `String`

```
Text = Text.ToUpper(); Frag = Frag.ToUpper();
```

и только потом выполнить поиск методом `IndexOf()`.

Про работу иных методов класса String всегда можно узнать в справочной системе компилятора Microsoft Visual C# (см. рис. 4.11).

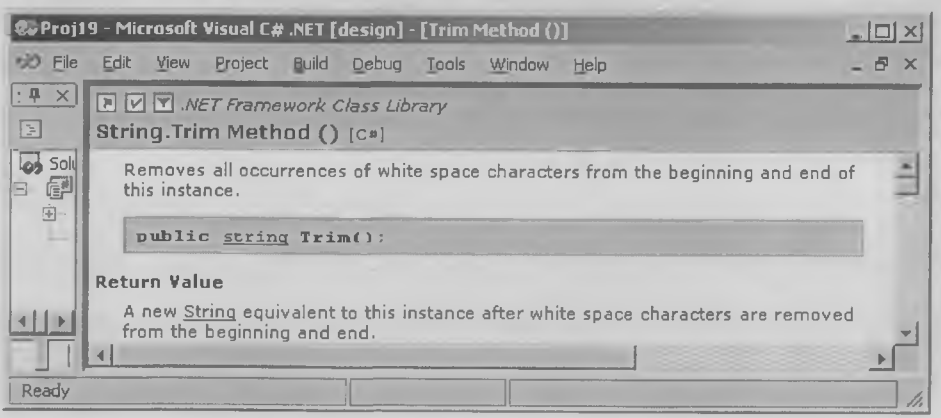


Рис. 4.11.

В частности, на рис. 4.11 показана справочная информация по методу `Trim()` класса String, который призван удалять пробелы с обоих концов строки. Во многих статьях справочной системы присутствуют примеры кода, использующие обсуждаемые методы.

4.4. Массивы строк в языке C#

Строки являются важным, можно сказать инфраструктурным типом языка C#. Многие программы, не направленные на обработку текстов как главную цель своей деятельности, все равно вынуждены работать с текстовыми строками, ибо общаться с пользователем удобно именно в текстовой форме. Для этого они хранят изрядное количество текстовых сообщений. Все эти сообщения удобно хранить централизованно в виде одного массива строк.

Массивы строк можно создать либо следующим подробным кодом:

```
string[] strArr = null;
strArr = new string[3];
strArr[0] = "first";
strArr[1] = "second";
strArr[2] = "third";
```

либо применить более короткую «инициализационную» форму:

```
string[] strArr = new string[3]{"first","second","third"};
```

В любом случае массив `strArr[]` (все про массивы языка C# было нами изучено в гл. 3) после этого содержит три элемента, каждый из которых является ссылкой на объект-строку в памяти компьютера, что с помощью окна QuickWatch (про вызов окна QuickWatch см. выше разд. 3.3) визуализируется достаточно наглядно (см. рис. 4.12).

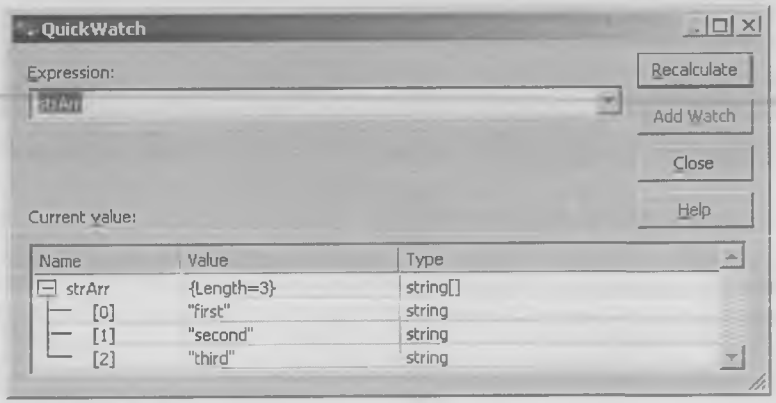


Рис. 4.12.

Проще простого вывести для обозрения содержимое массива `strArr` и в собственной программе:

Листинг 4.11.

```
using System;
class main
{
  static public int Main( )
  {
    string[] strArr = new string[3]{"second", "first", "third"};

    for( int i = 0; i < strArr.Length; ++i )
      Console.WriteLine( strArr[i] );

    Console.ReadLine();
    return 0;
  }
}
```

Абсолютно очевидный и ожидаемый нами результат ее вывода в консольное окно показан на рис. 4.13.

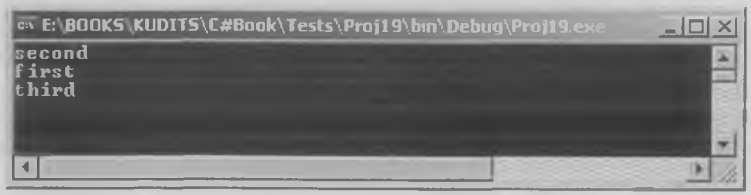


Рис. 4.13.

Если перед выводом содержимого массива строк отсортировать его статическим методом `Sort()` класса `Array` (см. про этот метод разд. 3.5)

```
Array.Sort( strArr );
```

то вывод строк будет осуществлен в лексикографическом порядке (см. рис. 4.14).

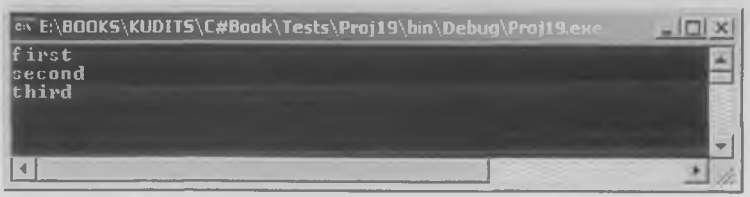


Рис. 4.14.

Мы специально в учебных целях в Листинге 4.11 применили операцию индексации для доступа к отдельным строкам массива `strArr`. На самом деле этот доступ удобнее и проще осуществлять в цикле `foreach`, пример использования которого мы рассмотрим чуть ниже, а сейчас мы поступим (опять-таки в учебных целях) прямо наоборот – пойдем еще более трудным и громоздким путем, осуществляя вывод строк массива `strArr` посимвольно (с теми же самыми картинками в консольном окне, что показаны на рис. 4.13 или 4.14):

```
for( int i = 0; i < strArr.Length; ++i )
{
  for( int j = 0; j < strArr[i].Length; ++j )
    Console.Write( strArr[i][j] );

  Console.WriteLine();
}
```

Здесь метод `Write()` осуществляет вывод символов (представлены выражением `strArr[i][j]`) i-ой строки массива в одну и ту же физическую строку экрана, после чего метод `WriteLine()` переходит к следующей физической строке (то есть осуществляет «перевод строки»).

Как мы и обещали выше, сразу же после учебных упражнений в технических усложнениях кода Листинга 4.11 перейдем к его практическому упрощения с помощью цикла foreach:

```
foreach( string el in strArr )
  Console.WriteLine( el );
```

Действительно, все кратко, просто и наглядно, и, разумеется, этот вариант наиболее предпочтителен в реальном практическом коде.

Цикл `foreach` как нельзя лучше подходит для решения задачи о расщеплении текстовой строки на отдельные ее фрагменты, отделенные друг от друга так называемыми *разделителями* (*separators*). К стандартным разделителям относят точку, запятую и пробел, но в методе `Split()`, выполняющем данную задачу, их можно указать явно в качестве второго параметра:

Листинг 4.12.

```
using System;
class main
{
  static public int Main( )
  {
    string str = "second, first, third";
    char[] sep = new char[3] {'.', ',',' '};
```

```
    foreach( string s in str.Split( sep ) )
      Console.WriteLine( s );

    Console.ReadLine();
    return 0;
  }
}
```

Программа с Листинга 4.12 разбивает с помощью метода `Split()` библиотечного класса `String` исходную строку "second, first, third" на три подстроки ("second", "first" и "third"), и с помощью цикла `foreach` выводит каждую из них в консольное окно.

4.5. Преобразование чисел в строки и обратно

В связи со строками языка C# часто возникает вопрос о возможности преобразования строкового представления числа в значения типа `double` или `int`. Например, может потребоваться строку "7.3921" преобразовать в значение типа `double`, или строку "-12345" – в значение типа `int`. Это может потребоваться в случаях, когда исходная информация поступила в текстовой форме, содержащей внутри себя строковые представления чисел, с которыми в программе нужно производить математические вычисления.

Действительно, еще в разд. 1.5 мы столкнулись с необходимостью преобразовывать строки в числа типа `double`, поскольку метод `ReadLine()` библиотечного класса `Console`, выполняющий ввод с клавиатуры, осуществляет возврат в текстовой форме. Поэтому в разд. 1.5 мы преобразовывали возврат метода `ReadLine()` в числа типа `double` с помощью метода `ToDouble()` библиотечного класса `Convert`:

```
x1 = Convert.ToDouble( Console.ReadLine() );
```

Но аналогичным образом можно преобразовывать ввод с клавиатуры в числа типа int, или в символы (то есть тип char):

```
int  I   = Convert.ToInt32( Console.ReadLine() );
char ch  = Convert.ToChar( Console.ReadLine() );
```

Ясно, что таким образом можно любую строку (а не только возврат метода `ReadLine()`) превратить в число или символ. Однако, следует иметь в виду, что от входной строки методы класса `Convert` ожидают *абсолютно корректного представления чисел*. В случае, когда строковое представление числа некорректное, например "4.567.67", где присутствуют две десятичные точки, методы класса `Convert` прерывают нормальный ход выполнения программы

и генерируют исключения среды CLR (ранее пример исключений среды CLR и соответствующих им сообщений приведен в разд. 3.2, рис. 3.2). Например, на попытку преобразования

```
int  I   = Convert.ToInt32( "4.567.67" );
```

вы получите следующее сообщение о необработанной исключительной ситуации

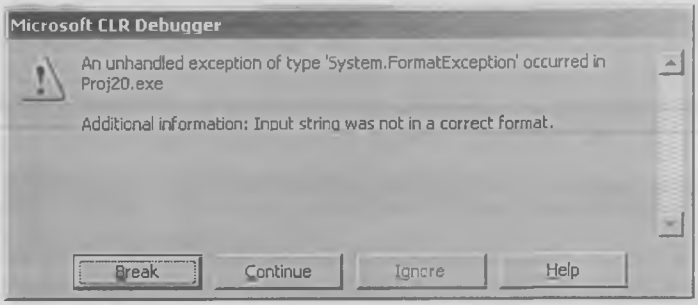


Рис. 4.15.

Одной из основных методик реагирования на возникновение ошибочных ситуаций в процессе выполнения программы является техника перехвата и программной обработки исключений, но в нашем пособии для начинающих мы этой темы не затрагиваем (см. на эту тему книгу [5]).

Для повышения надежности выполнения программы можно также осуществить самостоятельную программную проверку содержимого строки (для этого нам пригодится весь материал текущей главы) прежде, чем передавать ее методам класса `Convert`.

У класса `Convert` имеются методы и для решения *обратной задачи*: например, дано числовое значение (целое или дробное) и его нужно преобразовать в строковое представление. Вот элементарный пример на эту тему:

```
int I = 333; string st = Convert.ToString( I );
```

который не требует никаких комментариев.

Но даже и без класса `Convert` с этой задачей мы знакомы давно, ведь такое преобразование осуществляет метод `WriteLine()` библиотечного класса `Console`, выводя данные разных типов на дисплей компьютера в виде имеющих традиционный зрительный образ стандартных символов (то есть в текстовом виде). Но метод `WriteLine()` осуществляет все необходимые преобразования «на лету», сразу же выводя результаты на дисплей.

Если же нам нужно еще поработать с результатами преобразований, то вместо класса Console нужно использовать библиотечный класс StringBuilder (определен в пространстве имен System.Text), метод AppendFormat() которого делает все то же самое, что и метод WriteLine() класса Console, но результат она направляет не на дисплей, а формирует в своих внутренних структурах данных (в полях данных), которые методом ToString() всегда можно преобразовать в строку. Наша программа дальше может работать с этой строкой так, как ей требуется по смыслу решаемой задачи:

Листинг 4.13.

```
using System;
using System.Text;
class main
{
  static public int Main( )
  {
    double d = 3.456; int m = 33; string st;
    StringBuilder sb = new StringBuilder();

    // Прямой вывод методом WriteLine():
    Console.WriteLine( "num1 = {0}\nnum2 = {1}\n", m, d );

    // Предварительное формирование строки st:
    sb.AppendFormat( "num1 = {0}\nnum2 = {1}\n", m, d );
    st = sb.ToString();
    Console.WriteLine( st );

    Console.ReadLine();
    return 0;
  }
}
```

В программе с Листинга 4.13 применяются два способа вывода информации на дисплей компьютера: прямой способ, когда методом WriteLine() обычным образом выводят на дисплей значения переменных m и d, и способ вывода информации из промежуточной строковой переменной st, информацию в которую помещают с помощью метода AppendFormat() класса StringBuilder.

Все, что метод WriteLine() напрямую выводит на экран дисплея, все это мы предварительно помещаем в строку st. И только после этого вызываем метод WriteLine() для отображения содержимого этой строки на дисплее. Ниже на рис. 4.16 показано окно работающей программы с Листинга 4.13.

Рис. 4.16.

Из рис. 4.16 видно, что оба способа вывода приводят к одному и тому же результату.

К библиотечному классу `StringBuilder` стоит присмотреться повнимательнее, ибо он позволяет «одним взмахом руки» решать довольно крупные задачи. Например, метод `Replace()` позволяет за раз решить всю задачу из Листинга 4.10 (замена всех вхождений подфрагмента строки на иной фрагмент):

```
string Text = "When you say \"yes\", I say \"yes\" too";
StringBuilder sb = new StringBuilder( Text );
Console.WriteLine( sb.Replace( "yes", "no" ).ToString() );
```

Иной (перегруженный) вариант метода `Replace()` можно использовать для замены отдельных символов внутри некоторой строки.

Подводя итог данной главы, следует признать, что в рамках языка C# и платформы Microsoft NET Framework строковый тип данных реализован невероятно мощно (контраст с языком C просто гигантский). Использовать этот тип в программировании весьма просто и одновременно безопасно.

Часть II

Объектно-ориентированное программирование на языке C# и библиотека классов Microsoft NET Framework

Глава 5. Классы языка C#: свойства, агрегация, наследование, полиморфизм

5.1. Определение свойств в классах языка C# и их использование в клиентском коде

До сих пор мы создавали классы C#, объединяя поля данных и обрабатывающие их методы. Однако, в языке C# (в отличие от его предшественника – языка C++) для классов имеется еще один строительный элемент – так называемые *свойства* (*properties*). Мы сейчас рассмотрим синтаксис определения свойств в классах языка C# и их использование в клиентском коде сначала на формальном учебном примере, а затем применим полученные знания для дальнейшего улучшения разработанного нами ранее класса MyComplex, моделирующего поведение комплексных чисел (см. Листинг 2.5 из разд. 2.4).

Итак, вот исходный пример тривиального учебного класса (пока что не использующего свойства):

Листинг 5.1.

```
class Trivial
{
  // Methods:
  public void Set( double x ){ X = x; }
  public double Square( ){ return X * X; }
```

```
  // Private data field:
  private double X;
}
```

Объекты класса `Trivial` хранят в себе одно вещественное число, а методы `Set()` и `Square()` этого класса позволяют устанавливать значение числового поля и вычислять его квадрат, соответственно.

Работа клиентского кода с объектами класса `Trivial` выглядит следующим образом:

```
double res;
Trivial ob = new Trivial();
...
ob.Set( 3.5 );
res = ob.Square();
```

В общем, тут нет ничего сложного, но было бы естественнее, прямолинейнее (без лишних экивоков) и, в конце концов, проще, если бы клиентский код мог оперировать хранящимся в классовом объекте вещественным числом напрямую, используя самую наглядную форму доступа к целевому значению – по его имени:

Листинг 5.2.

```
ob.X = 3.5;
res = ob.X * ob.X;
```

Для класса `Trivial` из Листинга 5.1 такой вариант клиентского кода не проходит (возникает ошибка компиляции) из-за закрытого характера доступа к полю данных `X`, и пока что мы рассматриваем его просто как гипотетический пример. С точки зрения клиентского кода с Листинга 5.2 идентификатор `X` символизирует одно из *объектных свойств* (*атрибутов*), в совокупности определяющих текущее состояние объекта `ob` типа `Trivial`. Задавая напрямую значения атрибутов, клиент явным и наглядным образом формирует необходимое состояние классового объекта.

Рассмотренная схема явного управления объектными атрибутами (свойствами) известна «от веку» и традиционно рассматривалась во всех учебниках по объектно-ориентированному проектированию и программированию. Однако жизнь (практика) взяла свое и после накопления исторически изрядного опыта программирования на языке C++ (примерно в течение одного-двух десятилетий) было сформулировано правило, что *делать классовые поля данных открытыми* (public) *нехорошо* – в этом случае программный код класса слишком сильно связан с клиентским кодом и в процессе эксплуатации и постепенного совершенствова-

ния классового устройства код клиента также приходится непрерывно изменять (примерно, как постепенное совершенствование конструкции автомобиля вдруг потребовало бы непрерывного переучивания всех водителей). В соответствии с указанной рекомендацией мы и определили для поля данных X класса Trivial (см. Листинг 5.1) режим закрытого доступа (применили модификатор private).

В общем, противоречие в желательности простого атрибутного синтаксиса работы с классовыми объектами (Листинг 5.2) и практически обоснованным правилом закрытия полей данных (Листинг 5.1) налицо. В языке C++ это противоречие не разрешается, а приоритет отдается надежности и технологичности в ущерб простоте. В языке же C# это противоречие снимается за счет введения нового типа членов класса, так называемых *свойств*.

Вот вариант определения класса Trivial, использующий свойства классов языка C#:

Листинг 5.3.

```
class Trivial
{
  // Method:
  public double Square( ){ return m_X * m_X; }

  // Public property X:
  public double X                                                    /*1*/
  {                                                                  /*2*/
    get { return m_X;  } // get-block: to get m_X value             /*3*/
    set { m_X = value; } // set-block: to set m_x value             /*4*/
  }                                                                   /*5*/

  // Private data field:
  private double m_X;
}
```

Здесь именем X названо открытое (public) *свойство* класса Trivial, определение которого сосредоточено в строках /*1*/-/*5*/, а единственное закрытое (private) поле данных переименовано в m_X (совпадение имен членов класса не допускается).

Как мы видим из Листинга 5.3 *определение свойства* имеет *заголовок* с именем и типом свойства (здесь это double), а также с модификатором режима доступа (здесь это public). В *теле определения свойства* (заключено в фигурные скобки) присутствуют *get-* и *set-блоки*, называемые таким образом по соответствующим *ключевым словам get* и *set*. Каждый из этих блоков (может присутствовать и только один из них) представляет из себя завуалированное определение

классового метода, тело которого сосредоточено внутри фигурных скобок блока. После компиляции определения класса `Trivial` его get-блок превращается в метод `get_X()`, а set-блок – в метод `set_X()`. Второй их этих методов вызывается при установке нового значения свойства (когда оно используется слева от знака присваивания), причем параметр этого метода и есть это самое новое значение, и которому в завуалированном определении соответствует *ключевое слово value* языка C#.

Клиентскому коду ничего не надо знать о таких хитростях, и он может работать с открытым свойством X как-будто с открытым полем данных, и код с Листинга 5.2 теперь безошибочно компилируется и прекрасно работает. Причем, как мы уже догадываемся, для строки

```
ob.X = 3.5;
```

компилятором будет вызван код завуалированного метода `set_X()`, а для строки

```
res = ob.X * ob.X;
```

таковым уже будет метод `get_X()`.

В общем ясно, что свойства классов языка C# позволяют одновременно и «волкам быть сытыми, и овцам остаться целыми».

Как мы и обещали в самом начале раздела, применим свойства для дальнейшего улучшения разработанного нами ранее класса `MyComplex`, моделирующего поведение комплексных чисел (см. Листинг 2.5 из разд. 2.4):

Листинг 5.4.

```
class MyComplex
{
  // Constructors:
  public MyComplex( ){ R = I = 0; }
  public MyComplex( double r, double i ){ R = r; I = i; }
  public MyComplex( MyComplex z ){ R = z.R; I = z.I; }

  // Methods:
  public void Print( )
  { Console.WriteLine( "Complex number = {0}+{1}i", R, I ); }
  static public double Abs( MyComplex z )
  { return z.Length; }

  // Properties:
  public double Length
  { get { return Math.Sqrt( R*R + I*I ); } }
```

```
  public double Re { get{ return R; } set{ R = value; } }
  public double Im { get{ return I; } set{ I = value; } }

  // Overloaded operators:
  static public MyComplex operator+(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R + rOp.R, lOp.I + rOp.I ); }
  static public MyComplex operator-(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R - rOp.R, lOp.I - rOp.I ); }
  static public MyComplex operator*(MyComplex lOp,MyComplex rOp)
  {
    double Re = lOp.R * rOp.R - lOp.I * rOp.I;
    double Im = lOp.R * rOp.I + lOp.I * rOp.R;
    return new MyComplex( Re, Im );
  }

  // Private data fields:
  private double R;
  private double I;
}
```

Ранее открытое в классе `MyComplex` поле данных `Length` (см. Листинг 2.5) трансформировалось в Листинге 5.4 в одноименное свойство с одним лишь get-блоком (set-блок здесь, конечно, смысла не имеет), а вместо методов `SetR()` и `SetI()` появились свойства `Re` и `Im`, соответственно.

В результате клиентский код теперь имеет возможность устанавливать простыми и наглядными операциями присваивания значения вещественной и мнимой частей комплексных чисел, что выглядит просто замечательно:

```
MyComplex z1 = new MyComplex();
z1.Re = 1; z1.Im = 2;
```

В предыдущих двух главах пособия мы изучали встроенные типы данных – массивы и строки языка C#. Оба этих типа базируются на соответствующих библиотечных классах. Для объектов указанных типов мы самым активным образом использовали выражения, содержащие идентификатор `Length`. Сейчас самое время сообщить, что этим идентификатором в определениях соответствующих классов поименовано их открытое свойство (а не поле данных). Кроме того, во многих иных классах библиотеки FCL (встроенная библиотека классов платформы Microsoft NET Framework) свойства используются широчайшим образом, и мы, отталкиваясь от приобретенных в данном разделе знаний, уже вполне понимаем причины таких предпочтений.

5.2. Взаимосвязи и взаимозависимости классов: агрегация и наследование

Если в языке C# класс определен так, что он не содержит полей иных абстрактных типов данных, то класс этот является *независимым* по отношению к другим классовым типам программы:

Листинг 5.5.

```
class MyTest1
{
  public void Set( int x ){ X1 = x; }
  private int X1;
}
class MyTest2
{
  public void Set( int x ){ X2 = x; }
  private int X2;
}
class main
{
  public static int Main( )
  {
    MyTest1 ob1 = new MyTest1();
    MyTest2 ob2 = new MyTest2();
    ob1.Set( 1 ); ob2.Set( 2 );
    return 0;
  }
}
```

В программе с Листинга 5.5 мы ввели два *независимых абстрактных типа* (класса) – `MyTest1` и `MyTest2`. Действительно, формально любой из них может быть опущен без ущерба для существования другого типа.

Для наглядного графического отображения системы типов программы рекомендуется применять стандартизированный язык графической визуализации *UML* (*Unified Modelling Language*). На рис. 5.1 показана диаграмма UML для системы типов программы с Листинга 5.5:

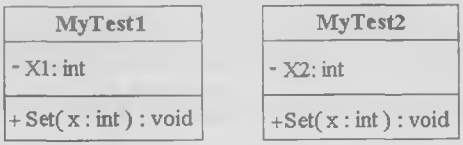


Рис. 5.1.

Из рис. 5.1 видно, что в стандарте UML абстрактные типы (классы) изображаются в виде трехсекционных прямоугольников, отдельные секции которых предназначены для отображения имени типа, его полей данных и методов. Минус на диаграммах UML символизирует закрытость члена класса, а плюс – открытость.

В укороченном варианте диаграммы UML для типов данных содержат лишь прямоугольники с именами типов.

Теперь изменим программу с Листинга 5.5 так, чтобы она содержала зависимую друг от друга систему типов:

Листинг 5.6.

```
class MyTest1
{
  public void Set( int x ){ X1 = x; }
  private int X1;
}
class MyTest2
{
  public MyTest2( )
  { X2 = 0; mt1 = new MyTest1(); }
  public void Set( int x, int y )
  { X2 = x; mt1.Set( y ); }

  // Private data:
  private int      X2;
  private MyTest1  mt1;
}
class main
{
  public static int Main( )
  {
    MyTest1 ob1 = new MyTest1();
    MyTest2 ob2 = new MyTest2();
    ob1.Set(1); ob2.Set(2,3);
    return 0;
  }
}
```

Теперь уже нельзя просто так опустить тип `MyTest1`, ибо в классе `MyTest2` имеется поле данных `mt1` этого типа. Получается, что тип `MyTest2` зависит от типа `MyTest1` (в обратном направлении зависимости нет). В графиче-

ском языке UML зависимость типов отображается с помощью соединяющего прямоугольники отрезка прямой со стрелкой на конце:

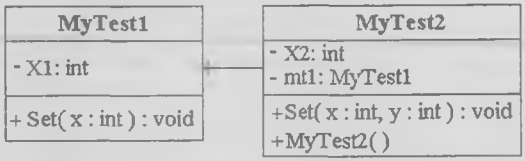


Рис. 5.2.

Стрелка показывает *направление зависимости*: от зависимого класса к независимому. В итоге, UML-диаграмма на рис. 5.2 наглядно показывает, что класс `MyTest2` зависит от класса `MyTest1`.

Помимо констатации зависимости вообще, в UML принято выделять более частные случаи зависимостей. Как раз такой случай и продемонстрирован в Листинге 5.6, когда конструктор класса `MyTest2` автоматически создает объект типа `MyTest1`, на который настраивается его поле данных mt1. В результате, для клиентского кода объект класса `MyTest2` выглядит очень цельным и монолитным объектом, *агрегирующим* у себя внутри (то есть, *включающим как часть*) объект класса `MyTest1` закрытым образом (у клиента нет к нему прямого доступа). Такой вариант зависимости классов принято называть их *агрегацией* (*classes aggregation*).

Для отношения агрегации в UML предусмотрена специальная форма графического отображения – отрезок прямой с ромбом на том конце, который соответствует агрегирующему классу (см. рис. 5.3).

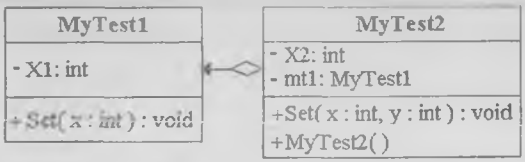


Рис. 5.3.

На рис. 5.3 наглядно показано, что класс `MyTest2` является *агрегирующим*, а класс `MyTest1` – *агрегируемым*.

Наибольшая степень агрегации классов достигается в случае, когда агрегируемый класс сам определяется внутри определения агрегирующего класса. Про такое определение класса говорят как о *вложенном определении класса* (*nested class*):

```
class MyTest2
{
  public MyTest2( )
  { X2 = 0; mt1 = new MyTest1(); }
  public void Set( int x, int y )
  { X2 = x; mt1.Set( y ); }

  // Data field of primitive type:
  private int X2;

  // Nested type:
  private class MyTest1
  {
    public void Set( int x ){ X1 = x; }
    private int X1;
  }

  // Data field of nested type:
  private MyTest1  mt1;
}
class main
{
  public static int Main( )
  {
    //MyTest2.MyTest1 ob1 = new MyTest2.MyTest1();   /*1*/
    MyTest2 ob2 = new MyTest2();
    ob2.Set(2,3);
    return 0;
  }
}
```

Определив класс `MyTest1` с модификатором доступа `private` внутри определения класса `MyTest2`, мы превратили его в чисто агрегируемый класс, так как стало невозможным в клиентском коде создавать объекты этого типа (поэтому мы и закомментировали строку `/*1*/` в данном примере).

Объекты агрегируемых классов представляются проектировщику программы составными частями (деталями), входящими в более крупные объекты агрегирующих классов. Поэтому часто говорят, что объекты агрегирующих классов содержат в себе объекты агрегируемых классов, а про *отношение агрегации типов* говорят как об их зависимости вида "*has a*" (*иметь в качестве составной части*). Отношение агрегации типов широчайшим образом применяется в реальной программистской практике.

Другой известной и широко применяемой формой зависимости классов (абстрактных типов) является отношение "*is a*" (*быть частным случаем более общего типа*). Эта форма классовой зависимости реализуется в языке C# (как и в языке C++) с помощью *механизма наследования классов*, к изучению которого мы сейчас и переходим.

Итак, пусть имеется некоторый достаточно общий (относительно последующих уточненных вариантов) класс MyBase:

```
class MyBase
{
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public void Print( ){ Console.WriteLine("X1={0}",X1); }
  private int X1;
}
```

Требуется создать класс, похожий на MyBase и содержащий те же самые члены, плюс дополнительное целочисленное поле и неизбежные в связи с этим модификации методов.

Вот как это автоматически реализуется с помощью *механизма наследования классов*:

```
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y )
  { Set( x ); X2 = y; }
  public void Print( )
  { base.Print(); Console.WriteLine( "X2={0}",X2); }
  private int X2;
}
```

В заголовке *производного класса* MyDerived после имени класса через двоеточие указывается имя *базового класса* (в нашем случае это MyBase). В результате производный класс MyDerived автоматически наследует все члены базового класса MyBase. Например, под объекты класса MyDerived в памяти компьютера будет отводиться область памяти, в совокупности достаточная для размещения как целочисленного поля X1, достающегося от базового класса, так и для целочисленного поля X2, явно добавленного в производном классе. Кроме того, код методов производного класса может напрямую обращаться ко всем открытым членам базового класса (как полям данных, так и к методам).

Например, в коде метода Set() производного класса MyDerived осуществляется вызов метода Set() базового класса MyBase (противоречия из-за совпадения имен снимаются, так как эти методы имеют разное число параметров), а в методе Print() производного класса осуществляется вызов одноименного метода базового класса с помощью спецификации уточняющим *ключевым словом base*. Также с помощью ключевого слова base в конструкторе производного класса осуществляется вызов и передача параметра конструктору базового класса.

Закрытое ключевым словом private поле X1 базового класса можно открыть исключительно и только лишь для методов производных классов с помощью *замены ключевого слова private на protected*:

```
class MyBase
{
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public void Print( ){ Console.WriteLine("X1={0}",X1); }
  protected int X1;
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public void Print( ){Console.WriteLine( "X1={0} X2={1}",X1,X2);}
  private int X2;
}
```

Теперь в методах производного класса MyDerived доступ к полю X1 базового класса осуществляется напрямую (отпала необходимость в вызове методов Set() и Print() базового класса), из-за чего эффективность кода только повышается.

Приведем пример программы, определяющей классы MyBase и MyDerived и использующей (в клиентской части) объекты этих классов:

Листинг 5.7.

```
using System;
class MyBase
{
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public void Print( ){ Console.WriteLine("X1={0}",X1); }
  protected int X1;
```

```
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public void Print( ){Console.WriteLine( "X1={0}
X2={0}",X1,X2);}
  private int X2;
}
class main
{
  public static int Main( )
  {
    MyBase    ob1 = new MyBase(1);
    MyDerived ob2 = new MyDerived(2,3);

    ob1.Set( 6 );  ob2.Set( 7, 8 );
    ob1.Print(); ob2.Print();

    Console.ReadLine();
    return 0;
  }
}
```

Программа с Листинга 5.7 успешно компилируется и после запуска демонстрирует работу, находящуюся в полном соответствии с выше сказанным про устройство объектов базового и производного классов и их поведение (см. рис. 5.4).

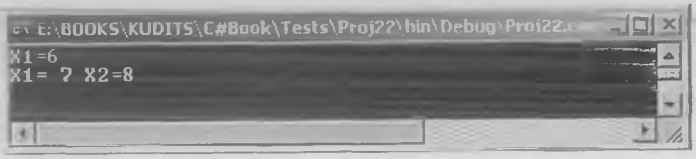


Рис. 5.4.

Для графического отображения *отношений наследования типов* язык UML предлагает использовать в качестве стрелки (в направлении от зависимого класса к независимому, то есть к базовому) незакрашенный треугольник (см. рис. 5.5).

Рис. 5.5.

На рис. 5.5 мы для краткости использовали укороченные односекционные варианты графического отображения типов (классов). Следует еще отметить, что из-за строгой направленности отношений наследования и явного отображения этого направления на диаграммах UML нет никакой практической необходимости в специальном порядке пространственного отображения базового и производного классов. Однако, так сложилось исторически, что в литературе чаще всего прямоугольник базового класса стараются отображать в вертикальном направлении выше прямоугольника производного класса (см. рис. 5.6).

Рис. 5.6.

Еще раз подчеркнем, что в случае аккуратного применения стандартизированных диаграмм UML в таком пространственном упорядочении изображающих классы прямоугольников нет никакой необходимости, и мы в нашем пособии будем поступать так, как нам будет удобнее в конкретных случаях.

Отношения наследования типов часто называют *отношениями обобщения*, ибо базовый класс представляет собой тип, более общий по отношению к производному типу. Действительно, так как *объект производного класса* помимо прочего содержит все поля данных базового класса и может абсолютно безопасно вызывать для их обработки методы базового класса (они для этого и предназначены), то он *может рассматриваться и как объект базового класса*. С этой целью в языке C# разрешается адресовать объекты производного класса ссылками на базовый класс:

```
MyBase  ob2 = new MyDerived(2,3);
ob2.Set( 7 );
```

В данном примере выражением `new MyDerived(2,3)` создается объект производного класса `MyDerived`, но адресуется он ссылкой `ob2` типа `MyBase`, то есть ссылкой на базовый класс. В результате с помощью выражения `ob2.Set(7)` для объекта производного класса вызывается метод `Set()` базового класса, имеющий один параметр. Как мы понимаем, в результате будет значением 7 прописано поле данных, достающееся этому объекту от базового класса `MyBase` (то есть поле `X1`, см. Листинг 5.7).

5.3. Обобщенный клиентский код: виртуальные методы и полиморфизм

В предыдущем разделе мы рассмотрели агрегацию классов и наследование классов с точки зрения синтаксиса языка C#. Оба этих механизма построения классов C# позволяют *эффективно использовать предыдущие наработки.*

Действительно, пусть имеется ранее разработанный класс `T1`, который в процессе выполнения нового проекта понимается как тип, описывающий *отдельную деталь более крупного типа* `T2`. Вместо того, чтобы тупо повторять устройство класса `T1` в рамках нового типа `T2`, быстрее и проще применить агрегацию:

```
class T2
{
...
T1 m_T1;
...
}
```

то есть агрегировать тип `T1` в тело создающегося класса `T2` в виде поля данных `m_T1` типа `T1`. Кроме экономии усилий можно еще отметить более четкую структуру получающегося при этом результирующего кода, что чрезвычайно важно для последующих его изменений (совершенствования), ибо такой код легче понять сторонним разработчикам (или вспомнить автору кода).

Мотивы и цели практического использования наследования классов те же самые и лишь чуть более сложные для успешного воплощения. Наследование классов целесообразно применять тогда, когда в рамках программной разработки возникает *множество похожих типов данных,* реализации которых по необходимости будут содержать большое количество одинаковых полей данных и методов.

Действительно, вместо того, чтобы реализовывать все множество типов независимым образом, с неизбежным тупым повтором кода и плохой структурированностью результирующей программы, можно сначала выделить небольшое количество базовых типов, содержащих общие для всех типов поля и методы. Остальные же типы удобно определять с привлечением механизма наследования, при котором все выполненные ранее наработки (код методов и структура полей данных базовых классов) «автоматически перетекут» в производные типы:

```
class Base1 { ... }
class Derived1 : Base1 { ... }
```

Кроме указанных выше экономии усилий и лучшей структурированности программного текста, при использовании наследования классов имеется возможность писать *обобщенный* (неизменный для разных типов данных) *клиентский*

код для выполнения одинаковых по смыслу действий с объектами разных классов (базовых и производных) – классов MyBase и MyDerived из Листинга 5.7:

Листинг 5.8.

```
using System;
class MyBase
{
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public void Print( ){ Console.WriteLine("X1={0}",X1); }
  protected int X1;
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public void Print( ){Console.WriteLine( "X1={0} X2={0}",X1,X2);}
  private int X2;
}
class main
{
  public static int Main( )
  {
    MyBase ob1 = new MyBase(1);
    MyBase ob2 = new MyDerived(2,3);

    GenericFunc( ob1, 8 );
    GenericFunc( ob2, 9 );

    Console.ReadLine();
    return 0;
  }
  static void GenericFunc( MyBase ob, int val )
  {
    ob.Set( val );
    ob.Print();
  }
}
```

Здесь обобщенный клиентский код (часть клиентского кода) сосредоточен в методе GenericFunc(), согласно определению принимающем в качестве первого параметра объекты базового класса MyBase. Но в предыдущем разделе

мы узнали, что компилятор объекты производного класса соглашается засчитывать за объекты базовых типов (см. Листинг 5.7), так что реально можно вызывать метод `GenericFunc()` для объектов обоих типов, что и делается в Листинге 5.8.

Кроме того, объекты базовых и производных классов по той же самой причине можно хранить в единственном массиве типа `MyBase[]`, а вызовы обобщенного метода `GenericFunc()` осуществлять для элементов массива:

```
MyBase[] obAr = new MyBase[2]{new MyBase(1),new MyDerived(2,3)};
GenericFunc( obAr[0], 8 );
GenericFunc( obAr[1], 9 );
```

В любом случае удается над объектами разных типов осуществлять обобщенным клиентским кодом одинаковую работу. В частности, для Листинга 5.8 наблюдается следующий результат в консольном окне работающей программы (см. рис. 5.7).

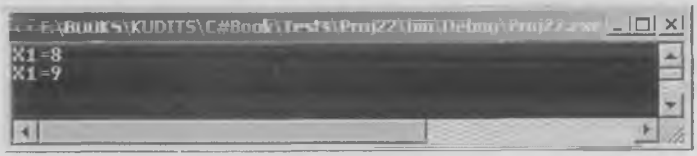


Рис. 5.7.

Из рис. 5.7 с наглядностью видно то, что мы и так знаем – обобщенный клиентский код вызывает для объектов базового и производного типов лишь методы базового типа, что безусловно допустимо и безопасно.

Но, тем не менее, было бы еще лучше, если бы имелась принципиальная возможность вызывать для таких объектов методы их родного класса. Например, когда обобщенный метод `GenericFunc()` вызывается для объекта производного класса `MyDerived`, желательно, чтобы вызывался метод `Print()` именно производного класса, ибо одноименный метод базового класса не может вывести полную информацию об устройстве объектов производного типа (что мы и наблюдаем на рис. 5.7). Такое желаемое поведение обобщенного клиентского кода принято называть *полиморфизмом* (много форм поведения у одного и того же кода).

Полиморфизм в языке C# (как и в языке C++) достигается за счет определения в классовой иерархии наследования *линейки виртуальных методов*:

```
class MyBase
{
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public virtual void Print( ){Console.WriteLine("X1={0}",X1);}
  protected int X1;
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public override void Print( )
  { Console.WriteLine( "X1={0} X2={0}",X1,X2); }
  private int X2;
}
```

Здесь линейка методов `Print()` (методы линейки должны иметь одинаковый набор параметров и выходных значений) помечена в своей исходной точке объявления, то есть в базовом классе `MyBase`, как виртуальная с помощью *ключевого слова virtual*. Если производный класс удовлетворен вариантом метода из базового класса, то он просто ничего не определяет с таким именем. А если для объектов производного класса желательно, чтобы вызывался иной, специфический для них вариант метода, то его нужно переопределить в производном классе с *ключевым словом override*.

В результате программа с Листинга 5.8 после показанной только что модификации классов `MyBase` и `MyDerived` демонстрирует полиморфное поведение обобщенного клиентского кода, сосредоточенного в методе `GenericFunc()` (см. рис. 5.8).

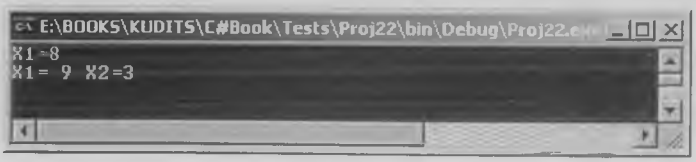


Рис. 5.8.

Из рис. 5.8 прекрасно видно, что теперь в теле метода `GenericFunc()` для объектов базового и производного классов реально вызываются разные варианты методов `Print()`.

Внимание: В отличие от языка C++ (см. [3]), где в производных классах в определениях методов из виртуальной линейки можно либо использовать ключевое слово virtual, либо вообще ничего не использовать, в языке C# требуется в тех же ситуациях в обязательном порядке применять ключевое слово override.

Истинно полиморфное поведение может демонстрировать лишь тот обобщенный клиентский код, который вызывает исключительно виртуальные методы классовой иерархии наследования. В этом случае обобщенный клиентский код становится предельно стабильным островом в океане подверженного изменениям окружающего кода, ибо его не нужно изменять совсем – даже в случае, когда классовая иерархия наследования увеличивается за счет добавления новых производных классов. Поэтому в крупных разработках стремятся выявить практическую возможность и реализовать истинно полиморфный обобщенный клиентский код.

В нашем конкретном случае программы с Листинга 5.8 и после внесенных в классы `MyBase` и `MyDerived` изменений, связанных с переходом на линейку виртуальных методов `Print()`, обобщенный клиентский код в виде метода `GenericFunc()` не становится истинно полиморфным по той причине, что в теле этого метода вызываются не только виртуальные методы. Действительно, в Листинге 5.8 методы `Set()` невиртуальные, и из них в принципе невозможно сформировать линейку виртуальных методов, ибо у них разное число параметров. В результате для всех типов объектов метод `GenericFunc()` всегда вызывает метод `Set()` базового класса.

Если мы все же захотим переделать метод `GenericFunc()` так, чтобы он вызывал для объектов разных типов разные варианты метода `Set()`, то нужно найти способ *динамически различить тип объекта во время выполнения программы*. Язык C# предоставляет такую возможность в виде *операции is*:

```
static void GenericFunc( MyBase ob, int val )
{
  if( ob is MyDerived )
   ((MyDerived)ob).Set( val, val+1 );
  else
    ob.Set( val );

  ob.Print();
}
```

В процессе выполнения метода `GenericFunc()` выражение

```
ob is MyDerived
```

вырабатывает значение `true` только, если ссылкой `ob` адресуется объект производного класса. В таком случае мы с помощью операции (`MyDerived`) приводим переменную *ob* к типу `MyDerived`

```
(MyDerived)ob
```

и, обладая полученной ссылкой на тип `MyDerived`, вызываем метод `Set()` производного класса

```
((MyDerived)ob).Set( val, val+1 );
```

который, в отличие от одноименного метода базового класса, имеет не один, а два параметра.

Подведем итоги рассмотренным вопросам о виртуальности методов, полиморфизме клиентского кода и динамическом определении типа на стадии выполнения программы, собрав для большей наглядности все разрозненные куски кода в единую демонстрационную программу:

Листинг 5.9.

```
using System;
class MyBase
{
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public virtual void Print( ){Console.WriteLine("X1={0}",X1);}
  protected int X1;
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public override void Print( )
  { Console.WriteLine( "X1={0} X2={0}",X1,X2); }
  private int X2;
}
class main
{
  public static int Main( )
  {
    MyBase[] obAr = new MyBase[2] { new MyBase(1),
                                    new MyDerived(2,3) };
    GenericFunc( obAr[0], 8 );
```

```
    GenericFunc( obAr[1], 9 );

    Console.ReadLine();
    return 0;
  }
  static void GenericFunc( MyBase ob, int val )
  {
    if( ob is MyDerived )
       ((MyDerived)ob).Set( val, val+1 );
    else
       ob.Set( val );

    ob.Print();
  }
}
```

Консольное окно работающей программы с Листинга 5.9 показано на рис. 5.9.

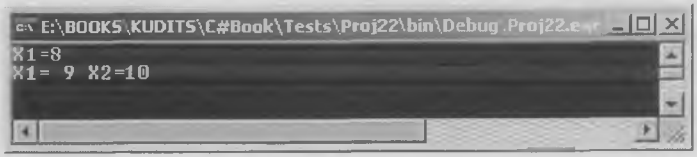


Рис. 5.9.

Из рис. 5.9 видно, что для объектов производного типа `MyDerived` в теле метода `GenericFunc()` теперь вызывается именно метод `Set()` производного класса, так что за один раз изменяются оба числовых поля этих объектов (в данном случае они принимают значения 9 и 10, соответственно).

5.4. Введение в иерархию классов библиотеки FCL. Окончательная версия класса MyComplex

Удобство программирования на *платформе Microsoft NET Framework* заключается, в значительной степени, в мощи и широте ее *библиотеки классов FCL* (Framework Class Library). Библиотека FCL не только интегрирует все встроенные типы языка C#, но и поддерживает разрабатываемые пользовательские типы, а также содержит богатейший набор классов для автоматизации программирования всех мыслимых типов программных приложений. Мы в настоящей главе еще воспользуемся услугами этой библиотеки для работы с дисковыми файлами, а в следующей главе познакомимся с программированием Windows-приложений с графическим интерфейсом пользователя.

В предыдущих разделах настоящей главы мы изучили такие фундаментальные для объектно-ориентированного программирования механизмы, как наследование классов и полиморфизм, а также динамическое определение типов на стадии выполнения программы. Все эти механизмы широчайшим образом применяются в библиотеке классов FCL, так что полученные знания пригодятся нам прямо сейчас.

Начнем с важнейшей структурной особенности классов библиотеки FCL, заключающейся в том, что *все классы этой библиотеки имеют в качестве базового класс Object*, определенный в пространстве имен System (по-другому говорят, что все классы наследуются от класса Object). Причем, это касается не только всех библиотечных классов, но и всех пользовательских классов, даже если в коде их определения механизм классового наследования (от класса Object) явным образом не применяется. Можно сказать, что все это встроено в компилятор языка C# и реализуется автоматически.

Отсюда следует, что и библиотечные классы `Array` и `String`, положенные в основу встроенных массивов и строк языка C# (см. выше гл. 3 и гл. 4), и разработанные нами в предыдущих разделах настоящей главы классы `MyBase` и `MyDerived` – все наследуют от корневого библиотечного класса `Object`, что и отображается графически в виде UML-диаграмм на рис. 5.10.

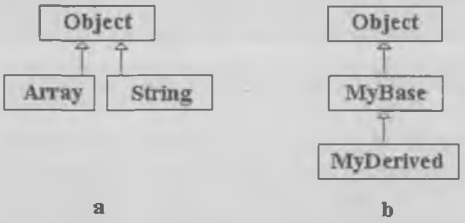


Рис. 5.10.

Убедиться в справедливости сказанного можно непосредственно с помощью графической среды компилятора Microsoft Visual C#, для чего достаточно выбрать в левой части ее главного окна закладку Class View (см. рис. 5.11).

Из рис. 5.11 видно, что в левой части окна компилятора помимо непосредственно запрограммированных нами методов `Set()` и `Print()`, а также конструктора класса `MyBase`, в этом классе присутствуют и иные методы, что может произойти только в том случае, если класс наследует от некоторого другого, базового класса. Здесь также точно поименован этот базовый класс – это класс `Object`.

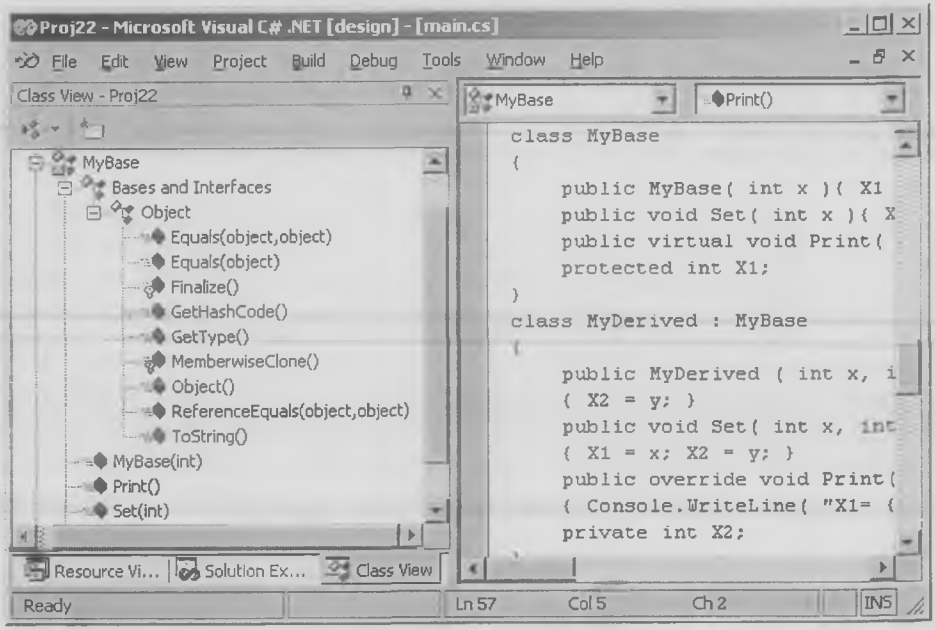


Рис. 5.11.

При всей чрезвычайной наглядности приведенного доказательства есть еще один канал сообщения информации о базовых классах, даже более оперативный, не требующий буквально никаких дополнительных усилий и действий со стороны программиста. Заключается он в способности графической среды компилятора Microsoft Visual C# давать подсказки на ходу, по мере того, как программист вводит с клавиатуры новые символы, составляющий для языка C# минимально законченные конструкции, и которые тут же пускаются в предварительную оценку с выдачей оперативных подсказок программисту в виде подчеркивающих ошибочные фрагменты линий, или в виде всплывающих списков, содержащих все имеющиеся в распоряжении программиста классовые методы, включая те, что достались от базовых классов (см. рис. 5.12).

На рис. 5.12 прекрасно видно, что как только программист ставит точку после имени объекта ob типа `MyBase`, графическая среда автоматически выводит во всплывающее окно указанный только что список методов. В этом списке мы-то и видим методы, не запрограммированные нами непосредственно в классе `MyBase`, то есть методы, доставшиеся классу `MyBase` неявно от класса `Object`.

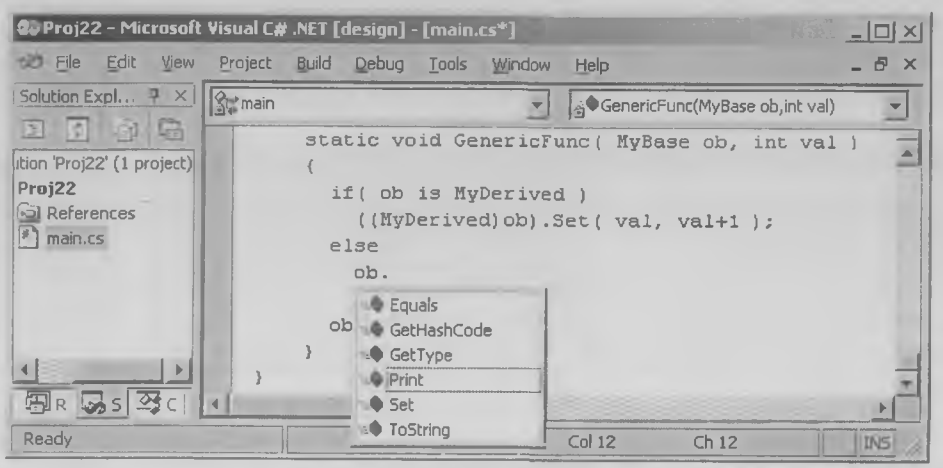


Рис. 5.12.

Получить справочные сведения по всем методам и другим деталям устройства корневого для библиотеки FCL класса `Object` можно с помощью встроенной в графическую среду компилятора Microsoft Visual C# справочной системы (команды меню Help | Contents... или Help | Index... или Help | Search...). Например, на рис. 5.13 показана справочная информация по методу `ToString()`.

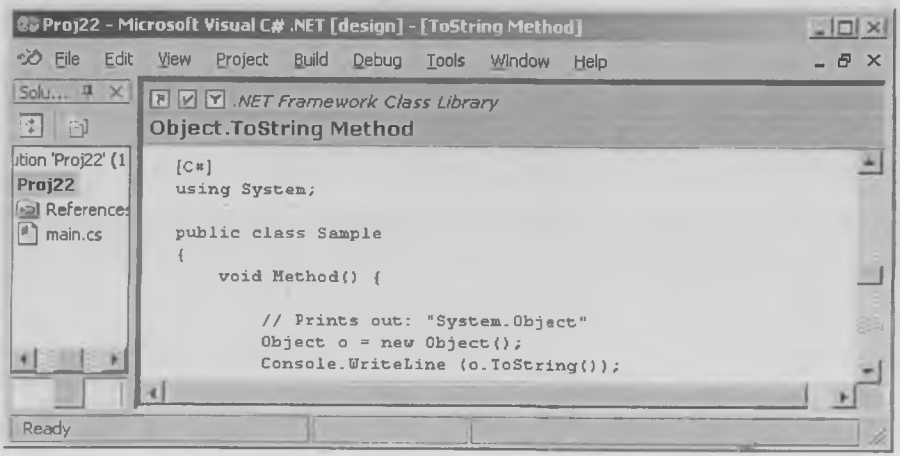


Рис. 5.13.

Из рис. 5.13 видно, что в рамках справочной информации встречается и код на языке C#, демонстрирующий примеры использования классовых методов. Например, здесь показано создание объекта типа `Object` и вызов метода `ToString()` от лица созданного объекта.

Мы не зря выбрали для иллюстраций виртуальный метод `ToString()` базового класса `Object`, так как он предназначен для представления объектов в текстовой форме, и поэтому используется методом `WriteLine()` библиотечного класса `Console`.

Сейчас мы собираемся применить метод `ToString()` для дальнейшего совершенствования разработанного нами ранее класса `MyComplex`, последний вариант которого представлен выше в Листинге 5.4. Конкретно мы хотим *заместить* (*override*) в классе `MyComplex` виртуальный метод `ToString()` так, чтобы он вырабатывал текстовую строку, содержащую информацию о вещественной и мнимой частях комплексного числа (а не вырабатывал строку с именем класса, как это делает вариант метода от базового класса `Object`).

Предлагаемый нами вариант метода `ToString()` использует в своей работе библиотечный класс `StringBuilder`, определенный в пространстве имен `System.Text`, и который мы уже изучали в разд. 4.5 из гл. 4:

```
public override string ToString( )
{
  StringBuilder sb = new StringBuilder();
  sb.AppendFormat( "Re = {0} Im = {1}\n", R, I );
  return sb.ToString();
}
```

Как только мы вводим в определение класса `MyComplex` это определение метода `ToString()`, так сразу же пропадает необходимость в отдельном методе `Print()`. В результате класс становится чище и прозрачнее, а самое главное, это упрощает и стандартизирует работу клиентского кода:

Листинг 5.10.

```
using System;
using System.Text;

class MyComplex
{
  // Constructors:
  public MyComplex( ){ R = I = 0; }
  public MyComplex( double r, double i ){ R = r; I = i; }
  public MyComplex( MyComplex z ){ R = z.R; I = z.I; }
```

```
  // Methods:
  public override string ToString( )
  {
    StringBuilder sb = new StringBuilder();
    sb.AppendFormat( "Re = {0} Im = {1}\n", R, I );
    return sb.ToString();
  }
  static public double Abs( MyComplex z )
  { return z.Length; }

  // Properties:
  public double Length
  { get { return Math.Sqrt( R*R + I*I ); } }
  public double Re { get{ return R; } set{ R = value; } }
  public double Im { get{ return I; } set{ I = value; } }

  // Overloaded operators:
  static public MyComplex operator+(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R + rOp.R, lOp.I + rOp.I ); }
  static public MyComplex operator-(MyComplex lOp,MyComplex rOp)
  { return new MyComplex( lOp.R - rOp.R, lOp.I - rOp.I ); }
  static public MyComplex operator*(MyComplex lOp,MyComplex rOp)
  {
    double Re = lOp.R * rOp.R - lOp.I * rOp.I;
    double Im = lOp.R * rOp.I + lOp.I * rOp.R;
    return new MyComplex( Re, Im );
  }

  // Private data fields:
  private double R;
  private double I;
}

class main
{
  public static int Main( )
  {
    MyComplex z = new MyComplex();

    z.Re = 1; z.Im = 2;
    Console.WriteLine("{0}", z );

    Console.ReadLine();
    return 0;
  }
}
```

Итак, после того, как мы в настоящей главе сначала ввели в определение класса `MyComplex` свойства с именами `Re` и `Im`, а теперь и заместили виртуальный метод `ToString()`, появилась возможность исключительно простой установки значений комплексных чисел и вывода этих значений на дисплей стандартным способом, то есть методом `WriteLine()` класса `Console`.

Компилируя Листинг 5.10 и запуская программу на выполнение, получаем вполне ожидаемую информацию в ее консольном окне (см. рис. 5.14).

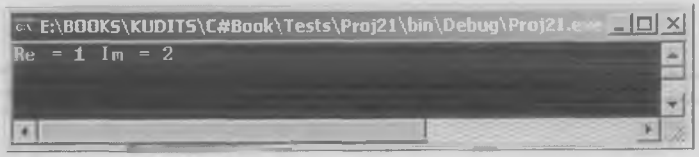


Рис. 5.14.

Класс `Object`, как корневой класс всей иерархии классов библиотеки FCL, настолько важен и вездесущ, что в языке C# для него даже выделено отдельное *ключевое слово object*, которое можно использовать для указания типа.

В связи с наследованием всех классов от одного базового класса `Object` объекты всех остальных типов можно рассматривать одновременно и как объекты типа `object`, что открывает неограниченные возможности для написания обобщенного кода для единообразного манипулирования наборами объектов разных типов. С примерами на эту тему мы познакомимся в следующей главе.

Глава 6. Перечисления. Интерфейсы. Библиотечные классы коллекций

6.1. Перечисления как типы данных

Помимо таких важнейших пользовательских типов данных как классы, в языке C# можно определять и иные варианты пользовательских типов. В настоящей главе мы рассмотрим определение и использование перечислений и интерфейсов.

Перечислением (*enumeration*) называется пользовательский тип, создаваемый с *ключевым словом enum* (вместо ключевого слова class), и содержащий в теле определения лишь список разделяемых запятыми идентификаторов, каждому из которых автоматически присваивается целое значение (по порядку, от нуля и выше).

Например, в результате следующего определения

```
enum MyBooks
{
  FirstBook,
  SecondBook,
  ThirdBook
}
```

в распоряжение программиста поступают именованные константы `MyBooks.FirstBook`, `MyBooks.SecondBook` и `MyBooks.ThirdBook`, фактически равные 0, 1 и 2. Тем не менее, присваивание этих значений целым переменным требует явного приведения типа:

```
int x = (int)SecondBook;
```

Перечисления в языке C# *неявно наследуются от библиотечного типа Enum*, включенного в общую иерархию типов с корневым классом Object.

Все сказанное про перечисления языка C# наглядно иллюстрируется следующей учебной программой:

Листинг 6.1.

```
using System;
enum MyBooks
{
  FirstBook,
  SecondBook,
  ThirdBook
```

```
}
class main
{
  public static int Main( )
  {
    MyBooks mb = MyBooks.FirstBook;
    Console.WriteLine( "{0}\n{1}", mb, (int)mb );

    mb = MyBooks.ThirdBook;
    string str = mb.ToString( );
    Console.WriteLine( "\n{0}", str );

    Console.ReadLine();
    return 0;
  }
}
```

Очевидные результаты работы этой программы показаны на рис. 6.1.

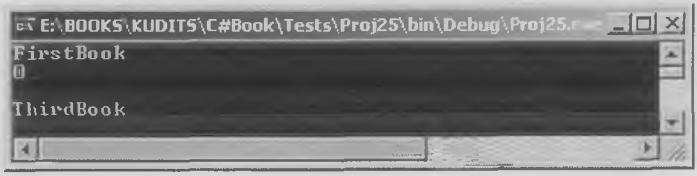


Рис. 6.1.

Можно явно назначать числовые значения элементам перечисления:

```
enum MyBooks
{
  FirstBook = 50,              // 50
  SecondBook,                  // 51
  ThirdBook = FirstBook + 50   // 100
}
```

В реальной практике перечислимые типы часто используются там, где имеется некоторый набор характерных числовых значений, в зависимости от которых нужно производить различные действия, что в свою очередь реализуется в программе с помощью *оператора-переключателя*, называемого также *оператором switch* (почти полностью аналогичен оператору switch языков C/C++, см. [2]):

```
using System;
enum MyBooks
{
  FirstBook = 50,
  SecondBook, //51
  ThirdBook = FirstBook + 50 //100
}
class main
{
  public static int Main( )
  {
    MyBooks mb = MyBooks.FirstBook;

    TellAboutBooks( mb );

    Console.ReadLine();
    return 0;
  }
  private static void TellAboutBooks( MyBooks mb )
  {
    switch( mb )
    {
      case MyBooks.FirstBook:
        Console.WriteLine( "The book is a FirstBook\n" );
        break;
      case MyBooks.SecondBook:
        Console.WriteLine( "The book is a SecondBook\n" );
        break;
      case MyBooks.ThirdBook:
        Console.WriteLine( "The book is a ThirdBook\n" );
        break;
    }
  }
}
```

В связи с оператором `switch` языка C# стоит отметить, что он не допускает пропусков ключевого слова `break` с целью провала в соседнюю case-ветвь (как это практикуется в языках C/C++), а требует применения одной из двух возможных форм *оператора goto* (*оператор безусловного перехода*).

Первый из возможных вариантов оператора `goto` в составе оператора `switch` использует ключевое слово `case`:

```
switch( mb )
{
  case MyBooks.FirstBook:
    Console.WriteLine( "The book is a FirstBook\n" );
```

```
    goto case MyBooks.SecondBook;
  case MyBooks.SecondBook:
    Console.WriteLine( "The book is a SecondBook\n" );
    break;
...
}
```

Другая допустимая в рамках оператора `switch` форма оператора `goto` использует ключевое слово `default`:

```
goto default;
```

и осуществляет переход на `default`-ветвь оператора `switch` (если она, конечно, присутствует в коде).

Для справки сообщим, что в языке C# вне оператора `switch` оператор безусловного перехода `goto` имеет вполне традиционную для языков C/C++ форму (см., например, [2]), которая использует *метку* для указания конечной точки перехода:

```
string str = "ABC";
if( str == "DEF" ) goto LABEL;
...
LABEL: str = "asdf";
...
```

В данном примере в качестве метки применен произвольно выбранный нами идентификатор `LABEL`. Напоминаем, что метка от остальной части строки кода (от оператора) должна отделяться двоеточием.

6.2. Интерфейсы: определение, реализация, применение

Выше в разд. 3.5 было показано, что встроенные массивы языка C# (то есть объекты библиотечного класса `Array`) могут сортировать свое содержимое при помощи статического метода `Sort()`. Ясно, что методу `Sort()` для выполнения сортировки (переупорядочения) нужно уметь сравнивать элементы массива на «больше-меньше». Со сравнением элементов встроенных типов проблем не возникает (о них компилятор знает все заранее), но как быть с элементами пользовательских типов? Это очень важный архитектурный вопрос, решение которого, например, позволяет сделать сортировку массивов методом `Sort()` библиотечного класса `Array` по-настоящему универсальной.

Более того, подобного рода *архитектурные вопросы* (вопросы о принципах построения и взаимодействия программных компонентов), *будучи стандартизованными*, позволяют получать *расширяемые программные решения*, пригодные

для многократного повторного использования с новыми пользовательскими типами данных, создаваемыми независимо, в любое время и любыми группами программистов (в том числе сторонними).

В индустрии программирования длительное время осуществлялся поиск наиболее приемлемых решений этого вопроса. В известном стандарте COM (Component Object Model) в основу всей архитектуры были положены так называемые интерфейсы COM, представляющие собой стандартизированные структуры данных, типичные для компиляторов языка C++ (см. [3]), и через которые разные программные компоненты могут надежно и предсказуемо взаимодействовать друг с другом. Серьезным недостатком интерфейсов стандарта COM была, однако, их изрядная сложность, связанная с необходимостью знать и вручную программировать великое множество мелких и низкоуровневых деталей.

На платформе Microsoft NET Framework в рамках языка программирования C# в основание стандарта взаимодействия программных компонентов (приложений и библиотек от сторонних производителей) положено существенно более высокоуровневое синтаксическое понятие интерфейса языка C#.

Определение *интерфейса в языке C#* внешне похоже на определение класса, но имеются следующие отличия:

- Вместо ключевого слова class используется *ключевое слово interface*.
- В интерфейсе не определяются поля данных.
- Методы (и свойства) в интерфейсе не имеют реализации (отсутствуют тела, содержащие операторы) и состоят из заголовка и точки с запятой.

Принято *имена интерфейсов начинать с заглавной буквы I*, например:

```
interface IMyInterface
{
  int Fun1( int k);
  int Fun2( );
}
```

что совершенно необязательно, но крайне желательно для достижения большей наглядности.

Довольно очевидно, что *интерфейс это не обычный тип данных* – для него, например, нельзя создавать никаких объектов, а значит, в интерфейсах не имеют смысла (и *запрещены* синтаксисом языка C#) спецификаторы режима доступа `public/protected/private`.

Внимание: Методы интерфейсов языка C# всегда *неявно открытые и виртуальные*, а явно использовать ключевые слова public и virtual запрещено.

Класс языка C# может реализовать интерфейс, для чего в определении класса после его имени нужно через двоеточие (тот же синтаксис, что и у механизма наследования классов) указать имя интерфейса (или несколько имен интерфейсов, перечисленных через запятую), а в теле определения класса представить полные определения интерфейсных методов:

```
class MyClass : IMyInterface
{
  // Реализация методов Fun1() и Fun2()
  // как открытых методов класса MyClass:
  public int Fun1( int k ){ return k; }
  public int Fun2( ){ return Get(); }

  // Собственные методы класса MyClass:
  public MyClass( ) { data = 777; }
  private int Get( ) { return ( 2 + data ); }

  // Поле данных класса MyClass:
  private int data;
}
```

В данном примере мы реализовали методы `Fun1()` и `Fun2()`, объявленные в интерфейсе `IMyInterface`, *как открытые* (с модификатором `public`) *методы класса MyClass*.

Другой возможный вариант реализации в классе интерфейсных методов будет рассмотрен ниже в Листинге 6.3, а сейчас завершим наш текущий иллюстрирующий пример, добавив в него несложный клиентский код:

Листинг 6.2.

```
using System;
interface IMyInterface
{
  int Fun1( int k);
  int Fun2( );
}
class MyClass : IMyInterface
{
  // Реализация методов Fun1() и Fun2()
  // как открытых методов класса MyClass:
  public int Fun1( int k ){ return k; }
  public int Fun2( ){ return Get(); }

  // Собственные методы класса MyClass:
  public MyClass( ) { data = 777; }
  private int Get( ) { return ( 2 + data ); }
```

```
  // Поле данных класса MyClass:
  private int data;
}
class main
{
  public static int Main( )
  {
    MyClass ob = new MyClass();

    // Вызов методов Fun1() и Fun2()
    // как методов класса MyClass:
    Console.WriteLine( "{0}", ob.Fun1( 5 ) );
    Console.WriteLine( "{0}", ob.Fun2() );

    Console.ReadLine();
    return 0;
  }
}
```

Клиентский код Листинга 6.2 наглядно показывает, что интерфейсные методы `Fun1()` и `Fun2()` действительно вызываются от лица объекта `ob` типа `MyClass` точно так же, как и обычные открытые классовые методы.

Часто бывает предпочтительным другой способ реализации интерфейсных методов в реализующем их классе: как *методов, относящихся к иному типу – к типу объявляющего их интерфейса*:

Листинг 6.3.

```
using System;
interface IMyInterface
{
  int Fun1( int k);
  int Fun2( );
}
class MyClass : IMyInterface
{
  // Реализация в классе MyClass методов Fun1()
  // и Fun2() интерфейса IMyInterface1:
  int IMyInterface.Fun1( int k ){ return k; }    /*1*/
  int IMyInterface.Fun2( ){ return Get(); }      /*2*/

  // Собственные методы класса MyClass:
  public MyClass( ) { data = 777; }
  private int Get( ) { return ( 2 + data ); }
```

```
  // Поле данных класса MyClass:
  private int data;
}
class main
{
  public static int Main( )
  {
    MyClass ob = new MyClass();

    // Вызов методов Fun1() и Fun2()
    // как методов интерфейса IMyInterface:
    Console.WriteLine( "{0}", ((IMyInterface)ob).Fun1( 5 ) );
    Console.WriteLine( "{0}", ((IMyInterface)ob).Fun2() );

    Console.ReadLine();
    return 0;
  }
}
```

Здесь строки `/*1*/` и `/*2*/` иллюстрируют следующий факт: чтобы реализация интерфейсных методов в классе относилась к типу интерфейса (а не класса), требуется *имя метода уточнить именем объявляющего их интерфейса* и *не применять никаких модификаторов режима доступа* (запрещено синтаксисом). В итоге, в рамках класса `MyClass` методы `Fun1()` и `Fun2()` по умолчанию закрытые, и использовать их в клиентском коде от имени объектов класса `MyClass` теперь уже нельзя.

Из клиентской части Листинга 6.3 видно, что *воспользоваться методами интерфейса* от имени объектов реализующего интерфейс класса можно, осуществив явное *приведение типа* (в рамках типа `IMyInterface`, как интерфейсного типа, методы `Fun1()` и `Fun2()` неявно открытые).

Более подробно, в два этапа, это выглядит следующим образом:

```
IMyInterface rInt = ob;                          /*3*/
Console.WriteLine( "{0}", rInt.Fun1( 5 ) );      /*4*/
```

где на первом этапе (в строке `/*3*/`) *получают ссылку на интерфейс*, а на втором этапе (в строке `/*4*/`) эту ссылку используют, вызывая интерфейсный метод `Fun1()`.

Если попытаться получить ссылку на интерфейс для объекта класса, не поддерживающего (не реализующего) интерфейс, то показанная выше строка кода `/*3*/` во время выполнения программы вызовет генерацию исключения со всеми вытекающими последствиями.

Более надежный способ получения ссылки на интерфейс (важен в ситуациях неопределенности с настоящим типом объектов) использует *операцию as*:

```
IMyInterface rInt = ob as IMyInterface;
if( rInt != null)
  Console.WriteLine( "{0}", rInt.Fun1( 5 ) );
```

Если объект, участвующий в операции as, относится к классу, не поддерживающему указанный справа от знака операции интерфейс, то результат операции равен null. Проверяя этот результат, можно писать надежный клиентский код.

6.3. Стандартный библиотечный интерфейс IComparable

Рассмотрев в предыдущем разделе синтаксис определения, реализации и использования интерфейсов, вернемся к вопросу о том, как статический метод Sort() библиотечного класса Array (тип встроенных массивов языка C#) мог бы работать с неизвестными для него пользовательскими типами данных, ведь он в принципе не может знать, как сравнивать объекты этих типов на «больше-меньше». Теперь мы можем дать ответ на этот вопрос – метод Sort() с указанной целью вызывает для сортируемых объектов метод CompareTo(), объявленный в стандартном библиотечном интерфейсе IComparable.

И класс Array, и интерфейс IComparable определены в рамках библиотеки FCL в пространстве имен System. Пользовательский же тип данных, создаваемый сторонними (по отношению к создателям библиотеки FCL) разработчиками, чтобы быть пригодным для использования вместе с методом Sort() библиотечного класса Array, должен просто реализовать методы определенного в библиотеке FCL интерфейса IComparable.

Естественно, что наряду с формальным определением интерфейса IComparable, разработчики библиотеки FCL предоставляют в распоряжение сторонних программистов подробное словесное описание семантики (смысла) работы единственного интерфейсного метода CompareTo() (см. рис. 6.2).

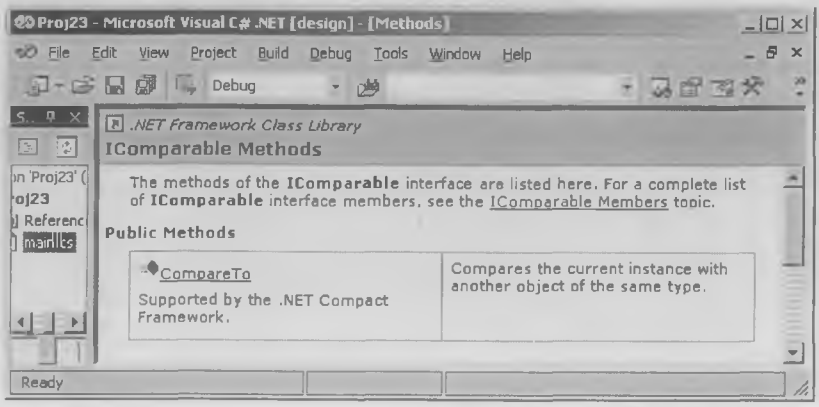


Рис. 6.2.

На рис. 6.2 показано, как в окне графической оболочки компилятора Microsoft Visual C# демонстрируется информация из встроенной системы помощи по поводу интерфейса IComparable. Перейдя в этом окне по ссылке CompareTo в другой раздел справочной системы, получаем всю необходимую нам информацию по параметрам, возвращаемому значению и семантике работы интерфейсного метода `CompareTo()`. (см. рис. 6.3).

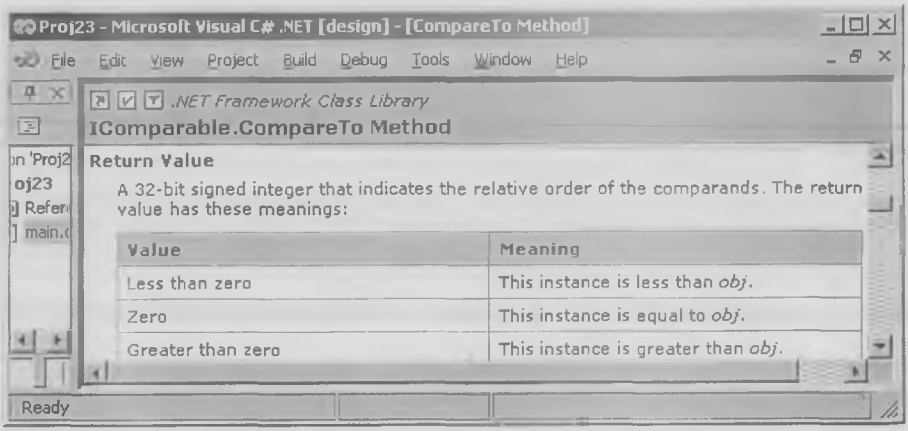


Рис. 6.3.

Метод вызывается от имени заданной объектной ссылки и получает в качестве параметра ссылку на иной, сравниваемый объект. Смысл работы метода раскрывается через информацию о его возвращаемом значении (показана на рис. 6.3): если объект больше сравниваемого, то возвращается положительное значение, если объекты равны – то нуль, и отрицательное значение, если объект меньше сравниваемого (получаемого через параметр) объекта.

Теперь мы сами можем реализовать интерфейс `IComparable`, например, в классе `MyBase` из разд. 5.3 (см. Листинг 5.9).

Листинг 6.4.

```
using System;
class MyBase : IComparable
{
  // Interface IComparable method:
  public int CompareTo( object ob )
  {
    if( X1 > ((MyBase)ob).X1 ) return 1;
```

```
    else if( X1 == ((MyBase)ob).X1 ) return 0;
    else return -1;
  }

  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public virtual void Print( ){Console.WriteLine("X1={0}",X1);}

  protected int X1;
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public override void Print( )
  { Console.WriteLine( "X1={0} X2={0}",X1,X2); }

  private int X2;
}
class main
{
  public static int Main( )
  {
    MyBase[] obAr = new MyBase[3] { new MyBase(6),
                                    new MyDerived(2,3),
                                    new MyDerived(9,1)  };

    // Сортируем массив obAr:
    Array.Sort( obAr );

    // Выводим элементы отсортированного массива:
    for( int i = 0; i < obAr.Length; ++i )
      obAr[i].Print();

    Console.ReadLine();
    return 0;
  }
}
```

Обсудим содержание Листинга 6.4. Напомним, что ключевое слово `object` в языкае C# является синонимом для имени корневого библиотечного класса `Object`, и больше про устройство интерфейсного метода `CompareTo()` по существу сказать нечего, так как все было объяснено выше. Важно, тем не менее, иметь ввиду, что в настоящем пособии мы пишем в первую очередь учебный код, который нежелательно перегружать нужными для реальной практики деталями,

и поэтому в методе CompareTo() мы сознательно не производим проверку фактического типа объекта ob на предмет, а вдруг это не объекты из иерархии классов MyBase и MyDerived, на которые мы неявно рассчитываем?

Далее обратим внимание на состав объектов, помещаемых в качестве элементов в массив obAr. В Листинге 6.4 в этот массив помещаются как объекты базового класса MyBase, реализующего интерфейс IComparable, так и объекты производного класса MyDerived, напрямую интерфейс IComparable не реализующего, но ему это все достается «задаром», по наследству от его базового класса.

Компилируем программу с Листинга 6.4, запускаем ее на выполнение и вот что мы видим в ее консольном окне (см. рис. 6.4).

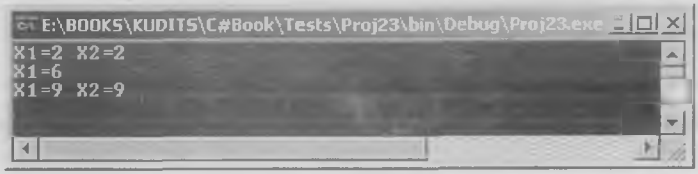


Рис. 6.4.

Из содержимого консольного окна, показанного на Рис.6.4, мы делаем два вывода. Первый вывод – сортировка массива методом Sort() библиотечного класса Array теперь действительно работает для элементов пользовательских типов MyBase и MyDerived. Второй вывод – сортировка выполняется по возрастанию значения поля X1.

Чтобы осуществить сортировку «по убыванию», переворачиваем все с ног на голову в нашей реализации интерфейсного метода CompareTo():

```
// Вариант метода для сортировки "по убыванию":
public int CompareTo( object ob )
{
  if( X1 > ((MyBase)ob).X1 ) return -1;
  else if( X1 == ((MyBase)ob).X1 ) return 0;
  else return 1;
}
```

и достигаем иного направления упорядочения элементов массива obAr (см. рис. 6.5).

Заканчивая данный раздел, отметим, что *класс может наследовать от некоторого базового класса и одновременно реализовывать несколько интерфейсов*, но в таком случае строго определен порядок перечисления имен в заголовке определения класса: после двоеточия сначала указывается имя базового класса,

Рис. 6.5.

и лишь после него – имена интерфейсов (перечисленные через запятую). Кроме того, из интерфейсов можно строить их собственные иерархии наследования.

6.4. Сравнение и клонирование классовых объектов

В предыдущем разделе мы обеспечили возможность сравнения на «больше-меньше» объектов из классовой иерархии наследования "MyBase–MyDerived" за счет реализации в базовом классе MyBase стандартного библиотечного интерфейса IComparable. Это позволяет статическому методу Sort() библиотечного класса Array выполнять сортировку элементов массива типа MyBase[].

Однако мы и сами можем спокойно применять интерфейсный метод CompareTo() для сравнения объектов типа MyBase или MyDerived на «больше-меньше», а также можем использовать этот метод для реализации перегруженных операций сравнения «>», «<», «==» и «!=» (перегрузку операций в классах C# мы рассматривали выше в разд. 2.4):

Листинг 6.5.

```
using System;
class MyBase : IComparable
{
  // Interface IComparable method:
  public int CompareTo( object ob )
  {
    if( X1 > ((MyBase)ob).X1 ) return 1;
    else if( X1 == ((MyBase)ob).X1 ) return 0;
    else return -1;
  }

  // Перегружаем операции сравнения парами:
  static public bool operator>( MyBase ob1, MyBase ob2 )
  {
    int res = ob1.CompareTo( ob2 );
```

```
    return ( res > 0 ) ? true : false;
  }
  static public bool operator<( MyBase ob1, MyBase ob2 )
  {
    int res = ob1.CompareTo( ob2 );
    return ( res < 0 ) ? true : false;
  }
  static public bool operator==( MyBase ob1, MyBase ob2 )
  {
    int res = ob1.CompareTo( ob2 );
    return ( res == 0 ) ? true : false;
  }
  static public bool operator!=( MyBase ob1, MyBase ob2 )
  {
    int res = ob1.CompareTo( ob2 );
    return ( res != 0 ) ? true : false;
  }

  // Конструктор и иные методы:
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public virtual void Print( ){Console.WriteLine("X1={0}",X1);}

  // Поле данных:
  protected int X1;
}
class MyDerived : MyBase
{
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public override void Print( )
  { Console.WriteLine( "X1={0} X2={0}",X1,X2); }

  private int X2;
}
class main
{
  public static int Main( )
  {
    MyDerived ob1 = new MyDerived(2,3);
    MyDerived ob2 = new MyDerived(2,1);
```

```
    // 1. Используем интерфейсный метод CompareTo():
    int res = ob1.CompareTo( ob2 );
    if( res > 0 )
      Console.WriteLine( "ob1 больше чем ob2" );
    else if( res == 0 )
      Console.WriteLine( "ob1 и ob2 равны" );
    else
      Console.WriteLine( "ob1 меньше чем ob2" );

    // 2. Сравниваем объекты с помощью операций:
    if( ob1 > ob2 )
      Console.WriteLine( "ob1 больше чем ob2" );
    else if( ob1 == ob2 )
      Console.WriteLine( "ob1 и ob2 равны" );
    else
      Console.WriteLine( "ob1 меньше чем ob2" );

    Console.ReadLine();
    return 0;
  }
}
```

Из Листинга 6.5 видно, что очень легко реализовать перегруженные операции сравнения, так как они опираются на реализацию интерфейсного метода `CompareTo()`.

В клиентском коде из Листинга 6.5 (то есть в методе `Main()` класса `main`) для сравнения объектов `ob1` и `ob2` класса `MyDerived` используются обе имеющиеся возможности (то есть вызов метода `CompareTo()` и применение перегруженных операций), причем, разумеется, с одинаковым итоговым результатом (см. рис. 6.6).

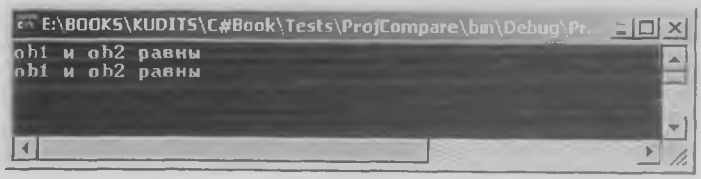


Рис. 6.6.

Если бы мы *не перегружали операций сравнения*, то все равно можно было бы использовать выражение со сравнением на равенство:

```
if( ob1 == ob2 )
```

однако, следует четко понимать, что *в этом случае сравниваются значения самих объектных ссылок* `ob1` и `ob2`, а не содержимого объектов, на которые «глядят» эти объектные ссылки.

От задачи сравнения существующих классовых объектов перейдем к задаче дублирования классовых объектов, то есть к задаче создания нового классового объекта (дубликата) с содержимым, идентичным таковому у имеющегося объекта. Подобного рода задачи в программировании принято называть *задачами копирования объектов* или *задачами клонирования объектов* (ниже мы поясним причины возникновения двух различных терминов).

Одно из потенциально возможных решений задачи дублирования объектов состоит в реализации и использовании копирующих конструкторов (см. разд. 2.3). Такое решение весьма популярно в программировании на языке C++, где параллельно с той же целью перегружается и операция присваивания. Подобного рода сдвоенное решение и называется копирующим. Однако, оно плохо подходит для программирования на языке C# по самым разным причинам, хотя бы из-за того, что перегрузка операции присваивания в языке C# запрещена (истинные причины, конечно, лежат глубже и здесь не место для их рассмотрения). В классах FCL, кстати, и копирующие конструкторы встречаются весьма редко.

Другим потенциально возможным решением задачи дублирования объектов является реализация специально для этого предназначенного классового метода с «говорящим» именем `Clone()` (отсюда и термин «клонирование»). В программировании на C# именно клонирование и является основным методом решения задачи дублирования классовых объектов.

После того, как в данной главе были рассмотрены причины и синтаксис определения и использования такого невероятно гибкого средства языка C#, как интерфейсы, не будет большим сюрпризом узнать, что клонирование в рамках библиотеки FCL реализуется через *стандартный интерфейс ICloneable*.

Библиотечный интерфейс `ICloneable` (определен в пространстве имен `System`) объявляет единственный метод с именем `Clone()`. Все остальные детали поясним на примере учебного кода:

Листинг 6.6.

```
using System;
class MyBase : ICloneable
{
  // Interface ICloneable method:
  public virtual object Clone( )
  { return new MyBase( X1 ); }
```

```
  // Конструктор и иные методы:
  public MyBase( int x ){ X1 = x; }
  public void Set( int x ){ X1 = x; }
  public virtual void Print( ){Console.WriteLine("X1={0}",X1);}

  // Поле данных:
  protected int X1;
}
class MyDerived : MyBase
{
  // Interface ICloneable method:
  public override object Clone( )
  { return new MyDerived( X1, X2 ); }

  // Конструктор и иные методы:
  public MyDerived ( int x, int y ) : base( x )
  { X2 = y; }
  public void Set( int x, int y ){ X1 = x; X2 = y; }
  public override void Print( )
  { Console.WriteLine( "X1={0} X2={0}",X1,X2); }

  // Поле данных:
  private int X2;
}
class main
{
  public static int Main( )
  {
    MyBase     obMB1 = new MyBase(1),      obMB2;
    MyDerived  obMD1 = new MyDerived(2,3), obMD2;

    // Cloning:
    obMB2 = (MyBase)( obMB1.Clone() );
    obMD2 = (MyDerived)( obMD1.Clone() );

    Console.ReadLine();
    return 0;
  }
}
```

Чтобы иметь возможность определить для базового и производного классов `MyBase` и `MyDerived` свои варианты метода `Clone()`, объявленного в интерфейсе `ICloneable`, *в базовом классе нужно определить этот метод с модификатором virtual* (нам уже известно, что интерфейсные методы неявно виртуальные – сейчас нужно вскрыть это обстоятельство явным образом), после

чего в производном классе заместить его на новый вариант при помощи модификатора `override`. Это довольно нетривиальное обстоятельство и его нужно иметь ввиду в тех случаях, когда требуется реализовывать интерфейсные методы не в одном изолированном классе, а в целой классовой иерархии наследования.

Так как методы `Clone()` в классовой иерархии наследования "`MyBase – MyDerived`" реализованы как линейка виртуальных методов, то их можно вызывать в обобщенном клиентском коде (см. разд. 5.3), но мы в Листинге 6.6 ограничились их явным применением для объектов точно специфицированного типа:

```
obMB2 = (MyBase)( obMB1.Clone() );
obMD2 = (MyDerived)( obMD1.Clone() );
```

Поскольку методы `Clone()` возвращают объекты типа `object` (синоним для корневого базового класса `Object`), то требуется явное приведение их типа к `MyBase` и `MyDerived`, соответственно (напоминаем, что приведение типа от базового класса к производному требует применения операции явного приведения, см. разд. 5.3).

В завершение раздела отметим, что клонирование, как и копирование (например, копирующими конструкторами, см. разд. 2.3 и разд. 3.4), может опираться на тривиальное копирование классовых полей (*поверхностный характер копирования* или *shallow copy*), а может и быть сколь угодно глубоким (*deep copy*) – это зависит от конкретной реализации клонирования.

6.5. Библиотечные классы коллекций

В библиотеке FCL множество классов так или иначе в своей архитектуре задействуют интерфейсы. Мы сейчас познакомимся с библиотечными *классами коллекций* (подобного рода классы в языке C++ принято называть *контейнерными классами*). Объекты классов коллекций, как и ранеее изученные нами объекты встроенных массивов языка C#, предназначены для хранения и манипулирования наборами объектов иных типов.

Самое принципиальное отличие классов коллекций от встроенных массивов языка C# состоит в том, что в момент создания массива нужно точно указать его неизменяемый в дальнейшем размер (емкость хранения), а объекты классов коллекций сами динамически управляют необходимым объемом внутреннего хранилища. Ясно, что классы коллекций предпочтительнее использовать в ситуациях неопределенности, когда нельзя заранее знать (или предвидеть) объем данных, подлежащих обработке.

В настоящем разделе мы познакомимся с классами коллекций библиотеки FCL на простейшем примере класса `ArrayList`. Определение библиотечных

классов коллекций, а также реализуемых ими стандартных интерфейсов, осуществлено в пространстве имен `System.Collections`, так что в последующих примерах мы будем использовать директиву using `System.Collections`.

Приведем пример учебного кода, иллюстрирующего основные приемы работы с объектом библиотечного класса `ArrayList`:

Листинг 6.7.

```
using System;
using System.Collections;

class main
{
  public static int Main( )
  {
    // Создаем контейнер arL типа динамического массива:
    ArrayList arL = new ArrayList();

    // Вносим в контейнер arL три числа:
    arL.Add( 3 );
    arL.Add( -5 );
    arL.Add( 8 );

    // Выводим информацию о текущем содержании контейнера:
    MyPrint( arL );

    // Манипулируем содержимым контейнера:
    // 1. Удаляем элемент со значением -5 и элемент с индексом 0.
    // 2. Изменяем значение оставшегося элемента.
    arL.RemoveAt( 0 );
    arL.Remove( -5 );
    arL[0] = 77;                                          /*1*/

    MyPrint( arL );
    Console.ReadLine();
    return 0;
  }
  static void MyPrint( ArrayList ar )
  {
    Console.WriteLine( "\nВ динамическом массиве " +
                       "находится {0} числ.", ar.Count );

    for( int i = 0; i < ar.Count; ++i )
      Console.WriteLine( "{0}", ar[i] );                  /*2*/
  }
}
```

Из Листинга 6.7 видно, что объект-контейнер `arL` типа класса коллекции `ArrayList` создается без указания его предельной емкости:

```
ArrayList arL = new ArrayList();
```

Далее, по мере необходимости мы добавляем в этот контейнер целочисленные значения методом `Add()`. С помощью выражения `arL.Count` можно узнать текущее количество хранящихся в контейнере элементов.

Традиционная для встроенных массивов операция индексации применима и к объектам-контейнерам типа `ArrayList`, причем как «по чтению» (строка `/*2*/` из Листинга 6.7), так и «по записи» (строка `/*1*/` из Листинга 6.7).

Методами `Remove()` и `RemoveAt()` элементы удаляются из контейнера, причем первому из перечисленных методов передается в качестве фактического параметра вызова значение удаляемого элемента, а второму методу – индекс элемента, подлежащего удалению.

На рис. 6.7 показаны результаты работы программы с Листинга 6.7.

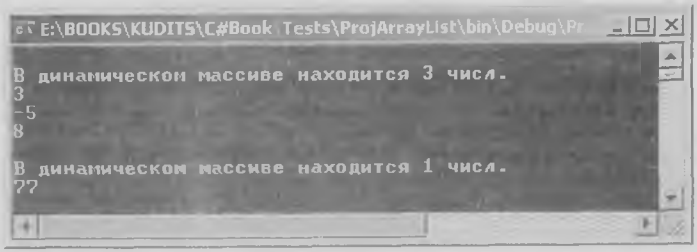


Рис. 6.7.

Содержимое консольного окна, показанное на рис. 6.7, подтверждает ожидаемое нами поведение программы с Листинга 6.7: в динамическом массиве из трех элементов (целых чисел) мы удаляем два из них, а значение последнего оставшегося элемента изменяем на 77.

Работать с индексами элементов (как встроенных массивов, так и контейнеров типа библиотечных классов коллекций) в языке C# значительно менее опасно, чем в языке C, ибо исполняющая среда CLR автоматически отслеживает их выход за граничные значения, генерируя исключения и прекращая выполнение программ. И тем не менее, лучше вообще избегать таких ситуаций, для чего в библиотечных классах коллекций дополнительно предусмотрен стандартизированный способ перебора (итерации, итерирования) всех элементов (только для чтения) при помощи *ссылки на стандартный интерфейс IEnumerator* (определен

в пространстве имен System.Collections), возвращаемой методом GetEnumerator():

```
static void MyPrint( ArrayList ar )
{
  System.Collections.IEnumerator enumer = ar.GetEnumerator();
  Console.WriteLine( "\nВ динамическом массиве " +
                     "находится {0} числ.", ar.Count );

  while( enumer.MoveNext() )
    Console.WriteLine( "{0}", enumer.Current );
}
```

Здесь в коде нового варианта метода MyPrint() показано, как от лица контейнерного объекта ar получить ссылку enumer на стандартный интерфейс IEnumerator. Эту ссылку часто называют просто *итератором*. Далее, с помощью метода MoveNext() итератор (то есть ссылка enumer) последовательно настраивается на все новые элементы контейнера, значения которых достигаются с помощью выражения enumer.Current.

Итераторный перебор элементов имеет место и для встроенных массивов языка C#, но об этом в гл. 3 мы ничего не говорили, поскольку в тот момент еще не были знакомы с концепцией и синтаксисом интерфейсов.

Как и встроенные массивы, библиотечные классы коллекций обеспечивают набор удобных в использовании методов сортировки и поиска элементов, их реверсирования и выполнения других типовых задач.

Например, ниже в иллюстративных целях мы осуществляем сортировку контейнера типа ArrayList, содержащего набор строковых значений:

Листинг 6.8.

```
using System;
using System.Collections;

class main
{
  public static int Main( )
  {
  // Создаем контейнер arL в виде объекта динамического массива:
    ArrayList arL = new ArrayList();

    // Вносим в контейнер arL несколько текстовых строк:
    arL.Add( "Москва" );
    arL.Add( "Лос-Анджелес" );
    arL.Add( "Париж" );
```

```
    // Сортируем содержимое:
    arL.Sort();

    // Выводим информацию о текущем содержании контейнера:
    for( int i = 0; i < arL.Count; ++i )
      Console.WriteLine( "{0}", arL[i] );

    Console.ReadLine();
    return 0;
  }
}
```

Компилируем и запускаем на выполнение программу с Листинга 6.8. Результат ее работы показан на рис. 6.8.

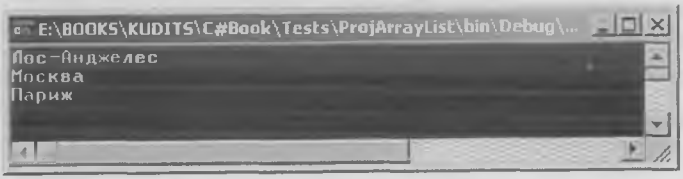


Рис. 6.8.

Из рис. 6.8 видно, что метод `Sort()` класса `ArrayList` корректно выполняет сортировку текстовых строк на русском языке.

Если мы будем хранить в контейнере типа `ArayList` объекты пользовательских типов данных, то для их корректной сортировки требуется, чтобы эти типы реализовывали стандартный интерфейс `IComparable` (см. выше разд. 6.3).

Под конец отметим, что контейнеры на базе библиотечных классов коллекций слишком «всеядные», так как их метод `Add()` принимает в качестве параметра объекты типа `object`, а мы знаем, что класс `object` является корневым базовым классов для всех остальных классов (в том числе пользовательских), и значит объекты всех классов являются по совместительству объектами типа `object`, и их можно вперемежку добавлять в контейнер:

Листинг 6.9.

```
using System;
using System.Collections;

class main
{
  public static int Main( )
```

```
  {
  // Создаем контейнер arL в виде объекта динамического массива:
    ArrayList arL = new ArrayList();

    // Вносим в контейнер arL элементы разных типов:
    arL.Add( "Москва" );
    arL.Add( 66 );
    arL.Add( 'A' );

    // Сортировка в этом случае создает проблемы:
    //arL.Sort();          /*1*/

    // Выводим информацию о текущем содержании контейнера:
    for( int i = 0; i < arL.Count; ++i )
      Console.WriteLine( "{0}", arL[i] );

    Console.ReadLine();
    return 0;
  }
}
```

В Листинге 6.9 показано, как в контейнер `arL` типа `ArrayList` методом `Add()` последовательно добавляются текстовая строка "Москва", целое число 66 и одиночный символ 'A'.

Такое разнородное содержимое контейнера корректно извлекается операцией индексации и выводится в консольное окно программы (см. рис. 6.9).

Рис. 6.9.

Однако, проблемы возникают, если мы попытаемся вызвать метод `Sort()` для разнородного содержимого контейнера `arL` – генерируется исключение и программа аварийно завершает свою работу (поэтому мы и закомментировали строку `/*1*/` в Листинге 6.9).

Чтобы ограничиться возможностью хранения в контейнере типа `ArrayList` какого-либо одного типа элементов, достаточно написать вокруг него простой *класс-обертку* (*wrapper class*), жестко задающий тип добавляемых в контейнер элементов.

Ниже представлен код собственного контейнерного класса-обертки MyStrArList, допускающего лишь строковой тип элементов:

Листинг 6.10.

```
using System;
using System.Collections;

class MyStringArList : IEnumerable
{
  // Реализация интерфейса IEnumerator:
  public IEnumerator GetEnumerator( )
  { return arList.GetEnumerator(); }

  // Конструктор и методы класса MyStringArList:
  public MyStringArList( ) { arList = new ArrayList(); }
  public void Add( string str ) { arList.Add( str ); }
  public void Remove( int i ) { arList.Remove( i ); }
  public void Sort( ){ arList.Sort(); }

  // Свойство Count (только для чтения):
  public int Count
  { get { return arList.Count; } }

  // Закрытое поле данных - контейнер типа ArrayList:
  private ArrayList arList;
}
class main
{
  public static int Main( )
  {
    // Создаем контейнер strArL "только для строк":
    MyStringArList strArL = new MyStringArList();

    // Вносим в контейнер strArL несколько текстовых строк:
    strArL.Add( "Москва" );
    strArL.Add( "Los Angeles" );
    //strArL.Add( 'A' );                    /*1*/
    MyIteratePrint( strArL );

    // Сортируем строковое содержимое контейнера strArL:
    strArL.Sort();
    MyIteratePrint( strArL );

    Console.ReadLine();
    return 0;
  }
```

```
  static void MyIteratePrint( MyStringArList ar )
  {
    System.Collections.IEnumerator enumer = ar.GetEnumerator();
    Console.WriteLine( "\nВ динамическом массиве " +
                       "находится {0} стр.", ar.Count );

    while( enumer.MoveNext() )
      Console.WriteLine( "{0}", enumer.Current );
  }
}
```

Так как в классе `MyStringArList` метод `Add()` принимает параметр типа `string`, а не `object`, то теперь добавить в контейнер что-либо кроме строк невозможно – компилятор выдаст ошибку (поэтому мы и закомментировали строку `/*1*/` в Листинге 6.10).

Консольное окно программы с Листинга 6.10 показано на рис. 6.10.

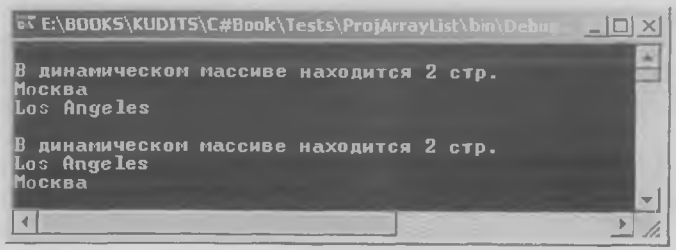


Рис. 6.10.

Класс-обертка `MyStringArList` агрегирует библиотечный класс `ArrayList` и использует его функциональность, и рис. 6.10 демонстрирует использование агрегирующим классом такой полезной функциональности класса `ArrayList`, как сортировка. Вывод содержимого в консольное окно опирается на итератор, предоставляемый опять-таки агрегируемым классом.

В настоящем разделе мы познакомились с классами коллекций из библиотеки FCL на примере класса `ArrayList`. Полный обзор разных типов классов коллекций дан в учебнике [6], кроме того, в вашем распоряжении всегда находится встроенная справочная система среды разработки Microsoft Visual C#.

Глава 7. Дисковые файлы и информационные системы

7.1. Понятие о файлах. Открытие/закрытие файлов. Запись/чтение файлов

Как известно, для долговременного хранения данных в компьютерах используются магнитные диски. Данные на дисках хранятся в виде файлов. *Файл* представляет собой единицу хранения данных на магнитных дисках компьютера, имеющий уникальное имя в пределах дискового каталога.

Специальная подсистема операционной системы Windows, так называемая *файловая система*, обеспечивает удобную работу с файлами, когда программисту не нужно знать подробных низкоуровневых деталей об устройстве и расположении записей на дисках. Нужно лишь знать имя файла и, иногда, имя каталога, в котором файл находится.

В нашем пособии по языку C#, как и ранее в книге [1], посвященной началам программирования на языке C, мы будем хранить файлы данных либо в том же самом каталоге, в котором находится исполняемый файл нашей программы, либо в главном каталоге проекта. В этих случаях при доступе к файлу не нужно явно указывать содержащего файл каталога, что значительно упрощает и укорачивает необходимые записи, поскольку достаточно указания одного лишь имени файла.

Как нам известно из книги [1], для работы с файлами на языке C нужно использовать функции (из стандартной библиотеки языка C) так называемого *потокового файлового ввода/вывода*. При этом программная работа с файлами начинается с их *открытия* библиотечной функцией `fopen()`:

```
#include <stdio.h>
...
FILE* pf = fopen( "FileName", "rb" );
```

В программах же на языке C# для работы с файлами нужно использовать соответствующие классы библиотеки FCL, закрывающие от программиста детали вызова служебных функций операционной системы, предназначенных для работы с файлами. Для того, чтобы начать работать с файлом, нужно *открыть файловый поток*, создав для этого *объект библиотечного класса FileStream* (определен в пространстве имен System.IO):

```
using System.IO;
...
FileStream fs = new FileStream( "FileName", FileMode.Open,
                                FileAccess.Read );
```

При открытии файловых потоков (как в программах на языке C, так и в программах на языке C#) файловая система (часть операционной системы) выполняет поиск на диске всей необходимой служебной информации, достаточной для того, чтобы при всех последующих операциях чтения данных из файла и записи данных в файл не рыскать по всему диску заново. Вся служебная информация о файле, собранная во время работы функции `fopen()` или во время работы конструктора класса `FileStream`, сохраняется, соответственно, в полях автоматически создаваемой переменной, имеющей тип структуры `FILE` (при программировании на C), и в объекте класса `FileStream` (при программировании на C#).

В обоих случаях строка "FileName" задает *имя файла* (как мы договорились – без учета каталогов). Дополнительная информация содержит уточняющие детали процесса открытия файлового потока. Например, для функции fopen() строка "`rb`" содержит символ `r`, означающий, что файл открывается *на чтение* (`r` – `read`, читать), и символ `b`, означающий *двоичный* (`b` – `binary`, двоичный или бинарный) *режим обмена данными между диском и потоковым буфером в памяти компьютера.* Для конструктора класса `FileStream` параметры `FileMode.Open` и `FileAccess.Read` (это элементы стандартных перечислений `FileMode` и `FileAccess`, определенных в пространстве имен `System.IO`) означают режим открытия реально существующего на диске файла «только для чтения», а двоичный или текстовый режим обмена данными между буфером и файлом на диске в момент открытия потокового объекта типа `FileStream` не фиксируется.

После успешного открытия файла с ним можно выполнять операции чтения/записи данных. По завершении работы с файлом его нужно *закрыть* либо с помощью функции `fclose()` из стандартной библиотеки языка C:

```
fclose( pf );
```

либо методом `Close()` класса `FileStream`:

```
fs.Close();
```

В общем, ясно, что работа с файлами в языках C и C# имеет много общего, хотя конкретные детали различаются. Далее мы будем говорить лишь о языке C#.

Успешность открытия файла в общем случае не гарантируется, и мы далее обсудим эту тему более подробно. Сейчас же сообщим, что при неудаче в по-

пытке открытия файла средой CLR генерируется исключение и программа принудительно завершает работу. Отсюда вытекает правило – *проверять факт существования файла* перед созданием объекта типа `FileStream` с помощью *статического метода Exists() библиотечного класса File*:

```
if( File.Exists( "FileName" ) )
{
...// здесь и создаем объект типа FileStream
}
```

В целом рассмотрев общую картину работы с файлами, перейдем к уточнению отдельных деталей. В первую очередь подробнее рассмотрим вопрос о *принципиальной негарантированности открытия файла*. Причин тут существует несколько, но самая простая состоит в том, что файла с указанным именем может и не быть. Тогда, если мы такой файл попытаемся открыть на чтение, то результат будет отрицательным (в смысле возврата логического значения `false` методом `Exists()` класса `File`).

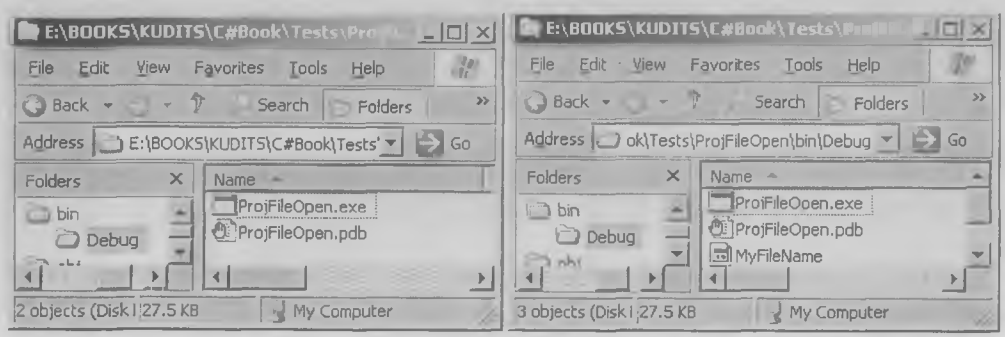
В то же время, попытка *открыть файл на запись* (в режимах `FileMode.Create` и `FileAccess.Write`) *всегда будет успешной*: при существующем файле его *прежнее содержимое аннулируется*, а при несуществующем файле он автоматически создается:

Листинг 7.1.

```
using System.IO;
class main
{
  public static int Main( )
  {
    FileStream fs = new FileStream( "MyFileName", FileMode.Create,
                                    FileAccess.Write );
    fs.Close();
    return 0;
  }
}
```

В Листинге 7.1 мы *открываем на запись* файл с именем `MyFileName`, которого ранее не существовало.

Компилируем программу с Листинга 7.1 и запускаем ее на выполнение, например, из графической среды компилятора Microsoft Visual C# клавишей F5 (см. выше разд. 1.2). В результате, «файловая картина» на жестком диске компьютера до запуска и после запуска нашей программы будет такой, как показано на рис. 7.1.



a b

Рис. 7.1.

На рис. 7.1 (a) показано содержимое подкаталога Debug нашего проекта до запуска программы, а на рис. 7.1 (b) – после запуска. Видно, что в результате работы программы был создан ранее не существовавший файл MyFileName, причем именно в каталоге, непосредственно содержащем исполняемый файл программы ProjFileOpen.exe.

Это является общим правилом: *если каталожный путь к файлу не указан, то файл данных ищется или создается в том каталоге, где находится исполняемый файл программы*.

В следующем разделе главы мы подробно рассмотрим запись в файлы целых и дробных чисел (типы данных `int` и `double`), а в разд. 7.3 – запись и чтение строковых данных. В настоящем же разделе мы лишь кратко рассмотрим основы процессов записи и чтения из файлов на примере единственного типа данных, с которым работает сам класс `FileStream` – типа `byte`.

Это самый короткий (в смысле отводимой памяти) целочисленный тип данных, значения переменных которого лежат в диапазоне от 0 до 255, и он соответствует одному байту памяти (то есть 8 битам). Мы нигде в нашем пособии с этим типом не работали и не будем работать. Просто сейчас он нам послужит в иллюстративных целях. Дело в том, что значения переменных этого типа являются минимальной порцией записываемой в файл информации. На магнитных дисках все измеряется в байтах (например, один сектор соответствует 512 байтам).

Для *записи в файл потока байтов* (набора последовательных байтовых значений) класс `FileStream` предоставляет *метод WriteByte()*:

Листинг 7.2.

```
using System.IO;
class main
{
  public static int Main( )
  {
    FileStream fs = new FileStream( "MyFileName", FileMode.Create,
                                    FileAccess.Write );

    for( int i = 0; i < 256; ++i )
      fs.WriteByte( (byte)i );

    fs.Close();
    return 0;
  }
}
```

Программа с Листинга 7.2 столь проста, что в ней стоит дополнительно обратить внимание лишь на необходимость явного приведения типа переменной `i` от типа `int` к типу `byte` (иначе компилятор выдает ошибку компиляции).

Компилируем программу с Листинга 7.2, запускаем ее на выполнение и получаем в каталоге Debug файл MyFileName (см. рис. 7.1, b), содержимое которого мы сейчас рассмотрим с помощью графической среды компилятора Microsoft Visual C#.

Для этого из среды нужно этот файл открыть в специальном режиме – на кнопке Open диалогового кона Open File нужно с помощью стрелки выбрать команду Open with..., так что появится диалоговое окно Open With (см. рис. 7.2).

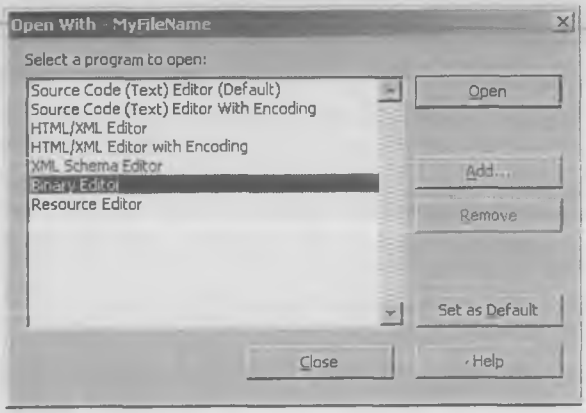


Рис. 7.2.

Как показано на рис. 7.2, в окне Open With нужно выбрать (отметить) строку Binary Editor и нажать кнопку Open. В результате в правой части главного окна компилятора Microsoft Visual C# мы наблюдаем числовое содержимое файла MyFileName (см. рис. 7.3).

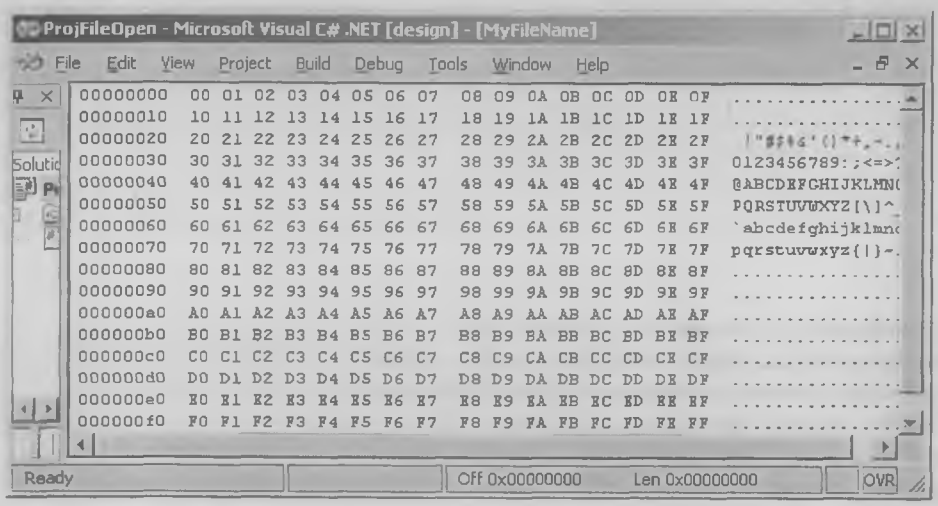


Рис. 7.3.

Из рис. 7.3 прекрасно видно, что в этот файл последовательно, байт за байтом, записаны возрастающие на единицу целые числа от 0 (00 в шестнадцатеричной записи) до FF (это шестнадцатеричная форма представления десятичного числа 255).

Но посмотреть содержимое файла MyFileName, сформированное программой с Листинга 7.2, можно и по-другому, написав собственную программу для чтения файла:

Листинг 7.3.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    if( File.Exists( "MyFileName" ) )
```

```
    {
      FileStream fs = new FileStream( "MyFileName", FileMode.Open,
                                      FileAccess.Read );

      for( int i = 0; i < 256; ++i )
        Console.WriteLine( "{0} byte in the file is {1}",
                           i, fs.ReadByte() );

      fs.Close();
    }
    else
      Console.WriteLine( "No file in the disk" );

    Console.ReadLine();
    return 0;
  }
}
```

Из рассмотренного выше про открытие файлов ясно, что здесь нам желательно заключить всю фактическую работу с файлом MyFileName в условный оператор, проверяющий с помощью статического метода `Exists()` класса `File` существование указанного файла данных.

Последовательное побайтное вычитывание файла MyFileName выполняется в программе с Листинга 7.3 с помощью метода `ReadByte()` класса `FileStream`.

Убедиться в успешности работы программы с Листинга 7.3 можно, взглянув на содержимое ее консольного окна (см. рис. 7.4).

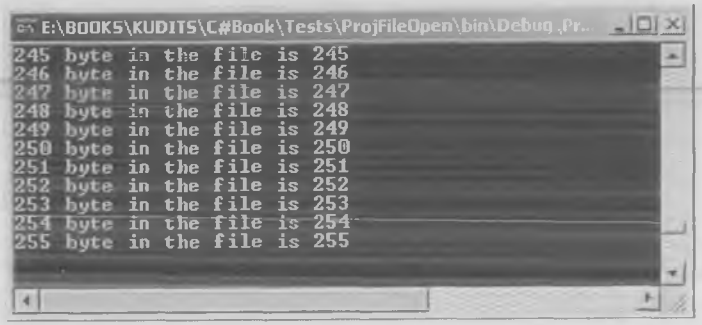


Рис. 7.4.

В отличие от рис. 7.3 на рис. 7.4 байтовое содержимое файла MyFileName демонстрируется в десятичном виде.

Можно запись в файл и последующее чтение из него совместить в рамках одной и той же программы, причем речь не идет о механическом объединении в рамках одной программы кода с Листингов 7.2 и 7.3, когда файл открывается и закрывается дважды. Вместо этого файл можно открыть и на запись, и на чтение одновременно, используя режимы `FileMode.OpenOrCreate` и `FileAccess.ReadWrite`:

Листинг 7.4.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    // Создаем файловый поток и на запись, и на чтение:
    FileStream fs = new FileStream( "MyFileName",
                                    FileMode.OpenOrCreate,
                                    FileAccess.ReadWrite );

    // Пишем данные (байты) в файл:
    for( int i = 0; i < 256; ++i )
      fs.WriteByte( (byte)i );

    /*-----------------------------------------*/
    // Устанавливаем внутренний файловый указатель
    // на начало файла:
    fs.Position = 0;
    /*-----------------------------------------*/

    // Последовательно вычитываем 256 байт из файла:
    for( int i = 0; i < 256; ++i )
      Console.WriteLine( "{0} byte in the file is {1}",
                         i, fs.ReadByte() );

    // Единожды закрываем файл:
    fs.Close();

    Console.ReadLine();
    return 0;
  }
}
```

При открытии файла в режимах `FileMode.OpenOrCreate` и `FileAccess.ReadWrite` *внутренний файловый указатель* (указатель текущей позиции) устанавливается в нуль, после чего по мере записи последовательных бай-

тов передвигается (каждый раз ровно на один байт) в направлении от начала к концу файла.

После выполнения всех операций записи в файл, нужно вернуть внутренний файловый указатель в его начальное состояние (чтобы он указывал на самый первый байт в файле):

```
fs.Position = 0;
```

что достигается установкой нулевого значения свойства `Position` для объекта типа `FileStream`. После этого можно вычитывать файл с самого начала.

При вычитывании файла мы опираемся на наши априорные знания о количестве записанных в файл байтовых значений. Можно, однако, действовать и более гибко, не опираясь на эти априорные сведения (которые в общем случае могут и отсутствовать):

```
int k = 0;
while( fs.Position != fs.Length )
{
  Console.WriteLine( "{0} byte in the file is {1}",
                     k, fs.ReadByte() );
  ++k;
}
```

Свойство `Length` соответствует длине (размеру) файла. Тогда при каждом новом вычитывании байта текущий внутренний файловый указатель передвигается вперед, пока не сравняется со значением `fs.Length`. Это и будет сигналом о полном вычитывании файла.

7.2. Файловое хранение числовых данных

Методы `WriteByte()` и `ReadByte()` класса `FileStream`, рассмотренные в предыдущем разделе, плохо приспособлены для записи/чтения числовых данных типа `int` или `double`. Поэтому для работы с указанными типами лучше создать объекты классов `BinaryWriter` и `BinaryReader`, чьи методы `Write()` (перегружен для разных типов данных), `ReadInt32()` и `ReadDouble()` как нельзя лучше подходят для этой задачи.

Проиллюстрируем работу методов `Write()`, `ReadInt32()` (для чтения целых чисел типа `int`) и `ReadDouble()` (для чтения дробных чисел типа `double`) на примере следующей учебной программы:

Листинг 7.5.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    int m = 5; double d = 8.88;
    int iRes = 0;  double dRes = 0;

    /*-------------  file writing  -------------------*/
    FileStream fsW = new FileStream( "MyFileName",
                                     FileMode.Create,
                                     FileAccess.Write );
    BinaryWriter bw = new BinaryWriter( fsW );
    bw.Write( m );
    bw.Write( d );
    bw.Close();
    fsW.Close();

    /*-------------  file reading  -------------------*/
    if( File.Exists( "MyFileName" ) )
    {
      FileStream fsR = new FileStream( "MyFileName",
                                       FileMode.Open,
                                       FileAccess.Read );
      BinaryReader br = new BinaryReader( fsR );
      iRes = br.ReadInt32();
      dRes = br.ReadDouble();
      br.Close();
      fsR.Close();
    }

    Console.WriteLine( "iRes is {0}; dRes is {1}", iRes, dRes );
    Console.ReadLine();
    return 0;
  }
}
```

В Листинге 7.5 файл с именем MyFileName сначала открывается на запись, в результате чего объектная ссылка `fsW` типа `FileStream` получает актуальное значение. Поверх этой ссылки строится объект `bw` класса `BinaryWriter`:

```
BinaryWriter bw = new BinaryWriter( fsW );
```

С помощью перегруженного метода `Write()` класса `BinaryWriter` в открытый файл последовательно записываются значения переменных `m` и `d`. Затем пишущий объект и файл закрываются одноименными методами `Close()`.

После этого файл снова открывается, но уже на чтение. Содержимое файла MyFileName (мы сами записывали в этот файл числовые данные, так что нам доподлинно известно, как он устроен) «вычитывается» от имени объекта `br` типа `BinaryReader` методами `ReadInt32()` и `ReadDouble()` в переменные `iRes` и `dRes`, соответственно:

```
iRes = br.ReadInt32();
dRes = br.ReadDouble();
```

Наконец, для проверки результата, значения переменных `iRes` и `dRes` выводятся в консольное окно программы с помощью статического метода `WriteLine()` библиотечного класса `Console` (см. рис. 7.5).

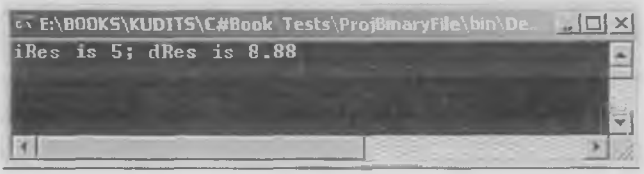


Рис. 7.5.

Из рис. 7.5 видно, что нам удалось успешно прочитать из файла MyFileName именно те числовые значения, которые мы записали в него на предыдущем шаге нашей программы.

Теперь повторим продемонстрированную Листингом 7.5 работу, но уже на примере не одиночных переменных, а массивов:

Листинг 7.6.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    int[] iArr = {5,6,7}; double[] dArr = {8.88,9.9,1};
    int[] iArrRes = new int[3];
    double[] dArrRes = new double[3];
    int i;
```

```
    /*-------------  file writing  -------------------*/
    FileStream fsW = new FileStream( "MyFileName",
                                     FileMode.Create,
                                     FileAccess.Write );
    BinaryWriter bw = new BinaryWriter( fsW );

    for( i = 0; i < iArr.Length; ++i )
      bw.Write( iArr[i] );
    for( i = 0; i < dArr.Length; ++i )
      bw.Write( dArr[i] );

    bw.Close();
    fsW.Close();

    /*-------------  file reading  -------------------*/
    if( File.Exists( "MyFileName" ) )
    {
      FileStream fsR = new FileStream( "MyFileName",
                                       FileMode.Open,
                                       FileAccess.Read );
      BinaryReader br = new BinaryReader( fsR );

      for( i = 0; i < iArrRes.Length; ++i )
        iArrRes[i] = br.ReadInt32();
      for( i = 0; i < dArrRes.Length; ++i )
        dArrRes[i] = br.ReadDouble();

      br.Close();
      fsR.Close();
    }

    for( i = 0; i < iArrRes.Length; ++i )
      Console.WriteLine( "iArrRes[{0}] = {1}; " +
                         "dArrRes[{0}] = {2}",
                         i, iArrRes[i], dArrRes[i] );

    Console.ReadLine();
    return 0;
  }
}
```

В Листинге 7.6 числовые массивы `iArr` и `dArr`, соответственно:

```
int[] iArr = {5,6,7}; double[] dArr = {8.88,9.9,1};
```

циклически поэлементно записываются в файл MyFileName:

```
for( i = 0; i < iArr.Length; ++i )
  bw.Write( iArr[i] );
for( i = 0; i < dArr.Length; ++i )
  bw.Write( dArr[i] );
```

После закрытия файла и его повторного открытия для чтения содержимое файла MyFileName последовательно вычитываем в массивы-приемники `iArrRes` и `dArrRes`:

```
for( i = 0; i < iArrRes.Length; ++i )
  iArrRes[i] = br.ReadInt32();
for( i = 0; i < dArrRes.Length; ++i )
  dArrRes[i] = br.ReadDouble();
```

Следует уделить внимание такому вопросу, как *емкость «принимающих» массивов* `iArrRes` и `dArrRes`: в Листинге 7.6 размеры массивов `iArr` и `dArr`, элементы которых записываются в файл MyFileName, и размеры массивов `iArrRes` и `dArrRes`, в которые содержимое файла MyFileName вычитывается, одинаковые. Если бы мы определили массивы-приемники с размером, большим размера массивов-источников, то код Листинга 7.6 вызвал бы генерацию исключения (с диагнозом – попытка чтения за концом файла), а если с меньшим размером, то произошла бы путаница в элементах массива целых и массива дробных чисел.

Результаты вычитывания файла MyFileName программой с Листинга 7.6 показаны на рис. 7.6.

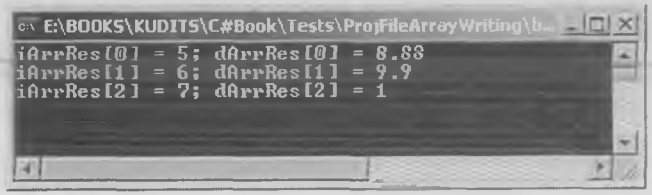


Рис. 7.6.

Из рис. 7.6 прекрасно видно, что прочитаны те же самые числа, что были туда перед этим записаны.

В Листингах 7.5 и 7.6 при вычитывании содержимого файлов мы опирались на точное знание количества записанных туда ранее чисел. Бывают, однако, ситуации, когда число записанных в файл чисел неизвестно. Если при этом требуется прочитывать все содержимое файла, то как нужно поступать в таких случаях?

Ответ на этот вопрос в случае записи в файл потока байт был дан в предыдущем разделе главы, а сейчас проиллюстрируем решение этой задачи для случая записи в файл числовых данных типа `int` или `double`:

Листинг 7.7.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    int[] iArr = {5,6,7};
    double[] iArrRes = new double[8];
    int i;

    /*-------------  file writing  -------------------*/
    FileStream fsW = new FileStream( "MyFileName",
                                     FileMode.Create,
                                     FileAccess.Write );
    BinaryWriter bw = new BinaryWriter( fsW );
    for( i = 0; i < iArr.Length; ++i )
      bw.Write( iArr[i] );

    bw.Close();
    fsW.Close();

    /*-------------  file reading  -------------------*/
    if( File.Exists( "MyFileName" ) )
    {
      FileStream fsR = new FileStream( "MyFileName",
                                       FileMode.Open,
                                       FileAccess.Read );
      BinaryReader br = new BinaryReader( fsR );

      i = 0;
      while( br.PeekChar() != -1 && i < 8 )
      {
        iArrRes[i] = br.ReadInt32();
        i++;
      }

      br.Close();
      fsR.Close();
    }
```

```
    for( i = 0; i < iArrRes.Length; ++i )
      Console.WriteLine( "iArrRes[{0}] = {1}",
                         i, iArrRes[i] );

    Console.ReadLine();
    return 0;
  }
}
```

В данной программе мы сначала записываем в файл MyFileName содержимое массива `iArr` из трех элементов типа `int`. Затем файл закрывается и снова открывается на чтение.

Вот тут-то мы и делаем вид, что не знаем, сколько целых чисел было в файл записано (чисто учебная постановка вопроса). Поэтому, прежде чем читать из файла методом `ReadInt32()` очередное целое значение, контролируем наличие невычитанных из файла данных методом `PeekChar()`:

```
while( br.PeekChar() != -1 && i < 8 )
{
  iArrRes[i] = br.ReadInt32();
  i++;
}
```

Пока метод `PeekChar()` возвращает не равные -1 значения, в файле MyFileName еще имеются невычитанные данные, которые мы и продолжаем циклически вычитывать методом `ReadInt32()`.

Как только при очередной попытке чтения метод `PeekChar()` вернет -1, то условие цикла станет ложным и он перестанет выполняться (другое условие – `i < 8`, дополнительно страхует от переполнения принимающего буфера `iArrRes`).

Результат работы программы с Листинга 7.7 показан на рис. 7.7.

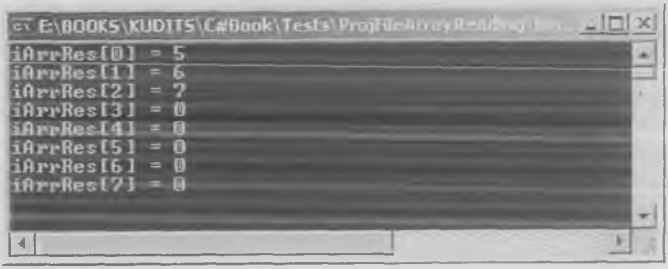


Рис. 7.7.

Из рис. 7.7 видно, что в принимающий массив iArrRes, размер которого достаточен для приема восьми целых чисел, из файла MyFileName было прочитано три целых значения.

Таким образом, разработанный алгоритм для полного вычитывания числовых файлов,основанный на применении метода PeekChar ( ),работает корректно.

7.3. Файловое хранение текстовых данных

Текстовые файлы призваны хранить содержимое текста на естественном языке, и для них сужается (по сравнению с нетекстовыми файлами, называемыми двоичными или бинарными) диапазон значений байтов (так как они кодируют ограниченный набор символов), но привносятся такие нюансы, как, например, необходимость пометить служебными кодами разбивку текста на *физические строки* (как мы их видим на дисплее или на бумаге). Поэтому с текстовыми файлами лучше работать с помощью пишуших и читающих объектов специализированных классов StreamWriter и StreamReader (вместо ранее использованных нами BinaryWriter и BinaryReader).

Работа с текстовыми файлами достаточно проста и наглядна. Продемонстрируем на учебном примере использование методов Write() и ReadLine() классов StreamWriter и StreamReader:

Листинг 7.8.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    string str1 = "Hello\n", str2 = ", world!\n";
    string Str1, Str2;

    /*-------------  file writing  --------------------*/
    FileStream fsW = new FileStream( "MyFileName",
                                     FileMode.Create,
                                     FileAccess.Write );
    StreamWriter sw = new StreamWriter( fsW );
    sw.Write( str1 );
    sw.Write( str2 );
    sw.Close();
    fsW.Close();
```

```
    /*-------------  file reading  -------------------*/
    FileStream fsR = new FileStream( "MyFileName",
                                     FileMode.Open,
                                     FileAccess.Read );
    StreamReader sr = new StreamReader( fsR );
    Str1 = sr.ReadLine();
    Str2 = sr.ReadLine();
    sr.Close();
    fsR.Close();

    Console.WriteLine( "Str1 = {0}\nStr2 = {1}\n",
                       Str1, Str2 );
    Console.ReadLine();
    return 0;
  }
}
```

В целом содержание Листинга 7.8 должно быть для нас понятно, ибо оно четко перекликается с материалом предыдущих разделов, посвященных бинарным файлам. Однако следует подчеркнуть что важным является включение в содержимое строковых переменных (это строки языка C#) `str1` и `str2` управляющих символов перевода строки `\n` (см. разд. 4.1):

```
string str1 = "Hello\n", str2 = ", world!\n";
```

Именно благодаря этим символам нам удается в процессе работы программы с Листинга 7.8 вычитать из файла две отдельные строки языка C# – Str1 и Str2 (см. рис. 7.8).

Рис. 7.8.

Если бы мы забыли про управляющие символы `\n` и сформировали бы содержимое исходных строковых переменных `str1` и `str2` в виде

```
string str1 = "Hello", str2 = ", world!";
```

то при попытке прочитать две строки из файла MyFileName

```
Str1 = sr.ReadLine();
Str2 = sr.ReadLine();
```

мы бы получили следующие результаты для принимающих строк `Str1` и `Str2` (см. рис. 7.9).

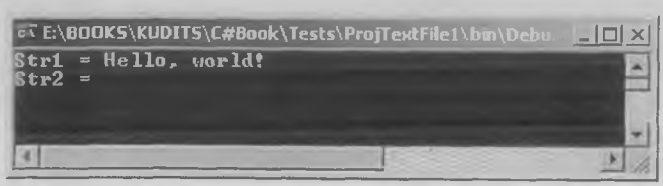


Рис. 7.9.

Объясняется такой эффект тем, что для метода `ReadLine()` именно служебный символ `\n` является разделителем записанных в файл строк. Если эта функция не встречает при чтении файла символа `\n`, то она вычитывает весь файл до конца.

Знание этого простого правила позволит нам впоследствии работать с открытыми в текстовом режиме файлами легко и безошибочно. Например, очень легко вычитывать из таких файлов все его строковое содержимое, не зная при этом заранее общего количества записанных в файл строк.

Действительно, запишем с помощью простого текстового редактора "Notepad" ("Блокнот" в русских версиях операционной системы Windows) несколько физических текстовых строк в файл MyTextFile (см. рис. 7.10).

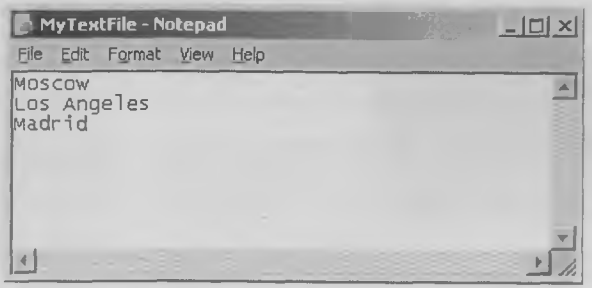


Рис. 7.10.

А теперь прочтем все содержимое сформированного таким образом файла MyFileName с помощью следующей программы:

Листинг 7.9.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    string Str;

    /*-------------  file reading  -------------------*/
    if( File.Exists( "MyTextFile" ) )
    {
      FileStream fsR = new FileStream( "MyTextFile",
                                       FileMode.Open,
                                       FileAccess.Read );
      StreamReader sr = new StreamReader( fsR );

      while( ( Str = sr.ReadLine() ) != null )
        Console.WriteLine( "Str = {0}", Str );

      sr.Close();
      fsR.Close();
    }

    Console.ReadLine();
    return 0;
  }
}
```

Компилируем программу с Листинга 7.9, запускаем ее на выполнение, и на дисплее возникает «картина», показанная на рис. 7.11.

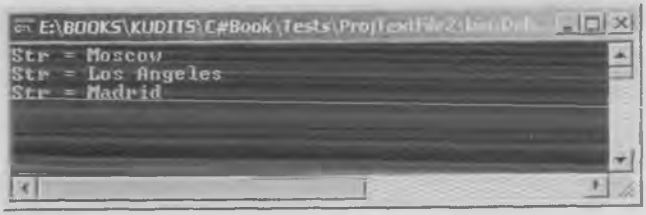


Рис. 7.11.

Сравнивая рис. 7.11 с рис. 7.10, убеждаемся, что программа с Листинга 7.9 с помощью метода `ReadLine()` читает из созданного редактором Notepad (или Проводник) файла MyTextFile по одной физической строке за один вызов этого метода.

Опираясь на *условие полной вычитки текстового файла – возврат методом ReadLine() значения null*,

```
while( ( Str = sr.ReadLine() ) != null )
  Console.WriteLine( "Str = {0}", Str );
```

нам действительно удается прочесть все три строки, ранее размещенные в файле MyTextFile.

Под конец снова вернемся к строковому разграничителю `\n`. По историческим причинам проблема строковых разграничителей не совсем тривиальна (об этом можно прочитать в книге [2]), так что, например, текстовый редактор Notepad требует использовать аж комбинацию служебных символов – `\r\n` («возврат каретки» и «перевод строки»). Чтобы за один раз решить все проблемы и не вдаваться слишком уж глубоко во все эти подробности, лучше запись строк в текстовый файл осуществлять не методом `Write()`, которым мы пользовались в настоящем разделе до сих пор (из соображений «тяжело в учении – легко в бою»), а *методом WriteLine()*. Тогда нам самим *не потребуется явно работать со специализированными строковыми разграничителями.*

Зафиксируем наши последние знания с помощью следующей учебной программы, представляющей из себя небольшую переделку текста Листинга 7.8, в которой мы также еще и напомним про возможность удобной работы с русским языком на платформе Microsoft NET Framework, в том числе и в смысле записи разноязычных текстов в файл и их последующей вычитки оттуда:

Листинг 7.10.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    string str1 = "Hello", str2 = ", Мир!";
    string Str1, Str2;

    /*-------------  file writing  -------------------*/
    FileStream fsW = new FileStream( "MyFileName",
                                     FileMode.Create,
                                     FileAccess.Write );
    StreamWriter sw = new StreamWriter( fsW );
    sw.WriteLine( str1 );
    sw.WriteLine( str2 );
    sw.Close();
    fsW.Close();
```

```
    /*-------------  file reading  -------------------*/
    FileStream fsR = new FileStream( "MyFileName",
                                     FileMode.Open,
                                     FileAccess.Read );
    StreamReader sr = new StreamReader( fsR );
    Str1 = sr.ReadLine();
    Str2 = sr.ReadLine();
    sr.Close();
    fsR.Close();

    Console.WriteLine( "Str1 = {0}\nStr2 = {1}\n", Str1, Str2 );
    Console.ReadLine();
    return 0;
  }
}
```

Результаты работы программы с Листинга 7.10 показаны на рис. 7.12.

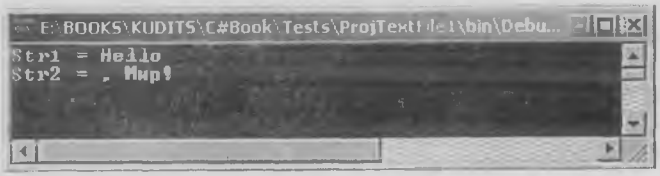


Рис. 7.12.

Из рис. 7.12 видно, что программа с Листинга 7.10 корректно обрабатывает разноязычный текст, а также разбивает текст в файле на две физические строки без явного применения служебного символа `\n` (метод `WriteLine()` «думает» за нас).

7.4. «Телефонная книга» – простейшая информационная система

Простейшую информационную систему – электронную телефонную книгу, будем хранить в текстовых файлах на диске компьютера. Это очень удобно, ибо позволяет создавать отдельные записи телефонной книги с помощью простейшего текстового редактора Notepad (Блокнот).

Будем в одной физической строке текста располагать фамилию абонента и его телефон:

Отделять фамилию и номер телефона друг от друга будем одним (или двумя) знаками табуляции, что и показано на рис. 7.13. Кроме того, файлу такой справочной телефонной системы мы присвоили имя Telephons.txt. Традиционное

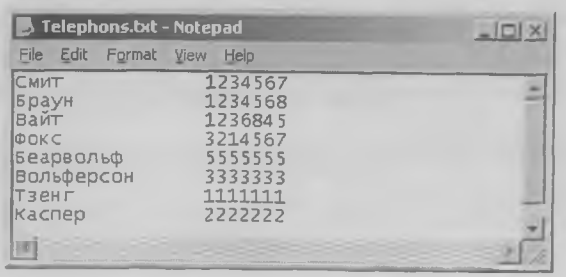


Рис. 7.13.

расширение имени файла txt наглядно показывает всем пользователям компьютера, что это текстовый файл и его содержимое всегда можно посмотреть с помощью стандартных текстовых редакторов.

Так как информационная система содержит символы русского языка, то файл Telephons.txt *нужно сохранять в кодировке UTF-8* (см. рис. 7.14).

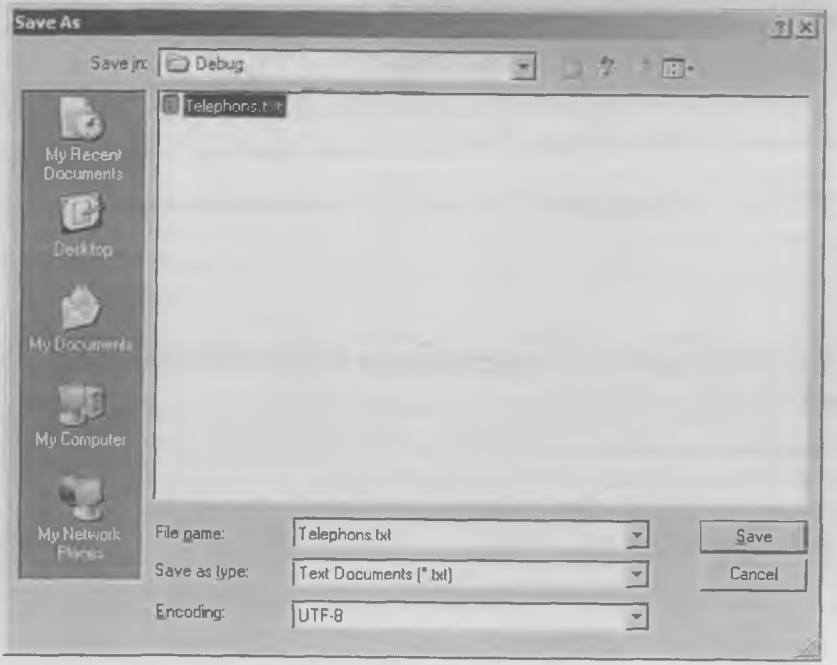


Рис. 7.14.

Недостатком интерактивной работы с текстовыми файлами с помощью стандартных текстовых редакторов является относительная медленность процесса и необходимость проявлять повышенную внимательность, чтобы ничего не пропустить. В связи с этим возникает вопрос, как автоматизировать процесс поиска телефона по фамилии абонента, чтобы не зависеть от качества работы человека и не перенапрягать его? Для этого нужно написать собственную программу, которая будет читать в автоматическом режиме из файла Telephons.txt строку за строкой, проверяя их на предмет вхождения в нее заданной фамилии.

Это и есть *пример простейшей справочной информационной системы*, для написания которой нам уже все известно, так что мы сразу приведем весь текст программы, а потом поясним отдельные фрагменты:

Листинг 7.11.

```
using System;
using System.IO;
class main
{
  public static int Main( )
  {
    string Name, PhoneNum, Str;
    int ind;

    Console.Write( "Введите фамилию: " );
    Name = Console.ReadLine();

    /*-------------  file reading  -------------*/
    if( File.Exists( "Telephons.txt" ) )
    {
      FileStream fsR = new FileStream( "Telephons.txt",
                                       FileMode.Open,
                                       FileAccess.Read );
      StreamReader sr = new StreamReader( fsR );

      while( ( Str = sr.ReadLine() ) != null )
      {
        ind = Str.IndexOf( Name, 0 );
        if( ind != -1 )
        {
          Console.WriteLine( "-------------------------" );
          Console.WriteLine( "{0}", Str );
        }
      }
      sr.Close();
      fsR.Close();
    }
```

```
    Console.WriteLine( "-------------------------\n" );
    Console.WriteLine( "The end." );
    Console.ReadLine();
    return 0;
  }
}
```

На рис. 7.15 пример сеанса работы с нашей справочной системой.

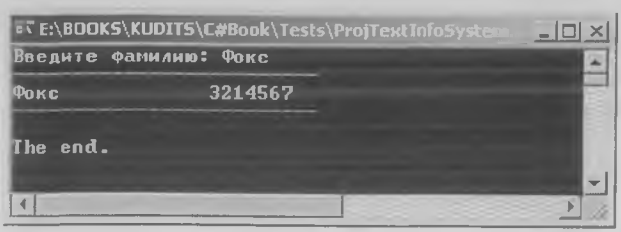


Рис. 7.15.

На запрос имени абонента мы вводим фамилию Фокс и нажимаем клавишу Enter, после чего на дисплее появляется строка из файла Telephons.txt, содержащая заданную фамилию и соответствующий номер телефона.

Процесс поиска введенной с клавиатуры фамилии абонента в читаемых из файла Telephons.txt (кодировка UTF-8) физических строках сосредоточен в следующем фрагменте из Листинга 7.11:

```
while( ( Str = sr.ReadLine() ) != null )
{
  ind = Str.IndexOf( Name, 0 );
  if( ind != -1 )
  {
    Console.WriteLine( "-------------------------" );
    Console.WriteLine( "{0}", Str );
  }
}
```

Потенциальное вхождение заданной фамилии в текущую строку, прочитанную только что из файла Telephons.txt методом `ReadLine()`, мы выявляем с помощью рассмотренного в разд. 4.3 метода `IndexOf()` библиотечного класса `String`. Если возврат этого метода не равен минус единице, то искомая фамилия в прочитанной строке найдена.

Читателям в качестве упражнения предлагается самостоятельно модифицировать информационную систему с Листинга 7.11 таким образом, чтобы поиск осуществлялся по номеру телефона.

Часть III

Ускоренная разработка Windows-приложений с графическим интерфейсом пользователя средствами библиотеки Microsoft NET Framework

Глава 8. Приложения Windows с графическим интерфейсом пользователя на базе диалоговых окон

8.1. Проект типа Win32 Application и его простейшие реализации на языках C/C++

Ограниченные возможности для предоставления удобного и наглядного *пользовательского интерфейса* (способов интерактивного взаимодействия с программой) в рамках консольных программ заставили создателей операционной системы Windows предусмотреть еще один тип приложений, имеющий развитый *графический интерфейс пользователя* (*GUI Windows applications*).

Язык программирования C# и платформа Microsoft NET Framework обеспечивают максимальную простоту и быстроту разработки подобного рода приложений, ибо богатейшая базовая функциональность всех типов окон Windows и специализированных *графических элементов управления* (*controls*) сосредоточена в готовом виде во многих классах библиотеки FCL, особенно в «оконном» классе Form. Чтобы наглядно проиллюстрировать указанные преимущества программирования GUI-приложений на платформе Microsoft NET Framework с применением языка C# над Win32-программированием тех же приложений с приме-

нением языков C/C++ ([1,2,3]), мы в настоящей главе выполним параллельную разработку простейшего модельного приложения средствами обеих технологий программирования.

Ранее в книге [1] рассматривалось создание простейших Windows-приложений с графическим интерфейсом пользователя на базе диалоговых окон средствами языка C и компилятора Microsoft Visul C++ 6.0. В настоящем разделе мы почти что полностью повторим этот путь, однако вместо языка C для разнообразия применим язык C++ (файлы с исходным кодом будут иметь расширение cpp, но в самом исходном коде отличия от языка C будут минимальнейшими и коснутся лишь однострочных комментариев, не стандартизованных для языка C). Кроме того, мы будем использовать компилятор Microsoft Visual C++ NET (2003), входящий в состав многоязыковой среды разработки Microsoft Developer Studio NET (2003) (см. Приложение к настоящему учебному пособию), ибо только в рамках этой среды программирования и ее более поздних версий возможно программирование на языке C#.

Ниже в настоящей главе мы повторим разработку приложения на базе диалогового окна, но уже на языке C# с применением компилятора Microsoft Visual C# NET (2003), также входящего в указанную среду разработки. Возможность прямого сравнения двух разработок явным образом продемонстрирует преимущества и недостатки каждого из этих подходов. Начинаем выполнять намеченный план с ретроспективного обзора программирования Win32-приложений на языке C++.

Из книги [1] нам известно, что в рамках компилятора Microsoft Visual C++ 6.0 программы с графическим интерфейсом пользователя создаются с помощью проектов типа Win32 Application. Для тех же целей в рамках компилятора Microsoft Visual C++ NET используется *проект Win32 Project* из семейства проектов Visual C++ Projects (см. рис. 8.1).

Диалоговое окно, показанное на рис. 8.1, вызывается по команде меню File | New | Project. Выбрав здесь местоположение каталога проекта на диске (поле Location с возможным применением кнопки Browse...) и имя проекта (поле Name), нажимаем кнопку OK, после чего в диалоговом окне Win32 Application Wizard нужно выделить *строку Application Settings* и установить *настройку Empty project* (поставить галочку напротив этой надписи в правой части диалогового окна, см. Приложение к настоящему учебному пособию).

В итоге создается пустой проект (созданы лишь служебные файлы проекта) с именем SimpleDialog, который мы далее будем наполнять следующим файловым содержимым:

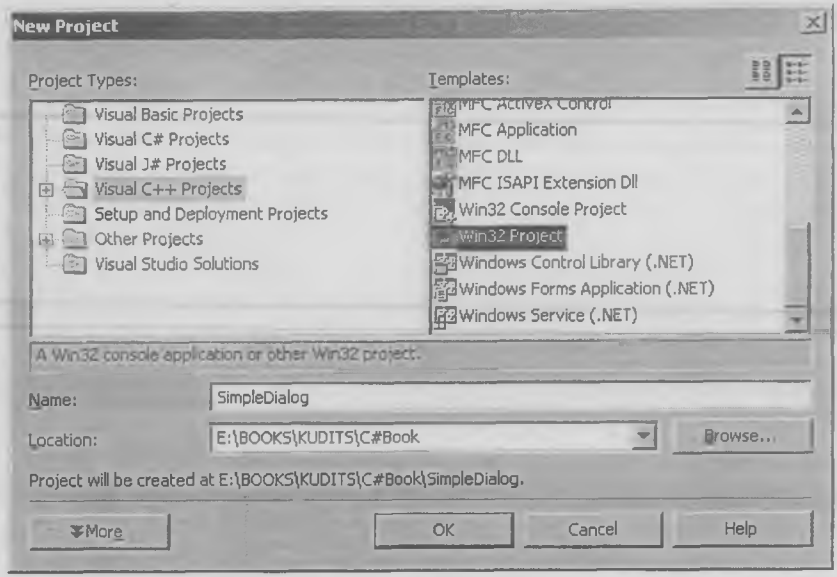


Рис. 8.1.

- Файлом main.cpp с исходным кодом программы на языке C++ (который в данном контексте не отличается от языка C)
- Ресурсным файлом Script.rc с текстовым описанием свойств диалогового окна и располагающихся на его поверхности кнопок
- Включаемым файлом resource.h с препроцессорными директивами #define для придания именованным макроконстантам уникальных числовых значений

Неторопливое и последовательное введение в логику и устройство перечисленных файлов дано в книге [1]. Поэтому здесь мы не будем повторно тратить время и усилия на подробное изложение этих основополагающих сведений. Вместо этого покажем на рис. 8.2 содержимое ресурсного файла SimpleDialog.rc так, как мы его набрали в простейшем текстовом редакторе Notepad (Блокнот):

Числовые значения макроконстант `IDB_OK` и `IDB_Cancel`, однозначно идентифицирующих кнопки с надписями OK и Cancel, определены с помощью директив препроцессора в файле resource.h:

```
#define IDB_OK     10
#define IDB_Cancel 11
```

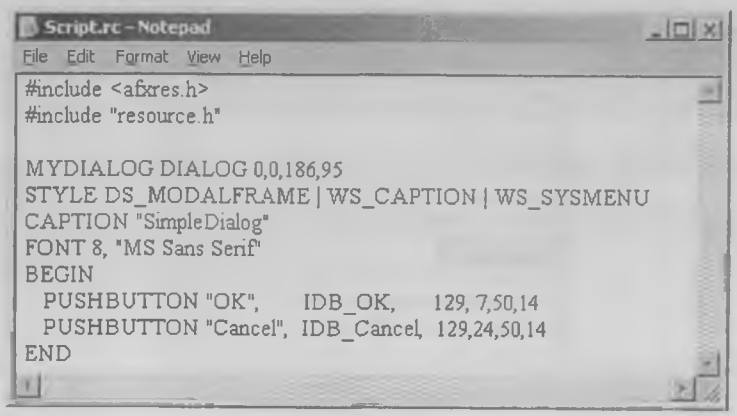


Рис. 8.2.

Файл Script.rc сохраняем на диске в корневом каталоге проекта SimpleDialog, и из графической среды компилятора Microsoft Visual C++ NET командой меню Project | Add Existing Item... подсоединяем файл Script.rc к проекту.

Содержимое файлов main.cpp и resource.h будем вносить непосредственно с помощью текстового редактора, встроенного в графическую среду компилятора Microsoft Visual C++ NET. Для этого создаем файлы и одновременно подсоединяем их к проекту одной и той же командой меню Project | Add New Item..., после чего и вносим в эти файлы исходный код на языке C++. Если содержимое файла resource.h состоит всего из двух директив препроцессора, показанных выше, то содержимое файла main.cpp существенно сложнее:

Листинг 8.1.

```
/*************************/
/*    File main.cpp      */
/*************************/
#include <windows.h>
#include "resource.h"

// Prototypes:
// Dialog window function:
int CALLBACK DlgProc( HWND, UINT, WPARAM, LPARAM );
// Message Handlers:
void OnClose( HWND hwnd );
void OnOK( HWND hwnd );
void OnCancel( HWND hwnd );
```

```
//////////////////////////////////////////////////////////
int WINAPI WinMain( HINSTANCE hI, HINSTANCE hI2,
                    LPSTR p, int n)
{
  int ret;
  ret = DialogBox( hI, "MYDIALOG", 0, (DLGPROC)DlgProc);

  if( ret == 0 )
    MessageBox( 0, "Close box", "Message", MB_OK );

  if( ret == 1 )
    MessageBox( 0, "OK button", "Message", MB_OK );

  if( ret == 2 )
    MessageBox( 0, "Cancel button", "Message", MB_OK );

  return 0;
}
//////////////////////////////////////////////////////////
int CALLBACK DlgProc( HWND hwnd , UINT iMsg,
                      WPARAM wParam, LPARAM lParam )
{
  if( iMsg == WM_CLOSE )
  {
    OnClose( hwnd );
    return 1;
  }

  if( iMsg == WM_COMMAND )
  {
    if( LOWORD( wParam ) == IDB_OK )
      OnOK( hwnd );

    if( LOWORD( wParam ) == IDB_Cancel )
      OnCancel( hwnd );

    return 1;
  }

  return 0;
}
//////////////////////////////////////////////////////////
void OnClose( HWND hwnd )
{
  EndDialog( hwnd, 0 );
}
void OnOK( HWND hwnd )
```

```
{
  EndDialog( hwnd, 1 );
}
void OnCancel( HWND hwnd )
{
  EndDialog( hwnd, 2 );
}
//////////////////////////////////////////////////////////
```

Окончательный файловый состав проекта SimpleDialog для наглядности показан в рамках окна компилятора Microsoft Visual C++ NET на рис. 8.3.

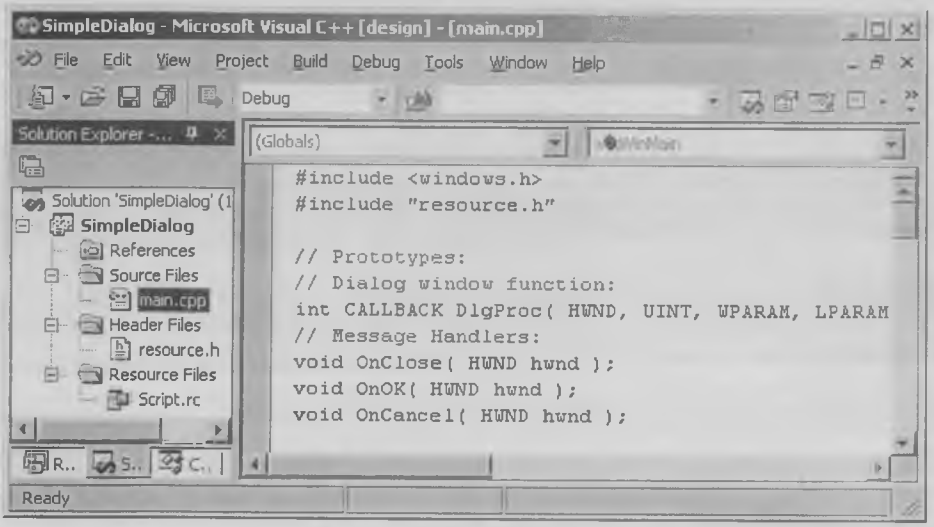


Рис. 8.3.

Компилируем проект в исполняемый файл SimpleDialog.exe, запускаем его на выполнение и получаем на дисплее компьютера визуальное изображение спроектированного нами (посредством текста ресурсного файла Script.rc, см. выше рис. 8.2) диалогового окна с надписью SimpleDialog, показанного на рис. 8.4.

Это диалоговое окно не обеспечивает богатой специфической (индивидуальной) функциональности, так как оно задумано в учебных целях исключительно для демонстрации принципиальных составляющих Windows-приложений с графическим интерфейсом пользователя. Оно обеспечивает лишь возможность за-

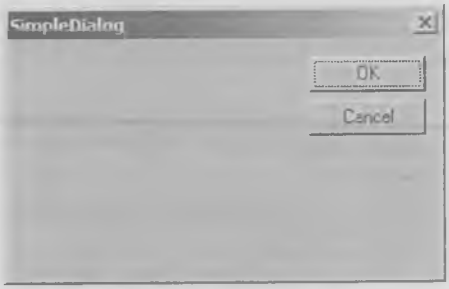


Рис. 8.4.

крытия окна с помощью мыши (о клавиатуре сообщим ниже): щелчок левой клавишей мыши на одной из кнопок или на изображении крестика в заголовке окна.

Наше приложение само «распознает», каким именно способом было закрыто окно (и, тем самым, само приложение), и выводит на дисплей то или иное окно сообщений. Например, на рис. 8.5 показано окно сообщений для случая, когда для закрытия окна была выбрана кнопка OK.

Рис. 8.5.

На самом деле, такая функциональность не столь и мала, особенно если ее рассматривать относительно наших дополнительных усилий на реализацию графического интерфейса пользователя. Более того, есть еще и скрытые возможности, например, использование *клавиатуры вместо мыши*. Поддержка клавиатуры достается нам даром. Если обратить внимание на *пунктирную прямоугольную рамку* на кнопке OK нашего диалогового окна, показанного выше на рис. 8.4, то можно догадаться, что речь идет о дополнительной визуальной информации про возможность «нажать на эту кнопку с помощью клавиатуры» – конкретно, *пробельной клавишей*. Кнопка с пунктирной рамкой говорит пользователю о том, что клавиша OK имеет в данный момент так называемый *клавиатурный фокус ввода*. Перевести клавиатурный фокус ввода на другую кнопку можно *клавишей Tab* (*клавиша табуляции*).

Более того, если пользователь нажмет на клавиатуре *комбинацию клавиш Alt+Пробел*, то в диалоговом окне появится *всплывающее меню*, показанное на рис. 8.6.

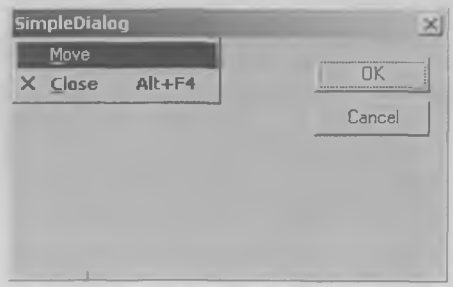


Рис. 8.6.

Нажав на клавишу Enter в ситуации, показанной на рис. 8.6, мы закроем наше диалоговое окно и получим на дисплее соответствующее окно сообщений (такое же, как если бы мы щелкнули мышью по изображению крестика в заголовке окна). Из рис. 8.6 мы видим также, что аналогичного эффекта можно добиться, сразу нажав на клавиатуре комбинацию клавиш Alt+F4.

8.2. Принципиальные особенности программирования графического интерфейса пользователя на языках C/C++

В общем случае, если Windows-приложения с графическим интерфейсом пользователя программируются на языках C/C++, то они напрямую взаимодействуют с механизмом сообщений этой операционной системы. Работа этого механизма заключается в том, чтобы преобразовать *аппаратные события* (*hardware events*), связанные с работой пользователя (например, с мышью или клавиатурой), в набор точно специфицированных числовых кодов (событий) и дополнительной информации (например, основной код говорит о нажатии клавиши на клавиатуре, а дополнительная информация специфицирует конкретную клавишу и, возможно, клавиатурную комбинацию), которая помещается в единую *очередь сообщений* (*messages queue*). За извлечение сообщений из очереди и их обработку отвечает само приложение.

Степень детальности работы Windows-приложения с системой сообщений в сильнейшей степени зависит от типа главного окна этого приложения. Если приложение базируется на модальном диалоговом окне, как в нашем приложе-

нии с Листинга 8.1, то большую часть работы берет на себя системная функция `DialogBox()`, а нам остается лишь реагировать на те или иные сообщения в *оконной процедуре* `DlgProc()`.

Оконная процедура вызывается системной функцией, относящейся к механизму сообщений (то есть вызов оконной процедуры осуществляется самой операционной системой), причем код сообщения передается в процедуру (функцию) `DlgProc()` через ее целочисленный параметр `iMsg`, а дополнительная сопутствующая информация передается через два других целочисленных параметра – `wParam` и `lParam` (см. Листинг 8.1).

Стартовая функция приложения имеет имя `WinMain()` и в первую очередь предназначена для вызова системной функции `DialogBox()`. Она, конечно, может дополнительно выполнить некоторую работу до вызова указанной системной функции, и после возврата из нее (в Листинге 8.1 мы как раз анализируем возвращаемое функцией `DialogBox()` значение и предпринимаем некоторые действия в ответ).

Если бы мы не стали базировать наше приложение на одном лишь модальном диалоговом окне, то стартовая функция `WinMain()` должна была бы самостоятельно запускать *цикл выборки сообщений* и напрямую создавать окна с помощью довольно сложных системных функций, предварительно не забыв зарегистрировать так называемый класс окон (не путать с классами языков C, C++ или C#). В общем, приложения стали бы многократно более сложными (см., например, гл. 11 из книги [2], или книгу [7]).

Но и наше упрощенное приложение на базе модального диалогового окна содержит изрядное количество сложных деталей и мелких подробностей взаимодействия с механизмом сообщений операционной системы Windows. Самое главное, что оно вынуждено иметь в своем составе невызываемую из стартовой функции `WinMain()` оконную процедуру (см. рис. 8.7).

Сама же оконная процедура, выполняя разбор и идентификацию пришедшего сообщения (в виде фактических параметров ее вызова), выполняет в ответ некоторые специфичные для каждого типа сообщения действия. Эти действия можно запрограммировать в виде кода, входящего в тело оконной процедуры (как в Листинге 8.8 из книги [1]), либо, как в представленном выше Листинге 8.1, реализовать в виде отдельных *функций-обработчиков*.

Показанную на рис. 8.7 архитектуру приложения с графическим интерфейсом пользователя (а ведь это упрощенное приложение на базе модального диалогового окна) следует признать весьма сложной. Причем не только по причине необходимости близкого знакомства с механизмом сообщений операционной системы Windows, но и по многочисленным иным причинам. Например, сильно

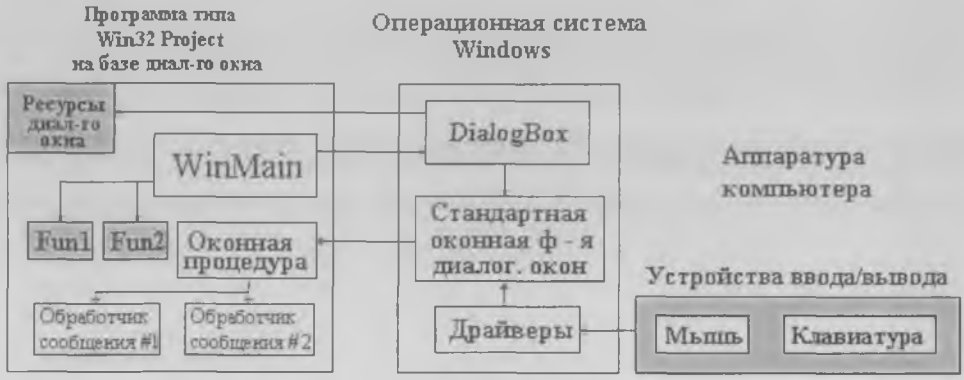


Рис. 8.7.

усложняет первоначальное изучение материала и практическое выполнение разработок серьезная неоднородность необходимых знаний: помимо языков программирования (C или C++) и плотного знакомства с механизмами операционной системы требуется еще понимание синтаксиса и мелких деталей создания ресурсного файла, знание параметров вызова многочисленных системных Win32 API функций (у нас в Листинге 8.1 из многих сотен имеющихся используются лишь функции `DialogBox()`, `EndDialog()` и `MessageBox()`), а также знакомство с *макроопределениями типов данных*, которыми буквально пестрит код приложений с графическим интерфейсом пользователя на языках C/C++.

Действительно, без применения макроопределений типов данных код простейшего варианта стартовой функции (для ничего не делающей программы)

```
int __stdcall WinMain( void* hI, void* hI2,
                       char* p, int n)
{
  return 0;
}
```

выглядит более-менее традиционно для учебной литературы по программированию на языках C/C++. Кратко поясним этот код. В качестве первого параметра функция `WinMain()` получает указатель `hI`, который ей следует хранить, ибо он с точки зрения операционной системы идентифицирует запущенный на выполнение экземпляр нашей программы. Часто, при вызове стандартных функций операционной системы (функций Win32 API) указатель `hI` нужно будет передавать в виде фактических параметров вызова. Ключевое слово `__stdcall`

(впереди стоят два знака подчеркивания) является низкоуровневым модификатором вызова функций языков C/C++, и его требуется ставить перед именем функции `WinMain()`.

Однако в учебной литературе и в текстах практических разработок Windows-приложений код стартовой функции выглядит по-другому из-за многочисленных переопределений типов данных:

```
#include <windows.h>

int WINAPI WinMain( HINSTANCE hI, HINSTANCE hI2,
                    LPSTR p, int n)
{
  return 0;
}
```

В таких заменах кроется большой практический смысл, поскольку в случае коммерческих программ с большим сроком эксплуатации возможна ситуация, когда произойдет радикальная смена версии операционной системы Windows, при которой нужно будет изменить многие типы данных. В случае использования переопределений основной текст программы менять не придется, достаточно лишь использовать иной вариант заголовочного файла windows.h и перекомпилировать весь текст программы заново.

Заголовочный файл windows.h (он сам включает в себя множество других заголовочных файлов, знать имена которых нам нет никакой необходимости) кроме прототипов стандартных функций операционной системы содержит великое множество директив препроцессора `#define`, с помощью которых вводятся многочисленные новые имена для стандартных и пользовательских типов данных языка C: например, тип данных `void*` превращается в результате переопределений в `HINSTANCE`, тип `char*` превращается в `LPSTR`, а модификатор `__stdcall` представлен теперь идентификатором `WINAPI`.

Про переопределенные типы, связанные с оконной процедурой (см. выше Листинг 8.1), можно сказать практически то же самое, что и про такого же рода типы, связанные со стартовой функцией `WinMain()`: все они являются переопределениями стандартных типов языков C/C++ с помощью директив препроцессора `#define`. Типы `UINT`, `WPARAM` и `LPARAM` сводятся к целому типу `int`, `CALLBACK` представляет собой малоинтересный для нас модификатор вызова функции, а `HWND` в итоге является указателем типа `void*` и по своему предназначению очень похож на ранее рассмотренный тип `HINSTANCE`. Как мы знаем, тип `HINSTANCE` предназначен для идентификации с помощью указателя данных, характеризующих запущенный на выполнение экземпляр нашей програм-

мы, и мы должны его хранить, чтобы предъявлять функциям Win32 API в качестве фактических параметров вызова. А тип *HWND* предназначен для идентификации с помощью указателя диалогового окна нашей программы.

Подводя итоги раздела, отметим, что рис. 8.7 наглядно демонстрирует тот факт, что хотя операционная система и изолирует эффективно Windows-приложения от необходимости работать с аппаратурой компьютера напрямую, она, взамен, требует работать напрямую уже с ней самой.

8.3. Программирование графического интерфейса пользователя на языке C#

Анализируя содержимое рис. 8.7, констатируем, что много усилий при разработке на языках C/C++ приложений Windows с графическим интерфейсом пользователя приходится уделять многочисленным мелким технологическим деталям в ущерб решению конкретной специфической задачи, стоящей перед программистом.

Платформа Microsoft NET Framework и язык C# позволяют преодолеть эту проблему, сосредотачивая технологические детали в классах библиотеки FCL (см. рис. 8.8).

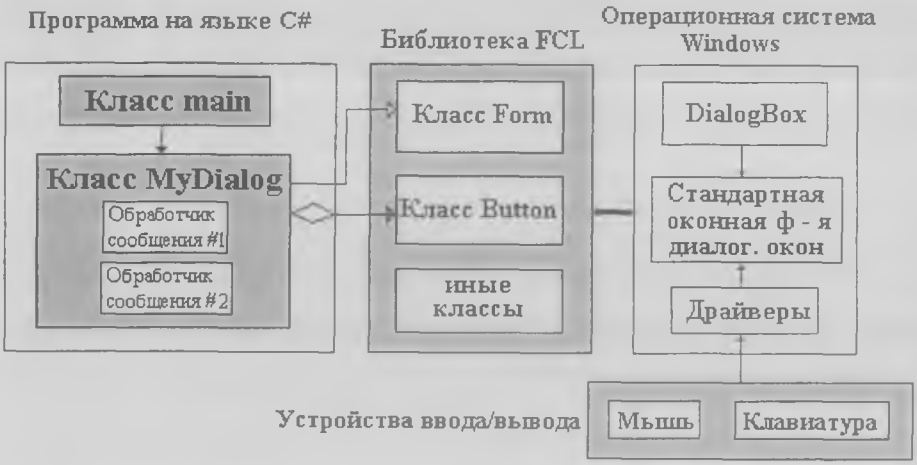


Рис. 8.8.

На рис. 8.8 показано, что классы библиотеки FCL закрывают от пользовательского приложения детали взаимодействия с операционной системой. Биб-

лиотека FCL состоит из множества классов, связанных между собой теми или иными отношениями, которые мы на рис. 8.8 отображать даже не пытаемся. Также мы не пытаемся выяснять, как именно классы библиотеки FCL взаимодействуют с функциями Win32 API операционной системы Windows (это нужно лишь самим разработчикам библиотеки FCL).

Для реализации графического интерфейса пользовательское приложение взаимодействует с необходимыми для этой цели классами FCL, например, с основным «оконным классом» Form и классом кнопок (устойчивое название для стандартного графического элемента управления) Button.

При этом пользовательское приложение состоит из двух классов – класса main со стартовым методом `Main()`, в котором и создается объект типа `MyDialog`, представляющий собой наше диалоговое окно. Чтобы класс `MyDialog` обеспечивал функциональность диалогового окна, его следует определить как производный от библиотечного класса `Form`. Наличие кнопок в нашем диалоговом окне обеспечивается тем, что класс `MyDialog` агрегирует библиотечный класс `Button` (про агрегацию улассов см. разд. 5.2). Указанные связи между классами отражены на рис. 8.8 соответствующего вида связующими линиями и стрелками.

В результате пользовательский код приобретает ранее недостижимую кристальную чистоту и прозрачность – никаких тебе ресурсных файлов с их уникальным синтаксисом, нет необходимости писать диалоговую процедуру, пестрящую типами `HWND`, `WPARAM`, `LPARAM` и так далее. То есть мы все время остаемся в рамках языка C#, и никаких «инородных примесей».

Ясно, что за это чем-то «нужно заплатить» (помимо разработки библиотеки FCL). Платой является необходимость использовать такие неординарные синтаксические конструкции языка C#, как *делегаты* и *события*. Подробное изучение этих конструкций не входит в наши планы (оно вообще неуместно для начинающих программистов, а неначинающих отсылаем к книге [8]), так что мы отделаемся общими описаниями. Реальная же практическая работа, затрагивающая стандартные делегаты и события, определенные в классах библиотеки FCL, весьма простая, в чем мы вскоре и убедимся.

Причина введения в язык C# указанных новых конструкций может быть пояснена на примере *обработчиков событий*, которые теперь уже не привязываются к оконной процедуре (мы ее вообще теперь не пишем), а являются просто методами нашего класса MyDialog. Однако же, мы все равно нуждаемся в механизме, с помощью которого можно «привязать» эти обработчики событий к реальным аппаратным событиям и представляющим их сообщениям операционной системы Windows. Для этого в язык C# и вводятся делегаты и события.

Делегат представляет собой необычно определяемый *тип данных* (помечается *ключевым словом delegate*, но как определение типа данных не выглядит), *производный от библиотечного класса MulticastDelegate*, и предназначенный как для хранения указателей (адресов) функциональных классовых методов, так и для их последующего косвенного вызова (достаточно скрытого от исходного кода на языке C#). Через механизм делегатов языка C# мы будем регистрировать наши обработчики событий в механизме сообщений операционной системы Windows (транзитом через классы библиотеки FCL, конечно).

Для этого в соответствующих классах библиотеки FCL (классе Button, например) определяются *события* (еще один *вид членов класса языка C#*, помечаемый *ключевым словом event*), которые выглядят как поля данных, имеющие тип некоторого делегата, но для них компилятор автоматически добавляет два скрытых вспомогательных функциональных метода, позволяющих связать делегатный объект с его целевым методом (для последующего косвенного вызова целевого метода через объект делегата). Кроме того, в этих классах определяются операции «+=» и «-=» для скрытого вызова указанных только что скрытых вспомогательных методов, предназначенных для привязки обработчиков событий к самим событиям.

В общем, это весьма сложные для первоначального изучения синтаксические конструкции языка C#, детальное знание которых нужно лишь разработчикам классов типа оконных классов библиотеки FCL, к практическому применению которых мы сейчас и приступаем. Для этого приведем целиком исходный код на языке C#, сосредоточенный в единственном файле main.cs, и предназначенный для реализации приложения с графическим интерфейсом пользователя на базе диалогового окна, вполне по своей функциональности аналогичного приложению, которое мы разрабатывали в разд. 8.1 на языке C++ (см. Листинг 8.1).

Листинг 8.2.

```
/************************/
/*    File main.cs      */
/************************/
using System;
using System.Drawing;
using System.Windows.Forms;

class main
{
  public static int Main( )
  {
    MyDialog myDlg = new MyDialog();
```

```
    DialogResult res = myDlg.ShowDialog( null );

    if( res == DialogResult.Cancel )
      MessageBox.Show( "Close Box", "Message" );

    return 0;
  }
}
class MyDialog : Form
{
  // Конструктор:
  public MyDialog()
  {
    // Внешний вид, размер и иные
    // характеристики диалогового окна:
    MaximizeBox = false;
    MinimizeBox = false;
    Text = "SimpleDialog";
    ClientSize = new Size( 186, 95 );
    FormBorderStyle = FormBorderStyle.FixedDialog;
    ShowInTaskbar = false;

    // Задание шрифта для окна и метрической системы,
    // сходной с той, что применяется в проектах на языках C/C++:
    Font = new Font( "MS Sans Serif", 8 );
    AutoScaleBaseSize = new Size( 4, 8 );

    // Создаем два новых "кнопочных объекта"
    // и прикрепляем их к диалоговому окну:
    btn1 = new Button();
    btn2 = new Button();
    Controls.Add( btn1 );
    Controls.Add( btn2 );

    // Местоположение и размер кнопки с надписью "OK":
    btn1.Location = new Point( 129, 7 );
    btn1.Size = new Size( 50, 14 );
    btn1.Text = "OK";

    // Местоположение и размер кнопки с надписью "Cancel":
    btn2.Location = new Point( 129, 24 );
    btn2.Size = new Size( 50, 14 );
    btn2.Text = "Cancel";
```

```
    //   !!!!
    // Привязка функциональных обработчиков к кнопочному
    // событию Click:
    btn1.Click += new EventHandler( OnOK );
    btn2.Click += new EventHandler( OnCancel );
  }
  // Определение функциональных обработчиков:
  private void OnOK( object obj, EventArgs e )
  {
    MessageBox.Show( "OK button", "Message" );
  }
  private void OnCancel( object obj, EventArgs e )
  {
    MessageBox.Show( "Cancel button", "Message" );
  }
  // Закрытые поля данных, агрегирующие кнопки:
  private Button btn1;
  private Button btn2;
}
```

Сравнение этого кода на языке C# с Листингом 8.1 для функционально идентичной программы на языке C++ наглядно демонстрирует все сказанное выше – код действительно стал предельно однородным (никаких вкраплений ресурсных файлов, никакого «столкновения» с мелкими подробностями из мира системы сообщений операционной системы Windows) и существенно более простым, как для написания, так и для его последующего восприятия.

Простоте кода с Листинга 8.2 сильнейшим образом способствует *тотальное применение операций присваивания* (вместо типичных для C/C++ функциональных вызовов), базирующихся на таких новаторских элементах классов языка C#, как свойства (про свойства см. разд. 5.1). Для клиента классов языка C# работа со свойствами неотличима от работы с открытыми полями данных, так что далее мы даже не будем вдаваться в подробности и скрупулезно фиксировать, с чем именно имеем дело в том или ином конкретном случае.

Единственный концептуально сложный момент в Листинге 8.2 – это привязка обработчиков событий к самим событиям, о чем мы уже в предварительном и обзорном плане говорили выше. Вот этот невеликий по размеру фрагмент кода из Литинга 8.2:

```
btn1.Click += new EventHandler( OnOK );
btn2.Click += new EventHandler( OnCancel );
```

в котором мы привязываем наши собственные методы-обработчики `OnOK()` и `OnCancel()` к *событию Click*, определенному в библиотечном классе `Button` (или в его родительских библиотечных классах, что нам сейчас непринципиально), и означающему «нажатие на кнопку» в самом широком смысле, то есть левой клавишей мыши, или с помощью клавиатуры.

Привязка осуществляется через создаваемый операцией new объект *стандартного библиотечного делегатного типа EventHandler*, конструктору которого передается идентификатор метода-обработчика. Этот делегатный тип предназначен как раз для обработки сообщений операционной системы Windows и требует от регистрируемых методов-обработчиков иметь два параметра типа `object` и `EventArgs`, соответственно:

```
private void OnOK( object obj, EventArgs e )
{ MessageBox.Show( "OK button", "Message" ); }
private void OnCancel( object obj, EventArgs e )
{ MessageBox.Show( "Cancel button", "Message" ); }
```

Первый параметр идентифицирует пославший сообщение объект, то есть конкретная кнопка в нашем случае. Этот параметр приходится использовать в случаях, когда для нескольких кнопок регистрируется единственный метод-обработчик, как в следующем примере (альтернативный код для программы с Листинга 8.2):

```
...
btn1.Click += new EventHandler( OnClick );
btn2.Click += new EventHandler( OnClick );
...
private void OnClick( object obj, EventArgs e )
{
  if( obj == btn1 )MessageBox.Show( "OK button", "Message" );
  else MessageBox.Show( "Cancel button", "Message" );
}
```

Второй параметр имеет *тип EventArgs* – это *базовый библиотечный класс, предназначенный для создания объектов, несущих полезную дополнительную информацию о событии*. Реально полезную информацию несут лишь объекты классов, производных от `EventArgs` (поэтому в наших примерах методов-обработчиков мы никак не используем параметр e типа `EventArgs`). Например, для событий мыши их обработчики имеют второй параметр типа `MouseEventArgs` (библиотечный класс, производный от `EventArgs`) и который несет дополнительную информацию о событии, например, координаты курсора мыши.

Если из наших кратких пояснений возникло ощущение, что механизм обработки сообщений в программах на языке C# все-таки слишком сложен, спешить возразить, что сложным является синтаксис определения делегатов и событий в классах C#, но их использование в клиентском коде абсолютно стандартное, так что мы уже показали практически все, что можно и нужно делать в собственном коде на эту тему.

От рассмотрения механизма обработки сообщений Windows в приложениях с графическим интерфейсом пользователя на языке C# перейдем к более простым вещам. Например, в стартовом методе `Main()` мы создаем объект `myDlg` типа `MyDialog` и запускаем работу графического интерфейса пользователя вызовом метода `ShowDialog()`:

```
DialogResult res = myDlg.ShowDialog( null );
```

Именно эта строка кода заставляет главное окно нашего приложения, имеющее тип `MyDialog`, реально работать *в режиме модального диалогового окна*. Дело в том, что библиотечный класс `Form` реализует функциональность всех типов окон Windows, а не только модальных диалоговых окон. Для реализации последнего типа поведения и вызывается специально для этого предназначенный метод `ShowDialog()` базового класса `Form`. Иные типы окон Windows будут нами рассмотрены в следующем разделе главы.

В конструкторе нашего класса `MyDialog`, наследующего от базового класса `Form`, выполняется вся работа по настройке свойств и характеристик самого диалогового окна и расположенных на его поверхности кнопок.

Сначала для самого окна задаются следующие характеристики:

```
MaximizeBox = false;
MinimizeBox = false;
Text = "SimpleDialog";
ClientSize = new Size( 186, 95 );
FormBorderStyle = FormBorderStyle.FixedDialog;
ShowInTaskbar = false;
```

Имена полей (или свойств) класса `Form` (или его базовых классов – что именно, нам сейчас нет нужды знать) указываются слева от знака присваивания и практически «говорят сами за себя». Например, значение `false` для поля `ShowInTaskbar` означает, что для нашего окна иконка в панели задач отображаться не будет, так как это соответствует стандарту поведения диалоговых окон. Если для поля `FormBorderStyle` задается значение `FormBorderStyle.FixedDialog`, то диалоговое окно имеет неизменный размер, что также является типичным поведением для указанного вида окон.

Обратим внимание на то, что в последнем случае мы использовали одно и то же имя FormBorderStyle для обозначения разных программных сущностей. Справа от знака присваивания это имя относится к типу (перечислению), определенному в библиотеке FCL в рамках пространства имен System.Windows.Forms. В том же пространстве имен определен библиотечный класс Form, поле которого (точнее, свойство, но мы далее не будем на этом фиксировать наше внимание и обращать внимание на употребление точной терминологии) также имеет имя FormBorderStyle, но здесь нет никакого конфликта имен из-за разной их области видимости, так как имена членов класса входят в классовую область видимости. Компилятор легко отличает разный смысл одного и того же идентификатора по разному синтаксису их использования.

Далее, перед созданием и настройкой кнопок мы выполняем несколько загадочный (на первый взгляд) подготовительный код:

```
Font = new Font( "MS Sans Serif", 8 );
AutoScaleBaseSize = new Size( 4, 8 );
```

Первая строка создает конкретный шрифт и назначает к использованию в нашем окне, а вторая строка заставляет компилятор трактовать нижележащие кнопочные координаты в специальных единицах измерения, кратных среднему размеру назначенного шрифта, что является стандартом в низкоуровневом программировании окон в рамках операционной системы Windows, и что мы, естественно, применяли при написании программы с Листинга 8.1 на языке C++ (точнее, при формировании содержимого ресурсного файла к этой программе, показанного на рис. 8.2). Более подробный материал на эту тему, а также на тему свойств многочисленных стандартных графических элементов управления Windows, на тему шрифтов, на тему рисования в окнах и на многие другие практически полезные темы представлен в замечательной книге [9].

Кнопки создаются как объекты класса Button и ссылки на них запоминаются в полях btn1 и btn2 нашего класса MyDialog (обычная агрегация классов):

```
btn1 = new Button();
btn2 = new Button();
```

после чего эти ссылки заносятся в список *дочерних элементов управления* нашего диалогового окна:

```
Controls.Add( btn1 );
Controls.Add( btn2 );
```

Для этого используется поле `Controls` класса `Form`, имеющее тип `ControlCollection`, представляющее из себя контейнер (или класс коллекций), аналогичный по функциональности рассмотренному нами выше в разд. 6.5 контейнеру `ArrayList` (тот же метод `Add()` для добавления элементов в контейнер).

Завершается код конструктора класса `MyDialog` настройкой характеристик кнопок (размер, положение и надпись на кнопках) и привязкой к ним методов-обработчиков.

Пояснив код с Листинга 8.2, перейдем к практическому созданию проекта, для чего опять выберем проект типа Empty Project языка C# с целью максимально самостоятельной работы без привлечения изрядной помощи от графической среды компилятора Microsoft Visul C#. Мы еще воспользуемся этой помощью в следующем разделе главы и проиллюстрируем тезис об этой среде как о среде ускоренной разработки приложений Windows всех типов, в первую очередь приложений с графическим интерфейсом пользователя. А сейчас нам лучше поработать вручную («тяжело в учении – легко в бою»).

Создав проект ProjDialog1 типа EmptyProject и внедрив в него файл main.cs с кодом Листинга 8.2, выполним дополнительные настройки проекта.

Сначала установим *ссылки* (*References*)на нужные в данном случае файлы библиотеки FCL – файлы System.dll, System.Drawing.dll и System.Windows.Forms.dll (про установку ссылок на библиотечные файлы см. разд. 1.4). Щелкаем правой клавишей мыши по строке References в левой части окна графической среды компилятора Microsoft Visual C# и из всплывающего контекстного меню выбираем команду Add Reference..., после чего появляется одноименное диалоговое окно (см. рис. 8.9).

Селектируя нужные файлы в левой части окна Add Reference (при активной закладке .NET) с помощью кнопки Select формируем требуемый список файлов в нижней секции Selected Components, после чего нажимаем кнопку OK и дело сделано. Как выглядит окно компилятора на этой стадии настройки проекта и внедрения в него файлов с исходным кодом (в нашем случае – единственного файла main.cs), показано на рис. 8.10.

Наконец, мы настраиваем такое свойство нашего проекта, как Output Type, которое по умолчанию для проектов типа Empty Project устанавливается равным Console Application (консольный тип Windows-приложения, с которым мы до сих пор всегда имели дело в нашем учебном пособии). Командой меню Project | Properties... вызываем диалоговое окно Property Pages (см. рис. 8.11), в котором при активной (подсвеченной) строке Common Properties из левой его части нужно выставить свойство Output Type равным значению Windows Application (Windows-приложение с графическим интерфейсом пользователя).

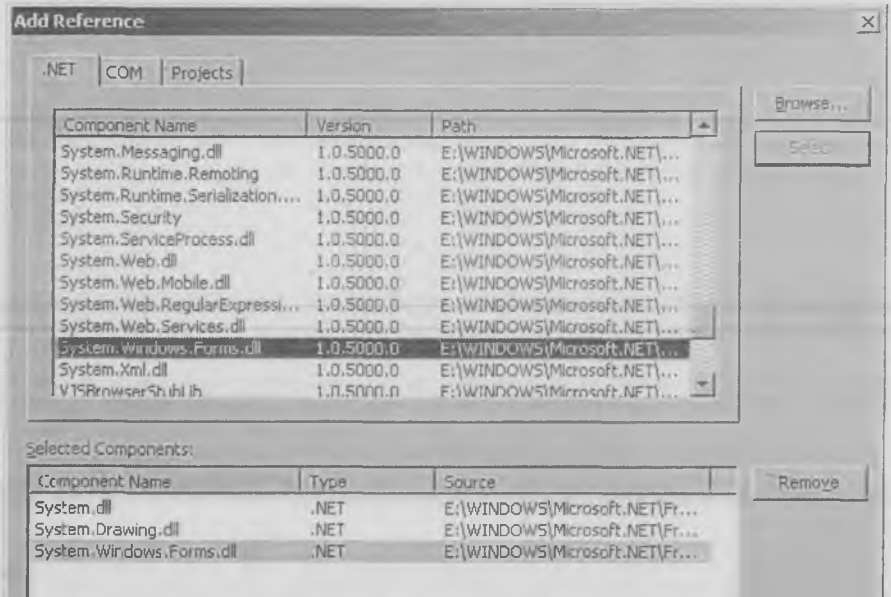

Add Reference
.NET
COM
Projects
Browse...
Select
Component Name
Version
Path
System.Messaging.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Runtime.Remoting 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Runtime.Serialization.... 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Security 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.ServiceProcess.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Web.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Web.Mobile.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Web.RegularExpressi... 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Web.Services.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Windows.Forms.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
System.Xml.dll 1.0.5000.0 E:\WINDOWS\Microsoft.NET\...
Selected Components:
Component Name
Type
Source
Remove
System.dll .NET E:\WINDOWS\Microsoft.NET\Fr...
System.Drawing.dll .NET E:\WINDOWS\Microsoft.NET\Fr...
System.Windows.Forms.dll .NET E:\WINDOWS\Microsoft.NET\Fr...

Рис. 8.9.

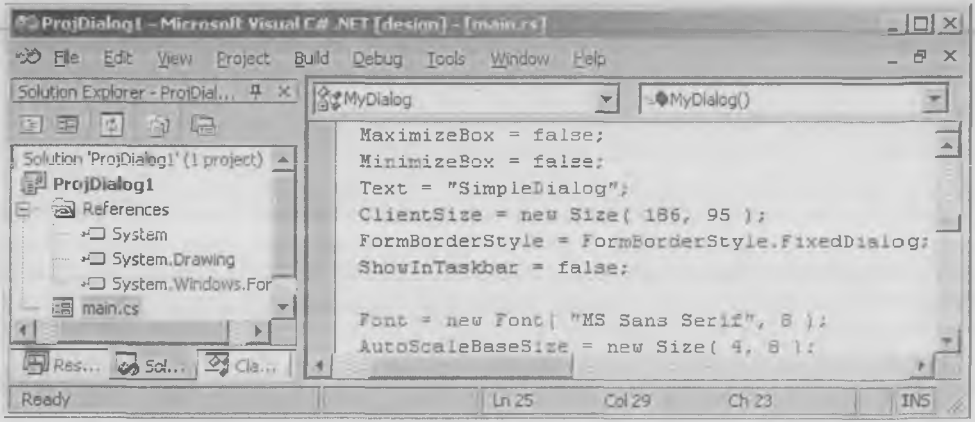


Рис. 8.10.

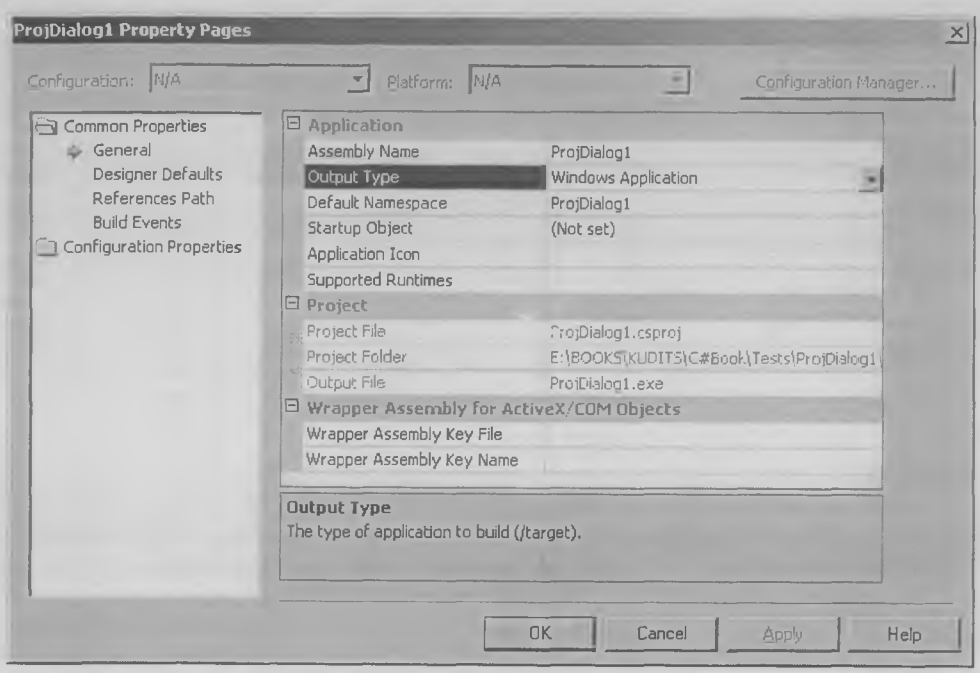


Рис. 8.11.

После выполнения указанных настроек проект ProjDialog1 компилируется обычным образом, запускается на выполнение и демонстрирует на дисплее тоже самое диалоговое окно с тем же самым поведением, которое ранее в разд. 8.1 мы программировали на языке C++ и которое иллюстрируется с помощью рис. 8.4 и рис. 8.5.

Несмотря на то, что программы с графическим интерфейсом на языке C# писать и поддерживать (менять и совершенствовать в процессе эксплуатации) значительно легче, чем на языке C++, все равно нужно приложить значительные начальные усилия, подробно и в деталях ознакомившись с принципиальными возможностями основных классов библиотеки FCL, имеющими отношение к графическому интерфейсу. Здесь можно порекомендовать книги [5,9], а также систему встроенной помощи графической среды компилятора Microsoft Visual C#. Например, поставив в окне (см. рис. 8.10) текстовый курсор на идентификаторе `MaximizeBox` и нажав клавишу F1, получим следующую справочную информацию по этому элементу класса `Form` (см. рис. 8.12).

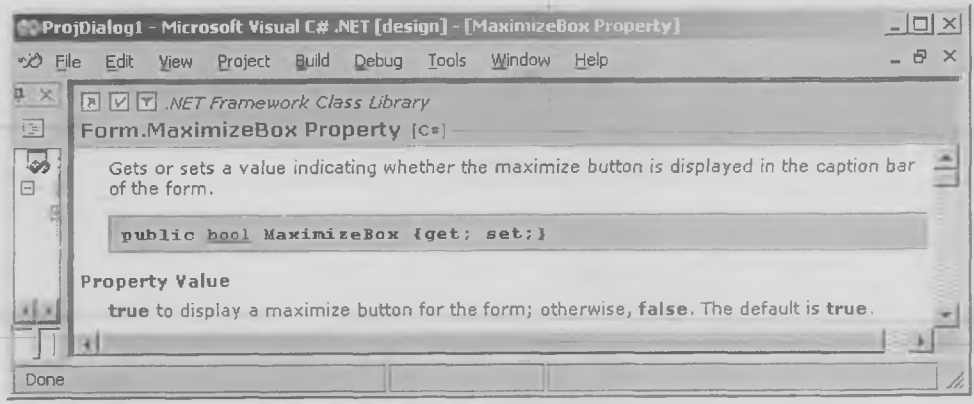


Рис. 8.12.

Минимально необходимая встроенная справочная информация часто дополняется примерами полезного практического кода (чаще всего на нескольких языках программирования, так что нужно выбрать кусок кода на языке C#). Например, на рис. 8.13 показан пример кода, перекликающийся по многим деталям (но не по всем) с нашим учебным Листингом 8.2.

Из рис. 8.13 видны, в частности, примеры настройки свойств `FormBorderStyle`, `MaximizeBox` и др.

Поэтому про встроенную справочную систему графической среды компилятора Microsoft Visual C# никогда не нужно забывать, и ее стоит использовать и как срочное справочное средство в случаях, когда забыты лишь отдельные детали, и даже как учебное средство, позволяющее овладеть некоторым первоначальным минимумом сведений (при наличии общего понимания проблемы и знания языка C# в объеме нашего учебного пособия).

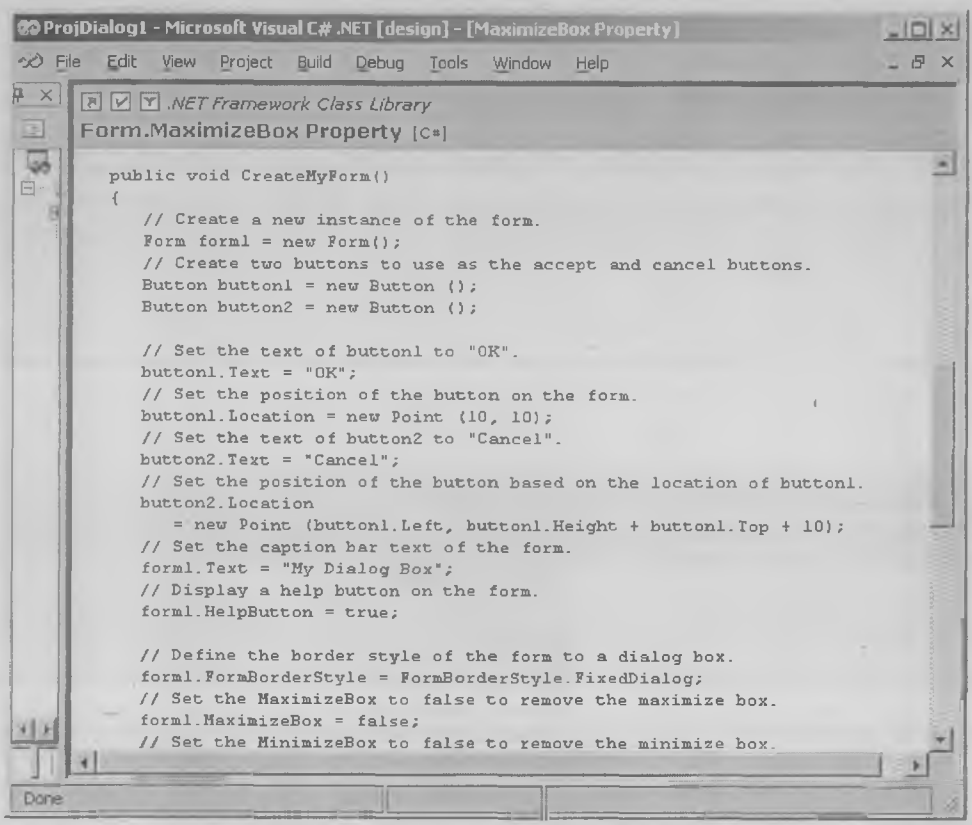

ProjDialog1 - Microsoft Visual C# .NET [design] - [MaximizeBox Property]
File Edit View Project Build Debug Tools Window Help
.NET Framework Class Library
Form.MaximizeBox Property [C#]
public void CreateMyForm()
{
// Create a new instance of the form.
Form form1 = new Form();
// Create two buttons to use as the accept and cancel buttons.
Button button1 = new Button ();
Button button2 = new Button ();
// Set the text of button1 to "OK".
button1.Text = "OK";
// Set the position of the button on the form.
button1.Location = new Point (10, 10);
// Set the text of button2 to "Cancel".
button2.Text = "Cancel";
// Set the position of the button based on the location of button1.
button2.Location
= new Point (button1.Left, button1.Height + button1.Top + 10);
// Set the caption bar text of the form.
form1.Text = "My Dialog Box";
// Display a help button on the form.
form1.HelpButton = true;
// Define the border style of the form to a dialog box.
form1.FormBorderStyle = FormBorderStyle.FixedDialog;
// Set the MaximizeBox to false to remove the maximize box.
form1.MaximizeBox = false;
// Set the MinimizeBox to false to remove the minimize box.
Done

Рис. 8.13.

8.4. Ускоренная разработка Windows-приложений с графическим интерфейсом на базе компилятора Microsoft Visual C#

Рассмотренный в предыдущем разделе ручной способ создания пустого проекта (Empty Project) и его последующей настройки, а также самостоятельное написание исходного кода на языке C# для создания Windows-приложения с графическим интерфейсом пользователя на базе диалогового окна хорошо подходит на этапе обучения, ибо позволяет все понять наилучшим образом. Однако для профессионального тиражного программирования не помешали бы вспомогательные автоматизирующие средства ускоренной разработки Windows-приложений. И такие средства в графической среде компилятора Microsoft Visual C# имеются в полной объеме. Сейчас мы ими и воспользуемся.

Для этого нужно создавать *проект типа Windows Application* (см. рис. 8.14).

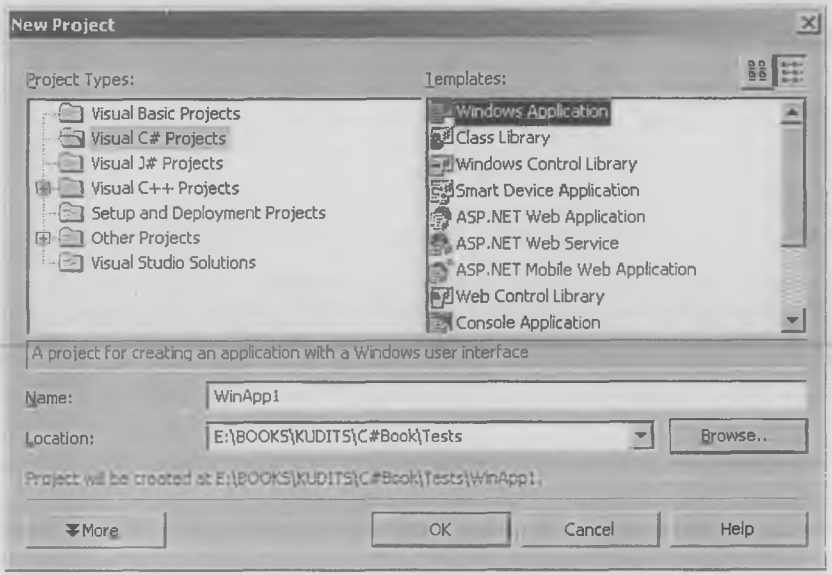


Рис. 8.14.

Как только мы в окне New Project выберем имя и каталог для нашего проекта и нажмем кнопку OK, в дело вступит автоматический генератор кода, который создаст *каркас полноценного приложения Windows*, базирующегося на главном окне свободного стиля (а не модальном диалоговом окне). Чтобы убедиться

в столь радостном факте, достаточно взглянуть на внешний вид главного окна графической среды компилятора Microsoft Visual C#, который оно принимает сразу же после описанных действий (сводящихся, в основном, к нажиманию кнопок), и который мы демонстрируем ниже на рис. 8.15.

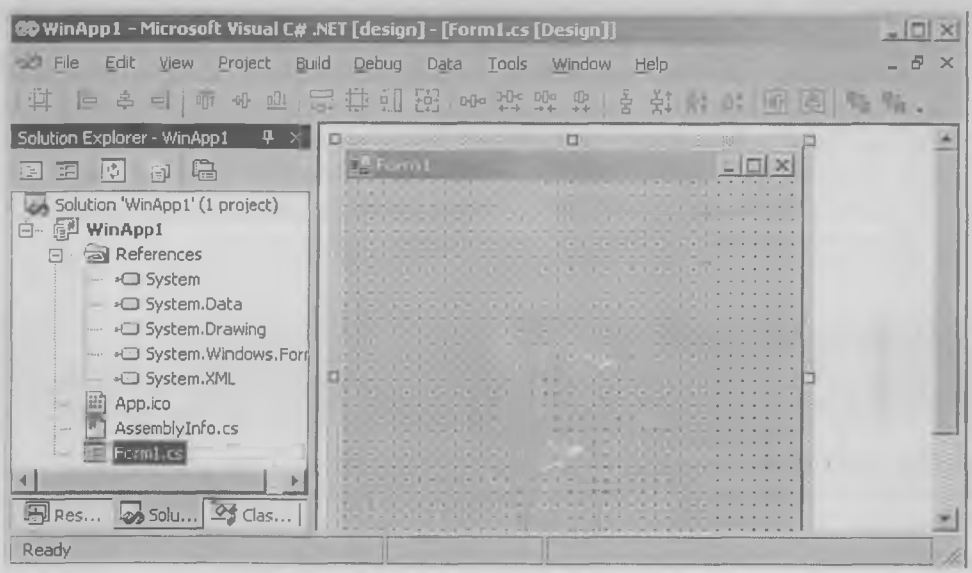


Рис. 8.15.

Из рис. 8.15 наглядно видно, что в случае проектов типа Windows Application ссылки на необходимые библиотечные файлы (References в левой части окна) устанавливаются автоматически. Кроме того, в проекте сразу же присутствует непустой файл Form1.cs с исходным кодом каркасного приложения, но по умолчанию в правой части окна компилятора показывается не сам исходный код на языке C#, а внешний вид главного окна приложения, которое соответствует этому коду. Такой режим показа называется *режимом оконного дизайнера* (отсюда и надпись [Form1.cs[Design]] в заголовке окна компилятора).

Чтобы наблюдать сам код на языке C#, нужно правой клавишей мыши щелкнуть по надписи Form1.cs в левой части окна компилятора и из всплывающего меню выбрать *строку View Code* (обратный переход выполняется по *строке меню View Designer*), после чего окно компилятора примет вид, показанный на рис. 8.16.

Действительно, теперь в правой части окна наблюдается автоматически сгенерированный графической средой компилятора исходный код на языке C#.

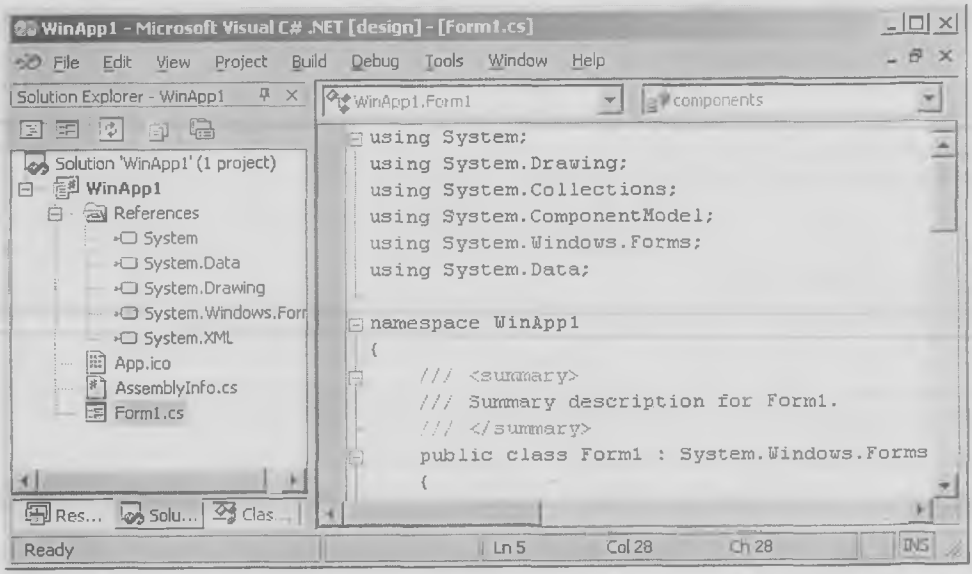


Рис. 8.16.

К его анализу приступим чуть позже, а сейчас выполним компиляцию проекта и запустим получающееся при этом приложение Windows, которое имеет главное окно произвольного стиля, показанное на рис. 8.17.

Рис. 8.17.

Это окно имеет рамку с возможностью изменения оконного размера, кнопки минимизации/максимизации, но более не содержит ничего. Ясно, что автоматический дизайнер окна, входящий в состав графической среды компилятора Mi-

crosoft Visual C#, не имеет никакой информации о наших конкретных намерениях, так что дальше нужно действовать самостоятельно.

Начнем с общего обзора исходного кода, сгенерированного автоматически (его частично можно наблюдать на рис. 8.16), из которого мы ради краткости удалим ничего для нас сейчас не значащий, но заметный объем *комментариев* (в общем случае безусловно полезных и даже нужных, так как они *помогают ориентироваться автоматическим генераторам кода*), пустых строк и специальных разметок, а также переставим местами пару функциональных методов:

Листинг 8.3.

```
/*************************/
/*    File Form1.cs      */
/*************************/
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WinApp1
{
  public class Form1 : System.Windows.Forms.Form
  {
    private System.ComponentModel.Container components = null;

    public Form1(){ InitializeComponent(); }
    private void InitializeComponent()
    {
      this.components = new System.ComponentModel.Container();
      this.Size = new System.Drawing.Size(300,300);
      this.Text = "Form1";
    }

    protected override void Dispose( bool disposing )
    {
      if( disposing )
       if( components != null )
        components.Dispose();

      base.Dispose( disposing );
    }
```

```
    [STAThread]
    public static void Main( )
    { Application.Run( new Form1() ); }
  }
}
```

Из Листинга 8.3 видно, что автоматический генератор кода сильно перестраховывается, используя одновременно и многочисленные директивы using, и полную квалификацию имен программных объектов, как префиксами с именами библиотечных пространств имен, так и *ключевым словом this* (означает, что последующее имя относится к объекту определяемого класса). К тому же, на всякий случай, разрабатываемый в проекте класс `Form1` помещается автоматическим генератором кода в собственное пространство имен `WinApp1`, что не всегда-то и нужно.

Чтобы не отвлекаться на перечисленные только что непринципиальные детали, уберем их из Листинга 8.3:

Листинг 8.4.

```
/*************************/
/*    File Form1.cs      */
/*************************/
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

public class Form1 : Form
{
  private Container components = null;

  public Form1(){ InitializeComponent(); }
  protected override void Dispose( bool disposing )
  {
    if( disposing )
     if( components != null )
      components.Dispose();

    base.Dispose( disposing );
  }
  private void InitializeComponent()
  {
    components = new Container();
```

```
    Size = new Size(300,300);
    Text = "Form1";
  }

  [STAThread]
  public static void Main()
  { Application.Run( new Form1() ); }
}
```

Добившись отсутствия в коде второстепенных деталей, обсудим теперь важные его черты. В отличие от построенного нами вручную Листинга 8.2, здесь определяется единственный класс `Form1`, который, таким образом, одновременно олицетворяет и главное окно приложения (за счет наследования от библиотечного оконного класса `Form`), и само приложение (за счет включения стартового метода `Main()`).

Для инициирования работы всех механизмов функционирования приложений с графическим интерфейсом пользователя (*запуск цикла выборки сообщений и показ главного окна приложения*) используется статический метод `Run()` библиотечного класса `Application`, которому в качестве параметра передается объект класса `Form1`, олицетворяющего приложение:

```
Application.Run( new Form1() );
```

Это исключительно простой код для запуска подобного рода приложений, ибо аналогичные программы на языках C/C++ на порядок (если не более) сложнее (см., например, книгу [2]).

В конструкторе класса `Form1` (точнее, в методе `InitializeComponent()`, вызываемом из конструктора) генератор кода настраивает размер окна и надпись в его заголовке.

Осталось сказать несколько слов про поле данных components библиотечного типа `Container`. Это контейнер (по внутренней логике управления элементами – «очередь»), предназначенный для хранения визуальных компонентов окон, требующих аккуратного освобождения занимаемых ими ресурсов. В библиотеке FCL такие компоненты реализуют интерфейс `IComponent`, и их как раз и нужно помещать на хранение в контейнер components. При уничтожении объекта типа `Form1` вызывается метод `Dispose()`, от которого требуется точная иерархическая последовательность освобождения ресурсов, связанная с его дочерними компонентами, что автоматически реализуется методом `Dispose()` объекта типа `Container`. В общем, это довольно сложная материя и мы не собираемся ее касаться в нашем приложении для начинающих, так что пусть этим элементом приложения лучше управляет автоматический генератор

кода, к услугам которого мы сейчас прибегнем, чтобы добавить на поверхность нашего окна две кнопки и снабдить их реакцией на нажатие.

Для этого описанным выше способом переходим к показу файла Form1.cs в ежиме *оконного дизайнера*, и с поверности *панели инструментов* (*Toolbox*) на поверхность нашего главного окна дважды перетаскиваем мышью (буксировка методом drag and drop) изображения кнопок (см. рис. 8.18).

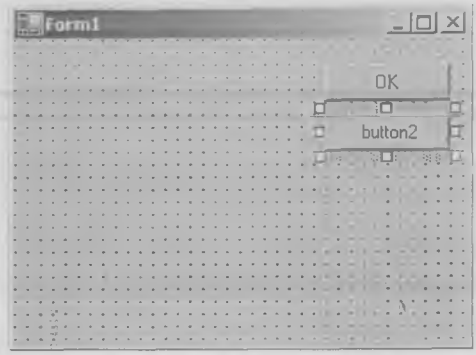


Рис. 8.18.

Далее кнопки располагаются на поверхности окна в положенном для них месте, им придается нужный размер (тянем мышью за один из восьми квадратиков – маркеров размера). Однократный щелчок мышью по кнопке (разумеется, левой клавишей мыши) и выбор строки Properties из всплывающего контекстного меню вызывает *окно свойств для этой кнопки* (см. рис. 8.19).

На рис. 8.19 показано, что оконный дизайнер автоматически присвоил новой кнопке надпись button2, а мы меняем здесь значение свойства Text на Cancel.

Осталось *связать новые кнопки с их функциональными обработчиками*, для чего нужно по каждой кнопке выполнить *двойной щелчок мышью*, после чего нам будет показан код *автоматически сгенерированного обработчика для этой кнопки*, пока что с пустым телом (см. рис. 8.20).

Из рис. 8.20 видно, что для кнопки с надписью Cancel (ее идентификатор как имя поля данных в классе `Form1` есть `button2`) сгенерирован функциональный обработчик события Click с именем `button2_Click()` и с пустым телом. Для большей наглядности приведем полностью состояние кода проекта на данной его стадии:

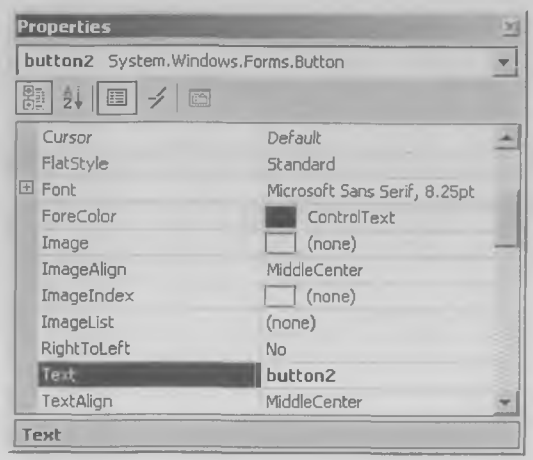


Рис. 8.19.

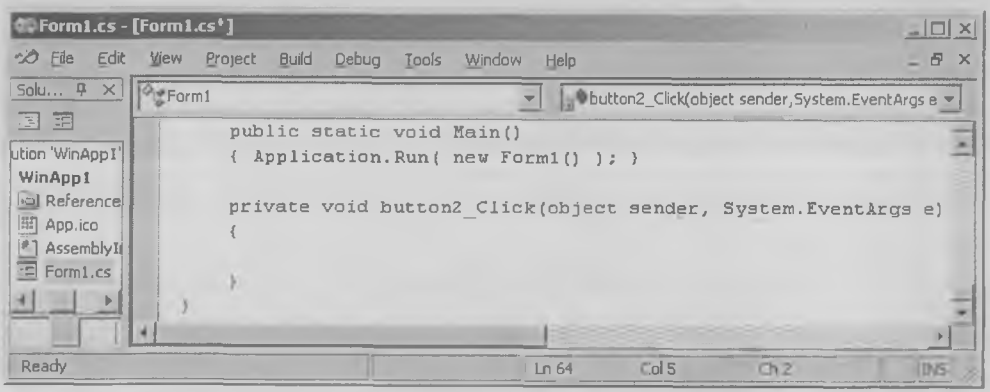


Рис. 8.20.

Листинг 8.5.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

```
public class Form1 : Form
{
  private System.Windows.Forms.Button button1;
  private System.Windows.Forms.Button button2;
  private Container components = null;

  public Form1(){ InitializeComponent(); }
  protected override void Dispose( bool disposing )
  {
    if( disposing )
     if( components != null )
      components.Dispose();

    base.Dispose( disposing );
  }
  private void InitializeComponent()
  {
    this.button1 = new System.Windows.Forms.Button();
    this.button2 = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // button1
    //
    this.button1.Location = new System.Drawing.Point(200, 16);
    this.button1.Name = "button1";
    this.button1.Size = new System.Drawing.Size(80, 24);
    this.button1.TabIndex = 0;
    this.button1.Text = "OK";
    this.button1.Click += new
                    System.EventHandler(this.button1_Click);
    //
    // button2
    //
    this.button2.Location = new System.Drawing.Point(200, 48);
    this.button2.Name = "button2";
    this.button2.Size = new System.Drawing.Size(80, 24);
    this.button2.TabIndex = 1;
    this.button2.Text = "Cancel";
    this.button2.Click += new
                    System.EventHandler(this.button2_Click);
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
```

```
    this.ClientSize = new System.Drawing.Size(292, 197);
    this.Controls.Add(this.button2);
    this.Controls.Add(this.button1);
    this.Name = "Form1";
    this.Text = "Form1";
    this.ResumeLayout(false);
  }

  [STAThread]
  public static void Main( )
  { Application.Run( new Form1() ); }

  private void button2_Click(object sender, System.EventArgs e)
  {
  }
  private void button1_Click(object sender, System.EventArgs e)
  {
  }
}
```

Так как поля данных `button1` и `button2`, а также содержимое метода `InitializeComponent()` в Листинг 8.5 вносятся автоматическим генератором кода оконного дизайнера, то снова восстановилась практика избыточных префиксов (полные идентификаторы пространств имен и ключевое слово `this`). Так как теперь они для нас никакой проблемы не представляют, то пусть себе стоят, а мы с ними бороться больше не будем.

В самом конце Листинга 8.5 прекрасно видны автоматически сгенерированные обработчики `button1_Click()` и `button2_Click()` с пустыми телами, в которые мы самостоятельно вносим собственный смысловой код, который, правда, пока что у нас очень тривиальный, сводящийся лишь к вызову статического метода `Show()` библиотечного класса `MessageBox`, то есть к показу стандартных окон сообщений операционной системы Windows:

```
private void button2_Click(object sender, System.EventArgs e)
{
  MessageBox.Show( "Cancel button", "Message" );
}
private void button1_Click(object sender, System.EventArgs e)
{
  MessageBox.Show( "OK button", "Message" );
}
```

Компилируем проект, запускаем исполняемый файл, и вот перед нами главное окно приложения (см. рис. 8.21).

Рис. 8.21.

Кнопки OK и Cancel работают так, как было нами запроектировано с помощью оконного дизайнера и собственного программирования тела обработчиков. Например, на рис. 8.22 показано окно сообщений, вызываемое в ответ на нажатие (мышью или клавиатурой) кнопки Cancel.

Рис. 8.22.

Подведем итоги. Применение встроенного в графическую среду компилятора Microsoft Visual C# дизайнера окон и автоматического кодогенератора обеспечивает возможность сверхнаглядной и чрезвычайно простой интерактивной работы, когда на всех стадиях разработки мы все время видим готовый зрительный образ проектируемого окна. Большинство действий совершается с помощью мыши, не нужно подбирать размеры вслепую, в целом цикл разработки от начального прототипа до финального варианта существенно ускоряется. Очень подробное и полезное рассмотрение практики работы с автоматизированными визуальными средствами среды программирования Microsoft Visual C# дано в книге [10]. По этой книге можно детально ознакомиться с построением графического интерфейса пользователя, содержащего все основные типы элементов управления. В ней также приведены полные примеры кода достаточно крупных Windows-приложений, в том числе приложений баз данных.

Заканчивая настоящий раздел, отметим, что несмотря на наличие автоматизированных средств быстрой разработки, умение работать вручную очень полезно, так как оно увеличивает уверенность программиста в своих силах. Поэтому, желательно твердо знать все базовые основы, но на практике активно применять автоматизирующие средства.

Бывают даже случаи, когда применение автоматизированных средств или невозможно, или предельно неудобно. Не будем сейчас сильно развивать эту тему, но в качестве примера приведем факт, что переделать только что разработанный с помощью оконного дизайнера проект с Листинга 8.5 под диалоговое окно в качестве главного окна приложения можно только вручную, правда очень легко. Нужно в Листинге 8.5 переписать код стартового метода `Main()`, используя знания, ранее полученные в разд. 8.3:

```
public static void Main( )
{
  //Application.Run( new Form1() );
  Form1 dlg = new Form1();

  // Дополнительная ручная настройка под диалоговое окно:
  dlg.MinimizeBox = false;
  dlg.MaximizeBox = false;
  dlg.FormBorderStyle = FormBorderStyle.FixedDialog;
  dlg.Text = "SimpleDialog";

  // Запускаем приложение на базе
  // диалогового окна:
  dlg.ShowDialog();
}
```

Так что возможен такой *комбинированный стиль разработки*: основная работа выполняется с помощью оконного дизайнера и автоматического кодогенератора, а потом отдельные участки кода поправляются вручную. Так, в частности, мы будем поступать уже в следующей главе, посвященной разработке быстродействующих компьютерных информационных систем, где воспользуемся и преимуществами работы с дизайнером окон и автоматической кодогенерацией, и ручным переводом приложения под диалоговое окно.

Последняя задача несколько искуственная, но в книге [1], на которую мы постоянно ссылаемся, разработка информационных систем велась именно на базе диалоговых окон (там этот поход диктовался чрезмерной сложностью разработки на языке C иных типов окон), и поэтому с целью скоррелировать наше в общем и целом параллельное изложение материала выбираем для следующей главы комбинированный стиль разработки.

Глава 9. Программирование быстродействующих информационных систем

9.1. База данных двоичного формата

В разд. 7.4 мы программировали справочно-информационную систему телефонных номеров с сохранением данных в файлах в текстовом формате. У такого решения есть как достоинства, так и недостатки. К достоинствам можно отнести простоту и универсальность, так как данные в эту систему вносятся с помощью сторонних текстовых редакторов и ими же могут просматриваться. Есть дополнительная возможность автоматического программного поиска (см. рис. 7.15).

Часть недостатков информационной системы текстового формата является продолжением ее достоинств. Например, внедрение данных сторонними текстовыми редакторами означает немонолитность такого решения. Кроме того, возможность просматривать содержимое системы обычными текстовыми редакторами сильно повышает риск непреднамеренной порчи данных.

Поэтому настоящие информационные системы, обычно называемые *базами данных*, опираются в своих методиках обработки и файлового хранения информации на те или иные *двоичные форматы данных*.

Основной целью данной главы является разработка быстродействующей информационной системы в форме базы данных телефонных номеров двоичного формата.

Аналогичная по назначению и функциональности база данных при программировании на языке C в книге [1] опиралась на структуру `TelPersonInfo`:

```
struct TelPersonInfo
{
  char Name[128];
  int  TelNum;
};
```

где поле `Name` использовалось для хранения фамилии абонента, а поле `TelNum` – для хранения номера телефона.

Отталкиваясь от такой структуры данных, создадим сейчас с той же самой целью одноименный класс языка C#:

```
class TelPersonInfo
{
  // Конструкторы класса TelPersonInfo:
```

```
  public TelPersonInfo( )
  { Name = null; TelNum = 0; }
  public TelPersonInfo( string nam )
  { m_Name = nam; m_TelNum = 0; }
  public TelPersonInfo( string nam, int num )
  { m_Name = nam; m_TelNum = num; }

  // Открытые свойства:
  public string Name
  {
    get { return m_Name;  }
    set { m_Name = value; }
  }
  public int TelNum
  {
    get { return m_TelNum;  }
    set { m_TelNum = value; }
  }

  // Закрытые поля данных:
  private string m_Name;
  private int    m_TelNum;
}
```

Здесь мы, опираясь на полученные выше в разд. 5.1 знания про классическую дилемму «открытые классовые поля данных или закрытые поля данных», реализовали в классе `TelPersonInfo` открытые свойства `Name` и `TelNum` при закрытых полях данных `m_Name` и `m_TelNum`.

Так как число абонентов в процессе сеанса работы может измениться в связи с добавлением новой информации, то нам нужно как-то работать с динамически формируемыми массивами элементов типа `TelPersonInfo`. Опыт работы с такими динамическими массивами у нас есть, так как в разд. 6.5 мы познакомились с библиотечными классами коллекций на примере контейнеров типа `ArrayList`. Это как раз то, что нам сейчас нужно.

После создания контейнера кодом

```
ArrayList arl = new ArrayList();
```

мы просто добавляем в него элементы типа `TelPersonInfo` на хранение с помощью метода `Add()` класса `ArrayList`:

```
TelPersonInfo tpi = new TelPersonInfo();
arl.Add( tpi );
```

Количество хранящихся в контейнере элементов в любой момент может быть определено с помощью выражением `tpi.Count`.

Доступ к полям i-ых элементов набора (контейнера) осуществляется с помощью следующих выражений:

```
(TelPersonInfo)arl[i]).Name
```

и

```
(TelPersonInfo)arl[i]).TelNum
```

Так как в контейнере типа `ArrayList` можно хранить элементы любых типов, то после извлечения i-го элемента нужно осуществить явное приведение этого элемента к его конкретному типу (в данном случае к типу `TelPersonInfo`), после чего только и можно обращаться к открытым свойствам `Name` и `TelNum` класса `TelPersonInfo`.

Теперь можно констатировать, что все необходимое для работы с несложными динамическими структурами данных нам уже известно, и перечисленные только что фрагменты кода будут активно использоваться дальше в настоящей главе.

9.2. Основные операции с базами данных

Разобравшись со структурами данных в памяти компьютера, сформулируем теперь основные операции, свойственные нашей базе данных с точки зрения работы с ней пользователя.

В нашей программе пользователь сможет выполнять две операции: запрашивать номер телефона по введенной с клавиатуры фамилии и вводить новые записи базы данных (пары значений «фамилия – телефон»).

Так как основной и тиражируемой операцией мы считаем именно запрос на нахождение телефонного номера (поиск), то нужно сделать все возможное, чтобы эта операция выполнялась как можно быстрее по времени. Для этого загруженная в память информация базы данных хранится в контейнере типа `ArrayList` в отсортированном по фамилиям (в алфавитном порядке) виде, так что для поиска можно применить быстродействующий метод `BinarySearch()` контейнерного класса `ArrayList`. Сортировку будет выполнять другой метод этого библиотечного класса – метод `Sort()`.

Применять перечисленные методы тривиально: с методом `BinarySearch()` мы знакомились еще в разд. 3.5 применительно к массивам языка C#, а метод `Sort()` применяли многократно в разных главах настоящего пособия, как для встроенных массивов языка C#, так и для контейнеров. Например, в разд. 6.3 мы

применяли метод Sort() для контейнеров с элементами классовых типов. И мы оттуда знаем, что для этого требуется, чтобы класс элементов контейнера реализовывал метод CompareTo() стандартного библиотечного интерфейса IComparable. Требование очевидное, ибо элементы контейнеров могут быть произвольных типов, так что именно класс элементов должен сравнивать элементы между собой.

Мы хотим сортировать записи базы данных по фамилиям, и для этого добавляем следующую реализацию интерфейсного метода CompareTo() в наш класс TelPersonInfo:

```
class TelPersonInfo : IComparable
{
  // Метод интерфейса IComparable:
  public int CompareTo( object ob )
  {
    if( String.Compare( m_Name,
                        ((TelPersonInfo)ob).m_Name ) > 0 )
      return 1;
    else if( String.Compare( m_Name,
                             ((TelPersonInfo)ob).m_Name ) == 0 )
      return 0;
    else
      return -1;
  }

  // Конструкторы класса TelPersonInfo:
  public TelPersonInfo( )
  { Name = null; TelNum = 0; }
  public TelPersonInfo( string nam )
  { m_Name = nam; m_TelNum = 0; }
  public TelPersonInfo( string nam, int num )
  { m_Name = nam; m_TelNum = num; }

  // Открытые свойства:
  public string Name
  {
    get { return m_Name;  }
    set { m_Name = value; }
  }
  public int TelNum
  {
    get { return m_TelNum;  }
    set { m_TelNum = value; }
  }
```

```
  // Закрытые поля данных:
  private string m_Name;
  private int    m_TelNum;
}
```

Мы уже имеем опыт практической реализации интерфейсного метода `CompareTo()` в Листинге 6.4 из разд. 6.3, так что сейчас стоит лишь отметить, что здесь мы для алфавитного сравнения двух строк (фамилии абонентов – это строки, а сортировать мы хотим именно по фамилиям) применяем статический метод `Compare()` библиотечного класса `String`.

Наш вариант интерфейсного метода `CompareTo()` заставит метод `Sort()` выполнять сортировку по алфавитному возрастанию. Для задачи выполнения быстрого поиска в базе данных телефонных номеров мы будем применять метод `BinarySearch()` контейнерного класса `ArrayList`, а ему требуется объект-контейнер, элементы которого отсортированы именно по возрастанию.

Разобравшись с окончательным вариантом класса `TelPersonInfo` и с нюансами работы с контейнером элементов этого типа, покажем код, который будет выполнять пользовательские операции вставки в базу данных новых записей и поиска телефона по заданной фамилии абонента. Так как эти операции будут инициироваться через элементы графического интерфейса пользователя, то весь этот код будет сосредоточен в обработчике нажатия на командную кнопку с надписью OK и с идентификатором `button1` (этот идентификатор нам «подарит» автоматизированный оконный дизайнер, к услугам которого мы прибегнем в следующем разделе):

```
private void button1_Click(object sender, System.EventArgs e)
{
  if( textBox1.Text != "" && textBox2.Text != "" ) // Вставка
  {
    int ind = ArL.BinarySearch(new TelPersonInfo(textBox1.Text));
    if( ind < 0 )  // Абонента в БД нет
    {
      ArL.Add( new TelPersonInfo( textBox1.Text,
                          Convert.ToInt32( textBox2.Text ) ) );
      ArL.Sort();
      textBox3.Text = "Запись внедрена";
    }
    else
      textBox3.Text = "Абонент уже присутствует в базе";
  }
  else if( textBox1.Text != "" && textBox2.Text == "" ) // Поиск
  {
```

```
    int ind = ArL.BinarySearch(new TelPersonInfo(textBox1.Text));
    if( ind >= 0 )
      textBox3.Text = Convert.ToString(
                              ((TelPersonInfo)ArL[ind]).TelNum );
    else
      textBox3.Text = "Абонент не найден";
  }
  else // Показать данные по первому (по алфавиту) абоненту
  {
    int num = ArL.Count;
    if( num > 0 )
    {
      textBox1.Text = ((TelPersonInfo)ArL[0]).Name;
      textBox2.Text = Convert.ToString(
                           ((TelPersonInfo)ArL[num - 1]).TelNum );
      textBox3.Text = "";
    }
  }
}
```

Автоматизированный оконный дизайнер поименует этот функциональный метод-обработчик именно как `button1_Click`.

Входные данные от пользователя будут формироваться как текст, введенный им с клавиатуры и сосредоточенный в редактируемых текстовых полях (элементах управления) `textBox1` (фамилия) и `textBox2` (номер телефона в текстовом виде). Третье текстовое поле – `textBox3`, мы приспособим для вывода информационных сообщений.

Сам текст из текстовых полей извлекается (и помещается туда же) с помощью выражений типа `textBox1.Text`, где `Text` – это свойство класса этих графических элементов управления. Они будут нами созданы с помощью автоматизированного дизайнера окон в следующем разделе главы.

9.3. Проектирование пользовательского интерфейса

Перейдем к вопросу о проектировании пользовательского интерфейса. Для нашей программы мы хотим использовать диалоговое окно с тремя *редактируемыми текстовыми полями* и двумя кнопками, не считая группирующей рамки, используемой как декоративный элемент оформления, улучшающий внешний облик окна.

Общий вид проектируемого окна показан на рис. 9.1.

Заголовок окна содержит надпись Telephons, совпадающую с именем файла базы данных. В группе Input data сосредоточены два редактируемых поля ввода.

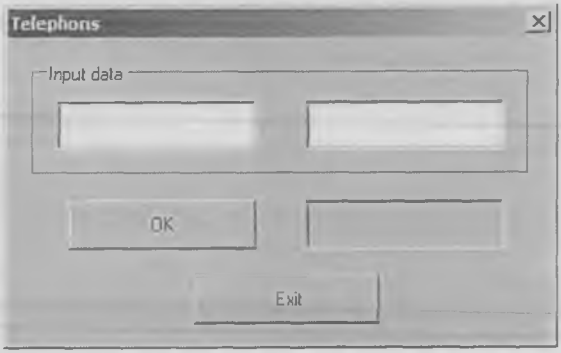


Рис. 9.1.

Левое из них предназначено для ввода фамилии абонента. Кнопка с надписью OK предназначена для инициирования процесса поиска телефона в случае пустого содержимого правого из этих редактируемых полей, и для инициирования добавления новой записи в базу данных в случае, когда правое редактируемое поле содержит телефонный номер.

Результат поиска телефонного номера, а также информационные сообщения о завершении операций демонстрируются в правом нижнем редактируемом поле, которое на самом деле уже не такое уж и редактируемое. Оно имеет серый фон поверхности, что означает, что данный элемент управления имеет атрибут *только для чтения* (см. рис. 9.2).

На рис. 9.2 показано, что для этого требуется в окне свойств этого элемента (вызывается по щелчку на элементе *правой клавиши мыши* и выборе позиции Properties всплывающего меню) отметить позицию ReadOnly как True (по умолчанию стоит False).

В таком случае редактируемый элемент не позволяет работать с ним в интерактивном режиме, так как ввод текста с клавиатуры для него заблокирован. Программным же образом в него можно вносить содержимое (текст) обычным образом:

```
textBox3.Text = "Абонент не найден";
```

Кстати, именно идентификатор `textBox3` присвоен этому редактируемому полю в нашем конкретном случае автоматизированным дизайнером окон. Идентификаторы `textBox1` и `textBox2` соответствуют редактируемым полям из группы, заключенной в группирующую рамку с надписью Input data. Командная кнопка с надписью OK получила идентификатор `button1`.

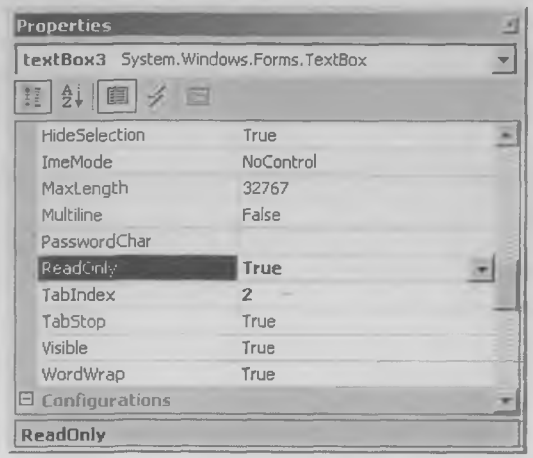


Рис. 9.2.

Приведем код файла `Form1.cs`, автоматически сгенерированный дизайнером окон, из которого мы выкинули метод `Main()` (мы его переводим в класс main из файла main.cs, о котором будет рассказано ниже в настоящей главе), а также наполнили кодом из предыдущего раздела тело обработчика нажатия кнопки `button1`:

Листинг 9.1.

```
/*************************/
/*     File Form1.cs     */
/*************************/
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace TelBookProj
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
```

```
private System.Windows.Forms.GroupBox groupBox1;
private System.Windows.Forms.TextBox textBox1;
private System.Windows.Forms.TextBox textBox2;
private System.Windows.Forms.Button button1;
private System.Windows.Forms.TextBox textBox3;
private System.Windows.Forms.Button button2;
/// <summary>
/// Required designer variable.
/// </summary>
private System.ComponentModel.Container components = null;

public Form1( ArrayList arl )
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();
  //
  // TODO: Add any code after InitializeComponent call
  //
  ArL = arl;
}

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
  if( disposing )
  {
    if (components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}

#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>
private void InitializeComponent()
{
```

```
this.groupBox1 = new System.Windows.Forms.GroupBox();
this.textBox2 = new System.Windows.Forms.TextBox();
this.textBox1 = new System.Windows.Forms.TextBox();
this.button1 = new System.Windows.Forms.Button();
this.textBox3 = new System.Windows.Forms.TextBox();
this.button2 = new System.Windows.Forms.Button();
this.groupBox1.SuspendLayout();
this.SuspendLayout();
//
// groupBox1
//
this.groupBox1.Controls.Add(this.textBox2);
this.groupBox1.Controls.Add(this.textBox1);
this.groupBox1.Location = new System.Drawing.Point(16, 16);
this.groupBox1.Name = "groupBox1";
this.groupBox1.Size = new System.Drawing.Size(320, 72);
this.groupBox1.TabIndex = 0;
this.groupBox1.TabStop = false;
this.groupBox1.Text = "Input data";
//
// textBox2
//
this.textBox2.AutoSize = false;
this.textBox2.Location = new System.Drawing.Point(176, 24);
this.textBox2.Name = "textBox2";
this.textBox2.Size = new System.Drawing.Size(128, 32);
this.textBox2.TabIndex = 1;
this.textBox2.Text = "";
//
// textBox1
//
this.textBox1.AutoSize = false;
this.textBox1.Location = new System.Drawing.Point(16, 24);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(128, 32);
this.textBox1.TabIndex = 0;
this.textBox1.Text = "";
//
// button1
//
this.button1.Location = new System.Drawing.Point(40, 104);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(120, 32);
```

```
        this.button1.TabIndex = 1;
        this.button1.Text = "OK";
this.button1.Click += new System.EventHandler(this.button1_Click);
        //
        // textBox3
        //
        this.textBox3.AutoSize = false;
        this.textBox3.Location = new System.Drawing.Point(192, 104);
        this.textBox3.Name = "textBox3";
        this.textBox3.ReadOnly = true;
        this.textBox3.Size = new System.Drawing.Size(128, 32);
        this.textBox3.TabIndex = 2;
        this.textBox3.Text = "";
        //
        // button2
        //
        this.button2.DialogResult =
                        System.Windows.Forms.DialogResult.Cancel;
        this.button2.Location = new System.Drawing.Point(120, 152);
        this.button2.Name = "button2";
        this.button2.Size = new System.Drawing.Size(120, 32);
        this.button2.TabIndex = 3;
        this.button2.Text = "Exit";
        //
        // Form1
        //
        this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
        this.BackColor = System.Drawing.SystemColors.Control;
        this.ClientSize = new System.Drawing.Size(352, 197);
        this.Controls.Add(this.button2);
        this.Controls.Add(this.textBox3);
        this.Controls.Add(this.button1);
        this.Controls.Add(this.groupBox1);
        this.Name = "Form1";
        this.Text = "Form1";
        this.groupBox1.ResumeLayout(false);
        this.ResumeLayout(false);
    }
    #endregion

    private void button1_Click(object sender, System.EventArgs e)
    {
      if( textBox1.Text != "" && textBox2.Text != "" ) //Вставка
      {
```

```
        int ind = ArL.BinarySearch( new
                                  TelPersonInfo(textBox1.Text));
        if( ind < 0 )  // Абонента в БД нет
        {
          ArL.Add( new TelPersonInfo( textBox1.Text,
                          Convert.ToInt32( textBox2.Text ) ) );
          ArL.Sort();
          textBox3.Text = "Запись внедрена";
        }
        else
          textBox3.Text = "Абонент уже присутствует в базе";
      }
      else if( textBox1.Text != "" && textBox2.Text == "" )
      { // Поиск
        int ind = ArL.BinarySearch( new
                                  TelPersonInfo(textBox1.Text));
        if( ind >= 0 )
          textBox3.Text = Convert.ToString(
                            ((TelPersonInfo)ArL[ind]).TelNum );
        else
          textBox3.Text = "Абонент не найден";
      }
      else // Показать данные по первому (по алфавиту) абоненту
      {
        int num = ArL.Count;
        if( num > 0 )
        {
          textBox1.Text = ((TelPersonInfo)ArL[0]).Name;
          textBox2.Text = Convert.ToString(
                         ((TelPersonInfo)ArL[num - 1]).TelNum );
          textBox3.Text = "";
        }
      }
    }

    // Собственное поле данных (не от оконного дизайнера):
    ArrayList ArL;
  }
}
```

Еще очень важно отметить, что в Листинге 9.1 мы вручную внесли в класс `Form1` поле данных `arL` типа `ArrayList` для связи с функционирующей в памяти компьютера базой данных. Первоначально ссылка на эту базу данных устанавливается в конструкторе класса `Form1`:

```
//
// TODO: Add any code after InitializeComponent call
//
ArL = arl;
```

в точном соответствии с комментарием от дизайнера окон, указывающим, что именно в это место следует вносить уникальную для конкретных программ пользовательскую инициализацию добавочных полей данных класса.

Все остальное в Листинге 9.1 «натворил» автоматизированный дизайнер окон, и мы это оставили здесь в неизменном виде, хотя в разд. 8.4, при первоначальном знакомстве с творчеством оконного дизайнера, мы старались «причесывать» такой код, выкидывая некоторые излишние элементы. Сейчас в этом уже нет никакой необходимости.

9.4. База данных и дисковые файлы

Помимо работы с данными в памяти компьютера, нужно уметь записывать всю имеющуюся информацию в двоичный файл базы данных, а также нужно уметь полностью вычитывать ее оттуда в память компьютера. Назовем файл проектируемой базы данных телефонных номеров именем `Telephons`.

Базовые сведения по работе с двоичными файлами известны нам из гл. 7, и мы их сейчас применим для написания функциональных методов для сохранения на диске информации из базы данных и чтения ее оттуда. Указанные методы будут сформированы как статические методы `PutDataIntoDB()` и `GetDataFromDB()` класса `MyFiles`:

Листинг 9.2.

```
/*************************/
/*    File MyFiles.cs    */
/*************************/
using System;
using System.Collections;
using System.IO;

class MyFiles
{
  public static void PutDataIntoDB(string FileName,ArrayList arl)
  {
    int size, len, Ind = 0; string strAux;

    // 1. Подготовка массива chAr элементов типа char
    // для последующей быстрой записи в файл:
    size = arl.Count;
```

```
    char[] chAr = new char[ (128+7) * size ];
    strAux = new String( ' ', (128+7) * size );
    strAux.CopyTo( 0, chAr, 0, (128+7) * size );

    // Заносим в цикле информацию в массив chAr
    // из контейнера arl:
    for( int i =0; i < size; ++i )
    {
      // Сначала - имя абонента:
      strAux = ((TelPersonInfo)arl[i]).Name;
      len = strAux.Length;
      strAux.CopyTo( 0, chAr, Ind, len );
      Ind += 128;

      // Затем - номер телефона:
      strAux = Convert.ToString(
                              ( (TelPersonInfo)arl[i]).TelNum );
      len = strAux.Length;
      strAux.CopyTo( 0, chAr, Ind, len );
      Ind += 7;
    }

    // 2. Запись массива chAr в файл:
    FileStream fsW = new FileStream( FileName, FileMode.Create,
                                     FileAccess.Write );
    BinaryWriter bw = new BinaryWriter( fsW );
    ////////////////////
    bw.Write( chAr );
    ////////////////////
    bw.Close();
    fsW.Close();
  }

  public static int GetDataFromDB(string FileName,ArrayList arl)
  {
    int TargetNum, len, Ind = 0; string TargetStr; char[] inBuf;

    if( File.Exists( FileName ) )
    {
      FileStream fsR = new FileStream( FileName, FileMode.Open,
                                       FileAccess.Read );
      len = (int)fsR.Length;
      if( len > 0 )
      {
        BinaryReader br = new BinaryReader( fsR );
        ////////////////////////////
```

```
            inBuf = br.ReadChars( len );
            ////////////////////////////
            br.Close();
            fsR.Close();

            // Формирование контейнера типа ArrayList:
            char[] NameBuf = new char[128];
            char[] NumBuf = new char[7];
            while( Ind < inBuf.Length )
            {
              // Сначала - имя абонента:
              Array.Copy( inBuf, Ind, NameBuf, 0, 128 );
              TargetStr = ( new string( NameBuf ) ).TrimEnd();
              Ind += 128;

              // Затем - номер телефона:
              Array.Copy( inBuf, Ind, NumBuf, 0, 7 );
              TargetNum = Convert.ToInt32(
                                ( new string(NumBuf) ).TrimEnd() );
              Ind += 7;

              // Заносим новый элемент в контейнер:
              arl.Add( new TelPersonInfo( TargetStr, TargetNum ) );
            }

            return 0;  // успех
          }

          fsR.Close();
        }

        return -7; // файл прочитать не удалось
      }
    }
```

На первый взгляд, методы `PutDataIntoDB()` и `GetDataFromDB()` получились довольно-таки сложными (и это действительно так). Объясняется это тем, что мы хотим оптимизировать операции записи в файл базы данных и полного ее вычитывания оттуда в память компьютера. Известно, что последовательное поэлементное чтение файла (или поэлементная запись в файл) при больших размерах базы данных представляет собой очень медленную операцию. Чтобы ее убыстрить, нужно воспользоваться однократными вызовами метода `Write()` библиотечного класса `BinaryWriter` и метода `ReadChars()` библиотечного класса `BinaryReader`. Но загвоздка заключается в том, что эти методы работают только с массивами элементов типа `char`, осуществляя запись в файл всего

такого массива целиком или его полное вычитывание оттуда в массив указанного типа, соответственно.

По этой причине нам и пришлось как следует «покрутиться» в коде Листинга 9.2: переводить все содержимое контейнера в элементы массива типа char, а также выполнять обратное действие.

Ничего этого делать не пришлось бы, если бы мы были знакомы с таким стержневым механизмом платформы Microsoft NET Framework, как *сериализация* (см. [5]). Но всего за один раз изучить в книге для начинающих невоможно, и поэтому механизм сериализации классов языка C# в нашем пособии не изучается. Но нет худа без добра – теперь у нас появился хороший учебный тренировочный пример.

Разберем основные детали кода по преобразованию набора элементов типа `TelPersonInfo` в значения элементов массива типа `char` (а обратное преобразование, которое требуется для программирования метода `GetDataFromDB()`, оставим читателю в качестве самостоятельного упражнения):

```
// 1. Подготовка массива chAr элементов типа char
// для последующей быстрой записи в файл:
size = arl.Count;
char[] chAr = new char[ (128+7) * size ];
strAux = new String( ' ', (128+7) * size );
strAux.CopyTo( 0, chAr, 0, (128+7) * size );

// Заносим в цикле информацию в массив chAr
// из контейнера arl:
for( int i =0; i < size; ++i )
{
  // Сначала - имя абонента:
  strAux = ((TelPersonInfo)arl[i]).Name;
  len = strAux.Length;
  strAux.CopyTo( 0, chAr, Ind, len );
  Ind += 128;

  // Затем - номер телефона:
  strAux = Convert.ToString(
                              ( (TelPersonInfo)arl[i]).TelNum );
  len = strAux.Length;
  strAux.CopyTo( 0, chAr, Ind, len );
  Ind += 7;
}
```

В книге [1] зафиксировано, что фамилию абонента мы ограничиваем 128 символами, а телефонные номера у нас семизначные. Поэтому в символьном

представлении суммарно под фамилию и номер нужно отвести по 128+7 позиций на каждую запись базы данных, после чего умножить это число на общее число элементов:

```
size = arl.Count;
char[] chAr = new char[ (128+7) * size ];
strAux = new String( ' ', (128+7) * size );
strAux.CopyTo( 0, chAr, 0, (128+7) * size );
```

Так как фамилии бывают разной длины, то символьный массив заполняется пробелами, для чего сначала с помощью удобного конструктора класса `String` из заданного количества пробелов формируется вспомогательная строка `strAux`, а затем методом `CopyTo()` того же класса все символы этой строки копируются в символьный массив `chAr`.

Далее, этот массив позиционно (точно в нужных позициях) прописывается символьными представлениями фамилий и номеров телефонов из базы данных, то есть из содержимого контейнера `arl` типа `ArrayList`:

```
for( int i =0; i < size; ++i )
{
  // Сначала - имя абонента:
  strAux = ((TelPersonInfo)arl[i]).Name;
  len = strAux.Length;
  strAux.CopyTo( 0, chAr, Ind, len );
  Ind += 128;

  // Затем - номер телефона:
  strAux = Convert.ToString(
                              ( (TelPersonInfo)arl[i]).TelNum );
  len = strAux.Length;
  strAux.CopyTo( 0, chAr, Ind, len );
  Ind += 7;

}
```

Строгая позиционность записи обеспечивается управляемыми скачками целочисленного индекса `Ind`:

```
Ind += 128;
```

и

```
Ind += 7;
```

После первого из этих скачков мы находимся в позиции для записи набора символов для телефонного номера, а после второго скачка (на 7 единиц) — для записи фамилии.

9.5. Сборка из разрозненных подпроектов целевого проекта «База данных телефонных номеров»

Нам осталось все, что мы ранее в данной главе обговаривали или писали в виде образцов, а также разработали окончательно в виде Листингов 9.1 и 9.2, собрать воедино. Но, самое главное, нам нужно еще написать в рамках исходного файла main.cs полные тексты класса main со стартовым методом `Main()` и скрепляющими все приложение статическими полями данных, содержащими имя файла базы данных и контейнер типа `ArrayList`, хранящий информацию об абонентах в памяти компьютера.

В графической среде компилятора Microsoft Visual C# создаем проект типа Windows Application с именем TelBookProj, настраиваем графический интерфейс пользователя с помощью автоматизированного оконного дизайнера и небольшой ручной правки так, как описано выше в разд. 9.3 (это влияет на содержимое файла Form1.cs), и дополнительно внедряем в него файлы main.cs, TelPersonInfo.cs и MyFiles.cs. На этой стадии проекта главное окно компилятора выглядит так, как показано на рис. 9.3.

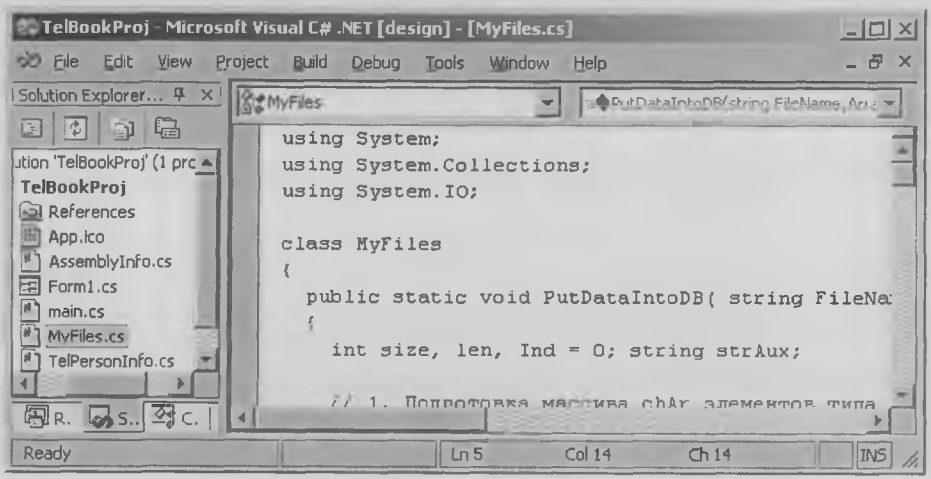


Рис. 9.3.

Хотя выше мы и представили основные фрагменты всех участвующих в данном проекте файлов, а окончательные варианты файлов Form1.cs и MyFiles.cs вообще даны в Листингах 9.1 и 9.2 полностью, мы все равно их здесь повторим для большей наглядности и точности:

Листинг 9.3.

```
/***************************/
/*  File TelPersonInfo.cs  */
/***************************/
using System;

class TelPersonInfo : IComparable
{
  // Метод интерфейса IComparable:
  public int CompareTo( object ob )
  {
    if( String.Compare( m_Name,
                        ((TelPersonInfo)ob).m_Name ) > 0 )
      return 1;
    else if( String.Compare( m_Name,
                             ((TelPersonInfo)ob).m_Name ) == 0 )
      return 0;
    else
      return -1;
  }

  // Конструкторы класса TelPersonInfo:
  public TelPersonInfo( )
  { Name = null; TelNum = 0; }
  public TelPersonInfo( string nam )
  { m_Name = nam; m_TelNum = 0; }
  public TelPersonInfo( string nam, int num )
  { m_Name = nam; m_TelNum = num; }

  // Открытые свойства:
  public string Name
  {
    get { return m_Name;  }
    set { m_Name = value; }
  }
  public int TelNum
  {
    get { return m_TelNum;  }
    set { m_TelNum = value; }
  }

  // Закрытые поля данных:
  private string m_Name;
  private int    m_TelNum;
}
```

```
/*************************/
/*    File MyFiles.cs    */
/*************************/
using System;
using System.Collections;
using System.IO;

class MyFiles
{
  public static void PutDataIntoDB(string FileName,ArrayList arl)
  {
    int size, len, Ind = 0; string strAux;

    // 1. Подготовка массива chAr элементов типа char
    // для последующей быстрой записи в файл:
    size = arl.Count;
    char[] chAr = new char[ (128+7) * size ];
    strAux = new String( ' ', (128+7) * size );
    strAux.CopyTo( 0, chAr, 0, (128+7) * size );

    // Заносим в цикле информацию в массив chAr
    // из контейнера arl:
    for( int i =0; i < size; ++i )
    {
      // Сначала - имя абонента:
      strAux = ((TelPersonInfo)arl[i]).Name;
      len = strAux.Length;
      strAux.CopyTo( 0, chAr, Ind, len );
      Ind += 128;

      // Затем - номер телефона:
      strAux = Convert.ToString(
                          ( (TelPersonInfo)arl[i]).TelNum );
      len = strAux.Length;
      strAux.CopyTo( 0, chAr, Ind, len );
      Ind += 7;
    }

    // 2. Запись массива chAr в файл:
    FileStream fsW = new FileStream( FileName, FileMode.Create,
                                     FileAccess.Write );
    BinaryWriter bw = new BinaryWriter( fsW );
    ////////////////////
    bw.Write( chAr );
    ////////////////////
```

```
    bw.Close();
    fsW.Close();
  }
  public static int GetDataFromDB(string FileName,ArrayList arl)
  {
    int TargetNum, len, Ind = 0; string TargetStr; char[] inBuf;
    if( File.Exists( FileName ) )
    {
      FileStream fsR = new FileStream( FileName, FileMode.Open,
                                       FileAccess.Read );
      len = (int)fsR.Length;
      if( len > 0 )
      {
        BinaryReader br = new BinaryReader( fsR );
        ////////////////////////////
        inBuf = br.ReadChars( len );
        ////////////////////////////
        br.Close();
        fsR.Close();
        // Формирование контейнера типа ArrayList:
        char[] NameBuf = new char[128];
        char[] NumBuf = new char[7];
        while( Ind < inBuf.Length )
        {
          // Сначала - имя абонента:
          Array.Copy( inBuf, Ind, NameBuf, 0, 128 );
          TargetStr = ( new string( NameBuf ) ).TrimEnd();
          Ind += 128;
          // Затем - номер телефона:
          Array.Copy( inBuf, Ind, NumBuf, 0, 7 );
          TargetNum = Convert.ToInt32(
                          ( new string(NumBuf) ).TrimEnd() );
          Ind += 7;
          // Заносим новый элемент в контейнер:
          arl.Add( new TelPersonInfo( TargetStr, TargetNum ) );
        }
        return 0;  // успех
      }
      fsR.Close();
    }
```

```
    return -7; // файл прочитать не удалось
  }
}

/*************************/
/*      File Form1.cs     */
/*************************/
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace TelBookProj
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>
  public class Form1 : System.Windows.Forms.Form
  {
    private System.Windows.Forms.GroupBox groupBox1;
    private System.Windows.Forms.TextBox textBox1;
    private System.Windows.Forms.TextBox textBox2;
    private System.Windows.Forms.Button button1;
    private System.Windows.Forms.TextBox textBox3;
    private System.Windows.Forms.Button button2;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container components = null;

    public Form1( ArrayList arl )
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
      //
      // TODO: Add any code after InitializeComponent call
      //
      ArL = arl;
    }
```

```
/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
  if( disposing )
  {
    if (components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}

#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>
private void InitializeComponent()
{
  this.groupBox1 = new System.Windows.Forms.GroupBox();
  this.textBox2 = new System.Windows.Forms.TextBox();
  this.textBox1 = new System.Windows.Forms.TextBox();
  this.button1 = new System.Windows.Forms.Button();
  this.textBox3 = new System.Windows.Forms.TextBox();
  this.button2 = new System.Windows.Forms.Button();
  this.groupBox1.SuspendLayout();
  this.SuspendLayout();
  //
  // groupBox1
  //
  this.groupBox1.Controls.Add(this.textBox2);
  this.groupBox1.Controls.Add(this.textBox1);
  this.groupBox1.Location = new System.Drawing.Point(16, 16);
  this.groupBox1.Name = "groupBox1";
  this.groupBox1.Size = new System.Drawing.Size(320, 72);
  this.groupBox1.TabIndex = 0;
  this.groupBox1.TabStop = false;
  this.groupBox1.Text = "Input data";
  //
  // textBox2
  //
```

```
        this.textBox2.AutoSize = false;
        this.textBox2.Location = new System.Drawing.Point(176, 24);
        this.textBox2.Name = "textBox2";
        this.textBox2.Size = new System.Drawing.Size(128, 32);
        this.textBox2.TabIndex = 1;
        this.textBox2.Text = "";
        //
        // textBox1
        //
        this.textBox1.AutoSize = false;
        this.textBox1.Location = new System.Drawing.Point(16, 24);
        this.textBox1.Name = "textBox1";
        this.textBox1.Size = new System.Drawing.Size(128, 32);
        this.textBox1.TabIndex = 0;
        this.textBox1.Text = "";
        //
        // button1
        //
        this.button1.Location = new System.Drawing.Point(40, 104);
        this.button1.Name = "button1";
        this.button1.Size = new System.Drawing.Size(120, 32);
        this.button1.TabIndex = 1;
        this.button1.Text = "OK";
this.button1.Click += new System.EventHandler(this.button1_Click);
        //
        // textBox3
        //
        this.textBox3.AutoSize = false;
        this.textBox3.Location = new System.Drawing.Point(192, 104);
        this.textBox3.Name = "textBox3";
        this.textBox3.ReadOnly = true;
        this.textBox3.Size = new System.Drawing.Size(128, 32);
        this.textBox3.TabIndex = 2;
        this.textBox3.Text = "";
        //
        // button2
        //
        this.button2.DialogResult =
                        System.Windows.Forms.DialogResult.Cancel;
       this.button2.Location = new System.Drawing.Point(120, 152);
        this.button2.Name = "button2";
        this.button2.Size = new System.Drawing.Size(120, 32);
        this.button2.TabIndex = 3;
```

```
    this.button2.Text = "Exit";
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.BackColor = System.Drawing.SystemColors.Control;
    this.ClientSize = new System.Drawing.Size(352, 197);
    this.Controls.Add(this.button2);
    this.Controls.Add(this.textBox3);
    this.Controls.Add(this.button1);
    this.Controls.Add(this.groupBox1);
    this.Name = "Form1";
    this.Text = "Form1";
    this.groupBox1.ResumeLayout(false);
    this.ResumeLayout(false);
  }
  #endregion

  private void button1_Click(object sender, System.EventArgs e)
  {
    if( textBox1.Text != "" && textBox2.Text != "" ) //Вставка
    {
      int ind = ArL.BinarySearch( new
                                  TelPersonInfo(textBox1.Text));
      if( ind < 0 )  // Абонента в БД нет
      {
        ArL.Add( new TelPersonInfo( textBox1.Text,
                        Convert.ToInt32( textBox2.Text ) ) );
        ArL.Sort();
        textBox3.Text = "Запись внедрена";
      }
      else
        textBox3.Text = "Абонент уже присутствует в базе";
    }
    else if( textBox1.Text != "" && textBox2.Text == "" )
    { // Поиск
      int ind = ArL.BinarySearch( new
                                  TelPersonInfo(textBox1.Text));
      if( ind >= 0 )
        textBox3.Text = Convert.ToString(
                          ((TelPersonInfo)ArL[ind]).TelNum );
      else
        textBox3.Text = "Абонент не найден";
    }
```

```
      else // Показать данные по первому (по алфавиту) абоненту
      {
        int num = ArL.Count;
        if( num > 0 )
        {
          textBox1.Text = ((TelPersonInfo)ArL[0]).Name;
          textBox2.Text = Convert.ToString(
                          ((TelPersonInfo)ArL[num - 1]).TelNum );
          textBox3.Text = "";
        }
      }
    }

    // Собственное поле данных (не от оконного дизайнера):
    ArrayList ArL;
  }
}

/*************************/
/*     File main.cs      */
/*************************/
using System;
using System.Collections;
using System.Windows.Forms;
using TelBookProj;

class main
{
  // Статические поля данных -
  // - контейнер с информацией и имя файла:
  static ArrayList arl = new ArrayList();
  static string FileName = "Telephons";

  [STAThread]
  public static void Main( )
  {
    // Готовим контейнер с данными к работе:
    int ret = MyFiles.GetDataFromDB( FileName, arl );
    if( ret == -7 )
      MessageBox.Show( "Нет файла БД", "ИНФОРМАЦИЯ" );

    //Application.Run( new Form1() );
    Form1 dlg = new Form1( arl );
```

```
        // Дополнительная ручная настройка под диалоговое окно:
        dlg.MinimizeBox = false;
        dlg.MaximizeBox = false;
        dlg.FormBorderStyle = FormBorderStyle.FixedDialog;
        dlg.Text = "Telephons";

        // Запускаем приложение на базе
        // диалогового окна:
        dlg.ShowDialog();

        // Сбрасываем текущую БД из памяти в файл:
        MyFiles.PutDataIntoDB( FileName, ar1 );
    }
}
```

Наверное, Листинг 9.3 самый большой по объему во всей книге. Тем не менее, почти все его детали должны быть читателям понятны (после внимательного изучения, конечно), поскольку все принципиальные моменты в настоящей главе уже были рассмотрены ранее.

Поясним лишь новые моменты, относящиеся к файлу main.cs. Во-первых, в этот файл мы добавили директиву

```
using TelBookProj;
```

поскольку дизайнер окон сгенерировал класс `Form1` в пространстве имен `TelBookProj`.

Во-вторых, здесь мы также, как и в конце разд. 8.4, код метода `Main()` в варианте от оконного дизайнера, то есть строку вида

```
Application.Run( new Form1() );
```

переделали вручную с тем, чтобы наше приложение базировалось на диалоговом окне (для полной аналогии с разработкой аналогичной базы данных из гл. 9 книги [1]):

```
Form1 dlg = new Form1( ar1 );

// Дополнительная ручная настройка под диалоговое окно:
dlg.MinimizeBox = false;
dlg.MaximizeBox = false;
dlg.FormBorderStyle = FormBorderStyle.FixedDialog;
dlg.Text = "Telephons";

// Запускаем приложение на базе
// диалогового окна:
dlg.ShowDialog();
```

Ну и, наконец, самое главное, на что следует обратить внимание в файле main.cs – это статические поля данных класса `main`:

```
static ArrayList arl = new ArrayList();
static string FileName = "Telephons";
```

через которые в приложении идет обращение к базе данных телефонных номеров, соответственно в памяти компьютера и на дисковом файле.

В памяти компьютера вся информация хранится в контейнере типа `ArrayList`, автоматически создающемся при старте приложения (до начала выполнения метода `Main()`). Далее метод `Main()` вычитывает в этот контейнер информацию из файла базы данных на диске:

```
int ret = MyFiles.GetDataFromDB( FileName, arl );
if( ret == -7 )
  MessageBox.Show( "Нет файла БД", "ИНФОРМАЦИЯ" );
```

причем, если этот файл открыть не удается (нет файла на диске в конкретный момент времени, или в самом начале работы пользователя с нашей программой), то выдается диагностическое сообщение и программа начинает работать с пустым контейнером.

В процессе работы записи могут добавляться (они при этом тут же сортируются), так что содержимое контейнера в принципе подвержено изменениям в течение сеанса работы с программой, но после того, как диалоговое окно по команде пользователя закрывается, метод `Main()` с помощью статического метода `PutDataIntoDB()` класса `MyFiles` перезаписывает файл базы данных новым (актуальным) содержимым:

```
// Сбрасываем текущую БД из памяти в файл:
MyFiles.PutDataIntoDB( FileName, arl );
```

и на этом наша программа завершает работу.

Основные операции базы данных – поиск телефона по запросу и внедрение в базу данных новых записей осуществляется при обработке сообщения от кнопки с надписью OK.

Например, на рис. 9.4 показано диалоговое окно нашего приложения, когда мы ввели новую фамилию (Браун) и номер телефона (1234568) и нажали кнопку OK. В результате в нижнем информационном поле появилась сигнальная надпись «Запись внедрена».

Успешное внедрение происходит тогда, когда оба редактируемых поля из группы «Input data» непустые. Кроме того, требуется, чтобы вводимая фамилия в текущей базе данных отсутствовала. Если, все же, попытаться повторно ввести

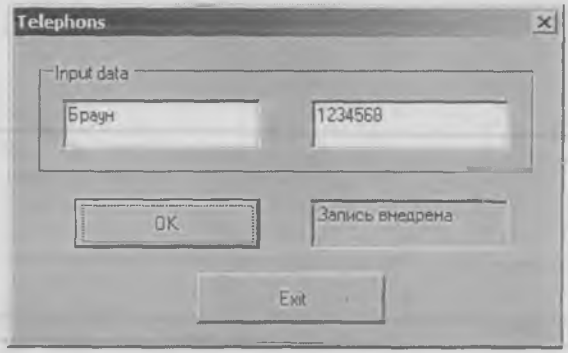


Рис. 9.4.

некоторый номер телефона для уже зарегистрированного абонента, то в информационном поле появится надпись «Повтор запрещен», сигнализирующая о неуспехе операции.

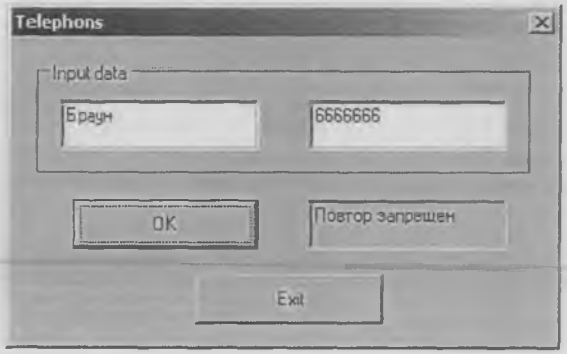


Рис. 9.5.

При пустом правом поле, предназначенном для ввода телефона нового абонента, нажатие на кнопку OK инициирует запрос телефона абонента с фамилией из левого верхнего редактируемого поля. В случае успеха найденный номер телефона демонстрируется в нижнем информационном поле (см. рис. 9.6).

На рис. 9.5 показано, что для фамилии Смит найден номер телефона 1234567.

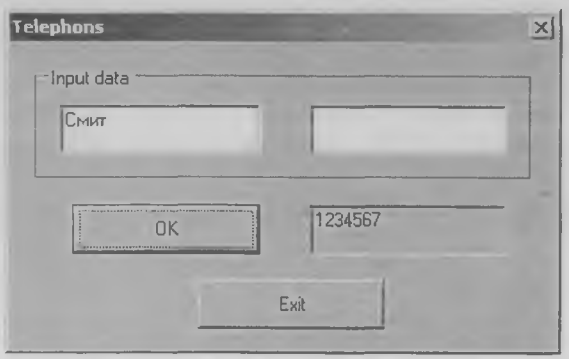


Рис. 9.6.

В случае, если запрос номера телефона по заданной фамилии абонента не увенчался успехом, то в нижнее информационное поле, как это показано на рис. 9.7, выводится сигнальная надпись «Абонент не найден».

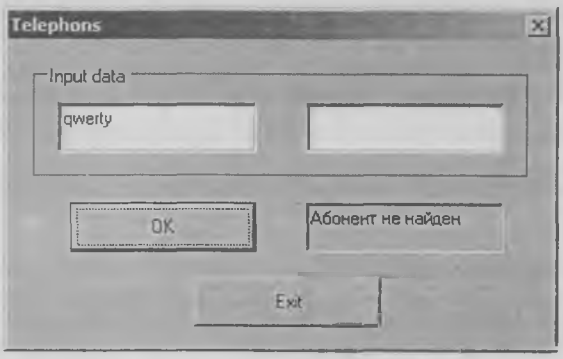


Рис. 9.7.

Программа с Листинга 9.3 получилась достаточно большой и свидетельствует о том, что читателями книги от ее начала и до конца пройден весьма большой путь. Если вы в состоянии полностью разобраться во всех нюансах программы с Листинга 9.3, то можете считать, что освоили программирование на языке C# для платформы Microsoft NET Framework операционной системы Windows на полупрофессиональном уровне.

Чтобы не удлинять текст программы с Листинга 9.3 еще больше, мы не поместили в нее абсолютно необходимые для практики проверки корректности вво-

да данных. Из-за этого возможна неприятная ситуация, когда вместо реальных цифр номеров телефона пользователь программы может ввести какую-нибудь абракадабру, в результате чего будет сгенерировано программное исключение, а сама программа будет принудительно закрыта. Но это наше сознательное решение для учебной книги ограниченного объема. Вы же можете самостоятельно улучшить практическое функционирование программы в самых разных направлениях, например, улучшив информационную насыщенность диалогового окна, выведя общее количество записей в базе данных в предназначенное для этого место на его поверхности.

Рассмотренное в данной главе пособия проектное решение с прямолинейной сортировкой записей не является единственно возможным и оптимальным. Более серьезное, хотя и по-прежнему учебное, решение в виде индексируемой базы данных, когда сортируется только специальный индекс (массив указателей на записи базы данных), а не сами записи, реализовано на языке C++ в монографии автора [2].

Приложение. Среда разработки Microsoft Developer Studio NET (2003) и компиляторы Microsoft Visual C# NET (2003) и Microsoft Visual C++ NET (2003)

Графическая среда разработки Microsoft Developer Studio NET (2003) (или эквивалентная ей среда выпуска 2005 года) является мультиязыковой средой разработки как NET-приложений, так и традиционных приложений Windows. Нас здесь интересуют лишь два языка программирования – C# и C++ (а также C). Программирование на этих языках осуществляется в рамках Microsoft Developer Studio NET (2003) при выборе проектов соответствующего типа.

Сразу после инсталляции пакета главное окно графической среды разработки выглядит так, как показано на рис. П.1.

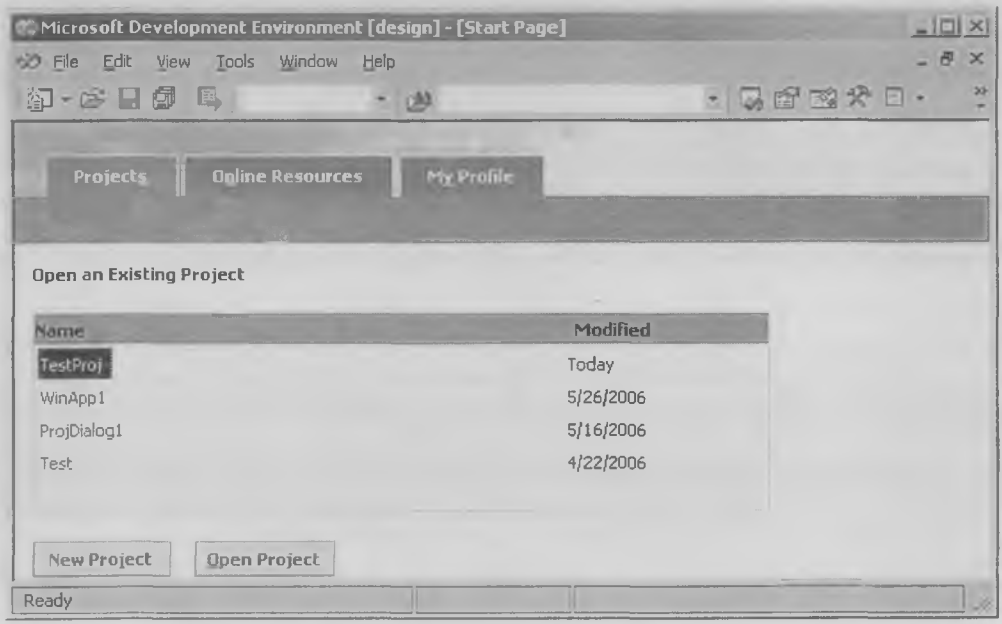


Рис. П.1.

Чтобы главное окно выглядело так, как это показано на рисунках в настоящем учебном пособии, его нужно настроить следующим образом.

Командой меню Tools | Options вызывается диалоговое окно Options (см. рис. П.2), в котором для опции At startup нужно вместо умолчательного значения "Show Start Page" выбрать из списка значение "Show empty environment", и нажать кнопку OK.

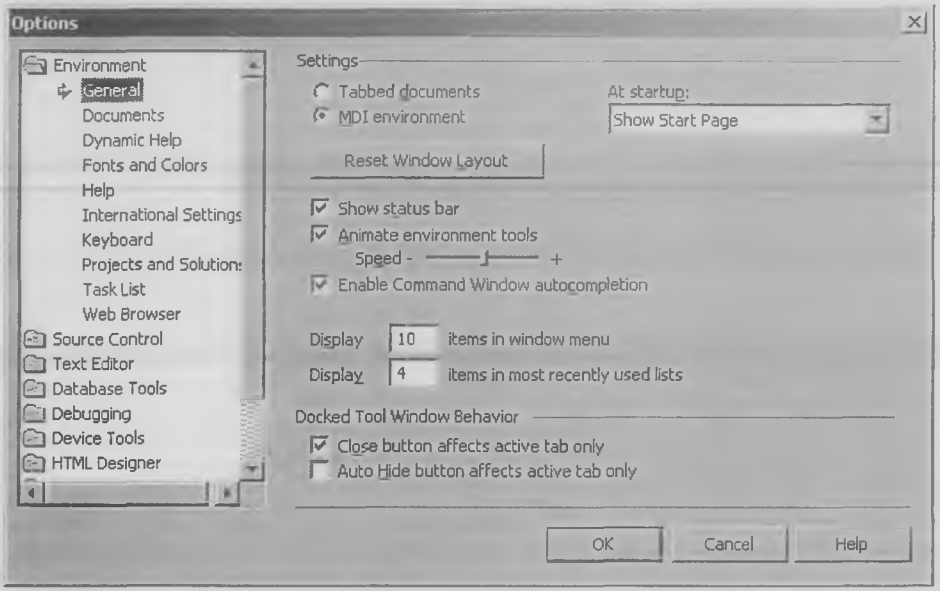


Рис. П.2.

Затем нужно последовательно выполнить команды меню View | Solution Explorer и View | Class View, и нажать на полосе инструментов кнопку Output (может быть определена по всплывающей подсказке Output).

После этого главное окно графической среды разработки Microsoft Developer Studio NET (2003) нужно закрыть и снова открыть. Теперь оно примет удобный и традиционный для нашего учебного пособия вид (см. рис. П.3).


Microsoft Development Environment [design]
File Edit View Tools Window Help
Solution Explorer
Solution Expl...
Class View
Output
Ready

Рис. П.3.

Компилятор Microsoft Visual C# NET (2003)

Прежде, чем создавать программы для Windows на языке C#, требуется выполнить команду меню File | New | Project, и тогда появляется диалоговое окно New Project, показанное на рис. П.4.

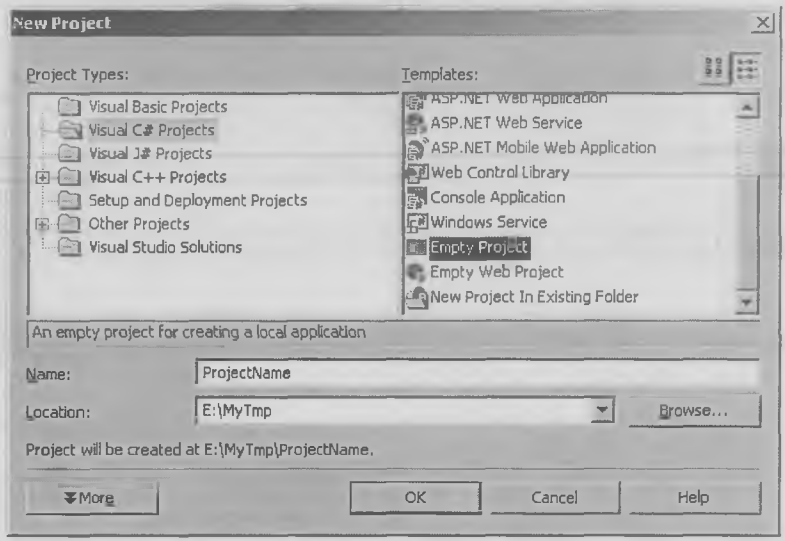


Рис. П.4.

При выбранной в левой части окна New Project позиции Visual C# Projects в правой части окна нужно выбрать конкретный тип проекта: для нашего учебного пособия это или *Empty Project*, или *Windows Application* или *Class Library*.

Вводим имя проекта и нажимаем кнопку OK, и проект готов. В корневом каталоге проекта на диске создаются при этом ряд вспомогательных файлов, из которых важнейшим является файл с расширением sln. При этом вид главного окна среды разработки принимает следующий вид (см. рис. П.5).

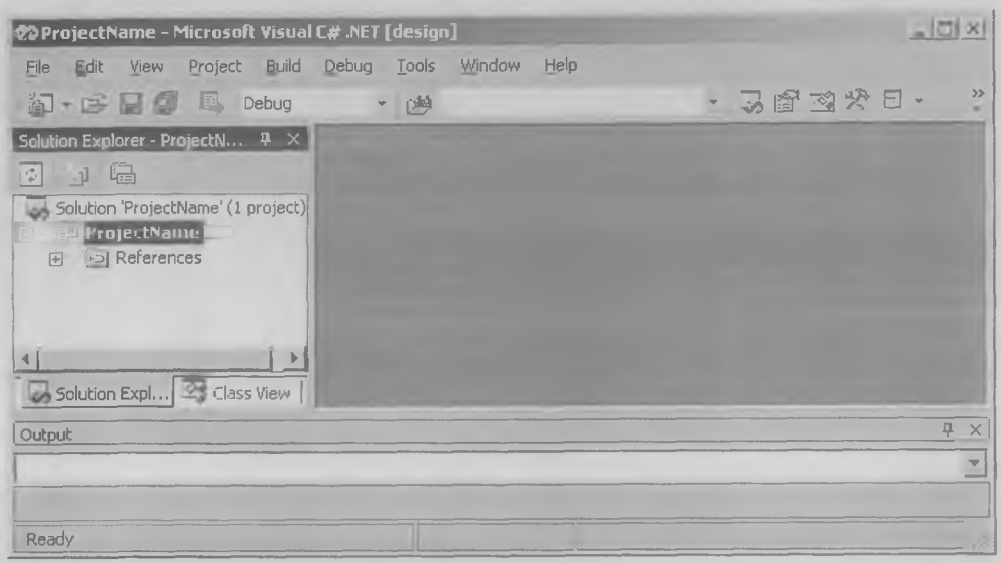


Рис. П.5.

В заголовке этого окна теперь появилась надпись Microsoft Visual C# .NET, а это означает, что в дело вступил компилятор с языка C#. Именно этот компилятор будет обрабатывать исходные файлы проекта.

Файлы с исходными текстами на языкех C# вносятся в проект двумя способами. Уже существующие на диске компьютера файлы внедряются в проект командой меню Project | Add Existing Item.

Новые файлы создаются (естественно, с пустым начальным содержимым) и одновременно подсоединяются к проекту командой меню Project | Add New Item... (см. рис. П.6).

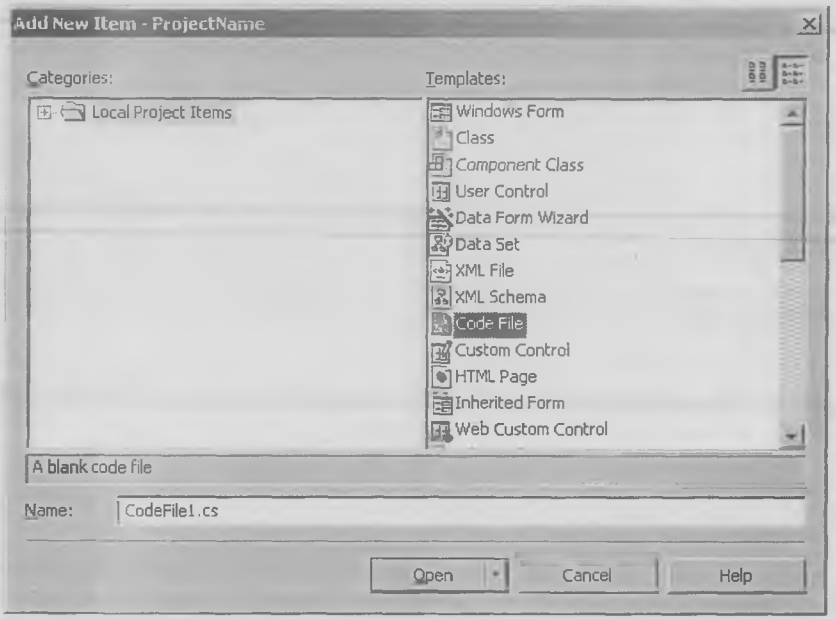


Рис. П.6.

В диалоговом окне "Add New Item" нужно выбрать тип создаваемого файла, причем в настоящем учебном пособии мы всегда выбираем тип *Code File* (файл с текстом программы на языке C#). Задав имя файла, нажимаем кнопку Open. В результате пустой файл с указанным именем создан и подсоединен к проекту. Осталось набирать его текст в текстовом редакторе компилятора.

Компиляция проекта осуществляется по команде меню Build | Build Solution. Запуск отладочного варианта исполняемой программы осуществляется клавишей F5.

Если мы выбрали проект типа Windows Application (приложение с графическим интерфейсом пользователя), то графическая среда разработки с помощью *встроенного дизайнера окон* генерирует стандартный (каркасный) код для отображения главного окна приложения. При этом внешний вид окна среды разработки сразу после создания такого проекта имеет вид, показанный на рис. П.7.

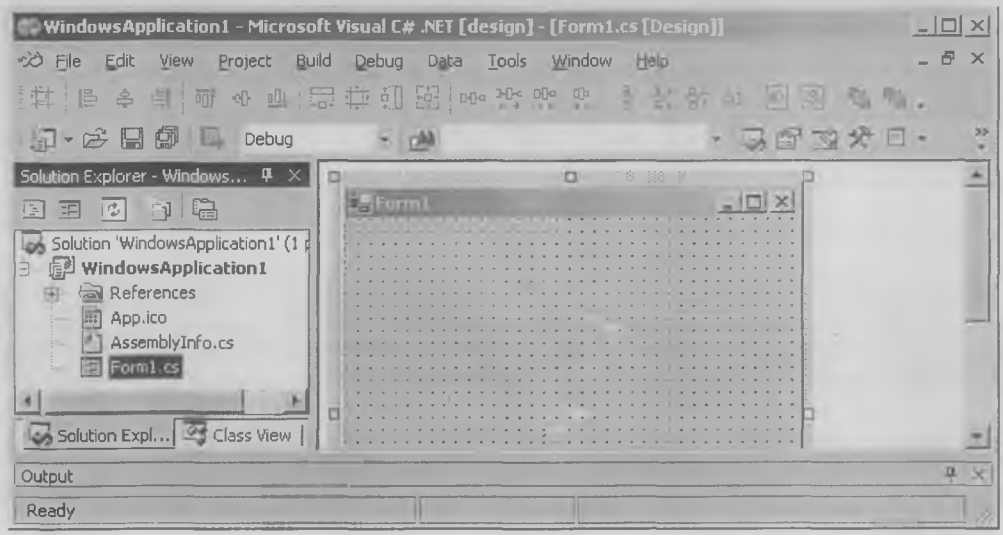


Рис. П.7.

В правой части окна показывается заготовка главного окна приложения, которое можно визуальными средствами компилятора наполнять различными графическими элементами управления.

Для перехода к просмотру кода на языке C#, нужно щелкнуть мышью в левой части окна по строке Form1.cs, и из всплывающего меню выбрать строку *View Code* (см. рис. П.8).

Обратный переход выполняется по строке *View Designer* всплывающего меню.

Для настройки свойств главного окна приложения или графических элементов управления нужно щелкнуть *правой клавишей мыши* по их визуальным изображениям и из всплывающего меню выбрать позицию Properties. В результате появляется окно свойств для выбранного элемента (см. рис. П.9).

Здесь можно выбирать различные значения для свойств, перечисленных в виде списка в левой части окна Properties.

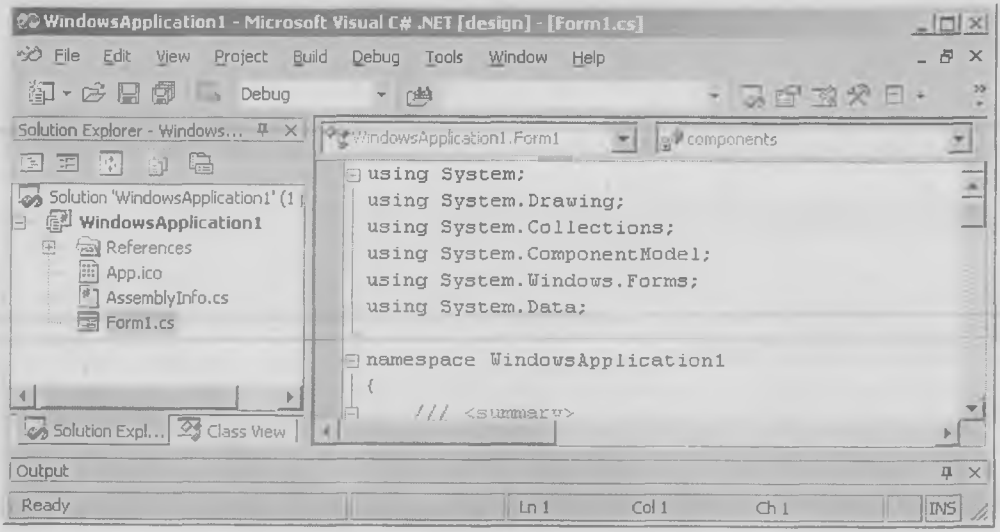


Рис. П.8.

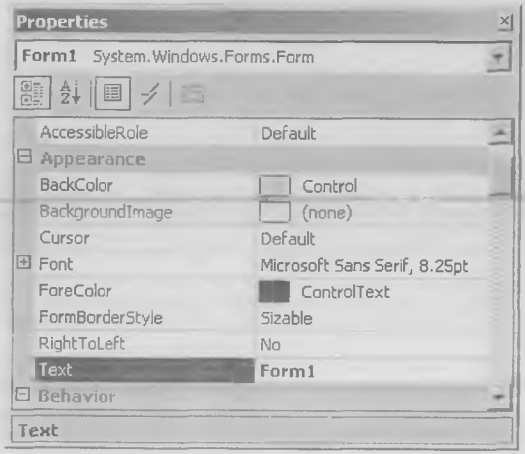


Рис. П.9.

Такова вкратце основная схема работы с компилятором Microsoft Visual C# NET (2003).

Компилятор Microsoft Visual C++ NET (2003)

В рамках среды программирования Microsoft Developer Studio NET (2003) компилятор Microsoft Visual C++ NET (2003) вступает в дело, когда по команде меню File | New | Project в окне "New Project" для типов проектов (Project Types) выбирается позиция Visual C++ Projects (см. рис. П.10).

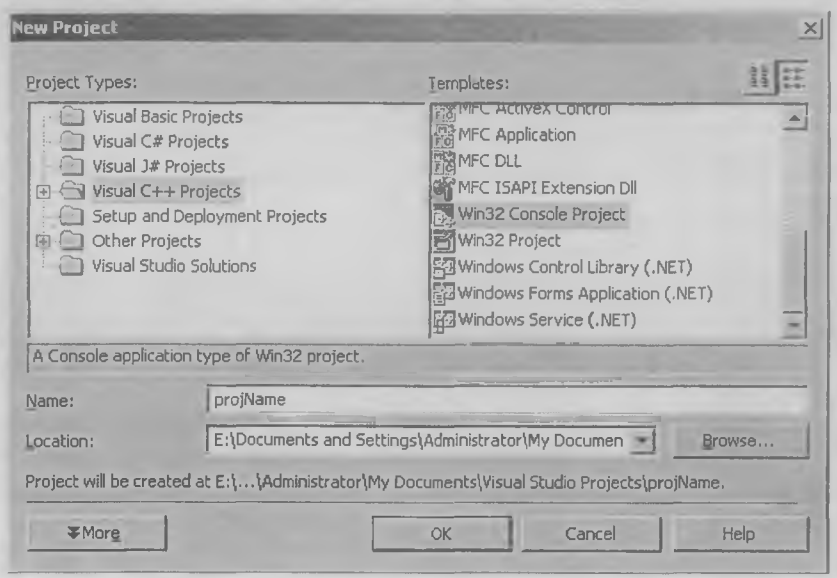


Рис. П.10.

Здесь в правой части окна можно выбрать конкретный тип проекта. Для нас наиболее интересны проекты *Win32 Console Project* (более простой проект с консольным окном и текстовым выводом) и *Win32 Project* (проект с графическим интерфейсом пользователя).

Введя имя проекта и нажав кнопку OK пользователь получит ряд информационных окон, из которых важным является следующее окно, показанное на рис. П.11.

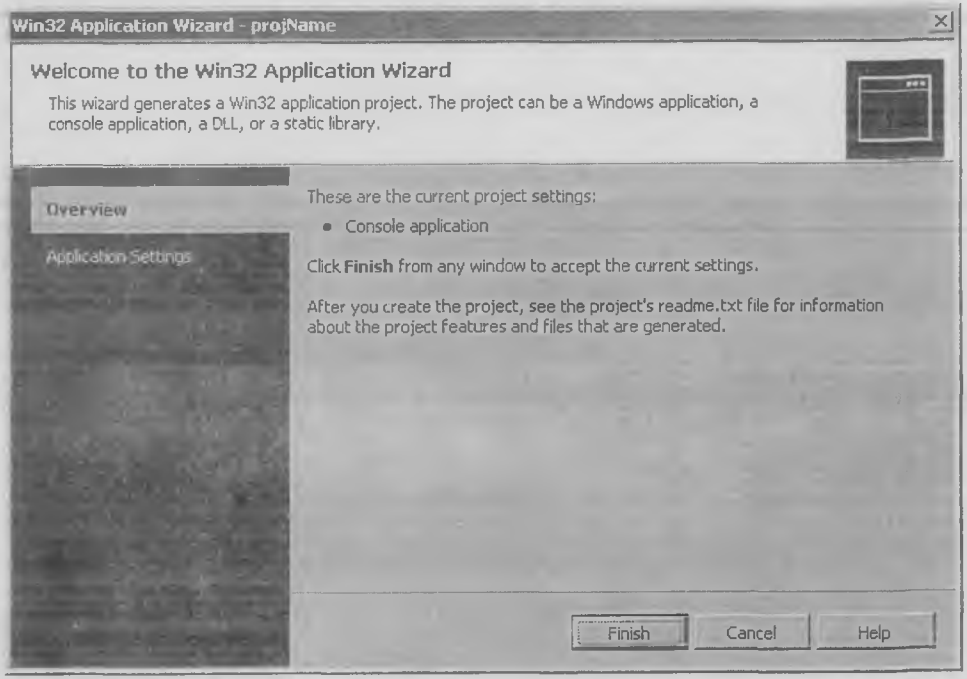


Рис. П.11.

Здесь крайне важно щелкнуть мышью по строке Application Settings, чтобы вызвать еще одно окно (см. рис. П.12).

В этом окне нужно отметить позицию Empty project, которая по умолчанию не устанавливается. Затем нажимают кнопку Finish и проект готов.

В корневом каталоге проекта создаются при этом ряд вспомогательных файлов, из которых важнейшим является файл с расширением sln (это сокращение от слова solution).

Файлы с исходными текстами на языках C или C++ вносятся в проект двумя способами. Уже существующие на диске компьютера файлы внедряются в проект командой меню Project | Add Existing Item.

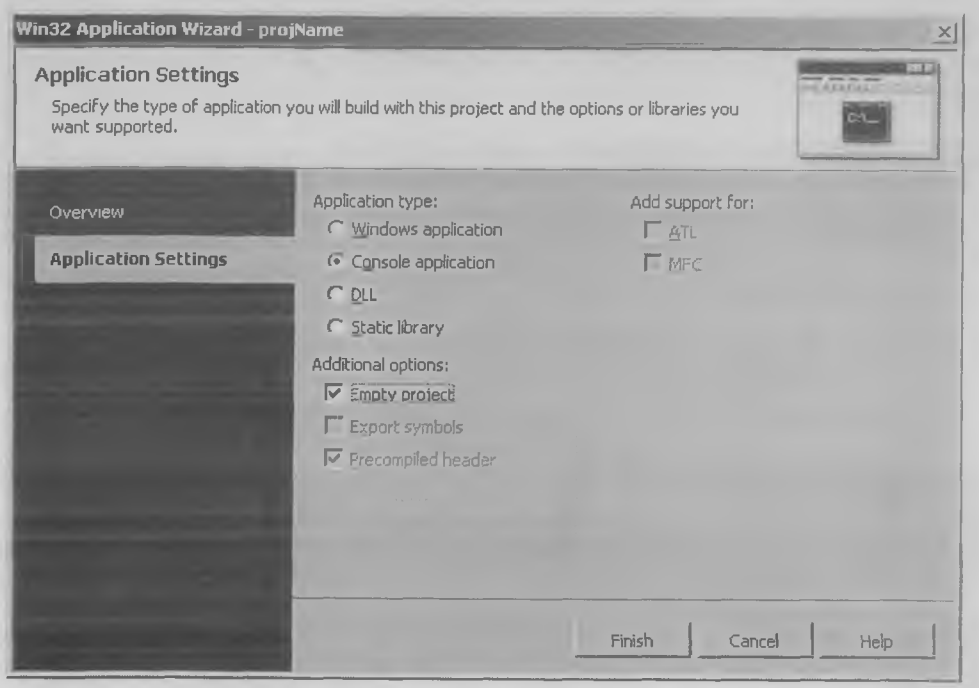

Win32 Application Wizard - projName
Application Settings
Specify the type of application you will build with this project and the options or libraries you want supported.
Overview
Application Settings
Application type:
Windows application
Console application
DLL
Static library
Additional options:
Empty project
Export symbols
Precompiled header
Add support for:
ATL
MFC
Finish
Cancel
Help

Рис. П.12.

Новые файлы создаются (естественно, с пустым начальным содержимым) и одновременно подсоединяются к проекту командой меню Project | Add New Item... и в окне "Add New Item" выбрать тип создаваемого файла (см. рис. П.13).

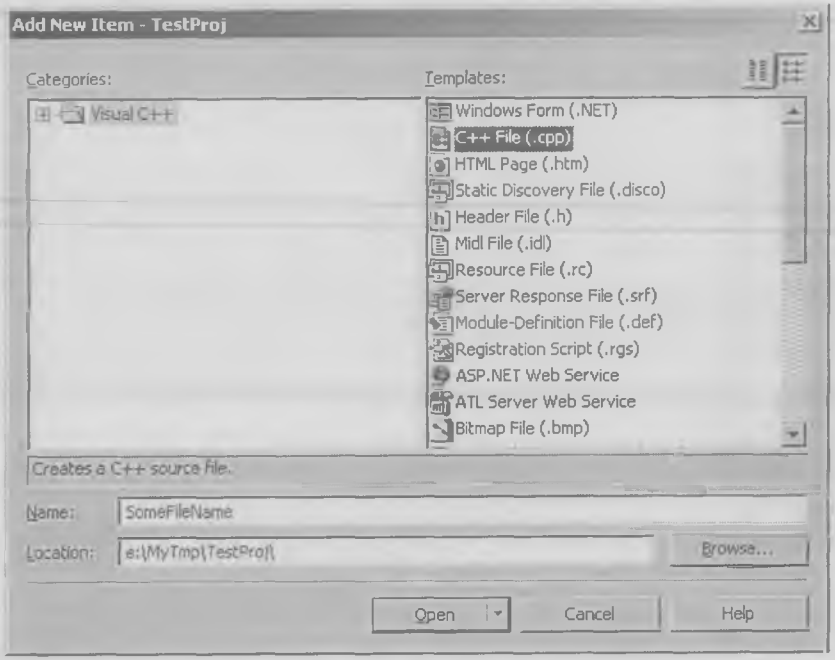


Рис. П.13.

Здесь можно выбрать либо позицию C++ File (.cpp) (подходит и для файлов с расширением c), либо Header File (.h).

Нажимаем кнопку Open и в нашем распоряжении возможность набирать текст пока еще пустого файла в текстовом редакторе компилятора.

Компиляция проекта осуществляется либо соответствующим пунктом меню, либо клавиатурной комбинацией Ctrl+F7.

Запуск отладочного варианта исполняемой программы осуществляется клавишей F5.

Если мы настроили графическую среду Microsoft Developer Studio NET (2003) так, как это описано в начале настоящего Приложения, то после запуска ее главное окно имеет вид, показанный на рис. П.14.

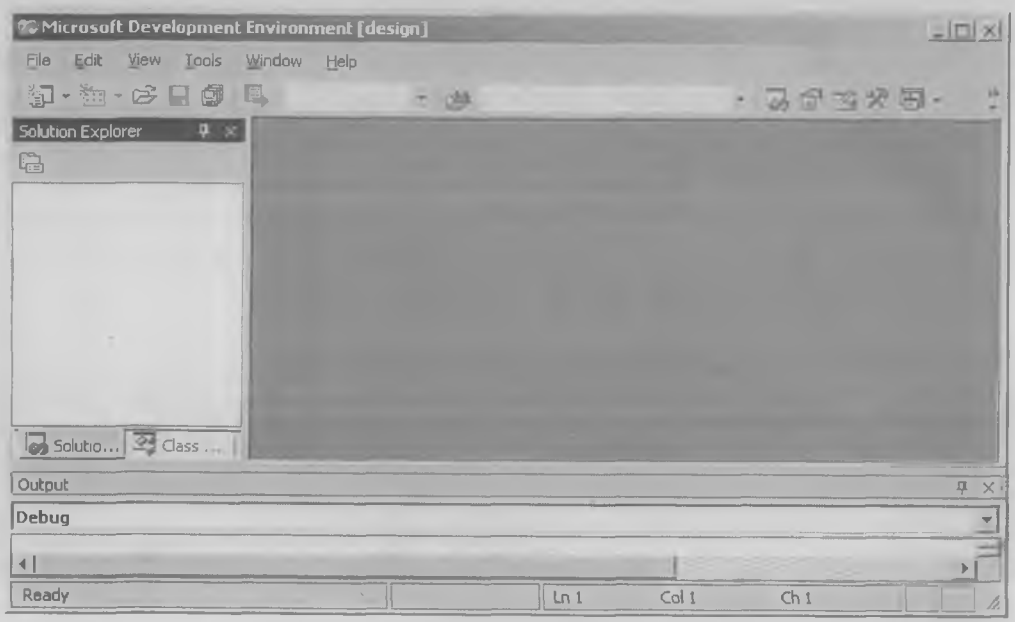


Рис. П.14.

Это же окно после создания нового или открытия существующего проекта имеет следующий типичный вид (см. рис. П.15).

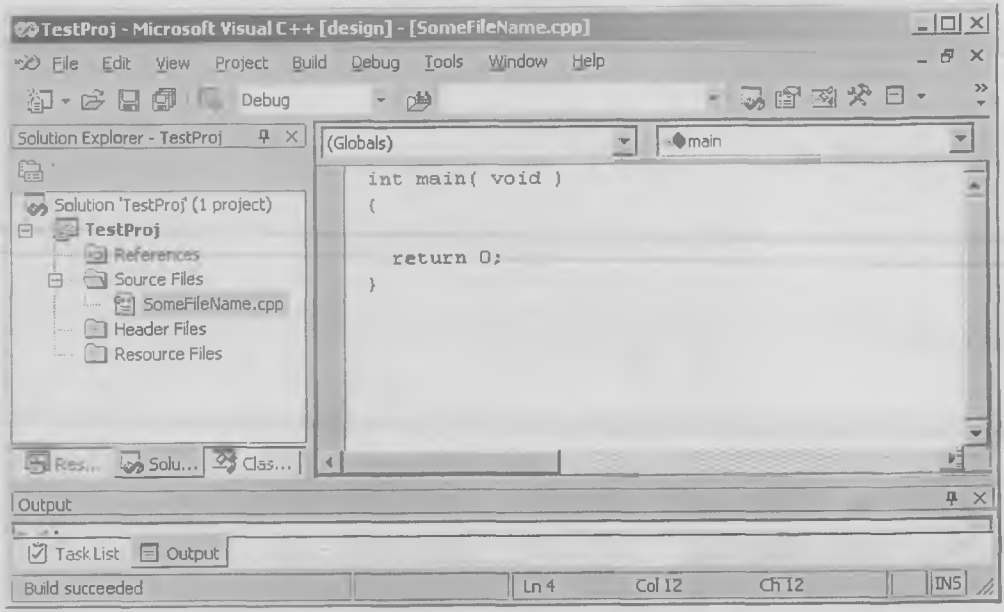


Рис. П.15.

В проекты Win32 Project ресурсы добавляются командой меню Project | Add Resource, после чего в окне "Add Resource" нужно выбрать тип ресурса (нас в первую очередь интересуют диалоговые окна - Dialog) - см. рис. П.16.

Создание ресурса требует нажатия на кнопку New.

После этого созданный ресурс нужно сохранить в виде ресурсного файла в корневом каталоге проекта. Затем сохраненный ресурсный файл (с расширением rc) и автоматически построенный заголовочный файл resource.h нужно присоединить к проекту командой меню Project | Add Existing Item.

Редактирование ресурсов осуществляется в окне *редактора ресурсов* (при активной закладке *ResourceView* в левой части окна компилятора) следующим образом. Мышью нужно щелкнуть по изображению ресурса и тогда в окне "Properties" показывается список свойств этого ресурса. Например, на рис. П.17 показано *окно свойств диалогового окна*.

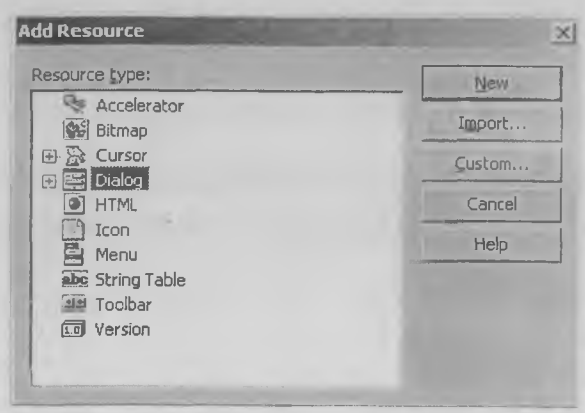


Рис. П.16.

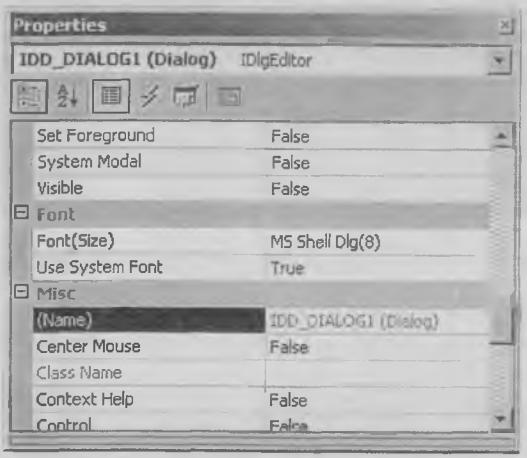


Рис. П.17.

В окнах свойств ресурсов выбираются возможные варианты их свойств и внешнего вида.

Такова вкратце основная схема работы с компилятором Microsoft Visual C++ NET (2003).

Список литературы

[1]. Мартынов Н. Н. Информатика. С для начинающих.– М.: КУДИЦ-ПРЕСС, 2006.

[2]. Мартынов Н. Н. Программирование для Windows на C/C++. Т.1.– М.: БИНОМ, 2004.

[3]. Мартынов Н. Н. Программирование для Windows на C/C++. Т.2.– М.: БИНОМ, 2006.

[4]. Луканкин Г. Л., Мартынов Н. Н., Шадрин Г. А., Яковлев Г. Н.. Высшая математика.– М.: Высшая школа, 2004.

[5]. Э. Троелсен. C# и платформа .NET.– Спб.: ПИТЕР, 2006.

[6]. Фролов А. В., Фролов Г. В. Язык C#. Самоучитель.–М.: ДИАЛОГ-МИФИ, 2003.

[7]. Петзолд Ч. Программирование для Windows 95. Т.1 и т.2.– Спб.: BHV, 1997.

[8]. Дж. Рихтер. Программирование на платформе Microsoft NET Framework.– Спб.: ПИТЕР, 2005.

[9]. Петзолд Ч. Программирование для Microsoft Windows на C#. Т.2.– М.: Русская редакция Microsoft Press, 2002.

[10]. Фролов А. В., Фролов Г. В. Визуальное проектирование приложений C#.– М.: КУДИЦ-ОБРАЗ, 2003.

Возможно, вас заинтересуют другие наши издания

Заказы принимаются по телефону **(495) 333-82-11**
или в Интернет-магазине **http://books.kudits.ru**; **zakaz@kudits.ru**

Мартынов Н.Н
Информатика: С для начинающих.

304 стр., 2006 г., ISBN 5-9579-0107-5

Книга является общедоступным учебником начального уровня по основам информатики и программированию на языке С. Она может быть рекомендована как школьникам и преподавателям средних школ, так и студентам вузов, испытывающим трудности при изучении программирования на языке С. Кроме того, книга будет полезна всем, кто интересуется применением компьютеров для решения задач математики, физики, химии, биологии и других дисциплин, в том числе гуманитарных.

От читателей не требуется специальной подготовки в области программирования, поскольку основной материал в Части I изучается подробно и постепенно, с большим числом практических примеров и наглядных графических иллюстраций. Все главы этой части дополняются обширным списком вопросов и упражнений, достаточных для оценки уровня овладения предметом.

Более сложные и специальные темы отнесены в конец учебника (Часть II) и могут изучаться факультативно. Рассматриваются основы построения приложений Windows с графическим интерфейсом пользователя.

Для практической работы с пособием можно использовать любой доступный компилятор языка С.

Уилсон Мэтью
C++: практический подход к решению проблем программирования.

736 стр., 2006 г., ISBN 5-91136-006-3

C++ – изумительных язык, но не идеальный. Если вы давно занимаетесь разработками на C++, эта книга поможет вам по-новому посмотреть на те сложные проблемы, с которыми приходится сталкиваться при программировании, и освоить мощные методы, которые вы никогда раньше не применяли. Если вы новичок в C++, то научитесь принципам программирования, которые позволят вам более эффективно реализовывать все ваши проекты.

Прилагаемый компакт-диск содержит много различной ценной информации: компиляторы, библиотеки, тестовые программы, инструментальные средства и служебные программы, а также подборку журнальных статей автора.

Климова Л. М.

СИ++. Практическое программирование. Решение типовых задач

596 стр., 2001 г., ISBN 5-93378-020-0

Данное учебное пособие содержит структурированное описание средств языка и основных приемов работы в среде Borland, которое сопровождается большим количеством примеров с результатами их выполнения.

Особое внимание уделено возможностям использования указателей. Например, для динамического формирования больших и свободных массивов различных типов с помощью массивов указателей и для формирования массивов указателей на функции.

Характерной особенностью учебного пособия является четкая систематизация рассматриваемых вопросов, широкое использование средств структурного программирования, в том числе схем алгоритмов и графического представления взаимосвязи указателей и адресуемых ими значений.

Все средства, изложенные в данном учебном пособии, могут быть использованы при программировании как на языке С, так и на языке С++. Учебное пособие предназначено для студентов, преподавателей, разработчиков программного обеспечения и инженеров, желающих в полной мере освоить и использовать возможности языка С++. Оно может быть использовано также для самостоятельного изучения программирования на языках С и С++.

Тэллес Мэтт, Хсих Юань

Наука отладки.

560 стр., 2003 г., ISBN 5-93378-063-4

Ошибки в программах неизбежны. Настоящая книга признает этот факт и учит образу мышления, позволяющему гарантированно отыскивать и устранять ошибки. Она нацелена на то, чтобы сделать отладку менее загадочной, более быстрой и эффективной, экипируя читателя знаниями и методиками, необходимыми для оперативной идентификации, отслеживания и устранения ошибок в программах. Этим книга не ограничивается и идет дальше, предлагая практические советы по минимизации ошибок и улучшению их распознаваемости в тех случаях, когда они все же случаются.

Хохгуртль Брайан
C# и Java: межплатформенные Web-сервисы.

416 стр., 2004 г., ISBN 5-9579-0015-X

Превалирующим трендом в развитии современной индустрии информационных технологий является интеграция информационно-вычислительных систем с использованием Интернет-технологий.

Последним словом в этих Интернет-технологиях являются веб-сервисы, повсеместное распространение которых только еще грядет.

Данная книга является уникальной в своем роде, так как в ней веб-сервисы рассматриваются в контексте сразу двух платформ, имеющих в настоящее время наибольшее распространение – Java и .NET.

Ермолаев В., Сорока Т.
C++ Builder: Книга рецептов

208 стр., 2006 г., ISBN 5-9579-0091-5

Данная книга написана специалистами в области разработки ПО по материалам дискуссий на самом известном российском сайте, посвященном C++Builder: http://bcbdev.ru. В книге, построенной как справочник, даются примеры решения типичных задач, встающих в процессе разработки приложения на C++Builder. Это позволяет разработчикам сконцентрироваться на предметной области, экономя время и не отвлекаясь на частности.

Кроме основной массы вопросов, касающихся создания пользовательского интерфейса, также затрагивается работа с файлами, реестром и рядом внутренних классов VCL.

Для профессиональных разработчиков. Также книга может быть полезна студентам и аспирантам соответствующих специальностей.

7500с.

ИЗДАТЕЛЬСТВО
«КУДИЦ-ПРЕСС»

ПРИОБРЕТАЙТЕ КНИГИ У НАШИХ ПАРТНЕРОВ

Алматы
ЧП Амреев Болат Аскарович
магазин "КОМПЬЮТЕРЫ"
(угол ул. Фурманова)
E-mail: amreev@hotmail.ru

Беларусь, г. Гродно
ЧП Баранов Дмитрий Алексеевич
(10-375-1522) 29-6-29
E-mail: logos-grodno@mail.ru

Вологда
ООО "Венал" Оптово-розничная торговля,
ул. Челюскинцев, д. 9 (8172) 75-21-43

Воронеж
"Книжный мир семьи", пр-т. Революции, 58,
(0732) 51-28-90

Донецк
ЧП Карымов Ратмир Гибадулович,
(10-380-62) 381-9232

Екатеринбург
КТК Дом Книги ООО
Екатеринбург, ул. Антона Валека, д.12
(343) 359-41-04

Иркутск
"Продалит", (3952) 232-862, 591-380, 590-990
E-mail: prodalit@irk.ru

Калининград
ООО "Контакт" (0112) 35-37-66

Киев
"Микроника", ул. М. Расковой, 13, (044) 517-73-77
"Технокнига", (044) 268-53-46

Комсомольск-на-Амуре
МУП "Планета" (42175) 0-46-36x

Краснодар
"БиблиоМан", biblioman1@mail.ru

Минск
Книготорговая компания "Делсар"
Беларусь, Минск, ул. Академическая, д. 28, к. 111
Тел.: (017) 284-16-55, 284-03-23
E-mail: sales@lit.by
Интернет-магазин "Центр Компьютерной
и Деловой Литературы": www.lit.by

Москва
"Дашков и К°"
(095) 182-42-01, 183-93-01, evgeniv@dashkov.ru
www.dashkov.ru
ООО ТЦ "Мир фото" Ленинский пр-т, 62/1
137-08-33, info@kinolubitel.com.ru

Новосибирск
"Книжный пассаж", ул. Ленина, 10а, (3832) 29-50-30
"Сибирский Дом Книги", Красный пр-т, 153, (3832) 26-62-39
"Книжный мир", пр-т К. Маркса, 51

Нижний Новгород
"Дельфин" (8312) 175-157, (8312) 168-125
nata@defis.kis.ru

Пермь
ИП Сер
(3422) 4

Орск
ИП Ше
Пл. Гаг

Ростов-на-
"Мир кн
(8632) 6
"Делова
Сеть кн
ул. Чех
magistr
"Феник

Самара
Агенств
ул. Анто
(8462)7

Санкт-Пете
"Новая Т
Измайл
Информ
ул. Раз

Саратов
"Книжн

Смоленск
"Эрудит
(0812) 3

Ставропол
"Книжн

Таганрог
"Компьк
(8634) 3

Томск
"Книжн

Уфа
ООО П
Оптова
Розничн
ул. Черн
Магазин

Ханты-Ман
Магазин

Харьков
Книжны
ул. Кло

Челябинск
"Книжны

Шахты
ООО "Ш
пр-т. По
E-mail:

Ярославль
Магазин
(0852) 2

ЗАКАЗ КНИГ НАЛОЖЕННЫМ ПЛАТЕЖОМ

Издательство «КУДИЦ-ПРЕСС» осуществляет рассылку книг по почте.
Заказы принимаются по адресу: 121354, Москва, а/я 18; через интернет-магазин http://books.kudits
Заказы из регионов России с авиадоставкой, а также заказы из стран ближнего и дальнего зару

Хохгуртль Брайан
C# и Java: межплатформенные Web-сервисы.

416 стр., 2004 г., ISBN 5-9579-0015-X

Превалирующим трендом в развитии современной индустрии информационных технологий является интеграция информационно-вычислительных систем с использованием Интернет-технологий.

Последним словом в этих Интернет-технологиях являются веб-сервисы, повсеместное распространение которых только еще грядет.

Данная книга является уникальной в своем роде, так как в ней веб-сервисы рассматриваются в контексте сразу двух платформ, имеющих в настоящее время наибольшее распространение – Java и .NET.

Ермолаев В., Сорока Т.
C++ Builder: Книга рецептов

208 стр., 2006 г., ISBN 5-9579-0091-5

Данная книга написана специалистами в области разработки ПО по материалам дискуссий на самом известном российском сайте, посвященном C++Builder: http://bcbdev.ru. В книге, построенной как справочник, даются примеры решения типичных задач, встающих в процессе разработки приложения на C++Builder. Это позволяет разработчикам сконцентрироваться на предметной области, экономя время и не отвлекаясь на частности.

Кроме основной массы вопросов, касающихся создания пользовательского интерфейса, также затрагивается работа с файлами, реестром и рядом внутренних классов VCL.

Для профессиональных разработчиков. Также книга может быть полезна студентам и аспирантам соответствующих специальностей.

7500с.

ИЗДАТЕЛЬСТВО
«КУДИЦ-ПРЕСС»

Тел.: (495) 333-65-67; zakaz@okc.ru,
http://books.kudits.ru

ПРИОБРЕТАЙТЕ КНИГИ У НАШИХ ПАРТНЕРОВ

Алматы
ЧП Амреев Болат Аскарович
магазин "КОМПЬЮТЕРЫ"
(угол ул. Фурманова)
E-mail: amreev@hotmail.ru

Беларусь, г. Гродно
ЧП Баранов Дмитрий Алексеевич
(10-375-1522) 29-6-29
E-mail: logos-grodno@mail.ru

Вологда
ООО "Венал" Оптово-розничная торговля,
ул. Челюскинцев, д. 9 (8172) 75-21-43

Воронеж
"Книжный мир семьи", пр-т. Революции, 58,
(0732) 51-28-90

Донецк
ЧП Карымов Ратмир Гибадулович,
(10-380-62) 381-9232

Екатеринбург
КТК Дом Книги ООО
Екатеринбург, ул. Антона Валека, д.12
(343) 359-41-04

Иркутск
"Продалит", (3952) 232-862, 591-380, 590-990
E-mail: prodalit@irk.ru

Калининград
ООО "Контакт" (0112) 35-37-66

Киев
"Микроника", ул. М. Расковой, 13, (044) 517-73-77
"Технокнига", (044) 268-53-46

Комсомольск-на-Амуре
МУП "Планета" (42175) 0-46-36x

Краснодар
"БиблиоМан", biblioman1@mail.ru

Минск
Книготорговая компания "Делсар"
Беларусь, Минск, ул. Академическая, д. 28, к. 111
Тел.: (017) 284-16-55, 284-03-23
E-mail: sales@lit.by
Интернет-магазин "Центр Компьютерной
и Деловой Литературы": www.lit.by

Москва
"Дашков и К°"
(095) 182-42-01, 183-93-01. evgeniy@dashkov.ru
www.dashkov.ru
ООО ТЦ "Мир фото" Ленинский пр-т, 62/1
137-08-33, info@kinolubitel.com.ru

Новосибирск
"Книжный пассаж", ул. Ленина, 10а, (3832) 29-50-30
"Сибирский Дом Книги", Красный пр-т, 153, (3832) 26-62-39
"Книжный мир", пр-т К. Маркса, 51

Нижний Новгород
"Дельфин" (8312) 175-157, (8312) 168-125
nata@defis.kis.ru

Пермь
ИП Сергеев Александр Владимирович
(3422) 45-96-55, E-mail: galina5@permonline.ru

Орск
ИП Шевченко
Пл. Гагарина, 1а, (3537) 22-25-99

Ростов-на-Дону
"Мир книги", Ворошиловский пр-т, 33;
(8632) 62-54-61
"Деловая литература", (8632) 62-36-55
Сеть книжных магазинов "Магистр"
ул. Чехова, 31, ул. Б. Садовая, 67
magistr@aaanet.ru
"Феникс 21 век", E-mail: wek163@phoenixrostov.ru

Самара
Агенство деловой информации "ЭЖ-САМАРА"
ул. Антонова-Овсеенко, 44 "А"
(8462)78-57-58, 78-57-59, 79-04-25

Санкт-Петербург
"Новая Техническая книга"
Измайловский проспект, 29
Информационно-Торговое агенство "Бизнес-Пресса"
ул. Разъездная, д. 39

Саратов
"Книжный Мир"; пр-т Кирова, 32, (8452) 32-98-14

Смоленск
"Эрудит", ул. Дохтурова, д. 3, оф. 901
(0812) 32-75-21, (0812) 65-62-94

Ставрополь
"Книжный Мир", ул. Мира, 337, (8652) 35-47-90

Таганрог
"Компьютерная книга", ул. Чехова, 31,
(8634) 37-13-12

Томск
"Книжный Мир", ул. Ленина, 141, (3822) 51-07-16

Уфа
ООО ПКП "Азия", тел./факс: (3472) 50-39-00
Оптовая торговля Ул. Зенцова, 70
Розничная торговля Магазин "Оазис",
ул. Чернышевского, 88
Магазин "Книжник", пр. Октября, 106

Ханты-Мансийск
Магазин "Книги", ул. Ленина, 39

Харьков
Книжный рынок "Райский уголок",
ул. Клочковская, 28, (0572) 549-116

Челябинск
"Книжный Мир", ул. Кирова, 90, (3512) 33-19-58

Шахты
ООО "Шахтинский книготорг", Ростовская обл.,
пр-т. Победы Революции, 130 "Б"
E-mail: domknigi@pochta.ru

Ярославль
Магазин "Наука", ул. Володарского, 63,
(0852) 25-95-04

ЗАКАЗ КНИГ НАЛОЖЕННЫМ ПЛАТЕЖОМ

Издательство «КУДИЦ-ПРЕСС» осуществляет рассылку книг по почте.
Заказы принимаются по адресу: 121354, Москва, а/я 18; через интернет-магазин http://books.kudits.ru, или zakaz@okc.ru
Заказы из регионов России с авиадоставкой, а также заказы из стран ближнего и дальнего зарубежья обслуживаются только по предварительной оплате.