Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА

Кафедра микропроцессорной техники
и информационно-управляющих систем

С. Н. ХАРЛАП, Н.В. РЯЗАНЦЕВА

Разработка приложений в среде C++Builder

Лабораторный практикум по дисциплине
«Программно-математическое обеспечение
микропроцессорных систем»

Гомель 2004

Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА

Кафедра микропроцессорной техники
и информационно-управляющих систем

С. Н. ХАРЛАП, Н.В. РЯЗАНЦЕВА

Разработка приложений в среде C++Builder

Лабораторный практикум по дисциплине
«Программно-математическое обеспечение
микропроцессорных систем»

Одобрено методической комиссией электротехнического факультета
Белорусского государственного университета транспорта

Гомель 2004

УДК 681.325.5 (076.5)
Х 211

Харлап С. Н., Рязанцева Н. В.

Х 211 Разработка приложений в среде *C++ Builder*: Лабораторный практикум по дисциплине “Программно-математическое обеспечение микропроцессорных систем” / Белорус. гос. ун-т трансп. – Гомель: БелГУТ, 2004. –с.

Раскрыты основные вопросы разработки приложений в интегрированной среде программирования *C++Bulder*. Приведены примеры использования визуальных компонентов, механизма *OLE*, *COM*-серверов и других элементов современных программных продуктов. Предназначены для студентов специальности “Автоматика, телемеханика и связь на транспорте”.

Рецензент –

ВВЕДЕНИЕ

Целью данного цикла лабораторных работ является изучение интегрированной среды программирования *C++ Builder* и получение практических навыков создания программного обеспечения. Описаны основные правила разработки программ под *Windows*.

Borland C++ Builder – выпущенное компанией *Inprise* средство быстрой разработки приложений, позволяющее создавать приложения на языке *C++*. При этом используется среда разработки и библиотека компонентов *VCL*, которые были разработаны для *Delphi*. В настоящем практикуме рассматривается среда разработки *C++ Builder* и основные приемы, применяемые при проектировании современных приложений для *Windows*.

Лабораторная работа № 1

ОБЗОР ОСНОВНЫХ СВОЙСТВ КОМПОНЕНТОВ C++ BUILDER

Цель работы. Ознакомиться с компонентами C++ Builder и изучить основные свойства компонентов.

1 Краткие сведения из теории

1.1 Среда разработки *C++ Builder*

C++ Builder представляет собой *SDI*-приложение, главное окно которого содержит настраиваемую инструментальную панель (слева) (рисунок 1.1) и палитру компонентов (справа). Помимо этого, по умолчанию при запуске *C++ Builder* появляются окно инспектора объектов (слева) и форма нового

приложения (справа). Под окном формы приложения находится окно редактора кода.

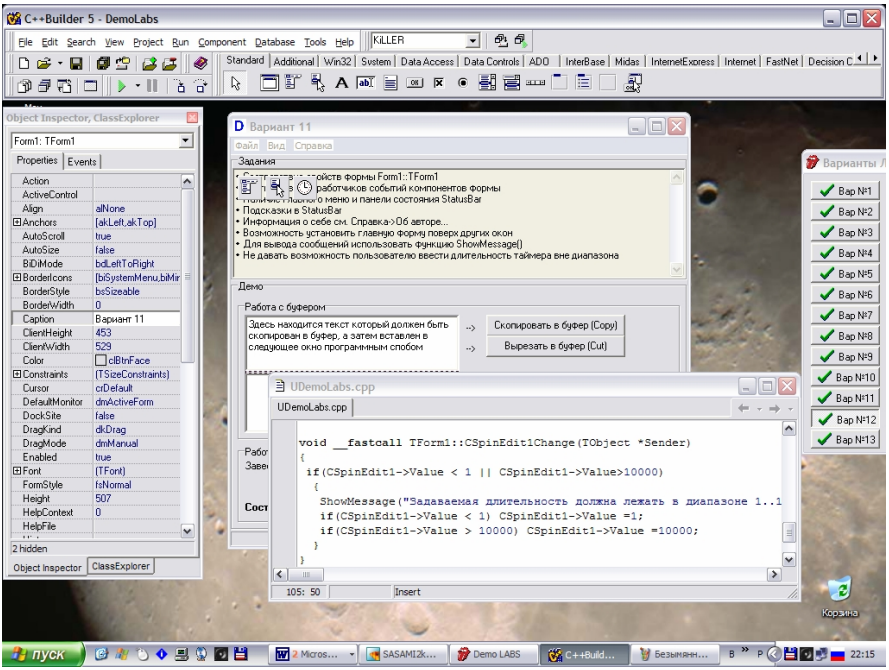

Рисунок 1.1 - Среда разработки *C++ Builder*

Формы являются основой приложений *C++Builder*. Создание пользовательского интерфейса приложения заключается в добавлении в окно формы элементов – объектов *C++Builder*, называемых компонентами. Компоненты *C++Builder* располагаются на палитре компонентов, выполненной в виде многостраничного блокнота. Важная особенность *C++Builder* состоит в том, что он позволяет создавать собственные компоненты и настраивать палитру компонентов, а также создавать различные версии палитры компонентов для разных проектов.

1.2 Создание приложений в *C++ Builder*

Первым шагом в разработке приложения *C++ Builder* является создание проекта. Файлы проекта содержат сгенерированный автоматически исходный текст, который становится частью приложения, когда оно скомпилиро-

вано и подготовлено к выполнению. Чтобы создать новый проект, нужно выбрать пункт меню *File/New Application.*

Следует запомнить, что *C++Builder* ассоциирует с каждым приложением три исходных файла со следующими именами по умолчанию:

• *Unit1.cpp* – хранит исполняемый код реализации вашего приложения. Именно в нем вы записываете обработчики событий, отвечающие за реакцию программы при воздействии пользователя на объекты компонентов;

• *Unit1.h* – содержит объявления всех объектов и их конструкторов;

• *Projectl.cpp* – обслуживает все объекты, заключенные в приложении. Любая новая форма, программный модуль или модуль данных автоматически включаются в проектный файл. Вы можете просмотреть в окне Редактора кода содержание исходного текста проектного файла с помощью команды главного меню *View* | *Project Source* или выбрав одноименную опцию из контекстного меню Администратора проекта. Ни в коем случае не редактируйте проектный файл вручную!

Кроме файла проекта *Project1.cpp C++ Builder* создает также проектный файл с именем по умолчанию *Project1.bpr* и файл *Project1.res,* который содержит иконку для проекта и создается автоматически. При внесении изменений в проект, таких как добавление новой формы, *C++ Builder* обновляет файл проекта.

Проект или приложение обычно имеют несколько форм. Добавление формы к проекту создает следующие дополнительные файлы:

– файл формы с расширением **.DFM,* содержащий информацию о ресурсах окон для конструирования формы;

– файл модуля с расширением **.CPP,* содержащий код на *C++*;

– заголовочный файл с расширением **.H*, содержащий описание класса формы. Когда вы добавляете новую форму, файл проекта автоматически обновляется.

1.3 Компоненты *C++* Builder

Компоненты разделяются на видимые (визуальные) и невидимые (невизуальные). Визуальные компоненты появляются во время выполнения точно так же, как и во время проектирования. Примерами являются кнопки и редактируемые поля. Невизуальные компоненты появляются во время проектирования, как пиктограммы на форме. Они никогда не видны во время выполнения, но обладают определенной функциональностью (например, обеспечивают доступ к данным, вызывают стандартные диалоги *Windows* и др.)

Для добавления компонента в форму можно выбрать мышью требуемый компонент в палитре и щелкнуть левой клавишей мыши в нужном месте

проектируемой формы. Компонент появится на форме, и далее его можно перемещать, менять размеры и другие характеристики (рисунок 1.2).

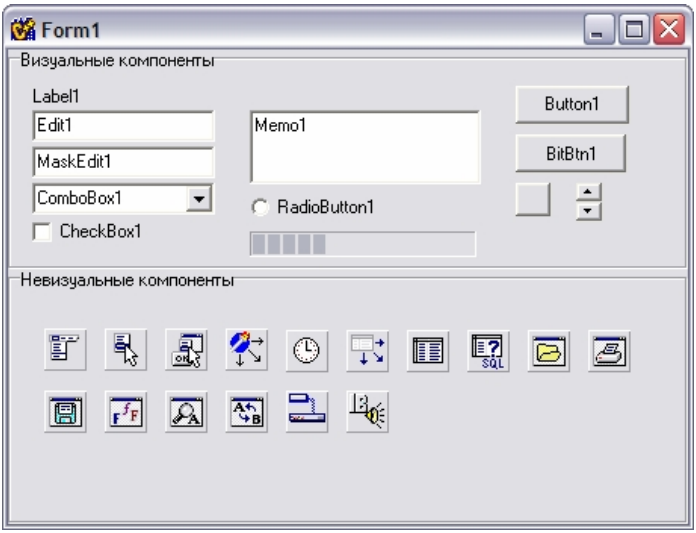

Рисунок 1.2 – Разновидности компонентов

Каждый компонент *C++ B*uilder имеет три типа характеристик: свойства, события и методы.

Если выбрать компонент из палитры и добавить его к форме, инспектор объектов автоматически покажет свойства и события, которые могут быть использованы с этим компонентом (рисунок 1.3). В верхней части инспектора объектов имеется выпадающий список, позволяющий выбирать нужный объект из имеющихся на форме.

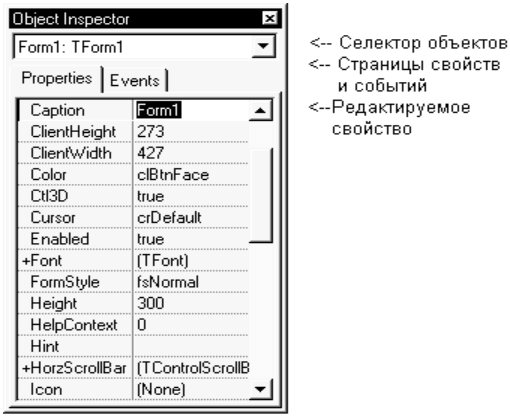

Рисунок 1.3 – Инспектор объектов

1.4 Свойства компонентов

Свойства являются атрибутами компонента, определяющими его внешний вид и поведение. Многие свойства компонента в колонке свойств имеют значение, устанавливаемое по умолчанию (например, высота кнопок).

Свойства компонента отображаются на странице свойств (*Properties*). Инспектор объектов отображает опубликованные (*published*) свойства компонентов. Помимо *published*-свойств, компоненты имеют общие (*public*) опубликованные свойства, которые доступны только во время выполнения приложения. Инспектор объектов используется для установки свойств во время проектирования. Список свойств располагается на странице свойств инспектора объектов. Можно определить свойства во время проектирования или написать код для видоизменения свойств компонента во время выполнения приложения.

При определении свойств компонента во время проектирования нужно выбрать компонент на форме, открыть страницу свойств в инспекторе объектов, выбрать определяемое свойство и изменить его с помощью редактора свойств (это может быть простое поле для ввода текста или числа, выпадающий список, раскрывающийся список, диалоговая панель и т.д.).

1.5 События

Страница событий (*Events*) инспектора объектов показывает список событий, распознаваемых компонентом (программирование для операционных систем с графическим пользовательским интерфейсом, в частности, для *Windows* предполагает описание реакции приложения на те или иные события, а сама операционная система занимается постоянным опросом компьютера с целью выявления наступления какого-либо события). Каждый компонент имеет свой собственный набор обработчиков событий. В *C++ Builder* следует писать функции, называемые обработчиками событий, и связывать события с этими функциями. Создавая обработчик того или иного события, вы поручаете программе выполнить написанную функцию, если это событие произойдет.

Для того чтобы добавить обработчик событий, нужно выбрать на форме с помощью мыши компонент, которому необходим обработчик событий, затем открыть страницу событий инспектора объектов и дважды щелкнуть левой клавишей мыши на колонке значений рядом с событием, чтобы заставить *C++ Builder* сгенерировать прототип обработчика событий и показать его в редакторе кода. При этом автоматически генерируется текст пустой функции, и редактор открывается в том месте, где следует вводить код. Курсор позиционируется внутри операторных скобок { ... } (рисунок 1.4). Далее нужно ввести код, который должен выполняться при наступлении события. Обработчик событий может иметь параметры, которые указываются после имени функции в круглых скобках.

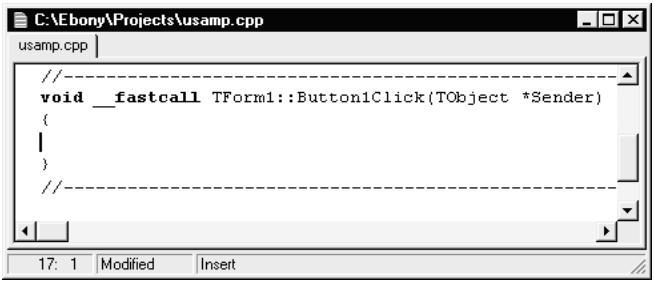

Рисунок 1.4 – Прототип обработчика событий

1.6 Методы

Метод является функцией, которая связана с компонентом, и которая объявляется как часть объекта. Создавая обработчики событий, можно вызывать методы, используя нотацию ->, например:

Edit1->Show();

Отметим, что при создании формы связанные с ней модуль и заголовочный файл с расширением **.h* генерируются обязательно, тогда как при создании нового модуля он может быть не связан с формой (например, если в нем содержатся процедуры расчетов). Имена формы и модуля можно изменить, причем желательно сделать это сразу после создания, пока на них не появилось много ссылок в других формах и модулях.

Большинство компонентов работают со строками, представленными в стандарте *AnsiString*, который рассматривается в следующем разделе.

1.7 Класс *AnsiString* языка C++

Этот расширенный тип *Delphi* объявлен в библиотеке *C++Builder* как одноименный класс, наследующий функциональность родителя. Видимо, из-за того, что название *AnsiString* не кажется особенно привлекательным, пользователям *C++Builder* рекомендуется простой синоним – класс *String* (имя пишется с заглавной буквы).

Класс *String* имеет ряд переопределенных операторов присваивания (=), конкатенации (+) (+=), сравнения (= =) (<) (<=) (>) (>=) (! =), характерных для строчных классов вообще. При сравнении разнотипных объектов неявное преобразование типа выполняется автоматически. Класс *String* не имеет своих свойств, однако имеет исчерпывающий набор функций, которые позволяют реализовать практически любые мыслимые операции над строками. Для выполнения основных операций над строчными объектами служат функции-члены класса, представленные в таблице 1.1.

Таблица 1.1

Метод	Действие
c_str ()	Возвращает указатель char* на символьный массив, содержащий ту же информацию, что и исходная строка
Length	Возвращает длину текста в строке (в байтах)
IsEmpty	Возвращает значение true, если строка пуста, т. е. когда Length = 0
Insert	Вставляет текст в указанную позицию строки
Delete	Стирает указанное число символов из строки
Pos	Возвращает указатель на заданную комбинацию символов в контексте строки
LastDelimiter	Возвращает указатель на последний встреченный символ-ограничитель
LowerCase	Возвращает новую версию исходной строки, все символы которой заменяются прописными буквами нижнего регистра клавиатуры
Uppercase	Возвращает новую версию исходной строки, все символы которой заменяются заглавными буквами верхнего регистра клавиатуры
Trim	Возвращает новую версию исходной строки, очищенную от управляющих символов, ведущих и хвостовых пробелов
ToInt	Преобразование строки к целочисленному значению
ToDouble	Преобразование строк вида –123; 3.14159 к двоичному эквиваленту числа с плавающей точкой
Substring	Создает новую строку символов, выбранных из исходного контекста

Перед использованием класса в приложении необходимо включить в заголовок кодового модуля файл предкомпиляции *cstring.h*, в котором этот класс объявлен. В практике программирования часто бывает необходимо выполнять *преобразование численных значений*, вводимых пользователем в объекты редактирования текста (например, в *TEdit* вкладки *Standard Палитры Компонент*). Существует по меньшей мере два способа корректной записи такого преобразования:

```
String str;
str = Editl–>Text;
int vail = atoi(str.c_str()); // функцией atoi
int val2 = str.ToInt(); // методом ToInt
```

Следует упомянуть особенность класса *String*: элементы его объектных массивов пронумерованы начиная с 1 (как принято в Паскале), а не с 0 (как принято в *C++*).

2 Индивидуальное задание

Изучить среду *Borland C++ Builder* и разработать простейшее приложение. Индивидуальное задание для разработки выдается преподавателем.

3 Порядок выполнения работы

3.1 Изучить краткие сведения из теории.
3.2 Разработать экранную форму.
3.3 Установить реакции на объекты.
3.4 Написать программу – обработчик событий.
3.5 Откомпилировать и отладить набранную программу.
3.6 Оформить отчет по лабораторной работе.

4 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, наименование и цель работы; индивидуальное задание; экранные формы разработанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

Лабораторная работа № 2

ИЗУЧЕНИЕ БИБЛИОТЕКИ ВИЗУАЛЬНЫХ КОМПОНЕНТОВ VCL

Цель работы. Изучить основные компоненты C++ Builder и научиться создавать приложения ориентированные на различные задачи.

1 Краткие сведения из теории

1.1 Библиотека Визуальных Компонентов *VCL*

Библиотека Визуальных Компонентов *VCL* приобрела статус нового промышленного стандарта и в настоящее время применяется более чем по-

лумиллионом пользователей, существенно ускоряя разработку надежных приложений любой степени сложности. VCL содержит около 100 повторно используемых компонент, которые реализуют все элементы пользовательского интерфейса операционной системы Windows. Кроме того, VCL предоставляют в распоряжение программистов такие оригинальные объекты, как записные книжки с закладками, табличные сетки для отображения содержимого баз данных и даже органы управления устройствами мультимедиа. Находясь в среде объектно-ориентированного Программирования C++Builder, компоненты можно использовать непосредственно, менять их свойства, облик и поведение или порождать производные элементы, обладающие нужными отличительными характеристиками.

Компоненты *C++ Builder* располагаются на Палитре компонентов, выполненной в виде многостраничного блокнота. Палитра компонентов – это визуальная опись библиотеки визуальных компонентов – *Visual Component Library* (*VCL*). Она позволяет сгруппировать визуальные компоненты в соответствии с их смыслом и назначением.

Набор и порядок компонентов на каждой странице являются конфигурируемыми. Так, Вы можете добавить к имеющимся компонентам новые, изменить их количество и порядок.

1.2 Стандартные компоненты

На первой странице *Палитры Компонентов* размещены основные объекты *Windows* – кнопки, списки, окна ввода и т. д.

Рассмотрим назначение этих компонентов.

- *TMainMenu* позволяет создавать двухуровневые меню, располагающиеся в верхней части окна. Чтобы заполнить пункты меню, необходимо дважды щелкнуть по компоненту на форме. Чтобы задать горизонтальную черту, разделяющую группы пунктов меню, в названии пункта необходимо ввести знак «-».

Каждый пункт меню является экземпляром класса *TMenuItem*, имеющим свои свойства. Рассмотрим наиболее важные свойства класса *TMenuItem*:

Caption – название пункта меню;

Checked – установка признака того, что пункт меню выбран;

Default – установка признака того, что пункт меню будет выбран по умолчанию;

Enabled – установка признака того, что пункт меню будет доступен во время выполнения проекта;

GroupIndex – идентифицирует логическую группу, к которой пункт меню принадлежит;

Name – идентификатор пункта меню;

ShortCut – назначение «горячих» клавиш пункту меню.

Для создания подменю необходимо щелкнуть правой кнопкой мыши на том пункте меню, к которому требуется подключить подменю, и выбрать пункт *Create SubMenu* или нажать клавиши *Ctrl*+→.

Нажав правую кнопку мыши на пункте меню можно сохранить этот пункт как шаблон для повторного использования (*Save As Template …*) или использовать уже готовый шаблон (*Insert From Template …*).

Для создания обработчика событий необходимо щелкнуть по нужному пункту меню. Меню подключается к форме автоматически. При запуске приложения, оно появляется в верхней строке.

Свойство *AutoMerge* вместе с методами *Merge* и *Unmerge* управляют процессом слияния меню разных форм.

- *TPopupMenu* позволяет создавать специальное всплывающие меню для формы или для другой компоненты. Именно для этой цели большое количество компонент имеет свойство *PopupMenu*, в котором можно задать ссылку на связанное с ними меню.

Если требуется, чтобы специальное меню появлялось при нажатии правой кнопки мыши на форму или другой элемент, с которым связан данный компонент, установите свойство *AutoPopup* в значение *true*. Работающее таким образом специальное меню называется контекстным. С помощью обработчика события *OnPopup* можно определить процедуру, которая будет выполняться непосредственно перед появлением специального меню.

Каждый пункт меню также является экземпляром класса *TMenuItem*. Поэтому все действия по организации меню *TMainMenu* полностью распространяются и на *TPopupMenu*. Для подключения меню к форме необходимо присвоить свойству *PopupMenu* формы имя компонента *TPopupMenu.*

- *TLabel* служит для отображения на форме прямоугольной области статического текста, который нельзя редактировать. Рассмотрим наиболее важные свойства класса *TLabel*:

Align – выравнивание объекта на форме;

Alignment – выравнивание текста по горизонтали;

Caption – текст, отображаемый на экране;

Color – цвет фона;

Font – определяет свойства шрифта;

Hint – текст подсказки, который будет появляться когда курсор мыши подводится к объекту, если свойство *ShowHint* имеет значение *true*;

Layout – выравнивание текста по вертикали;

PopupMenu – идентификатор всплывающего меню, которое будет активизироваться правой кнопкой мыши на объекте *TLabel*;

Visible – видимость объекта.

Чтобы размер шрифта автоматически соответствовал максимальному заполнению области, необходимо установить свойство *AutoSize* в значение *true*. Чтобы весь текст можно было увидеть внутри короткой области, требуется задать свойству *WordWrap* значение *true*. Установкой значения *true* свойства *Transparent* можно оставить видимой часть другого компонента сквозь название, расположенное прямо на нем.

- *TEdit* – стандартный управляющий элемент *Windows* для ввода информации. Он может быть использован для отображения короткого фрагмента текста и позволяет пользователю вводить текст во время выполнения программы. Объект *TEdit* имеет большое количество свойств, совпадающих со свойствами *TLabel***.** Кроме того он обладает следующими дополнительными свойствами:

PasswordChar – содержит символ, который будет отображаться при вводе текста. Позволяет организовать ввод паролей;

ReadOnly – запрещает редактирование текста;

Text – содержит текст, введенный в *TEdit*. Текст хранится в формате *AnsiString*.

- *TMemo* – отображает прямоугольную область для ввода и редактирования текста. Подразумевает работу с большими текстами. *TMemo* может переносить слова, сохранять в *Clipboard* фрагменты текста и восстанавливать их, выполнять другие функции редактора. *TMemo* имеет ограничения на объем текста в 32 Кб, это составляет 10–20 страниц.

Объект *TMemo* имеет большое количество свойств, совпадающих со свойствами *TEdit***.** Кроме того, он имеет свойство *Lines*, которое позволяет обрабатывать текст как массив строк.

- *TButton* позволяет выполнить какие-либо действия при нажатии кнопки во время выполнения программы. Поместив *TButton* на форму, Вы по двойному щелчку можете создать заготовку обработчика события нажатия кнопки. Далее нужно заполнить заготовку кодом.
- *TCheckBox* отображает строку текста с маленьким окошком рядом. В окошке можно поставить отметку о выборе какого-либо условия. Объект *TCheckBox* имеет свойство *Checked*, которое указывает, выбран ли объект.
- *TRadioButton* позволяет создать радио-кнопку. Обычно радио-кнопки размещаются внутри предварительно установленного на форме группового контейнера. Если выбрана одна кнопка, выбор всех прочих кнопок той же группе автоматически снимается. Например, две радио-кнопки на форме могут быть выбраны одновременно только в том случае, когда они размещены в разных контейнерах. Если группировка радио-кнопок явно не задана, то по умолчанию, все они группируются в одном из оконных

контейнеров (*TForm, TGroupBox* или *TPanel*).

- *TListBox* предназначен для отображения строчных списков на форме.

Компонент обладает всеми свойствами класса *TStringList* и имеет несколько дополнительных свойств.

Items – массив, содержащий строки, которые появляются в окне списка. *Items* – одно из основных свойств *TListBox,* используемых для работы со списком. *Items* используется, чтобы добавлять, вставлять, удалять и перемещать записи. По умолчанию, записи в окне списка имеют тип *TStrings*. Основные методы класса *TStrings* приведены в таблице 5.1. *Items* используется для обращения к методам и редактирования записей в списке, например:

ListBox1->Items->AddStrings(SringList1);

ItemIndex – возвращает номер выбранного элемента списка. Нумерация элементов списка начинается с 0.

BorderStyle – изменяет вид внешней границы списка. Список может иметь или не иметь границы.

Columns – определяет число колонок в списке. Список имеет горизонтальную полосу прокрутки, которая позволяет пользователям просматривать колонки. Ширина каждой колонки зависит от ширины списка и числа колонок.

Таблица 2.1

Метод	Назначение
Add	Добавляет элемент к концу списка
AddStrings	Добавляет группу строк, взятых из другого списочного объекта
Clear	Очищает список
Delete	Удаляет элемент с указанным номером
Exchange	Меняет два элемента местами
Find	Выдает номер строки, под которым она добавляется в сортированный список. Метод возвращает значение *false*, если такой строки еще нет в списке
IndexOf	Возвращает номер строки, под которым она впервые встретилась в списке (как сортированном, так и нет)
Insert	Вставляет элемент в указанную позицию
LoadFrom-File	Загружает строки списка из текстового файла
Move	Перемещает элемент из одной позиции в другую
SaveToFile	Сохраняет список в текстовом файле
Sort	Сортирует список

ExtendedSelect – определяет, может ли пользователь выбирать диапазон значений в списке. Если *ExtendedSelect* и свойство *MultiSelect* имеют значение «истина», то пользователь может выбрать несколько элементов списка. Если *ExtendedSelect* имеет значение «ложно», а *MultiSelect* – «истина», то пользователь может выбирать элементы без *Shift* или *Ctrl* клавиш, но не сможет выбирать диапазон. Если *MultiSelect* имеет значение «ложно», то *ExtendedSelect* не будет иметь никакого эффекта, поскольку пользователь не будет способен выбрать больше чем один элемент списка одновременно.

SelCount – указывает количество выбранных элементов списка, если свойство *MultiSelect* имеет значение «истина».

Selected – массив булевых переменных, указывающих, выбран ли элемент списка с заданным номером. Используется для определения индексов выбранных строк, если свойство *MultiSelect* имеет значение «истина».

Sorted – включает сортировку списка по алфавиту.

IntegralHeight – устанавливает высоту списка.

- *TComboBox* – комбинация области редактирования и выпадающего списка текстовых вариантов для выбора, во многом напоминает *ListBox*, за исключением того, что позволяет вводить информацию в маленьком поле ввода сверху. Есть несколько типов *ComboBox*, но наиболее популярен выпадающий вниз (*drop-down combo box*), который можно видеть внизу окна диалога выбора файла.

Значение свойства *Text* заносится непосредственно в область редактирования. Элементы списка, которые может выбирать пользователь, содержатся в свойстве *Items*, номер элемента, который будет выбран во время выполнения программы, – в свойстве *Itemlndex*, а сам выбранный текст – в свойстве *SelText*. Свойства *SelStart* и *SelLength* позволяют установить выборку части текста или обнаружить, какая часть текста выбрана.

Можно динамически добавлять, вычеркивать, вставлять и перемещать элементы списка с помощью методов *Add, Append, Delete* и *Insert* объекта *Items*, например:

ComboBoxl->Items->Insert(0, "Первый элемент списка");

- *TScrollbar* – линейка прокрутки с бегунком для просмотра содержимого окна, формы или другого компонента, например, для перемещения внутри заданного интервала значений некоторого параметра.

Поведение прокручиваемого объекта определяется обработчиком события *OnScroll*. Насколько должен продвинуться бегунок, когда пользователь щелкает мышью на самой линейке (по обеим сторонам от бегунка), определяет значение свойства *LargeChange*. Насколько должен продвинуться бегунок, когда пользователь щелкает мышью по кнопкам со стрелками (на концах

линейки) или нажимает клавиши позиционирования, определяет значение свойства *SmallChange*

Значения свойств *Min* и *Max* устанавливают интервал допустимых перемещений бегунка. Ваша программа может установить бегунок в нужную позицию, определяемую значением свойства *Position*. Метод *SetPcirums* определяет значения всех свойств *Min*, *Max* и *Position* одновременно.

- *TGroupBox* создает контейнер в виде прямоугольной рамки, визуально объединяющий на форме логически связанную группу некоторых интерфейсных элементов. Используется для визуальных целей и для указания *Windows*, каков порядок перемещения по компонентам на форме (при нажатии клавиши *TAB*).
- *TRadioGroup* создает контейнер в виде прямоугольной рамки, визуально объединяющий на форме группу логически взаимоисключающих радиокнопок. Все радио-кнопки в группе связаны между собой, при выборе одной из кнопок, остальные будут не активны. Количество кнопок определяется количеством строк с названиями кнопок, описанных в свойстве *Items*, номер выбранной кнопки хранится в свойстве *ItemIndex*.
- *TPanel* – управляющий элемент, похожий на *TGroupBox*, используется в декоративных целях. Чтобы использовать *TPanel*, просто поместите его на форму и затем положите другие компоненты на него. Теперь при перемещении *TPanel* будут передвигаться и эти компоненты. *TPanel* используется также для создания линейки инструментов и окна статуса.

1.3 Краткий обзор других компонентов

Компоненты вкладки *Win32:*

- *TTabControl* отображает набор частично перекрывающих друг друга картотечных вкладок. Названия вкладок вводятся в список свойства *Tabs*. Если все поля не умещаются на форме в один ряд, то можно установить свойство *MultiLine* в значение *true* или прокручивать вкладки с помощью кнопок со стрелками. Свойство *TabIndex* содержит номер выбранной вкладки. Установка свойства *Enabled* в значение *false* запретит выборку отдельных вкладок.
- *TPageControl* отображает набор полей, имеющих вид частично перекрывающих друг друга картотечных вкладок, для организации многостраничного диалога. Чтобы создать новую страницу диалога с соответствующей вкладкой, выберите опцию *New Page* из контекстного меню данного компонента. Можно активизировать конкретную страницу с помощью мыши, выбрав ее из выпадающего списка свойства *ActivePage* или перелистывая вкладки с помощью опций *Next Page* и *Previous Page* контекстного меню. Свойство *PageIndex* содержит номер активной страни-

цы. Установкой свойства *TabVisible* в значение *false* можно сделать эту страницу невидимой.

- *TTreeView* отображает древовидный перечень элементов – заголовков документов, записей в указателе, файлов или каталогов на диске.
- *TListView* отображает древовидный перечень элементов в различных видах – по столбцам с заголовками, вертикально, горизонтально, с пиктограммами.
- *TImageList* создает контейнер для группы изображений.
- *THeaderControl* создает контейнер для заголовков столбцов.
- *TRichEdit* отображает область редактируемого ввода множественных строк информации в формате RTF.
- *TStatusBar* создает строку панелей состояния для отображения статусной информации.
- *TTrackBar* создает шкалу с метками и регулятором текущего положения.
- *TProgressBar* создает индикатор процесса выполнения некоторой процедуры в приложении.
- *TUpDown* создает спаренные кнопки со стрелками "вверх" и "вниз". Нажатие этих кнопок вызывает увеличение или уменьшение значения свойства *Position*.
- *THotKey* используется для установки клавиш быстрого вызова во время выполнения программы.

Компоненты вкладки *Additional:*

- *TBitBtn* создает кнопку с изображением битового образа.
- *TSpeedButton* создает графическую кнопку быстрого вызова.
- *TMaskEdit* создает область редактируемого ввода данных специфического формата.
- *TStringGrid* создает сетку для отображения строк по строкам или столбцам.
- *TDrawGrid* создает сетку для отображения графических данных по строкам или столбцам.
- *TImage* создает на форме контейнер для отображения битового образа, пиктограммы или метафайла.
- *TShape* рисует простые геометрические фигуры.
- *TBevel* создает линии и рамки с объемным видом.
- *TScrollBox* создает контейнер переменного размера с линейками прокрутки, если это необходимо. Пока Вы в явном виде не отключите эту возможность, форма сама по себе действует так же. Однако могут быть случаи, когда понадобится выполнить скроллинг только части формы. В таких случаях используется *TScrollBox*.

2 Индивидуальное задание

Изучить компоненты VCL и разработать приложение в соответствии с индивидуальным заданием. Индивидуальное задание для разработки выдается преподавателем. Оформить отчет по лабораторной работе.

3 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, наименование и цель работы; индивидуальное задание; экранные формы разработанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

Лабораторная работа № 3

ОБРАБОТКА ИСКЛЮЧЕНИЙ

Ц е л ь р а б о т ы. Изучить способы обработки исключений при некорректном вводе информации.

1 Краткие сведения из теории

1.1 Механизм обработки исключений в *C++Builder*

При разработке приложений часто возникают ситуации, при которых ответственность за правильность выполнения операций, операторов и даже отдельных функций целиком возлагается на программиста. Арифметические вычисления (деление на нуль), преобразования типа, работа с индексами и адресами, корректная формулировка условий в операторах управления, работа с потоками ввода-вывода – это далеко не полный перечень неконтролируемых в *C++* ситуаций.

Ошибки времени выполнения, возникающие непосредственно в ходе выполнения программы, в терминах объектно-ориентированного программирования называются исключительными ситуациями. Исключительные

ситуации – это события, которые прерывают нормальный ход выполнения программы

Исключение – это объект специального вида, характеризующий возникшую в программе исключительную ситуацию.

Различают синхронные и асинхронные исключительные ситуации.

Синхронная исключительная ситуация возникает непосредственно в ходе выполнения программы, причем ее причина заключается непосредственно в действиях, выполняемых самой программой.

Асинхронные исключительные ситуации непосредственно не связаны с выполнением программы. Их причинами могут служить аппаратно возбуждаемые прерывания (например, сигналы от таймера), сообщения, поступающие от внешних устройств или даже от локальной сети.

Язык *C++* определяет стандарт обслуживания исключений в рамках объектно-ориентированного программирования. *C++Builder* предусматривает специальные механизмы для обработки исключений (ошибок), которые могут возникнуть при использовании Библиотеки Визуальных Компонентов. *C++Builder* также поддерживает обработку исключений самой операционной системы и модель завершения работы приложения.

Когда во время выполнения программы встречается ненормальную ситуацию, на которую она не была рассчитана, можно передать управление другой части программы, способной справиться с этой проблемой, и либо продолжить выполнение, либо завершить работу. В точке возникновения исключения автоматически собирается информация, которая может оказаться полезной для диагностики причин, приведших к нарушению нормально хода выполнения программы. Можно определить обработчик исключения – специальный фрагмент кода, выполняющий необходимые действия перед завершением программы.

Обслуживаются только так называемые синхронные исключения, которые возникают внутри программы. Такие внешние события, как нажатие клавиш *Ctrl+C*, исключениями не считаются.

Наиболее приемлемый путь борьбы с исключениями – отлавливание и обработка их с помощью блоков *try…catch*. Синтаксис этих блоков следующий:

```
try
{
    Критический код, выполнение которого может привести к возникновению ошибки времени выполнения;
}
catch (TypeToCatch ExceptionToCatch1)
{
```

```
    Код, исполняемый при возникновении исключения
    ExceptionToCatch1;
}
catch (TypeToCatch ExceptionToCatch2)
{
    Код, исполняемый при возникновении исключения
    ExceptionToCatch2;
}
catch (...)
{
    Код, исполняемый при возникновении остальных исключений;
}
__finally
{
    Код, выполняющийся всегда, независимо от того, произошло исключение или нет;
}
```

Блок кода, который может сгенерировать исключение, начинается ключевым словом *try* и заключается в фигурные скобки. Если блок *try* обнаруживает исключение внутри этого блока, происходит программное прерывание и выполняется следующая последовательность действий:

1 Программа ищет обработчик для данного типа исключения.

2 Если обработчик найден, стек очищается и управление передается обработчику исключения.

3 Выполняется код из секции *__finally*.

4 Если обработчик не был найден, вызывается функция *terminate* для завершения приложения.

Блок кода, который обрабатывает возникшее исключение, начинается ключевым словом *catch* и заключается в фигурные скобки. По меньшей мере один блок обработчика исключения должен следовать непосредственно за блоком *try***.** Для каждого исключения, которое может сгенерировать программа, должен быть предусмотрен свой обработчик. Обработчики исключений просматриваются по порядку, и выбирается обработчик исключения, тип которого соответствует типу аргумента в операторе *catch*.

Если тип исключения заранее неизвестен, то можно использовать блок *catch (...)*, который будет обрабатывать любое исключение. Это своего рода универсальный блок обработки исключений. Он должен завершать список обработчиков, поскольку ни один блок *catch* после него не сможет быть вы-

полнен для обработки данного исключения, так как все возможные исключения будут перехвачены этим блоком

Для демонстрации механизма исключений необходимо запускать *EXE*-файл приложения, так как при работе отладке приложения в интегрированной среде в первую очередь будут вызываться обработчики исключений интегрированной среды, а затем описанные в приложении.

1.2 Типы основных исключений

Все исключения являются объектами, порожденными от базового класса *Exception*. Свойство *Message* этого класса содержит информацию о типе ошибки.

Рассмотрим типы наиболее часто обрабатываемых исключений.

EArrayError – ошибочные действия с массивами (неверный индекс и т. д.)

EConvertError – ошибки преобразования строк.

EDatabaseError – ошибки при работе с базой данных.

EDBEngineError – ошибки при работе с *BDE*.

EIntError – исключения, возникающие при выполнении целочисленных математических операций. От данного класса порождены классы:

- *EDivByZero* – попытка целочисленного деления на 0.
- *EIntOverflow* – переполнение при выполнении операций с целыми числами.

EMathError – исключения, возникающие при выполнении вещественных математических операций. От данного класса порождены классы:

- *EInvalidArgument* – недопустимое значение параметра математической функции.
- *EOverflow* – переполнение регистра при выполнении операций с плавающей запятой.
- *EUnderflow* – потеря значащих разрядов.
- *EZeroDivide* – деление на 0 вещественного числа.

EPrinter – ошибки при печати.

EStreamError – ошибки потоков ввода-вывода. От данного класса порождены классы:

- *EFCreateError* – ошибка создания файла.
- *EFOpenError* – ошибка открытия файла.
- *EFilerError* – ошибки файловых потоков (операции чтения, записи).

1.3 Функции вывода сообщений

Чаще всего обработка исключений заключается в выводе пользователю сообщений об ошибке, устранении последствий ошибки (повторный ввод исходных данных) и повторении операций, вызвавших исключение.

Для вывода различных сообщений в *C++Builder* могут быть использованы функции *MessageBox* и *ShowMessage.*

Функция *MessageBox* – является методом класса *TApplication* и предназначена для вывода сообщения пользователю.

Формат функции:

int __fastcall MessageBox (char * Текст*, char* * Заголовок*, int* Flag*);*

Результатом вызова функции *MessageBox* является обобщенное диалоговое окно с сообщением, имеющее одну или большее количество кнопок. Параметр «Текст» определяет сообщение в диалоговом окне. Параметр «Заголовок» задает заголовок диалогового окна.

Значение параметра «Текст» может быть больше чем 255 символов в случае необходимости. Длинные сообщения автоматически будут разбиты на строки. Значение «Заголовок» появляется в названии диалогового окна. Заголовки могут быть больше чем 255 символов, но они не переносятся по словам. Длинный заголовок образует широкое окно сообщения.

Параметр Flag задает количество и тип кнопок диалогового окна. Возможные значения параметра *Flag* приведены в таблице 1.

Таблица 1

Flag	Значение
MB_OK	Окно сообщения содержит одну кнопку: *OK*. Это значение по умолчанию
MB_OKCANCEL	Окно сообщения содержит две кнопки: *OK* и *Отмена*
MB_ABORT-RETRYIGNORE	Окно сообщения содержит три кнопки: *Аварийное прекращение работы*, *Повторите*, и *Игнорировать*
MB_RETRY-CANCEL	Окно сообщения содержит две кнопки: *Повторите* и *Отмена*
MB_YESNO	Окно сообщения содержит две кнопки: *Да* и *Нет*
MB_YESNO-CANCEL	Окно сообщения содержит три кнопки: *Да*, *Нет*, и *Отмена*

Возвращаемые значения функции *MessageBox* представлены в таблице 2.

Таблица 2

Константа	Числовое значение	Описание
IDABORT	3	Пользователь выбрал кнопку *Abort*
IDCANCEL	2	“ “ “ *Cancel*
IDIGNORE	5	“ “ “ *Ignore*

IDNO	7	“	“	“	*No*
IDOK	1	“	“	“	*OK*
IDRETRY	4	“	“	“	*Retry*
IDYES	6	“	“	“	*Yes*

Если возвращаемое значение *MessageBox* равно 0, то отсутствует достаточно памяти, чтобы создать окно сообщения.

Например, если окно сообщения имеет кнопку *Cancel*, то функция возвращает значение *IDCANCEL* при нажатой клавише *ESC* или кнопке *Cancel*. Если окно сообщения не имеет кнопки *Cancel*, нажатие *ESC* не производит никакого эффекта.

Вызов функции *MessageBox* осуществляется следующим образом:

Application->MessageBox("Произошла ошибка - повторить?", “Error”, MB_OKCANCEL)

Частный случай *MessageBox* – это функция *ShowMessage*. В отличие от *MessageBox ShowMessage* выдаёт на экран окно сообщения с одной кнопкой *OK* (рисунок 2.1) и не возвращает значение. Вызов функции *ShowMessage* осуществляется следующим образом:

ShowMessage(“ShowMessage”);

Рисунок 2.1 – *ShowMessage*

1.4 Обработка ошибок математических функций

Пользовательская обработка ошибок математических функций в *C++Builder* может быть выполнена перегрузкой функций *_matherr* или *_matherrl*. Эти функции являются пользовательскими обработчиками прерываний и вызываются исключениями при вычислениях математических функций. Функция *_matherrl* вызывается при вычислениях с повышенной точностью (переменными типа *long double*).

Функции описываются следующим образом:

*int _matherr(struct _exception *e);*

*int _matherrl(struct _exceptionl *e);*

Если перегрузить данную функцию, то она замещает стандартный обработчик. При этом функция должна вернуть ноль, если данное исключение не обработано, и отличное от нуля значение – если исключение обработано успешно.

В качестве параметра в обработчик передается структура следующего вида:

```
struct _exception
{
  int   type;
```

```
  char  *name;
  double arg1, arg2, retval;
};
```

где *type* – тип математической ошибки;

**name* – указатель на строку, содержащую имя математической функции, вызвавшей исключение;

arg1, arg2 – аргументы функции, вызвавшие ошибку;

retval – заданное по умолчанию возвращаемое значение.

Обработчик *_matherr* поддерживает следующие виды ошибок (параметр *type*):

DOMAIN – параметр выходит за область определения функции;

SING – параметры приводят к специальным случаям (точкам разрыва). Например, вызов функции *pow*(0, -2), вызовет исключение данного типа, так как будет сделана попытка возвести ноль в отрицательную степень. Результатом в данном случае является бесконечность;

OVERFLOW – параметр приводит к переполнению, например, вызов exp(1000);

UNDERFLOW – параметр приводит к потере значащих разрядов, например, вызов exp(-1000);

TLOSS – параметр приводит к полной потере значащих цифр, например, sin(10e70).

В качестве примера рассмотрим перегрузку функции *_matherr*.

```
int _matherr (struct _exception *a)
{
  if (a->type == DOMAIN)
    throw Sysutils::EMathError("Error");
}
```

В данном примере при возникновении ошибки из-за вызова математической функции с недопустимыми параметрами (*DOMAIN*) с помощью оператора *throw* генерируется пользовательское исключение *EMathError*, которое в дальнейшем может быть обработано стандартным образом с помощью структуры *try – catch.*

2 Индивидуальное задание

Написать приложение-калькулятор, выполняющий сложение, вычитание, умножение, деление и математические функции согласно индивидуальному заданию. Выполнить обработку исключений, заданных в индивидуаль-

ном задании. Для выдачи сообщений об ошибках вычислений использовать функции *MessageBox* или *ShowMessage*. Оформить отчет по лабораторной работе.

Индивидуальное задание для разработки выдается преподавателем.

3 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, наименование и цель работы; индивидуальное задание; экранные формы разработанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

Лабораторная работа № 4

РАЗРАБОТКА ТЕКСТОВОГО РЕДАКТОРА

Цель работы. Изучить принципы построения текстовых редакторов.

1 Краткие сведения из теории

Для работы с текстовыми данными используются компоненты *Memo* и *RichEdit*. Они представляют такие возможности обработки текстовой информации, как ввод, удаление, вставку и т.д. Существует два базовых формата текстовых данных:

1. TXT – неформатированный текст
2. RTF – форматированный текст

Компонент *Memo* позволяет работать только с TXT, а *RichEdit* как с TXT, так и с RTF.

1.1 Компонентный класс *TRichEdit*

Для работы с текстовыми данными в *C++Builder* удобно использовать компонентный класс *TRichEdit*. Этот компонент отображает область редак-

тируемого ввода множественных строк информации в формате *RTF (Rich Text Format),* который предоставляет богатый выбор вариантов форматирования параграфов и шрифтовых атрибутов. Данный формат принимают многие профессиональные текстовые процессоры, в том числе *Microsoft Word*.

Основные свойства *TRichEdit* приведены в таблице 1.

Таблица 1

Команда	Описание
DefAttributes	Определить или установить заданные по умолчанию характеристики шрифта. Обратите внимание*: DefAttributes* доступен только во время выполнения
DefaultConverter	Используется для преобразования внутреннего формата файла в формат текстового модуля *TRichEdit*. Используется при работе с файлами, которые не имеют зарегистрированного расширения
HideSelection	Обеспечивает визуальную обратную связь выбранной части текста
Lines	Выполняет манипулирования типа: подсчет строк текста, добавление строк, удаление строк или замена текста в строках

Продолжение таблицы 1

SelAttributes	Определяет или устанавливает характеристики шрифта выбранного в настоящее время текста. *SelAttributes* – объект *TTextAttributes*, определяет характеристики типа, цвета, размера, стиля, и шага шрифта. Обратите внимание: *SelAttributes* доступен только во время выполнения	
SelLength	Определяет число выбранных символов	
Alignment	Форматирование текста. Возможные команды:	
	Команда	Значение
	taLeftJustify	Выравнивание текста по левой стороне
	taCenter	Выравнивание текста по центру
	taRightJustify	Выравнивание текста по правой стороне
WantTabs	Определяет, может ли пользователь вставлять символы табуляции в текст	
HideScrollBars	Определяет, исчезают ли полосы прокрутки когда необходимо. Если ScrollBars равно ssNone (значение по умолчанию для TRichEdit), то функция не работает	
Paragraph	Определяет параметры форматирования для текущего параграфа. Обратите внимание: эта опция доступна только во время выполнения	
SelText	Заменяет выбранный текст на новый. Если текст не выделен, то вставляется текст, начиная с позиции курсора	
ScrollBars	Полосы прокрутки. Возможные команды:	
	Команда	Значение
	ssNone	Без полос прокрутки
	ssHorizontal	Горизонтальная полоса прокрутки
	ssVertical	Вертикальная полоса прокрутки
	ssBoth	Горизонтальная и вертикальная полосы прокрутки
FindText	Ищет данный диапазон в тексте	
GetSelTextBuf	Копирует выбранный текст в буфер и возвращает скопированное число символов	
ClearSelection	Удаляет выбранный текст	
CopyToClipboard	Копирует выбранный текст в буфер обмена в формате *CF_TEXT*	
CutToClipboard	Копирует выбранный текст в буфер обмена в формате *CF_TEXT* и затем удаляет выделение	
PasteFromClipboard	Вставляет содержание буфера обмена, заменяя текущий ее выбор	

1.2 Разработка многодокументного редактора

Для разработки многодокументного редактора можно воспользоваться шаблоном многодокументного интерфейса (*MDI*). Для этого необходимо создать новый проект (*File|New*) и на вкладке *Projects* выбрать *MDI Application*.

Активируйте форму *MDIChild* и перетащите на нее компонент *TRichEdit*. Таким образом вы получили окно, в котором будет происходить работа с одним документом. Для организации работы с несколькими документами необходимо организовать список для хранения объектов *TMDIChild* с последующим доступом к ним.

1.2.1 Списки

Запоминание объектов в списках для последующих ссылок – типичный прием программирования. *C++Builder* предоставляет широкий спектр специализированных списочных компонентов (*TListBox, TComboBo*x и др.). По существу, все списки *VCL* (которые часто называют контейнерными классами) являются вариациями на тему одного из двух главных:

• строчный список *TStringList*;

• список общего назначения *TList*.

Основная задача объектов, порождаемых контейнерными классами, – распределение памяти. Контейнерные списочные классы *VCL* снимают с вас проблемы динамического распределения памяти и решают эти задачи самостоятельно. В результате размер контейнера автоматически меняется, реагируя на добавление и удаление своих элементов. Непродуктивный расход памяти исключается.

Другая особенность контейнерных классов состоит в их способности легко манипулировать элементами списка посредством объектных методов. Хотя списочные классы *TStringList* и *TList* не являются прямыми потомками одного предка, однако имеют много общего (таблица 1).

Важным дополнением к перечисленным методам является свойство *Count*, содержащее текущее число элементов списка.

Таблица 1

Метод	Назначение
Add	Добавить элемент к концу списка
Clear	Очистить список
Delete	Удалить элемент с указанным номером
Exchange	Поменять два элемента местами
Insert	Вставить элемент в указанную позицию
Move	Переместить элемент из одной позиции в другую
Sort	Отсортировать список

Класс *TStringList* – строчный список. Как следует из названия, класс *TStringList* специально предназначен для обслуживания строчных списков. Создание списка выполняется вызовом конструктора:

TStringList MyList = new TStringList;*

Как только объект создан, можно начинать добавлять к нему строки, например:

MyList–>Add("Гомель");
MyList–>Add("Минск") ;
MyList–>Add("Брест") ;

Элементы списка нумеруются, начиная с нуля, и адресуются посредством индексируемого свойства *Strings [i]*. Например, заголовку формы можно присвоить название, взятое из строки "*Брест*", следующим образом*:*

Forml–>Caption = MyList–>Strings[2];

Напротив, свойство Text содержит все строки списка, разделенные парами символов *<CR><LF>*.

Класс *TStringList* имеет еще одно полезное свойство – *Objects*, которое можно использовать для сопровождения строк дополнительной информацией. Хранение объектов – привилегия другого списочного класса *TList.* Помимо общих с ним методов, *TStringList* инкапсулирует ряд специальных функций. Некоторые из них представлены в таблице 2.

Класс *TList* – список общего назначения. Он предназначен для хранения данных любого типа, вплоть до указателей на другие объекты *VCL* или на экземпляры ваших собственных классов. Адресный указатель void* на объект любого типа передается как параметр таким общецелевым методам, как *Add* и *Insert*. Помимо них, *TList* инкапсулирует ряд специализированных функций. Некоторые из них представлены в таблице 3.

Таблица 2

Метод	Назначение
AddObject	Добавляет строку вместе с указателем на сопровождающий объект к концу списка
AddStrings	Добавляет группу строк, взятых из другого списочного объекта
Find	Выдает номер строки, под которым она добавляется в сортированный список. Метод возвращает значение false, если такой строки еще нет в списке
IndexOf	Возвращает номер строки, под которым она впервые встретилась в списке (как сортированном, так и нет)
LoadFromFile	Загружает строки списка из текстового файла
SaveToFile	Сохраняет список в текстовом файле

Таблица 3

Метод	Назначение
First	Возвращает первый указатель списка – элемент Индексируемого свойства *Items[0]*.
Last	Возвращает последний указатель списка – элемент Индексируемого свойства *ItemsfCount–1]*.
Pack	Вычеркивает незаполненные указатели (со значением *NULL*) из списка. Пустые элементы перемещаются вверх массива *Items*, а значение свойства *Count* уменьшается соответственно.

При выборе значений элементов списка необходимо преобразовать тип указателя к типу хранимого объекта. Класс *TList* не является владельцем объектов, которые в нем хранятся. Другими словами, вы несете ответственность за освобождение памяти, отведенной для хранения каждого объекта в контейнере. Можно было бы заключить, что это произойдет автоматически при уничтожении самого списка оператором *delete,* однако корректная процедура очистки списка потребует удаления каждого объекта списка пользователем.

1.2.2 Проектирование дочерней формы

В первую очередь необходимо создать список для хранения объектов *TMDIChild*. Список лучше создавать в конструкторе главной формы динамически:

```
ChildList = new TList;
```

Создание нового окна и загрузку в него документа можно оформить отдельной функцией, в которую в качестве параметра *Name* будет поступать имя файла.

```
Child = new TMDIChild (Application); // создать новое окно
Child->Caption = Name; // отобразить имя файла в заголовке окна
static int i;
ActiveMDIChild->Tag = i++; // запомнить номер нового окна
ChildList->Add(Child); // занести Child в список
Child->RichEdit->Lines->LoadFromFile(Name); // загрузить файл
```

Свойство главной формы *ActiveMDIChild->Tag* хранит порядковый номер активного дочернего окна, а свойство *MDIChildCount* – общее количество открытых окон.

Для ссылки на активное окно после того как вы поработали с другими окнами, необходимо установить указатель *Child* на новое активное окно:

```
Child = (TMDIChild *) ChildList->Items[ActiveMDIChild->Tag];
```

Теперь можно выполнять различные операции с активным окном (например, сохранение файла) через указатель *Child.*

1.3 Организация поиска и замены текста

Для организации поиска и замены текста можно воспользоваться компонентами *TFindDialog* и *TReplaceDialog.* Эти элементы, в отличие от других стандартных диалогов, являются немодальными, то есть приложение может продолжать работать в фоновом режиме, пока диалоговое окно остается открытым.

Свойство *FindText* обоих диалогов содержит искомое слово или фразу, которая будет отображаться в окне поиска текста. Если в это свойство записать строку (выделенный фрагмент текста) перед вызовом диалога, то эта строка появится в окне поиска:

ReplaceDialog1->FindText=Child->RichEdit1->SelText;

Вызов диалога выполняет метод *Execute*( ).

Прежде чем приступить к поиску, необходимо извлечь установленные пользователем опции:

- ✓ *Match whole word only* – искомый текст должен быть законченным словом;
- ✓ *Match case* – различать регистр клавиатуры.

Значение этих опций хранится в свойстве *Options*, имеющего тип «множество». Выборку элемента множества производит функция *Contains*:

```
if (FindDialog1->Options.Contains(frWholeWord))
      Option<<stWholeWord;
else Option>>stWholeWord;
if (FindDialog1->Options.Contains(frMatchCase))
      Option<<stMatchCase;
else Option>>stMatchCase;
```

Поиск текста реализует метод *FindText* с четырьмя параметрами: искомая строка, указатель начала поиска, длина текста и опции поиска:

```
Pos=Child->RichEdit1->FindText(FindDialog1->FindText,
Child->RichEdit1->SelStart, Child->RichEdit1->Text.Length(), Option);
```

Функция возвращает позицию курсора, с которой начинается найденная строка, или значение «-1», если указанный текст не найден. Чтобы выделить найденный текст необходимо выполнить следующие действия:

```
Child->RichEdit1->SelStart = Pos;
Child->RichEdit1->SelLength= FindDialog1->SelLength( );
```

Свойство *ReplaceText* диалога поиска и замены содержит текст замены, введенный пользователем в окно *Replace With*. Событие *OnReplace* возникает, когда пользователь нажал кнопку *Replace* или *Replace All*. В зависимости от результата предыдущего поиска обработчик этого события *ReplaceDialogReplace* будет производить нужное действие. Если пользователь нажал *Replace*, то следует предположить, что искомый текст *FindText* уже был найден, заместить его на *ReplaceText* и возобновить поиск. Если же пользователь нажал *Replace All*, то надо просмотреть весь документ и подменить все найденные образцы текста *FindText* значением *ReplaceText*. Эта операция реализуется рекурсивным вызовом функции *ReplaceDialogReplace*. Пример реализации функции *ReplaceDialogReplace* приведен ниже.

```
if (Child->RichEdit->SelLength == 0)
  FindDialogFind(Sender); // искать дальше
else
{ Child->RichEdit->Seltext = ReplaceDialog->ReplaceText;
  FindDialogFind(Sender); // заменить и искать дальше
}
if (ReplaceDialog->Options.Contains(frReplaceAll))
  while (Child->RichEdit->SelLength !=0)
    ReplaceDialogReplace(Sender); // рекурсивная замена
```

Метод FindText не поддерживает обратный поиск вверх по документу. Чтобы реализовать эту возможность необходимо включить соответствующую опцию в диалогах поиска и замены и программно перенести указатель начала поиска на начало документа. Длину текста принять равной длине всего документа.

2 Индивидуальное задание

Разработать текстовый редактор со следующими функциями: возможность сохранения, печати, чтения текстовых файлов, форматирование выделенного фрагмента текста (для *RichEdit*). Оформить отчет по лабораторной работе.

Индивидуальное задание для разработки выдается преподавателем.

3 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, наименование и цель работы; индивидуальное задание; экранные формы разработанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

Лабораторная работа № 5

ПЕРЕТАСКИВАНИЕ ОБЪЕКТОВ МЕТОДОМ DRAG&DROP

Ц е л ь р а б о т ы. Изучить методы реализации технологии перетаскивания объектов методом *Drag&Drop*.

1 Краткие сведения из теории

Все пользователи *Windows* знакомы с техническим приемом *Drag&Drop*. Считается, что любой профессиональный программный продукт должен обладать этой полезной способностью. Процесс копирования объектов нагляден и интуитивно понятен: вы выбираете объект или группу объектов из любого открытого источника и, удерживая кнопку мыши, перетаскиваете его (их) в приложение-приемник. Если приложение умеет обращаться с объектами данного типа, то оно обработает их соответствующим образом.

Библиотека *VCL* предоставляет ограниченные возможности перетаскивания объектов – только между компонентами одной формы или нескольких форм одного приложения. Для перемещения объектов между разными приложениями необходимо использовать специализированные функции *Windows API*. Рассмотрим обе эти возможности.

1.1 Перетаскивание объектов между компонентами одного приложения

Перетаскивание объектов между компонентами одного приложения реализуется с помощью программной обработки двух событий *OnDragDrop* и *OnDragOver*.

Событие *OnDragDrop* наступает в момент отпускания перетаскиваемого компонента над данным компонентом. В обработчике события надо описать, что в этот момент должно произойти. Обработчик имеет четыре параметра.

Параметр *Source* указывает на перетаскиваемый объект, а параметр *Sender* – на объект, над которым объект *Source* был отпущен. Параметры *X* и *Y* содержат координаты позиции курсора мыши над компонентом в системе координат клиентской области этого компонента.

Событие *OnDragOver* начинается в момент, когда перетаскиваемый объект пересек границу данного компонента и оказался внутри его контура. Заканчивается событие, когда объект, покидая компонент, пересек его границу. Обработчик события *OnDragOver* используется для того, чтобы дать сигнал о готовности компонента принять перетаскиваемый объект в случае, если пользователь отпустит его над данным компонентом. Если компонент готов принять объект, в обработчике надо задать значение параметра *Accept*, равное *true*. Впрочем, это значение по умолчанию равно *true*, так что его можно не задавать. Вообще в предельном случае обработчик может быть пустым, что будет означать готовность компонента принять любой объект. Но даже пустой обработчик нужен, так как иначе сообщения о приеме компонента приложение не получит.

Во время перетаскивания над компонентом объекта, который может быть принят, форма курсора мыши может изменяться, сигнализируя пользователю о готовности компонента принять объект. Чтобы это было так, надо до момента события *OnDragOver* (а обычно — во время проектирования) задать соответствующее значение свойства компонента *DragCursor*.

Параметр *Source* определяет перетаскиваемый объект, параметр *Sender* — сам компонент, параметры *X* и *Y* — координаты точки экрана в пикселах. Параметр *State* типа *TDragState* определяет состояние перетаскиваемого объекта по отношению к другим объектам. Возможны следующие состояния:

dsDragEnter – курсор мыши входит в пределы компонента;

dsDragMove – курсор мыши перемещается в пределах компонента;

dsDragLeave – курсор мыши выходит за пределы компонента.

В качестве примера рассмотрим реализацию перетаскивания строк между несколькими списками типа *TListBox,* размещенными на одной форме.

Во всех списках задаются значения свойств *DragMode*, равные *dmAutomatic*. Это обеспечивает автоматическое начало перетаскивания.

Далее для всех списков пишется единый обработчик события *OnDragOver* вида:

```
if (Sender  != Source)
     Accept = Source->ClassNameIs("TListBox");
else Accept = false;
```

В нем сначала проверяется, не являются ли данный компонент (*Sender*) и перетаскиваемый объект (*Source*) одним и тем же объектом. Это сделано, чтобы избежать перетаскивания информации внутри одного и того же списка. Если источник и приемник являются одним и тем же объектом, то параметр *Accept* становится равным *false***,** запрещая прием информации. Если же это разные объекты, то *Accept* делается равным *true***,** если источником является какой-то другой список (компонент класса *TListBox***),** и равным *false***,** если источник является объектом любого другого типа. Таким образом компонент сообщает, что готов принять информацию из любого другого списка.

Далее для всех списков пишется единый обработчик события *OnDrag-Drop* вида:

```
TListBox *S = (TListBox *)Source;
((TListBox*)Sender)->Items->Add(S->Items->Strings[S->ItemIndex]);
```

В этом обработчике первый оператор создает указатель *S* на объект класса *TListBox,* и передает в него ссылку на объект *Source***,** воспринимаемый как объект *TListBox.* Это сделано просто для того, чтобы не повторять несколько раз в следующем операторе приведение типа *Source* к указателю на объект класса *TListBox.* А такое приведение типа необходимо по следующей причине. Параметр *Source* объявлен как указатель на объект класса *TObject.* Но в этом классе отсутствуют свойства *Items* и *Itemlndex.* Они имеются только в классе *TListBox.* Поэтому прежде, чем обратиться к этим свойствам, надо произвести соответствующее приведение типа *Source.*

Второй оператор обработчика заносит методом *Add* выделенную строку списка-источника *S* в список-приемник *Sender.*

Подобные обработчики событий позволяют пользователю перетаскивать строки между любыми имеющимися списками.

Если приведенный выше пример по каким-то соображениям желательно осуществлять не в автоматическом режиме, а в режиме ручного управления (например, перетаскивание возможно только при нажатой клавише *Alt* или в каком-то конкретном режиме работы приложения), то отличия в реализации примера заключаются в следующем.

Во всех списках задаются значения свойств *DragMode,* равные *dmManual.* Это обеспечивает управление началом перетаскивания.

Затем в списках задается обработчик события *OnMouseDown* вида:

```
if ((Button  ==  mbLeft) && Shift.Contains(ssAlt))
   ListBoxl->BeginDrag(false,5);
```

В этом обработчике первое условие проверяет, нажата ли именно левая кнопка мыши, второе – нажатие клавиши *Alt* (можно задать какое-то другое условие, по которому нажатие кнопки мыши можно ассоциировать с началом перетаскивания). Затем методом *BeginDrag* начинается перетаскивание. Поскольку в параметре метода задано значение false, то перетаскивание в действительности начнется только после сдвига мыши на 5 пикселов.

Все остальные обработчики событий не отличаются от указанных в предыдущем примере.

1.2 Перетаскивание объектов между компонентами разных приложений

Для перемещения объектов между разными приложениями необходимо использовать специализированные функции *Windows API*. Для перемещения файлов между приложениями применяются функции *DragAcceptFiles* и *DragQueryFiles*. Прототипы этих функций содержатся в файле предкомпиляции *shellapi.h*, поэтому необходимо включить его в заголовок разрабатываемого интерфейсного модуля.

Для реализации технологии *Drag&Drop* необходимо выполнить следующие действия:

1 Информировать *Windows*, что приложение готово принимать файлы.

2 Поймать и обслужить сообщение *Windows WM_DROPFILES*.

3 Обработать перенесенные файлы в вашем приложении.

Чтобы информировать Windows о том, что приложение готово принимать файлы, надо просто обратиться к функции:

DragAcceptFiles(Handle, true);

Параметр *Handle* указывает на дескриптор окна, которое будет принимать перетаскиваемые файлы. Второй параметр говорит о готовности окна принять файл.

При вызове *DragAcceptFiles* произойдет следующее. Во-первых, при нажатии клавиши мыши на перетаскиваемом объекте и перемещении его над вашим приложением курсор мыши примет специфический вид, показывающий, что идет процесс перетаскивания файла в данное окно. Во-вторых, когда вы отпустите кнопку мыши и файл «упадет» в окно, поступит сообщение *WM_DROPFILES*. Вызывать функцию *DragAcceptFiles* лучше всего в обработчике *OnCreate* основной формы приложения.

Чтобы поймать сообщение *WM_DROPFILES*, надо в секции *public* объявления класса главной формы создать таблицу *MESSAGE_MAP*:

```
BEGIN_MESSAGE_MAP
MESSAGE_HANDLER(WM_DROPFILES, TWMDropFiles, WmDropFiles)
END_MESSAGE_MAP(TForm)
```

Посредством этой таблицы при возникновении сообщения *Windows WM_DROPFILES* приложение будет вызывать обработчик *WmDropFiles*. Второй параметр таблицы определяет дескриптор перемещаемого объекта и передается в обработчик сообщения. Объявление функции-обработчика *WmDropFiles* необходимо включить в секцию *private* объявления класса главной формы.

Все действия, необходимые для выполнения обработки перемещенных файлов, описываются в обработчике *WmDropFiles*. Обработка может выглядеть следующим образом.

```
char path[255]; // переменная для хранения полного имени файла
HDROP hDrop=(HDROP) Message.Drop;
// получение числа перенесенных файлов
int nFiles = DragQueryFiles(hDrop, -1,NULL, NULL);
// последовательная обработка перенесенных файлов
for (int i=0; i<nFiles; i++)
{
  // запись в переменную path имени файла
  DragQueryFiles(hDrop, i, path, sizeof(path));
  // создание нового MDI-окна и загрузка файла в RichEdit
  CreateMDIChild(path);
  Child->RichEdit->Lines->LoadFromFile(path);
}
DragFinish(hDrop); // завершение переноса
```

Функция *DragFinish* вызывается для того, чтобы *Windows* освободила память, занятую не нужной более информацией о перенесенных файлов. Таким образом, описание обработчика завершено. Приложение может принимать и обрабатывать файлы, перетаскиваемые из других приложений с помощью технологии *Drag&Drop*.

2 Индивидуальное задание

Разработать приложение, элементы которого поддерживают обмен информацией с помощью технологии *Drag&Drop*.

Индивидуальное задание для разработки выдается преподавателем.

3 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, на-

именование и цель работы; индивидуальное задание; экранные формы разработанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

Лабораторная работа № 6

ИЗУЧЕНИЕ ГРАФИЧЕСКИХ ОБЪЕКТОВ

Ц е л ь р а б о т ы. Изучить компоненты *C++ Builder* для отображения графической информации.

1 Краткие сведения из теории

В стандартную библиотеку визуальных компонент *C++ Builder* входит несколько объектов, с помощью которых можно придать своей программе оригинальный вид. Это – *TBevel, TShape*, *TImage* (*TDBImage*).

TBevel – объект для украшения программы, который может принимать вид рамки или линии. Объект предоставляет меньше возможностей по сравнению с *TPanel*, но не занимает ресурсов. Внешний вид указывается с помощью свойств *Shape* и *Style*.

TShape – отображает простейшие графические объекты на форме типа круг, квадрат и т.п. Вид объекта указывается в свойстве *Shape*. Свойство *Pen* определяет цвет и вид границы объекта. *Brush* задает цвет и вид заполнения объекта. Эти свойства можно менять как во время проектирования формы, так и во время выполнения программы.

Компонент *TImage* позволяет поместить графическое изображение в любое место на форме. Для использования компонента требуется выбрать его на странице *Additional* и поместить в нужное место формы. Рисунок можно загрузить во время проектирования с помощью Инспектора Объектов в редакторе свойства *Picture*. Рисунок должен храниться в файле в формате *BMP* (*bitmap*), *WMF* (*Windows Meta File*) или *ICO* (*icon*).

Форматов хранения изображений гораздо больше трех вышеназванных (например, наиболее известны *PCX, GIF, TIFF, JPEG*). Для включения в программу изображений в этих форматах нужно преобразовать их в формат *BMP*.

Следует помнить, что изображение, помещенное на форму во время проектирования, включается в файл *.DPR* и затем присоединяется к *.EXE* файлу.

Поэтому такой *.EXE* файл может получиться достаточно большой. Как альтернативу можно использовать загрузку картинки во время выполнения программы.

Свойство *Picture* позволяет легко организовать обмен с графическими файлами в процессе выполнения приложения. Это свойство является объектом, который имеет в свою очередь подсвойства, указывающие на хранящийся графический объект. Если в *Picture* хранится битовая матрица, на нее указывает свойство *Picture->Bitmap*. Если хранится пиктограмма, на нее указывает свойство *Picture->Icon.* На хранящийся метафайл указывает свойство *Picture->Metafile.* Наконец, на графический объект произвольного типа указывает свойство *Picture->Graphic.*

Объект *Picture* и его свойства *Bitmap, Icon, Metafile* и *Graphic* имеют методы файлового чтения и записи *LoadFromFile* и *SaveToFile.* Для свойств *Bitmap, Icon* и *Metafile* формат файла должен соответствовать классу объекта. При чтении файла в свойство *Graphic* файл должен иметь формат метафайла, при сохранении – любого разрешенного формата. При использовании методов *LoadFromFile* и *SaveToFile* объекта *Picture* методы чтения и записи автоматически подстраиваются под тип файла. Например, следующие операторы выполняют одинаковые действия:

```
Image1->Picture->LoadFromFile("BITMAP1.BMP");
Image1->Picture->Bitmap->LoadFromFile("BITMAP1.BMP");
```

Для всех рассмотренных объектов *Picture* определены методы присваивания значений объектов *Assign*. Однако для объектов *Bitmap, Icon* и *Metafile* присваивать можно только значения соответствующих объектов: битовых матриц, пиктограмм и метафайлов. Объект *Picture* – универсальный, ему можно присваивать значения всех остальных объектов.

Метод *Assign* можно использовать и для обмена изображениями с буфером *Clipboard*. Например, оператор

```
Clipboard ->Assign(Image1->Picture);
```

занесет в буфер обмена изображение, хранящееся в *Image1*. Аналогично оператор

```
Image1->Picture->Assign(Clipboard());
```

прочитает в *Image1* изображение, находящееся в буфере обмена. Причем это может быть любое изображение и даже текст.

Надо только не забыть при работе с буфером обмена вставить в модуль директиву:

#include <Clipbrd.hpp>

Важными являются свойства объекта *AutoSize*, *Center* и *Stretch,* имеющие булевский тип. Если установить свойство *AutoSize* в *true*, то размер компонента *Image* будет автоматически подгоняться под размер помещенной в него картинки. В противном случае изображение может не поместиться в компонент или, наоборот, компонент может оказаться много больше помещенного в него изображения. Если *Center* установлено в *True*, то центр изображения будет совмещаться с центром объекта *TImage*. Если *Stretch* установлено в *True*, то изображение будет сжиматься или растягиваться таким образом, чтобы заполнить весь объект *TImage*.

Еще одно свойство *Transparent* управляет прозрачностью объекта. Это свойство можно использовать для наложения изображений друг на друга. Свойство *Transparent* действует только на битовые матрицы. При установке этого свойства в *true* цвет левого нижнего пиксела битовой матрицы делается прозрачным, то есть выполняется его замена на цвет расположенного под ним изображения.

Для хранения графических объектов часто используется компонент *ImageList*, представляющий собой список для гранения набора изображений одинаковых размеров. При этом на изображения можно ссылаться по индексам, начинающимся с нуля.

Изображения в *ImageList* могут быть загружены в процессе проектирования с помощью редактора списков изображений. Окно редактора вызывается двойным щелчком на компоненте *ImageList* или щелчком правой кнопки мыши и выбором команды контекстного меню *ImageList Editor*.

Компонент *ImageList* имеет следующие свойства:

Count – количество изображений в списке (только для чтения);

Height и *Width* – высота и ширина изображений в списке. Изменение любого из этих свойств очищает список.

Остальные свойства и методы компонента используются гораздо реже и предназначены для программного добавления, вставки и замены элементов списка. На практике редактирование списка удобнее выполнять с помощью редактора списков изображений.

Наиболее часто компонент используется как свойство других компонентов, например, свойство *Images* компонента *MainMenu*.

2 Индивидуальное задание

Разработать приложение, использующее графические объекты *C++Builder* в соответствии с индивидуальным заданием.

Индивидуальное задание для разработки выдается преподавателем.

3 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, наименование и цель работы; индивидуальное задание; экранные формы разработанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

Лабораторная работа № 7

СОЗДАНИЕ ГРАФИЧЕСКОГО РЕДАКТОРА

Ц е л ь р а б о т ы. Создать простейший графический редактор.

1 Краткие сведения из теории

Основным элементом любого графического редактора является область рисования (холст). В *C++Builder* в качестве холста может быть использован специальный класс *TCanvas* (канва), который предоставляет широкие возможности для рисования. Этот класс является свойством таких объектов как *TBitmap, TComboBox, TDBComboBox, TDBGrid, TDBListBox, TForm, TImage, TListBox, TPaintBox, TStringGrid* и некоторых других.

1.1 Свойство объектов *Canvas*

Класс *TCanvas* является основой графической подсистемы *C++Builder*. Он обеспечивает:

- загрузку и хранение графических изображений;
- создание новых и изменение хранимых изображений с помощью пера (*Pen*), кисти (*Brush*) и шрифта (*Font*);
- рисование и закраску различных фигур, линий, текстов;

• комбинирование различных изображений.

Canvas является объектом, объединяющим в себе поле для рисования, перо (*Pen*), кисть (*Brush*) и шрифт (*Font*). *Canvas* обладает также рядом графических методов: *Draw*, *TextOut*, *Arc*, *Rectangle* и др. Используя *Canvas*, можно воспроизводить на форме любые графические объекты – картинки, многоугольники, текст и т.п. без использования компонентов *TImage*, *TShape* и *TLabel* (т.е. без использования дополнительных ресурсов), однако при этом необходимо обрабатывать событие *OnPaint* того объекта, на канве которого вы рисуете. Рассмотрим подробнее свойства и методы объекта *Canvas*.

Brush – кисть, является объектом со своим набором свойств:

Bitmap – картинка размером строго 8x8 пикселов, используется для заполнения (заливки) области на экране.

Color – цвет заливки.

Style – предопределенный стиль заливки; это свойство конкурирует со свойством *Bitmap* – какое свойство задано последним, то и будет определять вид заливки.

Handle – данное свойство дает возможность использовать кисть в прямых вызовах процедур *Windows API* .

ClipRect – (только чтение) определяет доступную область рисования на канве и область, подлежащую перерисовке при возникновении события *OnPaint*.

CopyMode – определяет, каким образом будет происходить копирование (с помощью метода *CopyRect*) на данную канву изображения из другого места: один к одному, с инверсией изображения и др.

Font – шрифт, которым выводится текст с помощью метода *TextOut*.

Pen – карандаш, определяет вид линий; как и кисть (*Brush*), является объектом с набором свойств:

Color – цвет линии.

Mode – режим вывода: простая линия, с инвертированием, с выполнением исключающего «ИЛИ» и др.

Style – стиль вывода: линия, пунктир и др.

Width – ширина линии в точках.

PenPos – текущая позиция пера; перо рекомендуется перемещать с помощью метода *MoveTo*, а не прямой установкой данного свойства.

Pixels – двухмерный массив элементов изображения (*pixel*), с его помощью можно получить доступ к каждой отдельной точке изображения.

Canvas имеет методы для рисования простейших графических примитивов – *Arc* (дуга окружности или эллипса), *Chord* (замкнутая фигура, ограниченная дугой окружности или эллипса и хордой), *Ellipse* (окружность или эллипс), *LineTo* (линия), *Pie* (сектор окружности или эллипса), *Polygon* (мно-

гоугольник), *PolyLine* (ломаная линия), *Rectangle* (прямоугольник), *RoundRect* (прямоугольник с закругленными углами). При прорисовке линий в этих методах используются значения свойств пера (*Pen*) канвы, а для заполнения внутренних областей – кисти (*Brush*).

Для закрашивания внутренних областей изображений используются методы *FillRect* (заполнение прямоугольной области) и *FloodFill* (заполнение замкнутой области канвы). При закрашивании используются текущие значения свойств кисти.

Для вывода рисунка на канву используются методы *Draw* и *StretchDraw*. Параметрами этих методов являются прямоугольник и графический объект для вывода (это может быть *TBitmap*, *TIcon* или *TMetafile*). *StretchDraw* отличается тем, что растягивает или сжимает картинку так, чтобы она заполнила весь указанный прямоугольник.

Если необходимо скопировать часть изображения с другой канвы, то можно использовать метод *CopyRect*.

Для вывода текста используются методы *TextOut* и *TextRect*. При выводе текста используется шрифт (*Font*) канвы. При использовании *TextRect* текст выводится только внутри указанного прямоугольника. Длину и высоту текста можно узнать с помощью функций *TextWidth* и *TextHeight*.

В качестве примера использования канвы приведем фрагмент программы для изображения на компоненте *TPaintBox* эллипса со звездой внутри:

```
PaintBox1->Canvas->Brush->Color = clBlack;
PaintBox1->Canvas->Brush->Style = bsDiagCross;
PaintBox1->Canvas->Ellipse (0, 0, PaintBox1->Width, PaintBox1->Height);
PaintBox1->Canvas->Pen->Color = clWhite;
PaintBox1->Canvas->MoveTo (40, 10);
PaintBox1->Canvas->LineTo (20, 60);
PaintBox1->Canvas->LineTo (70, 30);
PaintBox1->Canvas->LineTo (10, 30);
PaintBox1->Canvas->LineTo (60, 60);
PaintBox1->Canvas->LineTo (40, 10);
```

1.2 Событие *OnPaint*

При работе в многооконном режиме, выполняя рисование графических объектов, надо постоянно следить за рисунком, так как при наложении другого окна или перемещении его за пределы экрана, изображение затирается. С этой проблемой можно справиться с помощью события *OnPaint*. Оно наступает, когда приходит сообщение Windows о необходимости перерисовать

испорченное изображение. Обработчик данного события должен перерисовать изображение.

Если копия изображения, отображаемого на канве, хранится в компоненте *Bitmap*, то обработчик события *OnPaint* для формы может иметь вид:

Canvas->Draw(0,0,Bitmap);

Более быстрая перерисовка получается при использовании свойства *ClipRect* канвы, которое указывает область, внутри которой изображение испорчено:

Canvas->CopyRect(Canvas->ClipRect,Bitmap->Canvas,Canvas->ClipRect);

1.3 Компонент *TColorDialog*

Для изменения свойств пера и кисти, определяющих цвета изображаемых фигур удобно пользоваться стандартным диалогом *TColorDialog*.

TColorDialog – компонент, который отображает диалоговое окно *Windows* для выбора цветов. Диалог не отображается на экране, пока он не активизируется непосредственным обращением. Когда пользователь выбирает цвет и нажимает *OK*, диалог закрывается, и выбранный цвет сохраняется в переменной *Color*.

Основные свойства компонента *TColorDialog*:

TColorDialog::Color – возвращает выбранный цвет.

TColorDialog::CustomColors – определяет дополнительные цвета, доступные в диалоговом окне.

Каждый цвет представляется как строка формы *ColorX = HexValue*. Например, следующая строка устанавливает первый выбранный цвет.

ColorA = 808022;

Может быть установлено до 16 цветов *(ColorA - ColorP).*

TColorDialog::Options – определяет опции и заданный по умолчанию вид диалога. Возможные значения опций представлены в таблице 5.1. По умолчанию, все эти опции выключены.

Таблица 5.1

Значение	Описание
cdFullOpen	Отображает заказные цветовые комбинации при открытии
cdPreventFullOpen	Отключает кнопку *Define Custom Colors* в диалоге так, чтобы пользователь не мог определять новые цвета
cdShowHelp	Добавляет кнопку *Help* к диалогу
cdSolidColor	Открывает самый близкий цвет для выбранного цвета из стандартной палитры

cdAnyColor	Позволяет пользователю выбирать цвета (которые, вероятно, придется аппроксимировать), которые не входят в стандартные цвета

Пример использования диалога:

```
if (ColorDialog1->Execute())
   Form1->Color = ColorDialog1->Color;
```

1.4 Компоненты *TOpenPictureDialog* и *TSavePictureDialog*

Компоненты *TOpenPictureDialog* и *TSavePictureDialog* предназначены для загрузки и сохранения графических файлов.

TOpenPictureDialog отображает диалоговое окно *Windows* для выбора и открытия графических файлов. Этот компонент имеет точно такой же вид, как *TOpenDialog*, за исключением того, что окно включает прямоугольную область предварительного просмотра. Если выбранное изображение может читаться, то оно отображается в области предварительного просмотра. Поддерживаемые форматы файлов: точечный рисунок (*.BMP*), иконка (*.ICO*), метафайл *Windows* (*.WMF*) и расширенный метафайл *Windows* (*.EMF*). Если выбранное изображение не поддерживается, то оно не появляется в области предварительного просмотра. Свойства и события компонента *TOpenPictureDialog* аналогичны свойствам и событиям компонента *TOpenDialog*.

TSavePictureDialog отображает диалоговое окно *Windows* для сохранения графического файла. Свойства и события компонента *TSavePictureDialog* аналогичны свойствам и событиям компонента *TSaveDialog*.

1.5 Рисование графических примитивов манипулятором «мышь»

Рисование графических примитивов манипулятором «мышь» может быть реализовано через обработку соответствующих событий. Использовать события, связанные с манипулятором «мышь» в своих приложениях можно через раздел *Events* панели *Object Inspector*. Здесь можно увидеть такие события, как:

OnMouseDown – выполняется при нажатии любой клавиши «мыши».

OnMouseMove – выполняется при перемещении «мыши».

OnMouseUp – выполняется при отпускании любой клавиши «мыши».

Если требуется различная обработка событий в зависимости от того, какая кнопка «мыши» нажата или какая нажата вспомогательная клавиша, можно анализировать параметры *Button* и *Shift*.

Значение параметра *Button* определяет, какая кнопка мыши нажата: *mbLeft* – левая, *mbRight* – правая и *mbMiddle* – средняя.

Параметр *Shift* представляет собой множество, содержащее помимо обозначения нажатой кнопки (*ssLeft, ssRight, ssMiddle, ssDouble*) еще и обозначения нажатых одновременно с этим вспомогательных клавиш (*ssShift, ssAlt, ssCtrl*).

Параметры *X, Y* определяют текущие координаты «мыши» относительно верхнего левого края объекта.

В качестве примера можно привести программу, рисующую эллипс на форме. При нажатии кнопки «мыши» и ее перемещении эллипс двигается по форме. Когда кнопка «мыши» отпущена, эллипс появляется на форме.

```
int StartX,Flag,StartY,OldX,OldY,col,coll,kswitch,sty,key;
//-------------------Создание новой формы-----------------------------------
void __fastcall TForm1::FormCreate(TObject *Sender)
{
Flag=Image1->Canvas->Pixels[0][0];
col=clBlack;
kswitch=1;
Flag=2;
}
//--------Если «мышь» перемещается-------------------------------------------
void __fastcall TForm1::Image1MouseMove (TObject *Sender, TShiftState
Shift, int X, int Y)
{
  Image1->Canvas->Brush->Style=bsClear;
  Image1->Canvas->Pen->Mode=pmXor;
  if (Flag==1)
{
    Image1->Canvas->Pen->Color=col ^ clWhite;
    MyPrint(OldX,OldY);
    MyPrint(X,Y);
  }
  if (Flag==0) // Кнопка нажата
{
    Image1->Canvas->Pen->Color=col ^ clWhite;
    MyPrint(X,Y); }
  if (Flag<2) Flag=1; // Кнопка отпущена
  OldX=X;
  OldY=Y;
```

```
}
//----------------Кнопка «мыши» нажата----------------------------------------
void __fastcall TForm1::Image1MouseDown(TObject *Sender, TMouseBut-
ton Button, TShiftState Shift, int X, int Y)
{
  StartX=X;
  StartY=Y;
  Flag=0;
}
//----------------Кнопка «мыши» отпущена---------------------------------------
void __fastcall TForm1::Image1MouseUp(TObject *Sender, TMouseButton
Button, TShiftState Shift, int X, int Y)
{
  if (Flag==1) {
    MyPrint(OldX,OldY);
    Image1->Canvas->Brush->Style=sty;
    Image1->Canvas->Pen->Mode=pmCopy;
    Image1->Canvas->Pen->Color=coll;
    Image1->Canvas->Brush->Color=col;
    MyPrint(X,Y);
    Flag=2;
  }
}
//--------------------Рисует объект-----------------------------------------
void MyPrint(int OldX, int OldY)
{
Form1->Image1->Canvas->Ellipse(StartX, StartY, OldX, OldY);
;}
```

2 Индивидуальное задание

Разработать приложение графического редактора в соответствии с индивидуальным заданием.

Индивидуальное задание для разработки выдается преподавателем.

3 Содержание отчета

Отчет оформляется в электронном виде в виде документа *Word* и содержит: фамилию, имя, отчество и группу студента, выполнившего работу, наименование и цель работы; индивидуальное задание; экранные формы раз-

работанного приложения; документация на программное обеспечение, полученная с помощью программы *DoxyGen*; выводы по работе. К отчету прилагаются файлы проекта.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1 Архангельский А.Я. Программирование в C++Builder 6. – М.: ЗАО «Издательство БИНОМ», 2002.

2 Архангельский А.Я. Справочное пособие по C++Builder 6. Книга 1. Язык C++. М.: ЗАО «Издательство БИНОМ», 2002.

3 Архангельский А.Я. Справочное пособие по C++Builder 6. Книга 2. Классы и компоненты. М.: ЗАО «Издательство БИНОМ», 2002.

4 Холингворт Дж., Сворт Б., Кэшмэн М., Густавсон П. Borland C++Builder 6. Руководство разработчика. – М.: Издательский дом «Вильямс», 2003.

5 Шамис В. Borland C++ Builder 4. Техника визуального программирования. – М.: Нолидж, 2000.

С о д е р ж а н и е

Учебное издание

Сергей Николаевич Х а р л а п
Наталья Васильевна Р я з а н ц е в а

Разработка приложений в среде *C++ Builder*

Лабораторный практикум

Редактор О. В. З а н и н а
Технический редактор В. Н. К у ч е р о в а
Корректор М. П. Д е ж к о

Подписано в печать г. Формат бумаги 60x84 $^{1}/_{16}$. Бумага газетная.
Гарнитура *Times New Roman*. Печать офсетная.
Усл. печ. л. . Уч.-изд. л. . Тираж 100 экз.
Зак. № . Изд. № .

Редакционно-издательский отдел БелГУТа, 246653, г. Гомель, ул. Кирова, 34.
Лицензия ЛВ № 57 от 22.10.97 г.

Типография БелГУТа, 246022, г. Гомель, ул. Кирова, 34.
Лицензия ЛП № 360 от 26.07.99 г.