МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНА МЕТАЛУРГІЙНА АКАДЕМІЯ УКРАЇНИ

Г. Г. Швачич, О. В. Овсянніков, В. В. Кузьменко, Н. І. Нєчаєва

ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Розділ «Основи програмування та конструювання
прикладного програмного забезпечення
в інтегрованому середовищі Delphi»

Затверджено на засіданні Вченої ради академії
як навчальний посібник

Дніпропетровськ НМетАУ 2007

УДК 004 (075.8)
Швачич Г.Г, Овсянніков О.В., Кузьменко В.В., Нєчаєва Н.І. Прикладне програмне забезпечення. Навчальний посібник з основ програмування та конструювання прикладного програмного забезпечення в інтегрованому середовищі Delphi – Дніпропетровськ: НМетАУ, 2007. – 58 с.

Викладені основи об’єктно-орієнтованого програмування та конструювання прикладного програмного забезпечення в інтегрованому середовищі розробки прикладного програмного забезпечення Delphi.

Призначений для студентів спеціальності 6.020100 – документознавство та інформаційна діяльність.

Іл. 18. Бібліогр.: 5 найм.

Відповідальний за випуск Г.Г. Швачич, канд. техн. наук, проф.

Рецензенти: Б. І. Мороз, д-р техн. наук, проф. (Академія таможенної Служби України)
Т. І. Пашова, канд. техн. наук, доц. (Дніпропетровський державний аграрний університет)

© Національна металургійна академія України, 2007

Тема 1 Основные конструкции языка Object Pascal

В среде *Delphi* написание любой программы начинается с создания нового проекта в интегрированной среде разработки *IDE*. Созданный в среде *Delphi* модуль образует каркас, используемый в дальнейшем для написания программного кода. Идентификатор модуля, содержит имя модуля, которое должно быть уникальным. Оператор *uses* определяет те модули, к которым разрешен доступ из исходного модуля. После оператора *uses* следует общедоступное объявление *public*. В разделе *interface* указывается только заголовок процедур и функций, а сам программный код процедур и функций находится в секции *implementation* модуля. Помимо основных определений в программном модуле часто встречается текст, заключенный в фигурные скобки. Если первый символ после фигурной скобки не является символом доллара *{$}*, то речь идет о комментарии, который не учитывается при компиляции и выполнении программы. При вставке комментария в конце строки рекомендуется использовать двойную косую черту, например:

```
{Присвоение цвета панели}
Panel.Color := clRed;          //Установить красный цвет панели
```

Все данные, используемые программой должны принадлежать к заранее определенному типу – стандартному или пользовательскому. Имена стандартных типов данных являются предопределенными идентификаторами и действуют в любой точке программы. Пользовательские типы – это дополнительные типы, которые пользователь может определить самостоятельно.

Для обеспечения адресного пространства во время компиляции программы все переменные и константы должны объявляться в начале программы или программного блока. Для этого должен быть определен тип (*type*) переменных или констант.

Простой тип определяет упорядоченное множество данных. Это означает, что для данных простого типа применимы операторы: **=, <, >, >=** и **<=**. Простой тип может быть порядковым и вещественным. Данные порядкового типа представляют собой значения, упорядоченные в определенной последовательности. Данные этого типа, как правило, имеют смежные значения. Описанное правило не действительно для вещественных типов.

Порядковый тип задается пользователем как диапазон или интервал. Примером такого типа данных является тип **Integer**. В следующем примере число 0 задается как самое малое значение, а число 9 как самое большое значение диапазона, например:

Number = 0.9;

Символы кодовой таблицы **ASCII** также имеют порядковый тип:

UpperCaseCharacters = A.Z;

Перечисляемый тип, в отличие от порядкового типа дает разработчику возможность самому определить тип данных:

```
TColor = (Red, Green, Blue);    //Перечисляемый тип
THotColor = Red . Blue;         //Диапазон
```

Синтаксис строкового типа достаточно прост:

Строковый тип – > String [Целое число без знака].

Строка может иметь переменную или фиксированную длину. В приведенной ниже записи в прямоугольных скобках указывается длина строки, как целое число без знака:

```
Title = string;
FixSting = string[25];
```

В среде **Delphi** определены следующие строковые типы данных:

- Короткая строка – **ShortString**.
- Длинная строка – **AnsiString**.
- Широкая строка – **WideString**.

Короткие и длинные строки используются в выражениях. Короткая строка представляет собой строку, в которой область памяти объемом от 1 до 255 символов выделяется статически. Для длинной строки оперативная память выделяется динамически, причем максимальная длина строки ограничивается только максимальным объемом памяти компьютера. Тип *Широкая строка* состоит из *Unicode* символов (*DBCS*), которые определены как 16-ти разрядные символы. *Широкая строка* не имеет ограничений на размер, так как для нее память выделяется динамически.

Структурированный тип является типом данных, который составляется из двух типов. Тип **Массив** (**array**) является ничем иным как таблицей, доступ, к данным которого осуществляется с помощью индексов. К

структурированным типам также относятся: Тип **Запись**, Тип **Класс**, Тип **Ссылка на класс**, Тип **Множество** и тип **Файл**.

Синтаксическая диаграмма типа **Массив** в общем, виде выглядит так: Тип **Массив – > array** [Тип **Индекс**] **of** Тип.

Указатель является переменной типа **longInt**. Если указатель не содержит никакого значения, можно присвоить ему значение **nil** (Пустой указатель).

Синтаксическая диаграмма типа Указатель выглядит следующим образом: Тип **Указатель** – > **^ Базовый тип**, **Базовый тип** –> **Идентификатор типа**.

Процедурный тип данных позволяет трактовать процедуры и функции в качестве значений, которые можно присваивать переменным и передавать их в качестве параметров. Процедурные типы совместимы друг с другом, если выполняются следующие условия:

- Оба процедурных типа имеют одинаковое количество параметров.
- Параметры занимают одинаковые позиции и имеют один тип. Имена параметров не влияют на совместимость.
- Типы результатов возвращаемых функциями идентичны.
- Любой процедурный тип совместим со значением *nil*.

Использование процедурных типов:

Proc = **procedure**;
SwapProc = **procedure**(**var** X, Y: Integer);
StrProc = **procedure**(S: **String**);
MathFunc = **function**(X: Extended): Extended;
DeviceFunc = **function**(**var** F: Text): Integer;
MaxFunc = **function**(A, B, Extended; MaxFunc): Extended;

Типы процедур и функций:

- **Proc** – определен как процедурный тип для процедуры, не имеющей никаких параметров.

- **SwapProc** – представляет собой процедурный тип, которому передаются параметры – переменные.
- **StrProc** – является примером объявления процедурного типа с одним параметром - значением типа **String**.
- **MathFunc** и **DeviceFunc** представляют собой определения процедурного типа, в которых функция определяется в зависимости от переданного параметра – значения и параметра – значения переменной.
- При определении **MaxFunc** для параметра значения **F** указывается процедурный тип **MaxFunc**.

Константы объявляются в секции **Implementation** модуля программы. При использовании констант необходимо учитывать совместимость типов. Константа может получать значение другой константы, но не самой себя. Значение константы может задаваться посредством выражения. Простая константа представляет собой частный случай типизированной константы. При объявлении типизированной константы указывают не только ее значение, но и тип.

Константа типа *Массив* определяет значение элементов массива: Line: **array**[1..6] **of** integer = (1, 2, 6, 24, 120, 720);

Переменные являются наиболее важными идентификаторами, используемыми в программном коде. Переменные должны объявляться до момента их использования. Если переменная объявлена, то для нее резервируется память. Такие переменные представляют собой статические переменные, для которых существует непосредственная зависимость между идентификатором и местом хранения значения в памяти. Наряду с такими переменными существуют и динамические переменные. Память для этих переменных выделяется по мере необходимости в ходе выполнения программы. В следующем примере приводятся варианты объявления переменных:

```
var
A: integer;    B: boolean;    C: array[1..10] of extended;    D, E: file;
begin
end;
```

Целочисленные типы Object Pascal

Тип	Область значений	Физический формат
Integer	от – 2147483648 до 2147483647	32 разряда, со знаком
ShortInt	от – 128 до 127	8 разрядов, со знаком
SmollInt	от – 32768 до 32767	16 разрядов, со знаком
LongInt	от – 2147483648 до 2147483647	32 разряда, со знаком
Byte	от 0 до 255	8 разрядов, без знака
Word	от 0 до 65535	16 разрядов, без знака
Cordinal	4gb	32 разряда без знака

Действительные типы Object Pascal

Тип	Область значений	Точность	Размер в байтах
Real	от $2.9*10^{-39}$ до $1.7*10^{38}$	11 – 12 разрядов	6
Single	от $1.5*10^{-45}$ до $3.4*10^{38}$	7 – 8 разрядов	4
Double	от $5.0*10^{-234}$ до $1.7*10^{308}$	15 –16 разрядов	8
Extended	от $3.4*10^{-4932}$ до $1.1*10^{4932}$	19 – 20 разрядов	10
Comp	от -2^{63} до $2^{63}-1$	19 – 20 разрядов	8

Переменным целочисленного типа могут присваиваться значения в шестнадцатеричном представлении. Такие значения начинаются со знака **$**. В следующем примере показано присвоение переменной целочисленного типа шестнадцатеричного значения:

Var I : integer;

Begin I := $22FA1D **end**;

Булевы типы являются также порядковыми типами.

Базовый тип *Char* занимает, как правило, 1 байт памяти. На основании данного типа составляются новые базовые типы, такие как *ShortString*.

Если посмотреть на программу, написанную на языке *Object Pascal*, то сразу становится, очевидно, что весь программный код состоит из одних процедур и функций и находится в секции *implementation* модуля. Программный блок имеет ту же структуру. Сначала идет раздел объявлений, после чего следует программный код. Секция операторов содержит со-

ставной оператор. Составной оператор может содержать несколько простых или комплексных операторов, расположенных между ключевыми словами *begin* и *end*. Каждый оператор простой или составной заканчивается точкой с запятой. Обычно оператор представляет собой вызов процедур, либо выполнение операции присвоения (**:=**). Переменная, которой присваивается значение должна быть совместима по присвоению с переменной или функцией, расположенной в правой части данной операции. Выражение может быть простым выражением, либо соотношением между двумя простыми выражениями. В таком случае говорят о *реляционном выражении*. Реляционное выражение возвращает только значение *True* или *False*. Необходимо обратить внимание на то, что в реляционном выражении указывается только один реляционный оператор. Если необходимо использовать большее количество операторов, реляционное выражение следует записывать с применением круглых скобок, например:

If (X >= 10) **and** (Y >= 10) **and** (X < 100) **and** (Y < 100) **then**

begin */текст программы* **end;**

Простое выражение, в зависимости от типа *терма*, может состоять из нескольких операций сложения, логического **И**, деления, логического **ИЛИ** и вычитания нескольких термов. Терм состоит из нескольких операторов и операндов. Операторы ***, /, div, mod, and, shl** и **shr** имеют приоритет перед операторами **+, -, or** и **xor**. Кроме того, они имеют приоритет перед реляционными операторами **<, >, <=, >=, <>** и **in**.

Область видимости переменной представляет собой фрагмент программного кода, в котором переменная считается известной. Понятие области видимости относится, прежде всего, к идентификатору переменной. Он должен быть уникальным на том уровне программного блока, на котором объявлялась переменная. В зависимости от уровня, на котором объявляются переменные, различают следующие виды переменных:

- Локальные переменные.
- Глобальные переменные.

Область видимости глобальных переменных определяется или на уровне программы, или на уровне модуля. Они известны в пределах всего программного кода приложения.

```
Unit Test;
interface
uses  Windows;
type
  TForm1 = class(TForm);
var
  Form1: TForm1;
  A: integer;                //Общедоступная переменная A
implementation
procedure SetA;
var
  B: integer;                // Локальная переменная B
begin
  B:= 4;
end;          //Конец области видимости локальной переменной B
procedure TestA
begin
  A:= 3;      // Присвоение значения общедоступной переменной
  SetA;                      // Вызов процедуры SetA
  Writeln(A);                // Значение A равно 3
end;
```

Локальные переменные объявляются непосредственно в процедуре перед ключевым словом **begin**. Пользователю самостоятельно необходимо написать ключевое слово **var**. После имени переменной, перед объявлением ее типа ставится двоеточие. Объявление переменной заканчивается точкой с запятой.

Присвоение значений переменных выполняется непосредственно в теле процедуры, т.е. между **begin** и **end**. После выполнения операции присвоения ставится точка с запятой.

Оператор может быть простым и структурированным. Структурированные операторы дают возможность управлять последовательностью выполняемых действий, то есть управлять ходом выполнения программы. В теории структурированного программирования различают три вида структур:

- *Конкатенации.*
- *Условные операторы.*
- *Циклы.*

Структурированная программа должна строится только на основе этих структур. Конкатенация означает, что несколько команд ставятся друг за другом и в этой последовательности выполняются. Такую последовательность операторов можно рассматривать как блок. После завершения отработки последнего оператора блок считается выполненным.

Блок, содержащий операторы, имеет вход и выход. Если в блоке обнаруживается ошибка, то причина ее возникновения сразу становится ясной.

В случае условных операторов, речь также идет о блоке, содержащем несколько вложенных блоков, из которых выполняется только один. Выбор выполняемого блока определяется из условия. Существует два варианта условных операторов: оператор **if** и оператор **case**. Оператор **case** применяется при осуществлении выбора более чем из двух блоков. Хотя оператор *case* может применяться в любом случае, для выбора одного из двух выполняемых блоков чаще используется оператор **if**. В самом простом случае существует только один вложенный блок. Если заданное условие возвращает значение true, то вложенный блок выполняется, в противном случае – нет. Эту конструкцию можно считать блоком, который имеет вход и выход. При входе в блок анализируется условие его выполнения. С конструкцией "Если указанное условие верно, то выполняется блок 1, в противном случае блок 2" ситуация обстоит иначе. Две структуры конкатенации теперь вставлены теперь в структуру выбора, и одна из них выполняется теперь в любом случае.

В **Object Pascal** программные блоки обозначаются при помощи зарезервированных слов **begin** и **end**.

На рис.1 приведен фрагмент программы, реализующий описанную структуру. Указанный код, выполняет вывод в окне **Edit1** значений двух различных переменных в зависимости от состояния опции **CheckBox1**. На рис.2 показан результат выполнения программы.

Фрагмент программы

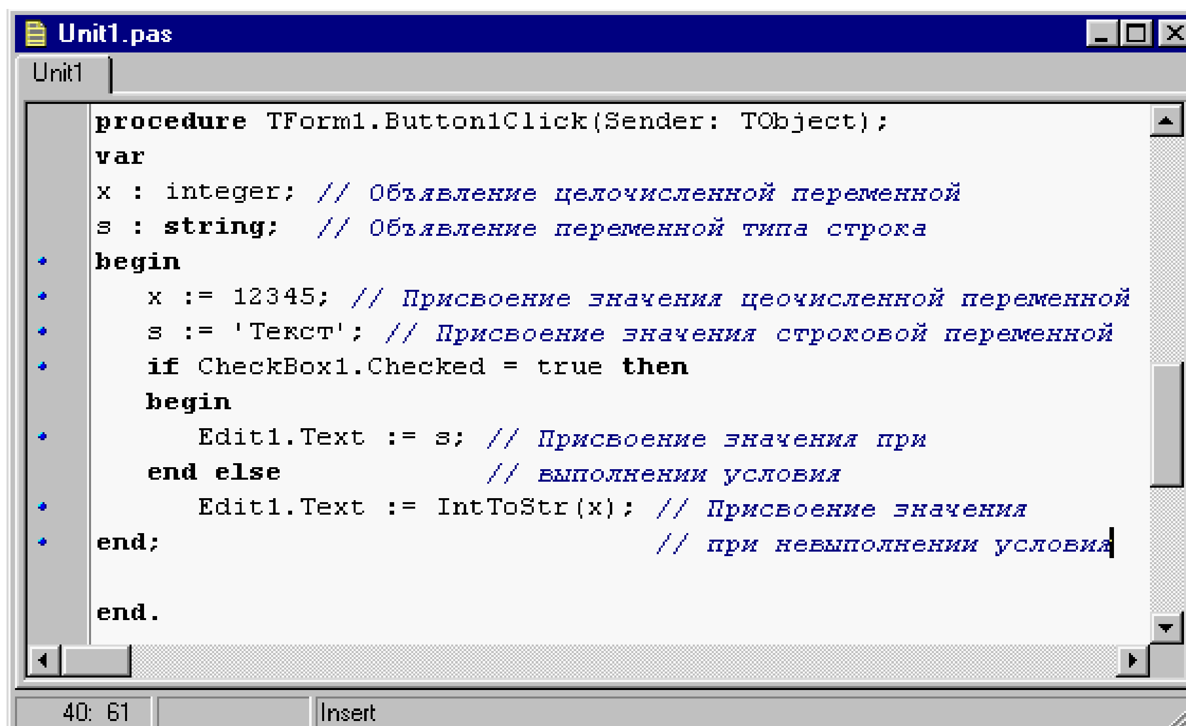

Рис.1

Результат выполнения программы

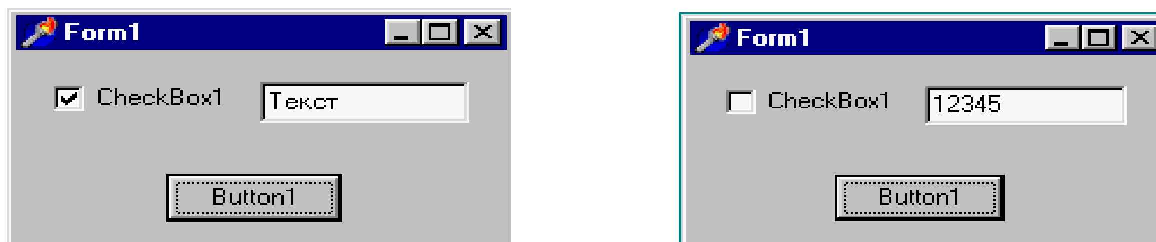

Рис.2

Другими структурами, влияющими на ход выполнения программы, являются структуры циклов. Данные структуры реализованы таким образом, что определенный программный блок может выполняться один раз, несколько раз или вообще не выполняться. Данное условие называется *условием выхода из цикла*. Существует три конструкции циклов, определенных операторами: **Repeat**, **While** и **For**. Выбор конструкции зависит от характера циклического процесса. Если известны начальные и конечные состояния, то рекомендуется применять оператор **For**. Если же существует уверенность в том, что программный блок должен быть выполнен, по меньшей мере один раз и может быть установлена необходимость повторного выполнения данного блока, то следует использовать оператор **Repeat**.

Этот оператор называется оператором цикла с условием завершения. Если неизвестно, должен ли выполняться программный блок вообще, рекомендуется применять оператор **While**. Существует также возможность организации обратного счета с использованием директивы **downto**.

Характерным свойством оператора **For** является то, что в отличие от других конструкций цикла, управление выходом из цикла не осуществляется посредством вложенного блока. Это обусловлено тем, что начальное и конечное состояния известны заранее и поэтому ясно, когда следует осуществить выход из цикла. На рис.3 приведен пример организации простого цикла.

Конструкция простого цикла

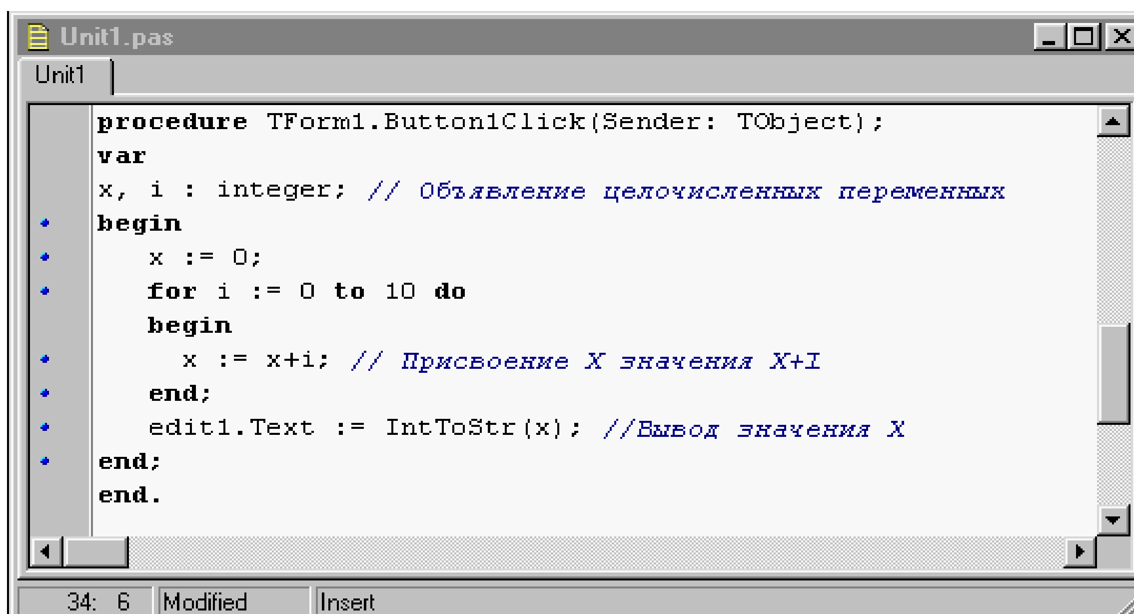

Рис.3

Из приведенного примера видно, что переменная цикла должна быть порядкового типа. Типы начального и конечного значений должны быть совместимы по присвоению с типом переменной цикла.

Рассмотрим условия выполнения цикла с оператором **Repeat**. Особый интерес в данной конструкции представляет условие выхода из цикла. Цикл должен когда-либо закончится, а именно после того, как выражение примет значение **true**, т.е. выражение анализируется до достижения условия **until**. Если данное условие не выполняется, то выполнение цикла продолжается. Фрагмент такого программного кода приведен в примере

Пример

```
Number := 9;
repeat
Number := Number - 2;
until Number = 0;
```

В этом случае цикл будет длиться бесконечно долго, поскольку переменная **Number** никогда не примет значения равного нулю. Рекомендуется условие выхода из цикла задавать посредством понятий больше или меньше, как показано в примере.

Пример

```
Number := 9;
repeat
Number := Number - 2;
until Number < 0;
```

Различие между оператором **while** и оператором **repeat .. until** состоит в следующем. В конструкции **while** сначала анализируется условие выхода из цикла, после чего вызывается сам оператор. В конструкции **repeat .. until** все происходит наоборот. Число циклов в конструкции **while** может быть 0 или больше, а в конструкции **repeat .. until** 1 или больше. В конструкции **while** итерация производится до тех пор, пока выполняется условие, в конструкции **repeat .. until** итерация выполняется только до тех пор, пока условие не будет выполнено. При написании программного кода необходимо учитывать, что в случае применения конструкции **while do** вначале должно быть инициализировано условие выхода из цикла. Пример демонстрирует использование конструкции **while do:**

Пример

```
Number := 9;
while Number > 0 do
Number := Number - 2;
```

В этом случае переменная **Number** после окончания цикла принимает значение – =1.

Перед рассмотрением вопросов связанных с практическими работами по изучению среды **Delphi** необходимо отметить, что знание стандартных процедур и функции языка **Object Pascal** позволяет быстро и корректно создавать сложные программы. Ниже приводятся только те из них, с которыми пользователь столкнется на первых этапах изучения языка.

- ABS – абсолютное значение.
- Ceil – округление в положительную сторону.
- Dec – уменьшение значения переменной.
- EXP – Экспоненциальное значение аргумента.
- Flor – округление в сторону меньшего числа.
- High – наибольшее значение элементов массива.
- Inc – увеличивает значение.
- Int – Возвращает целую часть аргумента.
- Ln – определяет натуральный алгоритм.
- Low – возвращает наименьшее значение элементов массива.
- Mean – ср. арифметическое значение элементов массива.
- Power – возводит число в степень.
- Round – округляет по правилам ср.
- SQR – квадрат.
- SQRT – корень кв.
- SUM – Сумма всех элементов массива.
- Trunc – отсекает дробную часть.

- AbcExtractFileDir – имя дисковода и каталога указанного файла.
- ExtractFileDrive – имя дисковода указанного файла.
- ExtractFileExt – расширение файла.
- ExtractFileName – имя файла.
- ExtractFilePath – путь к файлу.
- MkDir – создает новый каталог (папку).
- RemoveDir – удаляет пустой каталог.
- Rename – переименовывает файл.

- FloatToDecimal – преобразовывает значение с плавающей запятой в десятичное значение.
- FloatToStr – преобразовывает значение с плавающей запятой в строку.
- StrToFloat – преобразовывает строку в значение с плавающей запятой.
- Chr – возвращает символ, соответствующий коду ASCII.
- IntToStr – преобразовывает целое число в строку.
- StrToInt – преобразовывает строку в целое число.

ТЕМА 2 ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Объектно-ориентированное программирование (**ООП**) является одной из лучших технологий создания крупных программных проектов. Преимущества **ООП** определяются возможностью многократного использования однажды разработанного кода и построения приложений из объектов, которые представляют собой отображения реально существующих предметов и процессов.

Как следует из названия, сущность **ООП** заключается в использовании объектов. **ООП** представляет собой расширение традиционных языков программирования, таких как **C** и **Pascal**, новым структурированным типом данных – **классом**. Наиболее существенными отличительными чертами **ООП** от других структурированных языков программирования являются применение **классов наследования** (**Inheritance**), **инкапсуляции** (**encapsulation**) и **полиморфизма** (**Poly** – много, **Morphe** – вид, форма). В настоящем разделе детально рассматриваются понятия объекты и классы.

2.1 Классы

Класс – это абстрактное понятие, которое можно сравнить с понятием категория. По определенным свойствам любого элемента можно установить его принадлежность к конкретной категории. Сама категория определяется общими свойствами, которые имеют все экземпляры этой категории.

Наглядно, поясняющий данное понятие, пример приведен на рис.4 Здесь рассматриваются категории музыкальных инструментов. Воспользу-

емся этим примером, для рассмотрения понятий **Категория**, **Наследование** и **Инкапсуляция**.

Дерево категорий музыкальных инструментов

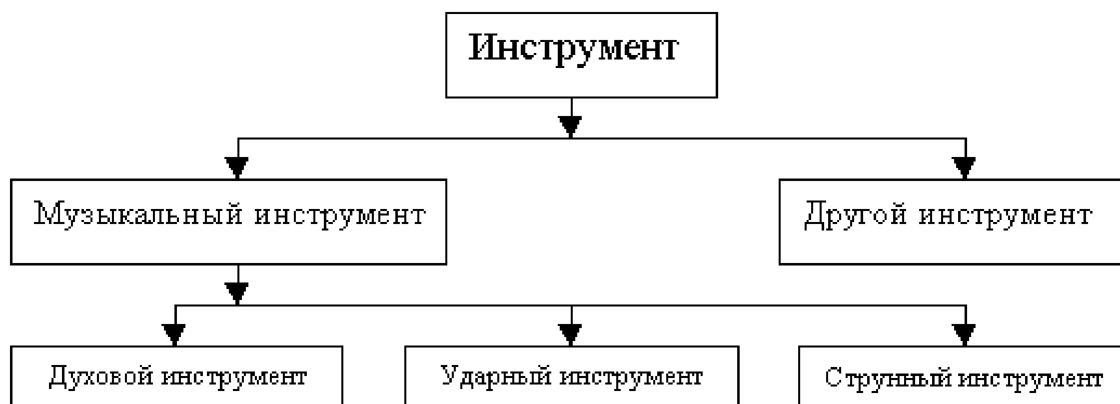

Рис.4

Музыкальные инструменты делятся на духовые инструменты, ударные и струнные инструменты. Все эти категории принадлежат к категории музыкальных инструментов. В свою очередь, категория музыкальных инструментов входит в общую категорию инструментов. Музыкальные инструменты имеют общие свойства, но каждый инструмент сам по себе обладает особыми свойствами, которые определяют его назначение и отличают его от других инструментов.

Определенный музыкальный инструмент является объектом. **Категория**, к которой этот инструмент принадлежит – это **Класс**. По этому же принципу можно описать классы в **ООП**.

Класс в **ООП** – это абстрактный тип данных, который включает не только данные, но и процедуры и функции. Функции и процедуры класса называются **методами**, которые содержат исходный код, предназначенный для обработки внутренних данных объекта данного класса.

2.2 Наследование

Классы содержат данные и методы. В **ООП** методы и данные одного класса могут передаваться другим классам. Это означает, что объекты могут наследовать свойства друг друга. Класс, наследующий свойства другого клас-

са, обладает теми же возможностями, что и класс, от которого он порожден. Этот принцип называется **наследованием** (**inheritance**). Порожденный **класс** называется **потомком** (**descendant**), а тот от кого он порожден – **предком** (**ancestor**). Благодаря новым свойствам, которыми дополняется потомок, порожденный класс может обладать большими возможностями, чем его предок.

В приведенном примере (рис.4) класс труба происходит непосредственно от класса духовые инструменты. Таким образом, уже определено, что для получения звука в трубу необходимо дуть, и это ее свойство не надо переопределять заново. Этот же принцип соблюдается и для свойства музыкальных инструментов издавать звуки. Это свойство класса **Музыкальные инструменты** перенесено на класс **Труба** через его прямого предка, класс **Духовые инструменты**.

Механизм наследования обеспечивает возможность многократного применения программного кода. Таким образом, классы могут быть представлены в виде иерархии. Библиотека **VCL** в среде **Delphi** и является такой иерархической системой классов.

2.3 Инкапсуляция

Совмещение данных и методов их обработки в одном объекте называется **инкапсуляцией** (**encapsulation**). Методы объекта определяют способы изменения данных.

Например, способность духового инструмента издавать звуки, обусловлена продуванием воздуха через мундштук и наличие последнего характерно для духового инструмента. Иными словами мундштук можно использовать только для определенных целей.

Пусть пользователю необходимо написать программу, которая выполнила бы дуэт духового и струнного инструментов. Для этого необходимо определить классы: Духовой инструмент и Струнный инструмент. Для класса Духовой инструмент надо определить, что имеется мундштук и что в него необходимо дуть для получения звука. Для класса Струнный инструмент предусматривается, что по струнам необходимо ударять, чтобы получить звук. Оба класса уже способны создавать музыку, но это свойство было унаследовано от их предка. Они унаследовали метод PlayMusic, который объявлен и

реализован как метод класса Музыкальный инструмент. Таким образом, этот метод уже не нужно создавать, и нет необходимости знать код реализации этого метода, чтобы использовать его в двух новых классах.

Этот принцип **сокрытия информации** (**information hiding**) характерен для инкапсуляции и существенно облегчает написание больших и стабильно работающих приложений.

2.4 Полиморфизм

Полиморфизм означает, что один и тот же метод выполняется по-разному для различных объектов. Например, метод класса Музыкальный инструмент PlayMusicForAnOrchestra – может быть определен как общий метод, который может использоваться с любой категорией музыкальных инструментов. Этот метод описан таким образом, что не важно, какой именно инструмент получает задание играть, однако для классов, описывающих конкретные инструменты, данный метод должен быть **переопределен** (**override**), что даст возможность определить конкретные действия, учитывающие особенности данного инструмента.

2.5 Класс - новый тип данных

Концепция объектно-ориентированного программирования предполагает использование нового типа данных – **Класс**. Тип **Класс** принадлежит к совокупности используемых в среде **Delphi** структурированных типов: множество, массив, запись и класс. Особенность типа **Класс** состоит в том, что он содержит **методы** и **свойства**. Это значит, что класс описывает группу данных и одну или более процедуру или функцию, которая имеет доступ к этим данным. Процедуры и функции называются **Методами**. Тип **Класс** – это структура, состоящая из некоторого количества элементов: **Полей**, **Методов**, **Свойств**.

Поля содержат данные определенного типа. **Методы** – это функции и процедуры, выполняющие определенные действия. **Свойства** – это поля данных, которые влияют на поведение объекта. Они служат для описания объекта и отличаются от обычных полей тем, что присвоение им значений связано с вызовом методов.

2.5 Объявление типов

Каждый новый класс в среде **Delphi** должен быть объявлен. Для этого используется зарезервированное слово **class**. Объявление определяет функциональные возможности класса. В языке программирования **Object Pascal** новый класс объявляется следующим образом:

Tnew = **class**(TOld);

Для объявления классов в модуле отведен особый раздел, который так и называется: **Раздел объявления типов**.

Классы должны быть объявлены на уровне программы или на уровне модуля. Классы не могут быть объявлены на уровне процедуры или функции.

Зарезервированное слово **class** используется для объявления **класса** или **метода** класса.

Объявление поля включает **идентификатор**, который обозначает **поле**, и **типа данных** поля.

2.6 Объекты, классы и экземпляры

Класс и **объект** – это часто употребляемые термины в **ООП**. Однако иногда эти термины употребляются неправильно. В **ООП** принято следующее определение этих понятий.

Класс – это определенный пользователем тип данных, который обладает внутренними данными и методами в форме процедур или функций и обычно описывает родовые признаки и способы поведения ряда очень похожих объектов.

Объект – это **экземпляр класса**. Предварительно выбранные объекты, которые используются в программе (компоненты среды **Delphi**) это экземпляры классов.

Экземпляр класса реализуется переменной определенного типа класса.

Пример объявления объектов иллюстрирует пример.

Пример

type

TForm1 = **class**(TForm)

```
    Label1: TLabel;
    Label2: TLabel;
    CloseBtn: TBitBtn;
    OkBtn: TBitBtn;
  private
    {Private declarations}
   public
    {Public declarations}
   end;
  var
   Form1: TForm1;
```

В объявлении типа определен новый класс - **TForm1**, наследуемый от класса **TForm**, содержащегося в **VCL**. На это указывает зарезервированное слово **class**. Данный тип содержит указатели на компоненты, которые были вставлены в форму: два компонента **Label** (объекты типа **TLabel**, или, иначе говоря, экземпляры класса **TLabel**) и два экземпляра класса **TbitBtn**.

Для определения экземпляра нового класса объявлена переменная **Form1**.

После того, как сделаны все объявления, можно создавать и инициализировать новые экземпляры классов или объектов.

Итак, на основании изложенного материала можно сделать следующие выводы.

1. **ООП** отличается от прежних методов программирования возможностью создания новых объектов.
2. **Экземпляры классов** могут вести себя различным образом. Каждый объект, располагает копией полей данных и типа класса, но разные объекты имеют различные поля и методы.
3. Переменная может содержать ссылку на экземпляр класса (объект), при этом переменная содержит не сам объект, а указывает на зарезервированную для него область памяти. Как и переменные-указатели, несколько переменных-объектов могут иметь и нулевое значение. Это означает, что переменная в данный момент ни на что не указывает.

2.7 Область видимости идентификатора

Область видимости идентификатора (имени переменной, процедуры, функции или типа данных) – это часть программного кода, в которой возможен доступ к идентификатору.

Область видимости идентификатора компонента, объявленного в описании класса простирается от его объявления до конца определения класса. Область видимости идентификатора распространяется на все потомки этого класса и на все блоки реализации методов класса.

Область видимости идентификатора компонента зависит от атрибута видимости – раздела, в котором объявлен этот идентификатор.

В среде **Delphi** используется четыре атрибута видимости, которые также называются директивами: **published, public, protected** и **private**. Рассмотрим их. В объявлениях типов классов имеются разделы частных (**private**) и общих (**public**) объявлений. В разделе частных объявлений, размещаются поля данных и методы, недоступные за пределами модуля, содержащего объявления данного класса. Данные, описанные в этом разделе, могут обрабатываться только путем вызова методов внутри класса, а также внутри данного модуля. За пределами класса все его частные элементы неизвестны и считаются несуществующими. Поля данных и методы, объявленные в разделе общих объявлений класса (**public**), доступны для всех процедур, программный код которых расположен в области видимости данного объекта. В таком разделе объявлений типа класса должны быть объявлены поля данных и методы, к которым будут иметь доступ методы объектов других модулей.

В разделе видимости (**protected**) объявляются те элементы, к которым за пределами данного модуля могут иметь доступ только методы классов, порожденных от данного класса.

Директива **published** похожа на другие атрибуты видимости тем, что она может встречаться в объявлении типа класса. Опубликованное (**published**) поле или метод может исполняться не только во время выполнения программы, но и во время ее разработки (дизайна). Все компоненты в палитре компонентов среды **Delphi** располагают **published** – интерфейсом, который используется в первую очередь инспектором объектов. Пра-

вила видимости для директивы **published** – те же, что и для **public**.

Различие между общими (**public**) и опубликованными (**published**) разделами состоит в том, что во время выполнения программы можно получить информацию о типах опубликованных элементов класса. С помощью этой информации в приложении можно динамически определять и использовать поля и свойства любого, в том числе и неизвестного, типа класса.

ТЕМА 3 СОЗДАНИЕ МЕТОДОВ В СРЕДЕ DELPHI

3.1 Технология создания пользовательских методов

Синтаксический скелет метода может быть сгенерирован при помощи визуальных средств. Для создания конкретного метода достаточно в инспекторе объектов выполнить двойной щелчок в пустой строке напротив названия выбранного события.

Для более глубокого понимания метода необходимо вспомнить концепцию объектно-ориентированного программирования, базовым понятием которой является **Класс**.

Класс – это категория объектов, обладающих одинаковыми свойствами и поведением. При этом **Объект** представляет собой экземпляр какого-либо класса.

Метод – это процедура или функция, которая определена как часть класса и инкапсулирована (содержится) в нем. Методы манипулируют полями и свойствами классов, а также могут работать и с любыми переменными. Методы имеют автоматический доступ к любым полям и методам своего класса.

Методы можно создавать как визуальными средствами, так и путем написания полного кода метода вручную.

Рассмотрим процесс создания программы, которая поможет пользователю изучить технику написания методов в **Delphi**. Предположим, что в форму помещено 5 компонентов **Edit** и два компонента **Button**. Имя элемента управления **Edit1** изменено на **Source**. Общий вид формы и компонентов приведен на рис.5.

Вид формы программы

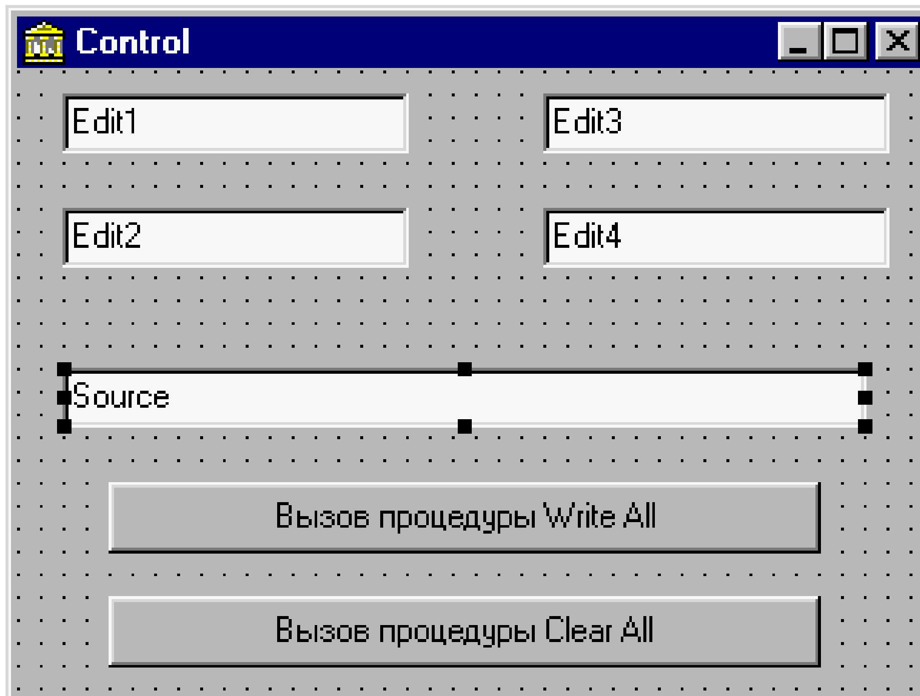

Рис.5

Для элемента управления **Source** создается пустуя процедура обработки события **OnDblClick** и затем записывается смысловая часть метода обработки события двойного щелчка.

Пример

procedure TForm1.SourceDblClick(Sender: TObject);
begin
  Source.Text:= ' Вы дважды щелкнули в поле редактирования ';
end;

Написанный таким образом обработчик события **OnDblClick** называется созданием методов при помощи визуальных средств.

Хотя внешний интерфейс разработанной программы достаточно прост, программа имеет строгую внутреннюю структуру. Известно, что каждая программа в среде **Delphi** состоит из одного файла проекта и одного или нескольких модулей. Модуль, в котором содержится главная форма проекта, называется главным. Указанием компилятору о связях между модулями является объявление **Uses**, которое определяет зависимость моду-

лей. Нет никакого функционального различия между модулями, созданными пользователем, и модулями, автоматически сгенерированными средой **Delphi**. В любом случае модуль подразделяется на три секции:

```
unit Unit1; {Заголовок модуля}
interface        {Секция Interface}
implementation     {Секция Implementation}
end.
```

В интерфейсной секции (**interface**) описывается все то, что должно быть видимо для других модулей: типы, переменные, классы, константы, процедуры и функции. В секции **implementation** помещается код, реализующий классы, процедуры или функции. Процедурам и функциям а, следовательно, и методам классов могут передаваться параметры для того, чтобы обеспечить их необходимой для работы информацией.

Предположим пользователю необходимо в разработанной программе заполнить все поля ввода, содержимым элемента управления **Source**, путем нажатия кнопки **Button1**. Естественно можно написать соответствующий обработчик события для элемента управления **Button1**. Однако нашей задачей является создание собственной процедуры заполнения полей редактирования **Edit1 - Edit4** путем вызова созданной пользователем процедуры **WriteAll** и передачи ей в качестве параметра содержимого поля редактирования **Source (EditSource.Text).** Вызов метода **WriteAll** приведен в примере.

Пример

```
procedure TForm1.Button1Click(Sender: TObject);
begin
WriteAll(EditSource.Text);
end;
```

Важно понять, что объект **Source** является экземпляром класса **TEdit** и, следовательно, имеет свойство **Text**, содержащее набранный в поле редактирования текст. Текст, который должен быть отображен в четырех полях редактирования, передается процедуре **WriteAll** как параметр. Чтобы передать параметр процедуре, необходимо написать имя этой процедуры и

включить передаваемый параметр или группу параметров в скобки. Заголовок процедуры должен выглядеть следующим образом:

procedure TForm1.WriteAll(NewString: **String**);

где передаваемый процедуре параметр **NewString** должен иметь тип **String**.

Заголовок процедуры, в общем случае, состоит из пяти частей. Первая часть заголовка процедуры – это зарезервированное слово **procedure**. Пятая, заключительная часть – это концевая точка с запятой "**;**". Обе эти части служат определенным синтаксическим целям, а именно: первая информирует компилятор о том, что определен синтаксический блок «**Процедура**», а вторая указывает на окончание заголовка. Вторая часть заголовка – слово **TForm1**, которое квалифицирует то обстоятельство, что данная процедура является методом класса **TForm1**. Третья часть заголовка – это имя процедуры. Имя процедуры может быть любым уникальным именем. В данном случае мы назвали процедуру **WriteAll**. Четвертая часть заголовка – передаваемый параметр. Параметр декларируется внутри скобок и, в свою очередь, состоит из двух частей. Первая часть параметра - это его имя, вторая - его тип. Эти части разделяются двоеточием. Если Вы описываете в процедуре более чем один параметр, то нужно разделить эти параметры точкой с запятой, например:

procedure TForm1.WriteCount(Param1: **String**; Param2: **integer**);

Разрабатываемая пользователем процедура должна находится в секции **implementation** программного модуля. После описания заголовка процедуры следует написать текст, реализующий процедуру. Метод **WriteAll**, записывающий в поля **Edit1 – Edit4** значения **NewString** приведен в примере.

Пример

```
procedure TForm1.WriteAll(NewString: String);
begin
  Edit1.Text := NewString;  Edit2.Text := NewString;
  Edit3.Text := NewString;  Edit4.Text := NewString;
end;
```

Заголовок процедуры, являющейся методом класса, пользователь должен включить в декларацию класса, например, путем его копирования. Напомним, что для методов, являющихся откликами на события, данное включение производится автоматически.

Пример

```
Type
  TForm1 = class(Tform)
    Source: TEdit;     Button1: TButton;     Button2: TButton;
    Edit1: TEdit;     Edit2: TEdit;     Edit3: TEdit;     Edit4: TEdit;
    procedure SourceDblClick(Sender: TObject);
    procedure WriteAll(NewString: String);
   private
    { Private declarations }
  public
    { Public declarations }
  end;
```

При декларировании процедуры нет необходимости оставлять в заголовке метода слово **TForm1**, так как оно уже присутствует в описании класса.

Для выполнения процедуры достаточно включить ее имя в обработчик события **OnClick** кнопки **Button1** и передать ей в качестве параметра содержимое поля **Source.Text**, как показано в примере.

Пример

```
procedure TForm1.Button1Click(Sender: Tobject);
begin
   WriteAll(Source.Text);
end;
```

Разрабатываемая пользователем процедура может не содержать никаких параметров. Предположим необходимо одновременно очищать содержимое всех полей ввода при нажатии на кнопку **Button2**. В таком случае процедура будет иметь вид приведенный в примере.

Пример

```
procedure TForm1.ClearAll;
begin
Edit1.Clear;   Edit2.Clear; Edit3.Clear;  Edit4.Clear;
end;
```

Заголовок процедуры также следует декларировать в соответствующем разделе**.**

Предположим, что разрабатываемая программа, содержит несколько форм, т.е. несколько программных модулей, и возникает необходимость в том, чтобы написанные пользователем процедуры были доступны только в модуле, где они описаны. В таком случае их необходимо декларировать их в разделе **private**.

Пример

```
private
   procedure WriteAll(NewString: String);
   procedure ClearAll;
   { Private declarations }
 public
   { Public declarations }
 end;
```

Аналогично процедурам создаются собственные функции. Синтаксическая структура функции подобна синтаксической структуре процедуры.

В общем случае заголовок функции состоит из шести частей. Первая часть заголовка функции – это зарезервированное слово **function**. Шестая, заключительная часть – это концевая точка с запятой "**;**". Вторая часть заголовка – слово **TForm1**, определяет, что данная функция является методом класса **TForm1**. Третья часть заголовка – это собственное имя функции. Имя функции может быть любым уникальным именем. Четвертая часть заголовка – передаваемый в функцию параметр. Параметр декларируется внутри скобок и, в свою очередь, состоит из двух частей. Первая часть параметра – это его имя, вторая – его тип. Эти части разделяются двоеточием. Функция может содержать несколько параметров. Если функ-

ция имеет несколько входных параметров, они разделяются точкой с запятой. В шестой части определяется тип переменной, которую возвращает функция. Шестая часть функции отделяется от пятой части двоеточием. Функция может содержать ключевое слово **Result**, которое возвращает результат выполненной функции.

Рассмотрим построение функции выполняющей преобразование переменной типа **Integer** в переменную типа **Byte**. Присвоим имя этой функции **IntToByte**. Известно, что переменная типа **Integer** может изменяться в пределах от – 2147483648 до 2147483648 (32 двоичных разряда со знаком), а переменная **Byte** может принимать значения от 0 до 255. Тогда программный код функции будет иметь вид, приведенный в примере.

Пример

```
function TForm1.IntToByte(i:Integer):Byte;
begin
  if      i>255 then Result:=255
  else if i<0   then Result:=0
  else             Result:=i;
end;
```

Программный код функции также должен находится в секции **implementation**, а ее декларирование должно быть выполнено в любом разделе секции **interface**.

Программный код функции может быть написан без ключевого слова **Result**. В примере приводится функция, которая возвращает максимальное значение одной из двух переменных типа **double**.

Пример

```
function TForm1.dmax(v1,v2:double):double;
begin
if v1>v2 then dmax:=v1 else dmax:=v2;
end;
```

Тема 4 Конструирование текстового редактора

Лабораторная работа №1

Порядок выполнения работы по дизайну приложения:

Запустите Delphi и последовательно выполните следующие пункты:

1. Сохраните в Вашей папке файлы проекта с именами *Lab1.pas*, *LabWork_1.dpr.*
2. Установите в форму следующие компоненты: *Memo, MainMenu, OpenDialog и SaveDialog.*
3. Присвойте имя форме – *MyForm*, а свойству *Caption* формы – *Текстовый редактор*. Установите свойство формы *ShowHint* в состояние *True (данное свойство формы определит в дальнейшем состояние аналогичных свойств других компонентов).*
4. Дважды щелкните мышью по пиктограмме *MainMenu*. В появившемся окне будет обозначен левый верхний прямоугольник, а в инспекторе объектов появятся свойства компонента *MainMenu*. Свойству *Name* присвойте имя *MnuFile*, а свойству *Caption* название *Файл*.
5. Подведите курсор мыши к синему прямоугольнику, в котором появилось название *Файл*, и щелкните по нему. После этого действия ниже него появится новый прямоугольник. Выделите мышью этот прямоугольник, и пустым свойствам *Name* и *Caption* соответственно присвойте значения *MnuSave* и *Сохранить*.
6. Повторите действие, и только свойству *Caption* присвойте зачение (**-**). Обратите внимание на то, что свойству *Name* автоматически присвоилось значение *N1*, а в нашем меню появилась разделительная черта.
7. Далее самостоятельно создайте *MnuClose – Закрыть*. Теперь коснитесь формы и Вы увидите, что в ней появилось созданное меню, которое Вы можите переключать.
8. Выберете в форме компонент *Memo* и в инспекторе объектов установите свойство *Scrolbars* в состояние *ssBoth*. После выполнения этого действия в компоненте *Memo* появятся полосы прокрутки. Для того чтобы компонент *Memo* занял всю площадь формы необходимо установить свойство *Align* в состояние *alClient*.

9. Далее измените имена компонентов *Memo1* на *Memo*, *OpenDialog1* на *OpenDialog* и *SaveDialog1* на *SaveDialog*.

После выполнения перечисленных пунктов вид разрабатываемого приложения должен быть подобен виду, приведенному на рис.6.

Общий вид среды разработки в процессе дизайна приложения

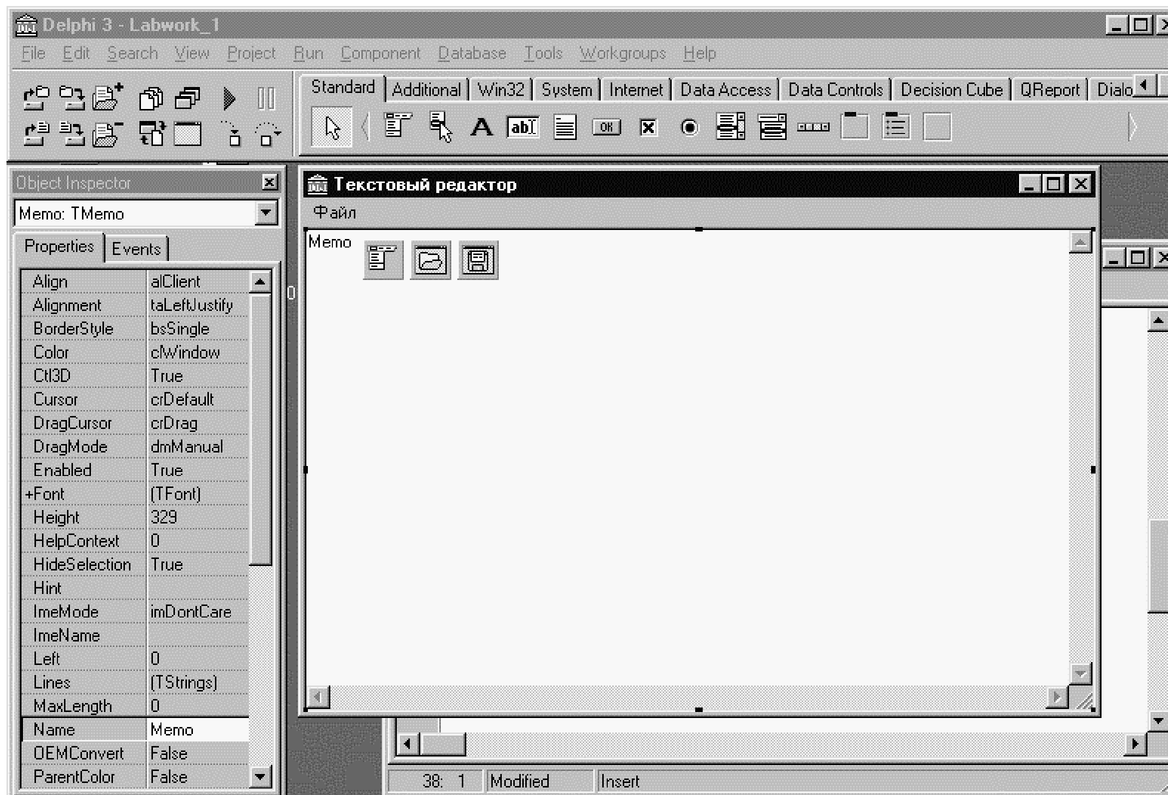

Рис.6

Далее приступим к написанию программного кода. Для этого выберете команду созданного меню *Файл/Открыть***,** и щелкните по нему мышью. Данное действие приведет к переключению интерфейса в Редактор кода с образованием конструкции:

```
procedure TmyForm.MnuOpenClick(Sender: TObject);
begin
end;
```

Затем между ключевыми словами *begin* и *end* напишите оператор *if*, а затем скопируйте и вставьте имя объекта *OpenDialog* и поставьте за ним *точку*. Это действие приведет к открытию окна дополнения кода (*Code Completion*). В окне из списка выберете метод *Execute* и напишите опера-

тор *then*. Далее перейдите на следующую строку. Скопируйте и вставьте в нее имя *Memo***,** за которым поставьте *точку*. Из списка *Code Completion* выберете свойство *Lines***,** и также поставьте *точку*. Снова откроется окно *Code Completion* из списка которого выберете метод *LoadFromFile* и откройте круглую *скобку*, что приведет к выводу контекстного списка типов переменных. Скопируйте и вставьте имя компонента *OpenDialog* и поставьте *точку*. Далее в окне *Code Completion* выберете *FileName*, закройте скобку и поставьте разделитеь -(**;**).

Полный программный код вызова диалога открытия файла имеет следующий вид:

```
procedure TmyForm.MnuOpenClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  Memo.Lines.LoadFromFile(OpenDialog.FileName);
end;
```

Итак, Вы увидели, что вручную были написаны только операторы *if, then* . Конструкции (*if … Then*) также можно создать автоматически. Для этого достаточно установить курсор между ключевыми словами *begin* и *end* в процедуре обработки события *TmyForm.MnuOpenClick* и нажать на клавиши [*Ctrl* + *J*]. Далее из списка помощника написания кода выбрать: *if (no begin/end)*. Это и есть автоматизация написания программного кода, благодаря которой значительно повышается скорость программирования и снижается количество ошибок при написании кода.

Аналогично создайте конструкции и напишите программный код для вызова окна сохранения файла и закрытия программы. Программный код должен выглядеть, как показано на рис.7.

В заключение настроим свойства *Filter* для *OpenDialog* и *SaveDialog*. Для компонента *OpenDialog* выберете свойство *Filter* и нажмите на кнопку расположенную рядом со свойством. Откроется окно установок фильтра. Заполните первую строку следующим образом: *Текстовые файлы|*.txt;*.pas*.

Для компонента *SaveDialog* выберете свойство *Filter* и нажмите на кнопку расположенную рядом со свойством. Откроется окно установок

фильтра. Заполните первую строку следующим образом: *Текст|*.txt.* Свойству *FileName* присвойте значение *New*. Свойству *DefaultExt* присвойте значение **.txt*.

Скомпилируйте и запустите созданную программу, выполнив команду *RUN* меню *RUN*.

Редактор кода, содержащий код написанной программы

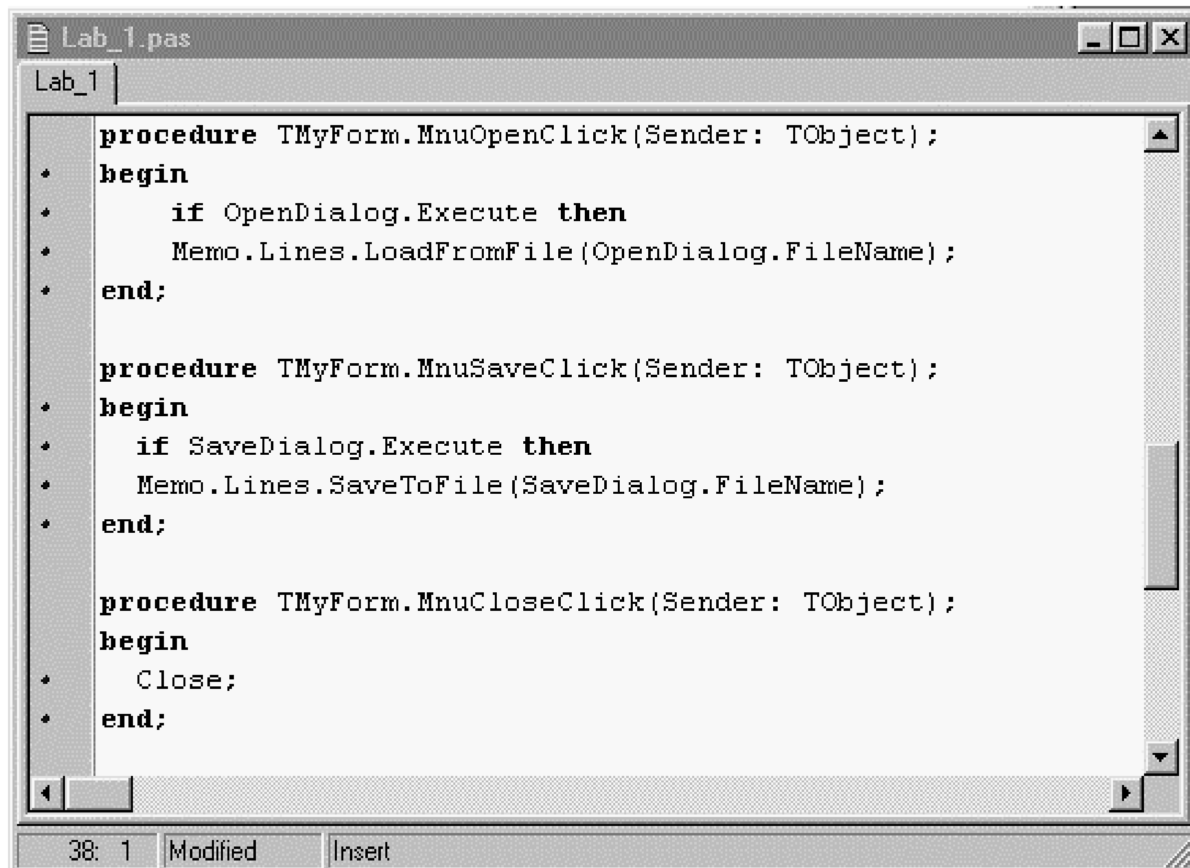

Рис.7

Далее закройте запущенную программу и сохраните файл проекта, выполнив команду *Save ALL* системного меню *File*. Выйдете из *Delphi*.

Откройте проводником *Windows* папку, содержащую проект программы и запустите созданный Вами файл *Labwork_1.exe*. Этот файл является полноценной программой, функционирующий в среде *Windows* и не требующий никаких других библиотек и среды программирования. Откройте в разработанном Вами приложении файл главного модуля программы *Lab_1.pas* (рис.8) и изучите его структуру.

Созданное приложение, с загруженным файлом текста программы

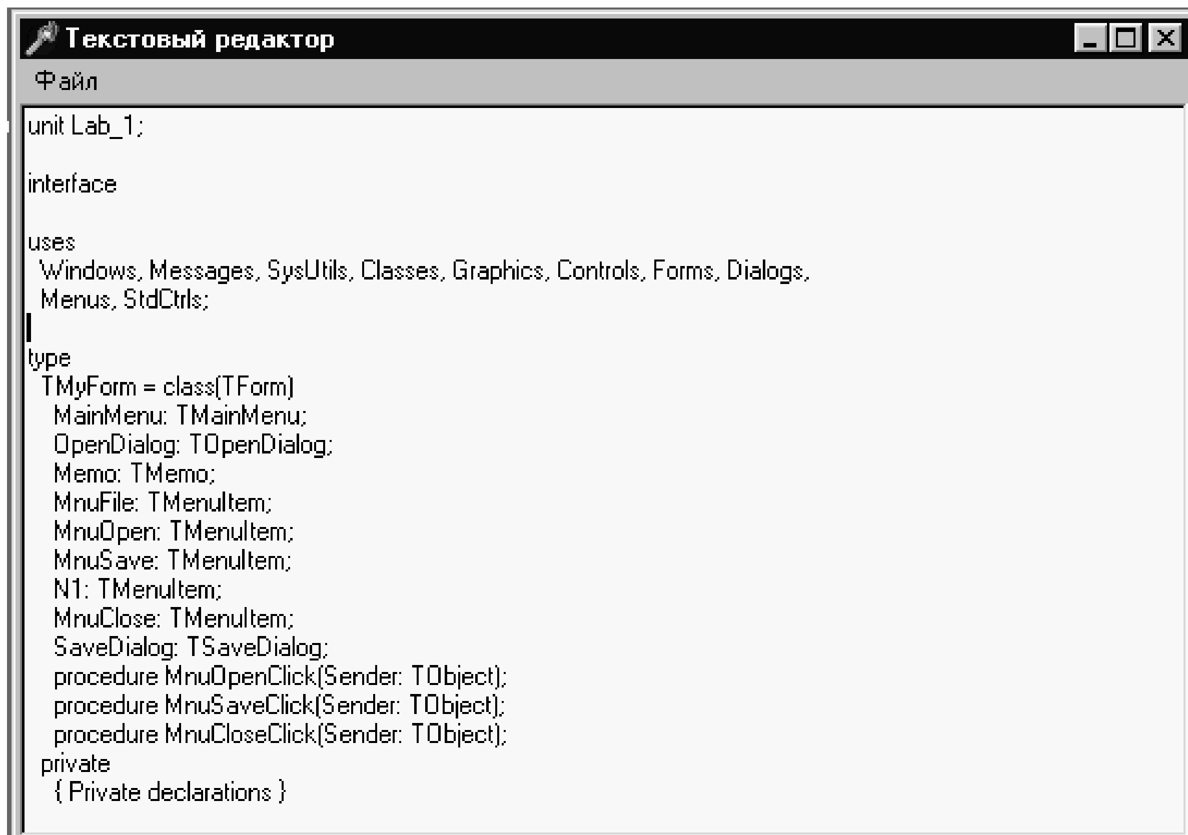

Рис.8.

Лабораторная работа №2

Установите в форму компоненты *FontDialog* и *ColorDialog*, изменив их имена на с *FontDialog1* на *FontDialog* и *ColorDialog1* на *ColorDialog*.

Далее приступим к созданию новых пунктов меню. Прежде всего, отредактируйте группу меню «*Файл*», добавив в нее пункт «*Новый*». Для этого дважды щелкните мышью по пиктограмме компонента *MainMenu*. В окне редактора меню пометьте первый пункт «*Открыть*» и нажмите на правую клавишу мыши. Выберите пункт *Insert* (вставить) и нажмите на левую клавишу мыши. При этом у Вас появится новый пустой пункт меню. Присвойте свойству *Name* значение: *MnuNew*, а свойству *Caption* значение: *Новый*.

Многие программы используют для управления так называемые «*Горячие клавиши*». Для того, чтобы Ваше приложение обладало этими возможностями, выберете из списка свойств *ShortCut* значение *Ctrl+N*. Аналогично выберите для имен меню: *MnuOpen* значение *ShortCut - Ctrl+O*, *MnuSave* значение *ShortCut - Ctrl+S* и *MnuClose* значение *ShortCut - Ctrl+Q*.

Теперь создайте код очистки окна *Memo*. Для этого дважды щелкните мышью по пункту меню «*Новый*», что приведет к передаче фокуса редактору. В образовавшейся конструкции, между ключевыми словами *begin* и *end*, запишите следующий код:

```
procedure TMyForm.MnuNewClick(Sender: TObject);
begin
  Memo.Clear;
end;
```

Чтобы проверить правильность произведенных действий, запустите программу на выполнение. Выберете сочетание клавиш клавиатуры *Ctrl + N*, которое приведет к очистке окна редактора. Аналогично, сочетание клавиш *Ctrl + O* вызовет диалоговое окно открытия файла, а сочетание клавиш *Ctrl + S* вызовет диалоговое окно сохранения файла. Закройте программу, нажав на клавиши *Ctrl + Q*.

При разработке любой программы, профессиональные программисты обязательно пишут комментарий к разрабатываемому коду. Комментарий к тексту программы, позволяет легко читать программу, не только разработчику, но и другому лицу. Напомним, что комментарий отличается от кода наличием двух косых разделителей, перед текстом или помещением текста в фигурные скобки. То есть комментарий может быть записан как:

1. *// Комментарий*
2. *{Комментарий}*

В соответствии с рекомендациями, перед процедурой, выполняющей команду очистки окна **Memo** вставьте следующий комментарий:

```
{команда - новый}
procedure TMyForm.MnuNewClick(Sender: TObject);
begin
   Memo.Clear;
end;
```

Далее создайте новую группу меню «*Редактировать*», присвоив ей имя *MnuEdit*. В этой группе создайте следующие пункты с соответствующими именами и сочетаниями клавиш:

Копировать - MnuCopy,   ShortCut - Ctrl+C;

Вырезать - MnuCut,   ShortCut - Ctrl+X;

Вставить - MnuPaste, ShortCut - Ctrl+V;

----------- - N2;

Выделить все - MnuSelAll;

Очистить выделенное - MnuDel;

----------- - N3;

Шрифт - MnuFont.

Напишите соответствующие коды для вышеуказанных команд меню и комментарий к ним:

```
{команда - копировать}
procedure TMyForm.MnuCopyClick(Sender: TObject);
begin
  Memo.CopyToClipboard;
end;
{команда - вырезать}
procedure TMyForm.MnuCutClick(Sender: TObject);
begin
  Memo.CutToClipboard;
end;
{команда - вставить}
procedure TMyForm.MnuPasteClick(Sender: TObject);
begin
  Memo.PasteFromClipboard;
end;
{команда - выделить все}
procedure TMyForm.MnuSelAllClick(Sender: TObject);
begin
   Memo.SelectAll;
end;
{команда -  очистить выделенное}
procedure TMyForm.MnuDelClick(Sender: TObject);
```

```
begin
  Memo.ClearSelection;
end;
{команда -  шрифт}
procedure TMyForm.MnuFontClick(Sender: TObject);
begin
   if FontDialog.Execute then   Memo.Font := FontDialog.Font;
end;
```

Дальнейшие действия по разработке приложения требуют определенного пояснения. Запустите программу и нажмите на правую клавишу мышки, установив курсор в окно текстового редактора программы *Memo*. Вы увидите открывшееся контекстное меню, в котором некоторые пункты закрыты для выполнения. Данное субменю Вы не разрабатывали. Его программный код реализуется в самом компоненте *Memo*. Некоторые пункты этого контекстного меню Вы продублировали в Вашем меню «*Редактировать*». Такое дублирование применяется во многих программах.

Нашей дальнейшей задачей является разработка управления доступа к пунктам меню. Если нет выделенного текста в окне редактора программы, то пункты меню «*Копировать*», «*Вырезать*» и «*Очистить выделенное*» должны быть закрыты для выполнения и наоборот открыты, если Вы выделите какой либо текст. Для выполнения этого условия установите свойство *Enabled* для соответствующих пунктов меню в состояние *False*. Запустите программу. Вы увидите, что к указанным пунктам меню нет доступа. Чтобы получить доступ к данным пунктам напишите следующий код и комментарий к нему для пункта меню «*Редактировать*»:

```
{Доступ к командам меню «Редактировать»}
procedure TMyForm.MnuEditClick(Sender: TObject);
begin
  if Memo.SelLength > 0 then
  begin
  MnuCopy.Enabled := true;   MnuCut.Enabled := true;
  MnuDel.Enabled := true;
```

```
  end else
    begin
    MnuCopy.Enabled := false;   MnuCut.Enabled := false;
    MnuDel.Enabled := false;
    end;
end;
```

Запустите программу на выполнение и проанализируйте состояние пунктов меню для случаев выделенного и невыделенного текста. Выполните любые действия по редактированию текста. Попробуйте выполнить команду «*Вырезать*» для фрагмента выделенного, текста используя сочетания клавиш клавиатуры *Ctrl* + *X*. Вы увидите, что данная команда не выполняется. Это происходит потому, что указанный выше код выполняется только при активизации пункта меню «*Редактировать*», т. е. обрабатывается событие *OnClick* для *MnuEdit*. Для устранения этого факта, выберете компонент *Memo*, переключите, инспектор объектов на страницу *Events* и для обработчика события *OnKeyDown* напишите код, активизирующий меню «*Редактировать*»:

```
{Вызов меню Edit}
procedure TMyForm.MemoKeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin
  MnuEdit.Click;
end;
```

Запустите программу на выполнение и повторите предыдущие действия.

Если проанализировать пункты созданного меню «*Редактировать*» и компоненты на странице *Dialogs*, Палитры Компонентов, то Вы увидите, что можно дополнить разрабатываемый редактор возможностью поиска указанного текста и его замены. Однако применение компонентов *FindDialog* и *ReplaceDialog* требует написания более сложного программного кода. В этой связи, на данном этапе продолжим ознакомление с теми свойствами и событиями компонентов, которые не требуют написания сложного кода.

Итак, для дальнейшего развития разрабатываемого текстового редактора расширим его меню следующими пунктами: «*Вид*» – *MnuView* и «*О програм-*

ме» – *MnuAbout*. В разделе меню «*Вид*» создайте следующие пункты:

- Панель управления – MnuShow.
- ------------------------ - N4.
- Цвет окна - MnuColor.

Установите компонент *Panel* в форму, предварительно изменив свойство *Align* компонента *Memo* в состояние *alNone*, уменьшив его высоту, для размещения компонента *Panel* в верхней части формы.

Присвойте компоненту *Panel1* имя *ToolPanel*. В инспекторе объектов измените значение свойств: *Height* на *42*, *Caption* на (*пусто*) и *Align* на *alTop*. После этих действий панель прижмется к верхней части формы.

Измените дизайн панели, установив свойство *BorderStyle* в состояние *bsSingle*. Далее разместите на панели компоненты *CheckBox*, *Label* и *ComboBox* как показано на рис.9 и затем, восстановите свойство *Align* компонента *Memo* в состояние *alClient*.

Размещение компонентов в панели

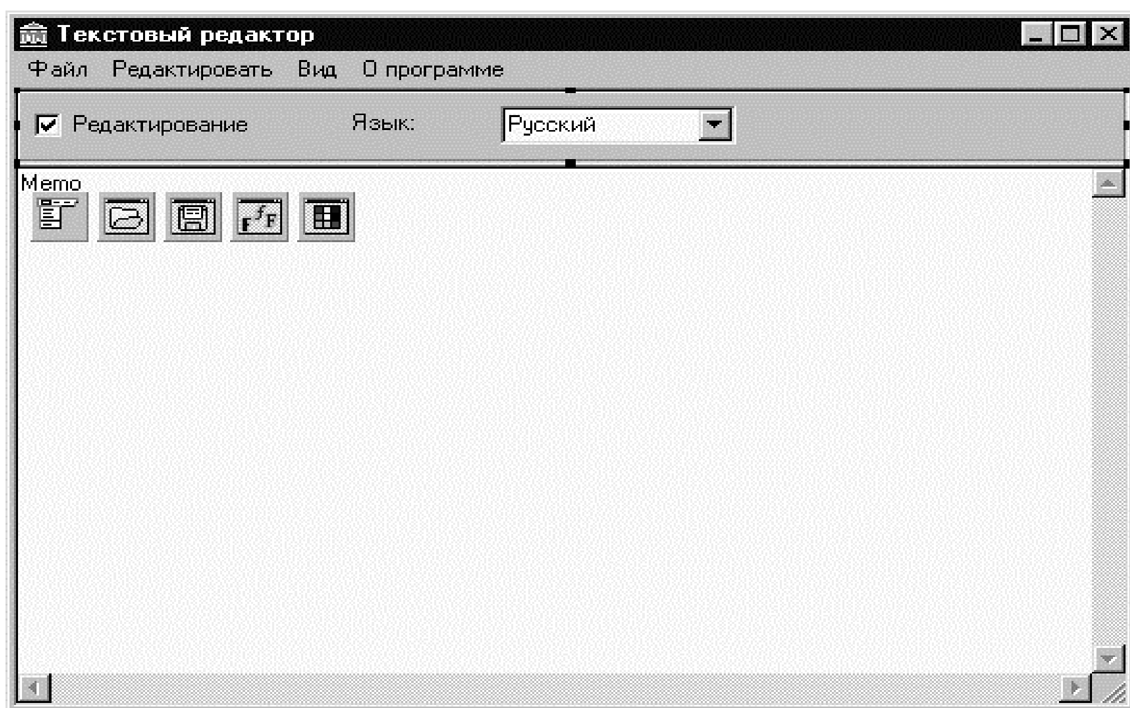

Рис.9

Присвойте компоненту *CheckBox1* имя *EditBox* и компоненту *ComboBox1* имя *LanguageBox*. Измените свойство *Caption* компонента

Label1 на «*Язык*». Установите свойство *State* компонента *EditBox* в состояние *cbChecked*. Для компанента *LanguageBox* измените свойство *Text* на «*Русский*». Щелкните мышью по кнопке *Items| (TStrings)* и в открывшемся окне редактора свойства запишите в каждую стрку:

Английский

Русский

Украинский

После выполнения указанных действий нажмите на кнопку Ok.

Перед тем, как приступить к написанию кода, присвойте свойству *Checked* пункта меню «*Панель управления*» значение *True*. После этого действия в пункте меню появится соответствующая опция. Далее дважды щелкните мышью по данному пункту меню и запишите код, соответствующий сокрытию и появлению панели в форме, а также состоянию опции пункта меню:

```
{Управление видимостью панели инмтрументов}
procedure TMyForm.MnuShowClick(Sender: Tobject);
begin
 if MnuShow.Checked = true then
  begin
    ToolPanel.Visible := false;   MnuShow.Checked := false;
  end else
     begin
     ToolPanel.Visible := true;   MnuShow.Checked := true;
     end;
end;
```

Для пункта меню «*Цвет окна*» запишите следующий код и комментарий к нему:

```
{Вызов диалога выбора цвета}
procedure TMyForm.MnuColorClick(Sender: Tobject);
begin
   ColorDialog.Color := Memo.Color;
   if ColorDialog.Execute then
   Memo.Color := ColorDialog.Color;
end;
```

Первая строка кода присваивает диалоговому окну выбора цвета значение цвета окна редактора программы. Последняя строка меняет цвет окна редактора на выбранный цвет в диалоговом окне.

Запустите программу на выполнение и проверьте управление цветом окна ввода разрабатываемого текстового редактора.

В завершении данной части работы рассмотрим пример, каким образом можно программно управлять свойством *Caption* компонентов на примере изменения языка интерфейса программы. С этой целью рассмотрим событие *OnChange* компонента *ComboBox*. Для того, чтобы вызвать это событие дважды щелкните мышью по компоненту *LanguageBox*. В окне редактора кода появится конструкция процедуры обработки события:

```
procedure TMyForm.LanguageBoxChange(Sender: TObject);
begin
end;
```

Включите в эту конструкцию следующий код:

```
{ Выбор языка }
if LanguageBox.Text = 'Русский' then          // Русский язык
  begin
  MnuFile.Caption := 'Файл';    MnuNew.Caption := 'Новый';
  MnuOpen.Caption := 'Открыть';    MnuSave.Caption := 'Сохранить';
  MnuClose.Caption := 'Закрыть';    MnuEdit.Caption := 'Редактировать';
  MnuCopy.Caption := 'Копировать';   MnuCut.Caption := 'Вырезать';
  MnuPaste.Caption := 'Вставить';   MnuSelAll.Caption := 'Выделить все';
  MnuDel.Caption := 'Очистить выделенное';  MnuFont.Caption := 'Шрифт';
  MnuView.Caption := 'Вид';   MnuShow.Caption := 'Панель управления';
  MnuColor.Caption := 'Цвет окна';   MnuAbout.Caption := 'О программе';
  EditBox.Caption := 'Редактирование';  Label1.Caption := 'Язык:';
  end;
  if LanguageBox.Text = 'Английский' then          // Английский язык
  begin
  MnuFile.Caption := 'File';    MnuNew.Caption := 'New';
  MnuOpen.Caption := 'Open';    MnuSave.Caption := 'Save';
  MnuClose.Caption := 'Close';    MnuEdit.Caption := 'Edit';
```

```
MnuCopy.Caption := 'Copy';    MnuCut.Caption := 'Cut';
MnuPaste.Caption := 'Paste';    MnuSelAll.Caption := 'Select All';
MnuDel.Caption := 'Clear';       MnuFont.Caption := 'Font';
MnuView.Caption := 'View';    MnuShow.Caption := 'Control Panel';
MnuColor.Caption := 'Colour window';   MnuAbout.Caption := 'About';
EditBox.Caption := 'Edit';    Label1.Caption := 'Language:';
end;
  if LanguageBox.Text = 'Украинский' then            //Украинский язык
begin
// Строки программы напишите самостоятельно !
end;
```

Напишите самостоятельно код выбора Украинского языка, а также для изменения названия формы программы. Сохраните файл проекта, выполнив команду *Save All*.

Запустите программу на выполнение. При правильно написанном программном коде все пункты меню программы будут соответствовать выбранному языку.

Лабораторная работа №3

Чтобы продолжить выполнение работы подберите или создайте в любом редакторе графический образ (картинку), символизирующий Вашу разработку. Затем откройте из среды **Delphi** файл проекта индивидуального задания **Labwork_1.dpr**, воспользовавшись командой **Reopen** меню **File**.

Прежде всего удалите название компонента **Memo**, в окне текстового редактора программы. Для этого пометьте его в дизайнере форм и в инспекторе объектов щелкните мышью по кнопке, расположенной справа от свойства **Lines**. В открывшемся окне **Stringlist Editor** удалите текст **Memo** и нажмите на кнопку **Ok**.

Теперь приступим к созданию формы **About** – «**О программе**», которая будет содержать Вашу авторскую информацию.

Чтобы создать новую форму выполните команду меню **File / New Form**. Присвойте новой форме имя **AboutForm**, а свойству Caption формы – текст «**О программе**». Установите размер формы 470 * 225, изменив в инспекторе объектов значения свойств **Width** и **Height**.

Форма «**О программе**» должна всегда иметь постоянные размеры и выводится поверх других форм в центре экрана монитора. Для того, чтобы данная форма обладала указанными свойствами, необходимо выполнить следующие действия:

Так как свойство **BorderStyle** определяет возможность интерактивного изменения размеров формы выбирете для этого свойства зачение **bsSingle**, а значения свойств **BorderIcons** + **biMinimize**, **biMaximize** и **biHelp** установите в состояние **False**.

Чтобы форма выводилась в центре экрана, установите свойство **Position** в состояние **poScreenCenter,** а для того, чтобы отображалась поверх других форм, определите свойство **FormStyle** как **fsStayOnTop**.

Как правило, информационная форма помимо сведений об авторах программы, содержит логотип фирмы, товарный знак или другой образ, характеризующий деятельность фирмы или назначение программы. Для того, чтобы Ваша форма имела соответствующий дизайн установите в форму объект **Image,** расположив этот компонент в верхнем левом углу формы. Присвойте свойствам объекта следующие значения: **Top = 8, Left = 8, Width = 234, Height = 154**. Далее замените его текущее имя **Image1** на **Image**.

Теперь можно вставить подобранную Вами картинку (изображение) в компонент **Image**. Для этого щелкните мышью по кнопке расположенной справа от свойства Picture. Это действие приведет к открытию диалогового окна **Picture Editor**. В этом окне выберите кнопку **Load** и загрузите картинку, а затем нажмите на кнопку **Ok**. После этого действия картинка появится в компоненте **Image**.

Для того чтобы картинка полностью соответствовала размерам компонента необходимо установить свойство **Stretch** объекта **Image** в состояние **True**. Чтобы выбранная картинка слилась с фоном формы, подберите цвет формы, используя свойство свойства **Color** формы.

Дизайн формы завершим установкой кнопки **SpeedButton**, добавив ей графический образ (картинку) и переименовав ее в **CloseBtn**. Картинку можно найти в папке **\Delphi\Images**. Картинка присваивается кнопке посредством свойства **Glyph**. Для того, чтобы кнопка имела цвет формы, присвойте свойству **Flat** значение **True**. В заключение дизайна формы установите свойство **Cursor** для **CloseBtn** в значение **crHandPoint,** свойство **ShowHint** в состояние **True** и присвойте свойству **Hint** значение – **Закрыть**.

Теперь напишем программные коды, соответствующие закрытию формы:

```
{Закрыть форму посредством кнопки}
procedure TAboutForm.CloseBtnClick(Sender: TObject);
begin
  Close;
end;
```

Сохраните разработанную форму и ее **модуль** под именем **About** в папке проекта. Для этого выполните команду **Save As** меню **File**.

Следующей задачей, которую Вам необходимо выполнить, является подключение формы **About** к проекту. Для этого воспользуйтесь командой **ADD to Project** меню **Project**, которая вызовет стандартное диалоговое окно доспука к файлам. В окне выберите файл **About.pas** и нажмите кнопку **Открыть**. После выполнения указанной команды вызовите окно списка модулей и форм проекта выполнив команду **Project Manager** из меню **View**. Указанная команда вызовет окно **Project Manager** (рис.10), в списке которого появится форма **About**.

*Обратите внимание на тот факт, что новую форму можно добавить к проекту из окна **Project Manager**, нажав на кнопку **ADD**. Аналогично можно удалить любую форму или модуль из проекта выполнив команду **Remove**. Для того, чтобы необходимая форма была видимой в среде **Delphi** достаточно выбрать ее в списке форм проекта и нажать на кнопку **Form**.*

Окно Project Manager, содержащее список форм и модулей проекта

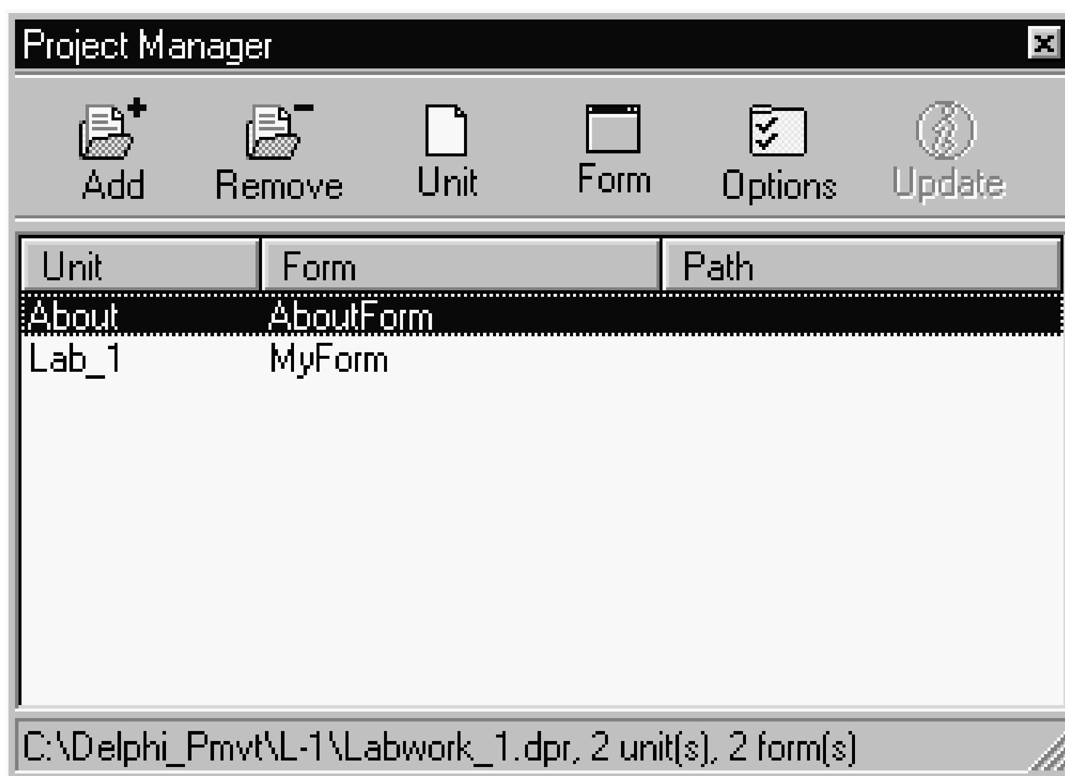

Рис.10

Для вызова формы «**О программе**», в период выполнения приложения, необходимо написать соответствующий код. С этой целью выберете в дизайнере главную форму и щелкните мышкой по пункту меню «**О программе**». В редакторе кода, включите комментарий и строку кода:

```
{Вызов формы «О программе»}
    procedure TMyForm.MnuAboutClick(Sender: TObject);
begin
AboutForm.ShowModal;
end;
```

Обратите внимание на тот факт, что постановка точки после **AboutForm** перед **ShowModal** в данном случае не приводит в действие помощник написания кода. Это объясняется тем, что среда **Delphi** еще не знает о принадлежности указанной формы к проекту.

Выполните команду **Run**, которая вместо запуска программы предложит зарегистрировать новую форму в проекте. После подтверждения регистрации, модуль формы появится в разделе **Uses** главного модуля:

```
var
  MyForm: TMyForm;
implementation
uses About;
{$R *.DFM}
```

Выполните команду **Run** повторно. После этого действия приложение будет скомпилировано и запущено.

В исполняемом приложении выполните команду «**О программе**», действие которой вызовет одноименную форму в модальном режиме. *Модальный режим означает, что Вы не можете осуществлять каких либо действий с программой*. Закройте Вашу форму и завершите работу приложения.

Сохраните Ваш проект, применив команду **Save All**. После этого закройте проект и заново загрузите его. Вы увидите, что у Вас загружена только главная форма проекта, а в редакторе кода находится только ее модуль (**Unit**). Для того чтобы увидеть форму **About** и ее модуль, воспользуйтесь менеджером проекта из меню **View**. В окне менеджера проектов выберите строку, содержащую **AboutForm,** и нажмите на кнопку **Form**. Далее закройте менеджер проектов, чтобы он не загромождал рабочее пространство среды разработки.

Запустите приложение и проверьте устранение ошибки. Кроме того, обратите внимание на то, что при выполнении команды создания нового текста **Файл** / **Новый** у Вас не задается вопроса об удалении набранного текста.

Для того чтобы вывести предварительное сообщение об удалении имеющегося текста замените код обработчика события **MnuNewClick** на следующий:

```
procedure TMyForm.MnuNewClick(Sender: TObject);
begin
if Memo.Text <> '' then
  begin
   if  MessageDlg('Очистить редактор ?', MtConfirmation,  [mbYes, mbNo], 0) = mrYes then
      Memo.Clear;
  end;
end;
```

Рассмотрим этот код. Сначала проверяется, содержит ли компонент **TMemo** текст (**if** Memo.Text <> '' **then**), а только затем выводится сообщение с вопросом на удаление существующего текста (**if** MessageDlg('Очистить редактор', MtConfirmation, [mbYes, mbNo], 0) = mrYes **then** Memo.Clear;).

Аналогично напишете программный код процедуры закрытия приложения для обработчика события **OnClose** главной формы:

```
procedure TMyForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  if Memo.Text <> '' then
  begin
    if MessageDlg('Сохранить файл ?', MtConfirmation, [mbYes, mbNo], 0) = mrYes then
      MnuSave.Click;
  end;
end;
```

Сохраните проект и запустите программу. Проверьте исполнение внесенных Вами изменений в программу. Далее наберите любой текст и сохраните его под любым именем. Внесите в него изменения и повторно сохраните текст под тем же именем в том же каталоге. Вы увидите, что при повторном сохранении файла **Windows** не выдает сообщения о наличии указанного файла на диске и заменяет его без предупреждения. Для того, чтобы **Windows** выводил предупреждение необходимо изменить свойство **SaveDialog Options** + **ofOverwritePrompt** с **False** на **True**. Внесите соответствующие изменения, сохраните проект и проверьте исполнение.

Если Вы хотите, чтобы главная форма программы открывалась в центре экрана, присвойте свойству **Position MyForm** значение **poScreenCenter**, а чтобы программа при запуске открывалась на полный экран, установите свойство **WindowState** главной формы в состояние **wsMaximized**. После выполнения этих действий сохраните проект.

Теперь приступим к заключительной части данной работы, связанной с полным оформлением приложения. Для этого воспользуйтесь командой **Option** меню **Project Delphi** и выберите страницу **Application**. В поле **Title** запишите **Текстовый редактор** и загрузите любую иконку, находящуюся в папке **/Delphi/Images/Icons**. Перейдите на страницу **VersionInfo** и установите опцию **Include version information in project**. Далее введите соответственно следующие значения, например:

CompanyName – НМетАУ, Кафедра ПМиВТ

FileDescription – Текстовый редактор

InternalName – Labwork_1

LegalCopyright - Ваша фамилия и инициалы

OriginalFilename - Labwork_1.exe

ProductName – Индивидуальное задание №1

Подтвердите ввод, нажав на кнопку Ok.

Известным Вам способом присвойте свойству **Icon** главной формы ту же иконку.

Сохраните проект командой **Save All** и запустите программу. Закройте программу и завершите работу в среде **Delphi**.

В заключение занятия проанализируем состав папки с Вашим проектом. Она содержит следующие файлы:

- Labwork_1.dpr – Файл проекта;
- Labwork_1.dof – Файл установок компилятора;
- Labwork_1.res – Файл ресурсов, содержащий картинки и иконки;
- Labwork_1.exe – Приложение (Исполняемый файл);
- Lab_1.pas – Главный модуль программы;
- Lab_1.dfm – Главная форма программы;
- Lab_1.dcu – Скомпилированный программный код главного модуля;
- About_1.pas – Модуль программы, содержащий авторские права;
- About_1.dfm – Форма окна авторских прав;
- About.dcu – Скомпилированный файл второго модуля.

Также в папке находятся, несколько файлов временного хранения с первым символом (~). Эти файлы Вы можете удалить.

Откройте проводником **Windows** папку проекта и выберете файл **Labwork_1.exe.** Просмотрите его свойства, вызвав контекстное меню правой кнопкой мыши. В свойствах файла Вы увидите присоединенную к файлу информацию, которую Вы ввели при помощи **Project Option**. Теперь запустите файл **Labwork_1.exe**, выполнив по нему двойной щелчок мышью. Разработанная Вами программа будет запущена в **Windows**, так как она является полноценным **Windows** приложением.

Лабораторная работа №4

1. Создайте в папке Вашей группы новую папку с именем: **Индивидуальное задание 2.**
2. Скопируйте в эту папку предыдущий проект, т.е. все файлы находящиеся в папке: **Индивидуальное задание 1.**
3. Откройте проект и сохраните его под новым именем **Labwork_2.dpr**., выполнив команду **Save Project As** меню **File**.
4. Запустите проект на выполнение, и затем закройте скомпилированную программу.
5. При помощи проводника **Windows,** откройте папку с новым проектом и проанализируйте ее состав. Вы увидите, что в папке появилось три новых файла, описывающие новый проект и новое приложение. Это следующие файлы: **Labwork_2.DPR, Labwork_2.DOF, Labwork_2.RES** и **Labwork_2.EXE.** Обратите внимание на то, что файлы форм и модулей не поменяли своего названия.
6. Удалите файлы проекта, относящиеся к предыдущему проекту: **Labwork_1.DPR, Labwork_1.DOF, Labwork_1.RES** и **Labwork_1.EXE.**
7. Используя команду, **Project Source** меню **View** откройте файл проекта и проанализируйте его. Обратите внимание на то, что теперь описание файла проекта имеет имя **Labwork_2** и в его состав включены модули предыдущего проекта **Lab_1.pas** и **About.pas** (рис.11).

Файл проекта после изменения его имени

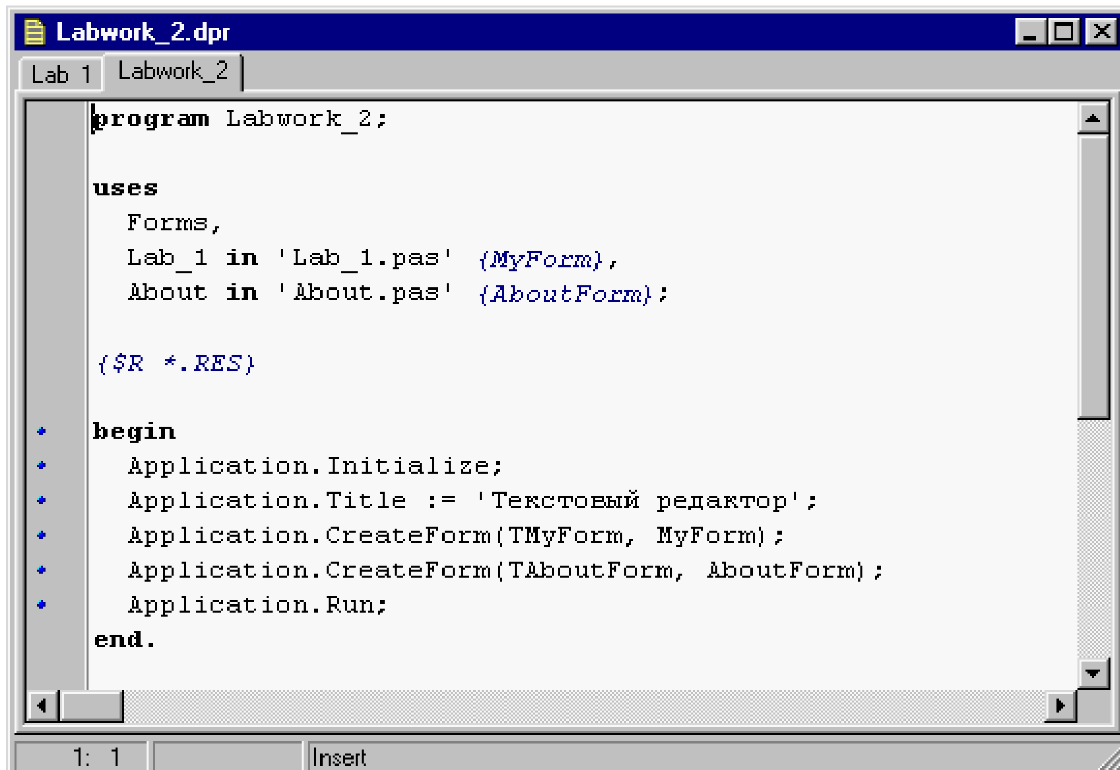

Рис.11.

Вы видите, что главной формой нового проекта по-прежнему является **MyForm** и соответствующий модуль **Lab_1.pas**.

Дальнейшей Вашей задачей, является замена компонента **Memo** на более мощный компонент **RichEdit**. Известно, что для корректного удаления компонента необходимо предварительно удалить программный код обработчиков событий. Выделите компонент **Memo** на форме и переключите инспектор объектов на страницу **Events**. Вы увидите, что использован только один обработчик события **OnKeyDown**, которому соответствует написанный ранее код – **MnuEdit.Click**. Удалите из конструкции процедуры данный код, но не саму процедуру.

```
procedure TMyForm.MemoKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
end;
```

Теперь можно удалить компонент **Memo** и установить в форму компонент **RichEdit**.

Присвойте компоненту **RichEdit** старое имя **Memo** и восстановите его обработчик события **OnKeyDown**, записав старый код – **MnuEdit.Click**.

procedure TMyForm.MemoKeyDown(Sender: TObject; var Key: Word;
 Shift: TShiftState);
begin
 MnuEdit.Click;
end;

Далее установите соответствующие свойства объекта **RichEdit**, чтобы внешний вид приложения не изменился.

Запустите программу на выполнение и протестируйте ее на предмет соответствия предыдущей версии программы. Вы увидите, что обе версии приложения функционируют одинаково. Это объясняется тем, что оба компонента имеют одинаковые свойства и методы обработки событий, которые Вы использовали. Закройте запущенное приложение.

Теперь проанализируем, что произойдет с программной, если изменить имя компонента **RichEdit** с **Memo** на **Rich**. Переименуйте компонент и попытайтесь запустить программу. При компиляции среда **Delphi** выдаст ошибку с сообщением о том, что в процедурах обработки событий имеется идентификатор **Memo**, который не декларирован, т.е. не объявлен его тип (рис.12). Это произошло потому, что при переименовании компонента с **Memo** на **Rich** в главной секции модуля он был переименован, а в процедурах нет.

Сообщение компилятора об ошибке

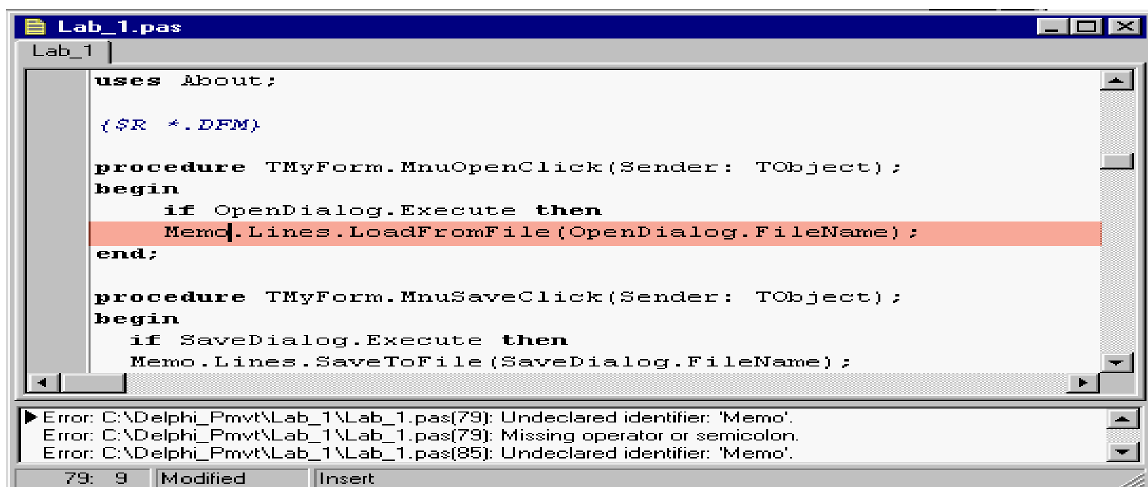

Рис.12.

Так выглядит фрагмент заголовка модуля до переименования компонента:

type

TMyForm = class(TForm)

Memo: TRichEdit;

procedure MemoKeyDown(Sender: TObject; var Key: Word;

Shift: TShiftState);

И после переименования компонента:

type

TMyForm = class(TForm)

Rich: TRichEdit;

procedure RichKeyDown(Sender: TObject; var Key: Word;

Shift: TShiftState);

Для того чтобы Ваше приложение было успешно откомпилировано, необходимо самостоятельно изменить идентификаторы в процедурах обработчиков событий.

Для этого воспользуйтесь командой **Replace** меню **Search**, предварительно установив курсор в главном модуле в позицию выше первой модифицируемой процедуры. В открывшемся диалоговом окне **Replace Text** заполните поля как показано на рис.13 и нажмите на кнопку **Replace All**.

Диалоговое окно замены текста

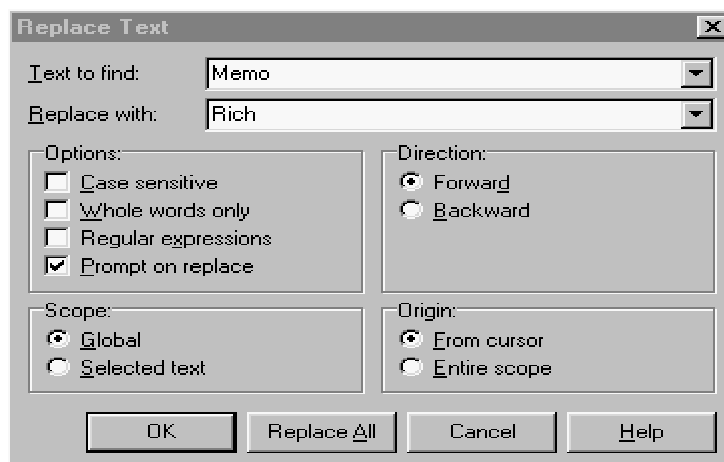

Рис.13

После выполнения этого действия запустите программу на Выполнение. При правильно выполненной замене, приложение успешно будет откомпилировано.

Теперь изменим внешний вид панели инструментов разрабатываемого приложения. Поместите в панель новый компонент группировки **GroupBox1**. Для того чтобы при изменении размеров формы приложения этот компонент находился всегда справа, установите свойство **Align** в состояние **alRight**. Для того чтобы контурная линия компонента замкнулась, удалите содержимое поля свойства **Caption**. Переместите компоненты **EditBox**, **Label1** и **LanguageBox** в **GroupBox1,** применив команды: **Cut** и **Paste** меню **Edit**. Далее выполните коррекцию размеров панели и группирующего компонента, а также расположения других компонентов на нем, чтобы внешний вид разрабатываемого приложения соответствовал виду, показанному на рис.14.

Внешний вид разрабатываемого приложения в процессе дизайна

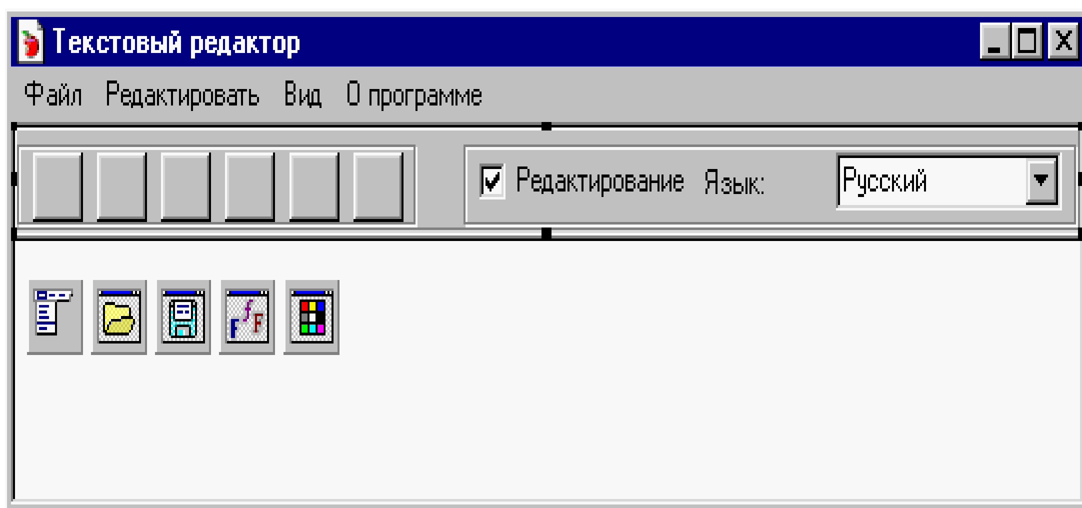

Рис.14

Известно, что многие программы содержат кнопки быстрого выполнения основных команд. Дальнейшей задачей является разработка подобной инструментальной панели. Для этого установите второй группирующий компонент и выполните его дизайн, самостоятельно изменив его свойства, чтобы внешний его вид соответствовал виду, показанному на рисунке 4.4. Разместите в нем шесть компонентов **TSpeedButton**, переименовав их соответственно в **NewBtn**, **OpenBtn**, **SaveBtn**, **CopyBtn**, **CutBtn** и **PasteBtn** (рис. 4). В папке **Delphi\Images\Button** найдите соответствующие картинки и присвойте их кнопкам (рис.15). Для того чтобы кнопки выглядели как в **MsWord**, присвойте свойству **Flat** всех кнопок значение **True**.

Установка картинок на кнопки

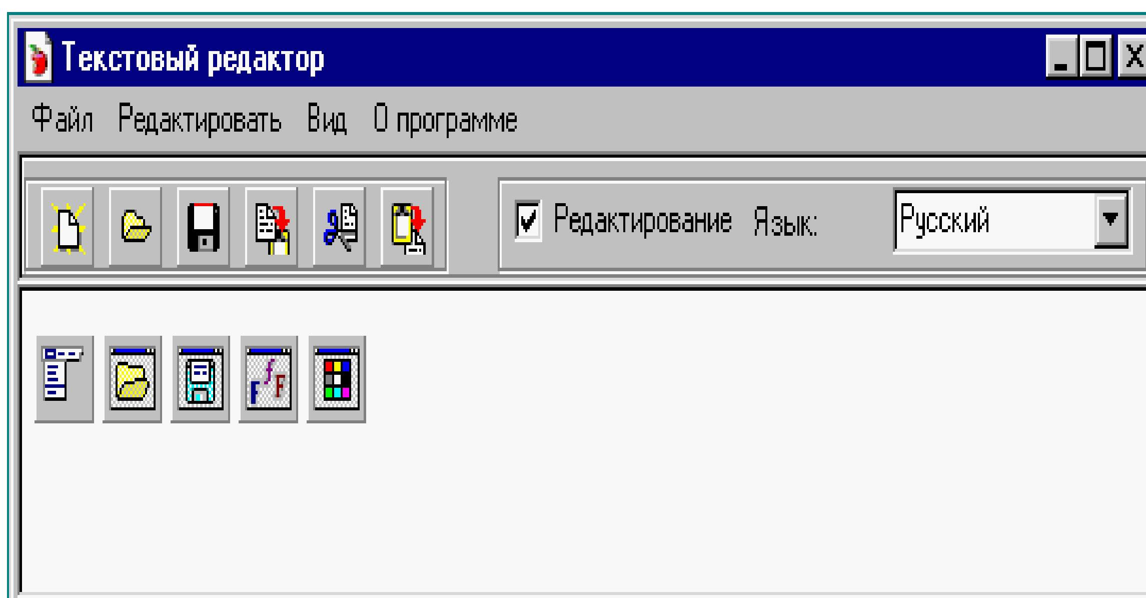

Рис.15

Теперь обратитесь к разделу **«Совместно используемые процедуры обработки событий»** и выполните следующие действия:

- Выделите компонент **NewBtn**.
- Переключите инспектор объектов на страницу **Events**.
- Выберете для обработчика события **OnClick** из списка расположенного справа, событие команды "**Новый**" меню "**Файл**" - **MnuNewClick**.

Запустите программу на выполнение и проверьте функционирование кнопки **NewBtn**.

Присвойте остальным кнопкам одноименные события. Для:

OpenBtn … OnClick := MnuOpenClick; **SaveBtn** … OnClick := MnuSaveClick;

CopyBtn … OnClick := MnuCopyClick; **CutBtn** … OnClick := MnuCutClick;

PasteBtn … OnClick := MnuPasteClick.

Сохраните файл проекта, выполнив команду **Save All**. Запустите программу на выполнение и проверьте функционирование всех кнопок.

Для того, чтобы создать всплывающую подсказку, необходимо свойство **ShowHint** каждой кнопки установить в состояние **True** и свойству **Hint** присвоить текстовое значение. Однако Вы можете одним действием установить для всех кнопок свойство **ShowHint** в **True**, установив одноименное

свойство компонента – владельцу кнопок **GroupBox2**. Выполните указанное действие и присвойте свойству **Hint** кнопок следующие значения:

NewBtn.Hint := Новый; OpenBtn.Hint := Открыть; SaveBtn Hint := Сохранить;

CopyBtn Hint := Копировать; CutBtn Hint := Вырезать; PasteBtn Hint := Вставить;

Чтобы курсор принимал вид указательного пальца при подведении его к кнопке, установите для всех кнопок свойство **Cursor** в значение **crHandPoint**.

Для того чтобы состояние кнопок соответствовало принятым правилам редактирования, установите для кнопок **CopyBtn** и **CutBtn** свойство **Enabled** в состояние **False** и напишите следующий обработчик события **OnSelectionChange** компонента **Rich**:

```
{Доступ к командам}
procedure TMyForm.RichSelectionChange(Sender: TObject);
begin
  CopyBtn.Enabled := Rich.SelLength > 0;
  CutBtn.Enabled := CopyBtn.Enabled;
end;
```

Приведенные выше строки программы анализирует, есть ли выделенный фрагмент текста в редакторе. Если есть, то кнопки **CopyBtn** и **CutBtn** доступны.

Так как компонент **RichEdit** позволяет работать с форматированным текстом, воспользуемся этой возможностью и модифицируем разрабатываемый редактор следующим образом. Свойства **WordWrap** и **WantReturns** установите в состояние **True**. Замените программный код обработчика события выбора шрифта на нижеуказанный:

```
{Выбор шрифта}
procedure TMyForm.MnuFontClick(Sender: TObject);
begin
  FontDialog.Font := Rich.Font;
  if FontDialog.Execute then
    Rich.SelAttributes.Assign(FontDialog.Font);
end;
```

Измените свойство **Filter** компонентов **OpenDialog** и **SaveDialog** в соответствии рис.16.

Присвойте свойству **DefaultExt** компонента **SaveDialog** значение ***.rtf**.

Установите в форму новый компонент **PrintDialog** и присвойте ему имя **PrintDialog**. Используя команду **Insert** редактора меню, создайте новый пункт меню "**Печать**" с именем **MnuPrint** (рис.17).

Список полей фильтров объектов OpenDialog и SaveDialog

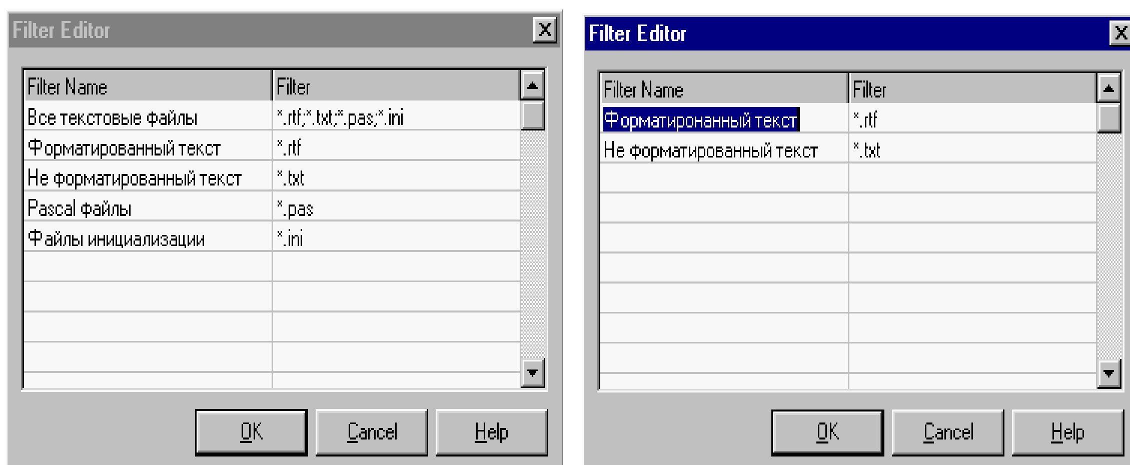

Рис.16

Создание пункта меню

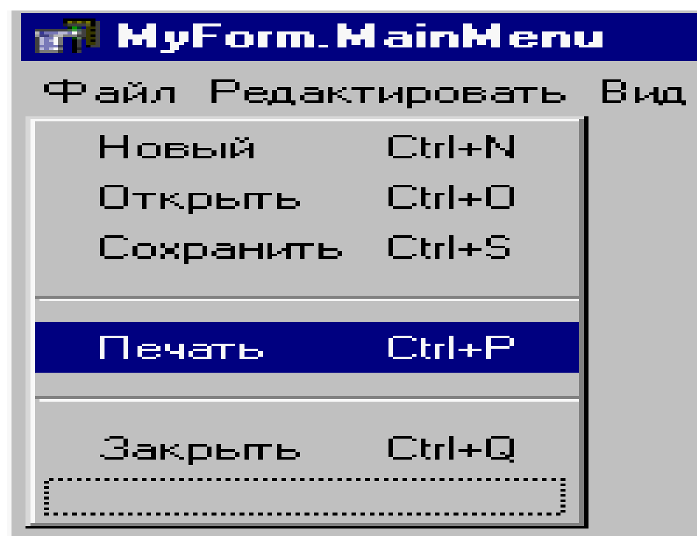

Рис.17

Далее напишите код обработчика события созданной команды:

```
{Печать документа}
procedure TMyForm.MnuPrintClick(Sender: TObject);
var
FileName: string;
begin
 if PrintDialog.Execute then
   Rich.Print(FileName);
end;
```

Для завершения дизайна приложения самостоятельно установите в инструментальную панель новую кнопку **TSpeedButton**. Присвойте ей имя **PrintBtn** и свяжите ее стандартный обработчик события с обработчиком события команды **Печать**. Также допишите код, который обеспечит изменение всплывающих подсказок и новых пунктов меню при выборе языка.

В заключении выполнения задания ознакомьтесь с новыми возможностями разработанного приложения. Откройте любой текстовый файл, например файл главного модуля проекта **Lab_1.pas** (рис.18) и выполните операции форматирования текста.

Новые возможности разработанного приложения

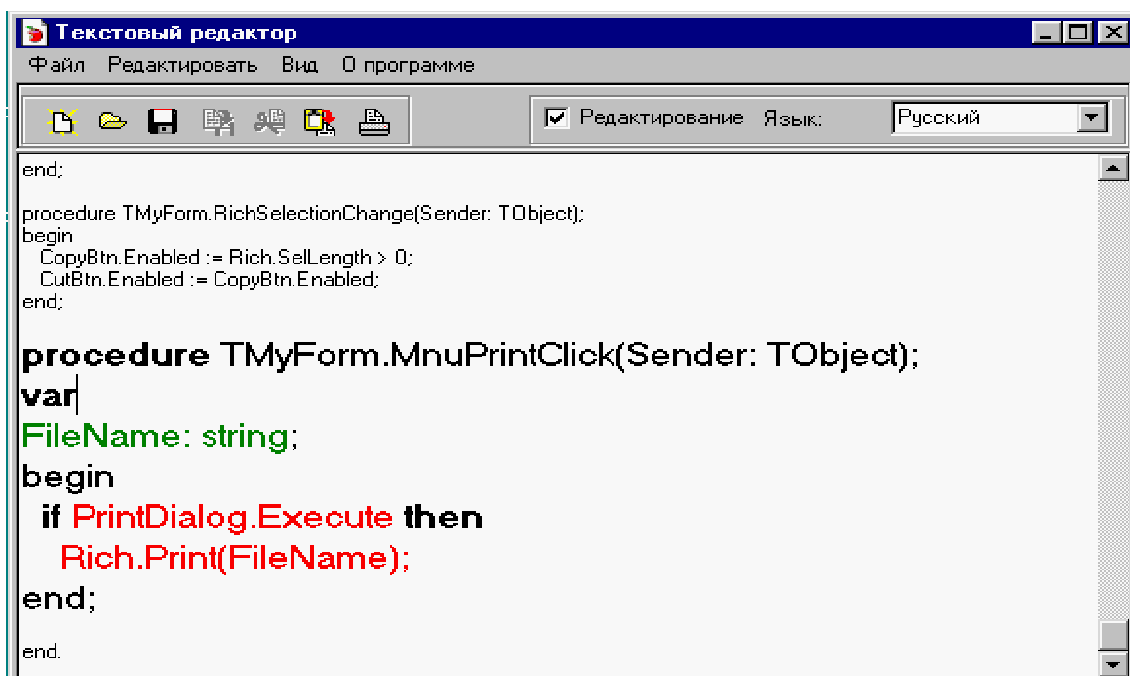

Рис.18

Література

1. Р. Боас, М. Фервай, Х. Гюнтер, Delphi 4 Полное Руководство, – Киев.: BHV, 1998. – 448с.
2. Developer's Guide for Delphi 3, Borland Inprise Corporation, 100 Enterprise Way, Scotts Valley, CA 95066-3249
3. Developer's Guide for Delphi 5, Borland Inprise Corporation, 100 Enterprise Way, Scotts Valley, CA 95066-3249
4. Object Pascal Language Guide, Borland Inprise Corporation, 100 Enterprise Way, Scotts Valley, CA 95066-3249
5. Анталогия Delphi, http://www.Torry.ru

ЗМІСТ

Тема 1 Основні конструкції мови Object Pascal ……….............. 3

Тема 2 Об'єктно-орієнтоване програмування …………...…....... 15

Тема 3 Створення методів у середовищі delphi …………………... 22

Тема 4 Конструювання текстового редактора ..………………….. 29

Література ……………..…………………………………………… 57

Навчальне видання

Швачич Геннадій Григорович
Овсяннiков Олександр Васильович
Кузьменко Вячеслав Віталійович
Нєчаєва Наталія Іванівна

Прикладне програмне забезпечення

Розділ «Основи програмування та конструювання прикладного програмного забезпечення в інтегрованому середовищі Delphi»

Навчальний посібник

Тем. план 2007, поз. 154

Підписано до друку 06.02.07. Формат 60х84 1/16. Папір друк. Друк плоский. Облік.-вид. арк. 3,41. Умов. друк. арк. 3,36. Тираж 100 пр. Замовлення №11

Національна металургійна академія України
49600, Дніпропетровськ – 5, пр. Гагаріна, 4

Редакційно – видавничий відділ НМетАУ