Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»

Т. А. Муратова

ПРОГРАММИРОВАНИЕ В СРЕДЕ C++ BUILDER

Утверждено в качестве учебного пособия
Ученым советом Федерального государственного бюджетного
образовательного учреждения высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»

Комсомольск-на-Амуре
2014

ББК 32.973.2-018.1я7
УДК 004.438(07)
М91

Рецензенты:
Кафедра «Информационные системы, компьютерные технологии и физика»
факультета информационных технологий, математики и физики
ФГБОУ ВПО «Амурский гуманитарно-педагогический
государственный университет»,
и. о. зав. кафедрой кандидат педагогических наук, доцент Н. Я. Салангина;
А. Н. Тачалов, кандидат технических наук, доцент,
ведущий инженер ОСДТУ ТЭЦ-3

Муратова, Т. А.
М91 Программирование в среде C++ Builder : практикум / Т. А. Муратова. – Комсомольск-на-Амуре : ФГБОУ ВПО «КнАГТУ», 2014. – 48 с.
ISBN 978-5-7765-1047-2

Практикум предназначен для студентов направления 230100 – «Информатика и вычислительная техника», изучающих дисциплину «Системы визуального программирования»; рекомендуется для студентов кафедрального проекта ФКТ, изучающих дисциплину «Программирование на языках высокого уровня».

В практикуме рассмотрены вопросы ввода/вывода данных различной сложности и отображения графической информации.

ББК 32.973.2-018.1я7
УДК 004.438(07)

ISBN 978-5-7765-1047-2

© ФГБОУ ВПО «Комсомольский-на-Амуре государственный технический университет», 2014

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ... 4
1. ПРОЕКТ C++ BUILDER... 4
2. КОМПОНЕНТЫ ВВОДА И ВЫВОДА... 5
3. ВВОД И ВЫВОД ДАННЫХ... 8
3.1. Ввод и вывод значения обычной переменной....................................... 8
3.2. Отладка приложения.. 20
3.3. Ввод и вывод значения переменной с помощью списков 21
3.4. Ввод и вывод целых чисел 26
3.5. Обработка массива значений 28
3.6. Перенос приложения на другой компьютер.. 31
3.7. Файловый ввод/вывод с помощью компонентов................................ 31
4. ОТОБРАЖЕНИЕ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ.......................... 39
4.1. Компоненты ProgressBar и CGauge 40
4.2. Компонент Chart.. 41
ЗАКЛЮЧЕНИЕ 48
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 48

ВВЕДЕНИЕ

«…современному обществу, которое не желает стагнации, жизненно необходимы в больших объемах высокообразованные специалисты, … а именно специалисты, у которых основной профессиональной сферой деятельности являются информационные (высокие) технологии» [1].

C++ Builder – система визуального объектно-ориентированного программирования, пользующаяся широкой популярностью, одинаково радующая и новичков, и асов программирования. Новичку она позволяет с небольшими затратами сил и времени создавать прикладные программы, которые внешне не отличимы от программ, созданных профессионалами, и удовлетворяют всем требованиям Windows. Для профессионала C++ Builder открывает неограниченные возможности для создания сколь угодно сложных распределенных приложений.

В данном практикуме подробно рассматриваются следующие темы: ввод/вывод с помощью компонентов Label, StaticText, Panel, Edit, MaskEdit, LabeledEdit, Memo, RichEdit, ListBox, ValueListEditor, ComboBox, StringGrid; файловый ввод/вывод; отображение графической информации с помощью компонентов ProgressBar, CGauge, Chart. Изложение сопровождается многочисленными практическими примерами, поясняющими излагаемый материал.

1. ПРОЕКТ C++ BUILDER

Проект – это набор файлов, которые компилятор использует для создания выполняемого файла программы (exe-файла). В простейшем случае проект составляют следующие файлы:

- файл описания проекта (bpr-файл);
- файл главного модуля (cpp-файл);
- файл ресурсов (res-файл);
- файл описания формы (dfm-файл);
- заголовочный файл формы (h-файл);
- файл описания функций формы (cpp-файл).

Чтобы сохранить проект, необходимо в меню **File** выбрать команду **Save Project As**. Если проект еще ни разу не был сохранен, то C++ Builder сначала предлагает сохранить модуль (содержимое окна редактора кода), и поэтому на экране появляется окно **Save Unit1 As**.

Обратите внимание, что имена файла модуля (cpp) и файла проекта (bpr) должны быть разными, т. к. C++ Builder в момент сохранения файла проекта создает одноименный cpp-файл (файл главного модуля). Кроме того, надо учесть, что имя генерируемого компилятором выполняемого файла совпадает с именем проекта. Поэтому файлу проекта следует при-

своить такое имя, которое, по вашему мнению, должен иметь выполняемый файл программы, а файлу модуля – какое-либо другое имя (например, полученное путем добавления к имени проекта порядкового номера модуля).

Для создания нового проекта необходимо выполнить команду **File | New | Application**. Работа над новым приложением начинается с создания стартовой формы – главного окна программы.

Рассмотрим основные свойства формы, которые определяют ее вид и поведение во время работы программы:

- **Name** – имя формы (используется для управления формой и доступа к ее компонентам);
- **Caption** – текст заголовка;
- **Width** – ширина формы;
- **Height** – высота формы;
- **Top** – расстояние от верхней границы формы до верхней границы экрана;
- **Left** –расстояние от левой границы формы до левой границы экрана;
- **BorderStyle** – вид границы (граница может быть обычной (bsSizeable), тонкой (bsSingle) или отсутствовать (bsNone), при этом размер окна с тонкой границей изменить нельзя);
- **Borderlcons** – кнопки управления окном. Значение свойства определяет, какие кнопки управления окном будут доступны пользователю во время работы программы. Свойство biSystemMenu определяет доступность кнопки **Свернуть** и кнопки системного меню, свойство biMinimaze – кнопки **Свернуть**, свойство biMaximaze – кнопки Развернуть, свойство biHelp – кнопки вывода справочной информации;
- **Icon** – значок в заголовке диалогового окна, обозначающий кнопку вывода системного меню;
- **Color** – цвет фона. Его можно задать, указав название цвета или привязку к текущей цветовой схеме операционной системы;
- **Font** – шрифт, используемый находящимися на поверхности формы компонентами по умолчанию.

2. КОМПОНЕНТЫ ВВОДА И ВЫВОДА

В библиотеке визуальных компонентов C++ Builder существует множество компонентов, позволяющих отображать, вводить и редактировать текстовую информацию. В табл. 2.1 приведен их перечень с краткими характеристиками.

Таблица 2.1

Компоненты ввода и отображения текстовой информации

Компонент	Вкладка	Описание
Label (метка)	**Standard**	Отображение текста, который не изменяется пользователем. Предусмотрено только изменение цвета самой метки и текста внутри нее. Основное свойство – **Caption**.
StaticText (метка с бордюром)	**Additional**	Подобен компоненту **Label**, но обеспечивает возможность задания стиля бордюра. Основное свойство – **Caption**.
Panel (панель)	**Standard**	Компонент является контейнером для группирования органов управления, но может использоваться и для отображения текста с возможностями объемного оформления. Основное свойство – **Caption**.
Edit (окно редактирования)	**Standard**	Отображение, ввод и редактирование однострочных текстов. Имеется возможность оформления объемного бордюра. Основное свойство – **Text**.
MaskEdit (окно маскированного редактирования)	**Additional**	Используется для форматирования данных или для ввода символов в соответствии с шаблоном. Основные свойства – **Text**, **EditMask**.
LabeledEdit (окно редактирования с привязанной к нему меткой)	**Additional**	Комбинация компонентов **Edit** и **Label**. Основные свойства – **Text** и **EditLabel.Caption**. *Только в C++ Builder 6.*
Memo (многострочное окно редактирования)	**Standard**	Отображение, ввод и редактирование многострочных текстов. Имеется возможность оформления объемного бордюра. Основное свойство – **Lines**.
RichEdit (многострочное окно редактирования в формате RTF)	**Win32**	Компонент представляет собой окно редактирования в стиле Windows в обогащенном формате RTF, позволяющее производить выбор атрибутов шрифта, поиск текста и многое другое. Основное свойство – **Lines**.

Продолжение табл. 2.1

Компонент	Вкладка	Описание
ListBox (окно списка)	**Standard**	Отображение стандартного окна списка Windows, позволяющего пользователю выбирать из него пункты. Основное свойство – **Items**.
CheckListBox (список с индикаторами)	**Additional**	Компонент является комбинацией списка **ListBox** и индикаторов **CheckBox**.
ValueListEditor (список специального вида)	**Additional**	Окно редактирования списка строк вида «имя = значение». Основные свойства – **Keys** (имена), **Values** (значения). *Только в C++ Builder 6.*
ComboBox (редактируемый список)	**Standard**	Объединяет функции **ListBox** и **Edit**. Пользователь может либо ввести текст, либо выбрать его из списка. Основное свойство – **Items**.
ComboBoxEx (список текстов и изображений)	**Win32**	Выпадающий список с возможностью отображения текстов и изображений. Основное свойство – **ItemsEx**. *Только в C++ Builder 6.*
StringGrid (таблица строк)	**Additional**	Отображение текстовой информации в таблице из строк и столбцов с возможностью перемещаться по строкам и столбцам и осуществлять выбор. Основное свойство – **Cells**.

Отображать текстовые надписи, помимо использования перечисленных компонентов, можно непосредственно на свойстве **Canvas** (холст) любого компонента, имеющего это свойство (в частности, непосредственно на форме).

Например, оператор вида

```
Canvas–>TextOutA(10,10,"Canvas");
```

обеспечивает печать текста «Canvas», начиная с точки с координатами (10, 10). Но его применение неудобно, так как при этом теряются преимущества визуального проектирования, и приходится рассчитывать координаты размещения надписи.

3. ВВОД И ВЫВОД ДАННЫХ

3.1. Ввод и вывод значения обычной переменной

Рассмотрим простой пример, позволяющий вычислять значение арифметического выражения. Для ввода данных используем компонент **Edit**, для вывода – компонент **Label**.

Рассмотрим основные свойства компонента **Edit**:

- **Name** – имя компонента, используемое в программе для доступа к компоненту и его свойствам;
- **Text** – текст, который находится в поле ввода/редактирования;
- **Left** – расстояние от левой границы компонента до левой границы формы;
- **Top** – расстояние от верхней границы компонента до верхней границы формы;
- **Height** – высота поля;
- **Width** – ширина поля;
- **Font** – шрифт, используемый для отображения вводимого текста;
- **ParentFont** – признак наследования шрифта родительского компонента.

Основные свойства компонента **Label**:

- **Name** – имя компонента, используемое в программе для доступа к его свойствам;
- **Caption** – отображаемый текст. Свойство Caption имеет тип строки AnsiString;
- **Font** – шрифт, используемый для отображения текста;
- **ParentFont** – признак наследования шрифта родительского компонента;
- **AutoSize** – признак того, что размер поля определяется его содержимым;
- **Left** – расстояние от левой границы поля вывода до левой границы формы;
- **Top** – расстояние от верхней границы поля вывода до верхней границы формы;
- **Height** – высота поля вывода;
- **Width** – ширина поля вывода;
- **Wordwrap** – признак того, что слова, которые не помещаются в текущей строке, автоматически переносятся на следующую строку (значение свойства **AutoSize** должно быть **false**, значение свойства **Wordwrap** – **true**).

Основные свойства компонента **Button**:

- **Name** – имя компонента, используемое в программе для доступа к компоненту и его свойствам;
- **Caption** – текст на кнопке;
- **Enabled** – признак доступности кнопки;
- **Left** – расстояние от левой границы кнопки до левой границы формы;
- **Top** – расстояние от верхней границы кнопки до верхней границы формы;
- **Height** – высота кнопки;
- **Width** –ширина кнопки.

Рассмотрим два важных понятия: событие и функция обработки события.

Событие (Event) – это то, что происходит во время работы программы. Реакцией на событие должно быть какое-либо действие. В C++ Builder реакция на событие реализуется как функция обработки события.

Рассмотрим некоторые события Windows:

- **OnClick** – событие при щелчке кнопкой мыши;
- **OnDblClick** – событие при двойном щелчке кнопкой мыши;
- **OnMouseDown** – событие при нажатии кнопки мыши;
- **OnMouseUp** – событие при отпускании кнопки мыши;
- **OnMouseMove** – событие при перемещении мыши;
- **OnKeyPress** – событие при нажатии клавиши клавиатуры;
- **OnKeyDown** – событие при нажатии клавиши клавиатуры. События **OnKeyDown** и **OnKeyPress** – это чередующиеся, повторяющиеся события, которые происходят до тех пор, пока не будет отпущена удерживаемая клавиша (в этот момент происходит событие **OnKeyUp**);
- **OnKeyUp** – событие при отпускании нажатой клавиши клавиатуры;
- **OnCreate** – событие при создании объекта (формы, элемента управления). Процедура обработки этого события обычно используется для инициализации переменных, выполнения подготовительных действий;
- **OnPaint** – событие при появлении окна на экране в начале работы программы; во время работы программы после появления окна (или его части), которое было закрыто другим окном или свернуто;
- **OnEnter** – событие при получении элементом управления фокуса;
- **OnExit** – событие при потере элементом управления фокуса.

Форма в режимах проектирования и выполнения разрабатываемого приложения показана на рис. 3.1.

Рис. 3.1. Форма разрабатываемого приложения
в режиме проектирования (а) и в режиме выполнения (б)

Данная форма содержит два поля редактирования для ввода данных, используемых при расчете; три надписи (две – для вывода информационных сообщений, одна – для вывода результата расчета) и две кнопки.

Листинг программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u, r, i; // диапазон значений float от 3.4·10–38 до 3.4·1038
// или: int u, r, i; диапазон значений int от –2 147 483 648
// до 2 147 483 647
// функции StrToFloat и StrToInt  переводят строку в число
u=StrToFloat(Edit1–>Text);
r=StrToFloat(Edit2–>Text);
i=u/r;
// функции FloatToStrF и IntToStr переводят число в строку
Label3–>Caption="Сила тока "+FloatToStrF(i,ffGeneral,7,2)+ " A";
// или: Label3–>Caption="Сила тока "+ IntToStr(i)+ " A";
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Form1–>Close(); }
```

Данная программа состоит из двух функций обработки события **Onclick** для кнопок «Рассчитать» и «Завершить». Чтобы программа работала корректно, пользователь должен ввести в каждое поле редактирования целое или дробное число в правильном формате (через запятую). Функция **StrToFloat** выполняет преобразование строки в число. Для преобразования числа в строку символов (свойство **Caption** – строкового типа) используется функция **FloatToStrF** (можно использовать функцию **FloatToStr**).

Выполните следующее:

- Внесите изменения в программу: для ввода данных используйте компонент **LabeledEdit**, для вывода – **StaticText**.
- Еще раз внесите изменения в программу: для ввода данных используйте компонент **MaskEdit**[*], для вывода – **Panel**.

Для выполнения приведенного выше задания воспользуйтесь следующими рекомендациями:

- Редактор кода поддерживает функцию контекстно-зависимой подсказки. Например, после того как будет набрано имя компонента и символы «–>», редактор кода автоматически выведет список свойств и методов объекта. Если список свойств и методов не появляется, то это значит, что в программе обнаружена ошибка.
- Окно редактора кода разделено на две части. В правой части находится текст программы. Левая часть, которая называется навигатор классов (ClassExplorer), облегчает навигацию по тексту (коду) программы.
- Для того чтобы в процессе набора текста программы воспользоваться шаблоном кода и вставить его в текст программы, нужно нажать комбинацию клавиш **Ctrl + J**. Программист может создать свой собственный шаблон кода и использовать его точно так же, как и стандартный. Чтобы создать шаблон кода, нужно в меню **Tools** выбрать команду **Editor Options** и в окне **Code Insight** щелкнуть по кнопке **Add**. В появившемся окне **Add Code Template** надо задать имя шаблона (**Shortcut Name**) и его краткое описание (**Description**). Затем, после щелчка на кнопке **OK**, в поле **Code** надо ввести шаблон.
- В процессе набора текста программы можно получить справку, нажав клавишу F1. Следует обратить внимание на то, что после имени функции может быть указано имя библиотеки, к которой эта функция относится (VCL или CLX). Библиотека VCL используется при разработке приложений для Windows, а библиотека CLX – при разработке кроссплатформенных приложений. Поэтому, выбирая раздел справочной системы, надо обращать внимание на то, к какой библиотеке он относится.

Во время работы приложения могут возникать ошибки, которые называются ошибками времени выполнения (run time errors) или исключениями (exceptions). В большинстве случаев причинами исключений являются неверные исходные данные. Например, в нашей программе исключение возникнет, если пользователь введет дробное число через точку (то, какой из двух символов (точка или запятая) является правильным, зависит от настройки Windows).

---

[*] Дополнительная информация по использованию компонента: Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : Бином-Пресс, 2003.– С. 153 (раздел 3.2.3).

Обработку исключений берет на себя автоматически добавляемый в выполняемую программу код, который обеспечивает в том числе и вывод информационного сообщения. Вместе с тем C++ Builder дает возможность программе самой выполнить обработку исключения.

Инструкция обработки исключения выглядит следующим образом:

```
try
{ // здесь инструкции, выполнение которых может вызвать исключение }
catch (Тип &e)
{ // здесь инструкции обработки исключения }
```

В этом примере **try** – ключевое слово, обозначающее, что далее следуют инструкции, при выполнении которых возможно возникновение исключений, и что обработку этих исключений берет на себя программа; **catch** – ключевое слово, обозначающее начало секции обработки исключения (инструкции этой секции будут выполнены, если в программе возникнет исключение указанного типа).

Рассмотрим типичные исключения:

- **EConvertError** (ошибка преобразования) – возникает при выполнении преобразования, если преобразуемая величина не может быть приведена к требуемому типу. Наиболее часто возникает при преобразовании строки символов в число.
- **EDivByZero** (целочисленное деление на ноль) – возникает при выполнении операции целочисленного деления, если делитель равен нулю.
- **EZeroDivide** (деление на ноль) – возникает при выполнении операции деления с дробными операндами, если делитель равен нулю.
- **EinOutError** (ошибка ввода/вывода) – возникает при выполнении файловых операций. Наиболее частой причиной является отсутствие требуемого файла или, в случае использования сменного диска, отсутствие диска в накопителе.

Модифицированный участок программы:

```
// возможно исключение – ошибка преобразования строки в число
try
{ u = StrToFloat(Edit1–>Text) ; r = StrToFloat(Edit2–>Text) ; }
catch (EConvertError &e)
{ ShowMessage ("При вводе дробных чисел используйте запятую"); return; }
// возможно исключение – деление на ноль
try
{ i = u/r; }
catch (EZeroDivide &e)
{
```

```
ShowMessage ("Величина сопротивления не должна быть равна нулю") ;
Edit2–>SetFocus () ; // курсор в поле Сопротивление
return;
}
// вывести результат в поле метки
Label3–>Caption = "Сила тока " + FloatToStrF(i,ffGeneral,7,2)+ " A" ;
```

Выход из функции может осуществляться несколькими способами.

Если функция не должна возвращать никакого значения, то выход из нее происходит или по достижении закрывающей ее тело фигурной скобки, или при выполнении оператора **return**.

Пример возврата из функции по достижении закрывающей ее тело фигурной скобки:

```
void SPrint(AnsiString S)
{ if (S != "") ShowMessage(S); }
```

*Пример возврата из функции при выполнении оператора **return**:*

```
void SPrint(AnsiString S)
{
if (S == "") return;
ShowMessage (S);
}
```

Если же функция должна возвращать некоторое значение, то выход из нее осуществляется оператором

return выражение;

где выражение должно формировать возвращаемое значение и соответствовать типу, объявленному в заголовке функции.

Пример:

```
double FSum(double XI,double X2, int A)
// диапазон значений double от 1.7·10^–308 до 1.7·10^308
{ return A * (XI + X2); }
```

Прервать выполнение функции можно также генерацией какого-либо исключения[*]. Наиболее часто в этих целях используется процедура **Abort**.

Возвращаемое функцией значение может включать в себя вызов каких-либо функций. Функция может вызывать и саму себя, т. е. допускается рекурсия.

[*] Дополнительная информация: Архангельский, А. Я. C++ Builder 6 : справ. пособие. Книга 1: Язык C++ / А. Я. Архангельский. – М. : Бином-Пресс, 2004. – С. 115 (раздел 1.12.6).

В качестве примера приведем функцию, рекурсивно вычисляющую факториал. Факториал можно вычислить с помощью простого цикла **for** (более простой вариант). Но можно факториал вычислять и с помощью рекуррентного соотношения n! = n * (n–1)!. Для иллюстрации рекурсии воспользуемся именно этим соотношением.

Функция ***factorial*** *вычисления факториала может быть описана следующим образом:*

```
unsigned long factorial (unsigned long n)
// диапазон значений unsigned long от 0 до 4 294 967 295
{
if (n <= 1)
return 1;
else
return n * factorial(n – 1);
}
```

Если значение параметра n равно 0 или 1, то функция возвращает значение 1. В противном случае функция умножает текущее значение n на результат, возвращаемый вызовом той же функции **factorial**, но со значением параметра n, уменьшенным на единицу. Поскольку при каждом вызове значение параметра уменьшается, то рано или поздно оно станет равно 1. После этого цепочка рекурсивных вызовов начнет свертываться и, в конце концов, вернет значение факториала.

В программе для вывода сообщений использована функция **ShowMessage**, которая выводит на экран окно с текстом и командной кнопкой **OK**. Инструкция вызова функции **ShowMessage** выглядит следующим образом:

ShowMessage (*Сообщение*);

где *Сообщение* – строковая константа (текст, который надо вывести).

Для вывода сообщений можно использовать функцию **MessageDlg**. Общий вид обращения к функции **MessageDlg** выглядит следующим образом:

MessageDlg (*Сообщение*, *Тип*, *Кнопки*, *КонтекстСправки*);

где *Сообщение* – текст сообщения; *Тип* – тип сообщения, который задается именованной константой; *Кнопки* – кнопки, отображаемые в окне сообщения; *КонтекстСправки* – параметр, определяющий раздел справочной информации, который появится на экране, если пользователь нажмет клавишу F1 (если вывод справочной информации не предусмотрен, то значение параметра должно быть равно нулю).

Сообщение, которое выводится с помощью функции **MessageDlg,** может быть информационным, предупреждающим или сообщением о критической ошибке. Каждому из этих типов сообщений соответствует определенный значок (табл. 3.1).

Таблица 3.1

Константы, определяющие тип сообщения

Константа	Тип сообщения	Значок
mtWarning	Внимание	Желтый восклицательный знак
mtError	Ошибка	Красный стоп-сигнал
mtInformation	Информация	Голубой символ «i»
mtConfirmation	Подтверждение	Зеленый вопросительный знак
MtCustom	Обычное	Без значка

Кнопки, отображаемые в окне сообщения, задаются операцией включения во множество элементов – констант (табл. 3.2).

Таблица 3.2

Константы, определяющие кнопки в окне сообщения

Константа	Кнопка	Константа	Кнопка
mbYes	Yes	mbAbort	Abort
mbNo	No	mbRetry	Retry
mbOK	OK	mbIgnore	Ignore
mbCancel	Cancel	mbAll	All
mbHelp	Help		

Кроме приведенных констант можно использовать константы mbOkCancel, mbYesNoCancel и mbAbortRetryIgnore. Они определяют наиболее часто используемые в диалоговых окнах комбинации командных кнопок.

Значение, возвращаемое функцией **MessageDlg** (mrAbort, mrYes, mrOk, mrRetry, mrNo, mrCancel, mrIgnore, mrAll), позволяет определить, какая из командных кнопок была нажата пользователем.

Пример:

```
MessageDlg ("Введены некорректные данные", mtWarning,
        TMsgDlgButtons() << mbOK << mbCancel, 0) ;
```

Рассматривая этот простой пример, мы внесли изменения в нашу программу, которые предполагают возможность ввода пользователем нечисловых данных и отсутствие ввода данных, но при этом в программе не предусмотрена защита от ввода отрицательных числовых значений.

Модифицированный участок программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u, i, r;
try
```

```
{ u=StrToFloat(Edit1–>Text); r=StrToFloat(Edit2–>Text); }
catch (EConvertError &e)
{ ShowMessage("Введены нечисловые данные"); return; }
if (u<0||r<0)
{ MessageDlg("Введено отрицательное число",
mtError,TMsgDlgButtons()<<mbOK,0); return; }
try
{ i= u/r; }
catch (EZeroDivide &e)
{ ShowMessage("Величина сопротивления не должна быть равна
нулю");
return; }
Label3–>Caption="Сила тока "+FloatToStr(i)+" А";
}
```

Обработку исключений в программе можно записать в том числе следующим образом:

```
try
{ u=StrToFloat(Edit1–>Text);
r=StrToFloat(Edit2–>Text);
i= u/r; }
catch (EConvertError &e)
{ShowMessage("Введены нечисловые данные"); return;}
catch (EZeroDivide &e)
{ShowMessage("Величина сопротивления не должна быть равна
нулю"); return;}
catch (...)
{ShowMessage("Обработка любых исключений"); return;}
```

Рассматриваемую задачу можно решить и другими способами. Далее приведены еще два варианта решения задачи.

Второй вариант программы:

```
// щелчок на кнопке Вычислить
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u; // напряжение
float r; // сопротивление
float i; // сила тока
// проверим, введены ли данные в поля Напряжение и Сопротивление
if ( ( (Edit1–>Text) .Length () == 0 ) || ( (Edit2–>Text) .Length () == 0) )
{
MessageDlg ("Надо ввести напряжение и сопротивление",
mtInformation, TMsgDlgButtons () << mbOK, 0) ;
```

```
if ( (Edit1–>Text) .Length () == 0)
Edit1–>SetFocus ( ) ; // курсор в поле Напряжение
else
Edit2–>SetFocus ( ) ; // курсор в поле Сопротивление
return;
}
// получить данные из полей ввода
u = StrToFloat (Edit1–>Text) ; r = StrToFloat (Edit2–>Text) ;
// вычислить силу тока
try
{ i = u/r; }
catch (EZeroDivide &e)
{
ShowMessage ("Величина сопротивления не должна быть равна
нулю") ;
Edit2–>SetFocus ( ) ; // курсор в поле Сопротивление
return;
}
// вывести результат в поле Label
Label3–>Caption = "Сила тока " + FloatToStrF(i,ffGeneral,7,2) + " A";
}
// нажатие клавиши в поле Напряжение
void __fastcall TForm1::Edit1KeyPress(TObject *Sender, char &Key)
{
// коды запрещенных клавиш заменим нулем, в результате
// символы этих клавиш в поле редактирования не появятся
// Key – код нажатой клавиши
// проверим, является ли символ допустимым
if ( ( Key >= '0') && ( Key <= '9')) // цифра
return;
// глобальная переменная Decimalseparator
// содержит символ, используемый в качестве разделителя
// при записи дробных чисел
if ( Key == DecimalSeparator)
{
if ( (Edit1–>Text).Pos(DecimalSeparator) != 0)
Key = 0; // разделитель уже введен
return;
}
if ( Key == VK_BACK) // клавиша Backspace
return;
if ( Key == VK_RETURN) // клавиша Enter
```

```
{
Edit2–>SetFocus ( ) ; return;
}
// остальные клавиши запрещены
Key = 0; // не отображать символ
}
// нажатие клавиши в поле Сопротивление
void __fastcall TForm1::Edit2KeyPress(TObject *Sender, char &Key)
{
if ( ( Key >= '0') && ( Key <= '9')) // цифра
return;
if ( Key == DecimalSeparator)
{
if ( (Edit2–>Text) .Pos (DecimalSeparator) != 0)
Key =0; // разделитель уже введен
return;
}
if (Key == 8) // клавиша Backspace
return;
if ( Key == 13) // клавиша Enter
{
Button1–>SetFocus ( ) ; // переход к кнопке Вычислить
// повторное нажатие клавиши Enter
// активизирует процесс вычисления тока
return;
}
// остальные клавиши запрещены
Key =0; // не отображать символ
}
// щелчок на кнопке Завершить
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Form1–>Close(); }
```

Третий вариант программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u; // напряжение
float r; // сопротивление
float t; // ток
char string1[11]  = "1234567890,";      // вспомогательная переменная
char string2[20];       // временная переменная
char string3[20];       // временная переменная
```

```
strcpy (string2,Form1–>Edit1–>Text.c_str ( ) ) ;
strcpy (string3,Form1–>Edit2–>Text.c_str ( ) ) ;
int i=0,k,k1;
// проверка на корректность ввода
do
{
k=0;
k1=0; // флаги для проверки совпадения строк
for (int j=0;j<=11;j++)
// если есть совпадение – k =1
{
if(string2[i]==string1[j]) k=1;
if(string3[i]==string1[j]) k1=1;
}
if ((k==0)||(k1==0)) break; // совпадений не найдено, выходим из цикла
i++;
}while((string2[i]!='\0')&&(string3[i]!='\0')); // цикл до конца массивов string2[] и string3[]
if (k==0) {Application–>MessageBox("Введено нечисловое значение или отрицательное число", "ошибка",MB_OK);}
else
{
// если все верно, считываем значения
u = StrToFloat (Edit1–>Text) ; r = StrToFloat (Edit2–>Text) ;
// вычислить силу тока
try
{ t = u/r; }
catch (EZeroDivide &e)
{
ShowMessage ("Величина сопротивления не должна быть равна нулю") ;
Edit2–>SetFocus ( ) ; // курсор в поле Сопротивление
return;
}
// вывести результат в поле Label
Label1–>Caption = "Ток : " + FloatToStrF(t,ffGeneral,7,2) + " A";
}
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Form1–>Close(); }
```

Внимательно просмотрите тексты приведенных выше программ и реализуйте их.

3.2. Отладка приложения

Компиляция с последующим выполнением приложения осуществляется командой **Run | Run** (F9). В этом случае производится компиляция программы, ее компоновка, создается и запускается на выполнение выполняемый модуль .exe.

Если нужно просто проверить, не содержат ли последние изменения кода каких-либо ошибок, то в этом случае лучше воспользоваться другими командами меню: **Project | Compile Unit** (компиляция только того модуля, который выделен в окне Редактора Кода), **Project | Make Project** (компиляция всех тех модулей, тексты которых были изменены с момента предыдущей компоновки проекта) или **Project | Build Project** (компиляция всех модулей, независимо от того, когда они в последний раз изменялись).

Выполните следующее: включите в окне опций проекта, вызываемом командой **Project | Options** на странице **Compiler**, в группе опций **Warnings** опцию **All**, чтобы компилятор отображал все свои замечания.

Если при выполнении программы произошла ошибка, то возможны несколько вариантов действий:

- прервать выполнение и отладку приложения: **Run | Program Reset** (Ctrl + F2);
- попытаться продолжить вычисления, несмотря на ошибку: **Run | Run** (F9);
- получить значение переменной, подведя курсор мыши к ее имени в коде;
- получить значения нескольких переменных, последовательно подводя курсор в коде к интересующим переменным и нажимая Ctrl + F5 (при этом откроется окно наблюдения **Watch List)**;
- использовать окно оценки и модификации **Evaluate/Modify** (Ctrl + F7), с помощью которого в процессе отладки можно не только наблюдать, но и изменять значения переменных;
- пройти часть программы по шагам (F8 – пошаговое выполнение строк программы, считая вызов функции за одну строку (т.е. вход в функции не производится); F7 – пошаговое выполнение программы с заходом в вызываемые функции; Shift + F7 – переход к следующей исполняемой строке; F4 – выполнение программы до того выполняемого оператора, на котором расположен курсор в окне редактора кода; Shift + F8 – выполнение программы до выхода из текущей функции, останов на операторе, следующем за вызовом этой функции; **Run | Show Execution Point** – помещение курсора на операторе, который будет выполняться следующим);
- другие действия*.

---

* Дополнительная информация: Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : Бином-Пресс, 2003. – С. 120 (раздел 2.8).

3.3. Ввод и вывод значения переменной с помощью списков

Рассмотрим основные свойства компонента **ListBox** (вкладка Standard):

- **Items** – служит для формирования списка строк типа **TStrings**;
- **MultiSelect** – разрешает пользователю множественный выбор в списке альтернатив;
- **ItemIndex** – определяет индекс выбранной строки в случае, если **MultiSelect** = false (свойство доступно только во время выполнения). Если ни одна строка не выбрана, то ItemIndex = -1;
- **Columns** – определяет число столбцов, в которых будет отображаться список, если он не помещается в окне компонента целиком;
- **Sorted** – позволяет упорядочить список по алфавиту;
- **AutoComplete** – позволяет пользователю быстро находить строку списка, нажимая клавишу, соответствующую ее первому символу.

Рассмотрим пример, позволяющий вычислять значения силы тока по значению напряжения, выбираемому из списка значений, и значениям сопротивления, задаваемым диапазоном значений. Для ввода данных используем компоненты **Edit** и **ListBox**, для вывода – **ListBox**.

Форма в режиме проектирования и выполнения разрабатываемого приложения показана на рис. 3.2.

Рис. 3.2. Форма разрабатываемого приложения
в режиме проектирования (а) и в режиме выполнения (б)

Листинг программы:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Form1–>Close(); }
void __fastcall TForm1::Button1Click(TObject *Sender)
{
ListBox2–>Clear(); ListBox3–>Clear();
```

```
// списки нужно очищать, иначе при повторном нажатии на кнопку
// Button1 в них выводятся некорректные данные
float u, r, rn, rk, i;
if (ListBox1–>ItemIndex<0)
{ ShowMessage("Не выбрано значение напряжения"); return; }
else
{
ShowMessage("Вы выбрали "+IntToStr(ListBox1–>ItemIndex+1)+" значение");
// это информационное сообщение
u=StrToFloat(ListBox1–>Items–>Strings[ListBox1–>ItemIndex]);
}
try
{ rn=StrToFloat(Edit1–>Text); rk=StrToFloat(Edit2–>Text); }
catch(EConvertError &e)
{ ShowMessage("Введены нечисловые данные"); return; }
if (rn<=0||rk<=0||rn>rk)
{ ShowMessage("Неправильно введен диапазон значений сопротивления");
return; }
r=rn;
while (r<=rk) // for (r=rn; r<rk; r+=h)
{ i=u/r;
ListBox2–>Items–>Add(FloatToStrF(i,ffGeneral,5,2));
ListBox3–>Items–>Add(FloatToStrF(r,ffGeneral,5,2));
r=r+(rk–rn)/10; }
}
```

Список **CheckListBox** (вкладка **Additional**) похож на **ListBox** и выглядит так же, но около каждой строки имеется индикатор, который пользователь может переключать (индикаторы можно переключать и программно). Свойства компонента **CheckListBox**, не связанные с индикаторами, аналогичны **ListBox**, за исключением свойств, определяющих множественный выбор (при использовании **CheckListBox** множественный выбор можно осуществлять установкой индикаторов).

Рассмотрим компоненты выпадающих списков **ComboBox** и **ComboBoxEx**.

Основные свойства компонента **ComboBox** (вкладка **Standard**):

- **Items** – служит для формирования списка строк типа **TStrings**;
- **Style** – определяет стиль изображения компонента;
- **Text** – определяет значение, выбранное пользователем из списка, или введенное им;

- **ItemIndex** – определяет индекс выбранного пользователем элемента списка;
- **MaxLength** – определяет максимальное число символов, которые пользователь может ввести в окно редактирования. Если MaxLength = 0, то число вводимых символов не ограничено;
- **DropDownCount** – указывает число строк, появляющихся в выпадающем списке без возникновения полосы прокрутки;
- **Sorted** – позволяет упорядочить список по алфавиту.

Компонент **ComboBoxEx** (вкладка Win32) во многом подобен **ComboBox**. Различие, прежде всего, заключается в том, что в **ComboBoxEx** легче, чем в **ComboBox**, вводить изображения в элементы списка.

Компонент **ValueListEditor** (вкладка **Additional**) представляет собой окно редактирования списка строк вида «имя = значение», состоит из двух колонок с заголовками: колонка «Key» для имен и колонка «Value» для значений.

Основные свойства компонента **ValueListEditor**:

- **TitleCaptions** (тип **TStrings**) – служит для изменения заголовков колонок «Key» и «Value»;
- **Strings** (тип **TStrings**) – служит для заполнения списка строк. Список может быть заполнен во время проектирования, во время выполнения пользователем или программно;
- **KeyOptions** – позволяет определить операции, доступные пользователю при редактировании колонки имен;
- **Cells[int ACol][int ARow]** (тип **AnsiString**) – служит для доступа к элементам строк (**ACol** = 0 соответствует колонке имен, **ACol** = 1 – колонке значений; **ARow** = 0 соответствует строке заголовков, **ARow** > 0 – строкам списка);

Внесем изменения в программу: для ввода данных используем компоненты **ComboBox** и **LabeledEdit**, для вывода – компонент **ValueListEditor**, для заполнения списка используем функцию **InsertRow**.

Листинг программы:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Form1–>Close(); }
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u, r, rn, rk, i;
if ((ComboBox1–>Text).Length()==0)
{ShowMessage("Вы не выбрали значение напряжения"); return;}
u=StrToFloat(ComboBox1–>Text);
// не будем повторяться, делая защиту от ввода нечисловых данных
```

```
rn=StrToFloat(LabeledEdit1–>Text); rk=StrToFloat(LabeledEdit2–>Text);
if (rn<=0||rk<=0||rn>rk)
{ ShowMessage("Неправильно введен диапазон значений сопротивления");
return; }
r = rn;
do
{ i=u/r;
ValueListEditor1–>InsertRow(FloatToStr(r),FloatToStr(i),true);
r=r+(rk–rn)/10; }
while (r<=rk);
}
```

Продолжим вносить изменения в задачу: для ввода значения напряжения используем компонент **ComboBox**, но заполнять список будем программно случайными числами; для ввода значений сопротивления используем компонент **ListBox**, но реализуем в нем множественный выбор в списке альтернатив; для вывода результатов используем компонент **ValueListEditor** (рис. 3.3.).

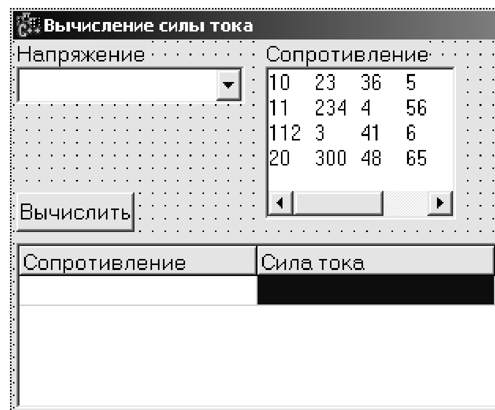

Рис. 3.3. Форма разрабатываемого приложения
в режиме проектирования

Листинг программы:

```
// формируем список ComboBox при активации формы
void __fastcall TForm1::FormActivate(TObject *Sender)
{
randomize();
// массив целых чисел, из которых будет состоять список ComboBox
int n[10];  int f; // флажок
// функция random генерирует случайное число
// в интервале от 0 до 100
```

```
n[0]=random(101)+1;
ComboBox1–>Items–>Add(IntToStr(n[0]));
for (int i=1;i<10;i++)
{ n[i]=random(101)+1;f=0;
for (int j=0;j<i;j++)
{
// делаем проверку, чтобы список заполнялся несовпадающими числами
if (n[j]==n[i]){f=1;break;}
// оператор break прерывает выполнение тела цикла и передает
// управление следующему за циклом выполняемому оператору
}
if (f==1) i=i–1;
else ComboBox1–>Items–>Add(IntToStr(n[i])); }
}
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u, r, i; int k;
if ((ComboBox1–>Text)=="")
{ShowMessage("Необходимо выбрать значение напряжения"); return;}
u=StrToFloat(ComboBox1–>Text);
// если в списке не выделено ни одного элемента
if ((ListBox1–>SelCount)<1)
{ShowMessage("Укажите значения сопротивления"); return;}
k=ListBox1–>Items–>Count; // общее количество элементов списка
for (int j=0;j<k;j++)
{ if (ListBox1–>Selected[j])
{ r=StrToFloat(ListBox1–>Items–>Strings[j]); i=u/r;
ValueListEditor1–>InsertRow(FloatToStr(r),FloatToStr(i),true); }
} }
```

Рассмотрим компонент **StringGrid**. Данный компонент предназначен в первую очередь для отображения текстовой информации, однако он может отображать и графическую информацию.

Основные свойства компонента **StringGrid**:

- **Cells** [**int** ACol][**int** ARow] – определяет содержимое ячейки с индексом столбца ACol и индексом строки ARow (доступно во время выполнения);
- **ColCount** и **RowCount** – определяют число столбцов и строк соответственно;
- **FixedCols** и **FixedRows** – определяют число фиксированных (не прокручиваемых) столбцов и строк;
- **FixedColor** – определяет цвет фона фиксированных ячеек;

- **LeftCol** и **TopRow** – определяют индексы первого видимого на экране в данный момент прокручиваемого столбца и первой видимой прокручиваемой строки соответственно;
- **ScrollBars** – определяет наличие в таблице полос прокрутки;
- **Options.goEditing** – определяет возможность редактирования содержимого таблицы;
- **Col** и **Row** – определяют индексы столбца и строки выделенной ячейки (компонент **StringGrid** в основном используется для выбора пользователем каких-либо значений, отображенных в ячейках).

Внесем изменения в нашу программу: используем для вывода результатов компонент **StringGrid**.

Модифицированный участок программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
float u, r, i; int k;
if ((ComboBox1–>Text)=="")
{ShowMessage("Необходимо выбрать значение напряжения");
return;}
u=StrToFloat(ComboBox1–>Text);
if ((ListBox1–>SelCount)<1)
{ShowMessage("Укажите значения сопротивления"); return;}
k=ListBox1–>Items–>Count;
int b=0; // номер строки таблицы StringGrid
// максимальное число строк таблицы = числу альтернатив списка ListBox
StringGrid1–>RowCount=k;
for (int j=0;j<k;j++)
{ if (ListBox1–>Selected[j])
{ r=StrToFloat(ListBox1–>Items–>Strings[j]); i=u/r;
StringGrid1–>Cells[0][b]=FloatToStr(r); StringGrid1–>Cells[1][b]=FloatToStr(i);
b++; } }
}
```

3.4. Ввод и вывод целых чисел

В C++ Builder имеются специализированные компоненты, обеспечивающие ввод целых чисел: **UpDown** (вкладка **Win32**) и **CSpinEdit** (вкладка **Samples**).

Компонент **UpDown** превращает окно редактирования **Edit** в компонент, в котором пользователь может выбирать целое число, изменяя его с помощью кнопок со стрелками.

Рассмотрим основные свойства компонента **UpDown**:

- **Orientation** – определяет, расположатся ли кнопки по вертикали или по горизонтали;
- **ArrowKeys** – определяет, будут ли управлять компонентом клавиши клавиатуры со стрелками;
- **Thousands** – определяет наличие или отсутствие разделительного пробела между каждыми тремя цифрами разрядов вводимого числа;
- **Min** – минимальное значение числа;
- **Max** – максимальное значение числа;
- **Increment** – задает приращение числа при каждом нажатии на кнопку;
- **Position** – определяет текущее значение числа;
- **Wrap** – определяет, как ведет себя компонент при достижении максимального или минимального значений.

Свойства компонента **CSpinEdit** похожи на свойства компонента **UpDown**, только имеют другие имена: свойства **Min**, **Max**, **Position** называются соответственно **MinValue**, **MaxValue**, **Value**, и в целом компонент **CSpinEdit** во многих отношениях удобнее сочетания компонентов **UpDown** и **Edit**.

Выполните следующее:

- Перенесите на форму компоненты **UpDown** и **Edit**, расположив компонент **Edit** там, где это требуется, а **UpDown** – в любом месте формы;
- в выпадающем списке свойства **Associate** компонента **UpDown** выберите **Editl** (компоненты соединятся);
- установите свойство **ReadOnly** компонента **Edit** в значение **true**, чтобы пользователь не смог ввести какой-либо свой текст, или, наоборот, установите значение **false**.

Создадим новое приложение, вычисляющее силу тока по задаваемым значениям напряжения и сопротивления с использованием рассмотренных компонентов.

Листинг программы:

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Form1–>Close(); }
void __fastcall TForm1::Button1Click(TObject *Sender)
{
int u,r,i;
u=UpDown1–>Position;
r=CSpinEdit1–>Value;
i=u/r;
Label3–>Caption="Сила тока "+IntToStr(i)+" A";
// рассчитанное значение можно вывести и так: CSpinEdit2–>Value=i;
}
```

3.5. Обработка массива значений

Создадим приложение, реализующее задание массива значений с помощью компонента **ValueListEditor**. Первый столбец компонента будем программно заполнять порядковыми номерами элементов, второй столбец – случайными числами. Во время выполнения приложения массив можно будет редактировать, обрабатывать и очищать.

Для создания приложения используем следующие компоненты: **ValueListEditor**, **Label** (4), **Button** (4), **CSpinEdit** (рис. 3.4.).

Рис. 3.4. Форма разрабатываемого приложения
в режиме проектирования (а) и в режиме выполнения (б)

Код программы:

```
int i, n, kp, ko;
float x[100], sum, sr;
// Обработать массив
void __fastcall TForm1::Button1Click(TObject *Sender)
{ n=ValueListEditor1–>RowCount–1;
sum=0;kp=0;ko=0;
try
{ for (i=1;i<=n;i++)
{ x[i]=StrToFloat(ValueListEditor1–>Cells[1][i]);
sum=sum+x[i];
if (x[i]>0) { kp=kp+1; }
if (x[i]<0) { ko++; }
} sr=sum/n; }
catch(EConvertError &e)
{ ShowMessage ("Ошибка при вводе данных"); return; }
```

```
Label2–>Caption="Среднее арифм = "+FloatToStr(sr);
Label3–>Caption="Количество элементов > 0 = "+IntToStr(kp);
Label4–>Caption="Количество элементов < 0 = "+IntToStr(ko);
}
// Завершить работу
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Close(); }
// Очистить массив
void __fastcall TForm1::Button3Click(TObject *Sender)
{
n=ValueListEditor1–>RowCount–1;
for (i=1;i<=n;i++)
{ ValueListEditor1–>Cells[0][i]=""; ValueListEditor1–>Cells[1][i]=""; }
Button4–>Enabled=true;
}
// Блокировка кнопок при создании формы
void __fastcall TForm1::FormCreate(TObject *Sender)
{ Button1–>Enabled=false; Button3–>Enabled=false; }
// Задать массив
void __fastcall TForm1::Button4Click(TObject *Sender)
{
randomize;
n=CSpinEdit1–>Value;
for (i=1;i<=n;i++)
{ ValueListEditor1–>InsertRow(i,random(100)–50,true); }
Button3–>Enabled=true; Button1–>Enabled=true; Button4–>Enabled=false;
}
```

Рассмотрим второй способ очистки массива:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{ ValueListEditor1–>Strings–>Clear(); }
```

Третий способ очистки массива:

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{ n=ValueListEditor1–>RowCount–1;
for (i=1;i<=n;i++)
{ ValueListEditor1–>DeleteRow(1); } }
```

Создадим приложение, реализующее задание массива значений с помощью компонента **StringGrid**. Компонент будем заполнять программно случайными числами. Во время выполнения приложения массив можно будет редактировать, обрабатывать и очищать.

Для создания приложения используем следующие компоненты: **StringGrid**, **Label** (2), **Button** (4), **CSpinEdit** (2) (рис. 3.5.).

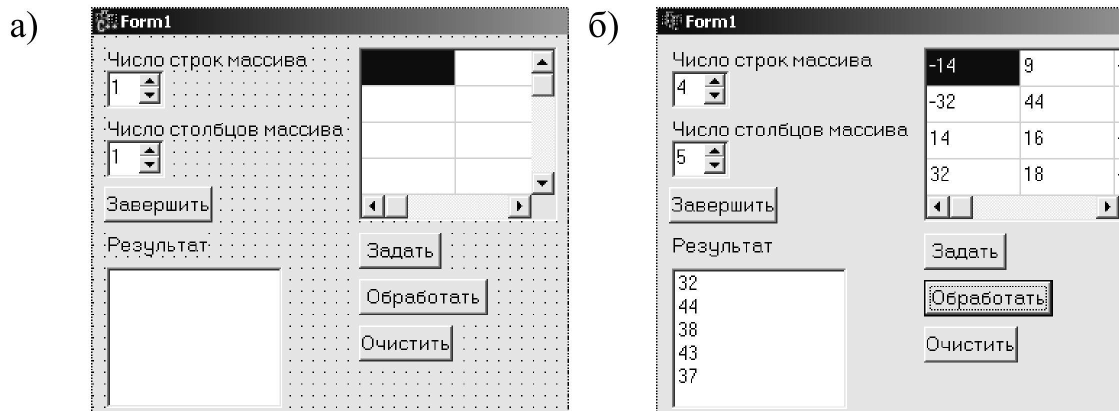

Рис. 3.5. Форма разрабатываемого приложения
в режиме проектирования (а) и в режиме выполнения приложения (б)

Примечание. Приведенная задача легко решается без использования индексированных переменных.

Код программы:

```
int i, j, n, m;
float a, max;
// Завершить работу
void __fastcall TForm1::Button4Click(TObject *Sender)
{ Close(); }
// Задать массив
void __fastcall TForm1::Button1Click(TObject *Sender)
{
randomize();
m=CSpinEdit1–>Value; n=CSpinEdit2–>Value;
StringGrid1–>RowCount=m; StringGrid1–>ColCount=n;
for (i=0;i<n;i++)
{ for (j=0;j<m;j++)
{ StringGrid1–>Cells[i][j]=random(100)–50; }
}
Button3–>Enabled=true; Button2–>Enabled=true;
}
// Обработать массив
void __fastcall TForm1::Button3Click(TObject *Sender)
{ ListBox1–>Clear();
try
{ for (i=0;i<n;i++)
{ max=StrToFloat(StringGrid1–>Cells[i][0]);
for (j=0;j<m;j++)
```

```
{ a=StrToFloat(StringGrid1–>Cells[i][j]);
if (a>max) { max=a; }}
ListBox1–>Items–>Add(FloatToStr(max)); }
}
catch(EConvertError &e)
{ ShowMessage("Ошибка ввода"); return; }
}
// Очистить массив
void __fastcall TForm1::Button2Click(TObject *Sender)
{ for (i=0;i<n;i++)
{ for (j=0;j<m;j++)
{ StringGrid1–>Cells[i][j]=""; } } }
```

3.6. Перенос приложения на другой компьютер

Программа, созданная в С++ Builder, использует различные библиотеки. Чтобы программа могла работать на другом компьютере, помимо exe–файла на этот компьютер надо перенести RTL–библиотеку и используемые программой пакеты или включить библиотеку и пакеты в exe–файл (что существенно увеличит размер exe–файла).

Чтобы включить в выполняемый файл RTL–библиотеку и используемые программой пакеты, необходимо в меню **Project** выбрать команду **Options** и во вкладках **Linker** и **Packages** сбросить флажки **Use dynamic RTL** и **Build with runtime packages** соответственно. После этого следует выполнить перекомпоновку программы.

3.7. Файловый ввод/вывод с помощью компонентов

Работа с файлами в С++ Builder может производиться несколькими принципиально различными (с точки зрения пользователя) способами:

- работа с использованием библиотечных компонентов;
- работа с файлами как с потоками в стиле С;
- работа с файлами как с потоками в стиле С++.

Работа с текстовыми файлами может осуществляться с помощью методов **LoadFromFile** (класс **TStrings**) и **SaveToFile** (класс **TStringList**). Эти классы описывают списки строк. Если в приложении требуется прочитать содержимое некоторого текстового файла и обработать его или сохранить какие-либо результаты в файле, то для этого необходимо объявить и создать две глобальные переменные:

- список типа **TStringList**, в котором будет храниться текст файла;
- строковую переменную типа **AnsiString**, в которой формируется имя файла.

Пример:

```
TStringList *List = new TStringList;
// List – имя списка, задаваемое программистом
AnsiString SFile = "Test.txt";
```

Если файл расположен не в текущем каталоге, то в этом случае необходимо указать путь к файлу и сдвоенные обратные слэши в записи пути, например: AnsiString SFile = "D:\\Prog\\Test.txt".

В момент, когда вы хотите загрузить в свой список файл, следует выполнить оператор

```
List–>LoadFromFile(SFile);
```

или выполнить следующий код с использованием защиты:

```
try
{ List–>LoadFromFile(SFile); }
catch ( ... )
{ ShowMessage("Файл \"" + SFile +"\" не найден"); }
```

Если файл нормально загрузился в список **List**, то вы можете работать с его текстом.

Основные свойства, определенные в классе **TStringList**:

- **Count** – число строк в списке (свойство только для чтения);
- **Strings [int Index]** – текст строки с указанным индексом в списке (нумерация индексов начинается с нуля, каждая строка имеет тип **AnsiString**).

К методам класса **TStringList**, наследованным от класса **TStrings,** относятся методы **Add, Clear**, **Delete**, **Exchange**, **IndexOf**, **Insert** и многие другие.

Примечание. При обработке отдельных строк можно использовать операции и методы, предусмотренные для строк типа **AnsiString**.

Чтобы сохранить какие-либо результаты в файле, нужно выполнить оператор

```
List–>SaveToFile (SFile);
```

где SFile – прежнее или новое имя файла.

При открытии и сохранении файла можно воспользоваться стандартными диалогами Windows, а именно: компонентами **OpenDialog** (диалог «Открыть файл») и **SaveDialog** (диалог «Сохранить файл как...») из вкладки **Dialogs**.

Рассмотрим основные свойства указанных компонентов:

- **FileName** – основное свойство, в котором возвращается в виде строки выбранный пользователем файл (значение этого свойства можно

задать и перед обращением к диалогу, тогда оно появится в диалоге как значение по умолчанию в окне «Имя файла»);

- **Filter** – определяет типы искомых файлов, появляющихся в диалоге в выпадающем списке «Тип файла»;
- **FilterIndex** – определяет номер фильтра, который будет по умолчанию показан пользователю в момент открытия диалога;
- **InitialDir** – определяет начальный каталог, который будет открыт в момент начала работы пользователя с диалогом;
- **DefaultExt** – определяет значение расширения файла по умолчанию;
- **Title** – позволяет вам задать заголовок диалогового окна;
- **Options** – определяет условия выбора файла*.

*Операции открытия и сохранения файла можно поместить в обработчик события **OnClick** компонента **Button**:*

```
if (OpenDialog1–>Execute())
List–>LoadFromFile(OpenDialog1–>FileName);
или:
if (SaveDialog1–>Execute())
List–>SaveToFile(SaveDialog1–>FileName);
```

Рассмотрим приложение, в результате выполнения которого будут генерироваться 100 случайных чисел в интервале от 0 до 100, записываться в двумерный массив и выводиться в файл.

Форма данного приложения, как минимум, должна содержать компоненты **Button** и **SaveDialog**. Нажатие первой кнопки будет приводить к созданию файла, нажатие второй – к закрытию приложения.

Листинг программного кода первой кнопки:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
randomize();
// объявили и создали две необходимые переменные
TStringList *List = new TStringList;
AnsiString SFile1 = "D:\\Rez.txt";
float a[10][10]; // описание массива
for (int i=0;i<10;i++)
{
for (int j=0;j<10;j++)
{ a[i][j]=random(100)+1; // генерация элементов массива
List–>Add(FloatToStr(a[i][j]));
```

* Подробная информация: Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : Бином-Пресс, 2003. – С. 240 (раздел 3.10.2).

```
// добавление элементов массива в список }
}
if (SaveDialog1–>Execute())
List–>SaveToFile(SaveDialog1–>FileName);
// сохранение списка в файл
}
```

Модифицируем приложение таким образом, чтобы массив записывался в файл в виде таблицы значений.

Модифицированный листинг программного кода первой кнопки:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
randomize();
TStringList *List = new TStringList;
AnsiString SFile1 = "D:\\Rez.txt";
float a[10][10];
AnsiString t; // для хранения строки массива
for (int i=0;i<10;i++)
{ t="";
for (int j=0;j<10;j++)
{ a[i][j]=random(100)+1;
t = t + FloatToStr(a[i][j])+" "; (через пробел)
// t = t + FloatToStr(a[i][j])+"\t"; (через табуляцию) }
List–>Add(t); }
if (SaveDialog1–>Execute())
List–>SaveToFile(SaveDialog1–>FileName);
}
```

Создадим новое приложение, в результате выполнения которого данные из созданного файла Rez.txt, представляющего собой один столбец числовых значений, будут считываться в одномерный массив, затем будут находиться максимальный и минимальный элементы массива и найденные значения будут выводиться на экран.

Для создания приложения используем обязательные компоненты формы (**Button**, **OpenDialog**, **Label** (2)) и необязательный компонент (**ProgressBar**).

Листинг программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TStringList *List = new TStringList;
AnsiString SFile = "D:\\Rez.txt";
int n;
```

```
// загрузка данных из файла
if (OpenDialog1–>Execute())
List–>LoadFromFile(OpenDialog1–>FileName);
n=List–>Count; // количество строк списка
float mpmax,mpmin,mp[5000];
// объявляем достаточно большой размер массива
for (int i=0;i<n;i++)
{ // ProgressBar1–>Position = 1000 * i / n;
// отображение хода процесса
mp[i]=StrToFloat(List–>Strings[i]); }
// данные из списка записываем в массив
mpmin=mp[0];mpmax=mp[0];
for (int i=0;i<n;i++)
{ if (mp[i]>mpmax) {mpmax=mp[i];}
if (mp[i]<mpmin) {mpmin=mp[i];} }
Label1–>Caption="min = "+FloatToStr(mpmin);
Label2–>Caption="max = "+FloatToStr(mpmax);
}
```

Рассмотрим приложение, в результате выполнения которого из файла будет считываться таблица значений и вычисляться максимальные элементы столбцов.

Для создания приложения используем следующие компоненты: **Button** (2), **OpenDialog**, **Label** (2), **Memo**, **ListBox** (рис. 3.6.).

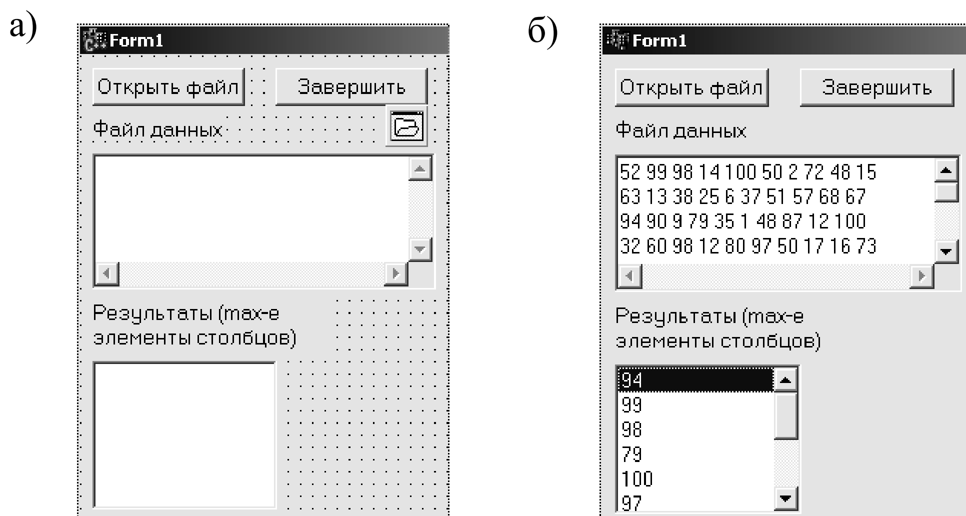

Рис. 3.6. Форма разрабатываемого приложения
в режиме проектирования (а) и в режиме выполнения (б)

Программный код:

```
// Завершить работу
void __fastcall TForm1::Button2Click(TObject *Sender)
{ Close(); }
// Открыть файл данных и обработать
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TStringList *List = new TStringList;
AnsiString SFile = "D:\\Rez.txt"; // можно без этой строки
AnsiString stroka, strChisl;
// stroka – для хранения строки файла; strChisl – для хранения отдель-
ного числа
int n, i, j, k, c, s;
// n – количество строк файла; k – длина строки; s – количество чисел
в строке
float m[2000][100], a;
Memo1–>Lines–>Clear(); ListBox1–>Clear();
if (OpenDialog1–>Execute())
List–>LoadFromFile(OpenDialog1–>FileName);
n=List–>Count; s=0;
for (i=0;i<n;i++)
{
stroka=List–>Strings[i]; Memo1–>Lines–>Add(stroka);
j=0; k=stroka.Length(); strChisl="";
// выделение чисел в строке
for(c=1;c<=k;c++)
{
if(stroka[c]!=' ') { strChisl=strChisl+stroka[c]; }
// stroka[c]!='\t' – если числа разделены с помощью Tab (и в конце
строки есть)
else
{ m[i][j]=StrToFloat(strChisl); j++; strChisl=""; }
}
if(j>s){s=j;}
}
// поиск максимальных элементов столбцов
for (j=0;j<s;j++)
{ a=m[0][j];
for (i=0;i<n;i++)
{ if(m[i][j]>a){a=m[i][j];}}
ListBox1–>Items–>Add(FloatToStr(a)); }
}
```

Программный код работает, если числа разделены пробелом и в конце строки тоже есть пробел. Такой файл создается в рассмотренном ранее примере. Если в конце строки нет пробела и не нажата клавиша Tab, то приведенный код не работает.

Модифицированное приложение (рис. 3.7.) решает указанную проблему.

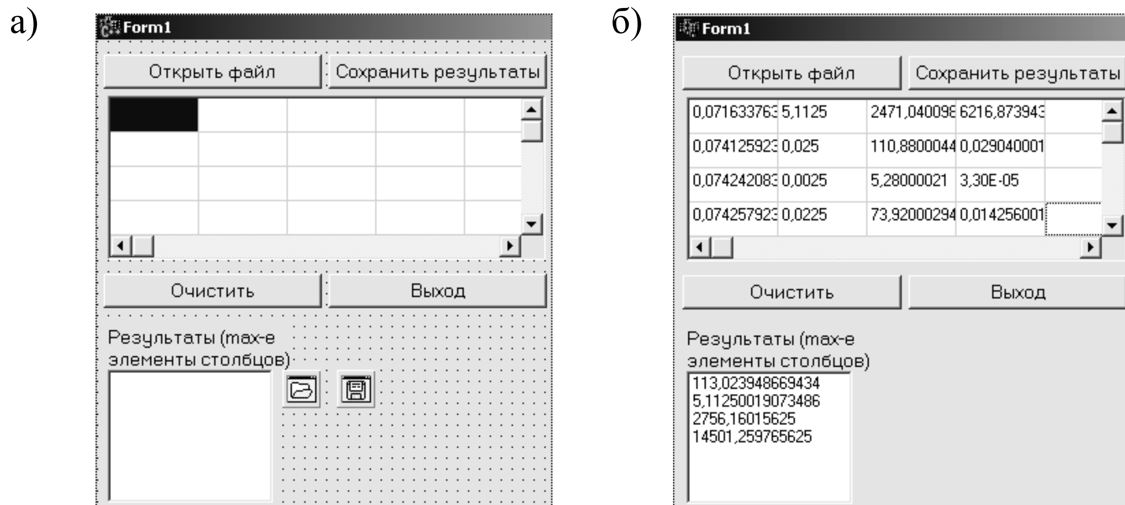

Рис. 3.7. Форма разрабатываемого приложения
в режиме проектирования (а) и в режиме выполнения (б)

Для создания приложения использовались следующие компоненты: **Button** (4), **OpenDialog**, **SaveDialog**, **Label**, **StringGrid**, **ListBox**.

Листинг программы:

```
int n, i, j, k, c, s;
// n – количество строк файла; k – длина строки
// s – количество чисел в строке
float m[2000][100], max[100], a;
// m[2000][100] – массив исходных данных;
// max[100] – массив результатов
// Завершить работу
void __fastcall TForm1::Button3Click(TObject *Sender)
{ Close(); }
// Открыть файл данных и обработать
void __fastcall TForm1::Button1Click(TObject *Sender)
{
TStringList *List = new TStringList;
AnsiString SFile = "D:\\Rez.txt";
AnsiString stroka, strChisl;
if (OpenDialog1–>Execute())
```

```
List–>LoadFromFile(OpenDialog1–>FileName);
n=List–>Count; s=0;
StringGrid1–>RowCount=n;
StringGrid1–>ColCount=10;
// максимально возможное число столбцов
for (i=0;i<n;i++)
{
stroka=List–>Strings[i]; j=0; k=stroka.Length(); strChisl="";
// выделение чисел в строке
for(c=1;c<=k;c++)
{
if(stroka[c]!='\t') { strChisl=strChisl+stroka[c]; }
else
{StringGrid1–>Cells[j][i]=strChisl;    m[i][j]=StrToFloat(strChisl);    j++;
strChisl="";}
if (c==k)
{StringGrid1–>Cells[j][i]=strChisl;    m[i][j]=StrToFloat(strChisl);    j++;
strChisl="";}
}
if(j>s){s=j;}
}
// поиск максимальных элементов столбцов
for (j=0;j<s;j++)
{
a=m[0][j];
for (i=0;i<n;i++)
{ if(m[i][j]>a){a=m[i][j];} }
max[j]=a; ListBox1–>Items–>Add(FloatToStr(a)); }
}
// Очистить компоненты
void __fastcall TForm1::Button4Click(TObject *Sender)
{
ListBox1–>Clear();
for (i=0;i<n;i++)
{ for (j=0;j<s;j++)
{ StringGrid1–>Cells[i][j]=""; } }
}
// Сохранение результатов
void __fastcall TForm1::Button2Click(TObject *Sender)
{
TStringList *List1 = new TStringList;
AnsiString SFile1 = "D:\\Rez1.txt";
```

```
for (j=0;j<s;j++)
{ List1–>Add(FloatToStr(max[j])); }
if (SaveDialog1–>Execute())
List1–>SaveToFile(SaveDialog1–>FileName);
}
```

Если файл необходимо открыть для того, чтобы просмотреть его, что-либо в нем отредактировать и сохранить, то можно обойтись без описанного выше объекта типа **TStringList**. Для этих целей проще воспользоваться многострочными окнами редактирования типов **TMemo** или **TRichEdit***. Свойства **Lines** этих компонентов имеют тип **TStrings**, что позволяет применять к ним непосредственно методы **LoadFromFile** и **SaveToFile**.

Пример:

```
Memol–>Lines–>LoadFromFile(SFile);
RichEditl–>Lines–>LoadFromFile(SFile).
```

С помощью компонентов C++ Builder можно работать не только с текстовыми файлами, но и с файлами изображений и мультимедиа**.

4. ОТОБРАЖЕНИЕ ГРАФИЧЕСКОЙ ИНФОРМАЦИИ

Характеристики компонентов отображения графической информации приведены в табл. 4.1.

Таблица 4.1

Компоненты отображения графической информации

Компонент	Вкладка	Описание
Image (изображение)	**Additional**	Используется для отображения графики (пиктограмм, битовых матриц, метафайлов).
PaintBox (окно для рисования)	**System**	Используется для создания на форме некоторой области, в которой можно рисовать.
DrawGrid (таблица рисунков)	**Additional**	Используется для отображения в строках и столбцах нетекстовых данных.
Chart (диаграммы и графики)	**Additional**	Используется для создания диаграмм и графиков.

* Подробное описание работы с компонентами Memo и RichEdit: Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : Бином-Пресс, 2003. – С. 155 (раздел 3.2.4).

** Подробная информация: Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : Бином-Пресс, 2003. – С. 373 (раздел 6).

Продолжение табл. 4.1

Компонент	Вкладка	Описание
Chartfx (диаграммы и графики)	**ActiveX**	Редактор диаграмм и графиков.
F1Book (страницы Excel)	**ActiveX**	Компонент ввода и обработки числовой информации, аналогичный страницам Excel (в том числе в графическом виде).
VtChart (диаграммы)	**ActiveX**	Окно построения диаграмм.
Animate (воспроизведение немых клипов)	**Win32**	Используется для воспроизведения немых клипов AVI, подобных используемым в Windows клипам, копирования файлов и т. п.
MediaPlayer (аудио- и видеоплеер)	**System**	Используется для создания панели управления воспроизведением звуковых файлов, видеофайлов, а также устройств мультимедиа.
ProgressBar (отображение хода процесса)	**Win32**	Используется для отображения хода процессов, занимающих значительное время.
CGauge (отображение хода процесса)	**Samples**	Используется для создания индикатора хода процесса в виде линейки, текста или секторной диаграммы.

Кроме того, отображать графическую информацию можно на поверхности любого оконного компонента, имеющего свойство **Canvas**.

4.1. Компоненты ProgressBar и CGauge

Рассмотрим основные свойства компонента **ProgressBar** (**CGauge**):

- **Max** (**MaxValue**) – максимальное значение позиции (Position, Progress), которое соответствует завершению отображаемого процесса (по умолчанию задается в процентах и равно 100);
- **Min** (**MinValue**) – начальное значение позиции (Position, Progress), которое соответствует началу отображаемого процесса;
- **Position** (**Progress**) – позиция, которую можно задавать по мере протекания процесса, начиная со значения Min (MinValue), и заканчивая значением Max (MaxValue). Если минимальное и максимальное значения выражены в процентах, то позиция – это процент завершенной части процесса;

- **Smooth** (–) – непрерывное (при значении true) или дискретное отображение процесса;
- **Step** (–) – шаг приращения позиции, используемый в методе Stepit (значение по умолчанию – 10);
- **Orientation** (–) – ориентация шкалы компонента;
- – (**ShowText**) – текстовое отображение процента выполнения на фоне диаграммы;
- – (**Kind**) – тип диаграммы.

Отображение хода процесса можно осуществлять, задавая значение свойства **Position** (**Progress**). Например, если полная длительность процесса характеризуется значением целой переменной **Count** (объем копируемого файла, количество циклов какого-либо процесса), а выполненная часть – целой переменной **Current**, то задавать позицию диаграммы в случае, если используются значения минимальной и максимальной позиции по умолчанию (т. е. 0 и 100), можно следующими операторами (соответственно для ProgressBar и CGauge):

```
ProgressBar1–>Position = 100 * Current / Count;
```

или

```
CGauge1–>Progress = 100 * Current / Count;
```

Можно поступать иначе: задать сначала значение максимальной величины, равное **Count**, а затем в ходе процесса задавать позицию, равную **Current.**

Пример:

```
CGauge1–>MaxValue = Count;
…
CGauge1–>Progress = Current;
```

Пример использования компонента ProgressBar приведен в подразделе 3.7 данного практикума.

4.2. Компонент Chart

Рассмотрим основные характеристики компонента Chart. Данный компонент является контейнером объектов **Series** типа **TChartSeries** – серий данных, характеризующихся различными стилями отображения.

Каждый компонент может включать несколько серий. При отображении графика каждая серия будет соответствовать одной кривой на графике. При отображении диаграммы можно несколько различных серий наложить друг на друга.

Рассмотрим некоторые свойства компонента **Chart**:

- **AllowPanning** – определяет возможность прокручивания наблюдаемой части графика во время выполнения при нажатии правой кнопки мыши;
- **AllowZoom** – позволяет изменять во время выполнения масштаб изображения, вырезая фрагменты диаграммы или графика курсором мыши;
- **Title** – заголовок диаграммы;
- **Foot** – подпись под диаграммой (по умолчанию отсутствует), текст подписи определяется подсвойством **Text**;
- **Frame** – определяет рамку вокруг диаграммы;
- **Legend** – легенда диаграммы (список обозначений);
- **MarginLeft, MarginRight, MarginTop, MarginBottom** – значения левого, правого, верхнего и нижнего полей соответственно;
- **BottomAxis, LeftAxis, RightAxis** – определяют характеристики нижней, левой и правой осей соответственно;
- **LeftWall, BottomWall, BackWall** – определяют характеристики левой, нижней и задней граней области трехмерного отображения графика соответственно;
- **SeriesList** – список серий данных, отображаемых в компоненте;
- **View3D** – разрешает или запрещает трехмерное отображение диаграммы;
- **View3DOptions** – характеристики трехмерного отображения;
- **Chart3DPercent** – масштаб трехмерности.

Рассмотрим пример построения графиков трех функций: sin(x), cos(x), x^3.

Выполните следующее:

- Добавьте на форму компонент **Chart** и установите значения свойств **View3D** = false; **Title** = Совмещенные графики; **Color** = цвет, который вас устраивает. Для установки всех свойств диаграммы необходимо вызвать окно **Editing Chart1** (редактор диаграммы) двойным щелчком мыши на компоненте **Chart** либо щелчком на нем правой кнопкой мыши и выбором команды **Edit Chart**.
- Вызовите окно редактора диаграммы. На вкладке **Series** (2-ого уровня) используйте кнопку **Add** для добавления трех серий (выберите тип графика **Line**), кнопку **Title** используйте для задания соответствующих заголовков. На вкладке **Axis | Title** задайте названия осей, на вкладке **Legend** – расположение легенды, на вкладке **Series** (1–ого уровня) используйте кнопку **Border** для задания толщины линий графиков.
- Область значений третьего графика (x^3) значительно отличается от первых двух, поэтому в окне редактора на вкладке **Series | General** установите для него в качестве вертикальной оси правую ось.

Для отображения графиков используются методы объекта **Series**. Рассмотрим три основных метода отображения графиков:

- Метод **Clear** очищает серию от занесенных ранее данных.
- Метод **Add** позволяет добавить на диаграмму новую точку:

Add (double AValue, AnsiString ALabel, TColor AColor);

где AValue – добавляемое значение; ALabel – название, которое будет отображаться на диаграмме и в легенде (необязательный параметр, его можно задать пустым: ""); AColor – цвет.

- Метод **AddXY** позволяет добавить новую точку в график функции:

AddXY (double AXValue, double AYValue, AnsiString ALabel, TColor AColor);

где параметры AXValue и AYValue соответствуют аргументу и функции; параметры ALabel и AColor имеют тот же смысл, что и в методе **Add**.

Программный код, обеспечивающий построение трех графиков, может иметь следующий вид:

```
Series1–>Clear();
Series2–>Clear();
Series3–>Clear();
double x;
x=-6.28;
while (x<6.28)
{
Series1–>AddXY(x, sin(x), "",clRed);
Series2–>AddXY(x, cos(x), "",clBlue);
Series3–>AddXY(x, x*x*x, "",clBlack);
x=x+0.1;
}
```

Этот программный код можно включить в обработку щелчка какой-либо кнопки, в команду меню или просто в событие **OnCreate** формы. Операторы **Clear** нужны, если в процессе работы приложения необходимо обновлять данные. Без этих операторов повторное выполнение методов **Add** и **AddXY** только добавит новые точки, не удалив прежние.

Кроме того, чтобы данный программный код работал (т. е. распознавались тригонометрические функции), необходимо подключить библиотеку math.h.

Модифицируем созданное приложение таким образом, чтобы начальные и конечные значения аргументов функций, а также количество точек задавались во время выполнения приложения (рис. 3.8.).

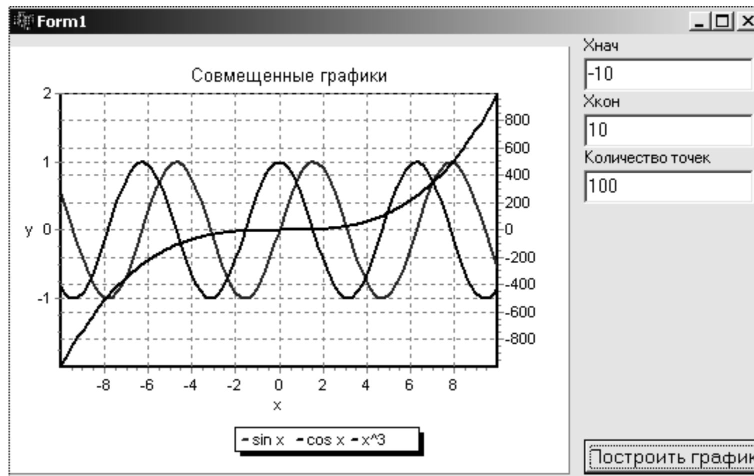

Рис. 3.8. Модифицированная форма приложения

Модифицированный листинг программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Series1–>Clear();
Series2–>Clear();
Series3–>Clear();
double x,xn,xk,h;
int n;
xn=StrToFloat(LabeledEdit1–>Text);
xk=StrToFloat(LabeledEdit2–>Text);
n=StrToInt(LabeledEdit3–>Text);
x=xn;
h=(xk–xn)/n;
while (x<=xk)
{
Series1–>AddXY(x, sin(x), "",clRed);
Series2–>AddXY(x, cos(x), "",clBlue);
Series3–>AddXY(x, x*x*x, "",clBlack);
x=x+h; }
}
```

Рассмотрим пример построения диаграммы. Например, отразим на диаграмме успеваемость учебных групп.

Выполните следующее:

- Добавьте на форму компонент **Chart** и установите значения свойств Title = Успеваемость групп; **Color**, **LeftWall | Color**, **BottomWall | Color**, **BackWall | Color** = цвета, которые вас устраивают.
- Вызовите окно редактора диаграммы. На вкладке **Series** (2–ого уровня) используйте кнопку **Add** для добавления серии (выберите тип диа-

граммы **Bar**); на вкладке **Series | Marks** задайте, что будет написано на ярлычках, относящихся к отдельным элементам диаграммы; на вкладке **Series | General** отключите отображение легенды.

- Добавьте на форму кнопку и включите в ее обработчик события **OnClick** следующий программный код:

```
float a1=3.5;
float a2=4.5;
float a3=3.25;
float a4=4.75;
// a1, a2, a3, a4 – средние баллы групп
Series1–>Clear();
Series1–>Add(a1,"9 ВТА",clBlue);
Series1–>Add(a2,"8 ВТА",clGreen);
Series1–>Add(a3,"7 ВТА",clYellow);
Series1–>Add(a4,"6 ВТА",clRed);
```

Для данных, отображаемых в диаграмме, можно предусмотреть две серии разных видов (например, **Bar** и **Horiz. Bar**), и ввести в событие **OnClick** второй кнопки код, изменяющий по требованию пользователя тип диаграммы. Для этого нужно вставить в конце приведенного кода операторы

```
Series2–>Assign(Series1);
Series2–>Active=false;
```

Первый из этих операторов переписывает данные, помещенные в **Series1**, в серию **Series2**, а второй оператор делает серию **Series2** невидимой.

Смену типа диаграммы можно осуществить с помощью кода

```
Series1–>Active=false;
Series2–>Active=true;
```

Внесем в созданное приложение необходимые изменения.

Листинг программы:

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
Series1–>Active=true;
float a1=3.5;
float a2=4.5;
float a3=3.25;
float a4=4.75;
Series1–>Clear();
Series2–>Clear();
```

```
Series1–>Add(a1,"9 BTA",clBlue);
Series1–>Add(a2,"8 BTA",clGreen);
Series1–>Add(a3,"7 BTA",clYellow);
Series1–>Add(a4,"6 BTA",clRed);
Series2–>Assign(Series1);
Series2–>Active=false;
Button2–>Enabled=true;
}
void __fastcall TForm1::Button2Click(TObject *Sender)
{
Series1–>Active=false;
Series2–>Active=true;
}
void __fastcall TForm1::FormCreate(TObject *Sender)
{ Button2–>Enabled=false; }
```

Усложним задачу: построим круговую диаграмму, отражающую затраты по каждому виду продукта в продовольственной корзине. Исходные данные для диаграммы будем задавать не программно, а во время выполнения приложения с помощью, например, компонента **ValueListEditor** (рис. 3.9.).

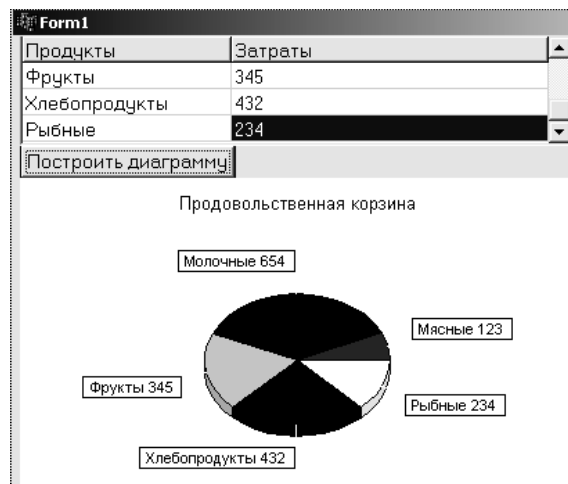

Рис. 3.9. Модифицированная форма приложения

Выполните следующее:

- Добавьте на форму компонент **ValueListEditor**. Измените по своему усмотрению текст заголовков (свойство **TitleCaptions**). Установите значения всех составляющих свойства **KeyOptions** равными **true**.
- Добавьте на форму компонент **Chart** и установите по своему усмотрению значения свойств **Title**, **Color**, **LeftWall | Color**, **BottomWall | Color**, **BackWall | Color**.

- Вызовите окно редактора диаграммы. Добавьте серию (выберите тип диаграммы **Pie**); на вкладке **Series | Format** включите опцию **Circled Pie**, которая обеспечивает при любом размере компонента **Chart** отображение диаграммы в виде круга; на вкладке **Series | Marks** задайте, что будет написано на ярлычках, относящихся к отдельным сегментам диаграммы.
- Добавьте на форму кнопку и включите в ее обработчик события **OnClick** следующий программный код:

```
Series1–>Clear();
int n;
AnsiString x; // для хранения названий продуктов
float y; // для хранения затрат на продукты
TColor z[10]; // для хранения различных цветов сегментов диаграммы
z[1]=clRed; z[2]=clNavy; z[3]=clLime; z[4]=clBlue; z[5]=clYellow;
z[6]=clAqua; z[7]=clBlack; z[8]=clGreen; z[9]=clMaroon; z[10]=clOlive;
n=(ValueListEditor1–>RowCount); // количество строчек
for (int i=1;i<n;i++)
{
x=ValueListEditor1–>Cells[0][i];
// взяли из столбца 0 название продукта
y=StrToFloat(ValueListEditor1–>Cells[1][i]);
// взяли из столбца 1 значение затрат
Series1–>Add(y,x,z[i]);
}
```

Рассмотрим некоторые свойства компонентов **Chartfx** и **VtChart**. Компонент **Chartfx** представляет собой законченный редактор диаграмм со встроенной инструментальной панелью. Свойства компонента во время проектирования можно менять или с помощью **Инспектора объектов,** или в окне свойств, вызываемом двойным щелчком мыши по компоненту, или правой кнопкой мыши.

В целом компоненты **VtChart** и **Chartfx** не могут составить конкуренцию компоненту **Chart**, владение которым позволяет программировать сколь угодно сложные многофункциональные отображения данных в графическом виде.

ЗАКЛЮЧЕНИЕ

Визуальное программирование, возникшее в Visual Basic и нашедшее дальнейшее воплощение в системах C++ Builder и Delphi фирмы Borland, стало шагом, значительно облегчившим жизнь программистов. Визуальное программирование позволяет свести проектирование пользовательского интерфейса к простым и наглядным процедурам, которые дают возможность за минуты или часы сделать то, что ранее требовало намного больше времени.

В данном практикуме детально рассмотрены две темы: ввод/вывод данных различной сложности и отображение графической информации, поэтому практикум предназначен в первую очередь для тех, кто учится программированию и делает первые шаги на пути освоения системы C++ Builder.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Становление информационного общества / С. В. Виноградов, Е. Б. Щелкунов, М. Е. Щелкунова, Л. Г. Косицина // Ученые записки Комсомольского-на-Амуре гос. техн. ун-та. Науки о природе и технике. – 2011. – № II–1(6). – С. 31-33.

2. Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : Бином-Пресс, 2003. – 1151 с.

3. Архангельский, А. Я. C++ Builder 6 : справ. пособие. Книга 1: Язык C++ / А. Я. Архангельский. – М. : Бином-Пресс, 2004. – 543 с.

4. Архангельский, А. Я. C++ Builder 6 : справ. пособие. Книга 2: Классы и компоненты / А. Я. Архангельский. – М. : Бином-Пресс, 2004. – 528 с.

5. Шамис, В. А. Borland C++ Builder 6. Для профессионалов / В. А. Шамис. – СПб. : Питер, 2004. – 798 с.

Учебное издание

Муратова Татьяна Александровна

ПРОГРАММИРОВАНИЕ В СРЕДЕ C++ BUILDER

Практикум

Научный редактор – кандидат технических наук,
профессор В. А. Тихомиров

Редактор Е. В. Назаренко

Подписано в печать 22.04.2014.
Формат 60 × 84 1/16. Бумага 60 г/м^2. Ризограф EZ570E.
Усл. печ. л. 3,02. Уч.-изд. л. 2,91. Тираж 50 экз. Заказ 26224.

Редакционно-издательский отдел
Федерального государственного бюджетного образовательного учреждения
высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»
681013, Комсомольск-на-Амуре, пр. Ленина, 27.

Полиграфическая лаборатория
Федерального государственного бюджетного образовательного учреждения
высшего профессионального образования
«Комсомольский-на-Амуре государственный технический университет»
681013, Комсомольск-на-Амуре, пр. Ленина, 27.