Министерство сельского хозяйства Российской Федерации

Федеральное государственное бюджетное
образовательное учреждение
высшего образования
«Пермский государственный аграрно-технологический университет»
имени академика Д.Н. Прянишникова»

А.Ю. БЕЛЯКОВ

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ В LAZARUS

Учебное пособие

Пермь
ФГБОУ ВО Пермский ГАТУ
2019

УДК 004.43
ББК 32.973-018.1

Рецензенты:

– профессор кафедры общей физики ПНИПУ, доктор физико-математических наук, профессор.

– доцент кафедры автоматики и телемеханики ПНИПУ, кандидат технических наук, доцент.

Прикладное программирование в Lazarus [Текст]: Учебное пособие / А.Ю. Беляков; М-во с.-х. РФ; ФГБОУ ВО Пермский ГАТУ. – Пермь: Изд-во ФГБОУ ВПО Пермский ГАТУ, 2019. –114 с.
ISBN _____

Данное издание является базовым пособием по изучению основ прикладного программирования в среде визуальной разработки приложений Lazarus. В пособии детально изложены принципы событийного программирования и на практических примерах проанализированы некоторые приемы работы с визуальными компонентами. Пособие ориентировано на самостоятельное освоение материала с исследованием программ в среде программирования Lazarus. Рассматриваемый материал требует первичного знания основ структурного и модульного программирования.

Пособие предназначено для студентов, обучающихся по направлению подготовки 09.04.03 Прикладная информатика.

УДК 004.43
ББК 32.973-018.1

Утверждено в качестве учебного пособия на заседании Методического совета ФГБОУ ВО Пермский ГАТУ (протокол № __ от __.__.2019 г.).

ISBN _____

Содержание

Введение

Современное программирование – это работа со множеством различных информационных технологий и технологий программирования. Для понимания текущей ситуации в сфере проектирования и реализации прикладных программ следует иметь представление об истории развития технологий программирования и их предназначении.

В процессе формирования и совершенствования языков программирования от низкоуровневых, предполагающих узкую специализацию и направленность на определенную архитектуру вычислительной машины, до языков высокого уровня, близких к естественному языку общения и обладающих значительно большими и универсальными возможностями, появлялись всё новые наработки, упрощающие процесс создания и последующего сопровождения прикладных программ. К основным технологическим вехам можно отнести: появление именованных переменных, структурных операторов, подпрограмм, технологию модульного программирования, технологию объектно-ориентированного программирования, динамические библиотеки, событийное программирование, визуальное проектирование приложений, использование фреймворков и платформ исполнения программного кода.

Все перечисленные технологии в настоящее время используются в большинстве самых популярных языков программирования и, в той или иной степени, обеспечивают прикладное программирование (application programming). Определим прикладное программирование как как процесс проектирования, разработки и отладки программных приложений определённой прикладной направленности, то есть ориентированных на выполнение определённых специфических задач,

в том числе, для автоматизации рутинных процессов пользователей в их повседневной работе за персональным компьютером.

Можно выделить, как минимум, две основные особенности приложений подобного рода:

– при разработке программы для непрофессионального пользователя следует уделять больше внимания проработке интерфейса пользователя, иногда даже в ущерб производительности и компактности приложения;

– нельзя выделить и ограничить некоторое подмножество необходимых технологий, компонентов для реализации прикладных программ из-за разной их направленности в зависимости от профессиональной принадлежности заказчика или возможного потребителя приложения.

Описанные обстоятельства предопределили основную линию данного учебного пособия, состоящую в большом внимании к элементам взаимодействия пользователя с программой и данными. По ходу изучения материала, вы будете встречать описание новых и разнообразных по функционалу компонентов, обеспечивающих дополнительные возможности интерфейса пользователя. Для развития пользовательских функций мы не будет ограничиваться только визуальными (отображаемыми) компонентами, но также будем пользоваться таймером и его событиями, обработкой событий мыши и клавиатуры, обработкой исключительных ситуаций и специализированными настройками компонентов с помощью инспектора объектов.

Глава 1. Модульное программирование

Приложение на этапе разработки состоит из нескольких файлов (модулей, ресурсов, описаний визуальных форм, основной программы), которые представляют собой проект. После компиляции проекта будет создана сборка в один файл с расширением *.exe. Для удобства работы с файлами проекта для каждого нового приложения следует создавать отдельный каталог. С помощью указанных на рис. 1 пунктов меню создайте новое приложение в среде Lazarus:

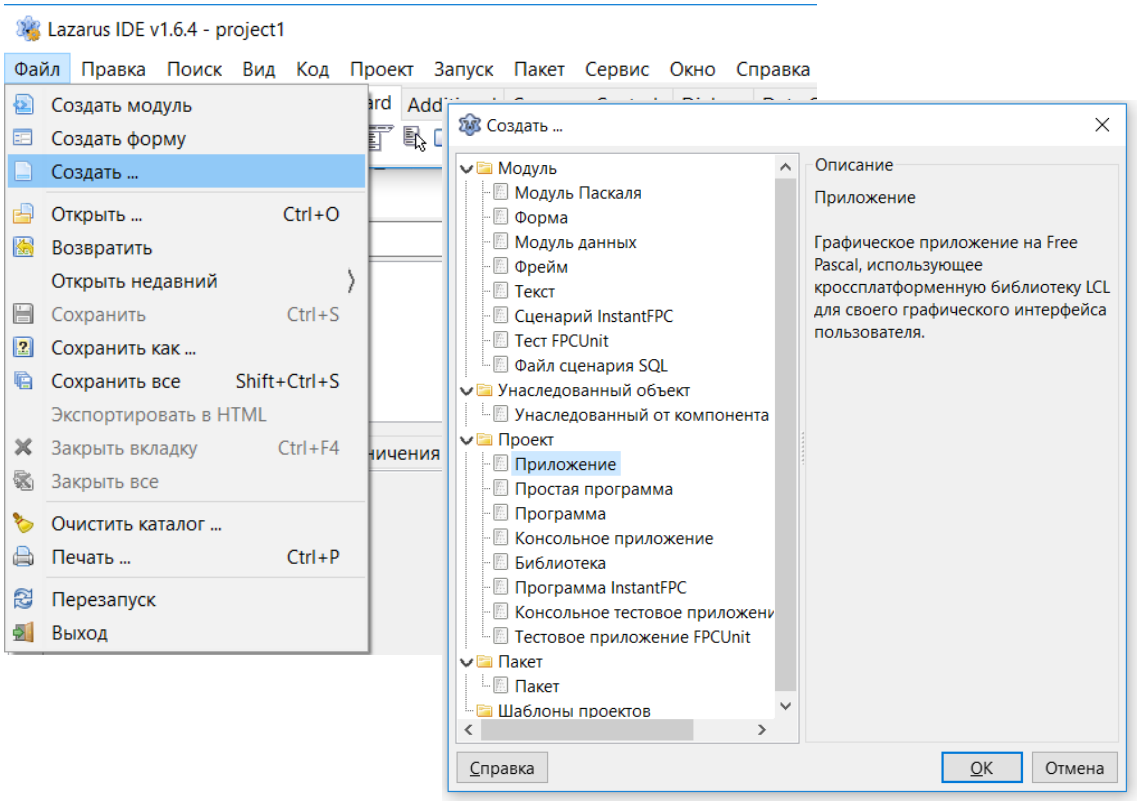

Рис.1. Окно выбора типа приложения.

Перейдите в окно кода программы, нажав клавишу F12 или кликнув по окну кода мышкой. Вы увидите текст модуля:

```
unit Unit1;
```

```
{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Di-
alogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

end.
```

Данное приложение уже работоспособно, его можно запустить на выполнение, нажав F9 или экранную клавишу пуск. Его (окно приложения) можно подвигать, изменить размеры, минимизировать, максимизировать, закрыть.

Для того чтобы правильно сохранить разрабатываемое вами приложение, включая все модули и формы, следует выбрать пункт меню Файл / Сохранить всё (Shift+Ctrl+S) и последовательно сохранить все модули приложения и сам проект в одну папку.

Обратим внимание на раздел подключения модулей. Ключевое слово Uses определяет начало списка модулей, которые используются текущим модулем, программой или библиотекой. Следует заметить, что при создании нового приложения Lazarus самостоятельно размещает несколько модулей в раз-

деле подключения модулей uses. Для такого простейшего приложения, которое вы только что запустили это количество избыточно. Попробуйте из раздела подключения модулей uses удалить все модули за исключением Forms и снова запустите программу.

Подключаемые модули, разрабатываемые программистом, имеют расширение ppu и представляют собой откомпилированный модуль проекта в промежуточном формате. Когда компилируется программа, все модули компилируются в файлы формата ppu, а потом собираются в один исполняемый exe файл. Если модуль не изменялся с последней компиляции, то Lazarus пропустит его, а во время сборки будет использовать существующий файл формата ppu, чтобы сократить время компиляции. Загляните в папку, где вы сохранили свой проект (в директорию \lib\x86_64-win64), и вы обнаружите файл, совпадающий по имени с модулем вашего проекта unit1, но с расширением ppu. Проведите эксперимент – запустите программу (F9) без внесения изменений в ваш модуль и убедитесь, что время создания файла unit1.ppu не изменилось, затем закройте запущенную программу и подвиньте немного мышкой форму Form1, после чего вновь запустите программу – убедитесь, что теперь был создан новый файл unit1.ppu.

При необходимости расширить функциональные возможности приложения, нужно подключать в разделе uses дополнительные модули, однако не обязательно их прописывать вручную. При добавлении на форму визуального компонента из библиотеки компонентов редактор Lazarus самостоятельно добавляет необходимые модули в раздел uses. Например, при добавлении такого компонента как стандартная экранная клавиша Button, в раздел uses добавляется модуль StdCtrls. Вы можете самостоятельно провести такой эксперимент. Добавить компонент на форму в среде визуального программирования

просто: кликните на соответствующую иконку компонента – она станет активной, затем кликните в том месте формы, где хотите компонент разместить, он там и окажется.

Начнем с того, что добавим на форму кнопку Button1, иконка этого компонента располагается на вкладке Standart (рис. 2).

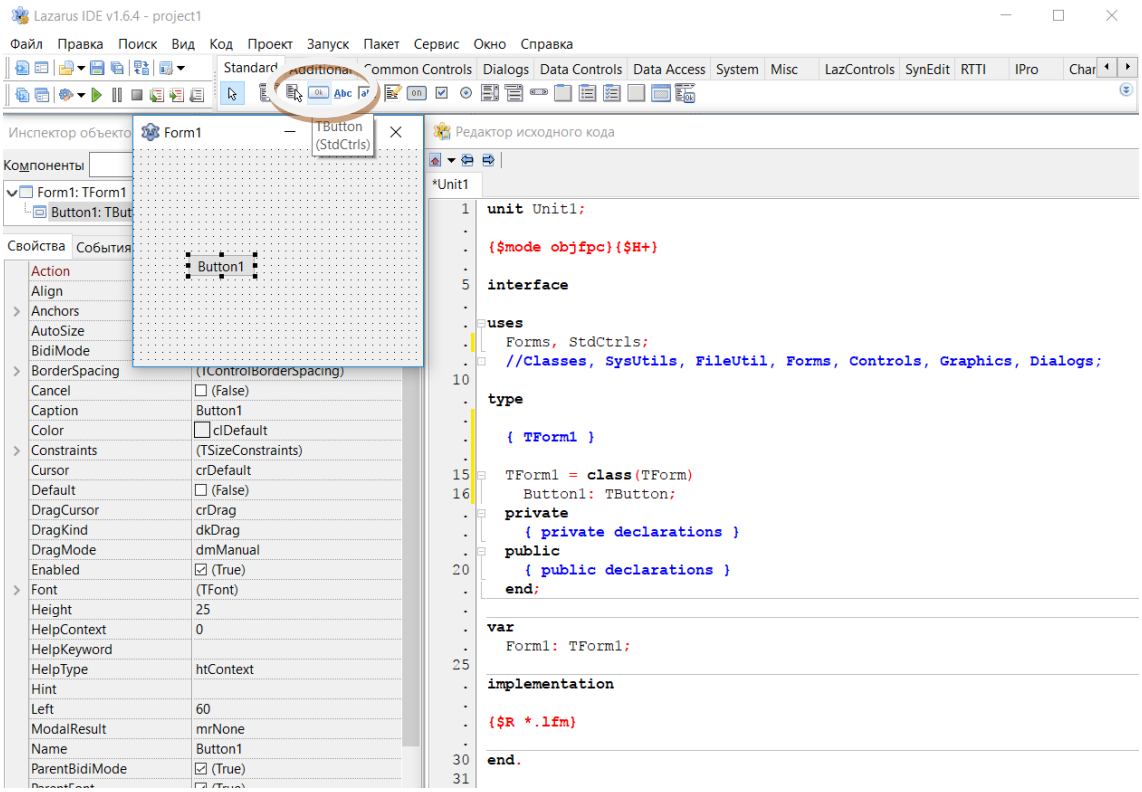

Рис.2. Панель стандартных компонентов.

Запустите программу на выполнение – она работает, кнопка нажимается, но пока не выполняет никаких действий, так как мы и не задали никаких обработчиков событий. Однако все эти новые возможности стали доступны благодаря тому, что Lazarus самостоятельно подключил необходимые модули из списка стандартных – обратите внимание вновь на раздел uses. Но, не редкость, возникновение таких ситуаций, когда программисту приходится самостоятельно прописывать подключаемый модуль, что характерно для модулей, не обслу-

живающих работу визуальных компонентов. К таким, например, относятся модули Math (содержит математические функции) или SysUtils (делает доступными множество подпрограмм манипулирования данными, таких как IntToStr).

Убедимся в этом при проведении простого эксперимента.

Остановите работу (закройте) вашей программы, если еще этого не сделали. Добавьте на форму ещё один компонент – однострочное редактируемое текстовое поле – Edit1 (рис. 3).

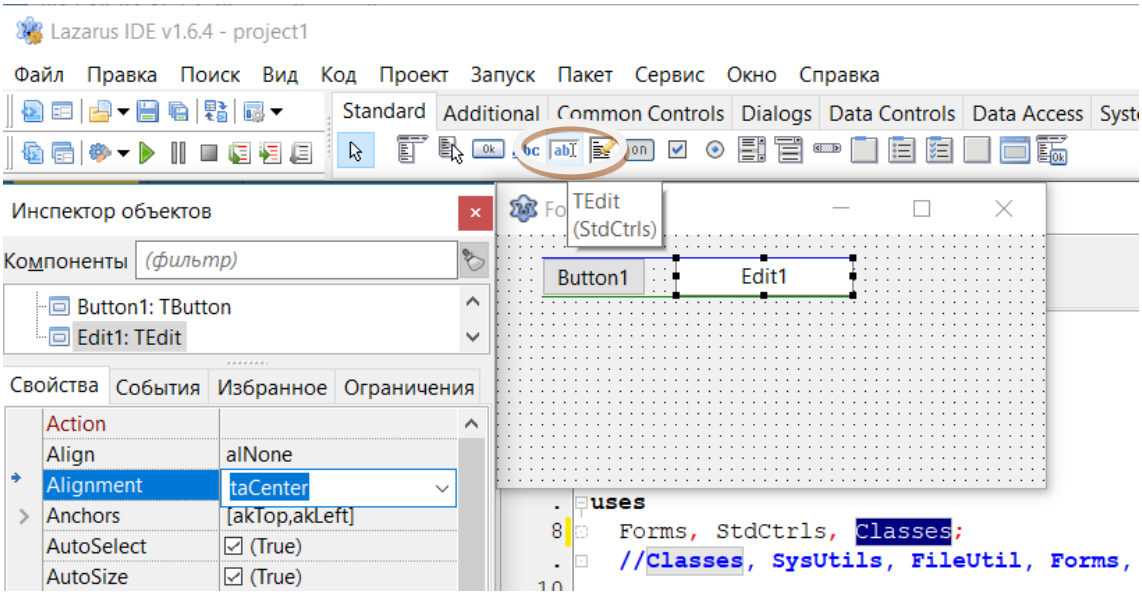

Рис.3. Выбор и настройка компонента.

Обратите внимание, что в среде визуального проектирования приложений сведены к минимуму действия программиста по ручной настройке приложения – автоматически добавляются не только необходимые для работы модули, но и сама настройка компонентов может производиться с помощью визуальных средств. Попробуйте поэкспериментировать с настройкой поля Edit1 через вкладку «Свойства» в окне Инспектора объектов, например, установите размер шрифта, настройте цвет фона или выравнивание текста. Кроме того, среда визуально проектирования приложений помогает писать программный код, автоматически создавая шаблоны процедур

или выдавая контекстные подсказки о правильности написания идентификаторов переменных или названий подпрограмм в зависимости от допустимости их использования в месте расположения курсора.

Кликните два раза на кнопке Button1, на что Lazarus отзовется созданием процедуры обработчика события – «клик по кнопке», при этом автоматически добавив (вернув на место) модуль Classes в раздел uses:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Начнём набирать программный код:

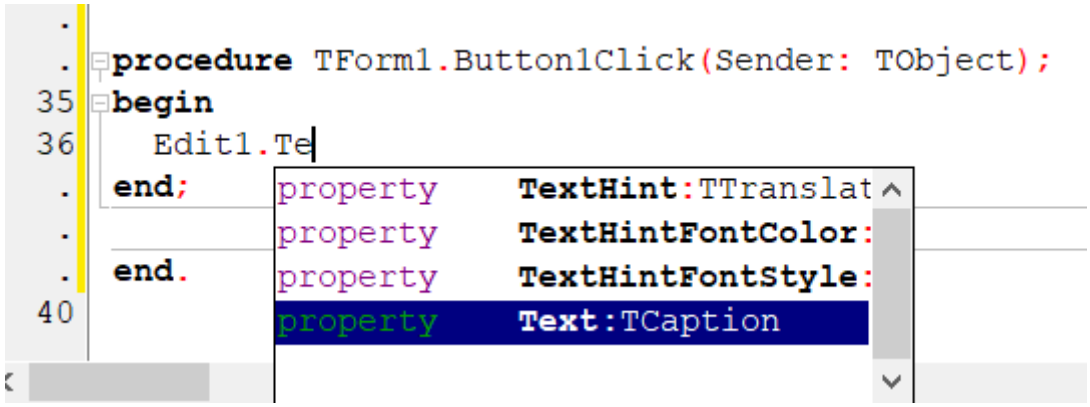

Рис.4. Контекстная подсказка.

Обратите внимание, что по мере набора кода Редактор предлагает подходящие в данной ситуации свойства или методы, которые можно не набирать полностью, а ввести нажатием клавиши Enter. Контекстную подсказку можно вывести на экран также сочетанием клавиш Ctrl+Пробел.

Заполним полностью кодом обработчик Button1Click:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text:=5;
end;
```

Предполагая, что по нажатию клавиши Button1 в текстовое поле будет выведена цифра 5. Сделаем попытку запуска нашей программы на выполнение (F9 – компиляция и запуск или Ctrl+F9 – только компиляция без запуска программы на исполнение, что удобно использовать для проверки корректности синтаксиса кода).

Попытка запуска закончится неуспешно – редактор сигнализирует про несовместимые типы строковый и целочисленный, а также про невозможность откомпилировать модуль (вы сейчас пишете код программы в модуле unit1). Действительно, текстовое поле оно потому текстовое, что предназначено для отображения строк. Внесем следующие изменения – сделаем обрамление из одинарных кавычек ‘5’ (так в языке программирования Pascal обозначают строки):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text:='5';
end;
```

и запустим программу. Работает. Однако, если в программе задумана какая-то обработка данных, то хранить их, видимо, придется все-таки в более подходящем – нетекстовом виде. Значит, в момент вывода результата вычислений в текстовое поле, следует полученное значение преобразовать из текстового в числовой формат. Для этих целей подойдут функции

модуля SysUtils – IntToStr (для целых чисел) и FloatToStr (для вещественных чисел).

Как узнать о формате подпрограммы и посмотреть примеры ее использования? Наберите в окне кода модуля Unit1 в любом месте такой текст: IntToStr, оставьте курсор где-то в пределах IntToStr и нажмите F1 (таким способом можно узнать много полезной информации и про другие функции). Откроется справка:

```
IntToStr
 Convert an integer value to a decimal string.
Declaration
function IntToStr(Value: LongInt):string;
function IntToStr(Value: Int64):string;
function IntToStr(Value: QWord):string;
```

из которой станет ясно, что аргументом этой функции является переменная целого типа, а результатом строковая переменная. Кроме того, наличие трёх описаний (деклараций) функции указывает, что эта она относится к перегружаемым подпрограммам. Наличие механизма перегрузки подпрограмм позволяет реализовывать подпрограммы с разной сигнатурой, то есть выполняющие одинаковые действия на основе параметров *разных типов* или *различного их количества*. Это упрощает работу программиста, унифицирует программный код, отменяет необходимость приведения параметров к конкретным типам данных. Для того чтобы компилятор мог выбрать правильную подпрограмму из нескольких перегруженных, они должны отличаться хоть чем-то в сигнатуре метода, например, последовательностью типов данных в списке параметров, их количеством или самим типом данных. В данном случае, в качестве параметра функции IntToStr может использоваться данное типа Integer, Int64 или QWord (первые два знаковые, последнее целое без знака, то есть только положительное, о числовых типах данных см. Приложение II).

Для приведения программы к корректному виду внесем соответствующие изменения:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.Text:=IntToStr(5);
end;
```

Запустим программу на выполнение … и опять не получим желаемого результата – оказывается идентификатор IntToStr не задекларирован. В чем же проблема? Вернемся к тому с чего начинали – Lazarus не всегда самостоятельно подключает необходимые стандартные модули. В данном случае не было добавления на форму визуального компонента и связанного с этим автоматического добавления необходимого модуля/модулей в список подключаемых модулей в раздел uses, поэтому чтобы добиться положительного результата следует самостоятельно добавить в раздел uses модуль SysUtils. Самостоятельно впишите необходимый модуль в список подключённых и заново проверьте работоспособность программы.

Чтобы в дальнейшем не испытывать затруднений при усложнении программы давайте вернем все заявленные по умолчанию модули в разделе uses на место:

```
uses
    Forms, StdCtrls, Classes, SysUtils,
    FileUtil, Controls, Graphics, Dialogs;
```

Если в дальнейшем нам нужно будет создавать многооконное приложение, наподобие самой среды Lazarus, то нужно уметь передавать значения между формами приложе-

ния. Обратите внимание, что для каждой формы автоматически создаётся свой модуль, совпадающий по идентификатору с именем формы. Таким образом, для того чтобы пользоваться возможностями новой/другой формы, необходимо соответствующий ей модуль подключить в разделе uses.

Давайте попробуем создать первое многооконное приложение, для чего через главное меню выберите пункт «Файл» / «Создать форму». Сохраните её в ту же папку, что и основное приложение, пока, не меняя имя, данное по умолчанию (Unit2.pas). На вторую форму поместите компонент Edit1, он будет иметь такое же имя, как и текстовое поле с первой формы. Для того чтобы выбрать место куда именно будет записываться результат, достаточно только указать предварительно имя контейнера, содержащего тот или иной компонент, например, так (**`Form2.Edit1.Text`**):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Edit1.Text:=IntToStr(5);
end;
```

Однако попытка запуска такой программы не закончится успешно, так как модуль первой формы не знает о существовании второй формы. Чтобы преодолеть данное затруднение достаточно просто в разделе подключения модулей указать соответствующий идентификатор модуля (Unit2):

```
uses
  Forms, StdCtrls, Classes, SysUtils,
  FileUtil, Controls, Graphics, Dialogs, Unit2;
```

После внесения изменений программа должна запуститься, но при нажатии на экранную клавишу Button1 вы не увидите желаемого результата. Причина этого кроется в том, что никто не указал, что должна отображаться сама форма Form2. Окончательно оформленный код выглядит следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.Edit1.Text:=IntToStr(5);
  Form2.Show;
end;
```

Следует отметить, что модули могут создаваться не только совместно с соответствующими экранными формами, но и отдельно. Как правило, в таких модулях располагают библиотеку подпрограмм, для выполнения часто использующихся задач. К таким задачам можно отнести: конвертацию данных, математические методы обработки данных, обработку массивов данных, вычислительные задачи и т.п.

Для уточнения порядка разработки модуля и подключения подпрограмм разработаем небольшой модуль, решающий задачу перевода двоичного числа в десятичное двумя видами подпрограмм – процедурой и функцией, а также апробируем технологию создания перегружаемых подпрограмм.

Создайте новое приложение в среде Lazarus и сохраните его в отдельной папке. Подготовьте интерфейс приложения, добавив два поля Edit (для ввода/вывода информации) и три кнопки Button для запуска подпрограмм, отличающихся сигнатурами (рис. 5).

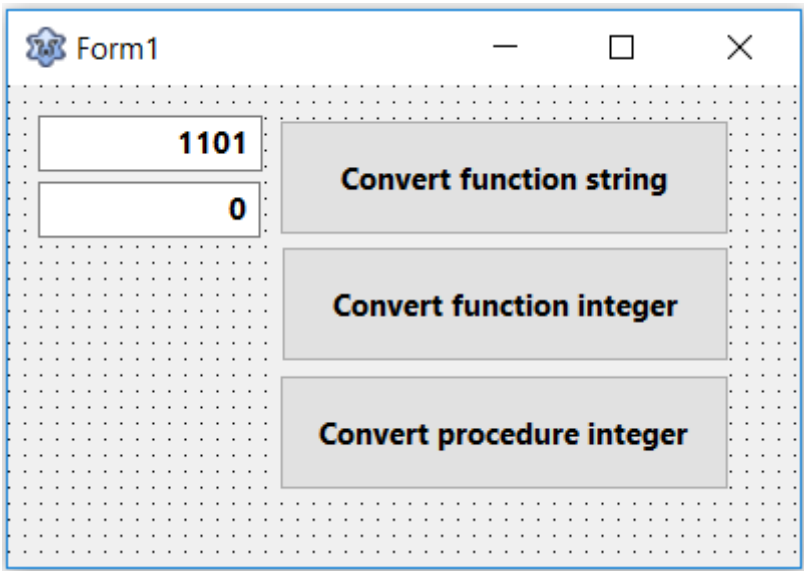

Рис.5. Внешний вид программы для перевода чисел.

Наименование у всех подпрограмм будет одинаковое – binToDec, но количество аргументов, их типы данных и способ возвращения результата в основную программу будет отличаться. Первая подпрограмма – функция, которая принимает строковый аргумент и возвращает ответ в виде строки, вторая подпрограмма тоже функция, но работающая с целыми числами и, наконец, третья подпрограмма – процедура, которая работает с целыми числами и возвращает ответ не через своё имя, а через дополнительный аргумент. Названия кнопок на форме соответствуют логике работы подпрограмм.

Все подпрограммы разместим в специальном модуле Utils (название можно дать любое, осмысленное), для чего в главном меню Lazarus выберите пункт «Файл» / «Создать модуль». Сгенерированный модуль имеет стандартную для модулей структуру: заголовок (имя даётся автоматически по умолчанию), интерфейсная часть (где декларируются подпрограммы, видимые вне данного модуля внешней программой), раздел подключения модулей и часть реализаций (где описывается содержательная часть подпрограмм):

```
unit Unit2;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

implementation

end.
```

Прежде всего, сохраните модуль в ту же папку, где хранятся файлы приложения и, при сохранении смените имя на Utils. В первом модуле Unit1, который работает с формой, подключите модуль Utils в разделе подключения модулей:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls,
  Graphics, Dialogs, StdCtrls, Utils;
```

Теперь можно приступать к формированию содержания модуля, для чего следует наполнить раздел реализаций кодом подпрограмм и необходимые подпрограммы вынести в интерфейсную часть. Зачастую бывает удобно использовать одно имя для разных подпрограмм, работающих с разными типами данных. Например, программист может подавать на вход функции как строковое представление двоичного числа, так и целочисленное, но программа может сама справится с данной проблемой, подобрав подходящую по сигнатуре функцию,

если есть варианты подпрограмм, соответствующим образом оформленные:

```
unit Utils;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

function binToDec(bin: integer): integer; overload;
function binToDec(bin: string): string; overload;
procedure binToDec(bin: integer; var dec: integer);

implementation

procedure binToDec(bin: integer; var dec: integer);
begin
  dec:=binToDec(bin);
end;

function binToDec(bin: integer): integer; overload;
var dec, step, tmp, last: integer;
begin
  dec:=0; // десятичное число
  step:=1; // значение 2 в степени
  tmp:=bin; // для хранения убывающего числа
  while tmp>0 do
  begin
    last:=tmp mod 10; // последняя цифра в числе
    tmp:=tmp div 10; // число без последней цифры
    dec:=dec+last*step; // цифра * на 2 в степени pos
    step := step * 2; // значение 2 в степени для следующего
разряда
  end;
  result:=dec;
end;

function binToDec(bin: string): string; overload;
var i, dec, step: integer;
begin
  dec:=0; // десятичное число
  step:=1; // значение 2 в степени
  for i:=Length(bin) downto 1 do
  begin
    dec:=dec+StrToInt(bin[i])*step;
    step := step shl 1;
  end;
  result:=IntToStr(dec);
end;

end.
```

Обратите внимание, что модификатор вызова overload не является обязательным, а оставлен в Lazarus для совместимости с Lazarus.

Итак, у нас есть три подпрограммы, выполняющие одинаковую задачу перевода числа из двоичной системы счисления в десятичную, но разными способами. Вы, конечно, заметили, что в коде процедуры

```
procedure binToDec(bin: integer; var dec: integer);
```

идёт вызов функции

```
function binToDec(bin: integer): integer; overload;
```

из самого же модуля и не содержится самого алгоритма перевода числа. Это самый обычный приём, имеющий логическое обоснование. И, действительно, зачем писать код решения задачи повторно, когда он уже есть в соседней подпрограмме. Но внешний пользователь данного модуля этого не знает, однако имеет возможность пользоваться тем видом подпрограммы, которым ему удобнее в его программе. Основываясь на полученных знаниях, попробуйте самостоятельно сократить текст модуля, упростив функцию:

```
function binToDec(bin: string): string; overload;
```

Содержание этой функции нужно вычеркнуть и заменить ссылкой на функцию, работающую с целыми числами по аналогии с вышеописанным.

Осталось только корректно, с учётом типов данных, подключить подпрограммы модуля к соответствующим экранным клавишам на форме приложения, заполнив содержимое первого модуля:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   Edit2.Text:=binToDec(Edit1.Text);
end;

procedure TForm1.Button2Click(Sender: TObject);
```

```
var bin, dec: integer;
begin
   bin:=StrToInt(Edit1.Text);
   dec:=binToDec(bin);
   Edit2.Text:=IntToStr(dec);
end;

procedure TForm1.Button3Click(Sender: TObject);
var bin, dec: integer;
begin
   bin:=StrToInt(Edit1.Text);
   binToDec(bin, dec);
   Edit2.Text:=IntToStr(dec);
end;
```

Апробируйте программу, добейтесь её работоспособности и опишите для себя разницу в использовании процедур и функций.

Сохраните текущие изменения в программе нажатием клавиш Ctrl+S, в дальнейшем не забывайте периодически производить сохранения во избежание случайных потерь кода. После первичного ознакомления с технологией визуального проектирования приложений и модульного программирования в среде Lazarus можно переходить к технологии событийного программирования.

Глава 2. Событийное программирование

Ещё одна технология, которая значительно облегчает труд программиста – событийное программирование. Основной принцип в событийном программировании заключается в разбиении всей программы на отдельные, имеющие конкретное предназначение подпрограммы, которые запускаются на исполнение только при наступлении определённого, заранее заданного события. События могут быть как реальными (нажатие сочетания клавиш, двойной клик мышкой), так и виртуальными (изменение размеров экранной формы, запрос на закрытие программы). Не во всех случаях имеет смысл строить программу, основанную на обработке событий. Тем не менее, есть ряд направлений, где событийное программирование особенно востребовано:

– написание игровых программ со значительным количеством управляемых объектов;

– разработка прикладных программ с пользовательским интерфейсом;

– создание серверных приложений с ограничением по генерации обслуживающих процессов;

Разберём подробнее наиболее актуальные события, встречающиеся в интерфейсах прикладных программ.

2.1. Обработка событий мыши.

Lazarus позволяет обрабатывать различные события манипулятора «мышь»: одиночный клик левой, средней или правой клавишей, дополнительные боковые клавиши мыши, двойной клик левой клавишей мыши; движение указателя мыши, про-

крутка колёсика мыши и т.п. Наличие обработчиков и их возможности зависят от компонентов (форма, кнопка, текстовое поле и др.), к которым привязаны обработчики событий.

Рассмотрим некоторые из событий мыши и попрактикуемся настраивать соответствующие обработчики на разрабатываемом нами приложении.

Создадим обработчик события – движение указателя по ***второй*** форме приложения. Пусть координаты указателя мыши во время движения отображаются в текстовом поле Edit1 (принадлежащего Form2).

Чтобы перейти к нужной форме нажмите сочетание клавиш Shift+F12 (к нужному модулю – Ctrl+F12), выберите Form2, выделите её кликом мыши один раз и в инспекторе объектов (Object Inspector) станут доступными для редактирования свойства (Properties) и события (Events), доступные для выделенного объекта (сейчас это Form2). В данный момент нас интересуют события мыши. Найдите в списке событий в левом столбце событие OnMouseMove и кликните два раза по правому (пустому) полю. Ваше действие приведет к декларации процедуры обработчика события движение мыши по форме (рис.6) и генерации шаблона кода процедуры:

```
procedure    TForm2.FormMouseMove(Sender:    TObject;    Shift:
TShiftState; X, Y: Integer);
begin

end;
```

Обратите внимание, что среди параметров процедуры есть переменные X и Y – именно они содержат текущие координаты указателя мыши в любой момент времени. Их можно выводить на экран в момент движения мыши в пределах границ формы, для чего добавьте в процедуру код:

```
procedure    TForm2.FormMouseMove(Sender:    TObject;    Shift:
TShiftState; X, Y: Integer);
begin
    Edit1.Text:=IntToStr(X)+':'+IntToStr(Y);
end;
```

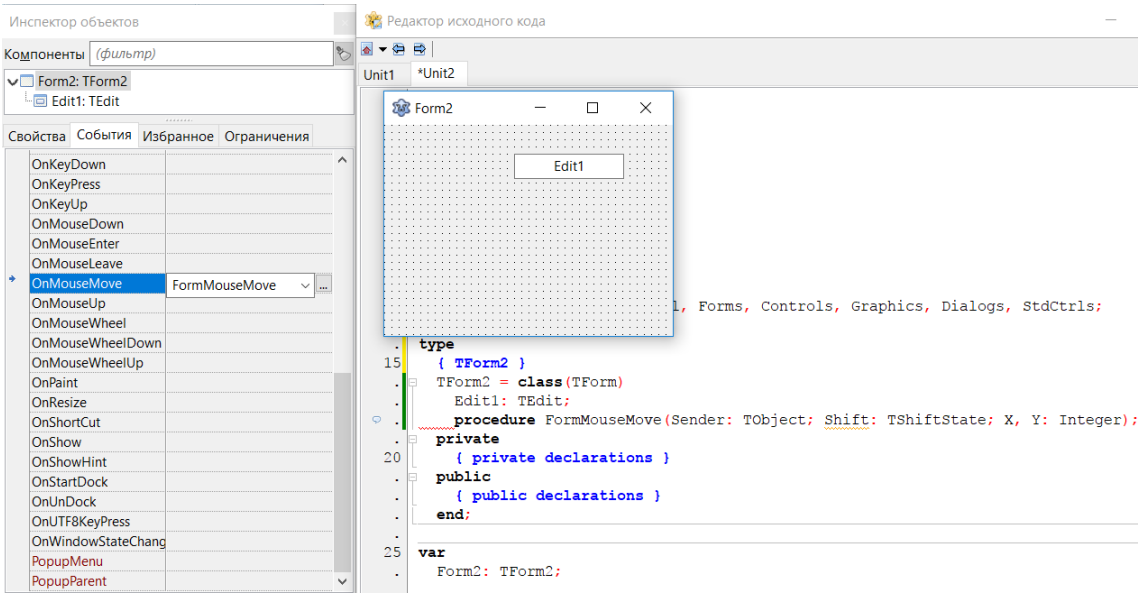

Рис.6. Автоматическая декларация процедуры обработчика события.

Теперь создадим процедуру, которая будет нам сообщать какой клавишей мыши кликнул пользователь по форме. Разместите на форме компонент Label с вкладки Standard и настройте его свойства по своему усмотрению. Сгенерируйте для формы Form2 обработчик события OnMouseDown. Обратите внимание на переменную Button – она содержит наименование нажатой клавиши мыши и имеет тип TMouseButton. Как же узнать какие именно значения она может приобретать? Для этого нажмите и удерживайте клавишу Ctrl и наведите указатель мыши на описание типа в процедуре (TMouseButton), которое приобретет вид гиперссылки. Клик-

ните на ней и Lazarus откроет вам программный код с описанием модуля Controls, в котором вы и увидите перечисление значений типа в круглых скобках:

```
TMouseButton = (mbLeft, mbRight, mbMiddle, mbExtra1, mbExtra2);
```

Закройте модуль Controls, кликнув правой клавишей мыши по его заголовку и выбрав опцию Закрыть вкладку или просто кликнув средней клавишей мыши.

Заполним процедуру программным кодом, который будет обеспечивать вывод наименования нажатой клавиши в метку Label1:

```
procedure   TForm2.FormMouseDown(Sender:   TObject;   Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
    with Label1 do
    case Button of
      mbLeft:   Caption:='левая';
      mbMiddle: Caption:='средняя';
      mbRight:  Caption:='правая';
      mbExtra1: Caption:='дополнительная вперёд';
      mbExtra2: Caption:='дополнительная назад';
    end;
end;
```

Оператор объединения with в процедуре позволяет избежать излишнего дублирования программного кода и не писать каждый раз Label1.Caption.

Событие OnMouseDown возникает в момент нажатия клавиши мыши, а событие OnMouseUp в момент её отпускания. Сгенерируйте процедуру FormMouseUp и заполните кодом:

```
procedure TForm2.FormMouseUp(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
begin
  Label1.Caption:=Label1.Caption+' отпущена';
end;
```

Испытайте её следующим образом: кликните мышкой по форме и удерживайте её нажатой, оцените содержимое поля Label1, затем отпустите клавишу мыши и посмотрите на изменения. Доработайте код процедуры, используя оператор with.

Теперь поработаем немного с колесиком мыши. Пусть от вращения колёсика зависит положение метки Label1 по вертикали. Сгенерируйте процедуры FormMouseWheelDown и FormMouseWheelUp и заполните их кодом:

```
procedure  TForm2.FormMouseWheelDown(Sender:  TObject;  Shift:
TShiftState; MousePos: TPoint; var Handled: Boolean);
begin
  if Label1.Top<Form2.Height then Label1.Top:=Label1.Top+1;
end;

procedure  TForm2.FormMouseWheelUp(Sender:  TObject;  Shift:
TShiftState; MousePos: TPoint; var Handled: Boolean);
begin
  if Label1.Top>0 then Label1.Top:=Label1.Top-1;
end;
```

Доработайте процедуры, используя оператор with.

Оцените возможности перемещения метки Label в пределах формы. Вы обнаружите, что при движении вверх метка упирается в верхнюю границу формы, а при движении вниз – пересекает её и уходит за пределы формы. Постарайтесь оценить, в чём некорректность написанного ранее кода и самостоятельно исправьте ошибку.

Обратите внимание, что процедуры содержат переменную Shift. Она содержит комбинацию нажатых управляющих клавиш (Ctrl, Alt, Shift) в момент вращения колёсиком мыши (и других событий мыши). Посмотрите самостоятельно описание типа TShiftState и определите, какие значения может принимать переменная такого типа. Использование управляющих клавиш позволяет разнообразить возможности разрабатывае-

мой процедуры. Пусть вращение колесика мыши будет двигать метку Label1 слева-направо, если при этом удерживается в нажатом состоянии клавиша Ctrl; сверху-вниз при удержании клавиши Alt; по диагонали – при удержании сразу обеих клавиш.

```
procedure  TForm2.FormMouseWheelDown(Sender:  TObject;  Shift:
TShiftState; MousePos: TPoint; var Handled: Boolean);
begin
  with Label1 do
  begin
    if Shift = [ssCtrl] then
      if Left<Form2.Width then Left:=Left+1;
    if Shift = [ssAlt] then
      if Top<Form2.Height then Top:=Top+1;
    if Shift = [ssCtrl, ssAlt] then
    begin
      if Left<Form2.Width then Left:=Left+1;
      if Top<Form2.Height then Top:=Top+1;
    end;
  end;
end;
```

Аналогичным образом самостоятельно заполните процедуру FormMouseWheelUp.

Для закрепления полученных навыков по событийному и визуальному программированию спроектируем и реализуем первое осмысленное приложение – калькулятор.

Процесс проектирования программного продукта будет включать такие этапы как:

– разработка дизайна (интерфейса пользователя) приложения,

– разработка алгоритма работы программы,

– разработка процедур обработчиков событий интерфейса.

Создадим программу – простой калькулятор с основными арифметическими действиями: «+», «–», «*», «/»; с наличием кнопки «Сброс», «=» и возможностью оперировать с числами вещественного типа, то есть с числами с запятой (рис. 7).

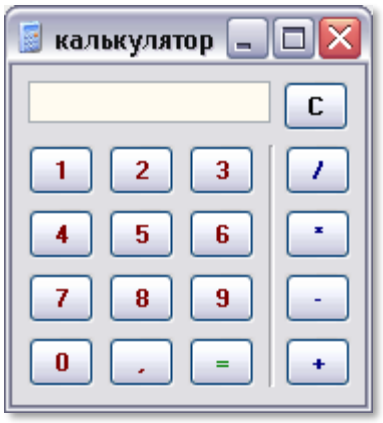

Рис.7. Пример дизайна программы «Калькулятор».

Создайте ***новое*** приложение в среде Lazarus, сохраните его в отдельной папке и последовательно выполните все этапы проектирования программного продукта. Обратите внимание, что ***заключительный этап вам предстоит выполнить самостоятельно***.

Этап 1. Разработка интерфейса.

Добавьте на форму необходимые для работы программы компоненты (рис. 8):

– Edit – текстовое поле для организации ввода чисел и вывода результата арифметического действия;

– BitBtn – кнопка из вкладки Additional, в отличие от кнопки Button с вкладки Standart, у BitBtn настраивается цвет надписи – нам это пригодится для цветового выделения кнопок (группа цифр и запятая, группа действий, сброс и равно), а также имеется возможность размещения картинки на поверхности кнопки. Таких кнопок потребуется несколько, в зависимости от функциональной наполненности калькулятора.

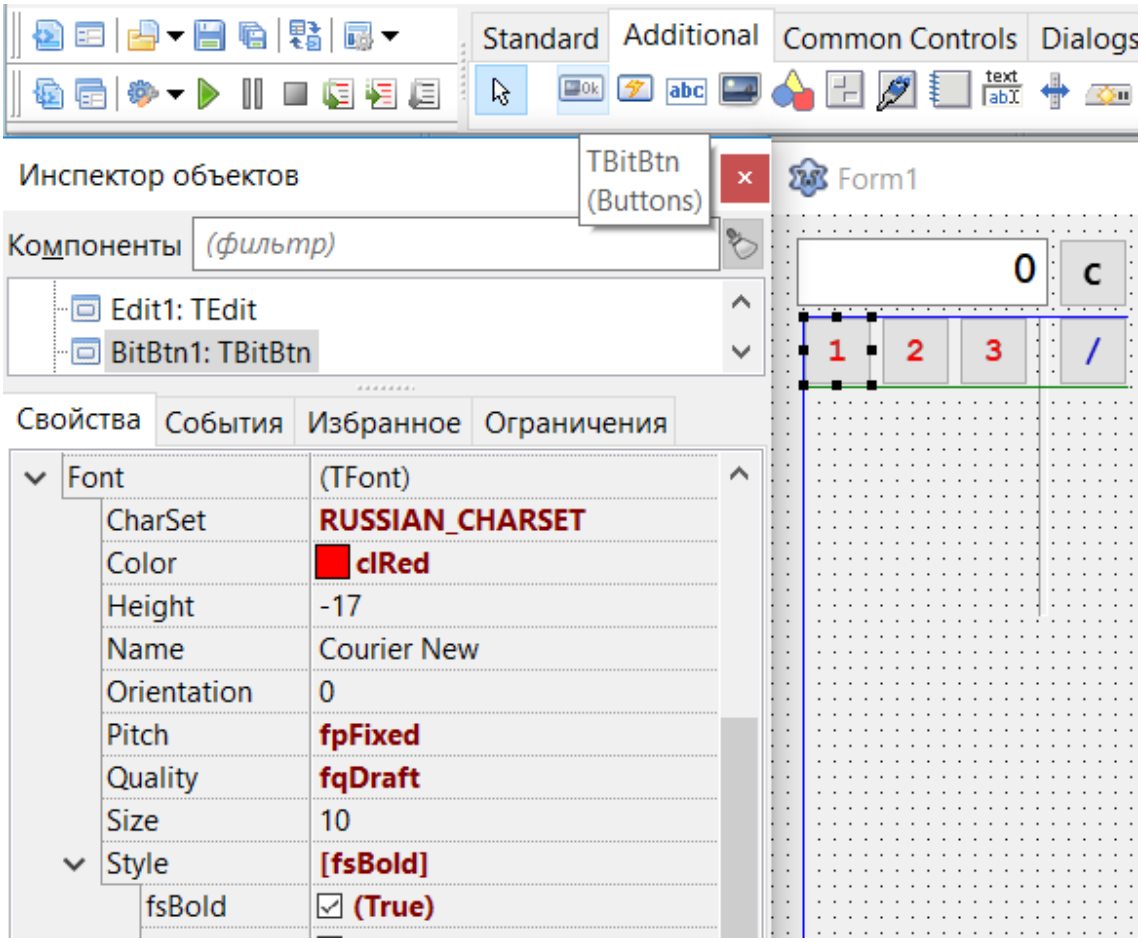

Рис. 8. Настройка визуальных компонентов программы.

В инспекторе объектов измените свойства формы с тех значений, которые были установлены по умолчанию на следующие:

Caption: калькулятор

Color: цвет по желанию

BorderIcons.biMaximize: False;

Для компонента Edit1 в инспекторе объектов задайте следующие свойства:

Text: 0;

Alignment: taRightJustify;

Color: цвет по желанию.

Далее разместите на форме необходимое число кнопок:

– цифры от 0 до 9,

– кнопка «,»,

– знаки арифметических операций,

– знак «=»,

– кнопка Сброс.

В инспекторе объектов задайте их свойства:

Name: пока можно оставить по умолчанию, но для последующего усовершенствования программы лучше сразу дать осмысленные имена, например, btn0 (для кнопки Ноль), btnDiv (для кнопки Деление), btnC (для кнопки Сброс) и т.п.,

Caption: соответствующее выполняемой функции название (цифра, запятая, знак действия),

Font: шрифт и цвет шрифта по желанию.

Этап 2. Разработка алгоритма работы калькулятора.

В этой части не нужно непосредственно писать текст программного кода. Сейчас мы обсуждаем структуру будущей программы.

Наиболее очевидным можно считать такой алгоритм работы калькулятора:

1) вводим первое число в поле Edit1;
2) нажимаем кнопку действия («+»,«–»,«*»,«/»), что приводит к сохранению в соответствующих глобальных переменных как первого числа из поля Edit1 так и наименования действия из Caption только что нажатой экранной клавиши действия для последующего использования;
3) затем вводим второе число в поле Edit1;
4) наконец нажимаем кнопку «=» – при этом результат выводим в поле Edit1;

#2. Задание для самостоятельного исполнения

Этап 3. Разработка процедур обработчиков событий.

Обработчики событий, естественно, зависят от задуманного алгоритма работы программы.

А. Напишите процедуру добавления цифры (0..9) к текстовому полю Edit1. В самом примитивном варианте ***следует создать обработчик события «клик на клавише» для каждой клавиши***. В процедуру обработчика события следует добавить одну строку, например, для клавиши «1» – Edit1.Text:=Edit1.Text+'1';

Аналогично напишите процедуры ввода остальных цифр и символа «,». В дальнейшем мы рассмотрим методы, позволяющие существенно сократить дублирование полученного сейчас программного кода.

Б. Создайте процедуру очистки поля Edit1 кнопкой «C», которая приводит к обнулению отображаемого значения.

В. Создайте процедуры обработки нажатий кнопок «+», «–», «/», «*».

Процедура обработчика события кнопки действия («+»,«–»,«*»,«/») не предполагает собственно выполнения арифметического действия, ведь ещё не введён второй аргумент, достаточно реализовать следующую последовательность:

– преобразовать строку из свойства Text поля Edit1 в число с использованием функции StrToFloat;
– сохранить результат в глобальную (если переменная будет локальной, то она не будет доступна в других процедурах) переменную X типа Real;
– обулить поле Edit1 для ввода второго аргумента;
– зафиксировать в глобальной переменной N (тип можете выбрать сами, char, byte или integer) наименование выполняемого арифметического действия («+»,«–»,«*»,«/»).

Следует уточнить, что для того чтобы ввести глобальную переменную её следует объявлять в разделе объявления глобальных для данного модуля переменных, то есть там же где и объявлена переменная формы (непосредственно перед разделом реализаций implementation), например, так:

```
var
  Form1: TForm1;
  znak: char;
```

Г. Создайте процедуру обработки нажатия кнопки «=».

В процедуре обработки этой кнопки необходимо выполнить следующие действия:

– преобразовать строку из свойства Text поля Edit1 в число с использованием функции StrToFloat,
– сохранить это число в локальной переменной Y;
– в зависимости от наименования операции, выполнить арифметическое действие с переменными X и Y и сохранить в переменную Result;
– результат преобразовать в строку с помощью функции FloatToStr и вывести в поле Edit1.

В качестве ***дополнительного задания для самоконтроля*** попробуйте доработать калькулятор таким образом, чтобы Сброс поля ввода производился при нажатии сочетания клавиш `Ctrl+«средняя клавиша мыши»` при фокусе ввода на самом поле Edit1, то есть без нажатия на экранную клавишу «Сброс».

Подведём промежуточные итоги. Мы разобрались, что программа управляется событиями и уяснили, что Lazarus может обработать несколько событий «мыши»: клик (левой, правой, средней и др.) клавишей, двойной клик, движение указателя, вращение колёсика, нажатие и отпускание клавиши

мыши. Однако разнообразить функции интерфейса пользователя, сделать его комфортнее можно добавив обработку событий реальной (физической) клавиатуры.

2.2. Обработка клавиатурных событий

Физические (не экранные) клавиатуры персонального компьютера имеют стандарты передачи информации о нажатых клавишах. Lazarus способен перехватывать клавиатурные события и обрабатывать, переданные в буфер обмена, коды нажатых клавиш. Программист, тем самым, имеет возможность значительно расширить функционал интерфейса пользователя. Прежде чем на практике отрабатывать доступные программисту методы работы, уточним список клавиатурных событий: ввод символа, нажатие и отпускание клавиши, раздельное и одновременное нажатие управляющих клавиш (Ctrl, Shift, Alt).

Создайте новое приложение в Lazarus и сохраните в специально созданной для данной темы папке. Разместите на форме компонент StaticText с вкладки Additional. Компонент StaticText является аналогом Edit, но без возможности ввода символов непосредственно с клавиатуры, только через специально назначенные программистом обработчики. Данный компонент можно использовать в тех случаях, когда нужно ограничить пользователя некоторым подмножеством разрешенных символов, например, для калькулятора это могут быть символы 0 .. 9 и «,».

Установите для свойств компонента следующие значения:

```
Alignment       taRightJustify
BorderStyles    bsSunken
AutoSize        False
Width           150
Caption         0
```

```
Font.Size        10
Font.Style       [fsBold]
Transparent      False
Color            по усмотрению, один из светлых.
```

Выравнивание текста внутри компонента по правому краю для обеспечения схожести с калькулятором среды Window (рис. 9).

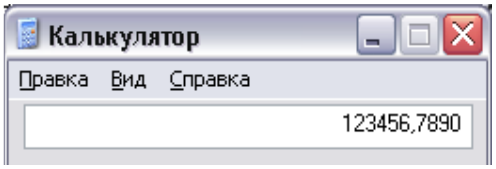

Рис.9. Дизайн поля ввода стандартного калькулятора.

Давайте рассмотрим возможности, которые предоставляет нам Инспектор Объектов.

Создайте для Form1 обработчик события OnKeyPress дважды кликнув по полю, находящемуся правее позиции OnKeyPress. В результате будет создана заготовка процедуры FormKeyPress. Одним из аргументов процедуры является переменная Key типа Char, которая возвращает символ, введенный с клавиатуры. Добавьте в заготовку процедуры код, который будет к содержимому StaticText1 добавлять символ ‘1’, если с клавиатуры введен символ ‘1’:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key='1' then
    StaticText1.Caption:=StaticText1.Caption+'1';
end;
```

Запустите программу и проверьте работоспособность процедуры.

Очевидная неловкость состоит в том, что когда исходное содержимое StaticText1 это ноль, то добавив к нему ‘1’ мы

должны получить не ‘01’, а ‘1’. Естественно, что для исключения этой ситуации можно добавить дополнительную проверку:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key='1' then
    if StaticText1.Caption='0'
      then StaticText1.Caption:='1'
      else StaticText1.Caption:=StaticText1.Caption+'1';
end;
```

Теперь добавим возможность ввода любой цифры, а не только ‘1’:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key in ['0'..'9'] then
    if StaticText1.Caption='0'
      then StaticText1.Caption:=Key
      else StaticText1.Caption:=StaticText1.Caption+Key;
end;
```

Не хватает запятой, таким же образом можно добавить и её:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if (Key in ['0'..'9']) or (Key=',') then
    if StaticText1.Caption='0'
      then StaticText1.Caption:=Key
      else StaticText1.Caption:=StaticText1.Caption+Key;
end;
```

Проверьте работоспособность такой процедуры, и вы без труда обнаружите некоторые нежелательные моменты в работе этой процедуры, а именно:

1) можно ввести несколько запятых,

2) когда начальное значение ноль, то при нажатии на ‘,’ следует получить результат ‘0,’, а не просто ‘,’. Поэтому внесем в процедуру дополнительный анализ:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
begin
  if Key in ['0'..'9'] then
    if StaticText1.Caption='0'
      then StaticText1.Caption:=Key
      else StaticText1.Caption:=StaticText1.Caption+Key;
  if ((Key=',') and (Pos(Key,StaticText1.Caption)=0))
    then StaticText1.Caption:=StaticText1.Caption+Key;
end;
```

Проверьте работоспособность доработанной процедуры. Всё бы хорошо, но отсутствует возможность внести исправление в случае ошибочного набора. Нужны два варианта: удалить *последний* набранный символ и удалить *всё* число. Давайте добавим в процедуру возможность исправления последнего набранного символа, для этого будем использовать клавишу Backspace. Однако на этом пути возникает одно препятствие – а как ввести в анализ символ Backspace? Давайте временно основную часть процедуры поместим в комментарии, но добавим строку вывода символа Backspace:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
Begin
  StaticText1.Caption:=Key;
  {
  if Key in ['0'..'9'] then
    if StaticText1.Caption='0'
      then StaticText1.Caption:=Key
      else StaticText1.Caption:=StaticText1.Caption+Key;
  if ((Key=',') and (Pos(Key,StaticText1.Caption)=0))
    then StaticText1.Caption:=StaticText1.Caption+Key;
  }
end;
```

Посмотрите на действие процедуры, нажимая различные клавиши – при вводе букв русского и английского алфавитов, цифр – эти символы выводятся в StaticText1 как есть, однако при нажатии на клавишу Backspace мы получаем неприемлемый результат. Обойти это затруднение можно сменив тип данных результата: давайте будем выводить тип данных не Char, а Byte. Это можно осуществить применив к переменной Key типа Char функцию Ord и преобразование типа IntToStr или метод ToString():

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
Begin
  StaticText1.Caption:=IntToStr(Ord(Key));
//StaticText1.Caption:=(Ord(Key)).ToString(); // так тоже можно

  {
  if Key in ['0'..'9'] then
    if StaticText1.Caption='0'
      then StaticText1.Caption:=Key
      else StaticText1.Caption:=StaticText1.Caption+Key;
  if ((Key=',') and (Pos(Key,StaticText1.Caption)=0))
    then StaticText1.Caption:=StaticText1.Caption+Key;
  }
end;
```

Поэкспериментируйте с нажатием клавиш и вы увидите коды вводимых символов, в частности по нажатию ‘0’ получите код 48, а по нажатию ‘ ’ (← это пробел) – код 32. *Узнайте* самостоятельно какой код у Backspace и Esc.

Итак, теперь добавим в нашу процедуру обработку клавиши Backspace:

```
procedure TForm1.FormKeyPress(Sender: TObject; var Key: Char);
Begin
  if Key in ['0'..'9'] then
    if StaticText1.Caption='0'
      then StaticText1.Caption:=Key
      else StaticText1.Caption:=StaticText1.Caption+Key;
  if ((Key=',') and (Pos(Key,StaticText1.Caption)=0))
    then StaticText1.Caption:=StaticText1.Caption+Key;
  if Ord(Key)=8 then StaticText1.Caption:=
     copy(StaticText1.Caption,1,length(StaticText1.Caption)-1);
```

```
end;
```

Хочу заметить, что предусмотрен и иной вариант проверки клавиш: Key = #8 – тут код приводится к символьному типу данных.

В обработке клавиши Backspace не учтены два момента:

1) перед удалением символа следует проверить, не является ли строка уже пустой,

2) при удалении последнего символа из строки в StaticText1 следует поместить ‘0’.

Учтем их следующим образом:

```
...
if Key=#8 then
  if length(StaticText1.Caption)=1
  then StaticText1.Caption:='0'
  else StaticText1.Caption:=
       copy(StaticText1.Caption,1,length(StaticText1.Caption)-1);
...
```

Пришла пора вспомнить наши планы по функционалу интерфейса калькулятора – нужны клавиши для удаления *последнего* набранного символа и клавиша для удаления *всего* числа. Первую мы смогли реализовать, теперь, аналогичным образом, попытайтесь добавить стирание всего содержимого StaticText1 по нажатию клавиши Delete. Попытка окажется неудачной, так как с помощью процедуры FormKeyPress нельзя проанализировать нажатие таких управляющих клавиш как стрелки, Insert, Delete, функциональных клавиш F1–F12 и т.п. Для преодоления этого препятствия следует использовать иной обработчик – не ввода символа, а нажатия клавиши.

К процедурам обработки нажатия клавиши следует отнести обработчики событий **OnKeyDown** (нажатие клавиши) и **OnKeyUp** (отпускание клавиши).

Создайте для Form1 обработчик события **OnKeyDown** дважды кликнув по полю, находящемуся в Инспекторе объектов правее позиции OnKeyDown. В результате будет создана заготовка процедуры FormKeyDown. Одним из аргументов процедуры является переменная Key типа Word (целочисленная, положительная, диапазон: 0..65535), которая возвращает код нажатой клавиши. Вторым аргументом является переменная Shift типа TShiftState. Справка Lazarus (загляните и убедитесь сами, это полезно для последующего самостоятельного познания сути иных процедур и их аргументов) гласит, что данный тип данных относится ко множествам:

```
Type TShiftState = set of TShiftStateEnum;
TShiftStateEnum =
     (ssShift, ssAlt, ssCtrl, ssLeft, ssRight, ssMiddle, ssDouble,
      ssMeta, ssSuper, ssHyper, ssAltGr, ssCaps, ssNum,
      ssScroll, ssTriple, ssQuad, ssExtra1, ssExtra2);
```

В данном случае в идентификаторе типа TShiftState слово Shift не относится именно к клавише Shift, а имеет самостоятельный смысл – перевод слова Shift гласит: изменять, переключать, а к клавиатурным переключателям, в свою очередь, относятся клавиши – Shift, Alt и Ctrl. Именно сочетание нажатых клавиш в данной процедуре возвращает эта переменная Shift. Если не нажата ни одна из клавиш переключателей, то значение Shift как множества будет []. Если нажата одна из клавиш, то в множество будет включен её идентификатор, например: [ssAlt]. Если нажаты 2 или 3 клавиши, то в множество будут включены и они, например: [ssAlt,ssCtrl].

Итак, давайте используем возможности данного обработчика события OnKeyDown для обработки нажатия клавиши Delete. Начнем с того, что узнаем какой именно код у клавиши Delete:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  StaticText1.Caption:=IntToStr(Key);
end;
```

Далее изменим код – создадим непосредственно обработчик события, выполняющий необходимое нам действие:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if Key=46 then StaticText1.Caption:='0';
end;
```

Для изучения возможностей использования переменной Shift исследуем способ замены стандартного сочетания клавиш для закрытия приложения **Alt+F4** на клавишу Esc. Код клавиши Esc равен 27 – соответственно добавим выполнение процедуры Close при возвращении переменной Key кода 27:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
  46: StaticText1.Caption:='0';
  27: Close;
  end;
end;
```

В данном обработчике присутствует некоторая неточность, так как не исключается нажатие клавиш переключателей совместно с клавишей Esc, то есть, например, выход из приложения будет происходить не только при нажатии клавиши Esc, но и при нажатии сочетания клавиш Shift+Esc. Для

исключения присутствия иных клавиш кроме Esc прямо укажем об этом в процедуре:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
    46: StaticText1.Caption:='0';
    27: if Shift=[] then Close;
  end;
end;
```

Как вы поняли, суть обработчика дословно звучит так: если множество нажатых клавиш переключателей пусто и нажата клавиша Esc, тогда закрой приложение.

Теперь исключим возможность выхода из приложения по сочетанию клавиш Alt+F4. Суть обработки состоит в том, чтобы сравнить множество нажатых клавиш переключателей с множеством [ssAlt] при нажатой функциональной клавише F4. В случае такого сочетания переменной Key присвоить код 0, то есть сообщить программе, что никакая клавиша и не нажималась (включая и клавишу F4). Код клавиши F4 узнать несложно, через код: «StaticText1.Caption:=IntToStr(Key);». Итоговая процедура выглядит изящно и компактно (стремитесь к этому и процесс кодирования будет доставлять вам удовольствие:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
    46:  StaticText1.Caption:='0';
    27:  if Shift=[] then Close;
    115: if Shift=[ssAlt] then Key:=0;
  end;
end;
```

Заметьте, что описание всех действий, выполняемых структурным оператором case значительно лаконичнее, чем оно же, но воспроизведенное на русском языке.

Итак, к настоящему моменту вы уже должны понимать разницу между обработкой введенного символа и нажатой клавиши. Однако для полноценной обработки всех возможных событий приложения этого недостаточно.

2.3. Приоритет обработки нажатия клавиш

Начнем с демонстрации неудачи. Добавьте на форму три кнопки, примерно так (рис.10):

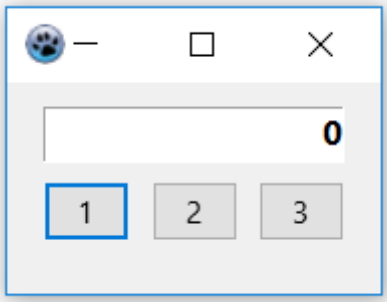

Рис.10. Компоненты для отработки приоритета ввода.

Теперь запустите приложение, и вы убедитесь, что всё, что было сделано ранее, уже не работает. В чем же дело? А в том, что эти экранные клавиши берут на себя фокус активности. Кстати, между клавишами можно перемещать фокус нажатием клавиши Tab или стрелочками. Теперь создайте процедуры обработки нажатия этих экранных клавиш:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if StaticText1.Caption='0'
    then StaticText1.Caption:='1'
    else StaticText1.Caption:=StaticText1.Caption+'1';
end;
```

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  if StaticText1.Caption='0'
    then StaticText1.Caption:='2'
    else StaticText1.Caption:=StaticText1.Caption+'2';
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  if StaticText1.Caption='0'
    then StaticText1.Caption:='3'
    else StaticText1.Caption:=StaticText1.Caption+'3';
end;
```

Запустите программу и убедитесь, что эти обработчики можно инициализировать кликом мыши по соответствующей экранной клавише или нажатием на клавишу ПРОБЕЛ при заблаговременно установленном фокусе на экранной клавише.

У каждого из компонентов есть свои обработчики событий и у экранных клавиш Button тоже. Посмотрите в Инспекторе Объектов на возможные события для Button1. Пока в программе есть обработчик события при клике мышкой по клавише – Button1Click, а обработчик события OnKeyDown пока не назначен. Поэтому ничего и не происходит при нажатии таких клавиш как Esc или Delete, даже при наличии фокуса на Button1. Давайте назначим обработчик для Button1:

```
procedure TForm1.Button1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  case Key of
  46:  StaticText1.Caption:='0';
  27:  if Shift=[] then Close;
  115: if Shift=[ssAlt] then Key:=0;
  end;
end;
```

При наличии фокуса на Button1 обработчик нажатия клавиш будет работать также, как и ранее, когда экранных клавиш на форме ещё не было. То есть один из возможных способов

обработки событий от множества объектов это сделать копии всех процедур обработки нажатия клавиш для всех компонентов (для Button1, Button2, Button 3 и т.д.). Но такой подход крайне неудобен, особенно, при значительном количестве компонентов (впоследствии мы рассмотрим способ, при котором можно избавиться от повторяющихся аналогичных процедур, сделав ссылки из разных компонентов на одну процедуру).

Кроме того, в калькуляторе Windows ни одна из клавиш не выделена (не имеет фокуса) и нам этого тоже не нужно. Отключить это можно в Инспекторе Объектов. Обратите внимание на свойства кнопок TabStop типа Boolean и TabOrder целочисленного типа. Свойство TabStop, равное true, означает, что на данном компоненте можно установить фокус. Свойство TabOrder позволяет назначить очередность (между компонентами) установки фокуса в работающем приложении последовательным переключением клавишей Tab. Нам фокус на клавишах ни к чему, поэтому для всех клавиш установите свойство TabStop равное false.

После выключения свойства TabStop отключается и возможность инициализировать созданные ранее обработчики нажатием клавиши ПРОБЕЛ, так как отсутствует возможность установить фокус на экранной клавише. Нас это вполне устраивает.

Правильный подход состоит в использовании встроенных возможностей среды Lazarus, в частности, свойства KeyPreview на форме и события OnShortCut.

> Внимание! В данной работе не обсуждается использование компонента HotKey с вкладки Win32 и назначение с помощью него «горячих клавиш» – комбинаций клавиш, на которые может реагировать приложение, даже если оно не имеет фокуса или запущено в трее. В данной работе обсуждается только непосредственная привязка обработчиков к событию: «нажатие клавиши».

Свойство KeyPreview (найдите в Инспекторе объектов) на форме позволяет отдать приоритет в обработке события от компонентов к форме, если перевести его в состояние True. Самостоятельно испытайте возможности и проверьте эффективность использования данного подхода.

Несколько сложнее использовать дополнительный обработчик клавиатурных событий OnShortCut. Дословно ShortCut можно перевести как кратчайший путь, а само событие OnShortCut можно найти в Инспекторе объектов и только для формы, но не для иных визуальных компонентов, принимающих фокус ввода (Button, Edit и др.). Итак, событие OnShortCut характерно только для формы приложения и возникает при нажатии на клавишу, но до инициализации события KeyPress или KeyDown формы или компонентов. Создайте заготовку процедуры обработки события OnShortCut дважды кликнув на соответствующей позиции в Инспекторе Объектов:

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled:
Boolean);
begin

end;
```

В процедуре используются два аргумента: Msg (сообщение) и Handled (глагол: обработать).

Переменная Msg типа record имеет несколько полей, но нас будет интересовать только поле CharCode, которое и возвращает код нажатой клавиши. Именно *нажатой клавиши*, а **не** *введенного символа*. Это, например, имеет значение при вводе символа ‘1’ с основной клавиатуры или дополнительного блока цифровой клавиатуры (физически клавиши разные, а вводимый символ один).

Переменная Handled типа Boolean позволяет управлять последующей (после процедуры FormShortCut) обработкой нажатия клавиши. Если эта переменная переведена в состояние True, то даже при наличии обработчиков событий OnKeyPress, OnKeyDown и/или OnKeyUp, иных действий, кроме описанных, в процедуре FormShortCut производиться не будет.

Проведем эксперимент.

Для начала уточним текущее состояние программы.

Выход из программы по нажатию клавиши Esc описан в процедуре FormKeyDown и в процедуре Button1KeyDown. Однако процедура FormKeyDown не действует (верните KeyPreview в исходное состояние), так как события «нажатие клавиши» перехватывается тем компонентом из списка Button1, Button2 и Button3, у которого находится фокус. Если у Button1, то выход по Esc возможен, так как Button1KeyDown содержит такое действие, если же у Button2 или у Button3, то выход по Esc не возможен, так как у этих компонентов обработчик события ButtonKeyDown не описан.

Что мы хотим получить?

При наличии фокуса на любом из компонентов формы обработчик нажатия клавиш должен обеспечить:

1) выход из программы по нажатию клавиши Esc;

2) ввод цифр и запятой в StaticText1;

3) обнуление StaticText1 клавишей Delete и стирание последнего введенного символа клавишей Backspace.

Будем решать задачу последовательно с проверкой результатов работы. Добавьте в процедуру FormShortCut один структурный оператор case:

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled:
Boolean);
begin
  case Msg.CharCode of
  27: close;
  end;
end;
```

Проверьте его работоспособность и, если работает, то «убейте» процедуру FormKeyDown – она нам больше не нужна.

Правильно избавиться от процедуры можно следующим образом:

1) выделите процедуру и все её содержимое, находящееся внутри операторных скобок begin end и удалите их,

2) нажмите Ctrl+F9 – произойдет проверка и компиляция кода без запуска приложения, при этом появится вопрос от среды программирования о возможности удаления, подтвердите ваше намерение, вследствие чего удалятся описание процедуры в разделе описания формы type TForm1 = class(TForm), а также ссылка на неё в Инспекторе Объектов. Чтобы не загромождать программу процедурами удалите и все остальные процедуры описанным выше способом за исключением, конечно, процедуры FormShortCut. Все необходимые действия переместим именно в неё.

Итак, в модуле у вас только одна процедура – FormShortCut, добавим в неё код по выводу символа «запятая» в компонент StaticText1. Для этого допишем в структурном операторе case соответствующий код:

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled:
Boolean);
begin
  case Msg.CharCode of
    27: close;
```

```
      110,188,191:
        if Pos(',',StaticText1.Caption)=0
          then StaticText1.Caption:=StaticText1.Caption+',';
    end;
  end;
```

Здесь коды 110, 188 и 191 соответствуют различным вариантам ввода символа «запятая». Убедитесь в этом сами, проанализировав значение Msg.CharCode при нажатии различных клавиш.

Теперь добавим к структурному оператору case еще и возможность ввода цифр в StaticText1:

```
...
48..57,96..105:
  begin
    if Msg.CharCode>95 then d:=48 else d:=0;
    if StaticText1.Caption='0'
      then StaticText1.Caption:=chr(Msg.CharCode-d)
      else StaticText1.Caption:=StaticText1.Caption+chr(Msg.CharCode-d);
  end;
...
```

Диапазон 48..57 означает обработку нажатия цифровых клавиш на основной клавиатуре, а – 96..105 – на дополнительной (цифровой) клавиатуре. Переменная d типа byte (которую, кстати, необходимо объявить в разделе var процедуры FormShortCut), позволяет обойтись одним обработчиком нажатия клавиши с цифрой, смещая код на 48 в случае, если нажата цифра на дополнительной клавиатуре.

Для правильной работы процедуры следует также добавить в неё строчку кода: «Handled:=true», которая означает, что обработка нажатия клавиши уже произведена и никакими иными процедурами текущее нажатие более обрабатывать не следует.

Добавим в тот же структурный оператор обработку клавиш Backspace и Delete и, в итоге, вместо множества отдельных процедур имеем одну, отвечающую за все предусмотренные нами функции:

```
procedure TForm1.FormShortCut(var Msg: TWMKey; var Handled: Boolean);
var d: byte;
begin
  case Msg.CharCode of
  8: case length(StaticText1.Caption) of
     0: ;
     1: StaticText1.Caption:='0';
     else
       StaticText1.Caption:=copy(StaticText1.Caption,1,length(
         StaticText1.Caption)-1);
  end;
  46: StaticText1.Caption:='0';
  27: close;
  110,188,191:
      if Pos(',',StaticText1.Caption)=0
        then StaticText1.Caption:=StaticText1.Caption+',';
  48..57,96..105:
      begin
        if Msg.CharCode>95 then d:=48 else d:=0;
        if StaticText1.Caption='0'
          then StaticText1.Caption:=chr(Msg.CharCode-d)
          else StaticText1.Caption:=StaticText1.Caption+chr(Msg.CharCode-d);
      end;
  end;
  Handled:=true;
end;
```

2.4. Переменная Sender

А как же экранные клавиши? Они есть, но ничего не обрабатывают. Напомню, что раньше мы привязывали к ним ввод соответствующей цифры. Теперь у нас работает ввод цифр с клавиатуры, но процедуры ввода цифр по нажатию мышкой на экранные клавиши мы «убили». И сделали это не зря. Дело в том, что ранее мы шли экстенсивным путем – на каждую экранную клавишу писали отдельную процедуру. Но программный код в них, по существу, повторялся. Сейчас мы исследуем возможности переменной Sender для оптимизации программного кода на примере создания заготовки для программы «Калькулятор».

Итак, необходимо разместить на форме нужные клавиши, назначение которых вводить цифры в компонент StaticText1. У нас на форме уже есть три кнопки, рассмотрим общий принцип использования переменной Sender на примере этих трех кнопок.

Для начала создайте процедуру Button1Click (кликнув дважды по самой клавише или через вкладку События Инспектора объектов) и заполните ее следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if StaticText1.Caption='0'
     then StaticText1.Caption:='1'
     else StaticText1.Caption:=StaticText1.Caption+'1';
end;
```

Проверьте работоспособность этой процедуры – очевидно, что она обеспечивает ввод цифры 1 при нажатии на экранную клавишу Button1.

Пока у экранных клавиш Button2 и Button3 нет аналогичных обработчиков. Давайте не будем для них создавать новых обработчиков, а назначим им уже существующий. Для этого выделите обе клавиши, кликнув один раз сначала по одной из них (для того чтобы выделить её) и затем при нажатой клавише Shift кликните один раз по другой. Теперь можно менять свойства или назначать обработчики событий сразу для обеих клавиш. Перейдите в Инспектор Объектов на вкладку События (Events) и кликните один раз на поле правее OnClick – активируется выпадающий список (рис.11):

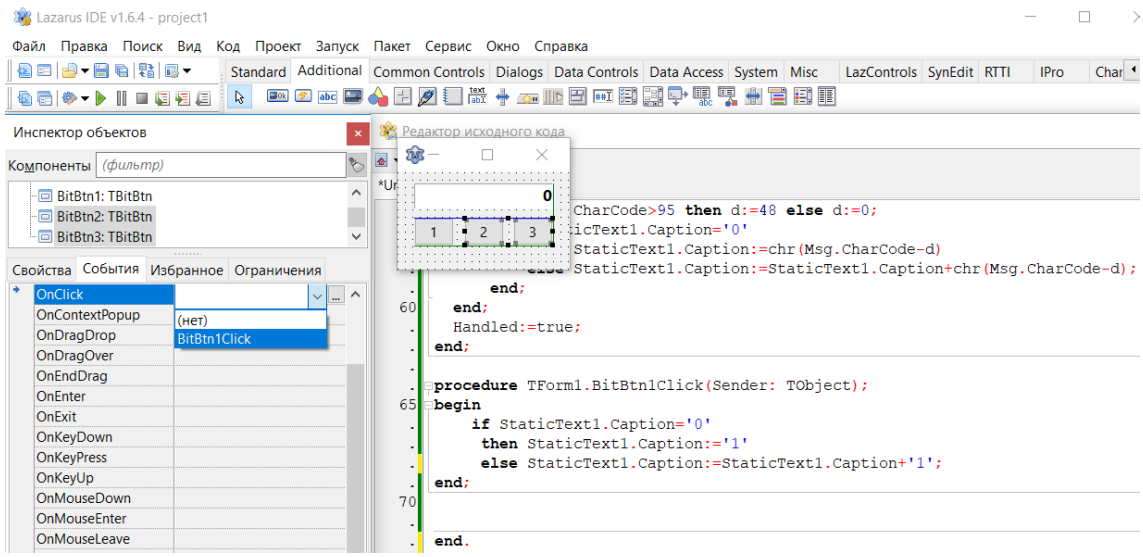

Рис.11. Подключение нескольких компонентов к одному обработчику события.

Там для выбора будет только один созданный нами обработчик Button1Click, его и следует выбрать. То есть для всех трех клавиш мы назначили один обработчик. Испытаем программу – очевидно, что при клике на любую из трех клавиш в StaticText1 будет добавляться одна и та же цифра – 1. При этом клавиатурный обработчик всех ранее созданных нами функций остается в работоспособном состоянии (Backspace, Delete и др.). То есть нам осталось только добавить анализатор нажатой экранной клавиши и каким-то образом передать эту информацию в процедуру Button1Click. Вот тут-то нам и пригодиться переменная **Sender**, кстати, в переводе на русский язык это слово означает отправитель. Отправителем в случае работы приложения является тот компонент, который и вызвал срабатывание данной процедуры. У нас это могут быть клавиши: Button1, Button2 и Button3. С учетом этих знаний доработаем процедуру:

```
procedure TForm1.Button1Click(Sender: TObject);
var c: char;
begin
  if Sender=Button1 then c:='1';
  if Sender=Button2 then c:='2';
```

```
  if Sender=Button3 then c:='3';
  if StaticText1.Caption='0'
     then StaticText1.Caption:=c
     else StaticText1.Caption:=StaticText1.Caption+c;
end;
```

Проверьте работоспособность процедуры, теперь ввод цифр происходит в соответствии с нажатой экранной клавишей. Однако сам код не выглядит оптимальным и универсальным, ведь компонентов на форме может быть существенно больше чем три.

Есть другой способ использования переменной Sender. Можно непосредственно через Sender обратиться к свойствам компонента (Name, Caption, Tag и др.) через конструкцию вида: Sender as TButton. Изменим процедуру:

```
procedure TForm1.Button1Click(Sender: TObject);
var c: char;
begin
  if (Sender as TButton).Name='Button1' then c:='1';
  if (Sender as TButton).Name='Button2' then c:='2';
  if (Sender as TButton).Name='Button3' then c:='3';
  if StaticText1.Caption='0'
     then StaticText1.Caption:=c
     else StaticText1.Caption:=StaticText1.Caption+c;
end;
```

Пока еще выглядит не оптимально, но есть уже намеки на лучший исход: конструкция (Sender as TButton).Name одинакова во всех условных операторах, но возвращает разные значения, соответствующие свойству Name нажатой экранной клавиши. Воспользуемся этим так:

```
procedure TForm1.Button1Click(Sender: TObject);
var c: char;
begin
  c:=(Sender as TButton).Name[7];
  if StaticText1.Caption='0'
     then StaticText1.Caption:=c
     else StaticText1.Caption:=StaticText1.Caption+c;
end;
```

Обратите внимание, что свойство Name имеет строковый тип, поэтому с ним можно обращаться как со строкой. Строка

– это массив символов и в процедуре мы анализируем седьмой символ, который и несет номер клавиши. Можно, конечно, так не усложнять и взять номер прямо из свойства Caption, ведь у нас прямо на кнопках написаны их номера:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if StaticText1.Caption='0'
     then StaticText1.Caption:=(Sender as TButton).Caption
     else StaticText1.Caption:=StaticText1.Caption+(Sender as TButton).Caption;
end;
```

Или пойти несколько иным путем:

```
procedure TForm1.Button1Click(Sender: TObject);
var s: TButton;
begin
  s:=(Sender as TButton);
  if StaticText1.Caption='0'
     then StaticText1.Caption:=s.Caption
     else StaticText1.Caption:=StaticText1.Caption+s.Caption;
end;
```

Таким образом, вы создали две процедуры, обеспечивающие работу калькулятора:

1) процедура обработки клавиатурных событий;

2) процедура обработки события мышкой при нажатии по экранным клавишам.

На форме калькулятора из управляющих элементов присутствуют только кнопки, поэтому данного подхода к обработке переменной Sender вам будет вполне достаточно. Однако этим не исчерпываются все возможности по оптимизации программного кода с использованием этой переменной.

Задание для самостоятельного исполнения.

Для закрепления полученных знаний и приобретения навыков использования клавиатурных событий **доработайте созданный ранее калькулятор** (работающий только на событиях мыши), используя переменную **Sender** для оптимизации обработчиков событий мыши и дополнив его обработчиком клавиатурных событий OnShortCut.

Глава 3. Обработка файлов

3.1. Текстовые файлы

3.1.1. Доступ к файлам

Текстовые файлы рассматриваются как совокупность строк переменной длины. Доступ к некоторой необходимой строке возможен лишь после последовательного чтения предыдущих. Начать чтение строк файла можно только с первой строки. Чтение строки организуется с помощью процедуры ReadLn. Данная процедура имеет два аргумента. Первый аргумент используется для обозначения имени файла, из которого осуществляется чтение. Второй аргумент – переменная в которую будет помещена очередная строка из файла. Пример: ReadLn(f, s) – здесь f – имя файловой переменной типа textfile, ассоциированной с конкретным файлом, а s – имя строковой переменной, в которую помещается считанная из файла строка.

Для того чтобы была возможность обращаться к файлу через имя файловой переменной следует сначала их связать процедурой AssignFile(f, n), где f – имя файловой переменной, а n – имя файла. Имя файла можно задать непосредственно через строковую константу:

```
AssignFile(f,'start.txt')
```

или через строковую переменную:

```
n:='start.txt';
AssignFile(f,n);
```

Организация доступа к файлу этим не исчерпывается. Кроме *связывания файловой переменной* следует также *инициализировать файл* одной из процедур: Reset, Rewrite или Append. Процедура Reset открывает файл для чтения, Rewrite – для записи, Append – для добавления записей.

После работы с файлом его следует закрыть процедурой CloseFile(f), где f – имя файловой переменной.

Создайте новое приложение в среде Lazarus. Разместите на форме компоненты: Memo и Button. Создайте на кнопке обработчик события Button1Click и заполните его кодом по образцу:

```
procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s: string;
begin
  AssignFile(f,'start.txt');
  Reset(f);
  readln(f,s);
  memo1.Lines.Add(s);
  closefile(f);
End;
```

Данная процедура при отсутствии файла start.txt, естественно, работать не будет. Сохраните проект в заранее подготовленную папку и в ней же разместите файл start.txt, в котором должны быть несколько произвольных строк текста.

Если вы создаете сложное приложение и, в основной папке программы впоследствии будут располагаться различные файлы и каталоги, то, скорее всего, возникнет необходимость читать файлы не только из основного каталога программы, но и из подкаталогов.

Создайте в основном каталоге вашей программы подкаталог INI и в него скопируйте файл start.txt. Переименуйте его в start.ini. Для того чтобы ассоциировать его с файловой переменной можно поступить так: AssignFile(f, 'ini/start.ini');

Доработайте самостоятельно процедуру, чтобы она читала строку из файла start.ini находящегося в подкаталоге INI. Апробируйте программу.

Другой способ состоит в том, чтобы сформировать полное имя файла, включая наименование диска и путь к файлу.

Для этого сначала следует определить текущую папку программы с помощью процедуры GetDir(0, t), где 0 – указывает на текущий диск, а t – переменная строкового типа, которая возвращает путь к основному каталогу программы (откуда она была запущена на исполнение).

Доработайте процедуру следующим образом и оцените результат её работы:

```
procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s: string;
begin
  getdir(0,s);
  memo1.Lines.Add(s);
  AssignFile(f,'ini/start.ini');
  Reset(f);
  readln(f,s);
  memo1.Lines.Add(s);
  closefile(f);
end;
```

В поле Memo выводится путь к основному каталогу программы. Путь возвращает переменная строкового типа s. Имя файла также может быть переменной или константой строкового типа, поэтому к ним можно применить операцию конкатенации строк. Например, если файл находится в подкаталоге INI, то можно взять путь к основному каталогу программы, к нему добавить имя подкаталога INI и имя файла:

```
procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s,n,k: string;
begin
  getdir(0,k);          // определяем основной каталог
  n:='start.ini';       // указываем имя файла
  s:=k+'/ini/'+n;       // определяем полное имя файла
  AssignFile(f,s);
  Reset(f);
  readln(f,s);
  memo1.Lines.Add(s);
  closefile(f);
```

```
end;
```

До сих пор мы только читали файл, теперь попробуем писать в файл, для чего доработайте процедуру следующим образом:

```
procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s,n,k: string;
begin
  getdir(0,k);          // определяем основной каталог
  n:='start.ini';       // указываем имя файла
  s:=k+'/ini/'+n;       // определяем полное имя файла
  AssignFile(f,s);
  Reset(f);
  readln(f,s);
  memo1.Lines.Add(s);
  closefile(f);

  n:='start2.txt';      // указываем имя файла
  s:=k+'/ini/'+n;       // определяем полное имя файла
  AssignFile(f,s);
  Rewrite(f);           // открываем файл для записи
  writeln(f,s);
  closefile(f);
end;
```

Испытайте её и проанализируйте содержимое нового файла. Объясните полученный результат и измените код процедуры так, чтобы первая строка из исходного файла переписывалась во второй файл.

3.1.2. Обработка ошибок обращения к файлу.

Пришло время обсудить *порядок обработки ошибок* ввода/вывода.

Рассмотрим два способа. **Первый** основан на использовании стандартного оператора обработки исключительных ситуаций: **try except**. Напомню кратко синтаксис этого опера-

тора: после ключевого слова **try** (попробовать) должны располагаться операторы, при выполнении которых возможно появление исключительной ситуации (в нашем случае, например, отсутствие файла); после ключевого слова **except** (исключить) – операторы, которые по замыслу программиста должны выполняться при наступлении исключительной ситуации.

```
Procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s,n,k: string;
begin
  getdir(0,k);          // определяем основной каталог
  n:='start.ini';       // указываем имя файла
  s:=k+'/ini/'+n;       // определяем полное имя файла
  try
    AssignFile(f,s);
    Reset(f);
    readln(f,s);
    memo1.Lines.Add(s);
    closefile(f);
  except
    on EInOutError do memo1.Lines.Add('файл '+s+' не
найден');
  end;
end;
```

Апробируйте эту процедуру при наличии файла start.ini и при его отсутствии. Если вы запускаете программу из среды Lazarus, то, прежде всего, сама среда будет обрабатывать ошибки и сообщит вам об этом (рис.12).

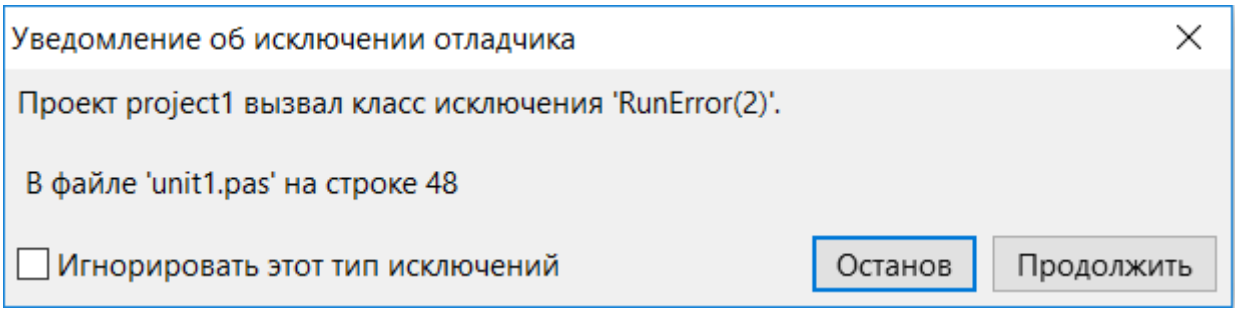

Рис.12. Окно уведомления об исключительной ситуации.

Если вы выберете вариант «Продолжить» исполнение программы, то далее уже сама программа будет заниматься обработкой исключений и, согласно коду, выведет предупреждающее сообщение в поле Memo1 (рис.13).

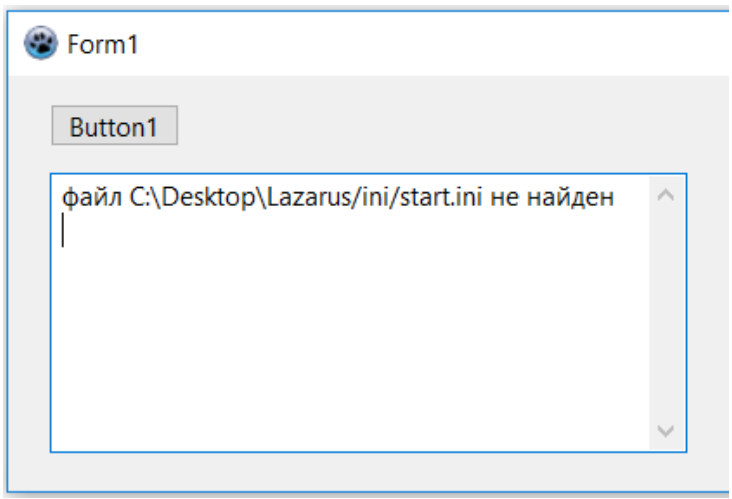

Рис.13. Результат обработки исключительной ситуации.

В структурном операторе try except не обязательно указывать тип исключительной ситуации (более подробно об операторе можно посмотреть в справке Lazarus). В нашем случае процедуру можно несколько сократить:

```
Procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s,n,k: string;
begin
  getdir(0,k);          // определяем основной каталог
  n:='start.ini';       // указываем имя файла
  s:=k+'/ini/'+n;       // определяем полное имя файла
  try
    AssignFile(f,s);
    Reset(f);
    readln(f,s);
    memo1.Lines.Add(s);
    closefile(f);
  except
    memo1.Lines.Add('файл '+s+' не найден');
  end;
end;
```

Организовать работу программы можно и иным способом. Второй способ основан на использовании стандартной

функции FileExists, которая возвращает True, если файл существует, и False – если не существует:

```
Procedure TForm1.Button1Click(Sender: TObject);
var f: textfile;
    s,n,k: string;
begin
  getdir(0,k);          // определяем основной каталог
  n:='start.ini';       // указываем имя файла
  s:=k+'/ini/'+n;       // определяем полное имя файла
  if FileExists(s) then
  begin
    AssignFile(f,s);
    Reset(f);
    readln(f,s);
    memo1.Lines.Add(s);
    closefile(f);
  end
  else
    memo1.Lines.Add('файл '+s+' не найден');
end;
```

3.1.3. Стандартные процедуры работы с файлами.

Приведу в качестве справки еще несколько стандартных процедур работы с файлами и каталогами:

1) procedure Rename(var f; NewName: String); – файл, ассоциированный с файловой переменной f, переименовывается в NewName, перед выполнением файл должен быть закрыт;

2) function EOF(var f): Boolean; – возвращает True, если достигнут конец файла;

3) procedure Flush(var f); – очищает буфер файла, обеспечивая сохранность последних изменений;

4) procedure MkDir(Dir: String); – создает новый каталог;

5) procedure RmDir(Dir: String); – удаляет каталог (если он пустой);

6) procedure Erase(var f); – удаляет файл, перед выполнением файл должен быть закрыт;

7) function DeleteFile(const FileName: string): Boolean; – удаляет файл без связывания с файловой переменной, перед выполнением файл должен быть закрыт;

8) function FindFirst(const Path: string; Attr: Integer; var F: TSearchRec): Integer; – функция возвращает 0 при успешном поиске, используется для инициализации переменной f, используемой при последующем поиске функцией FindNext.

Параметр Path может содержать путь к каталогу, в котором организуется поиск, и должен содержать маску выбора файлов. Маска формируется с помощью символов-заменителей:

? – означает, что на этой позиции может находиться один из разрешенных символов;

* – означает, что на этой позиции могут находиться несколько (включая 0) разрешенных символов.

Например:

. – любой файл из текущего каталога;

a*.* – любой файл из каталога, начинающийся с символа a;

c:\pas\a?.pas – файлы с расширением pas из каталога c:\pas\, имя которых начинается с символа a и содержит два символа (a0.pas, a1.pas, aa.pas, …).

Параметр Attr формируется битами разрядной сетки и может приобретать значения из списка констант:

faReadOnly – только чтение;

faHidden – скрытый файл;

faSysFile – системный файл;

faVolumeID – идентификатор тома;

faDirectory – имя вложенного каталога;

faArchive – архивный файл;

faAnyFile – любой файл.

Переменная f возвращает имя найденного файла.

9) function FindNext(var F: TSearchRec): Integer; – функция при успешном поиске возвращает 0.

Переменная f возвращает имя найденного файла.

10) procedure FindClose(var F: TSearchRec); – освобождает память, выделенную для поиска файлов.

3.1.4. Обсуждение примеров работы с файлами.

Разберем на простых примерах некоторые из приведенных стандартных подпрограмм.

Пример №1. Написать процедуру построения списка файлов с расширением ini, находящихся в подкаталоге INI.

Программу выполним на основе предыдущей, для чего добавьте на форму еще одну кнопку – Button2 (рис.14), а в коде создайте следующий обработчик события:

```
Procedure TForm1.Button2Click(Sender: TObject);
var sr: TSearchRec;
    m,k,p: string;
begin
  getdir(0,k);          // определяем основной каталог
  k:=k+'/ini/';         // назначаем текущий каталог
  m:='*.ini';           // назначаем маску
  p:=k+m;               // назначаем параметр Path
  memo1.Lines.Clear;    // очищаем memo1
  if FindFirst(p,faAnyFile,sr)=0 then
                        // если удалось найти файл
  repeat
    memo1.Lines.Add(sr.Name);  //  выводим  имя  файла  в
memo1
  until FindNext(sr)<>0;         // пока не закончатся
файлы
  FindClose(sr);              // освобождаем память
end;
```

Комментарии помогут вам разобраться с особенностями организации процедуры. В процедуре организован цикл с помощью структурного оператора repeat until. Особенность его состоит в том, что тело цикла выполнится хотя бы один раз, который наступит, если будет найден хотя бы один файл (`if`

`FindFirst(p,faAnyFile,sr)=0 then`), соответствующий заданным в маске условиям.

Тело цикла будет повторяться пока в каталоге функцией FindNext(sr) будет находиться очередной файл, соответствующий заданным в маске условиям. Цикл закончится, когда очередной файл не будет обнаружен – FindNext(sr)<>0.

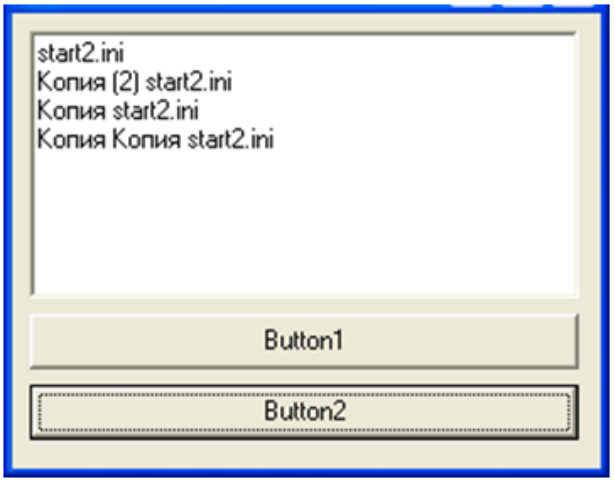

Рис.14. Вид формы по работе с файлами.

На скриншоте показан пример работы программы. Апробируйте процедуру.

Пример №2. Написать процедуру удаления файлов с расширением ini, находящихся в подкаталоге INI.

Программу выполним на основе предыдущей, для чего добавьте на форму еще одну кнопку – Button3. Создайте следующий обработчик события:

```
procedure TForm1.Button2Click(Sender: TObject);
var sr: TSearchRec;
    m,k,p,s: string;
    f: textfile;
begin
  getdir(0,k);           // определяем основной каталог
  k:=k+'/ini/';          // назначаем текущий каталог
  m:='*.ini';            // назначаем маску
  p:=k+m;                // назначаем параметр Path
  if FindFirst(p,faAnyFile,sr)=0 then // если удалось найти файл
  repeat
    s:=k+sr.Name;         // формируем полное имя файла
    AssignFile(f,s);      // ассоциируем имя с файловой переменной
    erase(f);             // удаляем файл
  until FindNext(sr)<>0;  // пока не закончатся файлы
  FindClose(sr);          // освобождаем память
end;
```

Комментарии помогут вам разобраться с особенностями организации процедуры. Изменено тело цикла – вместо вывода имени файла в memo1, производится удаление файла. Апробируйте процедуру.

Можно несколько упростить данную процедуру, используя процедуру DeleteFile вместо Erase. При этом не требуется проводить ассоциацию имени файла с файловой переменной – AssignFile(f,s); и, соответственно, нет необходимости объявлять ее в разделе var (f: textfile;). Произведите эту доработку самостоятельно.

Пример №3. Написать процедуру переименования файлов с расширением ini, находящихся в подкаталоге INI, в файлы с расширением txt.

Программу выполним на основе предыдущей, для чего добавьте на форму еще одну кнопку – Button4. Создайте следующий обработчик события:

```
procedure TForm1.Button2Click(Sender: TObject);
var sr: TSearchRec;
    m,k,p,s,sn: string;
    f: textfile;
begin
  getdir(0,k);              // определяем основной каталог
  k:=k+'/ini/';             // назначаем текущий каталог
  m:='*.ini';               // назначаем маску
  p:=k+m;                   // назначаем параметр Path
  if FindFirst(p,faAnyFile,sr)=0 then // если удалось найти файл
  repeat
    s:=k+sr.Name;           // формируем полное имя файла
    AssignFile(f,s);        // ассоциируем имя с файловой переменной
    sn:=copy(s,1,length(s)-3)+'txt';  // меняем расширение файла
    Rename(f,sn);           // переименовываем файл
  until FindNext(sr)<>0;    // пока не закончатся файлы
  FindClose(sr);            // освобождаем память
end;
```

Комментарии помогут вам разобраться с особенностями организации процедуры.

Следует заметить, что для визуальной (и не только) работы пользователя с файлами и каталогами удобнее использовать специальные компоненты из палитры Lazarus – компоненты FileListBox, DirectoryListBox, DriveComboBox и FilterComboBox с вкладки Win 3.1. Однако рассмотрение свойств, событий и методов этих компонентов не входит в содержание данной работы.

Задания для самостоятельного исполнения.

#3.1. В примере №1 разработана процедура построения списка файлов с расширением ini, находящихся в подкаталоге INI. Доработайте её таким образом, чтобы она выдавала список файлов ini, имеющих размер не менее заданного. Для задания размера файла в килобайтах разместите на форме поле Edit. Размер файла легко узнать, используя поле size типизированной переменной sr из примера №1. Пример обращения к полю size:

```
Repeat
    memo1.Lines.Add(IntToStr(sr.Size)+#9+sr.Name);
until FindNext(sr)<>0;
```

Это модификация цикла из процедуры примера №1. Добавление к строке кода #9 осуществляет вставку символа табуляции. Апробируйте процедуру, после чего выполните самостоятельно задание 3.1.

#3.2. Создайте простейший текстовый редактор на основе компонента Memo. Основная особенность редактора состоит в том, что в нем предусмотрена возможность автосохранения резервной копии (*.bak) редактируемого файла. Должна быть предусмотрена возможность отключения опции «автосохранение» через меню программы. Частоту автосохранения

($T_{авт}$ раз в 1 мин., 5 мин., 10 мин. и т.д.) можно менять. В момент открытия файла (или в момент включения опции «автосохранение») при включенной опции «автосохранение» включается отсчет времени и при наступлении $T_{авт}$ происходит сохранение резервной копии. В программе должна быть предусмотрена возможность отката к резервной копии (к *.bak файлу), то есть возможность загрузить последний сохраненный резервный файл – файл с именем редактируемого файла, но с расширением bak.

3.2. ТИПИЗИРОВАННЫЕ ФАЙЛЫ

3.2.1. Организация доступа к типизированным файлам.

Организация доступа к типизированному файлу аналогична организации доступа к текстовому файлу: необходимы процедуры связывания файла с файловой переменной (AssignFile) и инициализации файла (для чтения, записи или перезаписи). Однако есть и отличия – к записям типизированного файла можно обращаться как последовательно, так и в любом порядке. Это обстоятельство связано с тем, что длина любого компонента типизированного файла постоянна и, собственно, зависит от объявленного типа записей файла. Последовательное обращение к записям типизированного файла организуется стандартными процедурами Read и Write. Если есть необходимость сменить порядок чтения/записи, то можно воспользоваться процедурой Seek(f,k), которая смещает указатель текущего положения в файле f (f – файловая переменная) к требуемому компоненту k (k – переменная типа LongInt).

Для того чтобы узнать размер файла в компонентах его типа можно воспользоваться стандартной функцией FileSize(f). Текущую позицию в файле (порядковый номер

компонента файла, который будет обрабатываться следующей операцией ввода/вывода) возвращает функция FilePos(f).

3.2.2. Пример работы с типизированными файлами.

Рассмотрим организацию работы с типизированными файлами на примере.

Разработаем программу – заготовку под текстовый редактор. Основная особенность редактора состоит в том, что цвета фона и шрифта зависят от дня недели и есть возможность эти цвета настроить.

Создайте новое приложение, на главной форме разместите компоненты: Memo1 и TabControl1 (с вкладки Common Controls). В компоненте TabControl1 (в разрабатываемом приложении он нужен для настройки цветов по дням недели) добавьте вкладки (Пн, Вт, Ср, Чт, Пт, Сб, Вс) через свойство Tabs, набрав соответствующие строки.

На компонент TabControl1 положите две метки Label (с надписями – шрифт и фон) и два компонента ColorBox (с вкладки Additional). Для цвета шрифта у ColorBox1 оставьте выделенный цвет clBlack (свойство Selected), а для цвета фона у ColorBox2 измените свойство Selected на clWhite.

Выберите подходящую иконку для приложения и установите соответствующий заголовок формы (рис.15).

После визуальной настройки компонентов можно приступать к созданию обработчиков событий. Всего их будет пять:

1) при изменении цвета в ColorBox1 будет меняться цвет шрифта в Memo1;

2) при изменении цвета в ColorBox2 будет меняться цвет фона в Memo1;

3) при нажатии на ярлычок TabControl1 будут происходить изменения: устанавливаться цвета в ColorBox1, ColorBox2 и Memo1 для выбранного дня недели;

4) при запуске приложения подгружаются цветовые настройки из ini файла и устанавливается цветовая настройка Memo1, соответствующая текущему дню недели;

5) при закрытии приложения цветовые настройки по дням недели сохраняются в ini файл.

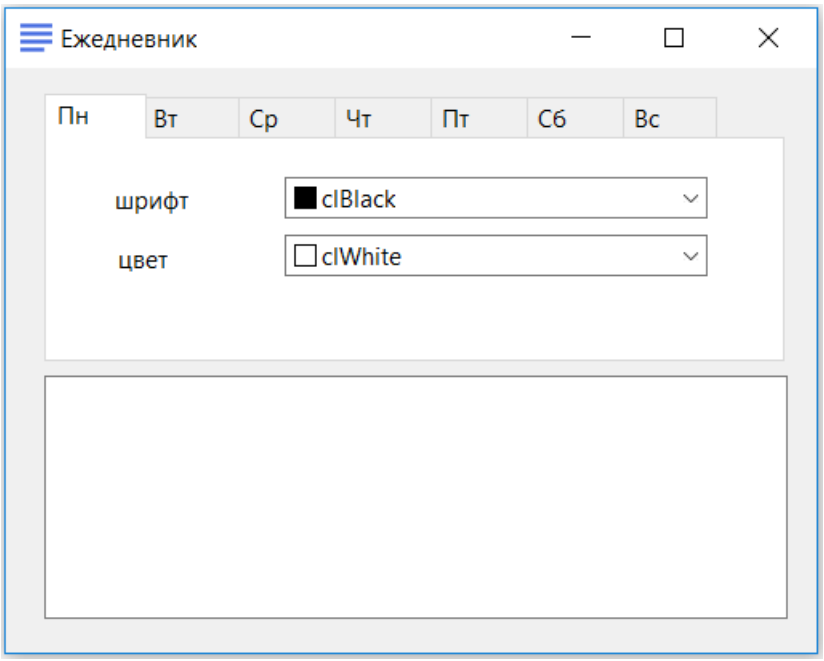

Рис.15. Дизайн формы «Ежедневника».

Первые два обработчика достаточно просты. Через инспектор объектов для компонентов ColorBox1 и ColorBox2 создайте заготовки под обработчики событий OnChange и заполните их следующим кодом:

```
procedure TForm1.ColorBox1Change(Sender: TObject);
begin
  memo1.Font.Color:=ColorBox1.Selected;
end;
procedure TForm1.ColorBox2Change(Sender: TObject);
begin
  memo1.Color:=ColorBox2.Selected;
end;
```

Апробируйте работу созданных процедур (рис.16).

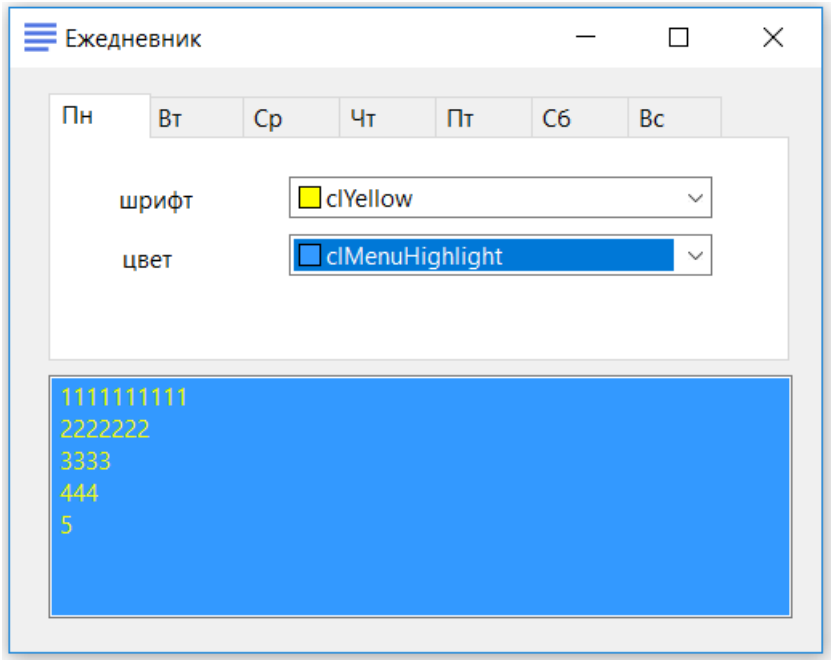

Рис.16. Дизайн формы «Ежедневника».

Переходим к *третьему обработчику* – он должен менять цветовые настройки многострочного текстового поля memo1 в зависимости от выбранного пользователем дня недели, то есть по клику на соответствующем ярлычке TabControl1. Значения цветов должны где то храниться, для чего создадим массив записей с двумя полями – одно поле для хранения цвета шрифта, а другое для хранения цвета фона.

В разделе глобальных переменных модуля добавим описание массива:

m: array[1..7] of dn;

где dn нестандартный тип данных, подразумевающий наличие двух полей записи: цвет шрифта и цвет фона. Идентификаторы полей у типа dn придумаем сами – например, cfont и cback. Нестандартный тип данных следует описать в соответствующем разделе модуля – в разделе Type:

```
Type
    dn = record
    cfont,cback: TColor;
  end;
var
  Form1: TForm1;
      m: array[1..7] of dn;
```

Итак, массив m описан как глобальная переменная и доступен в любой процедуре модуля. На начальном этапе, пока еще нет ini файла программы, при запуске приложения установим значения для всех дней недели одинаковыми: цвет фона – белый, цвет шрифта – черный. Чтобы реализовать эту возможность, создайте, через инспектор, объектов для главной формы приложения заготовку под обработчик события OnCreate и заполните его следующим кодом (*это 4-ый из 5-ти разрабатываемых обработчиков*):

```
procedure TForm1.FormCreate(Sender: TObject);
var i: byte;
begin
  for i:=1 to 7 do
  begin
    m[i].cfont:=clBlack;
    m[i].cback:=clWhite;
  end;
end;
```

Теперь уже есть возможность переключаться между цветовыми настройками дней недели. Через инспектор объектов для компонента TabControl1 создайте обработчик события OnChange и заполните его следующим кодом (*это 3-ий из 5-ти разрабатываемых обработчиков*):

```
procedure TForm1.TabControl1Change(Sender: TObject);
begin
  ColorBox1.Selected:=m[TabControl1.TabIndex+1].cfont;
  ColorBox2.Selected:=m[TabControl1.TabIndex+1].cback;
```

```
  memo1.Font.Color:=ColorBox1.Selected;
  memo1.Color:=ColorBox2.Selected;
end;
```

Первые две строчки обеспечивают изменение выбранных цветов шрифта и фона в соответствующих компонентах ColorBox1 и ColorBox2 в соответствии с выбранным днем недели (ярлычок TabControl1). Значения соответствующих цветов берутся из массива m. Тут следует пояснить, что, как и везде в Паскале, счет ярлычков в компоненте TabControl ведется с нуля, поэтому, для приведения в соответствие номеров ярлычков с номерами элементов массива m необходимо увеличивать номер TabIndex нажатого ярлычка на единицу (m[TabControl1.TabIndex+**1**].cfont). Например, при нажатии на самый правый ярлычок (с надписью ‘Вс’) TabIndex возвращает значение 6.

Завершающие две строчки устанавливают цветовые параметры поля memo1.

Апробируйте работу программы. Очевидно, что для всех вкладок цвета будут одинаковые, так как у нас не организовано сохранение выбора пользователя в массив данных.

Сейчас *самостоятельно* допишите код так, чтобы каждый раз при изменении настроек цвета данные сохранялись в массив m в соответствующий элемент и в соответствующее поле (`cfont, cback`). Если вы смогли выполнить данную задачу, то программа стала вполне функциональной за исключением того, что при выходе из программы все настройки будут утеряны.

Осталось реализовать сохранение цветовой настройки по дням недели в ini файле.

Добавим в раздел описания глобальных переменных модуля еще одну строчку – описание файловой переменной f типа dn, описанного нами ранее:

```
Var
  Form1: TForm1;
      m: array[1..7] of dn;
      f: file of dn;
```

Теперь можно организовать работу с типизированным файлом. Каждый компонент этого файла будет представлять собой запись с двумя полями: cfont, cback: TColor.

При закрытии приложения текущие цветовые настройки должны быть сохранены в ini файл. Через инспектор объектов создайте заготовку под обработчик события OnClose и заполните следующим кодом (*это 5-ый из 5-ти разрабатываемых обработчиков*):

```
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
var i: byte;
begin
  assignfile(f,'week.ini');
  rewrite(f);
  for i:=1 to 7 do write(f,m[i]);
  closefile(f);
end;
```

Здесь все достаточно очевидно и не требует дополнительных комментариев. Апробируйте программу: измените цветовые настройки разных дней. При выходе из приложения они будут сохранены в ini файл.

Если вы сразу же запустите программу повторно, то ваши настройки будут уничтожены процедурой FormCreate, так как в ней, при запуске приложения, устанавливаются цветовые значения по умолчанию – черный и белый для всех семи

дней недели. Пришло время убрать цикл (for i:=1 to 7 do) и заменить его на цикл чтения записей из типизированного файла.

Было так:

```
procedure TForm1.FormCreate(Sender: TObject);
var i: byte;
begin
  for i:=1 to 7 do
  begin
    m[i].cfont:=clBlack;
    m[i].cback:=clWhite;
  end;
end;
```

А следует сделать так:

```
procedure TForm1.FormCreate(Sender: TObject);
var i: byte;
begin
  assignfile(f,'week.ini');
  reset(f);
  for i:=1 to filesize(f) do read(f,m[i]);
  closefile(f);
end;
```

Итак, к настоящему моменту программа уже функционально закончена и разработанные пять процедур реализуют основные задуманные возможности. Однако, до сих пор не реализовано основное предназначение ini файла: программа должна не только хранить в типизированном файле цветовые настройки, но и пользоваться ими, то есть приводить цветовые настройки поля memo1 в соответствие с цветовыми настройками дня недели. При запуске программы (процедура FormCreate) следует определять текущий день недели и устанавливать соответствующие цвет шрифта и фона поля memo1.

Текущий день недели возвращает в строковом формате стандартная функция FormatDateTime('ddd',Now); извлекая

его из переменной Now, содержащей системное время. Давайте попробуем её применить, добавив в процедуру FormCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
var i: byte;
begin
  assignfile(f,'week.ini');
  reset(f);
  for i:=1 to filesize(f) do read(f,m[i]);
  closefile(f);
  Form1.Caption:=FormatDateTime('ddd',Now);
end;
```

Апробируйте работу программы. Текущий день недели в сокращенном формате (Пн, Вт, Ср, Чт, Пт, Сб, Вс) при запуске приложения будет выводиться в заголовок главной формы.

Осталось только по дню недели определять какой порядковый номер использовать для извлечения цветовых настроек из массива m. Сделать это довольно просто, так как список дней недели в сокращенном формате уже находится в свойстве Tabs компонента TabControl1. Свойство Tabs представляет собой список строк (Пн, Вт, Ср, Чт, Пт, Сб, Вс), каждая из которых имеет свой порядковый номер и номера строк начинаются с нуля. Например, TabControl1.Tabs[0] содержит строку ‘Пн’.

Дополним процедуру FormCreate циклом, последовательно проверяющим все строки свойства Tabs компонента TabControl1 на совпадение с текущим днем недели. При совпадении цикл прерывается и имитируется нажатие на ярлычок компонента TabControl1, соответствующий текущему дню недели:

```
procedure TForm1.FormCreate(Sender: TObject);
var i: byte;
begin
  assignfile(f,'week.ini');
```

```
  reset(f);
  for i:=1 to filesize(f) do read(f,m[i]);
  closefile(f);
  Form1.Caption:=FormatDateTime('ddd',Now);
  for i:=0 to 6 do
    if TabControl1.Tabs[i]=Form1.Caption then break;
  TabControl1.TabIndex:=i;
  TabControl1Change(TabControl1);
end;
```

На этом разработку программы можно считать законченной. Создано пять несложных и небольших по объему процедур. Программа может служить шаблоном для разработки приложений, использующих в своей работе файлы конфигурации, содержащие необходимые настройки.

#3.3. Задание для самостоятельного исполнения.

Дополните разработанное приложение таким образом, чтобы от дня недели зависели не только цвета шрифта и фона компонента memo1, но и сам шрифт (например, его имя, размер или стиль).

Глава 4. Обработка табличной информации

4.1. Табличное представление данных

Для отображения и редактирования табличной информации используют стандартный компонент TStringGrid (в переводе – сетка строк) с вкладки Additional. Уже из названия типа данных компонента вы можете понять, что информация хранится в нём в строковом виде. Из этого следует, что каждый раз при добавлении, изменении или выборке не строковых данных (целое или дробное число, логический тип и т.д.) нужно будет обеспечить преобразование данных для дальнейшего использования.

Изучение технологии обработки табличной информации начнём с создания приложения по работе с таблицей умножения. Создайте новое приложение в среде Lazarus и на форме разместите компонент TStringGrid, затем через инспектор объектов настройте некоторые из его свойств:

DefaultColWidth	36	ширина колонок
DefaultRowHeight	30	высота строк
FixedCols	1	количество фиксированных колонок
FixedRows	1	количество фиксированных строк

Ширину и высоту всей таблицы поставьте по своему усмотрению, в дальнейшем мы будем динамически менять эти размеры через специально подготовленный интерфейс пользователя. Сейчас же, как вы можете заметить, количество нефиксированных строк и столбцов, которые предназначены для размещения информации, ограничено четырьмя. Давайте напишем процедуру, которая при старте приложения будет заполнять пространство компонента таблицей умножения в заданных ограничениях. Кликните дважды по форме или через

инспектор объектов создайте обработчик события FormCreate и заполните его следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
var r, c: integer;
begin
  for r:=1 to 4 do
    for c:=1 to 4 do
      StringGrid1.Cells[c, r] := IntToStr(c*r);
end;
```

Откомпилируйте и запустите программу и оцените полученный результат (рис.17).

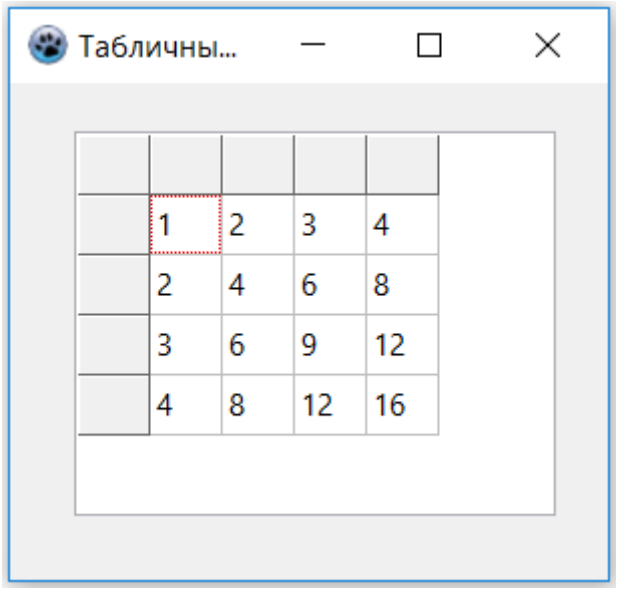

Рис.17. Базовый дизайн табличного процессора.

Вы можете самостоятельно убедиться в простоте и очевидности строчек кода – это двойной цикл (по строчкам и столбца), реализующий заполнение каждой ячейки соответствующим произведением номера столбца и номера строки с учётом необходимости перевода к строковому типу данных. Однако использование констант в шапке цикла не то чтобы недопустимо, но является плохим стилем в программировании. В дальнейшем нам потребуется менять размеры таблицы и

константы, проставленные в шапке цикла, будут неуместны, поэтому перепишем код в таком виде:

```
procedure TForm1.FormCreate(Sender: TObject);
var r, c: integer;
begin
  for r:=1 to StringGrid1.RowCount-1 do
    for c:=1 to StringGrid1.ColCount-1 do
      StringGrid1.Cells[c, r] := IntToStr(c*r);
end;
```

Обратите внимание, как часто повторяется идентификатор StringGrid1, к счастью, в языке Pascal есть специальный оператор (**with**) для объединения действий в подобных случаях, применим его:

```
procedure TForm1.FormCreate(Sender: TObject);
var r, c: integer;
begin
  with StringGrid1 do
    for r:=1 to RowCount-1 do
      for c:=1 to ColCount-1 do
        Cells[c, r] := IntToStr(c*r);
end;
```

Пришло время немного усовершенствовать интерфейс пользователя – пусть, после старта формы, у пользователя останется возможность изменять размеры таблицы, как по горизонтали, так и по вертикали. Добавим на форму два компонента TTrackBar с вкладки CommonControls, один из которых настройте на вертикальную ориентацию (рис.18). Оба регулятора должны иметь начальное значение, равное 4, и обеспечивать диапазон изменений размеров таблицы от 1 до 10, поэтому настроим соответствующие свойства (Min=1, Max=10, Position=4). Подберите размеры таблицы и регуляторов так,

чтобы они соответствовали друг другу, в том числе, при минимальном и максимальном значениях количества строк и столбцов.

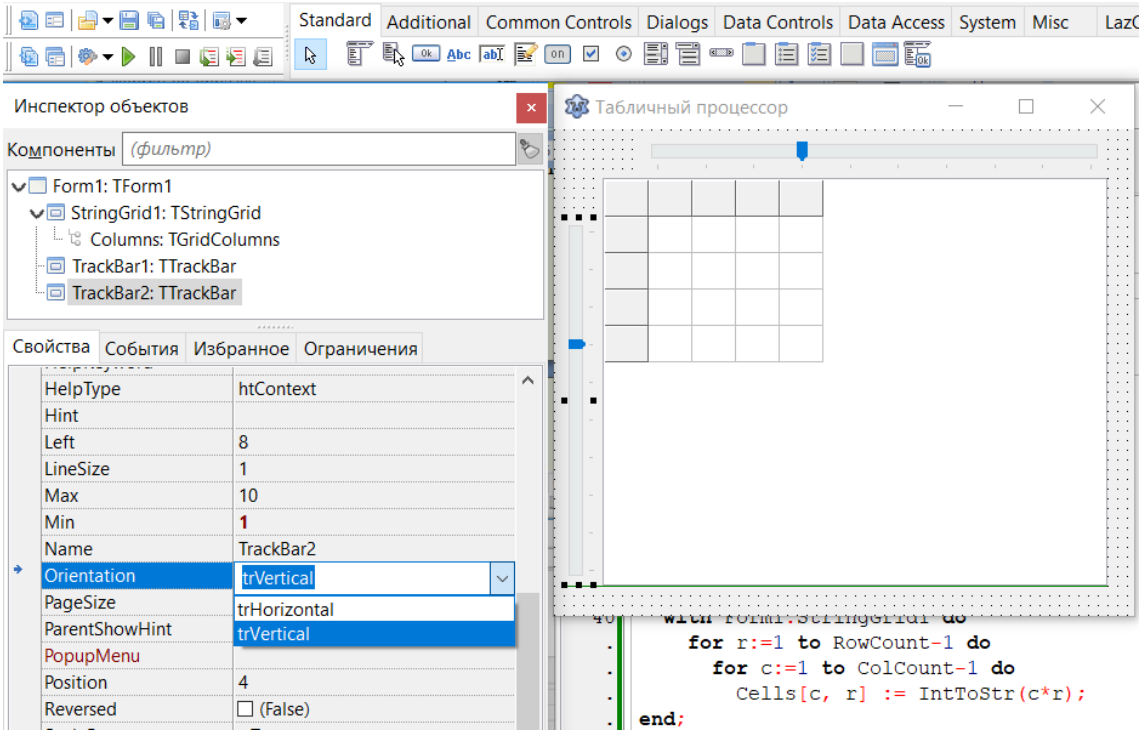

Рис.18. Настройка регуляторов размера таблицы.

Чтобы привязать размеры таблицы к позиции регуляторов, добавим соответствующие обработчики событий:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  StringGrid1.ColCount:=TrackBar1.Position+1;
end;

procedure TForm1.TrackBar2Change(Sender: TObject);
begin
  StringGrid1.RowCount:=TrackBar2.Position+1;
end;
```

Апробируйте корректность работы новых процедур и убедитесь, что размеры таблицы действительно меняются, но, при этом, само содержание ячеек таблицы не задействовано.

Напомню, что заполнение таблицы в нашем приложении осуществляется только в момент старта формы. Можно переписать эти строчки ещё два раза в каждую из которых добавить код по заполнению ячеек значениями таблицы умножения, но это будет экстенсивный путь написания текста программы. В данном случае имеет смысл выделить отдельную процедуру (**fillTable()**) по заполнению таблицы значениями и вызывать её в необходимый момент – при старте формы и при работе обоих регуляторов:

```
procedure fillTable();
var r, c: integer;
begin
  with Form1.StringGrid1 do
    for r:=1 to RowCount-1 do
      for c:=1 to ColCount-1 do
        Cells[c, r] := IntToStr(c*r);
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  fillTable();
end;

procedure TForm1.TrackBar1Change(Sender: TObject);
begin
  StringGrid1.ColCount:=TrackBar1.Position+1;
  fillTable();
end;

procedure TForm1.TrackBar2Change(Sender: TObject);
begin
  StringGrid1.RowCount:=TrackBar2.Position+1;
  fillTable();
end;
```

В рамках разработки интерфейса пользователя при работе с табличными данными очень важно предоставить пользователю возможность манипулирования указателем мыши. Давайте попробуем разработать процедуру подсчета суммы ячеек таблицы по событию выделение мышкой, а результат

будем выводить в специальный компонент, не заслоняющий основные элементы формы – TStatusBar с вкладки Common-Controls. Поместите StatusBar1 на форму и добавьте к нему в Редакторе панелей (вызывается через контекстное меню правой клавишей мыши) пару панелей. Сами панели – это объекты, которые размещаются в массиве StatusBar1.Panels. У панелей есть набор свойств, из которых самостоятельно настройте ширину отображаемого поля (свойство Width) и выравнивание текста внутри него (свойство Alignment). Сгенерируйте процедуру обработки события изменения выделения на нашей таблице и заполните её следующим содержимым:

```
procedure TForm1.StringGrid1Selection(Sender: TObject;
aCol, aRow: Integer);
var summa, numRow, numCol: integer;
begin
  summa:=0;
  with StringGrid1, Selection do
    for numRow:=Top to Bottom do
      for numCol:=Left to Right do
        summa:=summa+StrToInt(Cells[numCol,numRow]);
  StatusBar1.Panels[0].Text:=summa.ToString();
end;
```

Обратите внимание на следующие особенности приведённого выше кода:

– оператор объединения (**with**) можно использовать для нескольких объектов, включая и вложенные;

– поле текст панели строковое, поэтому перед размещением результата мы должны сумму привести к соответствующему типу данных (**summa.ToString()**).

Апробируйте программу и добейтесь корректности её работы.

На текущий момент настройки нашей таблицы таковы, что при перемещении по ней нажатого указателя мыши происходит выделение ячеек. Но такой алгоритм работы не всегда

удобен, например, если пользователю необходимо мышкой перемещать значения ячеек по таблице. Для реализации такой задачи необходимо подключить два обработчика событий: OnMouseDown и OnMouseUp. К первому подключим процедуру для запоминания начальной позиции, ко второму – процедуру для смены значений местами:

```
procedure TForm1.StringGrid1MouseDown(Sender: TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Inte-
ger);
begin
  StringGrid1.MouseToCell(X,Y,tmpC,tmpR);
  // переводит координаты X Y в столбцы и колонки
end;

procedure  TForm1.StringGrid1MouseUp(Sender:  TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Inte-
ger);
var tmpValue: string;
begin
  with StringGrid1 do
  begin
    tmpValue:=Cells[tmpC,tmpR];
    Cells[tmpC,tmpR]:=Cells[Col,Row];
    Cells[Col,Row]:=tmpValue;
  end;
end;
```

Обратите внимание, что, так как переменные tmpC и tmpR используются в разных процедурах, то их следует объявить в разделе глобальных переменных:

```
var
  Form1: TForm1;
  tmpC, tmpR: integer;
```

Однако, если в таком виде вы попробуете испытать программу, то результат будет не совсем удовлетворительный, так как мы не отменили процесс выделения ячеек таблицы при

зажатой мышке. Для управления этим свойством предназначено поле Options компонента StringGrid. Если в инспекторе объектов значение параметра goRangeSelect перевести в состояние False (по умолчанию оно равно True), то выделение прямоугольной области ячеек при зажатой клавиши мыши будет отменено. Но нас не устроит вариант настройки через инспектор объектов, так как он происходит до запуска программы и не позволяет подстраиваться под запросы пользователя. Следует поступить иначе – включить обработку запросов пользователя непосредственно в код процедур обработчиков событий. Например, если пользователь нажимает и двигает мышкой по таблице с зажатой клавишей Ctrl, тогда будет выполняться функция перемещения значений ячеек таблицы, а, если без клавиши Ctrl, то в обычном режиме – выделение ячеек и подсчёт суммы их значений.

Чтобы динамически (во время работы программы) менять возможность выделения ячеек, нужно работать со свойством Options. Данное свойство является множеством, поэтому и работать с ним можно как множеством, то есть добавлять или удалять его элементы (например, элемент goRangeSelect). В обработчике движения мыши, при запросе пользователя (при нажатой клавише Ctrl) из множества Options будем вычитать подмножество, состоящее из одного элемента – [goRangeSelect]. А, в ранее написанный обработчик отпускания мыши, добавим строчку кода по возвращению элемента goRangeSelect обратно, а также анализ зажатой клавиши Ctrl (`if ssCtrl in Shift then`):

```
procedure  TForm1.StringGrid1MouseMove(Sender:  TObject;
Shift: TShiftState; X, Y: Integer);
begin
  if ssCtrl in Shift
    then StringGrid1.Options:=
         StringGrid1.Options-[goRangeSelect];
```

```
end;

procedure  TForm1.StringGrid1MouseUp(Sender:  TObject;
Button: TMouseButton; Shift: TShiftState; X, Y: Inte-
ger);
var tmpValue: string;
begin
  if ssCtrl in Shift then
    with StringGrid1 do
    begin
      tmpValue:=Cells[tmpC,tmpR];
      Cells[tmpC,tmpR]:=Cells[Col,Row];
      Cells[Col,Row]:=tmpValue;
    end;
  StringGrid1.Options:=
  StringGrid1.Options+[goRangeSelect];
end;
```

#4.1. Задания для самостоятельного исполнения.

Дополните разработанное приложение таким образом, чтобы при работе в режиме выделения пользователь мог делать выбор (через компонент RadioGroup), что именно ему нужно получить от выделенного фрагмента: сумму, среднее арифметическое, минимальный/максимальный элемент.

Доработайте процедуру (`StringGrid1MouseUp`) замены значений ячеек таблицы так, чтобы при включённом компоненте CheckBox (его нужно добавить на форму самостоятельно) значения менялись местами как описано ранее, а при выключенном – значение, выбранное при событии OnMouseDown переносилось в новую ячейку, но, при этом, оставалось также и на месте (в своей первоначальной ячейке).

4.2. Построение графиков функций

Для наглядного представления статистической, финансовой и иной информации, представленной множеством значений анализируемого параметра, часто прибегают к построению диаграмм последовательностей или графиков функций. В Lazarus для реализации данной задачи предусмотрен специальный компонент – TChart с вкладки Chart.

Разработаем приложение, реализующее заполнение таблицы значениями функции sin(x) в заданном диапазоне и построение на основе полученных значений графика функции с последующим сохранением его в графический файл. Разместите на форме компоненты: StringGrid1, Chart1 и BitBtn1. Таблицу настройте так, чтобы у неё было только три строки (номера столбцов, значения X, значения Y) (рис.19).

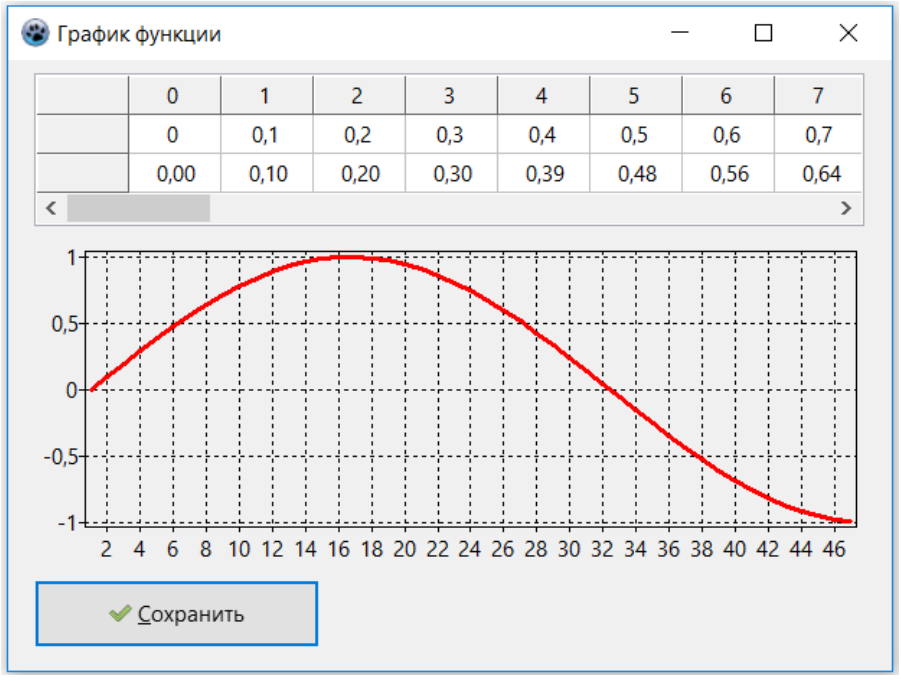

Рис.19. Форма для демонстрации графиков функции.

Для того чтобы настроить объект Chart1, кликните по нему правой клавишей мыши и в открывшемся контекстном

меню выберите опцию «Редактор диаграмм» (рис.20, а). В самом редакторе объектов следует настроить вид отображаемой диаграммы, в нашем случае (для построения синусоиды) удобнее всего подойдёт «График» (рис.20, б). После выбора вида диаграммы вы увидите в редакторе объектов новую добавленную строчку Chart1LineSeries1. Новая добавленная серия позволит вам отобразить одну линию графика, если же необходимо на одной оси координат расположить несколько линий, то следует их добавить аналогичным образом через Редактор диаграмм. Каждую серию можно настроить индивидуально по цвету, толщине линии и иным параметрам (рис.21).

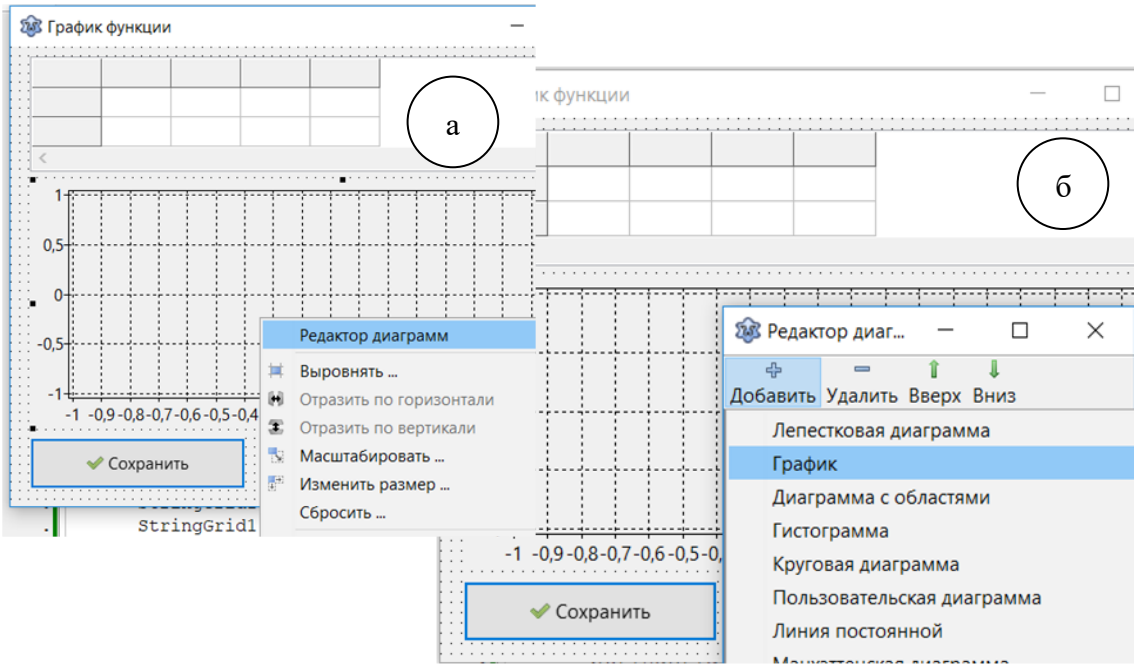

Рис.20. Вызов (а) и настройка (б) Редактора диаграмм.

Экранная клавиша «Сохранить» необходима для реализации сохранения построенного графика функции в графический файл, для чего вполне подойдёт метод (SaveToBitmapFile):

```
Chart1.SaveToBitmapFile('ris.bmp').
```

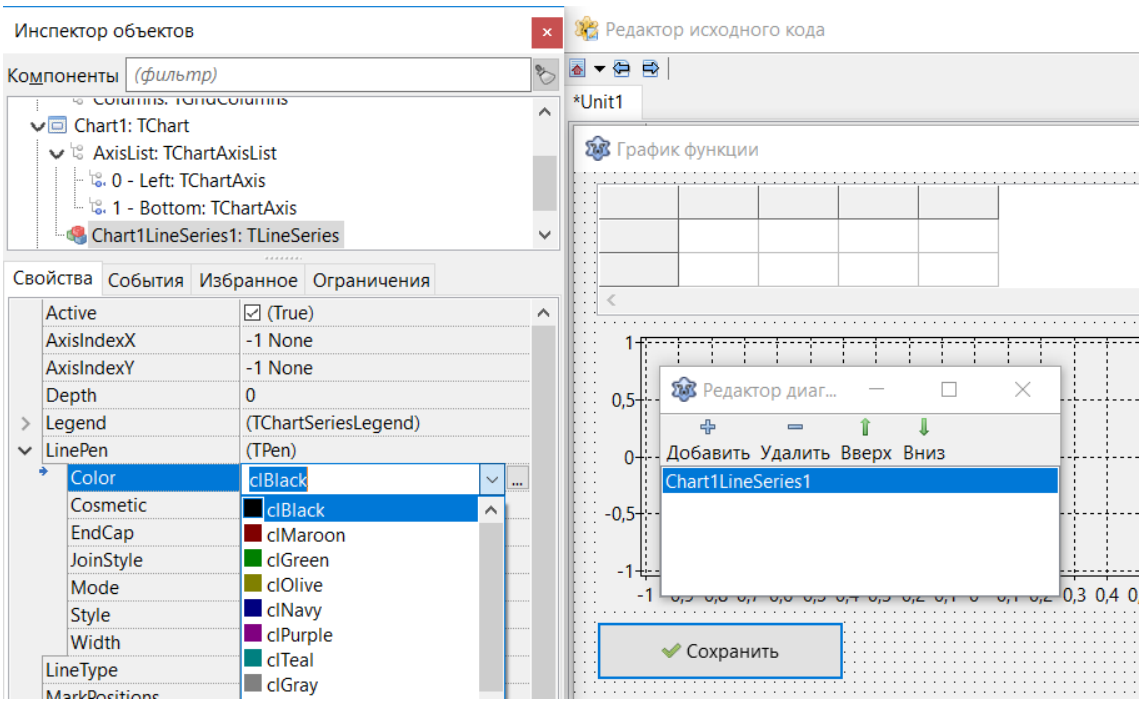

Рис.21. Настройка одной серии значений.

Только после предварительной настройки всех визуальных компонентов следует приступать к проектированию логики работы приложения. Прежде всего, определимся со структурой для хранения координат точек графика, для чего опишем соответствующий пользовательский тип данных (`xyValue`) и продекларируем в разделе глобальных переменных динамический массив для хранения значений данного типа:

```
Type
    xyValue = record // пользовательский тип данных
    x, y: real;
end;

var
  Form1: TForm1;
  arrXY: array of xyValue; // массив для хранения значений
```

Порядок работы приложения такой:

1) заполнение динамического массива координатами точек будущего графика,

```
procedure fillArray(); // заполнение массива
var x, y, step, maxX: real;
    tmp: xyValue;
begin
  SetLength(arrXY,0);
  x:=0; // начальное значение X
  step:=0.1; // шаг изменения X
  maxX:=1.5*Pi; // максимальное значение X
  while (x+step)<=maxX do
  begin
    y:=sin(x);
    tmp.x:=x; tmp.y:=y;
    SetLength(arrXY,(Length(arrXY)+1));
    arrXY[Length(arrXY)-1]:=tmp;
    x:=x+step;
  end;
end;
```

2) заполнение таблицы значениями точек графика для удобства пользователя,

```
procedure fillTable(); // заполнение таблицы
var i: integer;
begin
  with Form1 do
  begin
    StringGrid1.ColCount:=Length(arrXY)+1;
    for i:=0 to Length(arrXY)-1 do
    begin
      StringGrid1.Cells[i+1,0]:=i.ToString();
      StringGrid1.Cells[i+1,1]:=FloatToStr(arrXY[i].x);
      StringGrid1.Cells[i+1,2]:=
                  Format('%5.2f',[arrXY[i].y]);
    end;
  end;
end;
```

3) построение графика функции по точкам из массива значений.

```
procedure fillChart(); // прорисовка графика по точкам
var i: integer;
    c: TColor;
begin
  c:=RGBtoColor(255,0,0);
  with Form1 do
```

```
  begin
    Chart1LineSeries1.Clear();
    Chart1LineSeries1.LinePen.Width:=3;
    Chart1LineSeries1.LinePen.Color:=c;
    StringGrid1.ColCount:=Length(arrXY)+1;
    for i:=0 to Length(arrXY)-1  do
      Chart1LineSeries1.Add(arrXY[i].y,'',c);
  end;
end;
```

Все описанные процедуры запускаются не сами по себе, а при наступлении какого-то события, назначенного программистом. В данном случае в качестве такого события можно выбрать запуск приложения с созданием формы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  fillArray(); // заполнение массива
  fillTable(); // заполнение таблицы
  fillChart(); // прорисовка графика по точкам
end;
```

В качестве развития способов реализации интерфейса отметим, что настройку компонентов, включая и таблицу, можно производить динамически, то есть прямо во время работы приложения:

```
Procedure TForm1.StringGrid1PrepareCanvas(sender:
TObject; aCol, aRow: Integer; aState: TGridDrawState);
var MyTextStyle: TTextStyle;
begin // установим выравнивание по центру в ячейках таблицы
  MyTextStyle:=StringGrid1.Canvas.TextStyle;
  MyTextStyle.Alignment:=taCenter;
  StringGrid1.Canvas.TextStyle:=MyTextStyle;
end;
```

В развитом интерфейсе прикладной программы пользователю не нужно делать излишних движений, например, в данной программе имеет смысл сразу после заполнения таблицы

и построения графика функции фокус ввода переместить на экранную клавишу BitBtn1. В этом случае пользователю уже не обязательно именно мышкой кликать по ней, достаточно нажать «Пробел» или «Enter» для реализации функции сохранения графика в файл. Этот функционал можно погрузить в процедуру, запускаемую по событию «показ формы»:

```
procedure TForm1.FormShow(Sender: TObject);
begin
  BitBtn1.SetFocus();
  // кнопку "Сохранить" поместим в фокус
end;
```

#4.2. Задания для самостоятельного исполнения

Пусть есть текстовый файл с рейтингом популярности лучшей десятки языков программирования. В файле десять строчек, в каждой из которых сначала число, а затем, через символ табуляции, название языка программирования:

```
15.568%  Java
6.966%   C
4.554%   C++
3.579%   C#
3.457%   Python
3.376%   PHP
3.251%   Visual Basic
2.851%   JavaScript
2.816%   Delphi/Object Pascal
2.413%   Perl
```

Число обозначает популярность, вычисленную как отношение количества запросов пользователей в поисковике google.com про данный язык к общему количеству запросов про все языки программирования выраженную в процентах.

Разработайте приложение, которое будет загружать данные из текстового файла в таблицу, отсекая символ «%», и, затем, строить по ним гистограмму (это диаграмма из вертикальных столбиков) с возможностью её последующего сохранения в графический файл.

Глава 5. Динамические компоненты

До сих пор мы размещали визуальные компоненты на форме приложения ещё на этапе редактирования программы, но не редки такие ситуации, когда рациональнее добавлять компоненты в момент выполнения программы. Рассмотрим особенности работы с динамически создаваемыми компонентами в среде Lazarus.

Создайте приложение с графическим интерфейсом и на главную форму приложения разместите объект TLabel. Если теперь кликнуть по нему дважды мышкой, то автоматически сгенерируется код для обработки события «клик мышкой». В инспекторе объектов в свойствах объекта label1 вы можете настроить его визуальные характеристики (рис.__). Так, обычно, настраиваются статические компоненты, то есть те, которые присутствуют на форме ещё до её создания.

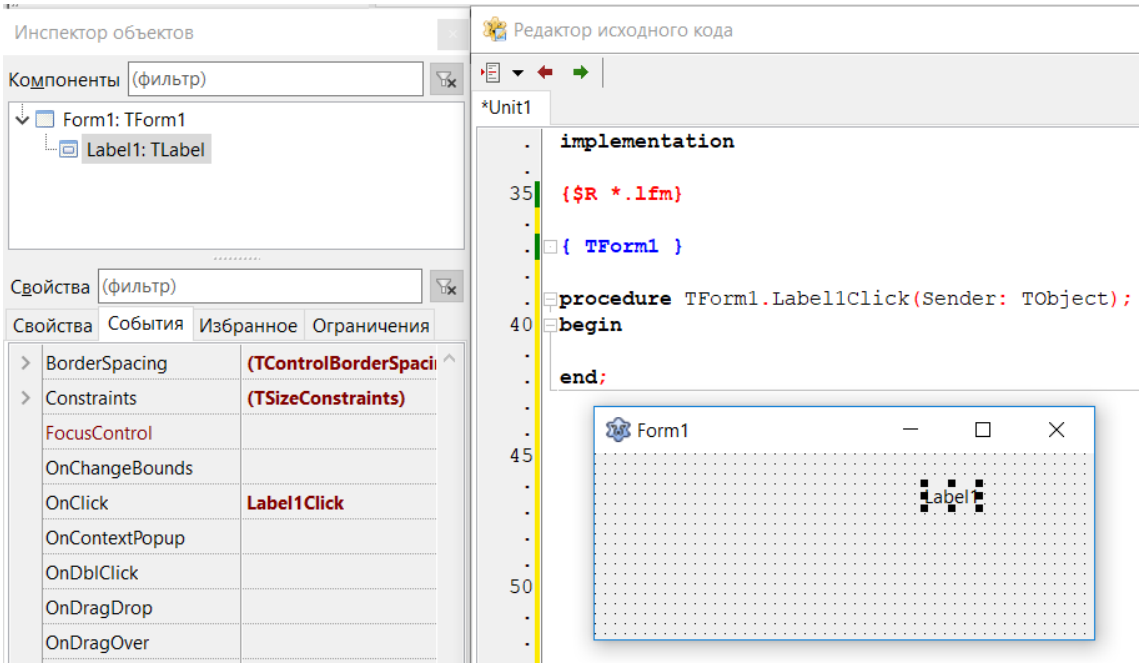

Рис.22. Настройка статического компонента.

Но всё это вы делаете ещё до запуска программы, когда не известно, сколько именно элементов управления нужно расположить на форме.

Напишем приложение с возможностью добавлять компоненты на форму по желанию пользователя, настроим эти компоненты и подключим к ним динамически обработчики событий такие, что можно будет мышкой взять любой из добавленных компонентов и передвинуть в новое место на форме.

Шаг первый. Рассмотрим работу с одним компонентом.

Пусть в момент старта формы создаётся компонент типа TLabel:

```
var // раздел глобальных переменных
   Form1: TForm1;
   tx, ty: Integer;
   lbl: TLabel;

procedure TForm1.FormCreate(Sender: TObject);
begin
  lbl := TLabel.Create(self);
  lbl.Name := 'Label_01';
  lbl.Caption := 'Label_01';
  lbl.Top := 15; lbl.Left := 15;
  lbl.OnMouseDown := @label1MouseDown;
  lbl.OnMouseMove := @label1MouseMove;
  lbl.Parent:=self;
end;
```

Далее программным образом подключаем обработчики событий для «взятия» компонента мышкой и перемещения его по форме:

```
procedure TForm1.Label1MouseDown(Sender: TObject; But-
ton: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  tx := x; ty := y;
end;

procedure    TForm1.Label1MouseMove(Sender:    TObject;
Shift: TShiftState; X,
  Y: Integer);
```

```
begin
  if Shift=[ssLeft] then
  begin
    lbl.Left := lbl.Left+X-tx;
    lbl.Top := lbl.Top+Y-ty;
  end;
end;
```

Давайте доработаем программу так, чтобы можно было добавлять несколько компонентов и всеми ими управлять во время работы приложения:

```
const count = 5;
var // раздел глобальных переменных
   Form1: TForm1;
   tx, ty: Integer;
   lbl: TLabel;
   albl: array[0..count] of TLabel;

procedure TForm1.FormCreate(Sender: TObject);
var i: Integer;
begin
 for i:=0 to count do
 begin
   albl[i]:=TLabel.Create(self);
   albl[i].Name:='Label_0'+IntToStr(i);
   albl[i].Caption:='Label_0'+IntToStr(i);
   albl[i].Top:=15+25*i; albl[i].Left:=15;
   albl[i].OnMouseDown:=@label1MouseDown;
   albl[i].OnMouseMove:=@label1MouseMove;
   albl[i].Parent:=self;
 end;
end;

procedure    TForm1.Label1MouseMove(Sender:    TObject;
Shift: TShiftState; X,
  Y: Integer);
begin
  if Shift=[ssLeft] then
  begin
```

```
    (Sender as TLabel).Left:=(Sender as TLabel).Left+X-
tx;
    (Sender as TLabel).Top:=(Sender as TLabel).Top+Y-ty;
  end;
end;
```

Для реализации подхода работы со множеством динамических компонентов в данном варианте мы использовали статический массив. Апробируйте работу программу, все динамически созданные компоненты должны иметь возможность передвигаться мышкой (рис.23).

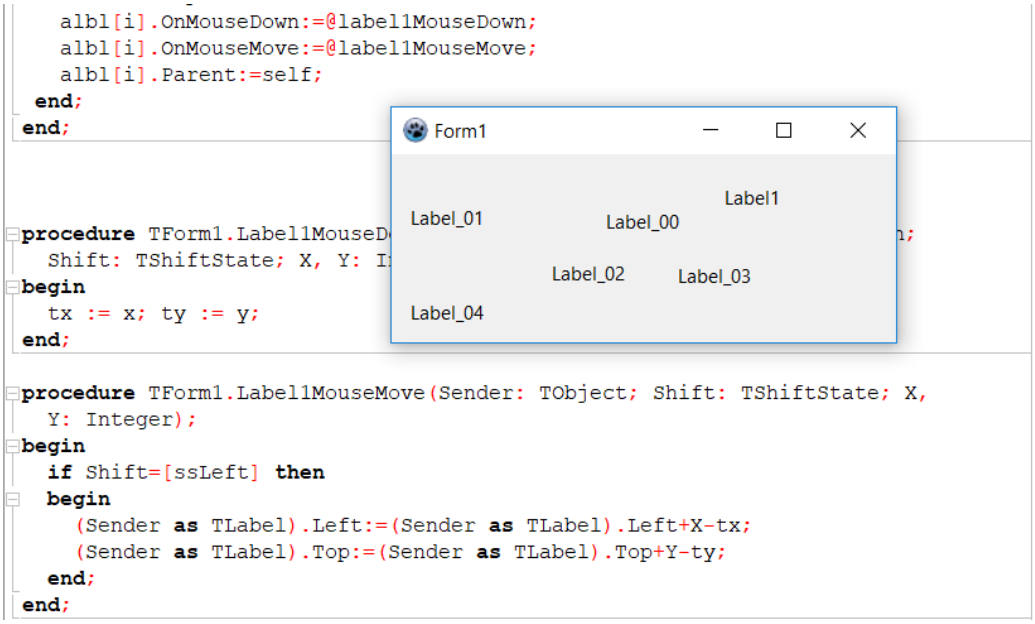

Рис.23. Пример работы с динамически созданными компонентами.

Однако мы до сих пор не можем во время работы программы менять количество компонентов на форме, мы определяем это число ещё на стадии написания программного кода. Данное затруднение можно преодолеть при замене статического массива, на динамический:

```
var // раздел глобальных переменных
   Form1: TForm1;
```

```
   tx, ty, count: Integer;
   albl: array of TLabel;

procedure TForm1.FormCreate(Sender: TObject);
begin
 count:=0; SetLength(albl,count);
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i: integer;
begin
  if Shift=[ssMiddle] then
  begin
    Inc(count); i:=count-1;
    SetLength(albl,count);
    albl[i]:=TLabel.Create(self);
    albl[i].Name:='Label_0'+IntToStr(i);
    albl[i].Caption:='Label_0'+IntToStr(i);
    albl[i].Top:=15+25*i; albl[i].Left:=15;
    albl[i].OnMouseDown:=@label1MouseDown;
    albl[i].OnMouseMove:=@label1MouseMove;
    albl[i].Parent:=self;
  end;
end;
```

Теперь при каждом клике средней клавишей мыши по форме приложения будет добавляться новый компонент типа TLabel. Все добавленные компоненты будут доступны через динамический массив и будут привязаны к обработчикам событий для управления мышкой.

Для полноценной работы с динамическими компонентами нужно научиться не только их добавлять на форму, но и корректно их убирать. Добавьте в обработчик события FormMouseDown условный оператор, который будет при нажатии правой клавиши мыши убирать из списка последний компонент:

```
if Shift=[ssRight] then
begin
  Dec(count); albl[count].Free;
  SetLength(albl,count);
end;
```

Задание для самостоятельного исполнения.

Доработайте программу так, чтобы можно было с формы удалять указанный компонент, а не с конца списка, как было показано в примере выше. Сделайте так, чтобы клик средней клавишей мыши по компоненту приводил к его удалению.

Глава 6. Динамические библиотеки

Динамически подключаемая библиотека или DLL (от англ. Dynamic Link Library – дословно «библиотека динамической компоновки») в операционной системе Microsoft Windows библиотека подпрограмм, допускающая своё использование различными программными приложениями одновременно. К DLL относятся также элементы управления ActiveX и драйверы.

Рассмотрим преимущества от использования динамически подключаемых библиотек:

– эффективное использование ресурсов оперативной памяти и снижение объема расходуемого дискового пространства, за счет использования одного экземпляра библиотечного модуля для различных приложений;

– повышение эффективности сопровождения и обновления программных продуктов за счёт их модульности, устранение «багов» (ошибок кода, выявленных уже после окончания проектирования) и обновление или наращивание приложений

путем замены динамически подключаемых библиотек с одной версии на другую;

– динамические библиотеки могут использоваться разнотипными приложениями от одного или даже разных производителей – например, Microsoft Office и Microsoft Visual Studio.

Для полноценного освоения данной темы вам необходимо:

– вспомнить особенности организации подпрограмм вида процедуры и функции,

– уточнить технологию модульного проектирования приложений,

– освоить порядок проектирования динамических библиотек.

ВНИМАНИЕ!

Если у вас есть затруднения в понимании модульной организации приложений – обновите свои знания, используя методической пособие к лабораторной работе №1 События мыши (https://pcoding.ru/delphi/labrab/labrab1.pdf), первый вопрос которой посвящен обсуждению модульной организации приложений.

Если у вас есть затруднения в понимании особенностей декомпозиции цельного программного кода на подпрограммы – обновите свои знания, используя презентацию к лекции №4 «Организация подпрограмм» (https://pcoding.ru/delphi/prez/prez_podprogr.swf), в которой рассматриваются следующие вопросы:

– вид и структура подпрограмм;
– параметры подпрограмм;
– глобальные и локальные переменные;
– доступ к подпрограммам;
– рекурсия и досрочный выход;
– опережающее описание процедур;
– перегрузка подпрограмм;
– процедурный тип данных.

Дополнительно можно восполнить базовые понятия о функциях и процедурах из презентации к лекции «Подпрограммы» для дисциплины «Информатика и программирование» – https://pcoding.ru/algopro/lek/lek11.htm .

Пример организации процедуры и функции.

Разработаем рекурсивные подпрограммы вычисления суммы чисел от 1 до N.

Напомню, что рекурсивная подпрограмма сама себя вызывает. Корректная рекурсия должна обладать двумя особенностями:

– иметь точку останова;

– менять значения аргументов с каждой итерацией.

Пользователь задает целое число N. Рекурсию запустим от N до 1, поэтому точкой останова будет проверка равенства текущего аргумента единице и на каждом шаге рекурсии аргумент будет уменьшаться на единицу.

Далее приведены примеры функции и процедуры. Исходное значение находится в первом текстовом поле, а результат, после вычисления, помещается во второе.

```
function sum_f(n: Byte): Word;
begin
  if n=1
    then result:=1
    else result:=sum_f(n-1)+n;
end;

procedure sum_p(n: Byte; var r: Word);
begin
  if n=1
    then r:=1
    else begin sum_p(n-1,r); r:=r+n; end;
end;

procedure TForm1.btn1Click(Sender: TObject);
begin
  edt2.Text:=IntToStr(sum_f(StrToInt(edt1.Text)));
end;

procedure TForm1.btn2Click(Sender: TObject);
var r: Word;
begin
  sum_p(StrToInt(edt1.Text),r);
  edt2.Text:=IntToStr(r);
end;
```

Используются сокращенные названия компонентов.

Создайте приложение и апробируйте работу рекурсивных подпрограмм.

Пример размещения процедуры и функции в модуле

Теперь перенесем рекурсивные подпрограммы в другой модуль. Создайте модуль Unit2 (меню File / New / Unit) и сохраните его в той же папке, что и само приложение с первым модулем. Сохранение приложения со всеми модулями является обязательным. Ниже приведено исходное содержание второго модуля:

```
unit Unit2;

interface

implementation

end.
```

В интерфейсном разделе следует декларировать подключаемые модули, пользовательские типы данных, константы и переменные, заголовки процедур и функций. В разделе реализаций описываются сами подпрограммы.

Перенесите разработанные ранее подпрограммы во второй модуль:

```
unit Unit2;

interface
  function sum_f(n: Byte): Word;
  procedure sum_p(n: Byte; var r: Word);

implementation

function sum_f(n: Byte): Word;
begin
  if n=1
```

```
    then result:=1
    else result:=sum_f(n-1)+n;
end;

procedure sum_p(n: Byte; var r: Word);
begin
  if n=1
    then r:=1
    else begin sum_p(n-1,r); r:=r+n; end;
end;

end.
```

Первый модуль сократите, но не забудьте произвести подключение второго модуля:

```
unit Unit1;

interface

uses
  Windows,  Messages,  SysUtils,  Variants,  Classes,
Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    edt1: TEdit;
    edt2: TEdit;
    btn1: TButton;
    btn2: TButton;
    procedure btn1Click(Sender: TObject);
    procedure btn2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
uses  Unit2;   // это обязательно !!!

{$R *.dfm}

procedure TForm1.btn1Click(Sender: TObject);
begin
  edt2.Text:=IntToStr(sum_f(StrToInt(edt1.Text)));
end;

procedure TForm1.btn2Click(Sender: TObject);
var r: Word;
begin
  sum_p(StrToInt(edt1.Text),r);
  edt2.Text:=IntToStr(r);
end;

end.
```

Итак, значимые подпрограммы мы вынесли в отдельный модуль, который уже можно совершенствовать и заменять независимо от основной программы. Однако его всё ещё нужно компилировать совместно с программным кодом основной программы.

Пример размещения процедуры и функции в динамической библиотеке

Сохраните все изменения и закройте приложение. Через меню создайте новую заготовку под библиотеку: меню Файл/Создать/Библиотека (рис.21). В созданной пока пустой библиотеке вы увидите комментарий, который гласит буквально следующее:

> Если динамическая библиотека в процессе работы использует переменные или функции, осуществляющие динамическое выделение памяти под собственные нужды (длинные строки, динамические массивы, функции New и GetMem), а также, если такие переменные передаются в параметрах и возвращаются в результатах, то в таких библиотеках обязательно должен использоваться модуль ShareMem. При этом в секции uses модуль должен располагаться на первом месте. Управление этими операциями осуществляет

специальная библиотека BORLANDMM.DLL, которая должна распространяться вместе с разрабатываемыми динамическими библиотеками, использующими модуль ShareMem.

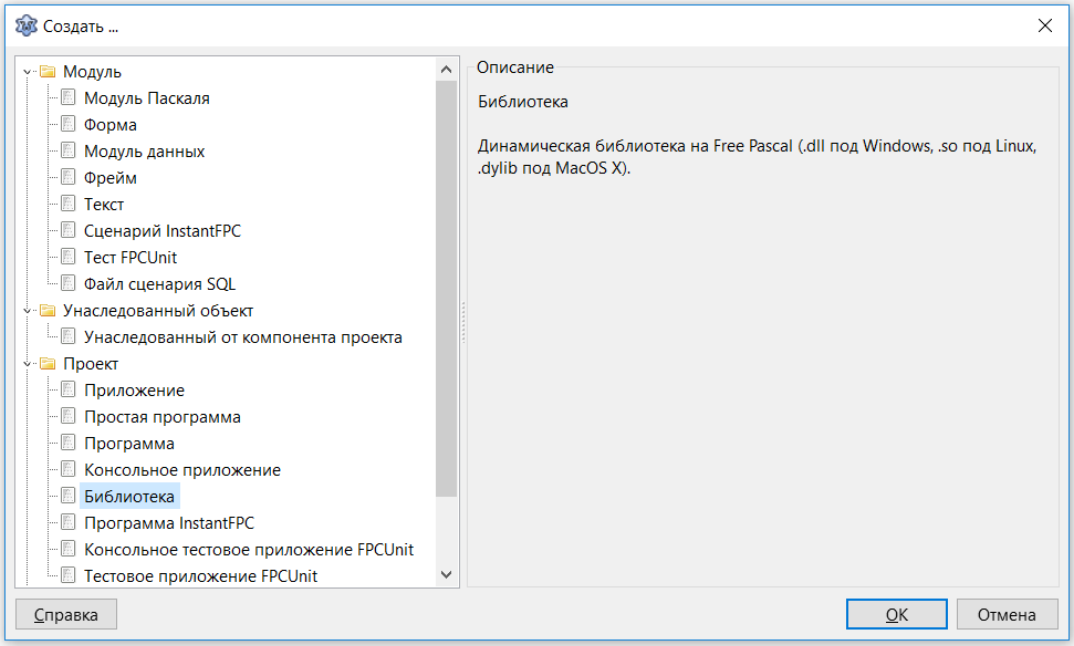

Рис.24. Диалоговое окно Lazarus для создания библиотеки.

В нашей библиотеке мы пока не используем указанные особенности, поэтому список подключаемых модулей оставим без изменений, комментарий уберем, а наши рекурсивные подпрограммы вставим:

```
library P_DLL;

uses
  // ShareMem этот модуль нужно ставить тут,
  // но сейчас он не нужен
  SysUtils,
  Classes;

{$R *.res}

function sum_f(n: Byte): Word;
begin
  if n=1
```

```
    then result:=1
    else result:=sum_f(n-1)+n;
end;

procedure sum_p(n: Byte; var r: Word);
begin
  if n=1
    then r:=1
    else begin sum_p(n-1,r); r:=r+n; end;
end;

exports
  sum_f, sum_p;

begin
end.
```

Обратите внимание на новый раздел exports, он нужен для того чтобы подпрограммы библиотеки были доступны во внешних приложениях. Сохраните текущий проект в ту же папку, что и основное приложение под интуитивно понятным именем (например, P_DLL). Откомпилируйте динамическую библиотеку нажав CTRL+F9 и проверьте, что в текущей папке появился новый файл P_DLL.dll уже готовый к использованию сторонними программами. Проверим это, для чего закройте редактируемую библиотеку и откройте наш основной проект. На текущий момент в нем два обработчика события (две процедуры), готовые использовать рекурсивные подпрограммы. Но пока адресация установлена на использование второго модуля. Удалите ссылку на второй модуль, но добавьте описание рекурсивных подпрограмм со ссылкой на нашу динамическую библиотеку:

```
unit Unit1;

interface

uses
```

```
  Windows,  Messages,  SysUtils,  Variants,  Classes,
Graphics, Controls, Forms, Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    edt1: TEdit;
    edt2: TEdit;
    btn1: TButton;
    btn2: TButton;
    procedure btn1Click(Sender: TObject);
    procedure btn2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

//uses  Unit2;  // это уже нужно убрать !!!

{$R *.dfm}

function sum_f(n: byte): Word;
  external 'P_DLL.dll' name 'sum_f';
procedure sum_p(n: byte; var r: Word);
  external 'P_DLL.dll' name 'sum_p';

procedure TForm1.btn1Click(Sender: TObject);
begin
  edt2.Text:=IntToStr(sum_f(StrToInt(edt1.Text)));
end;

procedure TForm1.btn2Click(Sender: TObject);
var r: Word;
begin
  sum_p(StrToInt(edt1.Text),r);
  edt2.Text:=IntToStr(r);
end;

end.
```

При наличии откомпилированной библиотеки наше приложение можно уже запустить и испытать работу рекурсивных подпрограмм.

Обратите внимание на параметр name в декларации внешней библиотеки. Он задает имя подпрограммы из библиотеки, однако в вашей программе вы можете использовать другое имя, главное, чтобы количество аргументов и их типы данных совпали.

Внимание! Подумайте над вопросом: для чего может потребоваться смена имени подпрограммы.

После внесения изменений раздел реализации может выглядеть так:

```
implementation

{$R *.dfm}

function sum_f(n: byte): Word;
  external 'P_DLL.dll' name 'sum_f';
procedure s_p(n: byte; var r: Word);
  external 'P_DLL.dll' name 'sum_p';

procedure TForm1.btn1Click(Sender: TObject);
begin
  edt2.Text:=IntToStr(sum_f(StrToInt(edt1.Text)));
end;

procedure TForm1.btn2Click(Sender: TObject);
var r: Word;
begin
  s_p(StrToInt(edt1.Text),r);
  edt2.Text:=IntToStr(r);
end;

end.
```

Внимание:

– удобнее всего обмениваться строковыми данными с динамической библиотекой через переменную типа TStringList, даже если там всего одна строка (*иначе можно столкнуться со сложностями передачи строковых*

данных, так как строковый тип String можно использовать для передачи в библиотеку, но обратно можно возвращать только через тип PChar; на странице https://pcoding.ru/delphi/faq.htm можете найти пример по передаче строки в динамическую библиотеку, обработке строки и возвращении её обратно из библиотеки в основную программу; код примера можете увидеть ниже в Приложении с поясняющими комментариями);

– для выполнения заданий вам может потребоваться подключение модуля ShareMem;

– при выполнении заданий можно не делать каждый раз новую библиотеку, достаточно просто дополнять нашу.

Теперь оцените пример оформления приложения для работы с динамической библиотекой, включающий в себя дизайн главной формы (рис.25), код динамической библиотеки и, собственно, код программы.

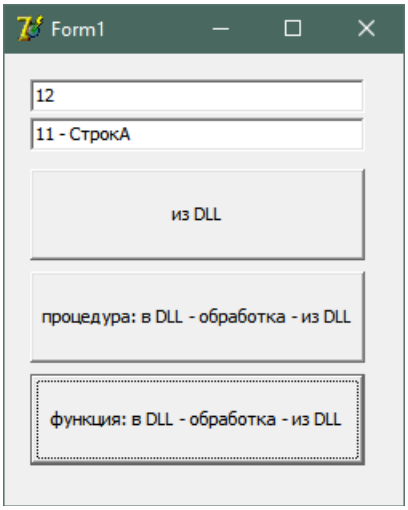

Рис.25. Главная форма приложения.

Код динамической библиотеки:

```
library P_DLL;

uses
  SysUtils,
  Classes;

{$R *.res}
const str='СтрокА';

procedure P_Str(var s: string; var pp: PChar);
var e: string; n: Integer;
begin
  n:=StrToInt(s)+1;
  e:=IntToStr(n)+' - '+str;
  pp:=PChar(e);
end;

function F_Str(s: string): PChar;
var e: string; n: Integer;
```

```
begin
  n:=StrToInt(s)-1;
  e:=IntToStr(n)+' - '+str;
  result:=PChar(e);
end;

procedure Get_Str(var p: PChar);
begin
  StrCopy(p,str);
end;

function Get_Len(var s: string): integer;
begin
  Result:=Length(str);
end;

exports
  Get_Len, Get_Str, P_Str, F_Str;

begin
end.
```

Код приложения:

```
unit Unit1;

interface

uses
  Windows,  Messages,  SysUtils,  Variants,  Classes,
Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  TForm1 = class(TForm)
    edt1: TEdit;
    edt2: TEdit;
    btn6: TButton;
    btn1: TButton;
    btn2: TButton;
    procedure btn6Click(Sender: TObject);
    procedure btn1Click(Sender: TObject);
    procedure btn2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```
var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure Get_Str(var p: PChar);
  external 'P_DLL.dll';
function Get_Len:integer;
  external 'P_DLL.dll';
procedure P_Str(var a: String; var z: PChar);
  external 'P_DLL.dll';
function F_Str(s: string): PChar;
  external 'P_DLL.dll';

procedure TForm1.btn6Click(Sender: TObject);
var p: PChar; // ссылка на символ в строке
  // PChar нужен для работы с ASCIIZ-строками
  // это строки длиной больше 255, в конце символ с кодом 0
begin
  GetMem(p,(Get_Len+1)*SizeOf(Char)); // выделяем па-
мять
  // +1 нужен на нуль-символ конца строки
  // SizeOf(Char) нужен чтобы работало и для двухбайтной
кодировки
  Get_Str(p);
  edt2.Text:=p;
  FreeMem(p);  // освобождаем память
end;

procedure TForm1.btn1Click(Sender: TObject);
var p: PChar; a: string;
begin
  a:=edt1.Text;
  P_Str(a,p);
  edt2.Text:=p;
end;

procedure TForm1.btn2Click(Sender: TObject);
begin
  edt2.Text:=F_Str(edt1.Text);
end;

end.
```

#6. Задания для самостоятельного исполнения

1. Разработайте динамическую библиотеку, содержащую рекурсивную функцию подсчета произведения нечетных чисел от 1 до N.

2. Разработайте динамическую библиотеку вычисления корней квадратного уравнения (в библиотеку передаются коэффициенты a, b, c; обратно возвращаются x1, x2, строка с текстом о количестве корней – через нулевой элемент переменной типа TStringList).

3. Разработайте динамическую библиотеку определения самой длинной строки в текстовом файле. В библиотеку передается полное имя файла, обратно возвращается искомая строка (через нулевой элемент переменной типа TStringList).

4. Пусть структура текстового файла такова: в каждой строке находятся Фамилия и Средний балл абитуриента, разделенные пробелом (или другим символом, по желанию). Разработайте динамическую библиотеку определения списка Фамилий абитуриентов, имеющих средний балл выше указанного пользователем. В библиотеку передается полное имя файла и средний балл, а обратно возвращается список фамилий (список строк TStringList). Результат можно вывести в поле Memo.

5. Задание на разработку программного обеспечения для коллектива из двух разработчиков. Каждый делает свою часть работы: либо интерфейс пользователя, либо динамическую библиотеку, но во время сдачи работы и защиты проекта оба разработчика должны уметь объяснить код своего коллеги…

> Обычно front-end и back-end development употребляют по отношению к web-разработкам – это клиентская и серверная части. Однако, в общем смысле, эти термины используются в программной инженерии при разработке любого сложного программного продукта, когда нужно разделить представительский уровень (GUI – графический интерфейс пользователя, то, что он видит и нажимает) и уровень доступа к файлам (работа с данными в файлах, динамические библиотеки). Такое разделение оправдано по нескольким причинам:

– упрощает разработку, последующее тестирование, улучшения;

– позволяет разбить задачу между исполнителями – независимая и параллельная разработка;

– можно кодировать на разных языках программирования, лучше подходящих для выполнения своей части задачи.

Техническое задание.

Разработать приложение, которое читает текстовый файл с рейтингом языков программирования (только первые двадцать по рейтингу).

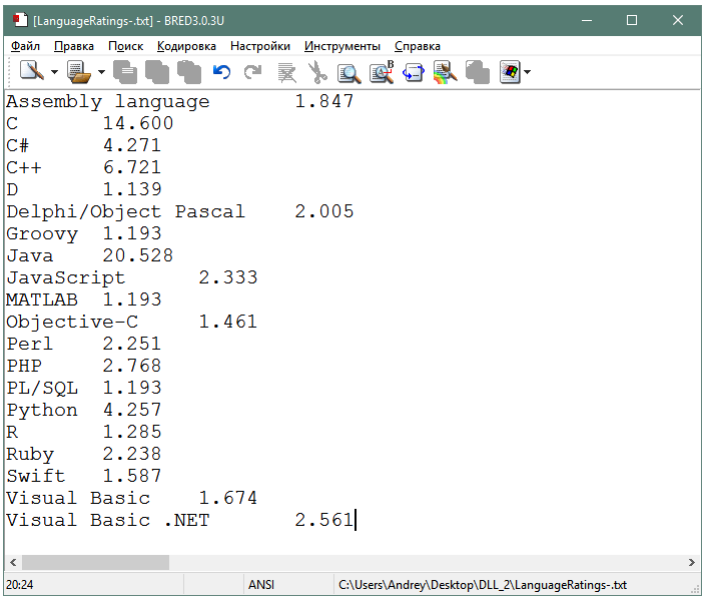

Рис.26. Формат входного потока данных.

Описание формат входного файла:

– в каждой строке информация об одном языке программирования;

– сначала идет название языка потом через символ табуляции его рейтинг в процентном отношении.

Файл неудобен для чтения и анализа по двум причинам:

– информация отсортирована по названию языка программирования;

– длина названия языков разная, поэтому второй столбец, хоть и через символ табуляции, не везде находится на одной и той же позиции.

Реализовать функционал программы:

– отсортировать по рейтингу;

– поменять столбцы местами для удобства чтения (между столбцами символ табуляции).

Рис.27. Требуемый вид интерфейса программы.

Обратите внимание на возможный внешний вид приложения.

Один студент (*front-end developer*) делает само приложение, обеспечивает открытие текстового файла через диалог пользователя (OpenDialog), считывание его в исходном виде в поле Memo, отправку исходного содержимого в динамическую библиотеку в виде TStringList, прием из динамической библиотеки уже в отсортированном по рейтингу виде с поменянными местами столбцами также в виде TStringList, сохранение в файл через диалог пользователя (SaveDialog).

Другой студент (*back-end developer*) осуществляет, возможно, самую сложную часть проекта, разработку динамической библиотеки. Подпрограмма библиотеки получает список строк (языки программирования с рейтингами через табуляцию), сортирует его по рейтингу и возвращает со столбцами поменянными местами (сначала рейтинг, потом символ табуляции, потом название языка.)

Заключение

В учебном пособии рассмотрены основные вопросы разработки программных приложений с графическим интерфейсом пользователя в среде быстрой разработки приложений Lazarus. Среда Lazarus постоянно совершенствуется, является открытой и свободно-распространяемой. В качестве аппаратной целевой платформы может быть использована одна из наиболее распространённых операционных систем: Windows, Linux, Mac OS, Android. Важным достоинством среды Lazarus является поддержка преобразования проектов коммерческой среды Delphi, таким образом позволяя начинающим программистам использовать для подготовки бесплатный программный продукт с последующим переносом проектов, при необходимости, в платную версию компилятора.

Следует отметить, что сама среда Lazarus достаточно функциональна для того, чтобы использовать её в качестве основной платформы для разработки. Так, например, в качестве известных программных продуктов, реализованных в Lazarus можно отметить: файловый менеджер Total Commander , аудиоредактор easyMP3Gain , кросс-платформенный архиватор PeaZIP, графический редактор LazPaint, кросс-платформенный редактор программного кода Cudatext.

Практические вопросы данного учебного пособия посвящены обсуждению возможностей среды Lazarus по реализации технологий событийного программирования, модульного проектирования приложений, обработки табличной информации, работы с файлами разных типов, реализации динамической структуры интерфейса пользователя.

Библиографический список

1. Гуриков С.Р. Основы алгоритмизации и программирования в среде Lazarus: учебное пособие. – М.: ФОРУМ: ИНФРА-М, 2019. – 336 с.

2. Культин Н.Б. Основы программирования в Delphi XE. – СПб.: БХВ-Петербург, 2011. – 416 с.

3. Чиртик А.А. Программирование в Delphi. Трюки и эффекты. – СПб.: Питер, 2010. – 400с.

4. Программирование на Lazarus [Электронный ресурс] / автор курса Ачкасов В.Ю., 2018. – режим доступа http://intuit.valrkl.ru/course-1265/, свободный. – Загл. с экрана.

5. Сухарев М. Золотая книга Delphi. – СПб.: Наука и Техника, 2010. – 1040с.

6. Беляков А.Ю. Программирование сложных информационных систем: учебное пособие. – Пермь: ИПЦ «Прокростъ», 2017. – 120с.

Учебное издание

Беляков Андрей Юрьевич

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ В LAZARUS

Учебное пособие

Подписано в печать 00.00.00.
Формат 60×84$^{1}/_{16}$. Усл. печ. л. 0,00.
Тираж 00 экз. Заказ №

ИПЦ «ПрокростЪ»

Пермского государственного аграрно-технологического университета
имени академика Д.Н. Прянишникова,
614990, Россия, г. Пермь, ул. Петропавловская, 23
тел. (342) 210-35-34