

Приложение 2

Обзор Borland C++ Builder

В этой книге приведено довольно большое количество исходных кодов компьютерных программ, используемых при различных вариантах «необычного» применения компьютера. Проекты всех опубликованных программ выполнялись в среде быстрого программирования Borland C++ Builder.

За последние годы было опубликовано довольно много книг, озаглавленных как самоучители программирования в среде C++ Builder. Поэтому найти описание собственно среды C++ Builder, по моему мнению, не составит большого труда.

Тем из читателей, которые еще не знакомы с каким-либо языком программирования, советую приобрести мою книгу «Быстрое программирование на C++» [3]. Большим преимуществом этой книги является наличие в ней массы примеров создания проектов компьютерных программ всевозможного назначения. В прилагаемом к книге компакт диске вы найдете программу Borland C++ Builder и большое число иных вспомогательных программ, необходимых для создания полноценного Windows-приложения.

Быстрая разработка приложений

Создание систем быстрой разработки компьютерных программ (от англ. Rapid Application Development — RAD) вызвано требованием нашего времени.

Начало было положено фирмой Microsoft, создавшей среду для возможностей визуального программирования Visual Basic. Эта среда программирования сформировала новый стиль взаимодействия разработчика программы с компьютером, позволяя наглядно конструировать пользовательский интерфейс с помощью мыши, а не обычным для прежних времен путем написания кодов, их последующей неоднократной трансляцией и выполнением программы, после чего только и можно было посмотреть, как же это выглядит на экране.

Созданные фирмой Borland системы Delphi и C++ Builder — это следующий шаг в развитии среды быстрой разработки приложений. Они исправляют многие дефекты, обнаруженные в Visual Basic. Разработчики этих систем создали инструменты, которые на первый взгляд выглядят похожими на среду Visual Basic, хотя в действительности они заметно лучше.

Перечисленные выше системы для быстрой разработки компьютерных программ служат только для операционной системы Windows.

Далее в этом приложении вы можете познакомиться со средой программирования C++ Builder 6, которая является великолепным инструментом, как для опытных программистов, так и для начинающих изучать программирование.

Но следует помнить, что все приведенные в этой книге исходные коды программ можно выполнить на других версиях C++ Builder, начиная с версии 4, которая располагается на компакт-диске, приложенном к [3].

Запускаем C++ Builder 6

После запуска программы на экране моего компьютера появляется картинка, которая состоит из нескольких отдельных окон. На экране вашего компьютера эта картинка может быть несколько иной. Все зависит от версии программы и ее настройки. Все открывшиеся на экране компьютера окна составляют вместе *интегрированную среду разработки* (IDE) в ее начальном состоянии. На рис. П2.1 представлена копия картинки с экрана моего компьютера.

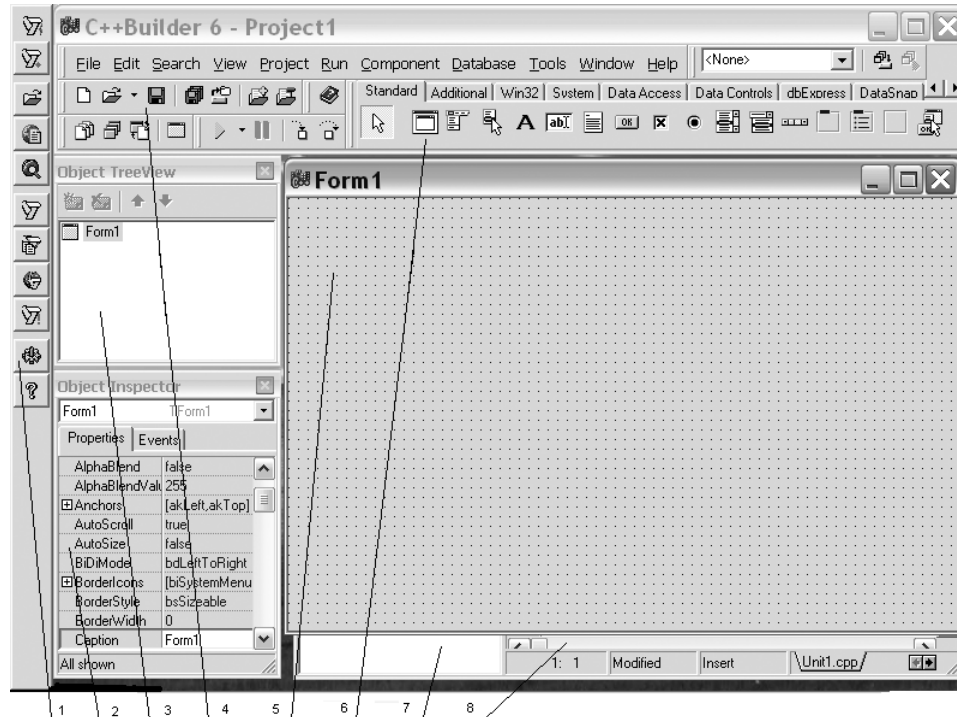


Рис. П2.1. Составные части интегрированной среды разработки

В нижней части рисунка находятся цифры, от которых идут указатели к основным окнам, расположенным на экране в начальный момент. Привожу перечень этих окон.

1. Интегратор программы машинного перевода PROMT.
2. Окно Диспетчер объектов (Object Inspector).
3. Окно древовидного каталога объектов (Object TreeView).
4. Окно панели управления C++ Builder.
5. Окно формы (Form1).
6. Окно панели компонентов (Component Palette).
7. Окно просмотрщика (исследователя) классов (Class Explorer).
8. Окно редактора кодов.

Далее рассмотрим назначение каждого из перечисленных окон.

Интегратор программы PROMT

Этот интегратор не имеет к IDE C++ Builder 6 никакого отношения. Программа PROMT используется мною для быстрого перевода пояснительных текстов системы C++ Builder 6 и комментариев, встретившихся в программах зарубежных авторов.

Большинство программ фирмы Borland позволяют легко и быстро получить помощь — подсказку по любому оператору, типовой функции или по выявленной при компиляции ошибке. Для этого достаточно установить курсор на нужный оператор и нажать сочетание клавиш <Ctrl+F1>. Тут же появляется подсказка, разумеется, на английском языке. Теперь нужно выделить мышкой требующий перевода текст подсказки и нажать сочетание клавиш <Ctrl+Insert>. Этими действиями мы выделенный текст копируем в буфер обмена (clipboard). После этого нужно выполнить перевод, для чего следует установить курсор на вторую от верха пиктограмму интегратора и нажать левую клавишу мышки. Качество машинного перевода, выполненного программой PROMT, мягко говоря, не выдерживает никакой критики, но лучшего варианта я пока не знаю.

Диспетчер объектов (Object Inspector)

Диспетчер объектов — очень важная часть среды разработки. Он предназначен для задания *свойств* (Properties) объектов и определения их реакции на *различные события* (Events). Понятие «объект» имеет очень широкий смысл, но в данном случае под этим термином подразумевается любой из компонентов, имеющих в составе данной версии C++ Builder.

Свойства объектов

Свойство объекта — это одна из его характеристик, такая, как, например, ширина кнопки, название окна, наличие полос прокрутки у списка, цвет и стиль шрифта, имя файла для рисунка и т.д. Как только мы выбираем и устанавливаем на форму какой-то компонент (объект), то программа автоматически в диспетчере объектов устанавливает доступные этому компоненту (объекту) свойства и события.

Каждый объект имеет большое число свойств, при этом многие из объектов имеют схожие свойства, свойства других объектов могут сильно различаться.

Диспетчер объектов позволяет быстро и удобно менять любые свойства текущего (выделенного на форме) объекта. При этом вносимые изменения немедленно сказываются на внешнем виде этого объекта. Например, если мы с помощью диспетчера изменим текст надписи на кнопке, это изменение мгновенно отобразится на самой кнопке в проектируемой форме.

События

Для каждого объекта имеется специальный перечень доступных этому объекту событий.

Событие позволяет программе реагировать на любые действия пользователя так, как это задано программистом. К событиям относятся щелчок мышью на кнопке, выбор пункта меню, изменение состояния переключателя, и иные происшествия как внутри самой программы, так и в операционной системе Windows.

Задавая реакции на различные события, мы тем самым определяем всю внутреннюю логику работы программы.

В окне диспетчера объектов имеются две вкладки — вкладка свойств выбранного объекта (Properties) и вкладка событий (Events), на которые этот объект может реагировать. Каждая из вкладок содержит панель, состоящую из двух колонок. В первой указываются названия свойств (их менять нельзя), во второй — текущие значения соответствующих свойств. Эти значения должны отвечать определенным ограничениям C++ Builder.

Некоторые свойства меняются простым вводом значения в соответствующей строке. Так, например, мы можем изменить толщину рамки окна в свойстве Border Width, введя вместо 0 число 5.

Некоторые свойства могут иметь вложенную структуру, как, например, свойство **Border Icons** в левой колонке, и диспетчер объектов откроет доступ к скрытым свойствам. Таким же способом можно свернуть ранее раскрытое свойство **Border Width** — снова дважды щелкнуть на его названии левой кнопкой мыши.

Каталог объектов (Object TreeView)

В окно каталога объектов программой автоматически заносятся все выбранные нами и установленные компоненты (объекты). Выделенный на форме объект тут же оказывается выбранным и в окне каталога объектов, и наоборот. Значительно улучшает удобство использования интегрированной среды разработки при выполнении сложных программ.

Панель управления

В верхней части окна, показанного на рис. П2.1, располагается главное меню, которое служит основным органом управления средой программирования C++ Builder 6. Чуть ниже меню, в левой стороне находится панель быстрого управления. Это специальная кнопочная панель, которая обеспечивает быстрый доступ к таким наиболее часто используемым командам меню, как **Run** (Старт), **Step Through** (Пошаговая отладка), **View Unit** (Просмотр модуля), **View Form** (Просмотр формы) и **Add to Project** (Добавить в проект). Эту панель разработчик может настроить по своему желанию, добавляя в нее наиболее нужные команды и удаляя те, которые используются не очень часто.

Одним из основных достоинств среды разработки является возможность ее настройки. Кнопку быстрого доступа к команде меню можно добавить или удалить из любой панели инструментов, за исключением панели компонентов Component Palette, которая содержит компоненты, а не команды. В C++Builder предусмотрены следующие панели инструментов.

- Standard (Стандартная панель).
- View (Панель просмотра).
- Debug (Панель отладки).
- Custom (Панель настройки).
- Component Palette (Панель компонентов).

Форма (Form)

Форма — это окно, которое в большинстве случаев является пользовательским интерфейсом создаваемого приложения. Пустая форма создается автоматически при создании нового приложения с помощью C++Builder. Для создания пользовательского интерфейса в форму нужно просто добавить соответствующие визуальные и невидимые компоненты. При этом невидимые компоненты во время создания приложения будут иметь вид пиктограммы компонента, а во время запуска будут скрыты.

При запуске приложения пользователем форма по умолчанию отображается в центре экрана. Исходное расположение формы и другие ее параметры можно изменить, задавая соответствующие свойства формы с помощью окна Object Inspector.

Панель компонентов (Component Palette)

Панель (или палитра) компонентов располагается сразу же под линейкой главного меню справа и содержит все типы компонентов библиотеки VCL. Эти компоненты сгруппированы на вкладках по своеобразным страницам, названия которых располагаются над ними. Для выбора компонента следует установить на этот компонент курсор и нажать левую клавишу мышки. Затем следует переместить курсор в нужное место на форме и снова нажать левую клавишу мышки. Для установки на форму некоторых компонентов, имеющих геометрические размеры, следует, не отпуская нажатую клавишу, переместить курсор в следующую конечную точку границы компонента и отпустить нажатую клавишу. При этом в окне диспетчера объектов появятся все доступные данному компоненту свойства.

Просмотрщик классов (Class Explorer)

В этом окне, по мере разработки программы, записываются сведения о всех задействованных в программе классах. Это очень удобно для контроля и быстрого получения необходимой информации.

Редактор кодов

На рис. П2.1 редактор кодов как бы «выглядывает» внизу из-под окна формы. Для активизации редактора следует установить курсор на видимый участок редактора и нажать левую кнопку мыши. При этом окно редактора кодов совместно с просмотрщиком классов переместится на первый план, а окно формы окажется скрытым, но верхняя часть окна формы останется видимой. Этим можно будет воспользоваться для быстрой активизации окна формы.

Активизировать любое из этих окон можно также с использованием главного меню.

На рис. П2.2 показано активизированное окно редактора кодов, но малый горизонтальный размер окна очень не удобен для работы, поэтому я обычно убираю **Class Explorer**, чтобы увеличить размер окна редактора кодов. Такой вариант показан на рис. П2.3.

Иногда бывает необходимо длительное время работать только с редактором кодов. В таком случае окно редактора можно развернуть до максимального размера. Этот вариант показан на рис. П2.4.

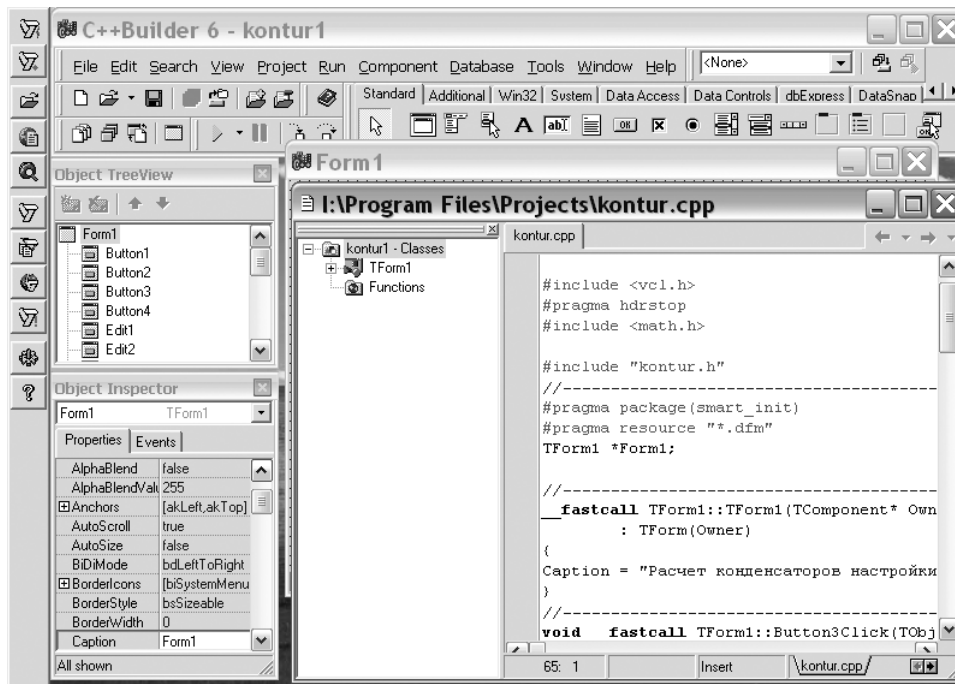


Рис. П2.2. Интегрированная среда с окном редактора кодов

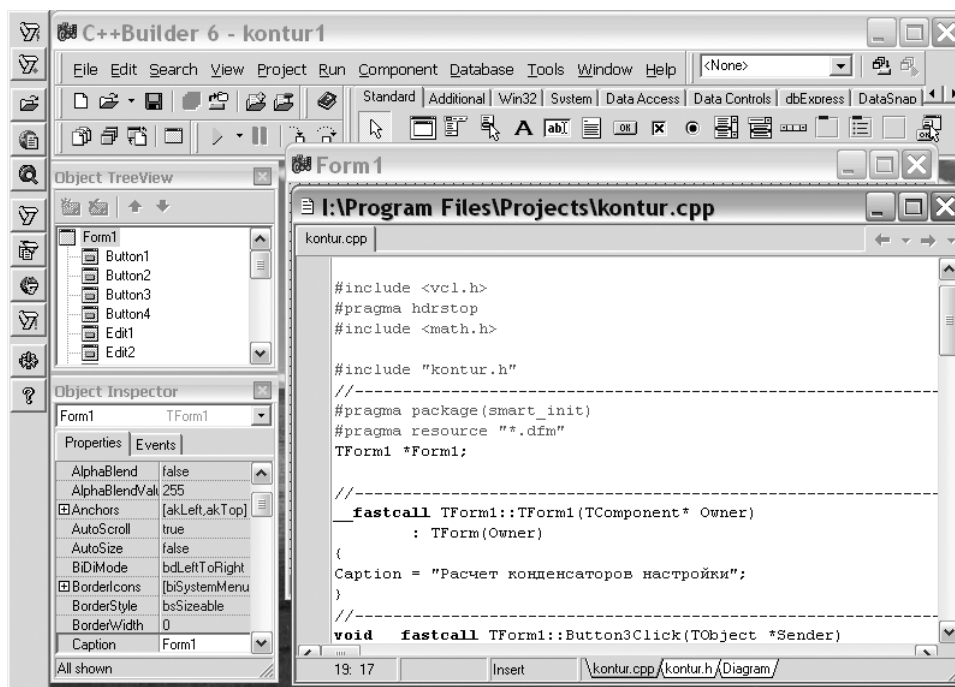


Рис. П2.3. Редактор кодов в рабочем состоянии

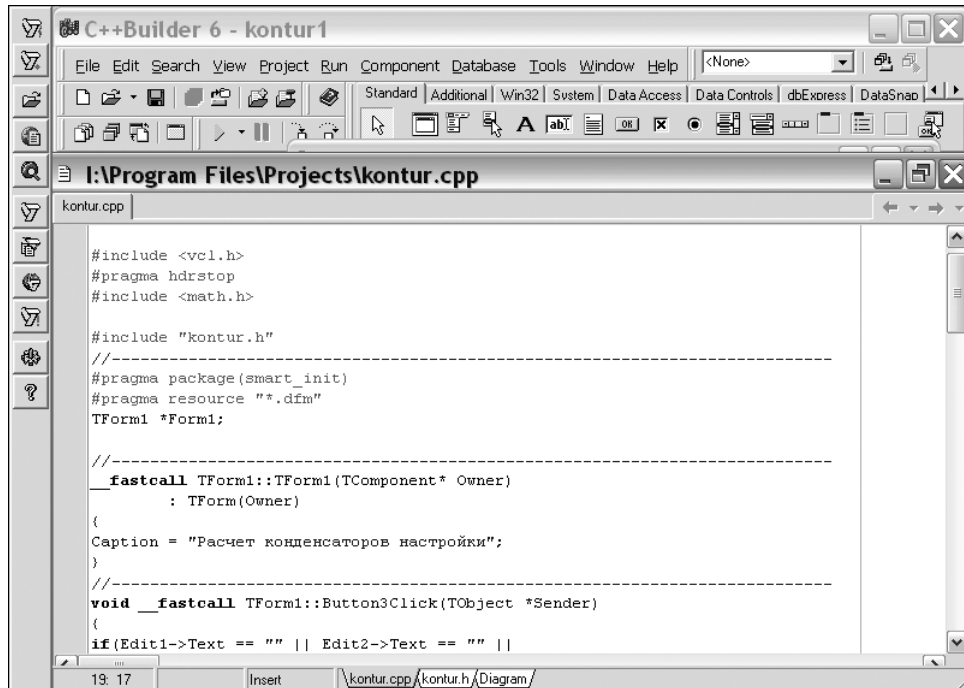


Рис. П2.4. Окно редактора кодов максимального размера

Большую часть времени при программировании приходится работать именно с редактором кодов, поэтому я посчитал нужным более подробно описать подготовку редактора к работе.

Библиотека VCL и компоненты

Библиотека визуальных компонентов — VCL — библиотека (Visual Component Library) — является хранилищем компонентов, используемых для создания приложений с помощью C++Builder. Компонентом называется объект, используемый для создания программы. Это может быть какая-то кнопка, комбинированный список или рисунок. Каждый из компонентов выбирается с помощью щелчка левой кнопкой мыши и перемещается в рабочую область. Компоненты VCL-библиотеки представляют собой код, который скомпилирован для выполнения определенных операций, что избавляет разработчика от необходимости всякий раз создавать его заново.

Кроме того, разработчик может добавлять и создавать собственные компоненты. Таким образом, компоненты VCL-библиотеки избавляют разработчика от выполнения трудоемких и рутинных операций. Они располагаются в панели компонентов Component Palette и некоторые из них более подробно описаны ниже в этой главе.

Все компоненты обладают свойствами, которыми можно управлять с помощью кода C++Builder. Свойства компонента определяют способ его работы, внешний вид, набор функциональных возможностей и т.д. Их можно модифицировать с помощью вкладки свойств **Properties** в окне **Object Inspector**. Параметры свойств

можно изменять с помощью кода, но без достаточного опыта работы C++Builder и VCL-библиотекой это делать не рекомендуется.

Окно Object Inspector также содержит вкладку событий Events, в которой можно создавать события для определения способа взаимодействия пользователя с этим компонентом. Кроме того, эти события могут выполнять другие определенные разработчиком действия.

Далее привожу описания особенностей работы с различными компонентами, которые приходится применять наиболее часто, и которые применены в программах, опубликованных в данной книге.

Компоненты для отображения текста — Label, Static Text, Panel

Для отображения различных надписей на форме используются в основном компоненты **Label**, **Static Text** и **Panel**. Первые два из этих компонентов специально предназначены для отображения текстов и называются «метки». Основное назначение панели **Panel** состоит в компоновке компонентов в окне формы. Однако панель можно использовать и для вывода текстов.

Примеры вывода текста в метки приведены практически в каждой программе следующей главы.

В свойство **Caption** перечисленных компонентов записываются необходимые тексты, которые после запуска программы должны изображаться на форме в заданном месте. Это свойство можно устанавливать в процессе проектирования или задавать и изменять программно во время выполнения приложения. Например:

```
Label1->Caption = «Это пример вывода нового текста».
```

Свойство **Caption** имеет тип строки **AnsiString**. При присваивании этому типу числовой информации происходит ее автоматическое преобразование в строку. Поэтому вы можете непосредственно осуществлять подобные присваивания. Например, оператор

```
Label1 - > Caption = 6.8.
```

На рис. П2.5 показаны некоторые из вариантов оформления текста компонентом **Label**.

Приведет к появлению в метке надписи «6,8». Но если вы хотите занести в метку смешанную информацию, в которой будут содержаться строковые символы и числовые переменные величины, то следует числовые величины преобразовать в строковые символы посредством функций **FloatToStr** и **IntToStr**, переводящими соответственно числа с плавающей запятой и целые числа в строку символов. Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию «+», которая для строк означает их склеивание (конкатенацию). Например, если в программе

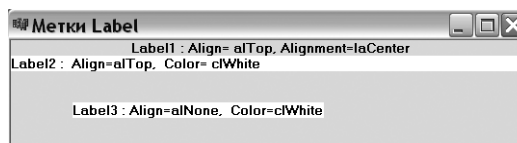


Рис. П2.5. Варианты оформления текстов в **Label**

имеется целая переменная `var`, отображающая число сотрудников некоторой организации, то вывести в метку **Label1** информацию об этом можно оператором:

```
Label1 -> Caption = «Число сотрудников: » + IntToStr(var) + «человек».
```

Во всех компонентах цвет фона определяется свойством **Color**, а цвет надписи — подсвойством **Color** свойства **Font**. Например, в большинстве меток задается цвет фона **clWhite** — белый. Если цвет специально не задавать, то цвет фона обычно сливается с цветом контейнера, содержащего метку, так что фон просто не заметен.

Для метки **Label** цвет и шрифт — единственно доступные элементы оформления надписи. Компоненты **StaticText** и **Panel** имеют, кроме того, свойство **BorderStyle**, определяющее рамку текста — бордюр. При стиле **sbsNone** метка **StaticText** по виду не отличается от метки **Label**. Вероятно, если уж использовать бордюр, то наиболее приятным будет стиль **sbsSunken**. Некоторые из вариантов оформления текста в метках **StaticText** представлены на рис. П2.6.

Компонент **Panel** кроме свойства **BorderStyle** имеет еще свойства **BevelInner**, **BevelOuter**, **BevelWidth**, **BorderWidth**, которые предоставляют богатые возможности оформления надписи. Таким образом, с точки зрения оформления выводимого текста максимальные возможности дает **Panel** и минимальные — **Label**. На рис. П2.7 представлены некоторые из вариантов оформления текста в **Panel**.

Размер меток **label** и **StaticText** определяется также свойством. **AutoSize**. Если это свойство установлено в **true**, то вертикальный и горизонтальный размеры компонента определяются размером надписи. Если же **AutoSize** равно **false**, то выравнивание текста внутри компонента определяется свойством **Alignment**, которое позволяет выравнивать текст по левому краю, правому краю или центру клиентской области метки. В панели **Panel** также имеется свойство **AutoSize**, но оно не относится к размерам подписи **Caption**. Однако свойство выравнивания **Alignment** работает и для панели.

В метке **Label** имеется свойство **WordWrap** — доступность переноса слов длинной надписи, превышающей длину компонента, на новую строку. Чтобы такой перенос мог осуществляться, надо установить свойство **WordWrap** в **true**, свойство **AutoSize** в **false** (чтобы размер компонента не определялся размером надписи) и сделать высоту компонента такой, чтобы в нем могло поместиться несколько строк. Если **WordWrap** не установлено в **true** при **AutoSize** равном **false**, то длинный текст, не помещающийся в рамке метки, просто обрезается.

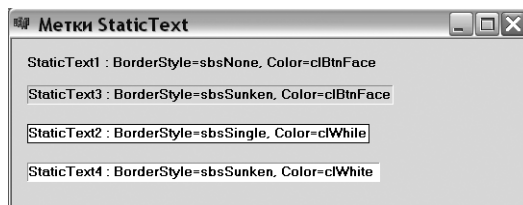


Рис. П2.6. Варианты оформления текста в **StaticText**



Рис. П2.7. Некоторые из вариантов оформления текста в **Panel**

В метке **StaticText** перенос длинного текста осуществляется автоматически, если значение **AutoSize** установлено в `false` и размер компонента достаточен для размещения нескольких строк. Для того чтобы в **StaticText** осуществлялся перенос при изменении пользователем размеров окна, надо осуществлять описанную выше перерисовку компонента методом **Repaint** в обработчике события формы **OnResize**.

В панели размещение надписи в нескольких строках невозможно.

Можно отметить еще одно свойство меток **Label** и **StaticText**, превращающее их в некоторое подобие управляющих элементов. Это свойство **FocusControl** — фокусируемый компонент. Если в свойстве метки `Caption` поместить перед одним из символов символ амперсанта «&», то символ, перед которым поставлен амперсанта, отображается в надписи метки подчеркнутым (сам амперсанта вообще не отображается). Если после этого обратиться к свойству метки **FocusControl**, то из выпадающего списка можно выбрать элемент, на который будет переключаться фокус, если пользователь нажмет клавиши ускоренного доступа: клавишу `<Alt+подчеркнутый символ>`. Подобные клавиши ускоренного доступа предусмотрены в управляющих элементах: разделах меню и кнопках. Благодаря свойству **FocusControl** метки могут обеспечить клавишами ускоренного доступа иные элементы, например, окна редактирования в режиме **ReadOnly**.

Окна редактирования Edit и MaskEdit

Примеры применения окон редактирования можно смотреть в программах, помещенных в следующей главе.

Внешнее оформление окон редактирования определяется свойством **BorderStyle**. В компонентах **Edit** и **MaskEdit** вводимый и выводимый текст содержится в свойстве **Text** и имеет тип **AnsiString**. Это свойство можно устанавливать в процессе проектирования или задавать программно. Выравнивание текста, как это имело место в метках и панелях, невозможно. Перенос строк тоже невозможен. Текст, не помещающийся по длине в окно, просто сдвигается и пользователь может перемещаться по нему с помощью курсора. Свойство **AutoSize** в окнах редактирования имеет смысл, отличный от смысла аналогичного свойства меток: автоматически подстраивается под размер текста только высота, но не ширина окна.

Окна редактирования снабжены многими функциями, свойственными большинству редакторов. Например, в них предусмотрены типичные комбинации «горячих» клавиш: `<Ctrl+C>` — копирование выделенного текста в буфер **Clipboard** (команда **Copy**), `<Ctrl+X>` — вырезание выделенного текста в буфер **Clipboard** (команда **Cut**), `<Ctrl+V>` — отмена последней команды редактирования.

Свойство **AutoSelect** определяет, будет ли автоматически выделяться весь текст при передаче фокуса в окно редактирования. Его имеет смысл задавать равным `true` в случаях, когда при переключении в данное окно пользователь будет вероятнее всего заменять текущий текст, а не исправлять его. Имеются также свойства только времени выполнения **SelLength**, **SelStart**, **SelText**, определяющие соответственно длину выделенного текста, позицию перед первым символом выделенного текста и сам выделенный текст. Например, если в окне имеется текст «выделение текста» и в нем пользователь выделил слово «текста», то **SelLength** = 6,

SelStart = 10 и **SelText** = «текста». Если выделенного текста нет, то свойство **SelStart** просто определяет текущее положение курсора.

Окна редактирования можно использовать и просто как компоненты отображения текста. Для этого надо установить в `true` их свойство **ReadOnly** и целесообразно установить **AutoSelect** в `false`. В этом случае пользователь не сможет изменять отображаемый текст и окно редактирования становится подобным меткам. Но имеются и определенные отличия. Во-первых, окна редактирования оформлены несколько иначе. А главное — окна редактирования могут вмещать текст, превышающий их видимую длину. В этом случае пользователь может прокручивать текст, перемещая курсор в окне. Такими особенностями не обладает ни одна метка. Некоторые из вариантов оформления окна **Edit** представлены на рис. П2.8.

Свойство **Text** окон редактирования имеет тип строки **AnsiString**. При вводе из окна числовой информации надо использовать функции **StrToFloat** — преобразование строки в значение с плавающей запятой, и **StrToInt** — преобразование строки в целое значение. Если вводимый текст не соответствует числу (например, содержит недопустимые символы), то функции преобразования генерируют исключение **EconvertError**. Поэтому в программе необходимо предусмотреть обработку этого исключения. Например:

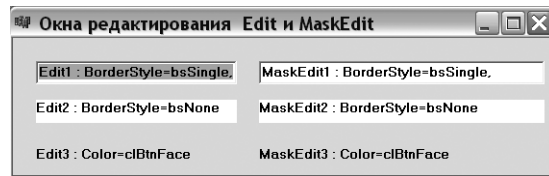


Рис. П2.8. Примеры оформления окна **Edit**

```
int var; // объявление переменной var
...
try
(
var = StrToInt(Edit1 ->Text);
)
catch (EconvertError&)
(
ShowMessage («Вы ввели ошибочное число; повторите ввод»);
)
```

Этот код обеспечивает сообщение пользователю об ошибке ввода и предотвращает ошибочные вычисления. Впрочем, окно **Edit** можно спроектировать так, что пользователь просто не сможет ввести в него неправильные символы. Свойство **MaxLength** определяет максимальную длину вводимого текста. Если **MaxLength** = 0, то длина текста не ограничена. В противном случае значение **MaxLength** указывает максимальное число символов, которое может ввести пользователь.

Свойство **Modified**, доступное только во время выполнения, показывает, проводилось ли редактирование текста в окне. Если вы хотите использовать это свойство, то в момент начала работы пользователя с текстом **Modified** надо установить в

`false`. Тогда при последующем обращении к этому свойству можно по его значению (`true` или `false`) установить, было или не было произведено редактирование.

Свойство **PasswordChar** позволяет превращать окно редактирования в окно ввода пароля. По умолчанию значение **PasswordChar** равно «#0» — нулевому символу. В этом случае это обычное окно редактирования. Но если в свойстве указать иной символ (например, символ звездочки «*»), то при вводе пользователем текста в окне будут появляться именно эти символы, а не те, которые вводит пользователь. Тем самым обеспечивается секретность ввода пароля.

Компонент **MaskEdit** отличается от **Edit** тем, что в нем можно задать строку маски в свойстве **EditMask**. Это позволяет обеспечить синтаксически безошибочный ввод пользователем таких данных, как номера телефонов, паспортные данные, адреса, даты, время и т. п. Маска состоит из трех разделов, которыми разделены точкой с запятой (;). В первом разделе — шаблоне записывается специальным образом символы, которые можно вводить в каждой позиции, и символы, добавляемые самой маской; во втором разделе записывается 1 или 0 в зависимости от того, надо или нет, чтобы символы, добавляемые маской, включались в свойство **Text** компонента; в третьем разделе указывается символ, используемый для обозначения позиций, в которых еще не осуществлен ввод. Прочитать результат ввода можно или в свойстве **Text**, которое в зависимости от вида второго раздела маски включает или не включает в себя символы маски, или в свойстве **Edit Text**, содержащем введенный текст вместе с символами маски. Некоторые примеры оформления окна **Mask Edit** представлены на рис. П2.8.

Вводить маску можно непосредственно в свойство **EditMask**. Но удобнее пользоваться специальным редактором масок **MaskEditor**, вызываемым при нажатии кнопки с многоточием в строке свойства **EditMask** в диспетчер объектов. В редакторе масок окно **SampleMasks** содержит наименования стандартных масок и примеры ввода с их помощью. В окне **InputMask** надо ввести маску. Если вы выбираете одну из стандартных масок, то окно **InputMask** автоматически заполняется и вы можете, если хотите, отредактировать эту маску. На рис. П2.9 представлено окно **MaskEditor**, в котором можно выполнять различные действия по редактированию масок.

Окно **Character for Blanks** определяет символ, используемый для обозначения позиций, в которых еще не осуществлен ввод (третий раздел маски). Индикатор *Save Literal Characters* определяет второй раздел маски: установлен, если второй раздел равен 1, и не установлен, если второй раздел равен 0.

Рассмотрим пример. Если с помощью **EditMask** надо ввести, например, целое число без знака, состоящее не более чем из двух цифр, можно задать маску «99;0;». Если число обязательно должно быть двузначным, то маска должна иметь вид «00;0;».

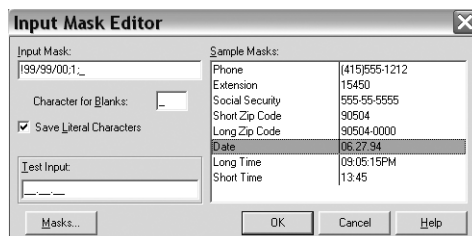


Рис. П2.9. Окно редактора **MaskEditor**

обязательно должно быть двузначным,

Многострочные окна редактирования Memo и RichEdit

Компоненты **Memo** и **RichEdit** являются окнами редактирования многострочного текста. Они так же, как и окно **Edit**, снабжены многими функциями, свойственными большинству редакторов. В них предусмотрены типичные комбинации «горячих» клавиш: <Ctrl+C> — копирование выделенного текста в буфер обмена **Clipboard** (команда **Copy**), <Ctrl+X> — вырезание выделенного текста в буфер **Clipboard** (команда **Cut**), <Ctrl+V> — вставка текста из буфера **Clipboard** в позицию курсора (команда **Paste**), <Ctrl+Z> — отмена последней команды редактирования.

В компоненте **Memo** формат (шрифт, его атрибуты, выравнивание) одинаков для всего текста и определяется свойством **Font**. При последующем чтении этого файла в **Memo** формат будет определяться текущим состоянием свойства **Font** компонента **Memo**, а не тем, в каком формате ранее вводился текст.

Компонент **RichEdit** работает с текстом в обогащенном формате **RTF**. При желании вы изменить атрибуты вновь вводимого фрагмента текста, для этого следует задать свойство **SelAttributes**. Это свойство типа **TextAttributes**, которое в свою очередь имеет подсвойства: **Color** (цвет), **Name** (имя шрифта), **Size** (размер), **Style** (стиль) и ряд других. Например, введите на форму компонент **RichEdit**, диалог выбора шрифта **FontDialog** со страницы библиотеки **Dialogs**, кнопку **Button**, которая позволит пользователю менять атрибуты текста и еще одну кнопку **Button** для завершения работы программы. В обработчик события щелчка кнопки мыши введите текст:

```
if(FontDialog ->Execute() )
RichEdit1 -> SelAttributes -> Assign(FontDialog1 -> Font);
RichEdit1 -> SetFocus();
```

Запустите приложение и увидите (рис. П2.10), что вы можете менять атрибуты текста, выполняя отдельные фрагменты различными шрифтами, размерами, цветами, стилями.

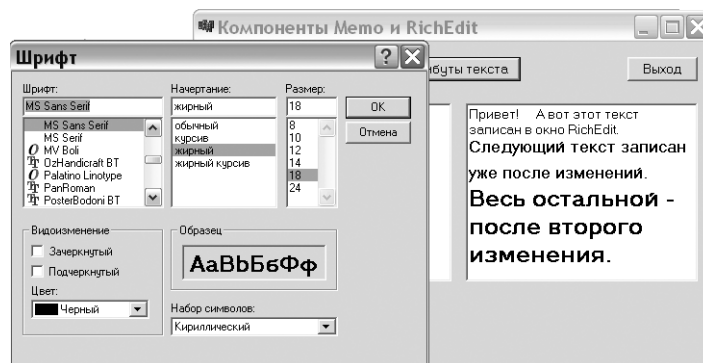


Рис. П2.10. Делаем выбор шрифта

Устанавливаемые атрибуты влияют на выделенный текст (рис. П2.11) или, если ничего не выделено, то на атрибуты нового текста, вводимого начиная с текущей позиции курсора (позиция курсора определяется свойством **SelStart**).

В компоненте имеется также свойство **DefAttributes**, содержащее атрибуты по умолчанию. Эти атрибуты действуют до того момента, когда изменяются атрибуты в свойстве **SelAttributes**. Но значения атрибутов в **DefAttributes** сохраняются и в любой момент эти значения могут быть методом **Assign** присвоены атрибутам свойства **SelAttributes**, чтобы вернуться к прежнему стилю.

Свойство **DelAttributes** доступно только во время выполнения. Поэтому его атрибуты при необходимости можно задавать, например, в обработчике события **OnCreate**.

Свойства **TabCount** и **Tab** имеют смысл при вводе текста только при значении свойства компонента **WantTabs** = **true**. Это свойство разрешает пользователю вводить в текст символ табуляции. Если **WantTabs** = **false**, то нажатие пользователем клавиши табуляции просто переключит фокус на очередной компонент и символ табуляции в текст не введется.

Это были основные отличия **Memo** и **RichEdit**. Теперь остановимся на общих свойствах этих окон редактирования.

Свойства **Alignment** и **WordWrap** имеют тот же смысл, что, например, в метках, и определяют выравнивание текста и допустимость переноса длинных строк. Установка свойства **ReadOnly** в **true** задает текст только для чтения. Свойство **MaxLength** определяет максимальную длину вводимого текста. Если **MaxLength** = 0, то длина текста не ограничена. Свойства **WantReturns** и **WantTab** определяют допустимость ввода пользователем в текст символов перевода строки и табуляции.

Свойство **ScrollBars** определяет наличие полос прокрутки текста в окне. По умолчанию **ScrollBars** = **ssNone**, что означает их отсутствие. Пользователь может в этом случае перемещаться по тексту только с помощью курсора. Можно задать свойству **ScrollBars** значения **ssHorizontal**, **ssVertical** или **ssBoth**, что будет соответственно означать наличие горизонтальной, вертикальной или обеих полос прокрутки.

Основное свойство окон **Memo** и **RichEdit** — **Lines**, содержащее текст окна в виде списка строк и имеющее тип **TStrings**. Начальное значение текста можно установить в процессе проектирования, нажав кнопку с многоточием около свойства **Lines** в окне диспетчера объектов. Перед вами откроется окно редактирования списков строк. Вы можете редактировать или вводить текст непосредственно в этом окне, или нажать кнопку **CodeEditor** (Редактора кода) и работать в обычном окне **CodeEditor**. В этом случае, завершив работу с текстом, выберите из контекстного меню, всплывающего при щелчке правой кнопкой мыши, команду **Close Page** и ответьте утвердительно на вопрос, хотите ли вы сохранить текст в соответствующем свойстве окна редактирования.

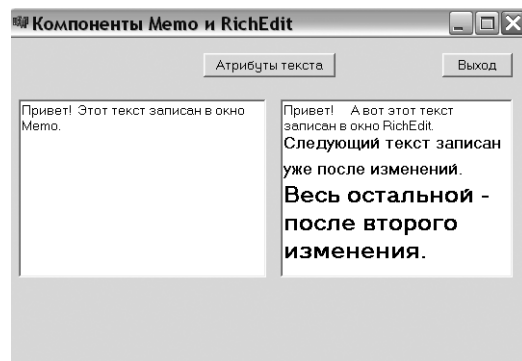


Рис. П2.11. Измененный текст

Во время выполнения приложения вы можете заносить текст в окно редактирования с помощью методов свойства **Lines** типа **Tstrings**. Весь текст, представленный одной строкой типа **String**, внутри которой используются разделители типа символов возврата каретки и перевода строки, содержится в свойстве **Text**.

Доступ к отдельной строке текста вы можете получить с помощью свойства **ansiString** `Strings[int Index]`. Индексы, как и везде в C++ Builder, начинаются с 0. Так что `Memo1->Lines->Strings[0]` — это текст первой строки. Учтите, что если окно редактирования изменяется в размерах при работе с приложением и свойство **WordWrap** = `true`, то индексы строк будут изменяться при переносах строк, так что в этих случаях индекс строки будет меняться.

Свойство только для чтения **Count** указывает число строк в тексте.

Для очистки текста в окне надо выполнить процедуру **Clear**. Этот метод относится к самому окну, а не к его свойству **Lines**.

Для занесения новой строки в конец текста окна редактирования можно воспользоваться методами **Add** или **Append** свойства **Lines**. Для загрузки текста из файла применяется метод **LoadFromFile**. Сохранение текста в файле осуществляется методом **SaveToFile**.

Свойство **SelStart** компонентов **Memo** и **RichEdit** указывает позицию курсора в тексте или начало выделенного пользователем текста. Свойство **CaretPos** указывает на структуру, поле **X** которой содержит индекс символа в строке, перед которым расположен курсор, а поле **Y** — индекс строки, в которой находится курсор. Таким образом, учитывая, что индексы начинаются с 0, значения `RichEdit1.CaretPos.y + 1` и `RichEdit1.CaretPos.x + 1` определяют соответственно номер строки и символа в ней, перед которым расположен курсор.

Компоненты выбора из списков — **ListBox** и **ComboBox**

Примеры компонентов **ListBox** и **ComboBox**, обеспечивающих выбор из списка, можно найти в последующих главах книги.

Компоненты **ListBox** и **ComboBox** отображают списки строк. Использование этих компонентов позволяет обеспечить безошибочный ввод информации пользователем в тех случаях, когда он должен выбрать ответ из конечного множества альтернатив. Компоненты списков отличаются друг от друга, прежде всего тем, что **ListBox** только отображает данные и позволяет пользователю выбрать из них то, что ему надо, а **ComboBox** позволяет также редактировать данные. Кроме того, различается форма отображения списков. **ListBox** отображает список в раскрытом виде и автоматически добавляет в список полосы прокрутки, если все строки не помещаются в окне компонента. **ComboBox** позволяет отображать список как в развернутом виде, так и в виде выпадающего списка, что обычно удобнее, так как экономит площадь окна приложения. О том, как выглядят эти компоненты на форме, показано на рис. П2.12.

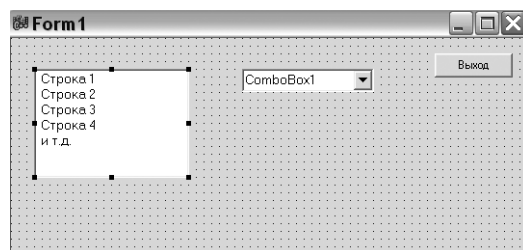


Рис. П2.12. Компоненты на форме

Основное свойство обоих компонентов, содержащее список, — **Items**, имеющее рассмотренный ранее тип **Tstrings**. Заполнить его во время проектирования можно, нажав кнопку с многоточием около этого свойства в окне Диспетчер объектов. При этом открывается окно редактирования. Каждая записанная в нем строка будет соответствовать строке списка.

Если **MultiSelect** = **false** (значение по умолчанию), то пользователь может выбрать только один элемент списка. В этом случае можно узнать индекс выбранной строки из свойства **ItemIndex**, доступного только во время выполнения. Если ни одна строка не выбрана, то **ItemIndex** = -1. Это означает, что ни один элемент списка не выбран. Если вы хотите задать этому свойству какое-то другое значение, т. е. установить выбор по умолчанию, который будет показан в момент начала работы приложения, то сделать это можно, например, в обработчике события **OnCreate** формы.

Если допускается множественный выбор (**MultiSelect** = **true**), то значение **ItemIndex** соответствует тому элементу списка, который находится в фокусе. При множественном выборе проверить, выбран ли данный элемент, можно проверив свойство **bool Selected[int Index]**.

На способ множественного выбора при **MultiSelect** = **true** влияет свойство **ExtendedSelect**. Если **ExtendedSelect** = **true**, то пользователь может выделить интервал элементов, выделив один из них, затем нажав клавишу <Sift> и переведя курсор к другому элементу. Выделить не прилегающие друг к другу элементы пользователь может, если будет удерживать во время выбора нажатой клавишу <Ctrl>. Если же **ExtendedSelect** = **false**, то клавиши <Shift> и <Ctrl> при выборе не работают.

Свойство **Columns** определяет число столбцов, в которых будет отображаться список, если он не помещается целиком в окне компонента **ListBox**.

Свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted** = **true** новые строки в список добавляются не в конец, а по алфавиту.

Свойство **Style**, установленное в **lbStandart** (значение по умолчанию) соответствует списку строк. Другие значения **Style** позволяют отображать в списке не только текст, но и изображения.

На рис. П2.13 показаны два рассматриваемых компонента в работающем приложении.

Рассмотрим теперь компонент **ComboBox**.

Выбор пользователя или введенный им текст можно определить по значению свойства **Text**. Если же надо определить индекс выбранного пользователя элемента списка, то можно воспользоваться обсуждавшимся в компоненте **ListBox** свойством **ItemIndex**, доступным только во время выполнения. Все сказанное ранее об **ItemIndex** и о задании его значения по умолчанию справедливо и для компонента **ComboBox**.

Причем для **ComboBox** задание начального значения **ItemIndex** еще актуальнее, чем для **ListBox**. Если начальное значение не задано, то в момент запуска приложе-

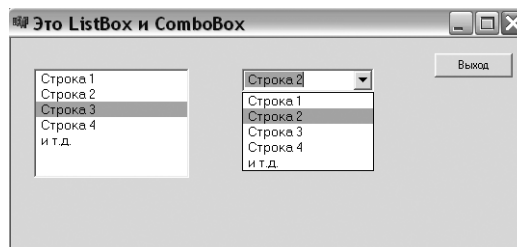


Рис. П2.13. Компоненты на окне работающего приложения

ния окно редактирования списка будет отображать на один из элементов списка, а значение свойства **Text**. Значит надо или вводить в приложение оператор, задающий в первый момент значение **ItemIndex**, или вводить во время проектирования в свойство **Text** какое-то приглашение к дальнейшим действием. Иначе пользователь будет в недоумении, что надо делать с этим списком.

Если в окне проводилось редактирование данных, то **ItemIndex** = -1. По этому признаку можно определить, что редактирование проводилось.

Свойство **MaxLength** определяет максимальное число символов, которые пользователь может ввести в окно редактирования. Если **MaxLength** = 0, то число вводимых символов не ограничено.

Как и в компоненте **ListBox**, свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted** = `true` новые строки в список добавляются не в конец, а по алфавиту.

Кнопки, индикаторы, управляющие элементы

Управляющие кнопки **Button** и **BitBtn**

Примеры кнопок **Button**, **BitBtn** и рассмотренной в следующем разделе кнопки **SpeedButton** встречаются очень часто во всех главах книги, поэтому делать для них специальный рисунок, по моему мнению, не стоит. Простейшей и, пожалуй, наиболее часто используемой кнопкой является кнопка **Button**, расположенная на странице библиотеки **Standard**. Реже используется кнопка **BitBtn**, отличающаяся, прежде всего, возможностью отобразить на ее поверхности изображение. Большинство свойств, методов и событий у этих видов кнопок одинаковы.

Основное с точки зрения внешнего вида свойство кнопки — **Caption** (надпись). В надписях кнопок можно предусматривать использование клавиш ускоренного доступа, выделяя для этого один из символов надписи. Перед символом, который должен соответствовать клавише ускоренного доступа, ставится символ амперсанта «&». Этот символ не появляется в надписи, а следующий за ним символ оказывается подчеркнутым. Тогда пользователь может вместо щелчка на кнопке нажать в любой момент клавишу <Alt> совместно с клавишей выделенного символа.

Например, если в вашем приложении имеется кнопка выполнения какой-то операции, вы можете задать ее свойство **Caption** равным «&Выполнить». На кнопке эта надпись будет иметь вид «Выполнить». И если пользователь нажмет клавиши <Alt+B>, то это будет эквивалентно щелчку на кнопке.

Основное событие любой кнопки — **OnClick**, возникающее при щелчке на ней. Именно в обработчике этого события записываются операторы, которые должны выполняться при щелчке пользователя на кнопке. Помимо этого есть еще ряд событий, связанных с различными манипуляциями клавишами и кнопками мыши. Свойство **Cancel**, если установить его в `true`, определяет, что нажатие пользователем клавиши <Esc> будет эквивалентно щелчку на данной кнопке. Это свойство целесообразно задавать равным `true` для кнопок Отменить в различных диалоговых окнах, чтобы можно было выйти из диалога, нажав на эту кнопку или нажав клавишу <Esc>.

Свойство **Default**, если его установить в `true`, определяет, что нажатие пользователем клавиши ввода <Enter> будет эквивалентно нажатию на данную кнопку, даже

если данная кнопка в этот момент не находится в фокусе. Правда, это сработает, если в фокусе находится какой-то оконный компонент. Если же в момент нажатия на клавишу <Enter> в фокусе находится другая кнопка, то все-таки сработает именно кнопка в фокусе.

Еще одно свойство — **ModalResult** используется в модальных формах. В обычных приложениях значение этого свойства должно быть равно **mrNone**.

Из методов, присущих кнопкам, имеет смысл отметить один — **Click**. Выполнение этого метода эквивалентно щелчку на кнопке, т. е. вызывает событие кнопки **OnClick**. Этим можно воспользоваться, чтобы продублировать какими-то другими действиями пользователя щелчок на кнопке. Пусть, например, вы хотите, чтобы при нажатии пользователем клавиши с символом «С» или «с» в любой момент работы с приложением выполнялись операции. Предусмотренные в обработчике события **OnClick** кнопки **Button1**. Поскольку неизвестно, какой компонент будет находиться в фокусе в момент этого события, надо перехватить его на уровне формы. Такой перехват осуществляется, если установить свойство формы **KeyPreview** в **true**. Тогда в обработчике события формы **OnKeyPresss** можно написать оператор

```
if((Key=='C') || (Key=='c')) Button1->Click( );
```

Если пользователь ввел символ «С» или «с», то в результате будет выполнен обработчик щелчка кнопки **Button1**.

Кнопка с фиксацией **SpeedButton**

Кнопки **SpeedButton** имеют возможность отображения пиктограмм и могут использоваться как обычные управляющие кнопки или как кнопки с фиксацией нажатого состояния. Обычно они используются в качестве быстрых кнопок, дублирующих различные команды меню, и в инструментальных панелях, в которых требуется фиксация нажатого состояния.

У кнопок **SpeedButton**, как и у других кнопок, имеется свойство **Caption** — надпись, но в этих кнопках оно обычно оставляется пустым, так как вместо надписи используется пиктограмма. Изображение на кнопке задается свойством **Glyph** точно так же, как для кнопок **BitBtn**. И точно так же свойство **NumGlyphs** определяет число используемых пиктограмм, свойства **Layout** и **Margin** определяют расположение изображения, а свойство **Spacing** — расстояние между изображением и надписью (если, конечно, вы все-таки хотите использовать надпись на кнопке).

Особенностью кнопок **SpeedButton** являются свойства **GroupIndex** (индекс группы), **AllowAllUp** (разрешение отжатого состояния всех кнопок группы) и **Down** (исходное состояние — нажатое). Если **GroupIndex** = 0, то кнопка ведет себя так же, как **Button** и **BitBtn**. При нажатии пользователем кнопки она погружается, а при отпускании возвращается в нормальное состояние. В этом случае свойства **AllowAllUp** и **Down** не влияют на поведение кнопки.

Если **GroupIndex** > 0 и **AllowAllUp** = **true**, то кнопка при щелчке пользователя на ней погружается и остается в нажатом состоянии. При повторном щелчке пользователя на кнопке она освобождается и переходит в нормальное состояние (именно для того, чтобы освобождение кнопки состоялось, необходимо задать **AllowAl-**

`!Up = true`). Если свойство **Dova** во время проектирования установлено равным `true`, то исходное состояние кнопки — нажатое.

Группы радиокнопок — компоненты **RadioGroup**, **RadioButton** и **GroupBox**

Радиокнопки образуют группы взаимосвязанных индикаторов, из которых обычно может быть выбран только один. Они используются для выбора пользователем одной из нескольких взаимоисключающих альтернатив. Впрочем, радиокнопки могут использоваться не только для выбора, но и для отображения аналогичных данных. В этом случае управление кнопками осуществляется программно. Примеры размещения радиокнопок вы можете увидеть на рис. П2.14.

Начнем рассмотрение радиокнопок с компонента **RadioGroup** — панели группы радиокнопок. Это панель, которая может содержать регулярно расположенные столбцами и строками радиокнопки. Надпись в левом верхнем углу панели (см. рис. П2.14) определяется свойством **Items**, имеющим тип **TStrings**. Щелкнув на кнопке с много-точием около этого свойства в окне диспетчера объектов, вы попадете в редактор списков строк. В этом редакторе вы можете занести надписи, которые хотите видеть около кнопок, по одной в строке. Сколько строчек вы запишете — столько и будет кнопок.

Кнопки, появившиеся в панели после задания значений **Items**, можно разместить в несколько столбцов (не более 17), задав свойство **Columns**. По умолчанию **Columns** = 1, т. е. кнопки размещаются друг под другом.

Определить, какую из кнопок выбрал пользователь, можно по свойству **ItemIndex**, которое показывает индекс выбранной кнопки. Индексы, как всегда в C++ Builder, начинаются с 0. По умолчанию **ItemIndex** = -1, что означает отсутствие выбранной кнопки. Если вы хотите, чтобы в момент начала выполнения приложения какая-то из кнопок была выбрана (это практически всегда необходимо), то надо установить соответствующее значение **ItemIndex** можно программно во время выполнения приложения.

В некоторых случаях желательно нерегулярное расположение кнопок. Такую возможность дают компоненты **RadioButton**, сгруппированные панелью **GroupBox**. Панель **GroupBox** выглядит на форме так же, как **RadioGroup**, и надпись в ее верхнем левом углу также определяется свойством **Caption**. Эта панель сама по себе пустая. Ее назначение — служить контейнером для других управляющих элементов, в частности, для радиокнопок **RadioButton**. Отдельная радиокнопка **RadioButton** особого смысла не имеет, хотя и может служить индикатором, включаемым и выключаемым пользователем. Но в качестве индикаторов обычно используются другие компоненты — **CheckBox**.



Рис. П2.14. Компоненты **GroupBox** и другие

Рассмотрим свойства радиокнопки **RadioButton**. Свойство **Caption** содержит надпись, появляющуюся около кнопки. Значение свойства **Alignment** определяет, с какой стороны от кнопки появляется надпись: **taLeftJustify** — слева, **taRightJustify** — справа (это значение принято по умолчанию). Свойство **Checked** определяет, выбрана данная кнопка пользователем или нет. Поскольку в начале выполнения приложения обычно надо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство **Checked** надо установить в `true` в процессе проектирования. Если вы поэкспериментируете, то заметите, что в `true` можно установить значение **Checked** только у одной кнопки из группы.

Радиокнопки **RadioButton** могут размещаться не только в панели **GroupBox**, но и в любой панели другого типа, а также непосредственно на форме. Группа взаимосвязанных кнопок в этих случаях определяется тем оконным компонентом, который содержит кнопки. В частности, для радиокнопок, размещенных непосредственно на форме, контейнером является сама форма. Таким образом, все кнопки, размещенные непосредственно на форме, работают как единая группа, т. е. только у одной из этих кнопок можно установить значение **Checked** в `true`.

Системные диалоги

В приложениях часто приходится выполнять стандартные действия: открывать и сохранять файлы, задавать атрибуты шрифтов, выбирать цвета палитры, производить контекстный поиск и замену и т. п.

Разработчики C++ Builder позаботились о том, чтобы включить в библиотеку простые для исполнения компоненты, реализующие соответствующие диалоговые окна. Они размещены на странице библиотеки **Dialogs**. В табл. П2.1 приведен перечень этих диалогов.

Таблица П2.1. Системные диалоги и их фрагменты

Компонент	Страница	Описание
OpenDialog «Открыть файл»	Dialogs	Предназначен для создания окна диалога «Открыть файл»
SaveDialog «Сохранить файл»	Dialogs	Предназначен для создания окна диалога «Сохранить файл»
OpenPictureDialog «Открыть рисунок»	Dialogs	Предназначен для создания окна диалога «Открыть рисунок», открывающего графический файл
SavePictureDialog «Сохранить рисунок»	Dialogs	Предназначен для создания окна диалога «Сохранить рисунок» — сохранение изображения в графическом файле
FontDialog «Шрифты»	Dialogs	Предназначен для создания окна диалога «Шрифты» — выбор атрибутов шрифта
ColorDialog «Цвет»	Dialogs	Предназначен для создания окна диалога «Цвет» — выбор цвета
PrintDialog «Печать»	Dialogs	Предназначен для создания окна диалога «Печать»
PrinterSetupDialog «Установка принтера»	Dialogs	Предназначен для создания окна диалога «Установка принтера»
FindDialog «Найти»	Dialogs	Предназначен для создания окна диалога «Найти» — контекстный поиск в тексте

Таблица П2.1. Окончание

Компонент	Страница	Описание
ReplaceDialog «Заменить»	Dialogs	Предназначен для создания окна диалога «Заменить» — контекстная замена фрагментов текста
FileListBox (список файлов)	Win 3.1	Отображает список всех файлов каталога
DirectoryListBox (структура каталогов)	Win 3.1	Отображает структуру каталогов диска
DriveComboBox (список дисков)	Win 3.1	Выпадающий список доступных дисков
FilterComboBox (список фильтров)	Win 3.1	Выпадающий список фильтров для поиска файлов
CDirectoryOutline (дерево каталогов)	Samples	Пример компонента, используемого для отображения структуры каталогов выбранного диска

Последние пять компонентов в табл. П2.1 являются не законченными диалогами, а их фрагментами, позволяющими строить свои собственные диалоговые окна.

Все диалоги являются невизуальными компонентами, так что место их размещения на форме не имеет значения. При обращении к этим компонентам вызываются стандартные диалоги, вид которых зависит от версии Windows и настройки системы. Так что при запуске одного и того же приложения на компьютерах с разными системами диалоги будут выглядеть по-разному. Например, при русифицированной версии Windows все их надписи будут русскими, а при англоязычной версии надписи будут на английском языке.

Основной метод, которым производится обращение к любому диалогу, — **Execute**. Эта функция открывает диалоговое окно и, если пользователь произвел в нем какой-то выбор, то функция возвращает `true`. При этом в свойствах компонента — диалога запоминается выбор пользователя, который можно прочесть и использовать в дальнейших операциях. Если же пользователь в диалоге нажал кнопку **Отмена** или клавишу <Esc>, то функция **Execute** возвращает `false`. Поэтому стандартное обращение к диалогу имеет вид:

```
If (<имя компонента - диалога> -> Execute())
    <оператор, использующий выбор пользователя>.
```

Главное меню — компонент MainMenu

В C++ Builder имеются два компонента, представляющие меню: **MainMenu** — главное меню, и **PopupMenu** — всплывающее меню. Оба компонента расположены на странице Standard. Эти компоненты имеют много общего. Начнем рассмотрение с компонента **MainMenu**. Это невизуальный компонент.

Основное свойство компонента — **Items**. Его заполнение производится с помощью конструктора меню, вызываемого двойным щелчком на компоненте **MainMenu** или нажатием кнопки с многоточием рядом со свойством **Items** в окне диспетчера объектов.

При выборе нового раздела вы увидите в диспетчере объектов множество свойств данного раздела. Дело в том, что каждый раздел меню, т. е. каждый эле-

мент свойства **Items**, является объектом типа **TMenuItem**, обладающим своими свойствами, методами, событиями.

Свойство **Caption** обозначает надпись раздела. Заполнение этого свойства подчиняется тем же правилам, что и заполнение аналогичного свойства в кнопках, включая использование символа амперсанта для обозначения клавиш быстрого доступа. Если вы в качестве значения **Caption** очередного раздела введете символ минус «-», то вместо раздела в меню появится разделитель.

Свойство **Name** задает имя объекта, соответствующего разделу меню. Очень полезно давать этим объектам осмысленные имена.

Свойство **Default** определяет, является ли данный раздел разделом по умолчанию своего подменю, т. е. разделом, выполняемым при двойном щелчке пользователя на родительском разделе.

Свойство **Checked**, установленное в true, указывает, что в разделе меню будет отображаться маркер флажка, показывающий, что данный раздел выбран.

Еще одним свойством, позволяющим вводить маркеры в разделы меню, является **RadioItem**. Это свойство, установленное в true, определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же значение свойства **GroupIndex**. По умолчанию значение **GroupIndex** равно 0. Для каждого раздела могут быть установлены во время проектирования или программно во время выполнения свойства **Enabled** (доступен) и **Visible** (видимый). Если установить **Enabled** = false, то раздел будет изображаться серой надписью, и не будет реагировать на щелчок пользователя. Если же задать **Visible** = false, то раздел вообще не будет виден, а остальные разделы сомкнутся, заняв место невидимого. Свойства **Enabled** и **Visible** используются для того, чтобы изменить состав доступных пользователю разделов в зависимости от режима работы приложения.

Начиная с C++ Builder 4, предусмотрена возможность ввода в разделы меню изображений. За это ответственны свойства разделов **Bitmap** и **ImageIndex**.

Мы рассмотрели все основные свойства объектов, соответствующих разделам меню. Основное свойство раздела — **OnClick**, возникающее при щелчке пользователя на разделе или при нажатии «горячих» клавиш быстрого доступа.

Способ объединения меню определяется свойством разделов **GroupIndex**. По умолчанию все разделы меню имеют одинаковое значение **GroupIndex**, равное нулю. Если требуется объединение меню, то разделам надо задать неубывающие номера свойств **GroupIndex**. Тогда, если разделы встраиваемого меню имеют те же значения **GroupIndex**, что и какие-то разделы основной формы, то эти разделы заменяют соответствующие разделы основного меню. В противном случае разделы вспомогательного меню встраиваются между элементами основного меню в соответствии с номерами **GroupIndex**. Если встраиваемый раздел имеет **GroupIndex** меньший, чем любой из разделов основного меню, то разделы встраиваются в начало.

На этом мы пока закончим очень краткое рассмотрение компонента **MainMenu**. Для большего количества информации смотрите информационные материалы в инструкциях для пользователей C++Builder.

Контекстное всплывающее меню — компонент **PopUpMenu**

Контекстное меню привязано к конкретным компонентам. Оно всплывает, если во время, когда данный компонент в фокусе, пользователь щелкнет правой кнопкой мыши. Обычно в контекстное меню включают те команды главного меню, которые в первую очередь могут потребоваться при работе с данным компонентом.

Контекстному меню соответствует компонент **PopUpMenu**. Поскольку в приложении может быть несколько контекстных меню, то и компонентов **PopUpMenu** может быть несколько. Оконные компоненты: панели, окна редактирования, а также метки и др. имеют свойство **PopUpMenu**, которое по умолчанию пусто, но куда можно поместить имя того компонента **PopUpMenu**, с которым будет связан данный компонент.

Формирование контекстного всплывающего меню производится с помощью конструктора меню, вызываемого двойным щелчком на **PopUpMenu**, точно так же, как это делалось для главного меню.

В остальном работа с **PopUpMenu** не отличается от работы с **MainMenu**. Только не возникает вопросов объединения меню разных форм: контекстные меню не объединяются.

Вместо заключения

На этом мы заканчиваем рассмотрение очень малой части компонентов библиотеки VCL. С каждой новой версией C++Builder число задействованных компонентов увеличивается, кроме того, многие разработчики сами создают компоненты собственной разработки, которые в больших количествах можно найти в Internet на сайтах фирм — разработчиков компьютерных программ. Если вы решили серьезно заняться программированием, то необходимо с самого начала собирать и сохранять всю попадающуюся вам в руки информацию о различных компонентах. Очень много различных компонентов можно найти у программистов, работающих с языком Паскаль. Дело в том, что Delphi, созданный на языке Паскаль, и C++ Builder, созданный на C/C++, являются «родными братьями» и за просто понимают языки друг друга.