Министерство общего и профессионального образования Свердловской области
государственное автономное профессиональное образовательное учреждение
Свердловской области
«Ирбитский мотоциклетный техникум» (ГАПОУ СО «ИМТ»)

**ПРОГРАММА ПОДГОТОВКИ СПЕЦИАЛИСТОВ СРЕДНЕГО ЗВЕНА
ПО СПЕЦИАЛЬНОСТИ
09.02.04 Информационные системы (по отраслям)**

**УЧЕБНАЯ ДИСЦИПЛИНА
ОП.06 Основы алгоритмизации и программирования**

КОМПЛЕКС МЕТОДИЧЕСКИХ УКАЗАНИЙ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

Составитель: А.А. Лагунов, преподаватель ГАПОУ СО «ИМТ»

Методические рекомендации по выполнению лабораторных работ работы разработаны в соответствии с рабочей программой дисциплины.

Содержание

Пояснительная записка…………………………………………...……… 3
Комплекс лабораторных работ……………………………………….. 5

Пояснительная записка

Лабораторная работа - это важный элемент учебного процесса. Именно на таких занятиях студенты получают практические умения и навыки работы с программным обеспечением, лучше усваивают и закрепляют изученный теоретический материал.

Если лекция закладывает основы научных знаний в обобщенной форме, лабораторная работа призвана углубить, расширить и детализировать эти знания, содействовать выработке навыков профессиональной деятельности. Лабораторные работы развивают научное мышление и речь студентов при защите этой работы, позволяют проверить их знания, в связи с чем, лабораторные работы выступают важным средством достаточно оперативной обратной связи.

Для успешной подготовки к лабораторной работе студенту невозможно ограничиться слушанием лекций. Требуется предварительная самостоятельная работа студентов по теме планируемого занятия. Не может быть и речи об эффективности занятий, если студенты предварительно не поработают над конспектом, учебником, учебным пособием, чтобы основательно овладеть теорией вопроса.

Лабораторная работа служит своеобразной формой осуществления связи теории с практикой. Структура лабораторной работы в основном одинакова — вступление преподавателя, где осуществляется постановка задач на занятие, вопросы студентов по материалу, который требует дополнительных разъяснений, собственно практическая часть, защита выполненной работы и заключительное слово преподавателя. Цель занятий должна быть понятна не только преподавателю, но и студентам. Это придает учебной работе жизненный характер, утверждает необходимость овладения опытом профессиональной деятельности, связывает их с практикой жизни.

Студенты, как правило, отдают себе отчет в том, в какой мере им необходимы данные лабораторной работы для предстоящей профессиональной деятельности. Если студенты поймут, что все учебные возможности занятий исчерпаны, интерес к ним будет утрачен. Учитывая этот психологический момент, очень важно организовать занятия так, чтобы студенты постоянно ощущали рост сложности выполняемых заданий, что ведет к переживанию собственного успеха в учении и положительно мотивирует студента. Если же студенты замечают «топтание на месте», уровень мотивации может заметно снизиться.

Преподаватель должен проводить занятия так, чтобы каждый студент получил возможность «раскрыться», проявить способности, поэтому при разработке плана занятий и индивидуальных заданий преподаватель должен учитывать подготовку и интересы каждого студента. Преподаватель при этом будет выступать в роли консультанта, наблюдающего за работой каждого студента и способного вовремя оказывать педагогически оправданную помощь. При такой организации проведения занятий в лаборатории не возникает мысли о том, что возможности занятий исчерпаны.

При проведении лабораторных занятий особенно важно, как, впрочем, и в учении вообще, учитывать роль повторений. Однообразие заданий, субъективное ощущение повторения как замедления движения вперед значительно ухудшают усвоение. Поэтому важно не проводить повторения в формировании заданий на лабораторных работах.

Существуют различные формы проведения лабораторной работы с применением компьютера:

1.Работа с готовой программой.

2.Самостоятельное решение предлагаемой преподавателем задачи.

3.Моделирование и усложнение предлагаемой преподавателем программы.

Преподаватель выполняет консультирующую, координирующую и направляющую функцию. Очень высока степень самостоятельности учащихся, на нее отводится 70% времени занятия.

Комплекс лабораторных работ

ОСНОВЫ ПРОГРАММИРОВАНИЯ
на примере языка Turbo Pascal
Методические указания по лабораторным работам

2012

ОГЛАВЛЕНИЕ

Лабораторная работа №1. Разработка программы с линейным алгоритмом выполнения......... 7
Лабораторная работа №2. Разработка программы с разветвленной структурой 18
Лабораторная работа №3. Разработка циклической программы с известным количеством повторений.. 29
Лабораторная работа №4. Разработка циклической программы с неизвестным количеством повторений.. 39
Лабораторная работа №5. Разработка программы с использованием процедур и функций.... 46
Лабораторная работа № 6. Обработка символов и строк на языке Pascal................................ 54
Лабораторная работа № 7. Работа с файлами на языке Pascal 60
Список литературы 68

Лабораторная работа №1. Разработка программы с линейным алгоритмом выполнения.

1.1. Цель работы

Приобретение навыков по работе с интегрированной средой разработки Turbo Pascal. Изучение принципов разработки программ линейной структуры. Получение навыков объявления типов переменных в зависимости от характера входных и выходных данных.

1.2. Задание на лабораторную работу

1) Изучить основные возможности интегрированной среды разработки (ИСР) Turbo Pascal для подготовки текста программы и запуска ее на выполнение.

2) Изучить структуру программы на языке Pascal, способы объявления переменных, операторы ввода и вывода данных.

3) Разработать линейную программу в соответствии с вариантом задания.

1.3. Требования к программе

Программа должна выводить:

– номер варианта и сообщение о назначении программы;

– фамилию и инициалы автора программы;

– информационные сообщения о необходимости ввода данных;

– сообщение с результатами, полученными в ходе работы программы; при этом текст сообщения должен включать наименование результата и вычисленное значение результата, например «Полученная сумма: 123.45»; при использовании переменных, объявленных как дробные, значение результата должно быть представлено в отформатированном виде (с указанием количества знаков после десятичной точки).

1.4. Порядок выполнения работы

1. **Важно!** Изучить правила работы с методическими указаниями (п. 1.5).

2. Получить вариант задания (п. 1.10).

3. Изучить функции системы Turbo Pascal для подготовки текста (исходного кода) программы и запуска ее на выполнение. Освоить функции редактора для подготовки текста программы (п. 1.6).

4. Изучить структуру Pascal-программы, способы определения переменных стандартных типов и операторы ввода и вывода (п. 1.7).

5. Разработать программу в соответствии с вариантом задания и выполнить ее запуск с помощью среды Turbo Pascal (п. 1.8).

6. Показать разработанную программу преподавателю.

7. Устно ответить на контрольные вопросы преподавателя (п. 1.12).

8. Оформить отчет в соответствии с рекомендациями, данными в п. 1.11.

1.5. Правила работы с методическими указаниями

Важно! Обучаемый должен внимательно и вдумчиво читать описание каждой лабораторной работы!

Методические указания построены по принципу «от простого к сложному». В первой лабораторной работе обучаемый знакомится с очень важными элементами языка Pascal – операторами ввода и вывода. Ничего сложного в первой лабораторной работе нет, поэтому любой студент с ней справится без труда, если уделит ей некоторое время.

В результате обучаемый должен четко понимать принцип работы этих операторов. Если студент, в силу каких-либо обстоятельств, не сможет справиться с данной лабораторной работой, то нет смысла приступать к следующим, т.к. все они основаны на операторах ввода и вывода.

Нельзя приступать к следующей лабораторной работе, если не сделана предыдущая, т.к. все они взаимосвязаны. Каждая очередная лабораторная работа основана на предыдущей.

Перед тем, как приступить к выполнению лабораторной работы, необходимо ознакомится с ее описанием. Для каждой лабораторной работы дано достаточное для ее

выполнения описание. Оно дается на русском языке в наиболее понятном виде, поэтому предполагается, что обучаемый может самостоятельно с ним ознакомиться.

Методические указания содержат всю необходимую теоретическую часть, поэтому дополнительная литература по языку Pascal не требуется. Однако при необходимости студент может усилить свои знания с помощью дополнительной литературы, которую он без труда сможет найти в сети Интернет.

Если после прочтения методических указаний у обучаемого остаются вопросы, следует обратиться к преподавателю.

1.6. Разработка программы в интегрированной среде Turbo Pascal

1.6.1. Почему Pascal?

У обучаемого может возникнуть справедливый вопрос: «почему мне преподают Pascal, если в мире все программируют на 1С, Delphi, Java, C#, C++ VisualBasic, PHP и т.д.». Но давайте вспомним, как мы учились разговаривать, ходить, читать, писать! У многих первое слово было «мама», перед уверенной походкой ребенок ползал на четвереньках, нас учили читать и писать сначала по буквам, затем по словам, после чего мы научились формулировать свою мысль в виде предложения.

Нельзя сесть за штурвал пассажирского лайнера, не проведя годы тренировок на простых самолетах, но даже их доверяют далеко не сразу, а лишь спустя годы учебы.

Так и в программировании: не будет никакого смысла пытаться освоить сложные языки Delphi, Java, C#, если вы не изучили основы программирования.

Для изучения основ программирования необходимо использовать простой язык. Таким языком является Pascal. Первоначально он разрабатывался с одной целью – стать учебным языком программирования, однако решение вышло настолько удачным, что Pascal стал лучшим языком профессионального программирования и удерживал эту планку длительное время.

Язык Pascal очень легко преподавать и очень легко учить. При этом совершенно исключена такая ситуация: студент спрашивает преподавателя «что означает этот оператор», а преподаватель ему отвечает: «не спрашивай, просто так надо, тебе это сейчас бесполезно объяснять, все равно не поймешь, этот материал вы будете проходить через полгода». Такая ситуация является совершенно обычной, если для основ программирования преподаватель выбирает другой язык, например Си++. Язык Pascal проектировался таким образом, чтобы исключить саму возможность появления таких «неудобных» вопросов и столь «нелепых» ответов.

Несмотря на свою простоту, язык Pascal позволяет разрабатывать программы какой угодно сложности. Такие важные понятия, как «строгая типизация», «предварительное объявление» и др. позволяют сформировать у обучаемого очень хороший фундамент, благодаря которому в дальнейшем любой другой язык программирования можно освоить за месяцы, или даже за недели (особенно, если в языке отсутствует строгая типизация, например 1С).

Кроме того, Pascal является современным и полноценным языком, которым до сих пор пользуются многие программисты, например в средах Free Pascal, Lazarus, Delphi и др. Инструменты Free Pascal и Lazarus являются «кроссплатформенными», поэтому разработанная в них программа сможет функционировать практически в любой операционной системе (Windows, Linux, Unix, MacOS и т.д.) и на любой аппаратной платформе (персональные и карманные компьютеры, смартфоны, iPhone, iPod и т.д.).

В дальнейшем, если потребуется программирование «железа», то у обучаемого не должно возникнуть больших сложностей в изучении «родного» для железа языка – Си, Си++. Примечательно то, что для изучения языка Си чаще всего быстрее выучить сначала Pascal, а уже затем Си, чем сразу начинать с изучения языка Си.

1.6.2. Порядок создания программы

При выполнении лабораторных работ используется ИСР Turbo Pascal (допускается ИСР Free Pascal), обеспечивающая следующие возможности:

– подготовка текста программы;

– компиляция (перевод исходного кода программы в машинный код);

– запуск на выполнение;

Ниже представлены необходимые сведения о возможностях ИСР.

Запуск интегрированной среды Turbo Pascal

Запустить программу «Проводник» (щелкнуть «Пуск \ Мой компьютер»), выбрать диск, на котором расположены исполняемые файлы, найти каталог «TP», затем «BIN», затем найти файл TURBO.EXE (либо просто TURBO с пометкой «Приложение») и запустить его путем двойного щелчка левой кнопки мыши или нажатием клавиши ввод.

В результате на экране появится окно Turbo Pascal, как показано на рисунке 1.1.

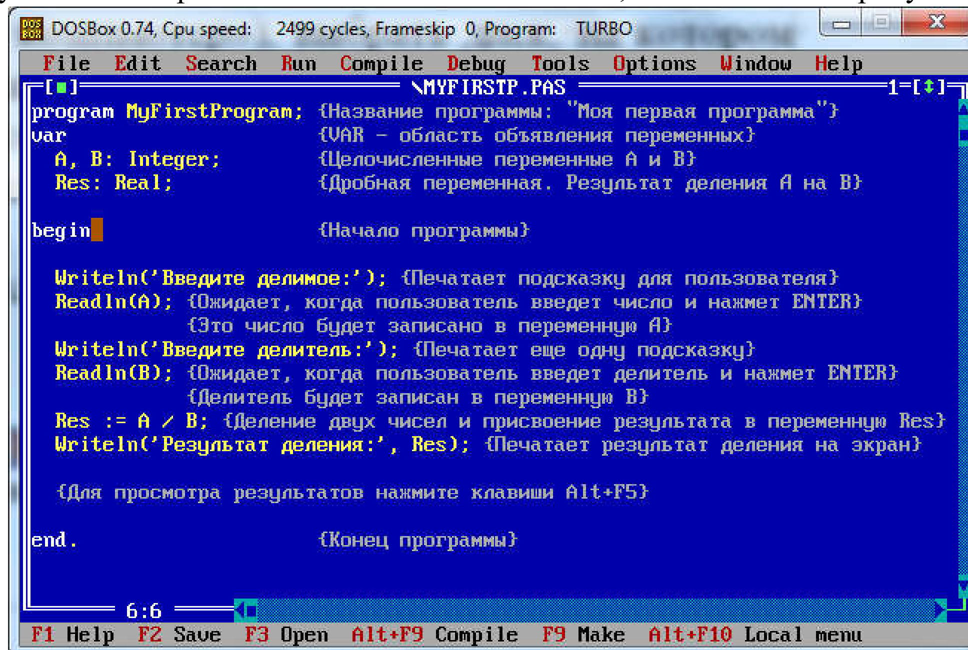


Рисунок 1.1 – Окно Turbo Pascal

Окно Turbo Pascal состоит из следующих элементов:

– пункты меню, расположенные в верхней части окна (File, Edit и т.д.);

– окно редактора кода, в котором следует вводить текст программы;

– строка подсказок в нижней части окна.

Назначение пунктов меню Turbo Pascal

– *File* – выполнение операций с файлами (открыть, сохранить и др.);

– *Edit* – команды редактирования текста (копировать текст, вставить текст, удалить текст и т.д.);

– *Search* – команды поиска;

– *Run* – запуск программы;

– *Compile* – компиляция программы;

– *Debug* – отладка программы;

– *Tools* – использование дополнительных программных инструментов;

– *Options* – содержит пункты, позволяющие выполнить гибкую настройку среды Turbo Pascal, включая компилятор, отладчик и редактор кода;

– *Window* – команды управления всеми открытыми окнами;

– *Help* – команды, обеспечивающие помощь программисту.

Для выполнения требуемой функции (т.е. для выбора меню) можно воспользоваться любым из способов, указанных ниже:

а) нажать клавишу F10, затем, перемещая клавишами ←, → курсор (выделенный прямоугольник), выбрать нужный пункт меню и нажать клавишу Enter;

б) одновременно нажать Alt и клавишу с выделенной буквой в нужном пункте меню; например, для раскрытия списка команд меню, содержащихся в пункте «File», следует нажать Alt+F;

в) щелкнуть мышкой на нужном пункте меню.

В дальнейшем, если, например, указано «выберите меню File \ Save», это означает, что необходимо любым из способов выбрать меню «File», а затем в раскрывшемся списке подчиненного меню выбрать пункт «Save».

Подготовка нового текста программы

Выбрать меню File \ New, после чего откроется новое окно редактора кода, которое можно использовать для набора текста программы.

Вся работа с текстом программы происходит в окне редактирования. При этом используются приемы, принятые во многих других текстовых редакторах (Блокнот, MS-Word, OpenOffice и др.). Самую подробную информацию о возможностях редактора кода Turbo Pascal можно узнать на английском языке в разделе помощи. Для этого нажмите F1 и в появившемся окне помощи выберите пункт *Using the editor*.

Описание клавиш редактирования текста

– *Ins* – переключение между режимами «вставка» и «замена». Рекомендуется работать в режиме «вставка» (при этом курсор представляется в виде мигающего символа подчеркивания «_»). Если по ошибке включен режим «замена» (случайно нажали клавишу Ins), то текст, расположенный справа от курсора, будет затираться.

– *Enter* – вставка новой строки; если перед нажатием Enter курсор находился в середине предложения, то оно в результате будет разделено и расположится на двух строках;

– *клавиши со стрелками* – сдвиг курсора в соответствующем направлении;

– *End* – перемещение курсора в конец строки;

– *Home* – перемещение курсора в начало строки;

– *PgUp* – сдвиг текста программы на страницу назад;

– *PgDn* – сдвиг текста программы на страницу вперед;

– *Ctrl+стрелка вправо* – перемещение курсора на слово вправо;

– *Ctrl+стрелка влево* – перемещение курсора на слово влево;

– *←(BackSpace)* – удаление символа слева от курсора;

– *Del* – удаление символа над курсором;

– *Ctrl+Del* – удаление выделенного участка текста;

– *Alt+BackSpace* – отменяет последнее изменение;

– *Ctrl+Ins* – копирование выделенного участка текста в буфер обмена;

– *Shift+Ins* – вставка текста из буфера обмена в позицию курсора.

Рекомендуется запомнить клавиши редактирования текста, поскольку благодаря этим клавишам работа с редактором кода значительно упрощается.

Сохранение текста программы в файл

Необходимо своевременно и регулярно сохранять исходный код программы. Для сохранения текста программы в файл следует**:**

– выбрать меню «File \ Change dir…» (Файл \ Сменить директорию) и указать диск и каталог, в который следует сохранить текст программы;

– выбрать меню «File \ Save as…» (Файл \ Сохранить как) и ввести имя файла в поле *Save file as* (по умолчанию Turbo Pascal устанавливает имя NONAMExx.PAS; необходимо указать другое наименование файла, используя при этом англоязычные символы);

В дальнейшем для сохранения изменений будет достаточно щелкнуть клавишу F2 (или выбрать меню File \ Save).

Загрузка текста программы из файла

– выбрать меню «File \ Change dir…» (Файл \ Сменить директорию) и указать диск и каталог, в котором находится pas-файл с сохраненным текстом программы;

– выбрать меню «File \ Open»; откроется окно с возможностью выбора имени файла программы (это окно можно вызвать также путем нажатия клавиши F4); нажмите Tab для перехода к списку файлов, выберите необходимый файл и нажмите Enter. Исходный код программы, содержащийся в указанном файле, будет загружен в окно редактора кода.

Запуск программы на выполнение

Для запуска программы на выполнение воспользуйтесь комбинацией клавиш Ctrl+F9. Будет произведена компиляция программы (преобразование исходного текста программы в машинный код), в ходе выполнения которой на экране будет отображаться окно компиляции «Compiling». Если в ходе компиляции обнаружены ошибки, то выдается сообщение об ошибке и курсор указывает место, где необходимо внести исправления (см. п. 1.9). Если ошибок не обнаружено, то программа будет запущена на выполнение. При этом программа выводит пользователю подсказки для ввода данных, осуществляет необходимые вычисления и печатает на экране результаты своей работы.

Просмотр результатов работы программы

После того, как программа полностью выполнилась, на экране вновь появится окно редактора кода. Если вы не успели увидеть результаты работы программы, то нажмите клавиши Alt+F5: в результате будет снова открыто окно с результатами работы программы.

Выход из среды Turbo Pascal

Выход из Turbo Pascal выполняется с помощью нажатия клавиш Alt+X или выбора пункта меню File \ Exit. Если программа не была сохранена или были внесены изменения после ее сохранения, то вам будет предложено сохранить текст программы.

1.6.3. Использование помощи в среде Turbo Pascal

Для помощи программисту в интегрированной среде Turbo Pascal предусмотрена обширная система справки (пункт Help главного меню).

Для открытия окна Help можно:

– нажать F1 (в любой момент времени);

– установить курсор под словом и нажать клавиши Ctrl+F1 для получения справочной информации по выбранному слову.

Для закрытия окна Help нажмите Esc.

1.7. Сведения по структуре и операторам Pascal-программы

Структура программы на языке Pascal имеет следующий вид:

```
program ProgramName; {Имя программы}
  {Раздел описаний}
begin           {BEGIN – это начало программы}
  {Раздел операторов}
end.            {END. (с точкой) – это конец программы}
```

Вначале должно быть указано имя программы (после ключевого слова PROGRAM), затем следует раздел описаний. **После** раздела описаний следует раздел операторов, расположенный внутри ключевых слов BEGIN и END. В конце программы, после ключевого слова END должна стоять точка.

Ниже представлен пример самой простой программы на языке Pascal, которая выводит на экране надпись «Hello World!» (Привет, мир!):

```
program Hello;
begin
  Writeln('Hello, World!'); {Печать текста «Hello, World!» на экране}
end.
```

Рекомендуется напечатать текст простой программы в среде Turbo Pascal и запустить ее на выполнение (см. п. 1.6.2). Для просмотра текста, напечатанного на экране, необходимо нажать Alt+F5.

Внимание! Если установленная у вас версия Turbo Pascal не поддерживает русский язык, то для вывода сообщений и печати комментариев используйте транслитерацию с помощью английских символов. Например, вместо сообщения «Введите любое число» вы можете написать «Vvedite luboe chislo». Если вы в достаточной мере владеете английским языком, то можно написать «Enter any number». В методических указаниях для вывода сообщений и комментариев используется преимущественно русский язык, но это совершенно не принципиально.

Ниже приведен пример программы, требующей от пользователя ввести некоторое число, далее это число возводится в квадрат и полученное значение печатается на экране:

```
program Kvadrat;   {Название программы: Квадрат}
var            {VAR - область объявления переменных}
  X, Xkvadr: Real; {Объявление переменных X и Xkvadr}
begin {Начало программы}
  Writeln('Введите любое число:'); {Печатает подсказку для пользователя}
  Readln(X); {Ожидает, когда пользователь введет число и нажмет ENTER}
          {Число будет записано в переменную X}
```

```
  {Возведение X в квадрат и присвоение результата в переменную Xkvadr}
  Xkvadr := X * X;

  {Печатает результат возведения в квадрат на экран: }
  Writeln('Результат возведения в квадрат:', Xkvadr)

  {Для просмотра результатов нажмите клавиши Alt+F5}
end. {Конец программы}
```

Ниже представлены необходимые пояснения к данному примеру:

1) Выделенные жирным шрифтом слова PROGRAM, VAR, BEGIN, END являются «ключевыми словами». Ключевые слова определяют структуру программы, они располагаются в строго определенных местах. Turbo Pascal выделяет ключевые слова определенным цветом (см. рисунок 1.1). Вы не можете объявить свою переменную с таким же именем.

2) Имя программы (Kvardat), имена переменных (X, Xkvadr), а также наименования любых объектов на языке Pascal являются «идентификаторами» и могут состоять только из латинских символов, цифр от 0 до 9 и знака подчеркивания «_». Длина идентификатора не должна быть более 32 символов, наименование идентификатора не может начинаться с цифры. Примеры правильных идентификаторов:

Laba1, **My_First_Program**, **Kolvo1**, **RadiusKruga**.

Неправильные идентификаторы: **1Laba** (цифра спереди), **Lab rab 1** (пробел недопустим), **ПерваяПрограмма** (русские символы недопустимы).

3) Каждая строка в примере снабжена подробным комментарием. **Комментарии** в программе являются поясняющим текстом, основная цель которого – не дать программисту забыть назначение тех или иных участков его программы даже по прошествии длительного времени. Для оформления комментариев служат фигурные скобки, например *{ Это комментарий }* или круглые скобки со звездочками, например *(* Это тоже комментарий *)*.

4) Каждый оператор (т.е. инструкция, выполняющая какое-либо действие) должен заканчиваться символом «;». Однако, если оператор расположен непосредственно **перед** END, то символ «;» не является обязательным. Символ «;» в языке Pascal отделяет операторы друг от друга.

5) При выводе сообщения на экран (с помощью оператора Writeln) текст должен располагаться внутри одинарных кавычек (апострофов). На клавиатуре символ апострофа «'» расположен на той же клавише, где буква «Э», рядом с «ENTER». Пример: **Writeln('Введите любое число:')**.

6) При использовании оператора Writeln, осуществляющего печать заданного текста на экране, все параметры должны находиться в круглых скобках. Если необходимо вывести на экран несколько значений (текстовых и числовых), то между ними должна стоять запятая «,».

7) При использовании оператора Readln, требующего от пользователя ввести необходимые данные, переменная, в которую будет записано введенное пользователем число, должна находиться в круглых скобках. Кроме того, с помощью одного оператора Readln можно потребовать от пользователя ввести сразу несколько чисел. В этом случае в программе должно быть объявлено соответствующее количество переменных, а в операторе Readln эти переменные должны быть перечислены через запятую «,».

8) Перед тем, как с помощью оператора Readln требовать от пользователя ввести какие-либо данные, необходимо с помощью оператора Writeln информировать пользователя о том, какие данные ему нужно ввести.

9) Оператор присваивания «:=» работает следующим образом: сначала вычисляется выражение, расположенное справа от оператора присваивания, затем полученный результат записывается в переменную, расположенную слева от оператора присваивания. Эта переменная должна быть использована в дальнейших операторах программы, например в операторе Writeln для вывода ее значения на экран.

10) Числовое выражение, расположенное справа от оператора присваивания, может включать математические операции сложения «+», вычитания «–», умножения «*», деления «/», целочисленного деления «div» (отбрасывается остаток), получения остатка от целочисленного

деления «mod». Операции умножения и деления являются более приоритетными, чем операции сложения и вычитания. Как и в математике, приоритетом вычисления можно управлять с помощью круглых скобок, например: Y := ((A + B) / (A – B)) * X.

В языке Turbo Pascal отсутствует операция возведения в степень, однако эту «проблему» можно легко обойти. Например оператор Y := X * X * X возводит число X в 3-ю степень и записывает результат в переменную Y.

11) Перед использованием любой переменной ее необходимо объявить в разделе описаний. Если переменная не объявлена, то ее нельзя указывать в операторах Writeln, Readln, а также в любых других операторах (будет выдаваться ошибка «Unknown identifier», см. п. 1.9).

12) Переменные в языке Pascal объявляются в секции VAR. Сначала указывается имя переменной, затем символ двоеточия «:», после чего следует тип переменной и завершается строка символом «;». В одной секции VAR можно объявить любое количество переменных. Если нужно объявить несколько переменных с одинаковым типом, то можно перечислить эти переменные через запятую. Пример:

```
var
  N: Integer;
  S, M1, M2: Real;
```

Тип INTEGER соответствует целым числам. Переменная N, объявленная целочисленной, может принимать любое **целое** значение от -32768 до 32767. Эти ограничения связаны с тем, что для любой переменной в языке Pascal выделяется некоторый ограниченный объем памяти. Для переменной, имеющей тип INTEGER, выделается 2 байта (16 бит) памяти. Кроме того, в языке Pascal вы можете для объявления целочисленной переменной использовать следующие типы:

Наименование типа	Диапазон значений	Размер памяти
Shortint	-128 … 127	1 байт
Longint	-2147483648 ...2147483647	4 байта
Byte	0 ... 255	1 байт
Word	0 ... 65535	2 байта

Тип REAL соответствует дробным (вещественным) числам. Это означает, что переменная S может иметь любое числовое значение, как целое, так и дробное, например: 0, 1, 1.5, -100, -123.12313 и т.д. В качестве разделителя целой и дробной части используется точка, а не запятая, как это принято в математике. Для переменной, объявленной как REAL, выделяется 6 байтов памяти, диапазон значений: от $-1.7E^{38}$ до $+1.7E^{38}$. Особенность вещественных чисел заключается в том, что они хранят дробные числа с ограниченной точностью. Например, число 3.3333333… (в периоде) **в принципе** невозможно хранить с бесконечной точностью (понадобилось бы бесконечное количество байтов).

Внимание! Вы должны хорошо понимать, какие действия выполняет каждый из операторов в данном примере. Если есть непонятные моменты, обратитесь к преподавателю за разъяснениями.

При выполнении данной лабораторной работы для вывода на экран сообщений (подсказок пользователю) можно использовать операторы типа:

```
Write('Сообщение о вводе данных');
Writeln('Сообщение о выводе результатов');
```

Оператор **Write** выводит на экран заданный текст и оставляет мигающий курсор на той же строке (в конце).

Оператор **Writeln** также печатает на экране заданный текст, но при этом переводит курсор на следующую строку.

Для ввода данных с клавиатуры можно использовать операторы типа:

```
Read(S1);
Readln(A, B, C);
```

В первом случае при выполнении оператора **Read(S1)** программа будет ждать, пока пользователь не введет с клавиатуры какое-либо число и нажмет Enter. После этого мигающий курсор останется на той же самой строке, а введенное пользователем число будет записано в переменную S1.

Во втором случае программа будет ожидать от пользователя, когда он введет друг за другом три числа, разделяя их пробелом, и нажмет Enter. Курсор перейдет на следующую строку, а введенные значения будут сохранены соответственно в переменные A, B и C.

Ниже представлен пример ввода данных: программа печатает приглашение для ввода данных, а пользователь на той же самой строке должен ввести число и нажать Enter. После этого курсор перейдет на следующую строку, а в переменную R будет записан радиус круга:

```
Write('Введите радиус круга: ');
Readln(R);
```

Для вывода результатов вычислений на экран можно использовать операторы типа:

```
Write(Res:8:2);        {Только если Res – дробная переменная}
Writeln(Res:8:2);
```

где «8» – общее количество символов, отведенных для представления числового значения на экране (включая знак числа, целую часть, десятичную точку и дробную часть), «2» – количество знаков после десятичной точки.

В том случае, если переменная-результат объявлена как целочисленная, подобное форматирование не требуется.

Пример вывода целочисленного (Integer) результата:

```
Writeln('Периметр прямоугольника: ', Per);
```

Пример вывода вещественного (Real) результата:

```
Writeln('Объем конуса V: ', V:8:2);
```

1.8. Пример действий по подготовке и выполнению программы

Запустите Turbo Pascal и введите текст своей программы, например:

```
{Программа вычисления суммы двух чисел}
program FirstProgram;
var
  A, B: Real; {вводимые данные}
  Sum: Real;  {сумма чисел }
begin
  Writeln('Программа вычисления суммы двух чисел');
  Writeln('Автор: Иванов И.И.');
  Write('Введите два любых числа через пробел: ');
  Readln(A, B);
  Sum := A + B; {вычисление суммы двух чисел... }
  Writeln('Сумма: ', Sum:8:2);
end.
```

Для сохранения программы выберите меню File \ Save (или F2).

Для выполнения программы следует выбрать Run \ Run в главном меню (или нажать Ctrl+F9). При наличии ошибки см. п. 1.9.

Пример работы описанной выше программы

1) На экран выводятся сообщения, которые заданы в операторах Writeln вашей программы:

> Программа вычисления суммы двух чисел
> Автор: Иванов И.И.
> Введите два любых числа через пробел:

2) Пользователь вводит через пробел два числа и нажимает Enter.

3) На экран выводится следующее сообщение:

> Сумма:

а за ним – вычисленная сумма двух введенных чисел.

Для просмотра результатов работы программы, выберите меню Debug \ User Screen (Отладка \ Окно пользователя) или нажмите Alt+F5.

1.9. Ошибки компиляции

Если в процессе компиляции текста программы Turbo Pascal обнаружит ошибку, то курсор будет автоматически перемещен на строку с ошибкой, а в красной рамке появится сообщение об ошибке в формате:

Error <код_ошибки>: Сообщение об ошибке на английском языке

Наиболее вероятные ошибки компиляции:

Error 2: Identifier expected {Ожидается идентификатор}
Error 3: Unknown identifier {Неизвестный идентификатор}
Error 4: Duplicate identifier {Идентификатор дублируется}
Error 12: Type identifier expected {Ожидается тип идентификатора}
Error 26: Type mismatch {Несоответствие типов}
Error 36: BEGIN expected {Ожидается BEGIN}
Error 37: END expected {Ожидается END }
Error 62: Division by zero {Деление на 0}
Error 85: ";" expected {Ожидается ";"}
Error 86: ":" expected {Ожидается ":"}
Error 87: "," expected {Ожидается ","}
Error 88: "(" expected {Ожидается "("}
Error 89: ")" expected {Ожидается ")"}
Error 91: ":=" expected {Ожидается ":="}
Error 94: "." expected {Ожидается "."}
Error 113: Error in statement {Ошибка в операторе}

1.10. Варианты заданий

Перечень задач:

1) Вычислить площадь треугольника. $S = \frac{1}{2}bh$

2) Вычислить площадь круга. $S = \pi R^2$

3) Вычислить площадь трапеции. $S = (a+b) * \frac{h}{2}$

4) Вычислить площадь квадрата. $S = \frac{1}{2}d^2$

5) Вычислить объем куба. $V = a^3$

6) Вычислить среднее арифметическое трех чисел.

7) Вычислить квадрат числа.

8) Вычислить периметр треугольника. $P = a + b + c$

9) Вычислить периметр прямоугольника. $P = 2(a+b)$

10) Вычислить квадрат разности двух чисел.

11) Вычислить квадрат суммы двух чисел.

12) Вычислить объем шара. $V = \frac{4}{3}\pi R^3$

13) Вычислить длину средней линии трапеции. $m = \frac{a+b}{2}$

14) Вычислить процент от заданного числа.

15) Вычислить куб разности двух чисел.

16) Вычислить объем цилиндра. $V = \pi R^2 h$

17) Вычислить объем конуса. $V = \frac{1}{3}\pi R^2 h$

18) Вычислить объем треугольной пирамиды. $V = \frac{1}{3} S_{osn} h$, где $S_{osn} = \frac{\sqrt{3}}{4} a^2$

19) Вычислить разность квадратов двух чисел.

20) Вычислить длину радиуса окружности заданной площади. $r = \sqrt{A\pi}$

1.11. Содержание отчета

1) Цель работы.

Допускается использовать формулировку, приведенную в соответствующем пункте методических указаний на выполнение лабораторной работы.

2) Задание на лабораторную работу.

Привести задание на лабораторную работу, требования к программе, указать номер варианта и задание из варианта.

3) Состав и структура данных.

Привести состав и тип входных и выходных данных. Если имеются ограничения по диапазону возможных значений (например, ноль в знаменателе, корень из -1 и т.п.), то указать их.

4) Описание программы.

Привести словесное описание программы, из которого должно быть понятно, какие операции выполняет программа и какие действия производит пользователь для ввода данных и получения результата.

5) Схема алгоритма программы.

Схема алгоритма программы должна быть составлена с учетом особенностей каждой лабораторной работы. Пример схемы алгоритма для лабораторной работы №1 приведен на рисунке 1.2:

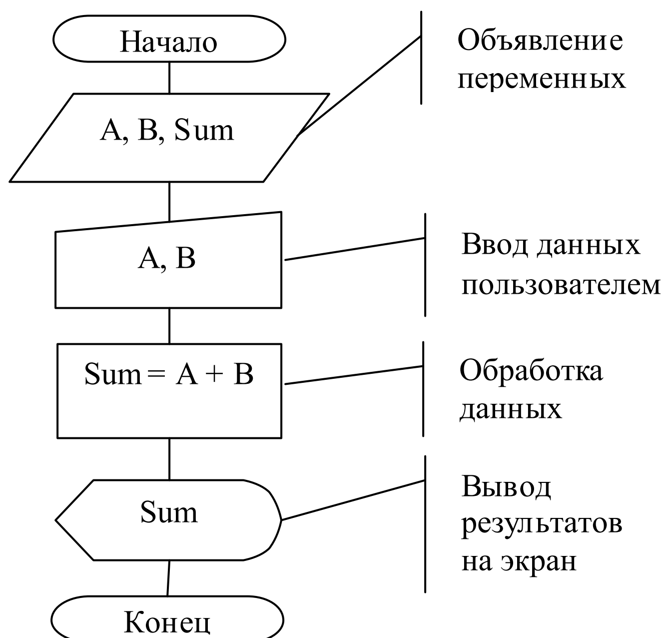


Рисунок 1.2 – Схема алгоритма программы

Для разработки схемы алгоритма программы допускается использовать любое программное средство, предоставляющее соответствующие возможности, в том числе Diagram Designer, MS Word, MS Visio.

6) Текст программы.

*Должен быть представлен полный текст разработанной программы. В том случае, если весь текст программы целиком вмещается в окне Turbo Pascal, допускается привести в отчете скриншот данного окна. Текст программы должен удовлетворять требованиям к программе (п. 1.3). Необходимо обеспечить выравнивание операторов в тексте программы. Для лабораторной работы №1 ключевые слова **program**, **const**, **var**, **begin**, **end** не должны иметь отступов от левого края, а все остальные строки кода должны иметь отступ от левого края – 2 пробела (см. п. 1.8).*

7) Входные данные и результаты работы программы.

Привести скриншот окна Turbo Pascal с результатами работы программы. Для получения скриншота можно воспользоваться клавишами Alt+PrintScreen

8) Ответы на контрольные вопросы.

Письменно кратко ответить на контрольные вопросы.

9) Выводы по проделанной работе.

Привести выводы по результатам выполненной лабораторной работы. Выводы – это краткое отражение наиболее значимых результатов выполненной работы. Для лабораторных работ допускаются выводы, совпадающие с основными положениями, приведенными в пункте «Цель работы».

1.12. Контрольные вопросы

1) Каков порядок создания программы в ИСР Turbo Pascal?

2) Какие основные функции выполняет ИСР Turbo Pascal?

3) Какие операции позволяет выполнять текстовый редактор Turbo Pascal при подготовке программы?

4) Как запустить программу, разработанную в ИСР Turbo Pascal?

5) Как сохранить программу под другим именем?

6) Как открыть новое окно редактирования?

7) Какими способами можно выйти из среды?

8) Как вызвать контекстную помощь?

9) Какова структура Pascal-программы?

10) Какие операторы используются для ввода и вывода значений переменных?

11) Какие действия выполняют операторы Write и Writeln?

12) Какие действия выполняют операторы Read и Readln?

13) Каким образом можно объявить переменную в языке Pascal?

Лабораторная работа №2.
Разработка программы с разветвленной структурой

2.1. Цель работы

Приобретение навыков разработки программ разветвленной структуры на языке Pascal с использованием операторов условного перехода IF, выбора CASE и безусловного перехода GOTO. Получение навыков использования встроенного отладчика.

2.2. Задание на лабораторную работу

1. Освоить основные функции встроенного отладчика интегрированной среды Turbo Pascal (п. 2.5).

2. Разработать программу с разветвленной структурой в соответствии с предложенным вариантом. Ввод данных, вычисления и вывод результатов организовать в диалоговом режиме с использованием оператора CASE (п. 2.7).

2.3. Требования к программе

Программа должна выполнять следующие действия:

– вывод номера варианта и сообщения о назначении программы;

– вывод фамилии и инициалов автора программы;

– вывод меню;

– ввод данных;

– вычисления и вывод результатов.

Результаты работы выводятся в отформатированном виде (с указанием количества знаков после десятичной точки).

2.4. Порядок выполнения работы

1. Получить вариант задания (п. 2.8).

2. Изучить основные функции отладчика Turbo Pascal (п. 2.5).

3. Подготовить текст программы и выполнить ее отладку с использованием интегрированной среды Turbo Pascal (п. 2.5, 2.6, 2.7).

4. Во время отладки использовать не менее двух контрольных точек останова (п. 2.5). Проверить работу программы при различных значениях исходных данных.

5. Ответить на контрольные вопросы (п. 2.10).

6. Оформить отчёт (п. 1.11).

2.5. Отладка программы с использованием встроенного отладчика Turbo Pascal

Очень часто при разработке программ возникает ситуация, при которой программа компилируется и запускается без ошибок, но в дальнейшем она выдает совершенно другой результат, нежели от нее ожидал программист. Это свидетельствует о том, что **программист допустил логическую ошибку**. Примеры логических ошибок:

– вместо Readln(N) написано Writeln(N);

– вместо R := A * B написано A := R * B;

– пропущен оператор или выражение;

– и т.д.

Перечень всех возможных логических ошибок, которые может совершить программист, является бесконечным. **Ошибка может быть допущена где угодно!** В большинстве случаев для исправления подобных ошибок достаточно просмотреть код программы и найти строку, в которой содержится ошибка. В тех ситуациях, когда программист подобным способом не может исправить допущенную ошибку, выручает отладчик. В состав интегрированной среды Turbo Pascal входит встроенный отладчик (debugger), обеспечивающий следующие возможности:

– построчная отладка (трассировка) текста программы с возможностью просмотра и изменения значения любой переменной (или выражения);

– автоматическое приостановление работы программы (с возможностью просмотра значений переменных или выражений) при достижении точки останова (breakpoint), либо строки, на которой установлен курсор;

– наблюдение за переменными и выражениями, включенными в список «Watches»;

– доступ к регистрам микропроцессора;

– просмотр цепочки вызова подпрограмм (call stack).

Для начального изучения основ работы с отладчиком выполните следующие шаги:

1) Набрать простую программу, например:

```
1:   program DebugTest;
2:   var
3:     A, B, R: Integer;
4:   begin
5:     Write('Введите A, B: ');
6:     Readln(A, B);
7:     R := A + B;
8:     R := R * A;
9:     Writeln('Результат: ', R);
10:  end.
```

В данной программе объявлены 3 целочисленные переменные A, B, R. Вначале программа печатает на экране подсказку «Введите A, B: », затем от пользователя требуется ввести 2 числа, которые будут записаны соответственно в переменные A и B. Далее программа складывает оба этих числа, и полученный результат присваивает в переменную R.

В следующей строке (№8) содержится оператор **R := R * A**, при этом сначала переменная R умножается на переменную A, после этого результат умножения присваивается опять в ту же самую переменную R. Таким образом, переменная R используется в данном примере несколько раз. В программировании повторное использование одних и тех же переменных встречается очень часто.

2) Для запуска программы под отладчиком следует щелкнуть клавишу F7 (меню Run \ Trace Into) или F8 (меню Run \ Step over). При отладке данного примера различия между F7 и F8 отсутствуют, однако если программа содержит процедуры или функции (см. лабораторную работу №5), созданные программистом, то F7 позволит попасть внутрь процедуры для ее отладки. В дальнейшем изложении будет упоминаться только клавиша F7. В результате нажатия клавиши F7 строка №4 (begin) окажется выделенной зеленым цветом, что свидетельствует об успешном запуске сеанса отладки.

3) Щелкнуть клавишу F7 еще раз – будет выделена строка №5 (Write…). Это означает, что при **очередном** нажатии F7 будет выполнен оператор, расположенный на выделенной строке.

4) Щелкнуть F7. В результате оператор Write будет выполнен, а дальнейшее управление перейдет к строке №6 (Readln).

5) Щелкнуть F7. Произойдет переключение в режим текстового ввода и программа будет ожидать от пользователя ввода двух чисел через пробел. После нажатия Enter в переменные A и B будут записаны числовые значения, которые ввел пользователь, а дальнейшее управление перейдет к строке №7 (R := A + B).

6) На данном этапе с помощью отладчика можно проверить, действительно ли переменные A и B содержат значения, введенные пользователем. Для этого следует установить курсор под переменной A, затем выбрать меню Debug \ Evaluate/modify (определить значение / изменить) или щелкнуть Ctrl+F4. Откроется окно «Evaluate and modify», в котором указано текущее значение переменной A (рисунок 2.1)

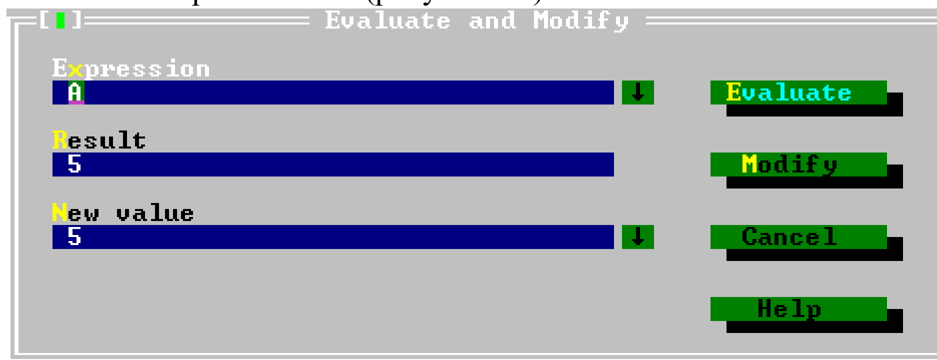


Рисунок 2.1 – Окно «Evaluate and modify»

В поле Expression можно указать имя переменной, но при необходимости можно ввести целое математическое выражение, например: (A * B) / (A + B). После нажатия кнопки «Evaluate» заданное выражение будет вычислено, а результат появится в строке «Result». При

необходимости (в отладочных целях) в данном окне можно изменить значение указанной переменной. Для этого в строке «New value» следует ввести новое значение и нажать Modify.

7) Щелкнуть F7. В результате выражение в строке №7 будет вычислено, а результат записан в переменную R. Для просмотра вычисленного значения переменной R следует установить на нее курсор и нажать Ctrl+F4.

8) В некоторых случаях отладка упрощается, если программист всегда видит на экране значение заданного выражения (без необходимости постоянного нажатия Ctrl+F4). Для этого следует установить курсор на переменную R, щелкнуть меню Debug \ Add watch (Ctrl+F7) и нажать ОК. В результате в нижней части окна Turbo Pascal появится окно «Watches», в котором присутствует наименование переменной и ее значение, вычисленное автоматически (рисунок 2.2). При необходимости в список «Watches» можно добавить произвольное количество переменных или выражений.

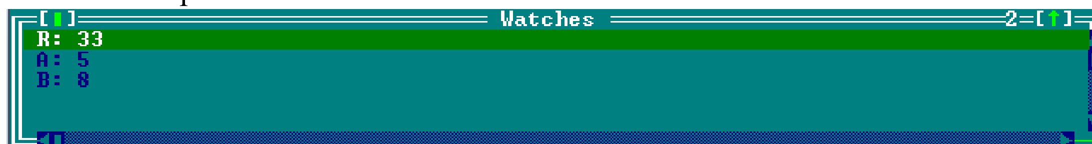


Рисунок 2.2 – Окно «Watches»

9) Нажать F7 и убедиться в том, что значения заданных переменных в окне «Watch» обновляются автоматически.

10) Прервать отладку программы можно в любой момент с помощью меню Run \ Program Reset (Ctrl+F2).

11) Очень часто к трассировке программы требуется приступить не с момента ее запуска, а только при достижении определенной строки, подозреваемой в наличии ошибки. Для этого на заданную строку следует поместить точку останова. Разместите курсор на строке №8 (R := R * A), щелкните по ней правой кнопкой мыши и выберите пункт Toggle Breakpoint или нажмите Ctrl+F8. В результате строка будет выделена красным цветом, что свидетельствует о наличии точки останова (повторное нажатие Ctrl+F8 позволяет снять точку останова).

12) Запустить программу с помощью меню Run \ Run (Ctrl+F9), ввести два числовых значения через пробел и нажать Enter. В результате управление перейдет к строке, на которую установлена точка останова (теперь она будет окрашена зеленым цветом), а Turbo Pascal перейдет в режим отладки программы, в котором доступны все ранее описанные действия.

13) При необходимости к отлаживаемой программе можно добавить несколько точек останова (например, при отладке программы с разветвляющейся структурой, переход в ту или иную ветвь которой может произойти при выполнении или невыполнении некоторого условия). Для просмотра списка всех точек останова следует выбрать меню Debug \ Breakpoints. С помощью данного окна можно изменить (Edit) параметры точек останова (критерии срабатывания точки останова, например количество проходов через нее или выполнение заданного условия), быстро перейти к строке исходного кода (View), удалить выбранную точку останова (Delete), либо удалить все точки останова (Clear all).

14) Если в программе планируется использование всего одной точки останова, то в качестве более простой альтернативы можно использовать команду выполнения программы до выделенной строки. Для этого установите курсор на строке №9 (Write) и выберите меню Run \ Go to cursor (F4). В результате Turbo Pascal перейдет в режим отладки, в котором можно совершать все действия, упомянутые ранее.

2.6. Справочная информация по операторам ветвления языка Pascal

2.6.1. Оператор IF

При разработке любой программы необходим механизм, позволяющий выполнить тот или иной участок программного кода, основываясь на некотором условии. Для реализации такого рода алгоритмов в Pascal предусмотрен оператор ветвления IF. Основная форма вызова данного оператора:

```
if <условие> then <оператор>;
<остальные операторы программы>
```

Принцип действия данного оператора следующий: осуществляется проверка заданного условия. В том случае, если условие выполняется (дает результат «ИСТИНА», т.е. «TRUE»), то осуществляется переход к оператору, расположенному справа от THEN. После окончания

работы этого оператора управление переходит к остальным операторам программы, которые расположены после символа «;». Если же условие не выполняется, то оператор, расположенный справа от THEN, пропускается и управление сразу переходит к остальным операторам.

Очень часто, в случае выполнения заданного условия, требуется выполнить не один, а сразу несколько операторов. В этом случае эти несколько операторов следует разместить внутри операторных скобок BEGIN..END.

Внимание! В программе может находиться несколько участков кода, размещенных внутри BEGIN..END, причем эти участки могут быть вложенными. Самый внешний BEGIN..END определяет начало и окончание программы, а остальные BEGIN..END выполняют роль операторных скобок.

Следует учитывать, что если некоторый набор операторов, состоящий из нескольких строк, находится внутри операторных скобок BEGIN..END, то всю конструкцию (с точки зрения оператора IF) следует рассматривать как «составной оператор»:

```
if A = B then
begin
  <оператор 1>;
  <оператор 2>;
  ...
  <оператор N>
end;
```

Условие – это любое выражение, результатом которого является логическое (Boolean) значение: **True** или **False** (Да или Нет). При составлении условия можно использовать операторы сравнения:

«=» – равно	«<>» – не равно	«>» – больше
«<» – меньше	«>=» – больше или равно	«<=» – меньше или равно

Операция (A = B) вернет **True** в том случае, если переменные A и B равны между собой; в противном случае вернет **False**;

Операция (A <> B) вернет **True** только в том случае, если переменные A и B не равны между собой.

Операция (A > B) вернет **True** в том случае, если переменная A имеет значение большее, чем значение переменной B.

Операция (A >= B) вернет **True** в том случае, если переменная A **равна** переменной B, **либо** имеет значение большее, чем у переменной B.

Кроме того, операции сравнения, возвращающие значения True или False можно комбинировать между собой с использованием логических операций:

«and» – логическое И	**«or»** – логическое ИЛИ
«not» – логическое НЕ	**«xor»** – исключающее ИЛИ

Выражение ((A < -1) **or** (A > 1)) вернет **True**, **если** значение переменной A **меньше** -1 **или больше** 1, т.е. оно не лежит в диапазоне [-1 ÷ 1].

Выражение ((A = B) **and** (A > C * C)) вернет **True**, если переменные A и B равны между собой и в то же время значение переменной A превышает значение переменной C, взятое в квадрате. Если не будет выполнено хотя бы одно из условий, то выражение вернет **False**.

Операция «**not**» инвертирует результат логического выражения, указанного справа, т.е. (**not True**) вернет **False**, а (**not False**) вернет **True**.

Выражение (**not** (A = B)) вернет True в том случае, если A не равно B.

Операция «**xor**» вернет True в том случае, одно из выражений дает **True**, а другое **False**. Например, выражение ((100 > 50) **xor** (29 = 30)) вернет **True**, поскольку одно из сравнений (100 > 50) дает True, а другое (29 = 30) – **False**.

Внимание! Программа на языке Pascal всегда должна иметь наименование, указанное после ключевого слова PROGRAM, секцию VAR с объявленными переменными и **внешние** операторные скобки BEGIN..END, в которых должны располагаться все необходимые операторы. Для краткости изложения материала эти ключевые слова в некоторых дальнейших

примерах пропущены! Также пропущен код, осуществляющий присвоение значения некоторым переменным.

Ниже представлен один из таких примеров. В нем осуществляется ввод числа A; если введенное число отрицательное, то его значение заменяется на ноль.

```
Readln(A);          {ввод числа A}
if A < 0 then A := 0; {заменяем на 0, если число отрицательное}
Writeln('A: ', A);    {выводим на экран значение переменной A}
```

В примере ниже осуществляется проверка делителя на 0. При равенстве нулю выдается сообщение об ошибке и происходит выход из программы.

```
program Delenie;
var
  Delimoe, Delitel, Res: Real;
begin
  Delimoe := 100;
  Readln(Delitel);            {ввод значения делителя}
  if Delitel = 0 then         {проверка на равенство нулю}
  begin                   {начало составного оператора}
    Writeln('Ошибка: на ноль делить нельзя!');
    Exit;                   {досрочный выход из программы}
  end;                    {составной оператор закончился}
  Res := Delimoe / Delitel;    {осуществляем деление}
  Writeln('Результат: ', Res);  {вывод результата на экран}
end.
```

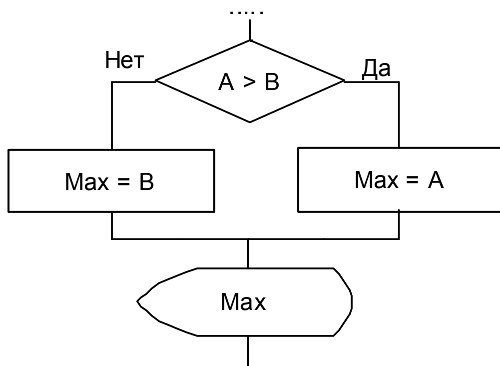
Очень часто возникает необходимость отреагировать не только на выполнение условия, но и на его невыполнение. Для этого к конструкции IF..THEN добавляется ключевое слово ELSE, определяющее начало альтернативной ветви выполнения программного кода. В этом случае конструкция IF..THEN выглядит следующим образом:

```
if <условие> then
  <оператор1>
else
  <оператор2>;
```

```
if <условие> then
begin
  <группа операторов 1>;
end
else
begin
  <группа операторов 2>;
end;
```

Следует отметить, что перед ключевым словом ELSE **не должна** стоять точка с запятой.

На рисунке 2.3 приведен пример схемы разветвляющегося алгоритма для двух форм оператора IF..THEN: с ключевым словом ELSE и без него.



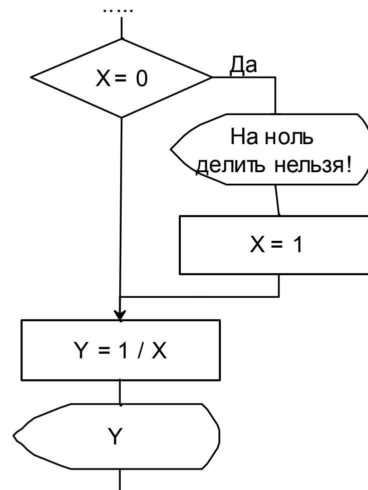


Рисунок 2.3 – Схема алгоритма с оператором **if…then** с **else** (слева) и без **else** (справа)

В представленном ниже примере осуществляется поиск наибольшего значения среди X и Y и сохранение найденного значения в переменную Max:

```
if X > Y then
  Max := X
else
  Max := Y;
Write('Максимум: ', Max);
```

```
if X >= Y then
begin
  Max := X;
  Write('X больше или равен Y');
end
else
begin
  Max := Y;
  Write('Y больше X');
end;
Write('Максимум: ', Max);
```

Кроме того, оператор IF..THEN может быть вложенным, причем уровень вложенности не ограничивается, например:

```
if X < -3 then
  Y := X + 1
else if (X > 3) and (X < 10) then
  Y := X * X
else if X >= 10 then
begin
  Y := Sqrt(X);
  Writeln('Y: ', Y);
end
else
  Y := Y * Y;
```

Следует отметить, что подобные конструкции на практике могут быть весьма громоздкими. Для улучшения читабельности кода рекомендуется чаще пользоваться операторными скобками BEGIN..END с необходимым выравниванием. Важно помнить, что ключевое слово ELSE относится только к одному, ближайшему оператору IF..THEN, расположенному выше по коду.

2.6.2. Константы

Перед тем, как познакомиться с оператором CASE, необходимо дать определение понятию «константа». Константой в языке Pascal является некоторое значение (например, числовое), заданное непосредственно в тексте программы (т.е. пользователь вашей программы его не вводит). Например, в операторе «A := 100» переменной A присваивается явно заданное значение «100», т.е. константа. В языке Pascal различают два вида констант: именованные и неименованные. Для того чтобы константа была именованной, ее необходимо указать в секции CONST в разделе описаний программы: сначала указывается имя константы, затем символ «=», далее указывается необходимое значение, например:

```
program ConstExample;
const
  MinLimit = 1;     {Минимальный лимит}
  MaxLimit = 100;   {Максимальный лимит}
  Pi = 3.14;        {Число Пи}
.......
begin
  A := MaxLimit;    {Это более осмысленно, чем A := 100}
  if B < MinLimit then...
.......
end;
```

После того, как константа объявлена, ее имя можно использовать в программе вместо числового значения, например «A := MaxLimit». В некоторых случаях это позволяет улучшить читабельность программы, а также упростить ее дальнейшую разработку. Значение именованной константы невозможно изменить при выполнении программы (в отличие от переменной).

В данной лабораторной работе рекомендуется использовать константы для обозначения постоянных параметров, которые не требуется вводить пользователю вашей программы. Например, стоимость 1 кВт/час является величиной постоянной, поэтому вы можете ее объявить с помощью именованной константы:

```
const
  KiloWattCost = 3.45;     {Стоимость 1 кВт/час }
```

2.6.3. Оператор выбора CASE

В том случае, если задана некоторая переменная порядкового типа (целочисленная, логическая или символьная) и на каждое возможное ее значение программа должна отреагировать индивидуально, рекомендуется использовать оператор выбора CASE. Логика работы оператора CASE аналогична логике IF..THEN, однако, использование оператора CASE в **некоторых случаях** позволяет значительно улучшить читабельность кода.

Оператор CASE имеет следующий формат:

```
case <переменная или выражение порядкового типа> of
  <константа или список констант 1> : <оператор 1>;
  <константа или список констант 2> : <оператор 2>;
  …………………………
  <константа или список констант n> : <оператор N>;
else
  <альтернативная ветвь: оператор или группа операторов>
end;
```

Логика работы оператора CASE следующая: сначала программа определяет значение переменной или выражения порядкового типа (например, целочисленное). Далее отыскивается константа, совпадающая с указанным значением, после чего выполняется оператор, расположенный после символа «:». Если программе не удалось найти константу, совпадающую с заданным значением, то выполняется оператор из альтернативной ветви, расположенной после ключевого слова ELSE. Ключевое слово ELSE не является обязательным (его следует указывать, когда в этом возникает необходимость).

В приведенном ниже примере пользователь вводит степень числа N от 1 до 3. Программа возводит переменную X в степень N. Отдельно обрабатывается случай, когда N равен нулю. Во всех остальных случаях устанавливается значение 0.

```
  Write('Введите значение n: ');
  Readln(n); {ожидаем, когда пользователь введет n}
  case n of
    0: {демонстрация использования операторных скобок begin...end}
      begin
        Writeln('Сообщение: любое число в степени 0 равно единице!');
        Y := 1;
      end;
    1: Y := X;
    2: Y := X * X;
    3: Y := X * X * X;
  else {альтернативная ветвь кода}
    {здесь дополнительный begin...end не требуется}
    Writeln('Вы ввели недопустимое число!');
```

```
  Y := 0;
end; {конец оператора case}
Writeln('Результат:', Y); {вывод результата на экран}
```

Кроме одиночных констант могут быть заданы списки и/или диапазоны значений. Например:

```
case n of
  0, 2..4 : Y := A * B; {оператор будет выполнен для n: 0, 2, 3, 4}
  1, 5    : Y := A / B;
  6       : Y := (A + B) * (A - B);
end;
```

Следует отметить, что при использовании оператора CASE действует ряд ограничений:

– значения констант не должны дублироваться; это ограничение действует также при использовании диапазонов;

– тип констант должен соответствовать типу заданной переменной; если переменная целочисленная, то и все константы должны быть целыми;

– заданная переменная должна иметь порядковый тип (например, Integer, Byte, Char, Boolean); она не может быть объявлена как Real (дробный тип) или **string** (строка).

2.6.4. Оператор GOTO

Оператор безусловного перехода GOTO (англ.: **перейти к**) позволяет прервать выполнение текущего участка кода и перейти к другому участку, если он отмечен меткой безусловного перехода. Метка безусловного перехода объявляется в разделе LABEL и должна соответствовать требованиям, предъявляемым к идентификаторам (в порядке исключения допускается вместо наименования метки использовать целочисленные значения). После объявления метки в разделе LABEL ее можно указать в любом месте (но только один раз) в тексте программы. Для того чтобы перейти на заданную метку, следует вызвать оператор GOTO <имя_метки>, причем количество операторов перехода на одну и ту же метку в программе не ограничено:

```
………………………
goto M1; {переходит вниз на метку M1}
<операторы>;
M1:
<операторы>;
………………………
goto M1; {переходит вверх на метку M1}
```

```
………………………
label
  M1;
var
  X, Y: Real;
begin
  Readln(X);
  ………………………
  goto M1;
  ………………………
M1:
  Y := X * 2 – 3 / X;
  Writeln('Y=', Y);
  ………………………
end.
```

Следует отметить, что в современном программировании использование оператора GOTO не приветствуется, поскольку злоупотребление данным оператора приводит к сильному «запутыванию» кода. Для избежания использования оператора GOTO следует применять другие методы, например циклы и подпрограммы (см. следующие лабораторные работы).

2.7. Пример программы с разветвленной структурой

Составить программу вычисления функции:

$$y = \begin{cases} \frac{1}{x} & \text{при } x > 0 \\ x^2 & \text{при } x < 0 \end{cases}$$

```
program Lab2;
label
  M1, M2; {объявление меток}
var
  n: Integer;
  X, Y: Real;
  Flag: Boolean;      {Признак выполнения пункта N1}
begin
  Writeln('Программа вычисления функции. Автор: Иванов И.И.');
  { Вывод на экран меню }
  Writeln('Введите цифру для выполнения действия:');
  Writeln('1 - Ввод данных');
  Writeln('2 - Вычисление функции и вывод результатов');
  Writeln('3 - Завершение работы программы');
  Flag := False; { Первоначальная инициализация флага }
M1:
  Write('Введите номер пункта меню: ');
  Readln(n); { Ввод номера пункта меню}
  case n of
    1: { Ввод данных }
      begin
      M2:
        Write('Введите значение аргумента X: ');
        Readln(X);
        { Проверка допустимости значения аргумента }
        if X = 0 then
        begin
          Writeln('X не может быть равным 0 по условию');
          goto M2; { переход к M2 для повторного ввода данных }
        end;
        Flag := True; {Пункт №1 выполнен, установка флага в True}
      end;
    2: { Вычисление значения функции }
      begin
        if not Flag then {Если пункт №1 не выполнен}
          Writeln ('Данные не введены, выполните пункт №1');
        else
        begin {пункт №1 был выполнен}
          { Операторы вычисления и вывода значения функции  }
          if X > 0 then {если X положительный}
            Y := 1 / X
          else {иначе X < 0}
            Y := X * X;
          Writeln('При X  =  ', X:7:2, ' Y  =  ', Y:7:2);
        end;
      end;
    3: Exit; { Выход из программы }
  end; { end case }
  goto M1; { переход в режим выбора пункта меню }
end. { Конец программы}
```

Данный пример требует пояснения. Программа начинается с объявления меток безусловного перехода (M1, M2) и объявления переменных, в том числе логической однобайтной переменной **Flag: Boolean**. Как ранее было сказано, логическая переменная может иметь всего два значения: **True** или **False**. В начале работы программы переменной **Flag**

присваивается значение **False**. Это необходимо, поскольку если не выполнить этого присвоения, то в начале работы программы значение переменной **Flag** не определено, т.е. она случайным образом может быть равна False или True. Следует обратить внимание, что для пункта №2 оператора CASE осуществляется проверка переменной Flag (**if not** Flag **then** …), а поскольку осуществляется обращение к переменной в режиме чтения, то значение переменной обязательно должно быть присвоено **заранее**. В приведенном примере проверка (**if not** Flag **then** …) будет препятствовать выполнению операторов вычисления до тех пор, пока пользователь в пункте №1 не введет допустимое значение аргумента X (Flag в этом случае будет выставлен в **True**).

<u>Обратите внимание</u> на выравнивание операторов в приведенных примерах! Очень легко поддается пониманию программа, в которой все операторы выровнены надлежащим образом.

Представьте, что в предыдущем примере не было произведено выравнивание операторов, например:

```
case n of
   1:.......
             2:
begin
  if not Flag then
Writeln ('Данные не введены, выполните пункт №1');
 else
     begin
 if X > 0 then
          Y := 1 / X
    else
  Y := X * X;
Writeln('При X = ', X:7:2, ' Y = ', Y:7:2);
             end;
      end;
            3: Exit;
end;
```

В таком виде разобраться с данным примером практически невозможно. Выравнивание должно быть выполнено таким образом, чтобы вложенные группы операторов имели отступ вправо относительно внешних операторов (рекомендуется 2 пробела). Помните, что каждый BEGIN должен иметь завершающий END, причем их следует располагать на одной вертикальной линии. Все, что находится внутри операторных скобок BEGIN..END, должно смещаться вправо.

2.8. Варианты заданий

1) Вычислить объем параллелепипеда со сторонами A, B, C и определить, является ли данное геометрическое тело кубом.

2) Вычислить площадь треугольника со сторонами A, B, C. Перед вычислением площади проверить условие существования треугольника с заданными сторонами.

3) Вычислить площадь треугольника со сторонами A, B, C. Определить, является ли треугольник равнобедренным.

4) Вычислить площадь прямоугольника со сторонами A и B и определить, является ли данная фигура квадратом.

5) Составить программу нахождения корней квадратного уравнения $y=ax^2+bx+c$.

6) Определить, можно ли сделать круглую заготовку с заданным радиусом R из квадратного листа фанеры с заданной стороной A.

7) Определить, хватит ли имеющейся суммы S на покупку N-го количества товара (при известной цене товара).

8) Определить, можно ли сделать две квадратных заготовки со стороной А из листа железа прямоугольной формы со сторонами В и С.

9) Определить, достаточно ли имеющейся ткани для изготовления изделий двух видов, если известны: расход ткани на каждое изделие, количество изделий каждого вида, количество имеющейся ткани.

10) Рассчитать сумму оплаты за потребленную энергию, если известны: стоимость 1 квт/час, расход энергии, коэффициент льгот (льготы могут отсутствовать).

11) Определить, достаточно ли бензина для поездки, если известны: длина пути, количество бензина в баке и расход бензина на 1 км.

12) Определить, будет ли начислена студенту стипендия по результатам экзаменов (стипендия начисляется, если все экзамены сданы на «хорошо» и «отлично»), если известны оценки по всем экзаменам.

13) Определить, будет ли зачислен абитуриент в студенты по результатам вступительных экзаменов, если известны: проходной балл; количество баллов, набранных абитуриентом по каждому экзамену.

14) Определить, изделия какой из двух групп товаров выгоднее сшить из одного рулона ткани, если известны: расход ткани на каждое изделие и цена готового изделия, количество метров в рулоне.

15) Определить, выполнен ли план по продаже товара за день, если известны: план продажи (в рублях), количество проданного товара и цена товара.

16) Определить, операции с какой из двух валют составили большую прибыль, если известны: курс продажи, количество продаж по каждой валюте.

17) Определить наибольшую выручку от продажи трех видов товаров, если известны: цена товара; количество проданных товаров каждого вида.

18) Определить количество корней уравнения $y=ax^2+bx+c$.

19) Определить, является ли число А четным или нечетным.

20) Вычислить площадь треугольника со сторонами А, В, С и определить, является ли данный треугольник равносторонним.

2.9. Содержание отчета (см. пункт 1.11)

2.10. Контрольные вопросы

1) Назначение, формы записи и порядок выполнения оператора IF.
2) Особенности использования вложенных условных операторов.
3) Каковы отличия оператора выбора CASE от оператора условия IF?
4) Какие правила должны выполняться при использовании оператора выбора CASE?
5) Назначение и особенности оператора безусловного перехода.
6) Для чего нужна отладка программы?
7) Как выполнять программу в пошаговом режиме?
8) Как поставить и как отменить точки останова?
9) Как выполнить программу «до курсора»?
10) Как открыть окно Watch?
11) Как внести переменную в окно Watch?

Лабораторная работа №3.
Разработка циклической программы с известным количеством повторений

3.1. Цель работы

Целью работы является освоение процесса разработки циклических программ с заданным (известным) числом повторений на языке Pascal.

3.2. Задание на лабораторную работу

Разработать программу с использованием оператора цикла FOR, осуществляющую следующие действия:

а) Вычисление заданной величины (суммы N слагаемых, произведения N сомножителей и т.п.);

б) Обработка данных с использованием двумерных массивов.

3.3. Требования к программе

Программа должна выводить:

– номер варианта, назначение программы и ФИО автора;

– меню выбора действия программы («а» или «б»); допускается сделать две отдельные программы без меню выбора;

– информационные сообщения о необходимости ввода данных;

– сообщение с результатами, полученными в ходе работы программы;

3.4. Порядок выполнения работы

1) Получить вариант задания (п. 3.8).

2) Изучить принцип действия FOR при разработке циклических программ на языке Pascal (п. 3.5).

3) Составить и отладить программу вычисления заданной величины (суммы N слагаемых, произведения N сомножителей и т.п.) в соответствии с подпунктом «а» варианта задания.

4) Изучить основы работы с данными типа «массив» (ARRAY) на языке Pascal (п. 3.6).

5) Разработать и отладить программу обработки данных с использованием двумерных массивов в соответствии с подпунктом «б» варианта задания.

6) Объединить обе разработанные программы в одну с использованием меню выбора (п. 3.7).

7) Ответить на контрольные вопросы (п. 3.10).

8) Оформить отчет (см. п. 1.11)

3.5. Оператор цикла FOR

Очень часто перед программистом стоит задача организовать работу программы таким образом, чтобы некоторый участок кода выполнялся многократно (с заданным количеством повторений). В языке Pascal для этой цели предусмотрен оператор FOR, имеющий 2 формы записи:

1) for I := N1 **to** N2 **do** <тело цикла>;
2) for I := N1 **downto** N2 **do** <тело цикла>;

Цикл FOR работает следующим образом. Вначале программа определяет значение переменных (выражений) N1 и N2. Далее переменная цикла **I** (счетчик цикла) получает начальное значение N1. После этого осуществляется сравнение переменной **I** с конечным значением N2 и если переменная **I** его не превышает (для первой формы FOR) либо если **I** больше или равно N2 (для второй формы FOR), то выполняется «тело цикла», т.е. заданный оператор (или группа операторов в BEGIN..END). После того, как тело цикла было выполнено, управление вновь переходит к оператору FOR, переменная **I** получает новое значение (на единицу больше, либо на единицу меньше, чем в прошлый раз), далее осуществляется ее сравнение с N2 и принимается решение о том, следует ли еще раз выполнить тело цикла.

Ниже представлены примеры для обеих форм оператора FOR. Программа осуществляет перемножение заданной переменной **Num** с текущим значением счетчика цикла **J** и выводит на экран результат вычисления выражения, порядковый номер итерации и значение счетчика цикла.

```
var
  Num, J: Integer;
```

```
begin
  Num := 3;
  {объявляем цикл «вверх» от 1 до 5}
  for J := 1 to 5 do
  begin
    {Выводим результат умножения на экран }
    Writeln(J, ') ', Num, '*', J, '=', Num * J);
    {Выполняем умножение}
    Num := Num * J;
  end;
end.
```

Результаты:
1) 3*1=3
2) 3*2=6
3) 6*3=18
4) 18*4=72
5) 72*5=360

```
var
  Num, J, K: Integer;
begin
  Num := 7;
  K := 1; {порядковый № итерации}
  {объявляем цикл «вниз» от 9 до 6}
  for J := 9 downto 6 do
  begin
    {Выводим результат умножения на экран}
    Writeln(K, ') ', Num, '*', J, '=', Num * J);
    {Выполняем умножение}
    Num := Num * J;
    K := K + 1; {Увеличиваем порядковый номер}
  end;
end.
```

Результаты:
1) 7*9=63
2) 63*8=504
3) 504*7=3528
4)3528*6=21168

Поскольку во втором примере используется оператор FOR с **downto**, а переменная J (счетчик цикла) изменяет свое значение от 9 до 6, то для определения порядкового номера итерации пришлось добавлять новую переменную «K». В первом примере этого не требовалось, поскольку счетчик цикла J изменяется от 1 до 5, а эти значения соответствуют номеру итерации.

Следующий пример демонстрирует использование цикла FOR для вычисления суммы $\sum_{i=1}^{n} ix^2$ (слева) и произведения $\prod_{i=1}^{n}\left(x+i^3\right)$ (справа):

```
const
  n = 5; {Количество шагов}
var
  I: Integer;
  Res, X: Real;
begin
  X := 10; {Ввод X}
  Res := 0; {Обнуление Res}
  for I := 1 to n do
    Res := Res + I * (X * X);
  Writeln('Res = ', Res:8:2);
end.
```

```
const
  n = 4; {Количество шагов}
var
  I: Integer;
  Res, X: Real;
begin
  X := 15; {Ввод X}
  Res := 1; {Присвоение единицы}
  for I := 1 to n do
    Res := Res * (X + I * I * I);
  Writeln('Res = ', Res:8:2);
end.
```

При использовании оператора FOR необходимо учитывать следующие замечания:

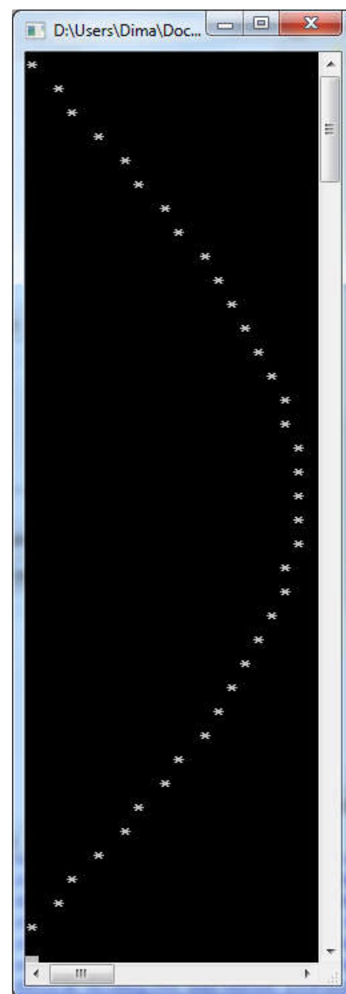
– переменная цикла (счетчик числа), а также выражения N1 и N2 должны иметь порядковый тип. Дробный тип REAL не допускается. В большинстве случаев для объявления счетчика цикла используется тип INTEGER;

– внутри тела цикла запрещено изменение переменной цикла;

– значение переменной цикла является актуальным только внутри тела цикла; после завершения работы цикла значение переменной цикла является неопределенным, т.е. установленным случайным образом;

– переменную цикла можно использовать повторно, например, при программировании других циклов.

Следующий пример демонстрирует печать на экране участка функции косинуса Cos. При этом используются 2 цикла FOR: внешний (со счетчиком цикла **I**) и внутренний (со счетчиком цикла **J**):




```
program Cosinus;
const
  Pi = 3.14159; {Константа Pi}
  {Границы внешнего цикла в градусах}
  MaxGradus = 90;
var
  I, J: Integer;
  Rad, Value: Real;
begin
  {Внешний цикл по I (от -90 до +90)}
  for I := -MaxGradus to MaxGradus do
  begin
    {Учитываются только градусы, кратные 5}
    if I mod 5 = 0 then
    begin
      {Пересчет градусов в радианы}
      Rad := Pi * I / 180;
      {Вычисление косинуса. В результате Value}
      {будет иметь значение от 0 до 1}
      Value := Cos(Rad);

      {Внутренний (вложенный) цикл (по J)}
      {Печатает на экране от 1 до 20 пробелов}
      {Round округляет число до целого}
      for J := 1 to Round(Value * 20) do
        Write(' ');

      {Печать символа «*»}
      Writeln('*');
    end;
  end;
  Readln; {ожидание нажатия ENTER}
end.
```

Ниже представлены необходимые пояснения к данному примеру:

1) Переменная внешнего чикла I меняет на каждой итерации значение, начиная с –90 и заканчивая +90. Этот диапазон выбран не случайно, поскольку функция Cos будет иметь в этом диапазоне только положительные значения от 0 до 1.

2) Операция «**mod**» возвращает остаток от деления двух целых чисел. Условие «**if** I **mod** 5 = 0 **then**» будет выполнено только в том случае, если значения I кратны 5 (т.е. -90, -85, -80,…, -10, -5…0, 5, 10…, 85, 90). Таким образом, символ «*» будет печататься на экране не для каждого градуса. Это сделано для того, чтобы весь график косинуса поместился на экране. Вместо 180 символов «*» на экране печатается только 37 символов.

3) Cos – это функция, принимающая в качестве аргумента величину угла, выраженную в радианах, и возвращающая значение косинуса в диапазоне от –1 до +1. Перед вызовом функции «Cos» осуществляется преобразование градусов в радианы: **Rad := Pi * I / 180**.

4) В примере присутствует дополнительный цикл FOR со счетчиком J. Он расположен внутри тела другого цикла, т.е. является «вложенным» или «внутренним». Он служит для заполнения строки пробелами. В зависимости от вычисленного значения косинуса, внутренний

цикл выполняется от нуля до 20 раз, т.к. переменная Value в данном примере может иметь значения от 0 до 1.

5) Для печати символа «*» используется оператор Writeln.

6) Оператор Readln, указанный в программе без скобок, приостанавливает программу до тех пор, пока пользователь не нажмет Enter. При этом никакого ввода данных не происходит. Это избавляет от необходимости нажимать Alt+F5 для просмотра результатов работы программы.

Важно отметить, что наименование переменной вложенного цикла должно отличаться от переменной внешнего цикла. Глубина вложенности циклов FOR может быть любой, однако на практике не приветствуется слишком большая вложенность, поскольку ухудшается читабельность кода. В таких случаях рекомендуется использовать методы процедурного программирования.

При необходимости можно досрочно прервать работу цикла. Для этого следует вызвать оператор **Break**. В том случае, если требуется пропустить операторы тела цикла и перейти к следующей итерации, следует вызвать оператор **Continue**. Не рекомендуется злоупотреблять оператором **Continue**, поскольку его использование ухудшает читабельность кода, тем более из любой ситуации можно выйти, не используя **Continue**. Оператор **Break**, вызванный из вложенного цикла, не прерывает работу внешнего цикла.

На рисунке 3.1 представлен пример участка схемы алгоритма, соответствующего циклу с известным числом повторений.

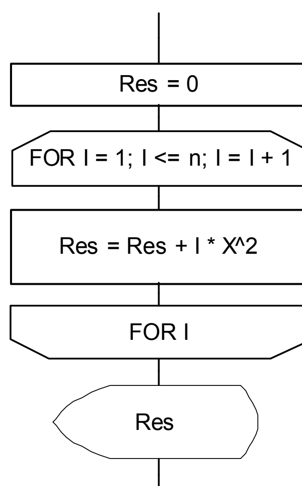


Рисунок 3.1 – Схема алгоритма участка программы с циклом **for**

Возведение в степень

В языке Pascal отсутствует операция возведения в степень. Для возведения любого числа в целую, положительную степень вы можете использовать следующий код:

```
{Пример демонстрирует возведение числа X в степень N}
program Stepen;
var
  X, Res: Real;
  N, I: Integer;
begin
  {*** Ввод данных ***}
  Write('Введите значения X и N:');
  Readln(X, N);
  {*** Возведение в степень ***}
  Res := 1;          {Присваиваем начальное значение переменной Res}
  for I := 1 to N do  {Цикл возведения в степень}
    Res := Res * X;
  {*** Вывод результата на экран ***}
  Writeln('X^N=', Res:5:2);
end.
```

Цикл возведения в степень целесообразно применять, если значение степени заранее неизвестно.

Если требуется возвести число в квадрат или в куб, то проще поступить следующим образом: **Y := X * X** или **Y := X * X * X**.

Для возведения числа «–1» в целую, положительную степень рекомендуется следующий код:

```
if Odd(N) then {Если число нечетное... }
  Y := –1
else        {Иначе (если число четное)... }
  Y := 1;
```

Odd – это функция, возвращающая логический результат **True**, если аргумент является нечетным числом, иначе возвращает **False**.

3.6. Тип данных «массив» – объявление и использование

Массив — упорядоченный набор данных для хранения множества элементов одного типа, идентифицируемых с помощью одного или нескольких индексов. Количество используемых индексов массива может быть различным. Массивы с одним индексом называют одномерными, с двумя — двумерными и т.д. Одномерный массив соответствует вектору в математике, двумерный — матрице. Массивы с числом измерений > 2 встречаются редко.

Формат объявления переменных типа «массив» представлен ниже:

```
var
  <список переменных> = array[<индексы>] of <тип данных>;
```

где **<список переменных>** – наименование одной или нескольких переменных (через запятую);

<индексы> – описание диапазона индексов для каждого измерения массива, например:

- [1..10] – соответствует одномерному массиву (вектору), первый элемент которого имеет индекс «1», а последний «10» (всего 10 элементов);

- [0..4] – соответствует одномерному массиву, первый элемент которого имеет индекс «0», а последний «4» (всего 5 элементов);

- [1..5, 1..4] – соответствует двухмерному массиву (матрице), состоящему из 5 строк (индексация от 1 до 5) и 4 столбцов (индексация от 1 до 4);

<тип данных> – любой тип данных, известный компилятору Pascal, например, Integer, Byte, Real, Boolean, Char, string, а также типы данных, объявленные программистом.

Ниже приведены примеры объявления переменных типа «массив»:

```
{2 вектора из 10 элементов типа Integer. Индексация начинается с «1»}
Vect1, Vect2: array[1..10] of Integer;
{Матрица вещественных чисел 8 x 6. Индексация начинается с «0»}
Matr: array[0..7, 0..5] of Real;
{Список из 20 строк. Индексация начинается с «1»}
StrList: array[1..20] of string;
```

Во многих случаях более удобным является однократное объявление типа данных «массив» в начале программы (в секции **type**). Объявленный таким способом тип данных в дальнейшем можно использовать при описании переменных. Данный способ имеет следующие преимущества:

- при необходимости внесения изменений в объявление массива достаточно это сделать в одном месте;

- переменные, объявленные в секции **var** с использованием отдельного типа, являются совместимыми, т.е. между ними допускается операция присвоения «:=»;

- при использовании в программе процедур и функций у программиста появляется возможность объявления параметров процедуры, имеющих тип «массив».

Ниже представлен пример объявления нового типа данных «массив», а также переменных этого типа:

```
type
  {Объявление нового типа данных «матрица»}
  TMatrix = array[1..10, 1..20] of Real;
```

```
var
  {Объявление переменных типа TMatrix}
  Matrix1, Matrix2, Matrix3: TMatrix;
.......................................
```

Для любого элемента массива можно узнать его текущее значение, а также записать новое значение. Каждый элемент массива имеет свои координаты, указать которые можно в квадратных скобках. Ниже приведены примеры чтения и записи элементов одномерного массива:

```
{Установка нового значения 3-го элемента вектора}
Vect[3] := 3.45; {Установка заданного числового значения}
Vect[3] := Random; {Установка случайного значения}
Readln(Vect[3]); {Значение вводится пользователем с клавиатуры}
{Чтение 2-го элемента вектора}
Value := Vect[2]; {Присвоение в переменную Value}
Writeln(Vect[2]); {Вывод на экран}
{Вычисление суммы первых трех элементов}
Summa := Vect[1] + Vect[2] + Vect[3];
```

Ниже приведены примеры чтения и записи элементов двухмерного массива (матрицы):

```
{Установка заданного значения элемента матрицы с координатами 4, 3}
{индекс 4 – это номер строки матрицы, 3 – номер столбца матрицы}
Matrix[4, 3] := 123.456;
Readln(Matrix[2, 4]); {Ввод элемента (2, 4) с клавиатуры}
Writeln(Matrix[I, J]); {Вывод элемента (I, J) на экран}
```

В качестве индекса элемента массива может выступать явно указанное целое значение, либо целочисленная переменная. Чаще всего обработка элементов массива выполняется в цикле, поэтому в качестве индекса используется переменная-счетчик цикла, например:

```
for I := 1 to 10 do {цикл по I от 1 до 10}
  Vect[I] := Random; {I-му элементу присваивается случайное число}
```

В приведенном ниже примере осуществляется заполнение элементов матрицы случайными числами с помощью функции Random, вывод полученных значений на экран, также осуществляется поиск наибольшего значения и вывод его (вместе с координатами) на экран.

```
program FindMaxElement;
const {Объявление констант}
  Rows = 5; {Число строк матрицы}
  Cols = 4; {Число столбцов матрицы}
var
  {Объявление массива размером Rows x Cols}
  Mas: array[1..Rows, 1..Cols] of Real;
  I, J: Integer; {Объявление переменных цикла}
  MaxValue: Real; {Максимальное значение}
  MaxRowIndex: Integer; {Номер строки максимального значения}
  MaxColIndex: Integer; {Номер столбца максимального значения}
begin
  {Цикл заполнения массива и вывода элементов на экран}
  for I := 1 to Rows do {Цикл по I (перебор строк матрицы)}
  begin {можно было обойтись и без этого begin..end }
    for J := 1 to Cols do {Цикл по J (перебор столбцов матрицы)}
    begin {а этот begin..end требуется обязательно}
      {Присваиваем элементу (I, J) случайное значение}
```

```
      Mas[I, J] := Random;
      {Печатаем значение элемента (все - на одной строке)}
      Write(Mas[I, J]:8:2, ' ');

      if J = Cols then {Если это последний столбец, }
        Writeln; {то осуществляем перевод строки}
    end;
  end; {Данный begin..end здесь - для улучшения читабельности кода}

  {Перед началом поиска максимального значения переменная
   MaxValue уже должна иметь любое значение, но это значение
   не должно быть больше, чем элементы матрицы. Поэтому
   достаточно взять в качестве первоначального значения
   любой элемент матрицы Mas, например самый первый}
   MaxValue := Mas[1, 1]; {Запоминаем первый элемент}
   MaxRowIndex := 1; {Запоминаем номер строки 1-го элемента}
   MaxColIndex := 1; {Запоминаем номер столбца 1-го элемента}

  {Цикл поиска максимального элемента}
  for I := 1 to Rows do {Цикл по I (перебор строк матрицы)}
    for J := 1 to Cols do {Цикл по J (перебор столбцов матрицы)}
    begin
      {Если текущий элемент (I, J) больше, чем MaxValue,
       то записываем этот элемент в MaxValue и сохраняем его координаты}
      if Mas[I, J] > MaxValue then
      begin
        MaxValue := Mas[I, J];
        MaxRowIndex := I;
        MaxColIndex := J;
      end;
    end;

  {Выводим найденные значения на экран}
  Writeln('Max value=', MaxValue:8:2, '; Row=',
    MaxRowIndex, '; Col=', MaxColIndex);
  Readln;
end.
```

Данный пример весьма прост и снабжен подробными комментариями. Очень важно детально разобраться с данным примером, поскольку от этого зависит успешность выполнения задания из варианта (п. 3.8).

3.7. Использование меню для объединения подзадач a и б.

```
program Laba3;
label
  M1; {Объявление метки безусловного перехода M1}
var
  SubTask: Char; {Переменная типа Char (символ)}
  {**** Здесь разместить дополнительные необходимые переменные ****}
begin
  Writeln('Выберите одну из подзадач:');
  Writeln('a – вычисление заданной величины в цикле');
  Writeln('b – обработка элементов матрицы');
  Writeln('e – выход из программы');

  M1:
```

```
  Write('Введите символ:');
  Readln(SubTask); {Ожидаем ввод любого символа}
  case SubTask of
   'a', 'A':
     begin {Если ввели букву "а" или "А", то выполнится эта ветка}
       Writeln('a) – вычисление заданной величины в цикле');
       {******* Здесь разместить операторы, необходимые
        для выполнения подзадачи "а" *******}
     end;
   'b', 'B':
     begin {Если ввели букву "b" или "B", то выполнится эта ветка}
       Writeln('b) - обработка элементов матрицы ');
       {******* Здесь разместить операторы, необходимые
        для выполнения подзадачи "б" *******}
     end;
   'e', 'E':
     begin {Если ввели английскую "e" или "E", то закрываем программу}
       Exit;
     end; {begin..end здесь указывать не обязательно}
  else {Для этого ELSE (от CASE) не нужен дополнительный begin..end }
   {Если ввели любой другой символ, то выполнится эта ветка}
   Writeln('Введен недопустимый символ!');
   Writeln('Вы должны повторить ввод!');
   goto M1; {Переход на метку M1}
  end;

  Readln; {Ожидание нажатия Enter}
 end.
```

Данный пример не должен вызывать вопросов, поскольку сочетает в себе элементы кода, которые были в достаточной степени разобраны в предыдущих лабораторных работах. Следует отметить, что **Char** – порядковый тип данных, обеспечивающий представление 256 символов, в том числе цифр, знаков препинания, всех больших и малых букв латинского алфавита, а также одного дополнительного алфавита (например, кириллицы).

3.8. Варианты заданий

1.а) Вычислить $\sum_{i=1}^{n} \frac{i}{\sqrt{x^2-1}}$

б) Определить количество элементов матрицы, значения которых не превышают заданное число.

2.а) Вычислить $\sum_{i=1}^{n} ix$.

б) Найти минимальный элемент матрицы.

3.а) Вычислить $\sum_{i=1}^{n} \frac{i}{x^i}$.

б) Найти максимальный элемент матрицы.

4.а) Вычислить $\sum_{i=1}^{n} i^2 - x^2$.

б) Поменять местами минимальный и максимальный элемент матрицы.

5.а) Вычислить $\sum_{i=1}^{n} \frac{x+i}{i}$.

б) Поменять местами два заданных элемента матрицы.

6.а) Вычислить $\sum_{i=0}^{n}\frac{x+i}{x^2}$.

б) Поменять местами две заданных строки матрицы.

7.а) Вычислить $\sum_{i=1}^{n}\frac{x-i}{i^2}$.

б) Транспонировать матрицу А.

8.а) Вычислить $\prod_{i=1}^{n}(i+x)$.

б) Сформировать массив В, содержащий сумму элементов каждого столбца матрицы А.

9.а) Вычислить $\prod_{i=1}^{n}\frac{1}{x^2}+i$.

б) Поменять местами два заданных столбца матрицы.

10.а) Вычислить $\prod_{i=1}^{n}x^i+i$.

б) Определить количество отрицательных элементов матрицы.

11.а) Вычислить $\prod_{i=1}^{n}\frac{(x+i)}{i^2}$

б) Определить количество положительных элементов матрицы.

12.а) Вычислить $\sum_{i=1}^{n}(x-i)^2$.

б) Вычислить среднее арифметическое элементов матрицы.

13.а) Вычислить $\sum_{i=0}^{n}(i^2-x^2)$.

б) Вычислить произведение отрицательных элементов матрицы.

14.а) Вычислить $\sum_{i=1}^{n}(x+i)^2$.

б) Вычислить сумму положительных элементов матрицы.

15.а) Вычислить $\prod_{i=1}^{n}\frac{(x-i)^2}{x}$.

б) Все отрицательные элементы матрицы возвести в квадрат.

16.а) Вычислить $\sum_{i=1}^{n}\frac{i}{\sqrt{x^2+1}}$

б) Сформировать массив В, содержащий максимальные элементы строк матрицы А.

17.а) Вычислить $\sum_{i=1}^{n}x^i$.

б) Сформировать массив В, содержащий суммы элементов строк
матрицы А.

18.а) Вычислить сумму n членов геометрической прогрессии.

б) Вычислить сумму всех отрицательных элементов матрицы.

19.а) Вычислить $\sum_{i=1}^{n}\frac{x}{2^i}$.

б) Заполнить матрицу А случайными числами с помощью функции Random.

20.а) Вычислить $\prod_{i=1}^{n}\frac{x}{2^i}$.

б) Вычислить сумму элементов главной диагонали матрицы.

3.9. Содержание отчета (см. п. 1.11)

3.10. Контрольные вопросы

1) Каково назначение оператора цикла FOR?

2) Каковы правила записи оператора цикла FOR?

3) Каковы алгоритмы работы оператора цикла FOR?

4) Какие циклы называются вложенными?

5) Какие ограничения наложены на оператор FOR?

6) Как определяются данные типа «массив»? Запишите примеры определения данных типа массив с использованием разделов **type** и **var** (или только **var**).

7) Какой тип могут иметь имеет переменные, которые используются в качестве индексов массива?

8) Как получить доступ к элементам одно-, дву-, n-мерного массива?

9) Как можно организовать ввод (вывод) элементов одно-, дву, n-мерного массива?

Лабораторная работа № 4.
Разработка циклической программы с неизвестным количеством повторений

4.1. Цель работы

Целью работы является освоение процесса разработки циклических программ с использованием условных циклов (с заранее неизвестным числом повторений) на языке Pascal.

4.2. Задание на лабораторную работу

Требуется разработать программу с использованием операторов повтора (циклических операторов) WHILE и REPEAT.

4.3. Требования к программе

Программа должна выводить:

– номер варианта, назначение программы и ФИО автора;

– информационные сообщения о необходимости ввода данных;

– сообщение с результатами, полученными в ходе работы программы;

Программа должна проверять допустимые значения аргумента при вычислениях, например, для избежания деления на ноль.

4.4. Порядок выполнения работы

1. Получить вариант задания (п. 4.8).
2. Изучить правила использования операторов WHILE и REPEAT для разработки циклических программ (п. 4.6 и 4.7).
3. Определить область сходимости.
4. Составить и отладить программу вычисления суммы ряда с заданной точностью с использованием циклического оператора WHILE.
5. Составить и отладить программу вычисления суммы ряда с заданной точностью с использованием циклического оператора REPEAT.
6. Проследить с помощью средств отладки системы Turbo Pascal (п. 2.5) изменения значений переменных и результатов проверки условий продолжения (окончания) цикла.
7. Ответить на контрольные вопросы (п. 4.9).
8. Оформить отчет (см. п. 1.11).

4.5. Оператор цикла REPEAT

Оператор REPEAT организует выполнение цикла с заранее неизвестным числом повторений. Тело цикла выполняется хотя бы один раз. Работа цикла прекращается, как только будет достигнуто условие выхода из цикла. Структура оператора REPEAT:

```
repeat
  <Оператор 1>;
  <Оператор 2>;
  ...
  <Оператор N>;
until <Условие выхода из цикла>;
```

Поскольку ключевые слова REPEAT и UNTIL играю роль операторных скобок, то выполняемые между ними операторы не требуется размещать в дополнительном блоке BEGIN..END.

Алгоритм работы оператора REPEAT:

1) выполняются операторы, расположенные в теле цикла;

2) проверяется условие выхода из цикла: если результат логического выражения равен **False** (т.е. условие выхода из цикла еще не достигнуто) то тело цикло будет выполнено еще раз; если результат равен **True** (достигнуто условие выхода из цикла), то происходит выход из цикла, т.е. переход на следующий оператор.

Следует отметить, что в теле цикла должны тем или иным образом корректироваться значения переменных, участвующих в логическом условии выхода из цикла. В противном случае программа может «зациклиться» (зависнуть), т.е. выход из цикла не произойдет никогда и придется аварийно прервать работу программы.

На рисунке 4.1 приведена схема работы цикла REPEAT..UNTIL:

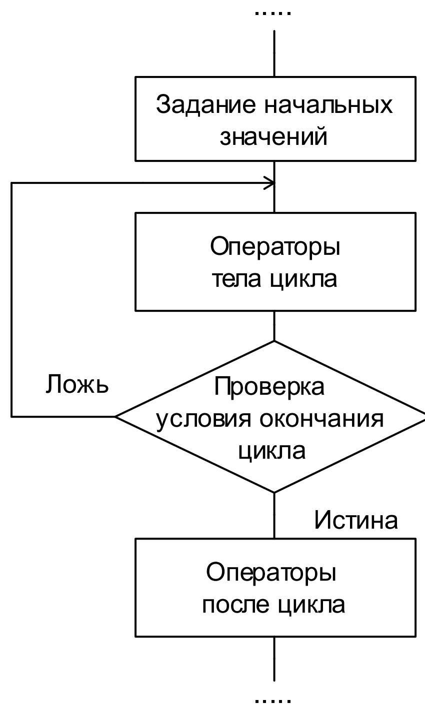


Рисунок 4.1 – Схема работы цикла **repeat**..**until**

В представленном ниже примере программа требует от пользователя ввести число, не равное нулю, т.е. осуществляет контроль правильности ввода данных.

```
program ControlInput;
var
  X: Integer;
begin
  repeat {Начало цикла REPEAT}
    Write('Введите число X: ');
    Readln(X);
    if X = 0 then
      Writeln('Ошибка! Вы ввели 0! Повторите ввод!');
  until X <> 0; {Цикл закончится, если X не равен 0}
  {Дальнейшие операторы...}
end.
```

В отличие от предыдущих примеров, в этом нет оператора GOTO.

Ниже представлен простой, но в то же время интересный пример использования оператора REPEAT. Вначале переменной Y присваивается некоторое значение, а затем в цикле переменная Y делится пополам и полученный результат записывается опять в Y. Цикл будет прерван после того, как в переменную Y будет записано значение «0».

```
program RepeatDivide;
var
  Y: Real; {Дробная переменная}
  Counter: Integer; {Счетчик}
begin
  Counter := 0; {Обнуляем счетчик вначале программы}
  Y := 10; {Присваиваем переменной Y начальное значение}
  repeat  {Начало цикла REPEAT}
    Y := Y / 2; {Делим Y на 2 и присваиваем результат в Y}
    Inc(Counter); {Увеличиваем счетчик на «1» (Counter := Counter + 1)}
  until Y = 0; {Выход из цикла, если Y = 0}

  Writeln('Y=', Y:16:14);
  Writeln('Counter=', Counter);
  Readln;
```

end.

На первый взгляд может показаться абсурдом то, что в качестве условия выхода из цикла используется выражение (Y = 0), ведь вначале программы было задано значение Y > 0, а деление Y на 2 никогда не даст «0». Но не следует забывать, что мы имеем дело с компьютерной программой, каждое число в которой имеет ограниченный диапазон значений, а дробные числа имеют конечную точность. В действительности условие (Y = 0) наступит уже через 132 итерации.

Ниже приведен пример вычисления суммы членов сходящегося числового ряда $\sum_{n=0}^{\infty}\left(-\frac{7}{9}\right)^n$ с заданной точностью α=0,0001. Смысл задачи заключается в том, что при увеличении целочисленного значения *N* происходит уменьшение значения функции $\left(-\frac{7}{9}\right)^n$. На рисунке 4.2 приведен соответствующий график зависимости:

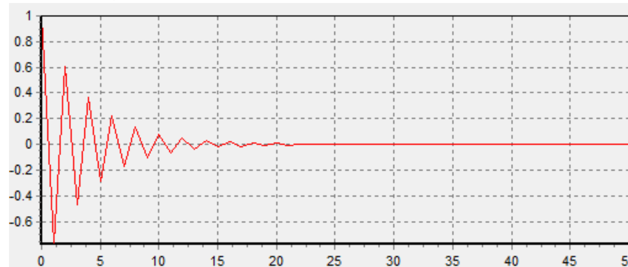


Рисунок 4.2 – График зависимости выражения от числа n

На этом графике по оси абсцисс отложены значения числа *N*. Как видно, функция становится ≈ 0 уже при *N* = 22. Очевидно, что при дальнейшем увеличении *N* значение функции изменяться практически не будет. Согласно условию задачи, необходимо запрограммировать цикл суммирования членов ряда для $0 \le N < \infty$. При этом цикл должен прерываться, как только результат функции $\left(-\frac{7}{9}\right)^n$ окажется ≤ α. Ниже представлен листинг программы, соответствующий этой задаче:

```
program NumSeriesDemo;
const
  Alpha = 0.0001;
var
  N, I: Integer;
  Sum, F: Real;
begin
  Sum := 0; {Начальное значение суммы числового ряда}
  N := 0;   {Начальное значение числа N}
  repeat    {Начало цикла REPEAT}
    { Вычисляем функцию (-7/9)^n }
    F := 1;
    for I := 1 to N do {Цикл возведения в степень}
      F := F * (-7/9);

    Sum := Sum + F; {Выполняем суммирование}
    N := N + 1;     {Увеличиваем N на единицу}
  until Abs(F) <= Alpha; {Условие выхода из цикла}

  Writeln('Sum: ', Sum:15:8); {Выводим на экран сумму (0.5624)}
  Writeln('N: ', N); {Выводим на экран количество итераций (38)}
  Writeln('F: ', F:15:8); {Выводим на экран последнее значение F}
  Readln; {Ожидаем нажатие ENTER}
end.
```

На экран программа выведет значение N=38. Это означает, что числовой ряд сходится с заданной точностью уже на 38-й итерации.

Следует отметить, что в Turbo Pascal отсутствует встроенная функция возведения в степень, поэтому в данном примере для этой цели был реализован дополнительный цикл FOR. Функция **Abs** отбрасывает знак «минус», т.е. возвращает модуль числа, его абсолютное значение.

4.6. Оператор цикла WHILE

Оператор цикла WHILE, аналогично циклу REPEAT, организует выполнение операторов тела цикла неизвестное заранее число раз. В начале каждой итерации осуществляется проверка заданного логического выражения (условия выполнения цикла), и если оно выполняется (равно **True**), то осуществляется переход к операторам тела цикла, в противном случае (**False**) работа цикла прекращается и осуществляется переход к операторам, расположенным после цикла. Структура оператора WHILE:

```
while <Условие выполнения цикла> do
begin
  <Оператор 1>;
  <Оператор 2>;
  ...
  <Оператор N>;
end;
```

Следует отметить, что при наличии в теле цикла WHILE нескольких операторов, все они должны находиться внутри операторных скобок BEGIN..END.

Схема работы цикла WHILE приведена на рисунке 4.3:

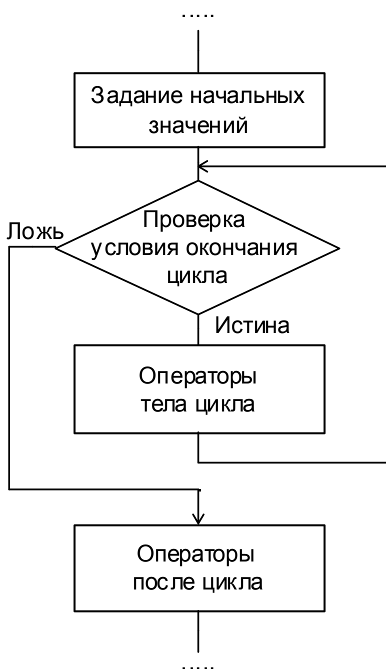


Рисунок 4.3 – Схема работы цикла **while**

Цикл WHILE удобно применять в качестве замены цикла FOR в тех случаях, когда переменная (счетчик) цикла должна быть дробной (Real), либо требуется осуществлять модификацию переменной в теле цикла, либо значение переменной цикла должно изменяться с некоторым заданным шагом, что позволяет обойти ограничение цикла FOR, в котором переменная цикла всегда изменяется с шагом «1». В приведенном ниже примере вычисляется сумма всех чисел от 1 до N, с шагом K:

```
program WhileDemo;
var
  N, K, I, Sum: Integer;
begin
  Write('Введите N, K: ');
  Readln(N, K);
```

```
  Sum := 0; {Начальное значение суммы}
  I := 1;   {Начальное значение счетчика цикла}
  while I <= N do {Условие выполнения тела цикла}
  begin
    Sum := Sum + I; {Вычисляем сумму}
    I := I + K; {Изменяем счетчик цикла с заданным шагом}
  end;
  Writeln('Сумма: ', Sum);
end.
```

Очень часто программисты используют WHILE для организации «бесконечных» циклов: в качестве условия выполнения цикла явно указывают «**True**», а прерывание цикла реализуют с помощью **Break** с использованием одного либо нескольких дополнительных условий:

```
while True do
begin
  Readln(N);
  if N = 0 then Break;
  {Прочие операторы цикла}
end;
```

Следующий пример демонстрирует вывод на экран последовательности чисел Фибоначчи, а также результат деления каждой пары полученных чисел. Последовательность этих чисел следующая: 1, 2, 3, 5, 8, 13, 21, 34, и т.д. Каждое очередное число получается как сумма двух предыдущих. При делении двух соседних чисел получается значение, которое стремится к 1,618 (34/21=1,619) или 0,618 (21/34=0,6176). Значение 1,618 называют пропорцией «золотого сечения», это «оптимальное» соотношение сторон, признак гармонии в природе. Ниже приведен текст программы:

```
program GoldenRatio;
var
  N: Integer;
  NewF, OldF: Integer; {Текущее и предыдущее числа}
  Tmp: Integer; {Временная переменная}
begin
  Write('Введите N: ');
  Readln(N);
  OldF := 1; {Предыдущее число = 1}
  NewF := 2; {Текущее число = 2}
  while NewF <= N do
  begin
    {Выводим текущее число Фибоначчи и "Золотое сечение"}
    Writeln('F=', NewF, '; Golden ratio=', NewF/OldF:8:5);
    Tmp := OldF; {Запоминаем предыдущее число}
    OldF := NewF; {Запоминаем текущее число}
    NewF := NewF + Tmp; {Вычисляем очередное число}
  end;
  Readln;
end.
```

4.7. Варианты заданий

№ варианта	Общий член ряда	Точность α
1	$\sum_{n=1}^{\infty}(-1)^{n+1}\frac{1}{3n^2}$	α =0,01
2	$\sum_{n=1}^{\infty}(-1)^{n+1}\frac{1}{(2n)^3}$	α =0,001

3	$\sum_{n=1}^{\infty}(-1)^n \frac{2n+1}{n^3(n+1)}$	α =0,01
4	$\sum_{n=1}^{\infty}\frac{(-1)^n \cdot n}{2^n}$	α =0,1
5	$\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n!}$	α =0,01
6	$\sum_{n=1}^{\infty}(-1)^n \frac{1}{n!(2n+1)}$	α =0,001
7	$\sum_{n=1}^{\infty}\frac{(-1)^n}{(2n+1)!}$	α =0,0001
8	$\sum_{n=1}^{\infty}\frac{(-1)^n \cdot n^3}{3^n}$	α =0,1
9	$\sum_{n=1}^{\infty}\frac{(-1)^n}{3n!}$	α =0,01
10	$\sum_{n=1}^{\infty}\frac{(-1)^n}{(2n)!}$	α =0,001
11	$\sum_{n=0}^{\infty}\left(-\frac{2}{5}\right)^n$	α =0,01
12	$\sum_{n=0}^{\infty}\left(-\frac{2}{3}\right)^n$	α =0,1
13	$\sum_{n=1}^{\infty}\frac{(-1)^n \cdot n}{7^n}$	α =0,0001
14	$\sum_{n=1}^{\infty}\frac{(-1)^n}{3^n \cdot n!}$	α =0,001
15	$\sum_{n=0}^{\infty}\frac{(-1)^n}{(n+1)^n}$	α =0,001
16	$\sum_{n=1}^{\infty}\frac{(-1)^n}{2^n \cdot n!}$	α =0,001
17	$\sum_{n=0}^{\infty}\frac{(-1)^n}{4^n(2n+1)}$	α =0,001
18	$\sum_{n=0}^{\infty}\frac{(-1)^n \cdot 2^n}{(n+1)^n}$	α =0,001
19	$\sum_{n=1}^{\infty}\frac{(-1)^n}{n^2(n+3)}$	α =0,01

20	$\sum_{n=0}^{\infty} \frac{(-1)^n}{1+n^3}$	α =0,01

4.8. Содержание отчета (см. п. 1.11)

4.9. Контрольные вопросы

1. Каково назначение условных операторов повтора (циклов)?

2. Какие требования предъявляются к выражениям, управляющим повторениями?

3. В чем отличия операторов повтора WHILE и REPEAT?

4. В каких случаях предпочтительнее использовать для организации циклов оператор повтора FOR? Что записывается в заголовке этого оператора?

5. Какие правила пунктуации необходимо соблюдать при записи операторов?

6. Что такое вложенные циклы? Какие дополнительные условия необходимо соблюдать при организации вложенных циклов?

Лабораторная работа №5.
Разработка программы с использованием процедур и функций

5.1. Цель работы

Приобретение навыков разработки программ с использованием процедур и функций пользователя на языке Pascal.

5.2. Задание на лабораторную работу

Разработать программу для вычисления значений функции в соответствии с вариантом задания и вывести результаты в табличном виде.

5.3. Требования к программе

Программа должна выводить:

– номер варианта, назначение программы и ФИО автора;

– информационные сообщения о необходимости ввода данных;

– результаты работы программы в виде таблицы:

Значение функции в интервале [5..9] с шагом 0.1	
Аргумент	Функция
X=<значение аргумента>	Y=<значение функции от X>
...	...

Программа должна состоять из основной части и двух подпрограмм:

1) процедура ввода данных (например, «Vvod»), требующая от пользователя ввести значения для формулы согласно варианту задания, например, A, B, C; программа не должна требовать от пользователя ввода значения X;

2) функция вычисления формулы (например, «CalcFormula») (п. 5.6).

Логика основной части программы должна быть построена из следующих элементов:

1) оператор вызова процедуры ввода данных, в объявлении которой перечислены параметры-переменные; данная процедура должна потребовать от пользователя ввода констант (например, A, B, C), за исключением X;

2) оператор цикла (по выбору: WHILE, REPEAT или FOR), в каждой итерации которого:

– наращивается значение переменной X (например, от 5 до 9 с шагом 0.1);

– осуществляется вызов функции вычисления формулы (например, «CalcFormula») с указанием аргумента X, а также остальных констант, при необходимости; результат вычисления формулы записывается в переменную Y;

– обе переменные X и Y выводятся на экран с помощью Writeln.

5.4. Порядок выполнения работы

1) Получить вариант задания (п. 5.6). Варианты задания содержат данные о функции, начальном и конечном значении аргумента и шаге его приращения.

2) Изучить структуру Pascal-программы, содержащей процедуры и функции пользователя (п. 5.5).

3) Разработать программу в соответствие с заданием.

4) Выполнить отладку программы с заходом в каждую из подпрограмм (см. п. 2.5). Это позволит значительно укрепить знания как по процедурам и функциям, так и по отладке приложений.

5) Ответить на контрольные вопросы (п. 5.8).

6) Оформить отчет (см. п. 1.11).

5.5. Программирование процедур и функций на языке Pascal

В программировании очень часто применяют способ структурирования программного кода, основанный на использовании подпрограмм. Подпрограмма – это оформленный особым образом именованный участок кода, который можно запустить на выполнение из любого места программы. Таким образом, отпадает необходимость многократного дублирования одних и тех же действий в разных частях программы. Достаточно проанализировать то, как часто в программе выполняются те или иные действия, и наиболее часто используемые оформить в виде подпрограммы. В программе допускается любое количество подпрограмм.

Рассмотрим пример решения задачи «**вычислить сумму $x^a + y^b + z^c$, где x, y, x – любые числа; a, b, c – целые числа ≥ 0**» без использования подпрограмм:

```
program PowerBadDemo;
```

```
var
  X, Y, Z: Real; {Дробные числа (будут возводиться в степень)}
  A, B, C: Integer; {Степени (целые числа)}
  Xa, Yb, Zc: Real; {Результаты возведения в степень}
  I: Integer; {Счетчик цикла возведения в степень}
begin
  Writeln('Программа вычисления суммы x^a + y^b + z^c');
  Write('Введите любые числа X, Y, Z: ');
  Readln(X, Y, Z);
  Write('Введите целые неотрицательные степени A, B, C: ');
  Readln(A, B, C);

  { Цикл возведения числа X в степень A }
  Xa := 1;
  for I := 1 to A do
    Xa := Xa * X;

  { Цикл возведения числа  Y в степень B }
  Yb := 1;
  for I := 1 to B do
    Yb := Yb * Y;

  { Цикл возведения числа Z в степень C }
  Zc := 1;
  for I := 1 to C do {Цикл возведения в степень C}
    Zc := Zc * Z;

  Writeln(Xa + Yb + Zc:8:2);
  Readln;
end.
```

В данной программе для каждой операции возведения в степень используется отдельный цикл FOR. При этом пришлось объявить дополнительные переменные: I (счетчик чикла), а также Xa, Yb, Zc (промежуточные результаты возведения в цикл). Несложно представить, во что превратится программа вычисления суммы $u^a + v^b + w^c + x^d + y^e + z^f$. Она будет в два раза больше. При этом увеличивается вероятность ошибки, усложняется отладка программы. Представим, что будет, если изменится условие задачи и потребуется обработка не только положительных, но и отрицательных значений степени: придется вносить значительные исправления в каждом случае возведения в степень, при этом вероятность ошибки увеличивается многократно.

Та же самая задача может быть элегантно решена с применением подпрограммы:

```
program PowerGoodDemo; {Демонстрация более удачной программы}

{Power - это ФУНКЦИЯ возведения числа Value в степень Stepen }
function Power(Value: Real; Stepen: Integer): Real;
var          {Объявление локальных переменных для работы функции}
  I: Integer;     {I - это ЛОКАЛЬНАЯ переменная}
  TmpValue: Real; {TmpValue - временная переменная (тоже локальная)}
begin {Начало тела функции}
  TmpValue := 1; {Инициализация временной переменной}
  for I := 1 to Stepen do {Цикл возведения в степень Stepen}
    TmpValue := TmpValue * Value;
  {ИМЕНИ функции присваиваем результат работы функции}
  Power := TmpValue;
end; {Конец тела функции. Она вернет результат, записанный  в Power}
```

```
{*** НАЧАЛО ОСНОВНОЙ ЧАСТИ ПРОГРАММЫ ***}
var             {Объявление переменных основной части программы}
  X, Y, Z: Real;    {Дробные числа (будут возводиться в степень)}
  A, B, C: Integer; {Степени (целые числа)}
begin           {Здесь программа начинает свою работу}
  Writeln('Программа вычисления суммы x^a + y^b + z^c');
  Write('Введите любые числа X, Y, Z: ');
  Readln(X, Y, Z);
  Write('Введите целые неотрицательные степени A, B, C: ');
  Readln(A, B, C);
  {Вычисления и вывод на экран}
  Writeln(Power(X, A) + Power(Y, B) + Power(Z, C):8:2);
  Readln;
end. {Конец программы}
```

В этом примере все действия, необходимые для возведения в степень, объединены в одну подпрограмму с именем «Power». Подпрограмма, так же как и любая простая программа на языке Pascal, имеет область объявления переменных «**var**» (кроме этого, допустимы разделы «**type**», «**const**», «**label**»), а также тело подпрограммы, содержащее необходимые операторы, расположенные внутри BEGIN..END. Любая подпрограмма должна иметь наименование, в данном случае **Power**. Подпрограмма в этом примере объявлена следующим образом:

```
function Power(Value: Real; Stepen: Integer): Real;
```

Здесь указаны: тип подпрограммы («**function**», т.е. является функцией, значит **должна возвращать некоторое значение-результат**), параметры вызова подпрограммы (Value, Stepen) с указанием их типа, а также тип результата, возвращаемого функцией (Real).

Вызов функции осуществляется путем указания ее имени и передаваемых значений (аргументов) в круглых скобках. Пример вызова функции Power:

Res := Power(5, 2);

При этом будет вызвана функция Power; значение «5» будет автоматически записано в параметр Value, значение «2» – в параметр Stepen, после чего в теле функции аргументы Value и Stepen можно использовать как обычные переменные.

Функция должна обязательно возвращать какое-нибудь значение. В данном примере сначала используется временная переменная TmpValue, но перед окончанием работы функции значение данной переменной присваивается имени функции:

Power := TmpValue;

Тем самым, был определен результат работы функции. Если этого не сделать, то функция все равно вернет некоторый результат, но это будет непредсказуемое (случайное) числовое значение.

После вызова функции Power(5, 2), в переменную Res будет записан результат «25».

Поскольку функция возвращает значение, ее вызов может находиться справа от оператора присваивания «:=» (но не слева).

При выполнении оператора

Writeln(Power(X, A) + Power(Y, B) + Power(Z, C));

программа 3 раза вызовет функцию Power с указанными аргументами, просуммирует результаты и выведет полученную сумму на экран.

Анализ приведенного выше примера позволяет утверждать, что применение подпрограммы позволило сделать код более компактным, понятным, простым и удобным в сопровождении. При необходимости внесения изменений достаточно сделать это в одном месте. Разработанную функцию можно вызвать из любого места программы.

5.5.1 Объявление функции

Подпрограмма, возвращающая значение-результат, называется функцией. Функция, состоит из заголовка и тела. Заголовок содержит зарезервированное слово «**function**», идентификатор (имя) функции, необязательный список формальных параметров, заключенный

в круглые скобки и тип возвращаемого функцией значения. Тело функции представляет собой блок операторов, заключенный в BEGIN..END:

```
function <имя>(<список формальных параметров>): <тип результата>;
const ... {объявление констант, используемых в функции}
var ...   {локальные переменные функции}
begin
  <операторы>
end;
```

где **<имя>** – любой допустимый идентификатор, например, Func1;

<список формальных параметров> – список имен переменных и их типов, разделенных точкой с запятой «;», например (Value: Real; Stepen: Integer);

Важно! Имена переменных, перечисленные в списке формальных параметров, **не обязаны** совпадать с именами переменных, указанных при вызове подпрограммы. Подпрограмма может вызываться из нескольких различных мест программы, причем в каждом случае переменные могут называться по-разному. В рассмотренном примере для функции «Power» были объявлены параметры «Value» и «Stepen», однако при ее вызове использовались разные переменные: X, Y, Z, A, B, C. Более того, при вызове функции можно указывать непосредственные значения, например, **Power(5, 2)**. В любом случае передаваемые в функцию значения будут скопированы в соответствующие формальные параметры, с которыми будет осуществляться дальнейшая работа (исключением является использование формальных параметров-переменных).

<тип результата> – тип возвращаемого функцией результата, например, Integer, Real и т.д.

Среди входящих в функцию операторов должен обязательно присутствовать **как минимум** один оператор присваивания «:=», в левой части которого указано имя данной функции, а в правой – значение-результат. В точку вызова возвращается результат последнего присваивания.

5.5.2 Объявление процедуры

Подпрограмма, которая **не возвращает** значение-результат, называется процедурой (в терминологии языка Pascal). Процедура, состоит из заголовка и тела. Заголовок содержит зарезервированное слово «**procedure**», идентификатор (имя) процедуры и необязательный список формальных параметров, заключенный в круглые скобки. Тело процедуры представляет собой блок операторов, заключенный в BEGIN..END:

```
procedure <имя>(<список формальных параметров>);
const ... {объявление констант, используемых в процедуры}
var ...   {локальные переменные процедуры}
begin
  <операторы>
end;
```

Следующий пример демонстрирует использование процедуры для ввода двух значений A и B, где A > 0 (Real), B <> 0 (Integer). В процедуре «InputAB» организован контроль правильности вводимых данных. Для выхода из программы требуется, чтобы произведение двух чисел было равно 100.

```
program ProcDemo;
{ InputAB – процедура для ввода двух чисел. ValueA и ValueB – параметры-
  переменные, поэтому в момент вызова процедуры InputAB в нее будут
  переданы непосредственно сами переменные A и B, а не их копия }
procedure InputAB(var ValueA: Real; var ValueB: Integer);
begin
  repeat
    Write('Введите значение A (любое положительное число): ');
```

```
    Readln(ValueA);
    if ValueA <= 0 then
      Writeln('Ошибка! Число должно быть > 0. Повторите ввод!');
  until ValueA > 0;

  repeat
    Write('Введите значение B (целое ненулевое число): ');
    Readln(ValueB);
    if ValueB = 0 then
      Writeln('Ошибка! Число не может быть = 0. Повторите ввод!');
  until ValueB <> 0;
end;

var
  A: Real;
  B: Integer;
begin
  repeat {Начало цикла REPEAT}
    InputAB(A, B); {Вызов процедуры InputAB}
    Writeln('Произведение A * B = ', A * B:8:2);
  until A * B = 100; {Условие завершения цикла REPEAT}
end.
```

Особенностью данного примера является то, что в процедуре «InputAB» формальные параметры «ValueA» и «ValueB» являются параметрами-переменными (используется ключевое слово «**var**»), поэтому в момент вызова процедуры «InputAB» в нее будут переданы непосредственно сами переменные «A» и «B», а не их копия. Любое изменение формальных параметров «ValueA» и «ValueB» внутри процедуры приведет к тому, что будут изменены переменные «A» и «B».

Следующий простой пример демонстрирует вывод на экран информации о разработчике программы с помощью процедуры About:

```
program ProcDemo2;

procedure About; {Процедура объявлена без параметров}
begin
  Writeln('Программа для демонстрации процедуры без параметров');
  Writeln('Автор программы: Иванов И.И.');
  Writeln('Версия программы: 1.0');
end;

begin
  About;
  {Прочие операторы программы}
end.
```

В данном случае процедура About не имеет параметров, поскольку они не требуются. Выполненная в этом примере структуризация программного кода позволила вынести часть кода из основной программы в процедуру, тем самым позволив автору программы сосредоточиться на более важных проблемах.

5.5.3 Передача аргументов в подпрограмму с использованием параметров-значений и параметров-переменных

Переменные, перечисленные в заголовке подпрограммы, называются «формальными параметрами», в языке Pascal они отделяются друг от друга точкой с запятой «;». Формальные параметры получают фактические значения аргументов в момент вызова подпрограммы. Существуют 2 основных способа передачи аргумента в подпрограмму: «по значению» (с

использованием параметров-значений) и «по ссылке» (с использованием параметров-переменных).

В случае использования параметров-значений (способ «передача по значению») происходит копирование передаваемого значения аргумента в соответствующий формальный параметр. Присвоение нового значения параметру в теле подпрограммы не может привести к изменению переменной, указанной при вызове подпрограммы. При вызове подпрограммы допускается указывать константы, как именованные, так и заданные явным образом. Пример объявления процедуры с передачей «по значению»:

```
procedure MyProc(MyParam: Integer); {MyParam - параметр-значение}
begin
  ...
end.
```

В случае использования параметров-переменных (способ «передача по ссылке») в подпрограмму передается непосредственно сама переменная-аргумент, а не ее копия. Присвоение нового значения параметру-переменной в теле подпрограммы приведет к изменению переменной, указанной при вызове подпрограммы. Тип параметров-переменных должен совпадать с типом переменных, которые указываются при вызове подпрограммы.

Ниже представлен пример вызова процедуры, в которой «MyParam» является параметром-переменной (используется ключевое слово «**var**»):

```
procedure MyProc(var MyParam: Integer); {MyParam - параметр-переменная}
begin
  MyParam := MyParam * 2; {изменяем значение параметра MyParam}
end;

var
  A: Integer;
begin
  A := 5;
  Writeln(A); {Будет напечатано значение "5"}
  MyProc(A);  {Вызов процедуры MyProc}
  Writeln(A); {Будет напечатано значение "10"}
end.
```

Следует отметить, что в подпрограмме могут одновременно быть объявлены и параметры-значения и параметры-переменные, например:

```
procedure MyProc(A, B: Integer; var X: Real; C: Real; var Y: Integer);
```

В данном примере A, B, C являются параметрами-значениями, а X, Y являются параметрами-переменными.

5.5.4 Локальные и глобальные переменные и их область действия

Переменные, описанные внутри процедур и функций, называются **локальными**. Они создаются при каждом вызове подпрограммы и уничтожаются при выходе из нее, т.е. локальные переменные существуют только при выполнении подпрограммы и недоступны в основной программе. К локальной переменной в языке Pascal возможен доступ из любого участка подпрограммы, в которой объявлена эта переменная. Если в подпрограмме имеются вложенные подпрограммы (допускается любой уровень вложенности), то все они могут иметь доступ к данной переменной, кроме случаев, когда объявление переменной находится ниже, чем код подпрограммы.

Переменные, объявленные в основной программе, называются **глобальными**. К ним существует доступ не только из основной программы, но и из любой процедуры или функции,

за исключением случаев, когда объявление глобальной переменной находится ниже, чем код подпрограммы.

Количество обращений к глобальным переменным из подпрограмм должно быть минимальным, поскольку это ухудшает читабельность кода, а также усложняет отладку и исправление ошибок.

5.5.5 Предварительное описание подпрограммы

Подпрограмму (допустим, A) можно вызвать не только из основной программы, но и из любой другой подпрограммы (допустим, B) при условии, что объявление подпрограммы A расположено выше, чем объявление подпрограммы B. В действительности данное условие не всегда является выполнимым. Обычно ограничения связаны с обеспечением более наглядной логической структуры программы, например, требуется, чтобы вначале располагались все подпрограммы, ответственные за ввод данных, за ними следовали подпрограммы, обеспечивающие все необходимые вычисления, а подпрограммы вывода результатов на экран располагались после всех предыдущих. Как в таком случае из подпрограммы ввода данных вызвать подпрограмму для проведения вычислений (ведь она расположена ниже)? Для разрешения подобных ситуаций применяется предварительное описание подпрограммы. В языке Pascal для предварительного описания подпрограммы вместо тела подпрограммы указывается ключевое слово «**forward**». Пример:

```
procedure ProcA(S: string); forward; {Предварительное описание }

procedure ProcB(S: string);
begin
  ProcA(S); {Процедура ProcA объявлена выше, поэтому ошибки не будет}
end;

procedure ProcA; {Параметры не обязательно указывать еще раз }
begin
  Writeln(S); {Будет напечатана строка «Hello!»}
end;

begin {Начало основной программы}
  ProcB('Hello!'); {Вызываем процедуру ProcB с аргументом 'Hello!' }
end.
```

5.5.6 Рекурсивный вызов подпрограммы

Вызов некоторой подпрограммы из той же самой подпрограммы называется рекурсивным, а последовательность нескольких таких вызовов – рекурсией. При программировании рекурсии важно предусмотреть критерий, завершающий цепочку рекурсивных вызовов, в противном случае рекурсия может оказаться бесконечной (в реальности программа будет аварийно завершать свою работу с ошибкой «переполнение стека»). На практике рекурсия чаще всего используется для обработки элементов древообразных (ветвящихся) структур, в которых невозможно заранее предугадать глубину ветвления.

5.6. Варианты заданий

№ варианта	Функция	Начальное значение	Шаг	Конечное значение
1	$a \cdot \cos(x) - b + c$	1	0,1	3
2	$\frac{a \cdot \sin(x)}{b + c}$	2	0,2	4
3	$a \cdot \sin(bx) + c$	3	0,5	5
4	$a \cdot \sin(x) \cdot b + c$	4	0,1	6
5	$a \cdot x^2 + b \cdot x + c$	5	0,2	7

№ варианта	Функция	Начальное значение	Шаг	Конечное значение
6	$a \cdot e^{x} + b \cdot x + c$	6	0,5	8
7	$a \cdot \ln(x) + bx^{2} + c$	7	0,1	9
8	$a \cdot \ln(x) + b + c$	8	0,2	10
9	$a \cdot \ln(b \cdot x) + c$	9	0,5	11
10	$a \cdot x \cdot \ln(b \cdot x) + c$	10	0,1	12
11	$a \cdot e^{x} + b \cdot \sin(c \cdot x)$	11	0,2	13
12	$a \cdot e^{-x} - b \cdot \cos(x) + c$	12	0,5	14
13	$a \cdot x^{3} + b \cdot x + c$	13	0,1	15
14	$a \cdot e^{(2x)} + b \cdot x + c$	14	0,2	16
15	$a \cdot \ln(x) + b \cdot e^{x} + c$	15	0,5	17
16	$a \cdot e^{x}$	16	0,1	18
17	$a \cdot x^{2} + e^{x} + c$	17	0,2	19
18	$a \cdot \cos(x) - b$	18	0,5	20
19	$a \cdot \sin(x) + c$	19	0,1	21
20	$a \cdot \ln(bx) + c \cdot e^{x}$	20	0,5	22

5.7. Содержание отчета (см. п. 1.10)

5.8. Контрольные вопросы

1) Что называется подпрограммой?
2) Какова структура программы с подпрограммами?
3) Какова структура подпрограммы-процедуры?
4) Какова структура подпрограммы-функции?
5) В чем состоит различие и сходство процедур и функций?
6) Как осуществляется вызов процедур и функций?
7) Что называется параметром и каково его назначение?
8) Каково назначение формальных и фактических параметров и какова их взаимосвязь?
9) Опишите последовательность событий при вызове процедур или функций.
10) Для чего при отладке используется пошаговый режим с заходом в подпрограммы и как его осуществить?
11) В чем разница между способами передачи аргументов в подпрограмму «по значению» и «по ссылке»?
12) Чем отличаются локальные и глобальные переменные? Какова их область действия?

Лабораторная работа № 6.
Обработка символов и строк на языке Pascal

6.1. Цель работы

Приобретение навыков обработки символьных данных на языке Pascal.

6.2. Задание на лабораторную работу

Составить программу обработки символьных данных в соответствии с вариантом задания (п. 6.7).

6.3. Требования к программе

Программа должна выводить:

– номер варианта, назначение программы и ФИО автора;

– информационные сообщения о необходимости ввода данных;

– результаты работы в соответствии с вариантом задания. Обработка символьных данных должна выполняться в подпрограмме пользователя (п. 6.6).

6.4. Порядок выполнения работы

1) Получить вариант задания (п. 6.7).

2) Изучить операторы объявления и обработки символов и строк (п. 6.5).

3) Разработать и отладить программу обработки символьных данных.

4) Ответить на контрольные вопросы (п. 6.9).

5) Оформить отчет (см. п. 1.10).

6.5. Операторы объявления и обработки символов и строк на языке Pascal

6.5.1. Операторы определения и обработки данных символьного типа

Символьный тип обеспечивает программисту возможность работы с символами, к которым относятся: цифры (от 0 до 9), буквы латинского алфавита (как строчные, так и прописные), буквы национального алфавита (как строчные, так и прописные), знаки препинания и др. В связи с тем, что элемент, объявленный как **Char**, занимает в памяти 8 бит (1 байт), то всего доступно 256 символов. Каждый символ имеет собственный числовой код (от 0 до 255), причем коды от 0 до 127 являются стандартными (соответствуют американскому стандарту ANSI). Коды этих символов одинаковы практически в любых устройствах, компьютерах, операционных системах, языках программирования, Unicode-кодировках и т.д.

Значения от 128 до 255 предназначены для кодирования символов из других национальных алфавитов (в том числе кириллицы). Поскольку этого числового пространства хватает на кодирование алфавита лишь одного языка, то оперировать одновременно несколькими национальными алфавитами не представляется возможным. Более того, существуют языки, содержащие в алфавите свыше 127 символов (например, китайский, в котором более 50 тыс. иероглифов), поэтому использовать стандартные средства языка Pascal для работы с такими алфавитами весьма затруднительно.

Представляемые типом **Char** символы (в том числе кириллица) и их коды приведены на рисунке 6.1.

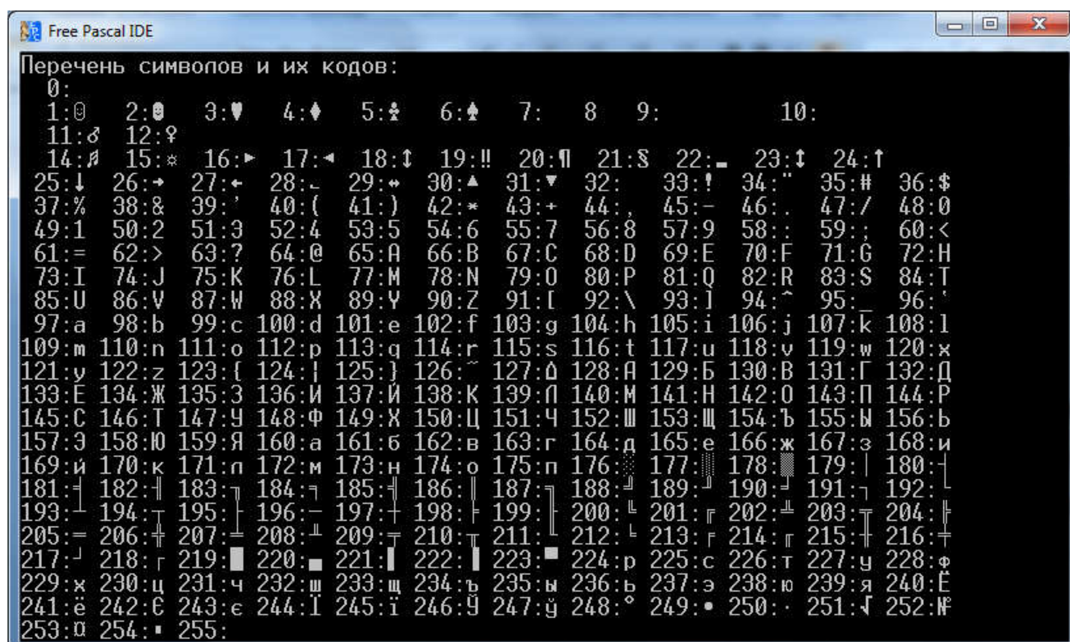


Рисунок 6.1 – Символы и их коды

Особое значение имеют символы с кодами:

– 9 – код символа табуляции. Используется для выравнивания текста. Осуществляет смещение курсора вправо на одну или несколько (но не более 8) позиций в зависимости от текущего расположения курсора. Новое положение курсора будет кратно 8 плюс 1.

– 10 – код перевода строки. В результате следующие символы будут напечатаны на новой строке.

– 13 – код возврата каретки. Возвращает курсор ввода на начало строки. Новые символы будут затирать предыдущие.

– 32 – код пробела.

Ниже представлен текст программы, с помощью которой был подготовлен рисунок 6.1:

```
var
  I: Integer; {Объявление счетчика цикла FOR }
  C: Char;    {Объявление переменной «символ»}
begin
  WriteLn('Перечень символов и их кодов:');
  for I := 0 to 255 do {Цикл перебора всех кодов символов}
  begin
    C := Chr(I); {Получаем символ из его кода}
    Write(I:3, ':', C, ' '); {Печать на экране символа и его кода}
    if I mod 12 = 0 then {Если код символа кратный 12,}
      WriteLn;          {то переходим на следующую строку}
  end;
end.
```

Значение переменной **Char** можно задать также с помощью оператора Readln, либо путем непосредственного присвоения отдельного символа (символьной константы), заключенного в одинарные кавычки (апострофы), например:

```
C := 'F';
```

Поскольку тип **Char** является порядковым, допускается использование по отношению к нему операций сравнения, например:

```
if (C >= 'A') and (C <= 'Z') then
  Writeln('Это символ из латинского алфавита!');
```

Стандартные функции языка Pascal, применимые к типу Char, представлены к таблице 6.1.

Таблица 6.1 – Стандартные функции Pascal для работы с символами

Функция	Назначение	Тип аргумента	Тип функции	Пример
Chr	Получение символа по его коду	Byte	Char	x := 68; y := Chr(x); {y = 'D'} y := Chr(2*x-5); {y='Г'}
Ord	Определение кода символа	Char	Byte	x := 'G'; y := Ord(x); {y=71}
Pred	Возвращение предыдущего символа	Char	Char	x := 'Б'; y := Pred(x); {y='A'}
Succ	Возвращение следующего символа	Char	Char	x := 'Г'; y := Succ(x); {y='Д'}
Upcase	Преобразование строчной буквы в прописную	Char	Char	Ch := 'd'; Y := Upcase(Ch); {Y='D'}

6.5.2. Определение переменных строкового типа STRING. Операторы обработки строк

Последовательность нескольких символов, расположенных друг за другом, образует строку. В общем случае, в строке может присутствовать произвольное количество символов, в том числе, ни одного. В случае языка Pascal, максимальная длина одной строки ограничена 255 символами. В современных языках программирования подобные ограничения, как правило, отсутствуют. Для объявления строки в языке Pascal служит ключевое слово STRING. Примеры объявления строки и присвоения строковых значений:

```
const
  sConstStr = 'Строка 1'; {Объявление именованной константы}
var
  S1: string; {объявление строки максимальной длины 255 символов}
  S2: string[10]; {объявление строки длиной 10 символов}
  S3: string[8]; {объявление строки длиной 8 символов}
begin
  S1 := 'А роза упала на лапу Азора'; {присвоение строковой константы}
  Readln(S2); {Строку должен ввести пользователь}
  S3 := sConstStr; {присвоение именованной строковой константы}
end;
```

Для определения количества символов в строке служит функция Length:

```
Writeln(Length(S1)); {В нашем случае напечатает значение 26}
```

В языке Pascal существует второй способ определения длины строки – обращение к «нулевому» символу, например:

```
StrLen := Ord(S1[0]); {Присвоит в переменную StrLen значение 26}
```

Однако, поскольку второй способ специфичен для языка Pascal и не принят в других языках программирования, рекомендуется первый способ.

С помощью нулевого символа можно также изменить длину строки:

```
S1[0] := Chr(Length(S1) - 1); {Уменьшает длину строки на 1 символ}
```

Поскольку строки состоят из символов, а каждый символ имеет свой числовой код, то по отношению к строкам можно применять операции сравнения, например:

```
if S1 = 'hello' then
  Writeln('Строки одинаковые')
else
  Writeln('Строки разные');
```

Кроме того, допустимы операции <> (не равно), > (больше), >= (больше или равно), < (меньше), <= (меньше или равно). В практическом программировании при сравнении строк используют только операции = или <>.

Операция сравнения ('Hello' > 'hello') вернет False, поскольку код символа 'H' (72) меньше, чем код символа 'h' (104). Если бы эти символы оказались одинаковыми, то программа бы продолжила сравнение оставшихся символов.

Для объединения двух или более строк в одну служит операция сцепления «+», например:

```
S2 := 'Новый';
S3 := 'Год';
S1 := S2 + ' ' + S3; {Соединяем строки S2 и S3 через пробел }
Writeln(S1); {Напечатает строку: Новый Год}
```

Обращение к отдельному символу строки осуществляется по индексу (аналогично работе с массивами), например:

```
St[1] := 'F'; {В первый символ строки будет записано «F»}
Writeln(St[2]); {Печать на экране второго символа строки}
```

Информация о процедурах и функциях языка Pascal, применяемых для обработки строк, приведена в таблице 6.2

Таблица 6.2 – Процедуры и функции обработки строк

Заголовок	Назначение	Пример
procedure Delete(**var** S: **string**; Index: Integer; Count: Integer);	Удаление **Count** символов строки **S**, начиная с позиции **Index**	S := 'абвгде'; Delete(S, 4, 2); *{St='абве'}*
procedure Insert(Source: **string**; **var** S: **string**; Index: Integer);	Вставка подстроки **Source** в строку **S**, начиная с позиции **Index**	St1 := 'абвгде'; Insert('*', St1, 4); *{Str2='абв*где'}*
procedure Str(X: Real; **var** S: **string**);	Преобразует числовое значение величины **X** (Integer или Real) в строку и помещает результат в **S**	V := 1500; Str(V:6, St); *{St:=' 1500'}*
procedure Val(S: **string**; **var** V: Real; **var** Code: Integer);	Преобразует строку **S** в ее числовое представление и помещает результат в **V** (Integer или Real) величину. Параметр **Code** возвращает код ошибки	St := '1.455'; Val(St, V, Cod); *{V=1.455, Cod=0}*
function Copy(S: **string**; Index, Count: Integer): **string**;	Возвращает заданный участок строки **S** длиной **Count** символов, начиная с позиции **Index**. Если значение **Count** слишком большое, то возвращает символы до конца строки.	St := 'абвгде'; Y := Copy(St, 2, 3); *{Y='бвг'}*
function Pos(Substr, S: **string**): Integer;	Отыскивает подстроку **Substr** в строке **S** и возвращает позицию первого символа найденной подстроки, либо ноль, если подстрока не найдена	St2 := 'abcdef'; Y := Pos('de', St2); *{Y=4}*

6.6. Пример программы

Приведенный ниже пример демонстрирует программу для упорядочивания по алфавиту ФИО, заданных в строке S:

```
program SortFIO;

type {Объявлям пользовательские типы TFIOElem и TFIOArray}
  TFIOElem = string[40]; {Элемент массива с Ф.И.О.}
  {Тип - массив из 10 строк, каждая длиной до 40 символов: }
  TFIOArray = array[1..10] of TFIOElem;

{Сортировка строковых элементов массива самым простым способом}
{Используется алгоритм "Сортировка выбором"}
procedure DoSortArray(FioCount: Integer; var FioArray: TFIOArray);
var
  I, J: Integer;
  STmp: TFIOElem;
  MinStrIdx: Integer;
begin

  for I := 1 to FioCount - 1 do {Внешний цикл перебора слева-направо}
```

```
  begin
    MinStrIdx := I; {Пока считаем i-ый элемент минимальным}
    {Отыскиваем строку, которая должна располагаться раньше по алфавиту}
    for J := I + 1 to FioCount do {Внутренний цикл перебора}
      if FioArray[J] < FioArray[MinStrIdx] then
        MinStrIdx := J; {Запоминаем индекс найденной строки}

    {Если такая строка найдена, то меняем ее с i-ым элементом}
    if MinStrIdx <> I then
    begin
      STmp := FioArray[I]; {Запоминаем i-ый элемент}
      {Записываем в него новое значение}
      FioArray[I] := FioArray[MinStrIdx];
      FioArray[MinStrIdx] := STmp;
    end;
  end;

end; {END of DoSortArray}


{Процедура MakeSort осуществляет сортировку строки с Ф.И.О.,
 заданной с помощью аргумента St }
procedure MakeSort(var St: string);
var
  {Объявляем массив из 10 строк, каждая длиной до 40 символов}
  StrAr: TFIOArray;
  I, Len, CurArIdx: Integer;
begin
  {Заменяем все последовательности ", " на ","}
  while Pos(', ', St) > 0 do
    Delete(St, Pos(', ', St) + 1, 1);

  Len := Length(St); {Запоминаем длину строки}
  I := 1; {Инициализируем счетчик цикла WHILE}
  CurArIdx := 1; {Инициализируем счетчик текущего элемента массива}
  StrAr[CurArIdx] := ''; {Предварительная очистка строки}

  {В цикле осуществляем разбивку исходной строки. В результате этого
   каждое Ф.И.О. будет храниться в отдельном элементе массива StrAr,
   а в переменную CurArIdx будет записано общее количество Ф.И.О.}
  while I <= Len do {Пока не достигнут конец строки}
  begin
    if St[I] = ',' then {Если встретили запятую}
    begin
      Inc(CurArIdx); {Делаем текущим следующий элемент массива}
      StrAr[CurArIdx] := ''; {Предварительная очистка строки}
    end
    else {Добавляем символ к текущему строковому элементу массива}
      StrAr[CurArIdx] := StrAr[CurArIdx] + St[I];
    Inc(I); {Увеличиваем счетчик цикла на единицу}
  end; {Конец цикла WHILE}

  {Выводим на экран промежуточные результаты работы программы
   - список всех найденных Ф.И.О., каждое на отдельной строке}
  for I := 1 to CurArIdx do
    Writeln('FIO ', I, ': ', StrAr[I]);
```

```
  {Сортировка элементов строкового массива}
  DoSortArray(CurArIdx, StrAr);

  St := ''; {Очищаем строку St}
  {Записываем в var-параметр St отсортированный список Ф.И.О.}
  for I := 1 to CurArIdx do
  begin
    St := St + StrAr[I]; {Выполняем "сцепление" строк}
    if I < CurArIdx then {Если Ф.И.О. не последнее}
      St := St + ', ';   {то добавляем после него ", "}
  end;
end; {END of MakeSort}


var
  S: string;
begin {Начало основной программы}
  Writeln('Введите несколько Ф.И.О. через запятую:');
  Readln(S);   {Ввод строки}
  MakeSort(S); {Обработка строки (сортировка Ф.И.О.)}
  Writeln('Список Ф.И.О. после сортировки:');
  Writeln(S);  {Вывод результатов на экран}
  Readln;
end.
```

6.7. Варианты заданий

Количество слов в строке и максимальный размер каждого их слов выбираются студентом. Рекомендуется работать с латинскими символами.

1) Поменять местами слова с максимальной и минимальной длиной при выполнения условия, что такие слова единственные

2) Заменить окончания (последние два символа) на 'xz' в словах, длина которых равна 5

3) Поменять местами слово, начинающееся на 'a', со словом, оканчивающимся на 'z', при условии, что такие слова существуют и являются единственными

4) Удалить последние 3 символа из слов, начинающихся на 'a'

5) Удалить первые 3 символа из слов, оканчивающихся на 'th'

6) Дополнить символом '*' слова, имеющие длину меньше заданной (максимальной) до максимальной

7) Заменить первые 3 символа слов, имеющих выбранную длину, на символ '*'

8) Удалить все символы 'a' из слов, длина которых равна выбранной

9) Заменить все символы 'a' на 'd' в словах, длина которых меньше выбранной

10) Заменить первые строчные буквы на заглавные в каждом слове, длина которого больше выбранной

11) Вставить пробел после первых 2-х символов в слова, имеющие длину, на 1 меньше заданной

12) Заменить первую строчную букву на заглавную в словах, имеющих выбранную длину

13) Вставить пробел перед последними 2-мя символами в слова, имеющие минимальную (заданную) длину

14) Посчитать количество гласных букв в строке и заменить их на '*'

15) Упорядочить символы в строке по алфавиту

16) Вывести все слова, у которых первая и последняя буквы одинаковые

17) Упорядочить строку по убыванию длин слов

18) Проверить в математическом выражении, заданном строкой, соответствие открывающих и закрывающих скобок

19) Вывести все слова в строке в обратном порядке

20) Удалить двойные пробелы и переместить все найденные в тексте цифры в конец строки

6.8. Содержание отчета (см. п. 1.10)

6.9. Контрольные вопросы

1) Что такое строка?
2) Каким идентификатором определяются данные строкового типа?
3) Какова максимальная длина строки? Как определить длину строки?
4) Какие выражения называются строковыми?
5) Какие операции допустимы над строковыми данными?
6) Каким образом производится сравнение строк?
7) Как можно обратиться к отдельным символам строки?
8) Каково назначение специальных процедур и функций обработки данных строкового типа?

Лабораторная работа № 7. Работа с файлами на языке Pascal

7.1. Цель работы

Приобретение навыков разработки Pascal-программы для работы с файлами с целью длительного хранения и загрузки необходимой информации.

7.2. Задание на лабораторную работу

Составить программу на языке Pascal, осуществляющую ведение информационного справочника в соответствии с вариантом задания (п. 7.7).

7.3. Требования к программе

Программа должна выводить:

– номер варианта, назначение программы и ФИО автора;
– информационные сообщения о необходимости ввода данных;
– результаты работы в соответствии с вариантом задания.

Программа должна обеспечивать следующие операции (п. 7.6):

1) загрузка справочника из файла;
2) добавление новой записи;
3) вывод на экран списка всех записей;
4) поиск записи по заданному атрибуту;
5) сохранение справочника в файл.

7.4. Порядок выполнения работы

1) Получить вариант задания (п. 7.7).
2) Изучить операторы объявления данных типа «запись» и операторы обработки типизированных и текстовых файлов (п. 7.5).
3) Разработать и отладить программу, осуществляющую ведение информационного справочника.
4) Ответить на контрольные вопросы (п. 7.9).
5) Оформить отчет (см. п. 1.11).

7.5 Основы работы с записями и файлами на языке Pascal

7.5.1. Определение типа данных RECORD

Помимо простых типов данных (Real, Integer, Byte, Char, Boolean и т.п.), а также массивов (array) и строк (string), в языке Pascal присутствует структурированный тип данных «запись», объявляемый с помощью ключевого слова RECORD. В ряде других языков программирования аналогичная языковая конструкция называется «структурой» (STRUCT). Формат объявления типа «запись»:

```
type
  <наименование_типа> = record
    <поле_1>: <тип_поля>;
    <поле_2>: <тип_поля>;
    <поле_N>: <тип_поля>;
  end;
```

Таким образом, запись состоит из произвольного количества полей, каждое их которых может иметь любой известный компилятору тип (в том числе RECORD). Пример объявления типа RECORD и переменных:

```
type {Объявление типа}
  TBirthDay = record
    Fam: string[40];  {фамилия}
    Day, Month: Byte; {день и месяц рождения}
    Year: Word;       {год рождения}
  end;
var {Объявление переменых}
  a, b: TBirthDay;
```

В этом примере тип TBirthDay (день рождения) представляет собой запись с полями Day, Month и Year (день, месяц и год), а также Fam (фамилия). Переменные a, b являются записями типа TBirthDay. Таким образом, в одной переменной объединено (инкапсулировано) сразу несколько разнотипных данных.

Между однотипными переменными-записями допускается операция присваивания, например: a := b.

Для доступа к полю записи (с целью его чтения или модификации) необходимо использовать составное имя, в котором имя поля отделяется от имени переменной символом «.» (точка), например:

```
a.Day   := 1;
a.Month := 12;
a.Year  := 2011;
a.Fam   := 'Ivanov';
```

Следующий пример демонстрирует объявление массива, состоящего из элементов-записей, а также вывод содержимого массива на экран:

```
var
  Mas: array[1..10] of TBirthDay;
....................
for I := 1 to 10 do
  Writeln('Запись №', I, ': Фамилия=', Mas[I].Fam,
  ', Дата рожд=', Mas[I].Day, '.', Mas[I].Month, '.', Mas[I].Year);
```

Как видим, элементы массива, помимо простых типов и строк, могут быть также структурированными.

Приведенный пример является простым и очевидным, однако этого удалось достичь лишь за счет использования структурированных типов. Объединение (инкапсуляция) некоторого набора разнотипных данных в одной структуре – очень распространенный приём в программировании.

7.5.2. Операторы для работы с файлами в Pascal

В независимости от используемого языка программирования (Pascal, C++, Java и т.д.), операционной системы (Windows, Linux, Unix, Mac-OS и т.д.), аппаратной архитектуры (персональный компьютер, сотовый телефон, промышленный контроллер и т.д.), рано или поздно перед программистом возникнет задача длительного хранения данных, необходимых для работы программы. В качестве энергонезависимого носителя, на который может быть сохранена информация для длительного хранения, могут выступать: дискета, кассетная лента, flash, fram, жесткий диск (магнитный или полупроводниковый) и т.д. В зависимости от используемой операционной системы, различаются способы организации информации на тех или иных носителях. Крайний и наиболее сложный случай заключается в том, что программист должен полностью продумывать механизмы хранения информации, вплоть до оперирования с памятью на уровне отдельных ячеек или кластеров. При использовании современных операционных систем (Windows, Linux и т.д.) вся сложность работы с тем или иным устройством прячется от программиста. В замен ему предлагается работать с более понятным и универсальным объектом – «файл». Файл – это именованный участок данных, сохраненных в энергонезависимой памяти (например, на флешке). Совершенно не важно, как физически хранятся данные (везде – по-разному). Программиста это вопрос не должен интересовать.

Операционная система предоставляет программисту готовые функции для работы с файлами, осуществляющие: создание, открытие, закрытие файла, запись в файл, чтение из файла и ряд других функций.

После изучения (в достаточной мере) принципов работы с файлами, реализация (в будущей работе) иных способов хранения информации не должна представлять чрезмерных трудностей.

Хотя информация в файле всегда хранится в виде некоторого набора байтов, принято различать текстовые файлы (их можно открыть в простейшем текстовом редакторе, например «Блокнот», и внести необходимые изменения) и двоичные файлы (попытка открытия таких файлов в блокноте ничего полезного не даст). Как бы то ни было, перед началом работы с файлом необходимо объявить и инициализировать некоторую переменную (в терминологии языка Pascal – файловую переменную), после чего ее необходимо указывать в каждом операторе, выполняющем ту или иную работу с файлом.

Наиболее простой и понятной является работа с текстовыми файлами, поскольку пользователь в любой момент может открыть созданный файл из программы «Блокнот» и, при необходимости, внести в него свои изменения. В приведенном ниже примере программа в цикле требует от пользователя ввести строку, пока он не введет символ «*». Все вводимые строки будут сохраняться в указанный текстовый файл.

```
var
  F: Text; {Объявление файловой переменной}
  S: string[100];
begin
  Assign(F, 'C:\TEMP\MyFile.txt'); {Инициализируем файловую переменную}
  Rewrite(F); {Создаем текстовый файл с правом на запись}
  repeat {основные действия выполняются в цикле}
    Write('vvedite stroku> '); {вывод приглашения на экран}
    Readln(S); {ожидаем, пока пользователь не введет строку}
    Writeln(F, S); {запись введенной строки в файл}
  until S = '*'; {если ввели "*", то выход из цикла}
  Close(F); {Закрываем файл}
end.
```

После окончания работы с программой предлагается открыть полученный файл с помощью блокнота и убедиться в наличии всех введенных строк.

При работе с файлом необходимо в первую очередь инициализировать файловую переменную (в данном случае F: Text). Файловая переменная внутренне объявлена как RECORD с многочисленными полями, в том числе «имя файла» и «файловый указатель». Таким образом, данная структура является самодостаточной, что избавляет от необходимости многократного указания имени файла. Далее файл необходимо открыть. Существует несколько процедур открытия файлов (это связано с существованием большого количества режимов открытия), в результате успешной работы которых происходит открытие файла, после чего можно с ним работать, т.е. выполнять операции ввода/вывода. После завершения всех операций ввода/вывода файл необходимо закрыть. Если этого не сделать, то дальнейшие попытки открыть файл (например, в другой процедуре) могут окончиться неудачно.

Помимо текстовых файлов, язык Pascal существенно упрощает работу также и с двоичными файлами. Двоичный файл в общем случае нельзя открыть с помощью блокнота, однако, это более предпочтительный, универсальный и экономичный формат для хранения разного рода информации. Проще всего работать с «типизированными» двоичными файлами, поскольку в этом случае работа с файлом напоминает работу с одномерным массивом. Например, так выглядит код сохранения элементов массива в типизированный файл:

```
var
  F: file of TBirthDay; {Объявляем файловую переменную}
  I: Integer;
begin
  Assign(F, 'MyFile.bin'); {Инициализируем файловую переменную}
  Rewrite(F);            {Открываем файл для записи}
  for I := 1 to ACount do {Цикл перебора записей}
```

```
    Write(F, Mas[I]);     {Сохраняем запись в файл}
  Close(F);               {Закрываем файл}
end;
```

Кроме того, Pascal позволяет работать с файлами, которые невозможно отнести ни к одному известному типу. В терминологии языка Pascal такие файлы являются «нетипизированными» (фактически – это массив байтов). На самом деле это наиболее привычный для большинства программистов способ представления, поскольку в таких файлах можно хранить совершенно любую информацию, как текстовую, так и двоичную. Разумеется, организация работы с нетипизированными файлами является несколько более сложной. Для ввода/вывода вместо процедур Read и Write используются процедуры BlockRead и BlockWrite. Более подробную информацию по работе с нетипизированными файлами в языке Pascal читатель может с легкостью получить в других источниках.

Процедуры по обработке текстовых и типизированных файлов, требуемые для выполнения данной лабораторной работы, приведены в таблице 7.1.

Таблица 7.1 – Процедуры для работы с файлами

Процедура	Назначение
Assign()	Связывает имя файла с файловой переменной
Append()	Открывает текстовый файл для добавления новых записей в конец файла. Если файл не найден, то возникнет ошибка
Reset()	Открывает текстовый файл для чтения, или двоичный файл для чтения/записи. Если файл не найден, то возникнет ошибка
Rewrite()	Открывает текстовый или двоичный файл для записи. Файл будет создан и обнулен автоматически
Write()	Записывает заданное значение в файл
Writeln()	Записывает строку текста и символы перевода строки в текстовый файл
Read()	Считывает информацию из файла в заданную переменную
Readln()	Считывает строку из текстового файла
Eof()	Проверяет, не достигнут ли конец файла при чтении
Erase()	Уничтожает заданный файл
Close()	Выполняет закрытие файла

7.6 Пример программы

В приведенном ниже примере реализована программа «Телефонный справочник», отвечающая требованиям к данной лабораторной работе (п. 7.3). Программа подробно прокомментирована, поэтому не нуждается в дополнительных пояснениях. Рекомендуется тщательным образом проанализировать предлагаемый пример, поскольку в нем найдутся ответы на многие вопросы, которые, несомненно, возникнут при выполнении работы.

```
program PhoneBook; {Программа "Телефонная книга"}

const
  AFileName = 'phones.spr'; {Имя файла}
  MaxRecordCount = 100; {Максимальное число записей в справочнике}

type
  TFam = string[20]; {Объявление типа "фамилия абонента"}

  {Описание записи из телефонной книги}
  TPhoneRec = record
    Fam: TFam;          {фамилия абонента}
    Phone: string[15]; {телефон абонента}
    Pol: Boolean;       {пол абонента TRUE-муж, FALSE-жен}
    BirthYear: Word;    {год рождения}
  end;
```

```
  {Объявление типа-массива записей телефонной книги}
  TPhoneTable = array[1..MaxRecordCount] of TPhoneRec;

{Процедура выполняет загрузку справочника из файла}
procedure LoadFromFile(var ATable: TPhoneTable; var ACount: Integer);
var
  F: file of TPhoneRec; {Объявляем файловую переменную}
begin
  Assign(F, AFileName); {Инициализируем файловую переменную}
  {$I-}     {Отключаем генерацию ошибок ввода/вывода}
  Reset(F); {Открываем файл для чтения}
  {$I+}     {Возвращаем директиву "I" в исходное состояние}
  if IOResult = 0 then {Если файл успешно открыт}
  begin {то считываем из него записи}
    ACount := 0; {Обнуляем счетчик записей}
    while not Eof(F) do {Пока не достигнут конец файла...}
    begin
      Inc(ACount); {Увеличиваем ACount на единицу }
      Read(F, ATable[ACount]); {Считываем запись из файла}
    end;
    Close(F); {Закрываем файл}
    Writeln('Spravochnik uspeshno zagruzhen iz faila: ', AFileName,
      '. Kolichetvo zapisei: ', ACount);
  end else {Иначе выводим сообщение "Файл не найден!"}
    Writeln('ERROR: Fail ne naiden!');
end;

{Процедура выполняет сохранение справочника в файл}
procedure SaveToFile(const ATable: TPhoneTable; const ACount: Integer);
var
  F: file of TPhoneRec; {Объявляем файловую переменую}
  I: Integer;
begin
  Assign(F, AFileName);   {Инициализируем файловую переменную}
  Rewrite(F);             {Открываем файл для записи}
  for I := 1 to ACount do {Цикл перебора записей}
    Write(F, ATable[I]);  {Сохраняем запись в файл}
  Close(F);               {Закрываем файл}
  Writeln('Spravochnik uspeshno sohranen v fail: ', AFileName);
end;

{Процедура выводит на экран запись с указанным номером}
procedure ShowRecord(const ATable: TPhoneTable; Num: Integer);
var
  ARec: TPhoneRec;
  C: Char;
begin
  ARec := ATable[Num];
  if ARec.Pol then {if ARec.Pol = True}
    C := 'm'
  else
    C := 'f';

  Writeln('No', Num, ': Fam=', ARec.Fam, ', Tel=',
    ARec.Phone, ', Pol=', C, ', God rozhd=', ARec.BirthYear);
end;
```

```
{Процедура выводит на экран список всех записей}
procedure ShowAllRecords(const ATable: TPhoneTable;
  const ACount: Integer);
var
  I: Integer;
begin
  Writeln('Spisok vseh zapisei:');
  for I := 1 to ACount do
    ShowRecord(ATable, I);
end;

{Процедура добавления новой записи в справочник}
procedure AddNewRecord(var ATable: TPhoneTable; var ACount: Integer);
var
  ARec: TPhoneRec;
  C: Char;
begin
  Write('Vvedite familiu: ');
  Readln(ARec.Fam);
  Write('Vvedite telefon: ');
  Readln(ARec.Phone);
  Write('Vvedite pol ("m" / "f"): ');
  Readln(C);
  if C = 'm' then
    ARec.Pol := True   {Мужчина}
  else { if C = 'f' }
    ARec.Pol := False; {Женщина}
  Write('Vvedite god rozhdenia: ');
  Readln(ARec.BirthYear);

  Inc(ACount); {Увеличиваем счетчик записей на 1}
  ATable[ACount] := ARec; {Записываем запись ARec в массив ATable}

  Write('Dobablena zapis: ');
  ShowRecord(ATable, ACount); {Печатаем запись на экран}
end;

{Процедура отыскивает запись по заданной фамилии и печатает на экране}
procedure FindRecord(const ATable: TPhoneTable; const ACount: Integer);
var
  I: Integer;
  IsFind: Boolean;
  Fam: TFam;
begin
  Write('Vvedite familiu: ');
  Readln(Fam);

  IsFind := False; {Сбрасываем флаг перед поиском}
  for I := 1 to ACount do
    if ATable[I].Fam = Fam then
    begin
      IsFind := True; {Устанавливаем флаг "Успешный поиск"}
      Write('Zapis naidena: ');
      ShowRecord(ATable, I);
      Break; {Выход из цикла}
```

```
    end;

  if not IsFind then {Если запись не найдена}
    Writeln('Zapis ne naidena!');
end;

var
  MenuNum: Byte;
  PhoneTable: TPhoneTable; {Переменная-справочник}
  PhoneCount: Integer; {Текущее количество записей в справочнике}
begin
  PhoneCount := 0; {При запуске программы справочник еще пуст}
  Writeln('Telefonnaya kniga. Avtor: Ivanov I.I');
  repeat
    Writeln('Vvedite cifru dlya vypolneniya deistviya:');
    Writeln('1 - zagruzka spravochnika iz faila');
    Writeln('2 - novaya zapis');
    Writeln('3 - spisok vseh zapisei');
    Writeln('4 - poisk zapisi po familii');
    Writeln('5 - sohranenie spravochnika v fail');
    Writeln('6 - vyhod iz programmy');
    Write('> ');
    Readln(MenuNum);
    case MenuNum of
      1: LoadFromFile(PhoneTable, PhoneCount);
      2: AddNewRecord(PhoneTable, PhoneCount);
      3: ShowAllRecords(PhoneTable, PhoneCount);
      4: FindRecord(PhoneTable, PhoneCount);
      5: SaveToFile(PhoneTable, PhoneCount);
    end;
  until MenuNum = 6;
end.
```

Следует отметить, что благодаря разбивке программного кода на подпрограммы, основная часть программы оказалась очень простой. Несмотря на «солидный» размер (более 150 строк кода), разобраться в данном примере не составит чрезмерных трудностей (при условии последовательного выполнения всех предыдущих лабораторных работ).

7.7. Варианты заданий

№	Наименование справочника, поля	Тип файла	Поле поиска
1	**Контрагенты**. Поля: наименование контрагента, ИНН, тип (True-юридическое, False-физическое лицо), год регистрации	Текстовый	Наименование контрагента
2	**Владельцы сотовых телефонов**. Поля: ФИО владельца, модель телефона, идентификатор телефона (IMEI), дата приобретения	Типизированный	ФИО владельца
3	**Участники интернет-форума**. Поля: ФИО участника, пол (True-мужской, False-женский), пароль, дата регистрации,	Текстовый	ФИО участника
4	**Улицы вашего города**. Поля: название улицы, количество домов, протяженность (км), год основания	Типизированный	Название улицы
5	**Товарно-материальные ценности**. Поля: наименование ТМЦ, штрих-код, количество на	Текстовый	Наименование ТМЦ

	складе, стоимость		
6	**Города России.** Поля: наименование города, год основания, число жителей, площадь	Типизированный	Наименование города
7	**Учебная нагрузка группы.** Поля: наименование предмета, количество часов, ФИО преподавателя, система оценки знаний (True-экзамен, False-зачет)	Текстовый	Наименование предмета
8	**Расписание занятий группы.** Поля: день недели, тип недели (True-первая, False-вторая), наименование предмета, время начала	Типизированный	Наименование предмета
9	**Расписание назначенных встреч.** Поля: ФИО, место встречи, дата и время встречи	Текстовый	ФИО
10	**Сотрудники.** Поля: ФИО, должность, дата приема на работу, оклад	Типизированный	ФИО
11	**Поступления ТМЦ.** Поля: наименование ТМЦ, наименование поставщика, количество, дата оприходования	Текстовый	Наименование ТМЦ

7.8. Содержание отчета (см. п. 1.11)

7.9. Контрольные вопросы

1) Дать определение записи (record) в языке Pascal.

2) Дать определение понятия «файл». Какова общая особенность любых файлов?

3) Перечислить основные операции с файлами.

4) Дать определение понятия «текстовый файл». Каким образом в языке Pascal объявляется переменная текстового файла?

5) Дать определение понятия «двоичный файл».

6) Дать определение понятия «типизированный файл». Каким образом в языке Pascal объявляется переменная типизированного файла?

7) Перечислить операторы языка Pascal для работы с текстовыми файлами.

8) Перечислить операторы языка Pascal для работы с типизированными файлами.

Список литературы

1 Фаронов В.В. Turbo Pascal 7.0. Начальный курс. Учебное пособие. – М.: «Нолидж», 2001. – 576 с.

2 Лукин С.Н. Pascal 7.0. Самоучитель. – 1999. – 214 с.

3 Долинский М.С. Алгоритмизация и программирование на Turbo Pascal: от простых до олимпиадных задач: Учебное пособие. – СПб.: Питер, 2005. – 234 с.

4 Деревенец О.В. Песни о паскале. – 2011. – 590 с.

5 Фаронов В.В. Delphi 6. Учебный курс. – СПб.: Питер, 2002. – 512 с.

6 Сухарев М.В. Основы Delphi. Профессиональный подход. – СПб: Наука и техника, 2004. – 600 с.

7 Стив Тейксейра, Ксавье Пачеко. Delphi5. Руководство разработчика. – 2000. – 1816 с.

Государственное автономное профессиональное образовательное
учреждение Свердловской области
«Ирбитский мотоциклетный техникум»

Комплекс лабораторных работ по программированию на языке Delphi

Разработал: Лагунов А.А.

Содержание

Урок 1 - Открываем Delphi, рассматриваем окна, создаем нашу первую программу!. 72
Урок 2 - Переменные и их типы ... 78
Урок 3 - Конструкция IF...THEN...ELSE ... 81
Урок 4 – Циклы ... 84
Урок 5 – Функции ... 87
Урок 6 - Одномерные массивы ... 89
Урок 7 - Многомерные массивы ... 91
Урок 8 - Форма и её свойства ... 92
Урок 9 - События. Программное изменение свойств ... 96
Урок 10 - Знакомство с компонентами (часть 1/12) ... 98
Урок 11 Знакомство с компонентами (часть 2/12) ... 100
Урок 12 Знакомство с компонентами (часть 3/12) ... 104
Урок 13 Знакомство с компонентами (часть 4/12) ... 107
Урок 14 Знакомство с компонентами (часть 5/12) ... 111
Урок 15 Знакомство с компонентами (часть 6/12) ... 116
Урок 16 Знакомство с компонентами (часть 7/12) ... 118
Урок 17 Знакомство с компонентами (часть 8/12) ... 120
Урок 18 Знакомство с компонентами (часть 9/12) ... 123
Урок 19 Знакомство с компонентами (часть 10/12) ... 125
Урок 20 Знакомство с компонентами (часть 11/12) ... 127
Урок 21 Знакомство с компонентами (часть 12/12) ... 128
Урок 22 Принцип работы с файлами ... 131
Урок 23 - Функции для работы с мышью ... 133
Урок 24 - Изучаем компонент PaintBox ... 134
Урок 25 - Подробное изучение RichEdit'a ... 136
Урок 26 - Создаем игру Ping-pong - часть(1/3) ... 140
Урок 27 - Создаем игру Ping-pong - часть(2/3) ... 142
Урок 28 - Создаем игру Ping-pong - часть(3/3) ... 144
Урок 29 - Работа с DLL ... 145
Урок 30 - Знакомство с базами данных ... 147
Урок 31 - Продолжение работы с базами данных ... 149
Урок 32 - Объединение всего изученного про базы данных ... 151
Урок 33 Отчеты в Delphi ... 155
Урок 34 Пример создания отчета с использованием Rave Reports ... 157
Урок 35 - Автовыключатель компьютера ... 159
Урок 36 - Шифрование информации ... 161
Урок 37 - Создаем Веб браузер ... 161
Урок 38 - Взаимодействие с веб страницей ... 164
Урок 39 - Запись рабочего стола ... 166
Урок 40 - Запись рабочего стола, интерфейс ... 168
Урок 41 - Панель быстрого запуска (часть 1/2) ... 169
Урок 42 - Панель быстрого запуска (часть 2/2) ... 171
Урок 43,44 - Структурные типы данных ... 173
Урок 45,46 - Динамическое создание компонентов ... 178
Урок 47 - Исключительные ситуации ... 180
Урок 48,49 - Потоки в Delphi ... 182
Урок 51,52 - Создание собственных процедур и функций Delphi ... 186
Урок 53 - Классы Delphi ... 189
Урок 54 - INI файлы ... 193
Урок 55 - Реестр Windows, (часть 1/2) ... 194
Урок 56 - Реестр Windows, (часть 2/2) ... 195
Урок 57 - Динамические библиотеки DLL ... 199
Урок 58 - Работа с сжатыми файлами ... 202
Урок 59 - Получение хеша файла ... 203

Урок 60 - Указатели ... 204
Урок 61 - Создание и использование интерфейса (часть 1/2) ... 208
Урок 62 - Создание и использование интерфейса (часть 2/2) ... 210
Урок 63 - Работа с реестром ... 212
Урок 64 - Использование потоков данных (часть 1/3) ... 213
Урок 65 - Использование потоков данных (часть 2/3) ... 215
Урок 66 - Использование потоков данных (часть 3/3) ... 217
Урок 67 - Работа с памятью в системе Windows32 (часть 1/3) ... 218
Урок 68 - Работа с памятью в системе Windows32 (часть 2/3) ... 221
Урок 69 - Работа с памятью в системе Windows32 (часть 3/3) ... 222
Урок 70 - Создание своих компонентов (часть 1/3) ... 224
Урок 71 - Создание своих компонентов (часть 2/3) ... 226
Урок 72 - Создание своих компонентов (часть 3/3) ... 230
Урок 73 - Оператор Case ... 233
Урок 74 - Оператор GOTO ... 234
Урок 75 - Рекурсия ... 235
Урок 76 - Множества ... 237
Урок 77 - Создание компонентов для Delphi ... 241
Урок 78 – Написание интерфейса для БД ... 251
Урок 79- Соеденение двух форм ... 256

Урок 1 - Открываем Delphi, рассматриваем окна, создаем нашу первую программу!

Здравствуй уважаемый новичок! В этом уроке мы познакомимся с Delphi 7 и научимся компилировать программу.

Итак, если вы приняли решение изучать язык программирования Delphi, то сразу без предисловий перейдем к делу.

Программа Delphi 7 состоит из четырех основных окон:

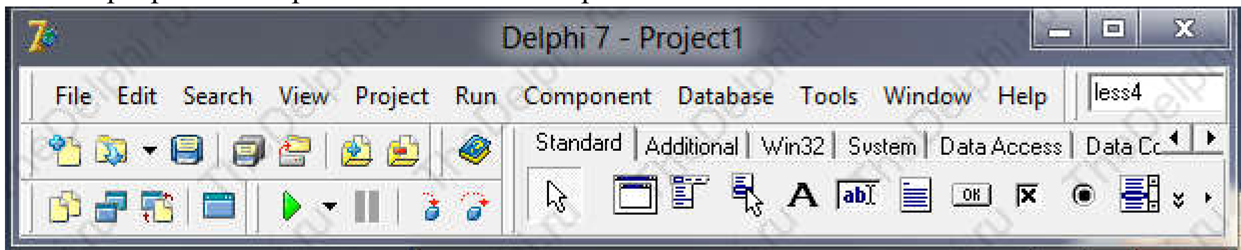


Рис. 1. Главное окно

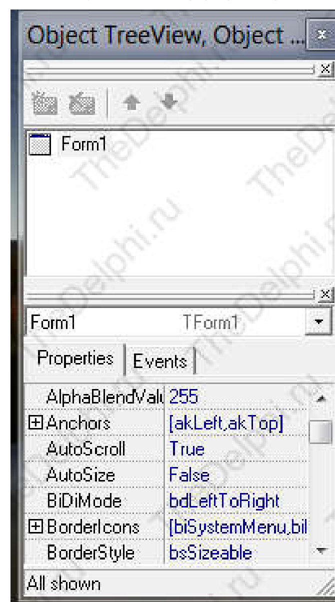


Рис. 2. Инспектор объектов

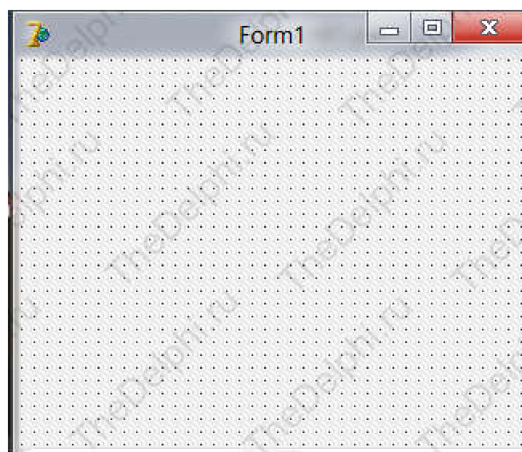


Рис. 3. Окно форм

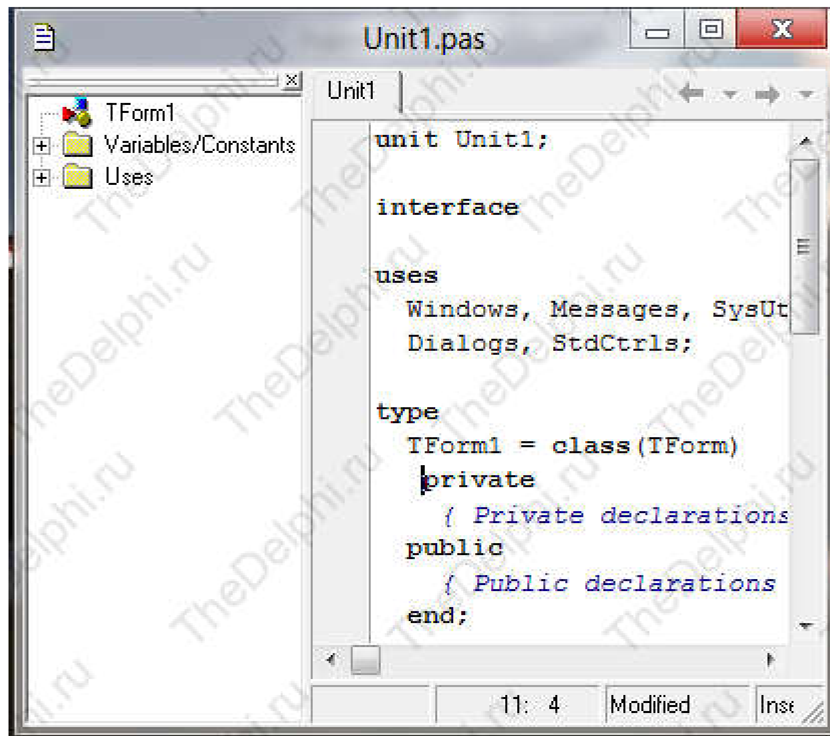


Рис. 4. Редактор кода

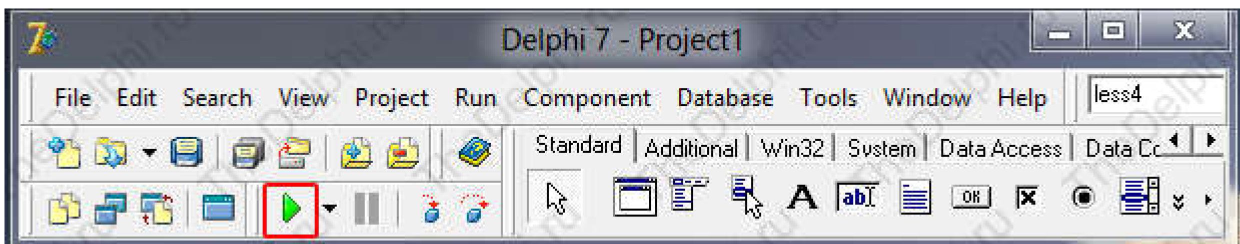
Главное окно – окно управления проектом и средой разработки. Здесь же находятся вкладки с компонентами (Рис. 1).

Инспектор объектов – окно, в котором задаются свойства различным компонентам (Рис. 2).

Окно форм – визуальное окно программы. (Рис. 3).

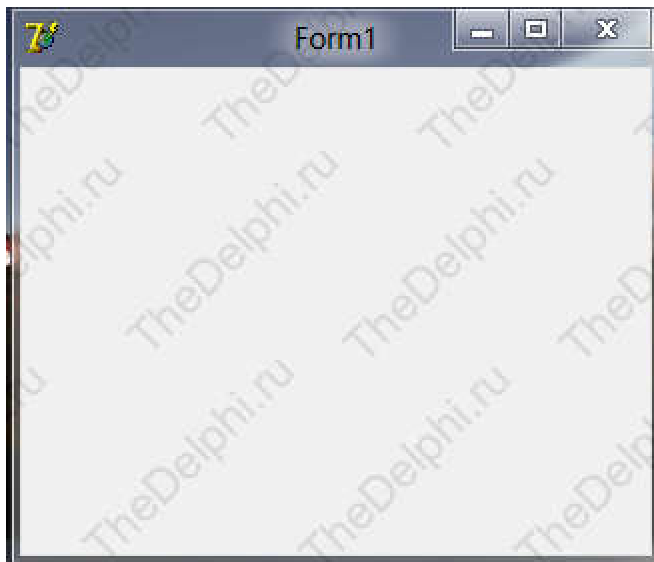
Редактор кода – окно, в которое мы будем записывать код (Рис. 4).

!!! Давайте уже напишем нашу первую программу! Нажмите на зеленый треугольник в главном окне.

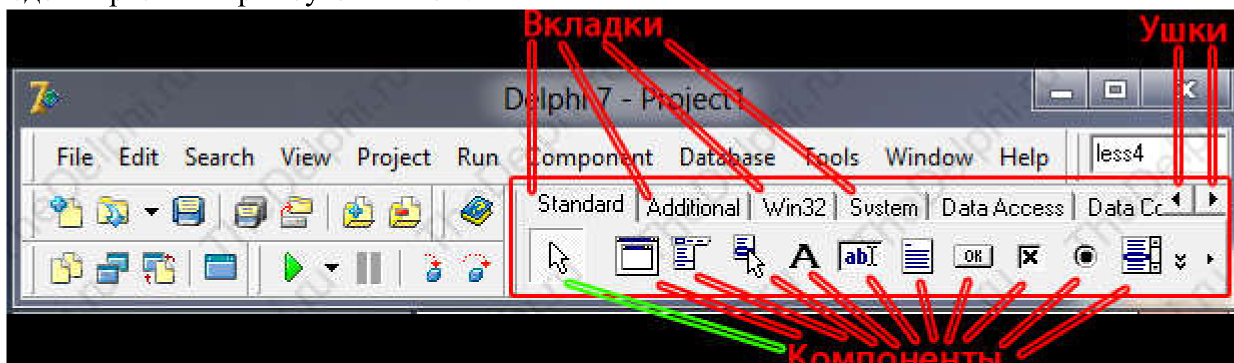


После нажатия на зеленый треугольник, наша программа начнет компилироваться (создаваться), то есть компилятор проверит окно редактора кода на наличие ошибок, но так как мы в окно редактора кода еще ничего не писали, то ошибок возникнуть не должно. После компиляции, Delphi запустит нашу программу для тестирования.

Вот что мы увидим:



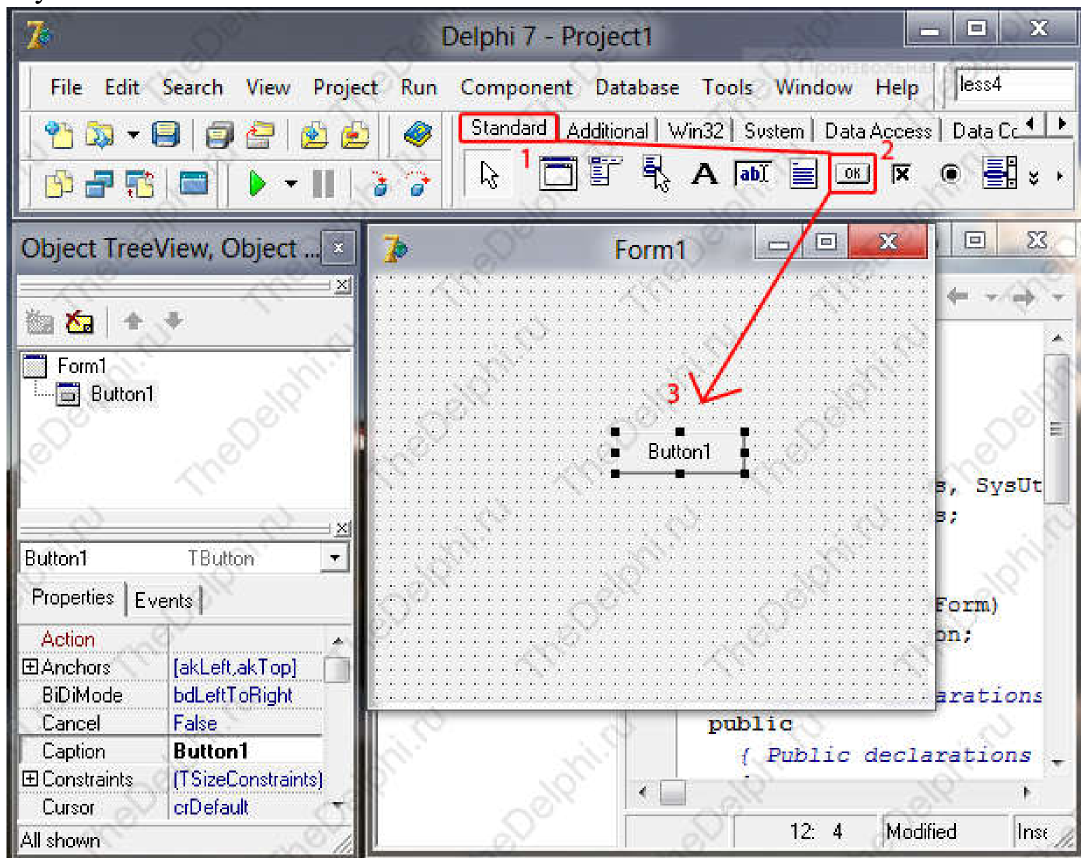
Теперь закройте скомпилированную программу (Рис. выше) и мы продолжаем знакомство с Delphi. В главном окне можно увидеть палитру компонентов. На рисунке она обведена красным прямоугольником.



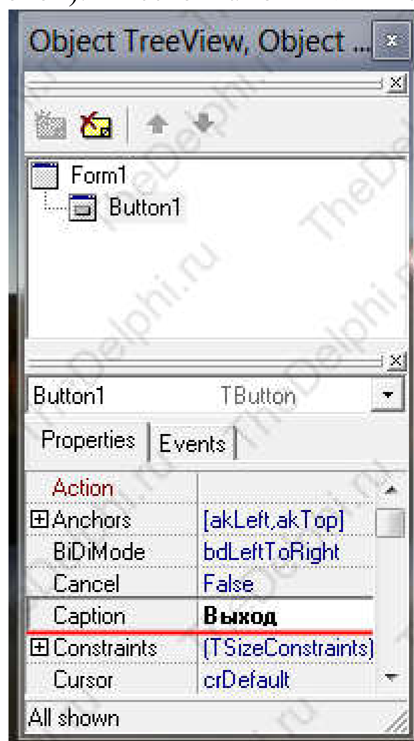
Между компонентами можно переключаться, щелкая по вкладкам. Каждая вкладка имеет свои уникальные компоненты. Так же, если у вас не помещаются все вкладки в палитру компонентов, то появляются так называемые ушки, для перелистывания вкладок в ту или иную сторону. На вкладке Standard самым первым стоит значок курсора (к нему проведена зеленая линия), это не компонент. Он нужен для отмены выбора компонента. Вот, например, вы выбрали какой-то компонент и для того, чтобы отменить свой выбор, нужно просто щелкнуть по этому значку курсора. Существуют не только стандартные компоненты Delphi, но и создаваемые любителями. О том, как устанавливать компоненты мы поговорим в следующих уроках.

А теперь добавим на форму компонент под названием Button (кнопка). Он находится на вкладке Standard. Нажмите на него, а потом нажмите где-нибудь на форме.

Получится вот так:

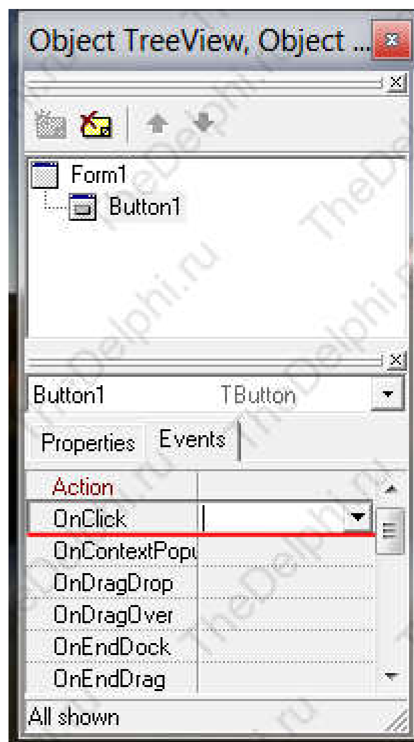


Готово! Кнопка на форме! Перейдем в инспектор объектов. На вкладке Properties отображены все свойства компонента, в данном случае свойства нашей кнопки. На вкладке Events отображены все события компонента, в данном случае опять же свойства нашей кнопки. Изменим свойство Caption (заголовок). Вместо Button1 напишем Выход:

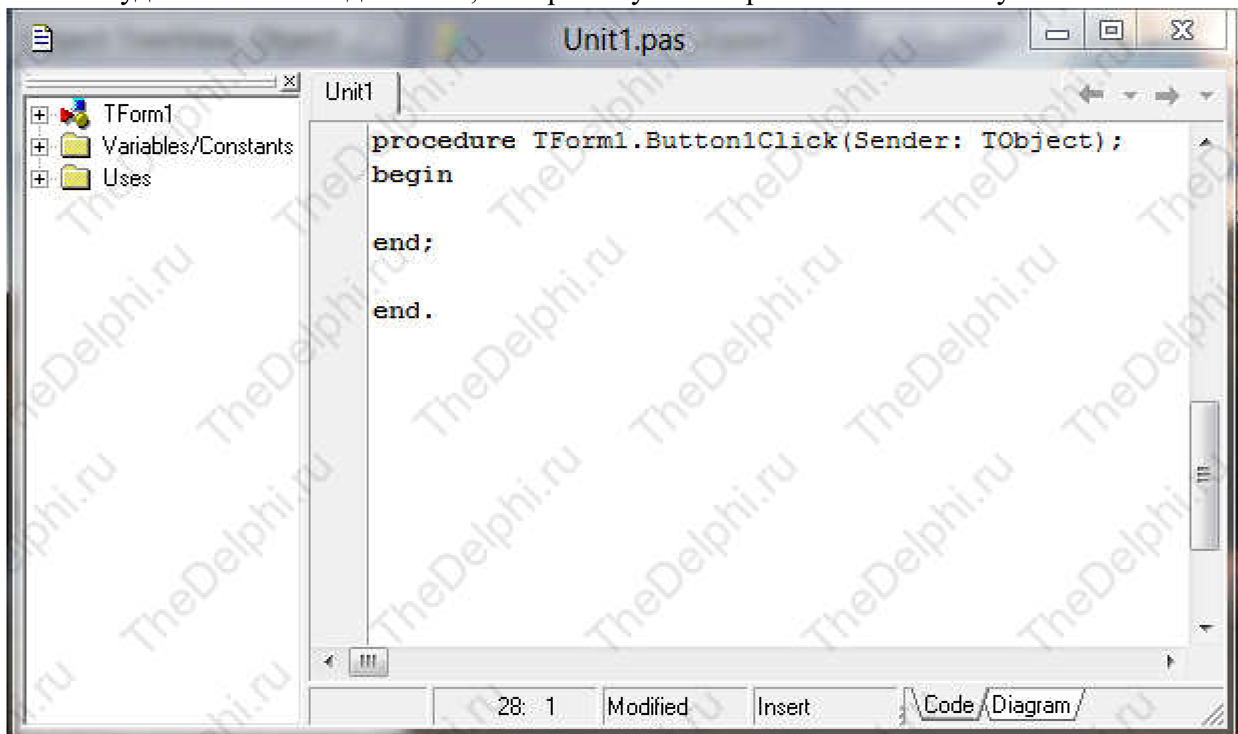


Вы можете поэкспериментировать с другими свойствами, изменяя их значения. Например, попробуйте изменить свойства Width и height (ширина и высота), left и top (расстояние слева и сверху внутри формы), Cursor (выберите любой курсор из списка и после компиляции при наведении на кнопку будет отображаться выбранный вами курсор).

Маленько отвлеклись от курса. Продолжаем. На форме изменилось название кнопки. Теперь перейдем на вкладку Events и кликнем 2 раза напротив надписи OnClick:



Если все сделано правильно, то автоматически станет активным окно редактора кода, в него мы и будем вписывать действие, которое случится при клике на кнопку.



Кликая 2 раза по событию OnClick, мы вставили процедуру, такую некую заготовку. Думаю, по названию кнопки вы догадались, что при нажатии на нее программа закроется. Нам надо осуществить это действие. Пишем команду **close** между ключевыми словами begin и end. Все команды в Delphi заканчиваются точкой с запятой, по этому, между ключевыми словами должен быть код close. Вообще в Delphi, все команды пишутся между ключевыми словами begin и end, то есть начинаем и заканчиваем.

После всех манипуляций получилось вот такое чудо:

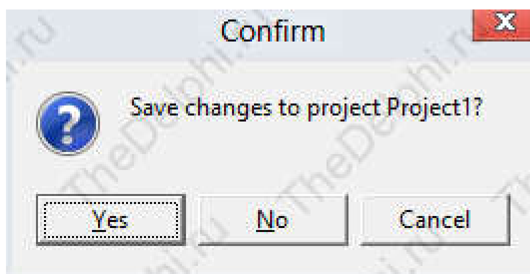
```
1  procedure TForm1.Button1Click(Sender: TObject);
2  begin
3  close;
4  end;
```

Компилируем нашу программу, нажимая на зеленый треугольник или на клавишу F9. Кстати, для того, чтобы просто скомпилировать программу и проверить код на ошибки, не запуская программу потом для тестирования, нужно нажать Ctrl + F9.

По умолчанию программа компилируется в папке C:\Program

Files\Borland\Delphi7\Projects. Давайте, сохраним проект в другую папку. Нажимаем в меню главного окна File->Save Project As… и выбираем папку для сохранения. Сейчас в папке находится только проект, для того чтобы там появился exe-файл, нужно еще раз скомпилировать программу.

Теперь давайте закроем наш проект, а потом снова откроем. Нажимаем в главном окне File->Close All, если вы не сохранили проект или после сохранения где-то его изменили, то у вас вылезет окно с запросом на сохранение проекта, нажмите на кнопку Yes и сохраните проект.



Далее в главном окне нажимаем File->Open Project... и открываем проект. У меня проект называется Project1.dpr, если вы при сохранении изменяли название, то соответственно открывайте то, что сохранили.

Что бы создать новый проект, нажмите File->New->Application.

Ну вот и всё!

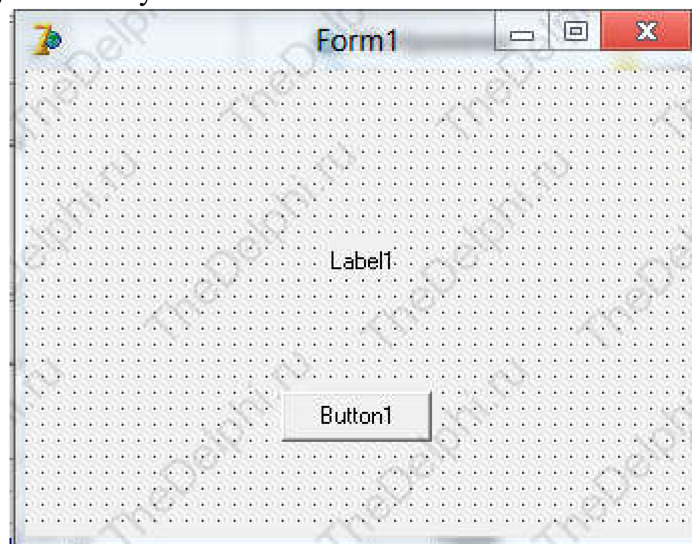
Урок 2 - Переменные и их типы

Продолжаем наше обучение! В Delphi очень важную роль играют переменные. В процессе работы программы в переменных можно, как хранить, так и извлекать информацию. Переменные могут иметь разный тип. Например, для того, чтобы в переменную записать какой-нибудь текст, используется тип String. Для того, что бы записать в переменную число, используют тип Integer.

Вот список основных типов переменных в Delphi:
- Integer - целые числа из диапазона: -2147483648..+2147483647;
- Shortin - целые числа из диапазона: -128..+127;
- Byte - целые числа из диапазона: 0..+255;
- Real - как целые так и дробные числа из диапазона: 5e-324..1.7e+308;
- Double - схож с типом Real;
- String - строковый тип данных;
- Char - символьный тип данных;
- Bollean - логический тип данных. Может принимать значение True - истина или False – ложь.

С теорией мы закончили, теперь давайте откроем Delphi 7 и создадим новый проект. После этого кидаем на форму знакомый нам компонент Button и еще не знакомый Label. Компонент Label эта такая полезная вещь, в которую можно записать какую-нибудь надпись. Например, надпись для другого компонента или просто записать в него автора программы. Попробуйте отыскать компонент Label сами, наводя на все компоненты во вкладке Standard и читая всплывающую подсказку. Кому сложно найти, то это четвертый компонент слева, не считая значка курсора.

Я всё сделал и у меня получилось вот так:



Сейчас нам нужно создать событие OnClick на кнопке, я надеюсь, что вы помните, как это делать.

Переменные объявляются между ключевыми словами procedure и begin. Объявление начинается с ключевого слова **var**, потом пишется имя переменной и через двоеточие её тип. Заканчивается все как всегда точкой с запятой.

Создадим переменную **S** типа String в процедуре OnClick:

```
1 procedure TForm1.Button1Click(Sender: TObject);
2 var S:string;
3 begin
4
5 end;
```

После этого между ключевыми словами begin и end присвоим переменной значение равное **'Моя первая переменная'**. Присвоение пишется следующим образом. Пишем имя переменной, оператор присвоения **:=** и значение. Если мы записываем информацию типа String, то информация заключается в одинарные кавычки.

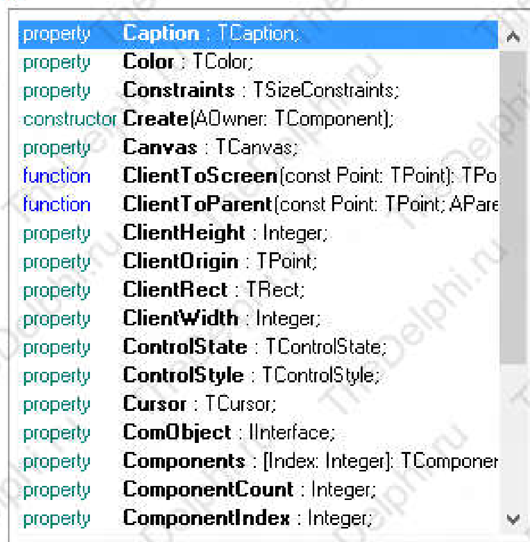
Общий вид:

```
1 procedure TForm1.Button1Click(Sender: TObject);
2 var S:string;
3 begin
4 S:='Моя первая переменная';
5 end;
```

Теперь если скомпилировать программу и нажать на кнопку ничего существенного не произойдет, просто в переменную запишется значение и всё. Попробуем вывести значение из переменной. Делается это также просто как и записывается. Выводить значение мы будем в наш Label.

Синтаксис такой: **Label1.Caption:=S;**

Разберем этот код подробно. Сначала мы написали **Label1**, потом пишем **точку** и в Delphi появляется огромный список со свойствами данного компонента. Можно конечно порыться и отыскать там Caption, но мы будем умнее! Мы, после того как поставили точку, напишем еще букву **C** и Delphi как бы отсортирует все свойства и найдет все, которые начинаются с буквы C. Первым в списке как раз будет свойство Caption.



Выбираем его из списка и нажимаем на Enter. Заметьте, что мы писали **Label1.C**, но после того, как нажали Enter, Delphi сам дописал название свойства. Далее опять же идет оператор присвоения и наша переменная.

Вы наверняка спросите: "Зачем же переменная, если можно было написать Label1.Caption:='Моя первая переменная'; ?". Ответ простой. Это нужно затем, что мы изучаем переменные.

Нет, на самом деле так присваивать тоже можно, но представьте такую ситуацию, что вы написали очень большую, популярную программу и у вас, там в программе, пятидесяти компонентам присваивается одно и тоже значение и вот перед вами встала задача: "Поменять это значение на более универсальное и понятное для пользователя".

Что вы будете делать?

В первом случае у вас всем этим компонентам присваивается одна и та же переменная и чтобы изменить всем этим пятидесяти компонентам значение вам просто нужно поменять значение в переменной.

Во втором случае вы сидите 20 минут и всё копируете и копируете значение всем пятидесяти компонентам.

Вывод делайте сами.

И так, продолжаем! В общем виде должно быть так:

```
procedure TForm1.Button1Click(Sender: TObject);
var S:string;
begin
S:='Моя первая переменная';
Label1.Caption:=S;
end;
```

Компилируем нашу программу и нажимаем на Button (кнопку). Сразу же компонент Label вместо Label1 будет показывать **Моя первая переменная**.

На этом хотелось бы закончить, так как я уже устал писать урок :), но я еще не познакомил вас с типом Integer и как присваивать переменную с таким типом. Вы думаете, что присваивать ее нужно точно так же как и переменную типа String, но вы ошибаетесь. Дело в том, что свойству Caption вообще у всех компонентов можно присвоить только текстовые значения. Как мы будем присваивать числовой тип, если можно только текстовой? Всё проще некуда. Между типами переменных можно как бы переключаться, то есть можно из числового типа сделать текстовой и присвоить его компоненту Label. Этим мы сейчас и займемся.

Для начала нужно начать сначала. Объявим переменную с именем **I** и типом Integer, дописав ее к переменной S. Код:

```
procedure TForm1.Button1Click(Sender: TObject);
var S:string;  I:integer;
begin
...
```

Далее присвоим переменной I значение **21**.

```
I:=21;
```

Заметьте, что числовое значение записывается без одинарных кавычек! Теперь присвоим свойству Caption значение переменной I, для этого нужно воспользоваться оператором **IntToStr()**. Он как бы конвертирует числовой тип в текстовой. В скобках указывается переменная, которую нужно конвертировать.

Общий вид кода:

```
procedure TForm1.Button1Click(Sender: TObject);
var S:string;  I:integer;
begin
S:='Моя первая переменная';
Label1.Caption:=S;

I:=21;
Label1.Caption:=IntToStr(I);
end;
```

Скомпилируйте программу и вы увидите, что Label будет отображать значение переменной I, то есть **21.**

Ну вот и всё!

Урок 3 - Конструкция IF...THEN...ELSE

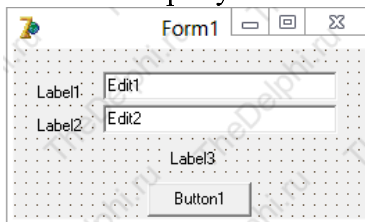
Здравствуйте, в этом уроке я познакомлю вас с конструкцией if...then...else и мы напишем программу проверки логина и пароля! И так, приступим! Конструкцией if...then...else можно проверять какое-нибудь условие, дословно она переводится так: если...то...иначе.

Конструкция имеет такой вид:

if (условие) then
begin
//Если условие верно, выполняем действия
end
else
begin
//Если условие не верно, выполняем действия
end;

Кстати, после двойного слеша "//" пишется комментарий. Он никак не влияет на код программы, потому что компилятор его игнорирует.

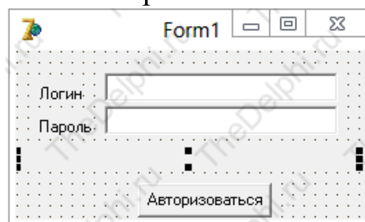
Теперь я попытаюсь вам объяснить всё на практике. Открываем Delphi и создаем новый проект. Кидаем на форму уже знакомые нам компоненты Button, Label 3 штуки и еще не знакомый Edit 2 штуки (он находится правее от компонента Label). Edit это обычное поле ввода, которое встречается в повседневной жизни, например при регистрации на сайте. Располагаем компоненты так, как показано на рисунке:



Переходим в **инспектор объектов** и изменяем свойства у компонентов по очереди:

Label1.Caption = 'Логин';
Label2.Caption = 'Пароль';
Label3.AutoSize = False. Выставив значение False у свойства AutoSize, мы запретили автоматически менять размер компоненту;
Label3.Caption = ' '. Когда мы стёрли весь текст, размер компонента не изменился;
Edit1.Text = ' ';
Edit2.Text = ' ';
Button1.Caption = 'Авторизоваться'.

Сейчас выделяем компонент Label3 и растягиваем его ширину на всю форму:



Нужно изменить еще одно свойство у компонента Label3, оно называется Alignment и отвечает за выравнивание текста по центру. Выставляем значение taCenter. Теперь весь текст будет появляться в этом лейбле по центру.

Мы завершили с настройкой формы, теперь создаем обработчик OnClick на нашей кнопке. И так, принцип работы нашей программы: если логин и пароль верны, то оповещаем об этом, иначе - выдаем ошибку.

Начнем с простого - проверка логина:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 begin
03 if Edit1.Text = 'admin' then //Если логин верный, то...
04 begin
05 Label3.Caption:='Вы авторизованны!'; //Авторизовываем пользователя
06 end
07 else //Иначе
08 begin
09 Label3.Caption:='Логин не верный!'; //Выдаем ошибку
10 end;
11 end;
```

Компилируем программу и вводим в наш Edit1 слово admin, нажимаем на кнопку и вуаля -Label3 оповестит о том, что мы авторизованы, теперь напишите что-нибудь другое в Edit1 и нажмите на кнопку. Label3 оповестит, что логин не верный.

Идем дальше - проверяем логин и пароль. Так как нужно проверять 2 условия, каждое из условий нужно окружить скобками, а между скобок напишем ключевое слово **and**:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 begin
03 if (Edit1.Text = 'admin') and (Edit2.Text = 'pass') then //Если логин И пароль верны то...
04 begin
05 Label3.Caption:='Вы авторизованны!'; //Авторизовываем пользователя
06 end
07 else //Иначе
08 begin
09 Label3.Caption:='Логин ИЛИ пароль не верный!'; //Выдаем ошибку
10 end;
11 end;
```

Компилируем программу и проверяем правильность работы кода.

Идем дальше - так же проверяем логин и пароль, но теперь между условиями напишем ключевое слово **or** вместо and это означает что авторизация пройдет если верно хоть одно из условий:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 begin
03 if (Edit1.Text = 'admin') or (Edit2.Text = 'pass') then //Если логин ИЛИ пароль верны то...
04 begin
05 Label3.Caption:='Вы авторизованны!'; //Авторизовываем пользователя
06 end
07 else //Иначе
08 begin
09 Label3.Caption:='Логин И пароль не верны!'; //Выдаем ошибку
10 end;
11 end;
```

Вернемся к коду, где использовали ключевое слово and между условиями. У нас там выдается ошибка 'Логин ИЛИ пароль не верный', если пользователь ошибся. Давайте сделаем ошибку конкретней, чтобы она сообщала, что именно не верно, логин или пароль? Для этого сотрем нашу не конкретную ошибку и добавим еще 3 конструкции if..then. Вместо знака равенства в условии, мы будем использовать знак не равенства <>.

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 begin
03 if (Edit1.Text = 'admin') and (Edit2.Text = 'pass') then //Если логин И пароль верны то...
04 begin
05 Label3.Caption:='Вы авторизованны!'; //Авторизовываем пользователя
06 end
07 else //Иначе
08 begin
09 if Edit1.Text <> 'admin' then //Если логин не верный
10 begin
11 Label3.Caption:='Логин не верный!'; //Выдаем ошибку
12 end;
13 if Edit2.Text <> 'pass' then //Если пароль не верный
14 begin
15 Label3.Caption:='Пароль не верный!'; //Выдаем ошибку
16 end;
17 if (Edit1.Text <> 'admin') and (Edit2.Text <> 'pass') then //Если логин и пароль не верны
18 begin
19 Label3.Caption:='Логин и пароль не верны!';  //Выдаем ошибку
20 end;
21 end;
22 end;
```

Задание на закрепление: дополнить в программу двух пользователей со своими паролями. Программа должна соответствовать следующим критериям:

- если вход осуществляется под определенным пользователем, то должна выходить соответствующая запись (например: зашли под пользователем Иванов, то выходит запись «Вы

авторизованы Иванов»);
- если ввели не существующий логин, то должна выходить запись: «Такого логина не существует».

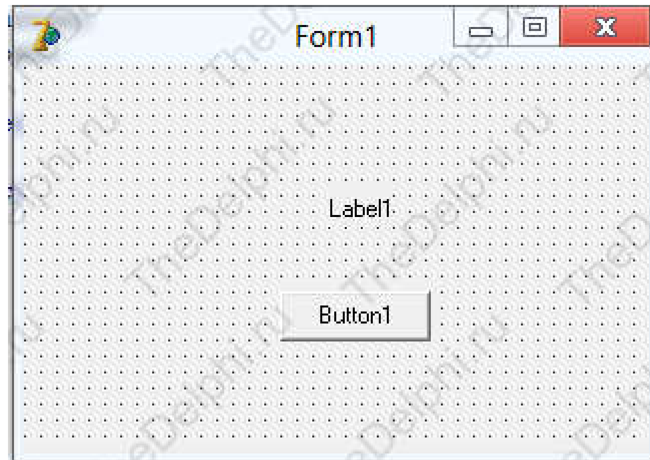
Урок 4 – Циклы

Цикл - это многократно повторяющаяся последовательность действий. Первый цикл, с которым мы познакомимся в этом уроке называется While...Do (делай пока верно условие).

Синтаксис:

```
While условие Do
begin
//тело цикла
end;
```

Сейчас нам нужно открыть Delphi и создать новый проект. Кидаем на форму компоненты Button и Label:



Создаем на кнопке процедуру OnClick и первое, что нам надо сделать - это ввести переменную **A** типа Integer:

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

Теперь между ключевыми словами begin и end установим значение переменной A равное **1**:

```
A:=1;
```

И сейчас мы напишем сам цикл, с условием A<>100, то есть пока A не равно 100 будет выполняться цикл. Если же A = 100 - цикл остановится:

```
While A<>100 do
begin
//Тело цикла
end;
```

Далее, нам нужно что-то сделать в теле цикла. Давайте будем увеличивать значение переменной A на единицу и выводить значение переменной в Label. Вместо комментария (//Тело цикла) мы напишем:

```
A:=A+1;
Label1.Caption:=IntToStr(A);
```

Общий вид кода:

```
procedure TForm1.Button1Click(Sender: TObject);
var A:integer;
begin
A:=1; //Присваеваем единицу
While A <> 100 do //Пока A не равно 100 - делаем
begin
A:=a+1; //прибавляяем единицу
Label1.Caption:=IntToStr(a); //Выводим значение A
end;
end;
```

Компилируем программу, нажимаем на кнопку и видим, что лабел показывает нам сотню. Но почему сразу сотню? Почему не 1,2,3,4 и так до ста. Дело в том, что цикл выполняется на столько быстро, что мы не замечаем как лабел выводит нам сначала 1 потом 2 и потом 3. По этому мы видим только сотню - конечный результат. Кто-то может подумать, что так не интересно. Хорошо, сейчас сделаем так, чтобы видеть как Delphi выполняет цикл.

Дописываем после строки:

```
Label1.Caption:=IntToStr(a);
```

Вот эти две строчки:

```
Application.ProcessMessages;
sleep(100);
```

Они делают следующее:

- Application.ProcessMessages - это метод, позволяющий выводить значения переменных во время работы цикла. Не смотря на то, что мы и так выводим переменную в лабел, этот метод необходим.

- sleep(100); - функция Sleep() говорит программе, что нужно поспать, как бы заморозиться на какое-то количество миллисекунд. Миллисекунды указываются в скобках. В секунде 1000 миллисекунд.

Общий вид кода:

```
procedure TForm1.Button1Click(Sender: TObject);
var A:integer;
begin
A:=1; //Присваеваем единицу
While A <> 100 do //Пока A не равно 100 - делаем
begin
A:=a+1; //прибавляяем единицу
Label1.Caption:=IntToStr(a); //Выводим значение A
Application.ProcessMessages;
sleep(100);
end;
end;
```

Компилируйте и проверяйте.

С циклом While мы закончили, теперь разберем цикл со счетчиком или другое его название For...To...Do. Данный цикл удобно применять, когда нам точно известно кол-во повторений.

Синтаксис:

```
For переменная счетчик:=Выражение №1 To выражение №2 Do
begin
//тело цикла
end;
```

Этот цикл называется со счетчиком, потому что он сам увеличивает переменную счетчик на единицу.

Первым делом нам нужно добавить переменную **S** типа Integer.

```
procedure TForm1.Button1Click(Sender: TObject);
var A, S:integer;
begin
...
```

Далее, пишем программу, которая будет считать сумму чисел от 1 до 100. То есть имеется ряд чисел 1 2 3 4 5 6 7 ... 100. Программа будет складывать эти числа между собой, то есть 1+2+3+4+5+6+7+...+100.

Стираем цикл While и пишем цикл **For**, но перед ним присвойте переменной S **ноль**:

```
For A:=1 to 100 do
begin
//Тело цикла
end;
```

Этот цикл повторит действия в теле 100 раз. В тело цикла мы запишем:

```
S:=s+a;
```

Программа будет считать сумму чисел от 1 до 100, прибавляя к переменной S переменную счетчик A. И после цикла выводим результат в лабел.

```
Label1.Caption:=IntToStr(S);
```

Общий вид:

```
procedure TForm1.Button1Click(Sender: TObject);
var A, S:integer;
begin
S:=0; //Присваеваем ноль
For A:=1 to 100 do
begin
S:=s+a; //Вычисления
end;
Label1.Caption:=IntToStr(S);
end;
```

У цикла For есть цикл двойник, он может считать в обратном порядке. Для этого нужно изменить ключевое слово To на DownTo.

Пример той же самой программы, но с обратным счетчиком:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 var A, S:integer;
03 begin
04 S:=0; //Присваеваем ноль
05 For A:=100 downto 1 do
06 begin
07 S:=s+a; //Вычисления
08 end;
09 Label1.Caption:=IntToStr(S);
10 end;
```

Далее. Знакомимся с циклом **Repeat**.

Синтаксис:

```
1 Repeat
2 //Тело цикла
3 Until условие;
```

Этот цикл сначала выполняет действие, а потом проверяет условие. Цикл выполниться в любом случае хотя бы один раз.

Стираем цикл For в нашей программе и пишем цикл **Repeat**:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 var A, S:integer;
03 begin
04 S:=0; //Присваеваем ноль
05 a:=0;
06
07 Repeat
08 a:=a+1;
09 S:=s+a; //Вычисления
10 Until a=100; //цикл будет выполняться пока а не достигнет 100
11
12 Label1.Caption:=IntToStr(S);
13 end;
```

Эта программа выполнит тоже самое что и предыдущая.

Ну вот мы и закончили обучение циклам! Сейчас выучим 2 команды для управления ими. Сразу приведу пример программы, который вам **выполнять не надо**, а потом прокомментирую что и как в ней работает:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 var A, S:integer;
03 begin
04 S:=0; //Присваеваем ноль
05 For A:=1 to 1000 do
06 begin
07 S:=s+a; //Вычисления
08 if S>100 then break else continue;
09 end;
10 Label1.Caption:=IntToStr(S);
11 end;
```

В теле цикла присутствует условие, которое проверяет переменную S. Если S больше 100, то мы экстренно выходим из цикла при помощи команды **break**, иначе продолжаем цикл командой **continue**. Пример программы не очень удачный, так как цикл будет работать даже если мы сотрем команду continue, но я надеюсь, что суть вы уловили.

Задание на закрепление: напишите программу, которая вычислит сумму двухзначных чисел в диапазоне от 1 до 100 и выведет ее в Label.

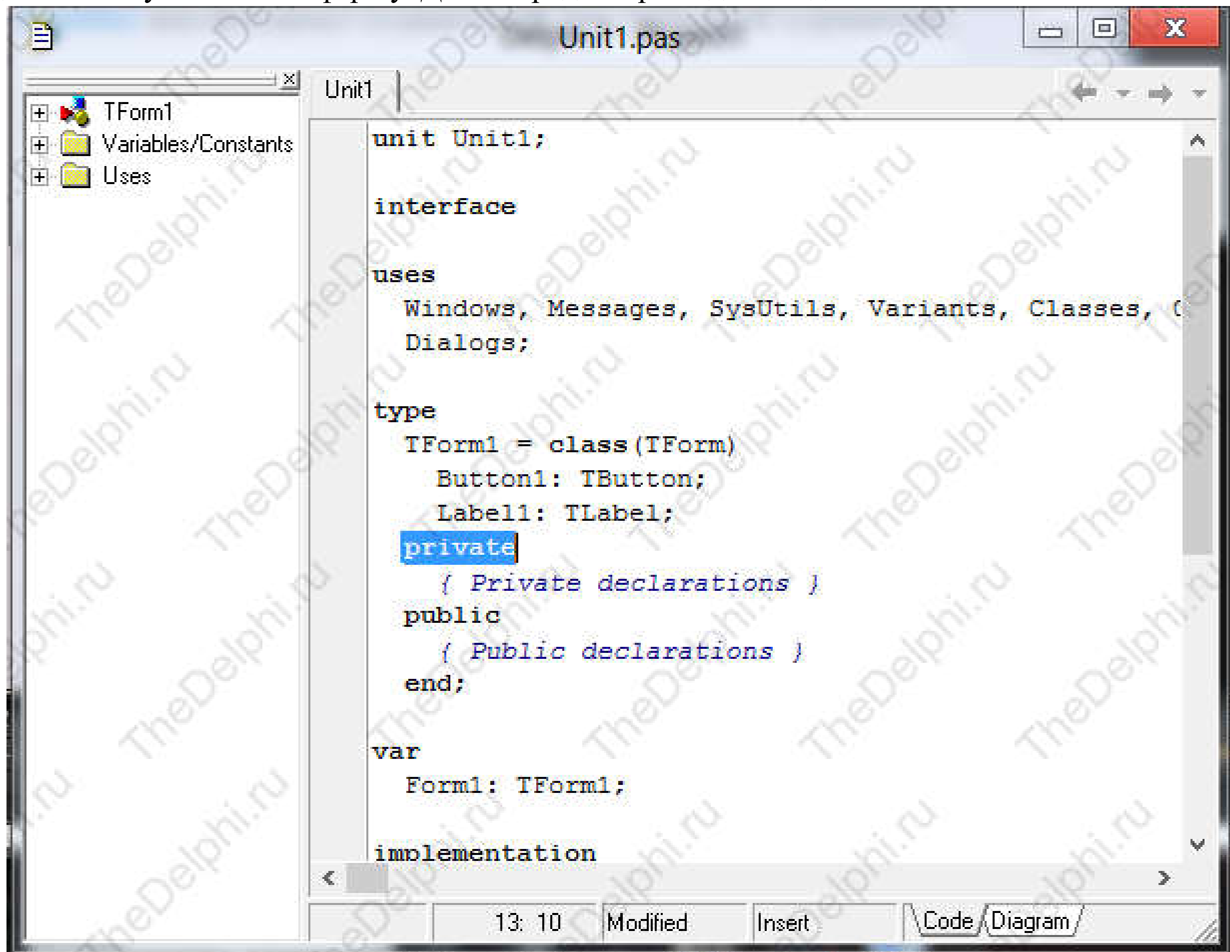
Урок 5 – Функции

Продолжаем обучение Delphi и в этом уроке мы познакомимся с функциями. Представьте, что вы написали очень большую программу в которой более 2000 строк и у вас десятки раз повторяется один и тот же участок кода. Функция позволяет избегать такие повторения. То есть мы выносим повторяющийся код в функцию, а на месте тех десятков кода просто вызываем нашу функцию.

Синтаксис функции:

```
1 function имя (входные параметры):тип выходной переменной;
```

Рассмотрим подробнее на примере программы. Запускаем Delphi, создаем проект и кидаем кнопку и лабел на форму. Далее в редакторе кода находим слово Private.



После слова Private объявляем функцию:

```
1 function sum (a, b:integer):integer;
```

Сейчас нажимаем комбинацию клавиш Ctrl+Shift+C и Delphi автоматически создает заготовку:

```
1 function TForm1.sum(a, b: integer): integer;
2 begin
3
4 end;
```

И между ключевыми словами begin и end пишем то, что будет делать наша функция, а функция будет делать простейшее - сложение переменных a и b, которые мы уже записали, когда объявляли функцию.

Тело функции:

```
1 function TForm1.sum(a, b: integer): integer;
2 begin
3 sum:=a+b; //Присваиваем функции сумму переменных
4 end;
```

Если сейчас запустить программу, то ничего не произойдет, так как функция у нас нигде не вызывается, да и значение переменным a и b мы не указали.

Создаем событие OnClick на кнопке и пишем:

```
procedure TForm1.Button1Click(Sender: TObject);
var i:integer; //создаем переменную
begin
i:=sum(5,7); //ссумируем
Label1.Caption:=IntToStr(i);//выводим
end;
```

Разберем строчку, где происходит суммирование. Пишем имя функции, потом в скобках значения для переменных a и b и результат присваиваем переменной i, которую потом выводим в лабел.

Конечно я привел самый простой и понятный пример. В функцию можно записать громадную формулу, по которой вы будете вычислять конец света и что бы потом не переписывать эту формулу можно просто указывать имя функции и вводить значения переменным.

Задание на закрепление: создайте функцию, которая будет перемножать переменные a и b.

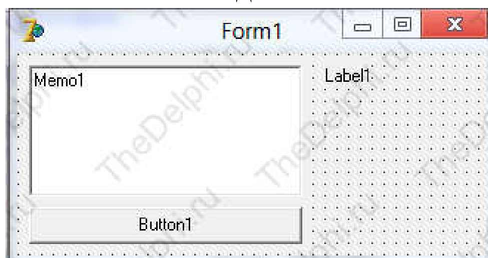
Урок 6 - Одномерные массивы

Представьте себе поезд, у которого есть определенное количество вагончиков. У каждого вагона есть номер и внутри каждого, пронумерованного вагона можно хранить информацию. Так вот, массив примерно так и выглядит, только он не поезд. Массив записывается туда же, куда мы записываем обычные переменные.

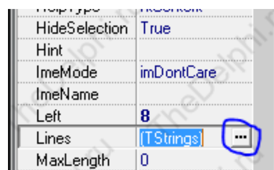
Синтаксис:

```
var имя массива: array[интервал] of тип;
```

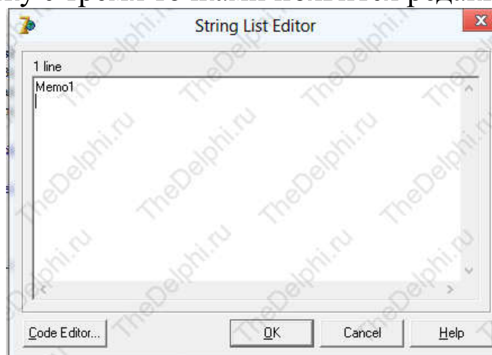
Теперь открываем Delphi и кидаем на форму компонент Button, Label и Memo. Компонент Memo находится на 6 месте на вкладке Standard.



Компонент Memo имеет свойство под названием Lines. Оно схоже со свойством Caption у компонента Label. Найдите это свойство в инспекторе объектов и нажмите на кнопочку с тремя точками:



После нажатия на кнопку с тремя точками появится редактор:



Сотрите всё, что написано в редакторе и создайте событие OnClick на кнопке. Объявим массив с именем mas, интервалом от 1 до 3 и типом string. Да, и еще нам нужно объявить переменную i типа integer, она нам пригодиться позже.

```
procedure TForm1.Button1Click(Sender: TObject);
var mas: array[1..3] of string;
i:integer;
begin
...
```

Сейчас запишем в наши воображаемые вагоны массива значения:

```
mas[1]:='Первая строка';
mas[2]:='Вторая строка';
mas[3]:='Третья строка';
```

А сейчас выводим второй элемент массива в Label и все 3 элемента в Memo.

В лабел:

```
Label1.Caption:=mas[2]; //Лабел отобразит содержимое второго элемента массива
```

Теперь, для того чтобы вывести все 3 значения в Memo, нужно воспользоваться циклом For:

```
for i:=1 to 3 do
Memo1.Lines.Add(mas[i]);
```

Так как в теле цикла выполняется всего одна строка, то можно опустить ключевые слова begin и end. Строка Memo1.Lines.Add(mas[i]); при первом круге цикла добавит в memo первый элемент массива, так как переменная i равна 1. При втором круге переменная i равна 2 и строка Memo1.Lines.Add(mas[i]); добавит второй элемент массива. Точно так же и с третьим.

Общий вид программы:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 var mas: array[1..3] of string;
03 i:integer;
04 begin
05 mas[1]:='Первая строка';
06 mas[2]:='Вторая строка';
07 mas[3]:='Третья строка';
08
09 Label1.Caption:=mas[2]; //Лабел отобразит содержимое второго элемента массива
10
11 for i:=1 to 3 do
12 Memo1.Lines.Add(mas[i]);
13 end;
```

Без цикла For вывод элементов массива осуществлялся бы таким образом:

```
1 Memo1.Lines.Add(mas[1]);
2 Memo1.Lines.Add(mas[2]);
3 Memo1.Lines.Add(mas[3]);
```

Нам придется долго копировать и изменять номер элемента, если их будет сто, а бывает и больше тысячи! Цикл же помогает справиться с этим, всего ценой двух строчек.

На этом я завершаю наш урок, но перед этим сохраните пожалуйста программу, потому что она нам понадобится в следующем уроке.

Задание на закрепление: заполните массив из 100 чисел при помощи цикла for, выведите в memo числа 20, 50, 70 из массива при помощи цикла for и условия в нём.

Урок 7 - Многомерные массивы

Многомерные массивы - это практически то же самое, что и одномерные, только они представляют из себя матрицу.

	A	B	C
1			
2			
3			

Из программы прошлого урока мы помним, что одномерный массив объявляли так:

```
1 var mas: array[1..3] of string;
```

Многомерный массив требует записи двух диапазонов:

```
1 var mas: array[1..3,1..3] of string;
```

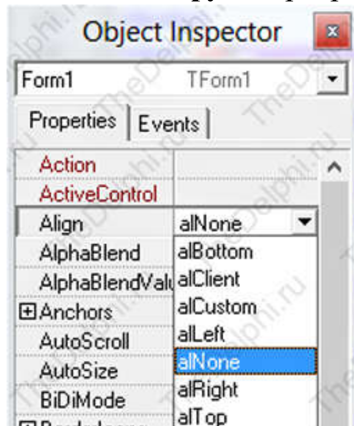
Сильно затягиваться не будем в многомерный массив так как он схож с одномерным, а про одномерный я вам уже всё рассказал.

!!! Самостоятельно заполните многомерный массив размером 10x10 произвольными данными с помощью любого цикла, используя объект StringGrid на вкладке Additional.

Урок 8 - Форма и её свойства

Пришло время погрузиться в практическую часть и в этом уроке мы рассмотрим форму и ее основные свойства. Как вы уже знаете, все свойства любого объекта в Delphi располагаются в Object Inspector.

Открываем Delphi, создаем новый проект и выделяем форму. Вкратце пробежимся по основным её свойствам. Первое свойство, которое мы рассмотрим это свойство Align. Отвечает оно за позиционирование нашей формы на мониторе. Открываем это свойство и видим несколько значений. Выбрав значение, **компилируем** программу и наблюдаем результат.



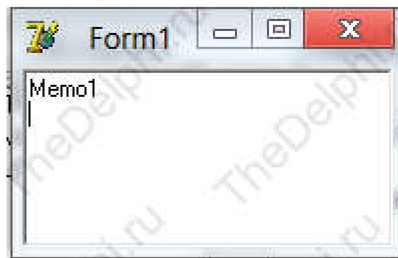
- alBottom - прилипание формы к низу экрана;
- alClient - растягивание формы на весь экран;
- alCustom - по умолчанию;
- alLeft - прилипание формы к левому краю экрана;
- alNone - без выравнивания;
- alRight - прилипание формы к правому краю экрана;
- alTop - прилипание формы к верху экрана.

Вернем значение **alNone**. Следующие свойство AlphaBlend. Это свойство отвечает за прозрачность формы. Выставляем его значение True и ниже видим свойство AlphaBlendValue. Оно может принимать значения от 1(полная прозрачность) до 255(полная не прозрачность). Выставляем значение **200** и видим прозрачную форму.

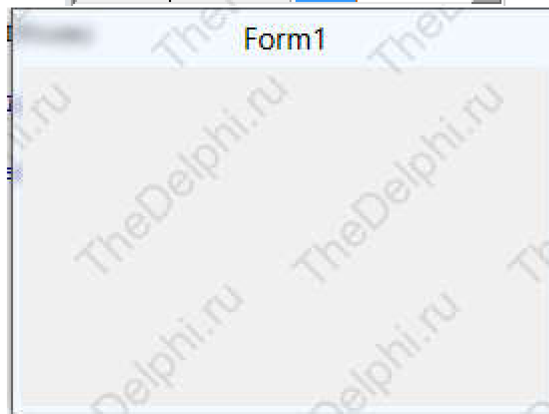
Выключаем прозрачность формы и двигаемся дальше.

Следующие свойство AutoScroll. Принимает значения Boolean. Я думаю по названию свойства понятно его назначение. Если какой-либо компонент выходит за границы формы, то на форме автоматически появляется скролл.

Следующие свойство AutoSize, оно находит и подгоняет оптимальный размер формы. Если мы сейчас выставим значение True и кинем на форму компонент Memo, то это свойство при компиляции уберет все пустые места на форме и как бы обтянет компонент Memo. Из-за этого мы не сможем изменить размер формы.



Вернем свойству AutoSize значение False и продолжим рассматривать остальные свойства. Свойство BorderIcons. В нём мы можем задать, какие кнопки мы будем видеть в шапке формы. Сейчас у нас видны кнопки "свернуть", "развернуть" и "закрыть". Если мы всем этим кнопкам выставим значение False, то шапка формы будет совершенно пустая.



Возвращаем все значения обратно и переходим к следующему свойству.

Свойство BorderStyle, оно отвечает за стиль границ нашей формы. Вы можете сами поэкспериментировать с различными значениями. Хотелось бы только выделить значение bsNone, то есть после компиляции наша форма совсем не будет иметь границ, это позволяет разрабатывать свои дизайны и скины для формы.

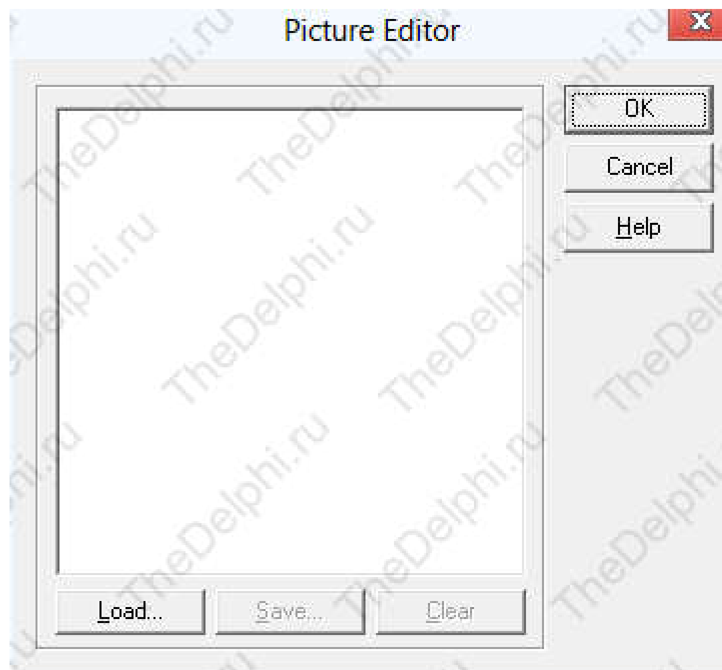
Далее рассматриваем свойство Caption. Оно есть практически у каждого компонента. Сейчас оно имеет значениеForm1, давайте изменим его на **Программа** и мы видим, что заголовок нашей формы поменялся.

Свойство Color отвечает за цвет формы. Очень простое свойство и я думаю вам будет полезнее поэкспериментировать самим.

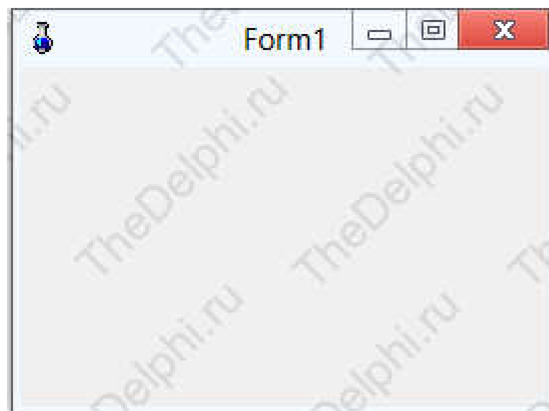
Свойство Cursor отвечает за то, какой будет курсор при наведении на форму. Выставим значение crHourGlass и скомпилируем программу. Мы видим, что появляется как бы ложный курсор, обозначающий зависание программы.

Вы наверняка видели в различных программах всплывающие подсказки, которые появляются при задержке курсора на каком-нибудь компоненте. Свойство Hint как раз отвечает за текст этой подсказки. Напишем **Форма** и скомпилируем программу. Задерживаем курсор на форме и ничего у нас не всплывает :). А всё потому, что мы не включили отображение этой подсказки. Включить его можно в свойстве ShowHint, выставив значение True. Если сейчас скомпилировать и задержать курсор на форме, то мы увидим подсказку.

Далее свойство Icon, оно отвечает за иконку в левом верхнем углу формы. Выделяем свойство Icon, нажимаем на кнопку с тремя точками и у нас открывается окно загрузки иконки.



Нажимаем на кнопку Load... и выбираем картинку с расширением .ico. Иконки от Delphi лежат в папке C:\Program Files (x86)\Common Files\Borland Shared\Images\Icons. После того, как выбрали иконку, нажимаем кнопку и после компиляции видим, что значок программы изменился.



В свойстве Left задается расстояние в пикселях от левого края экрана до левого края формы.

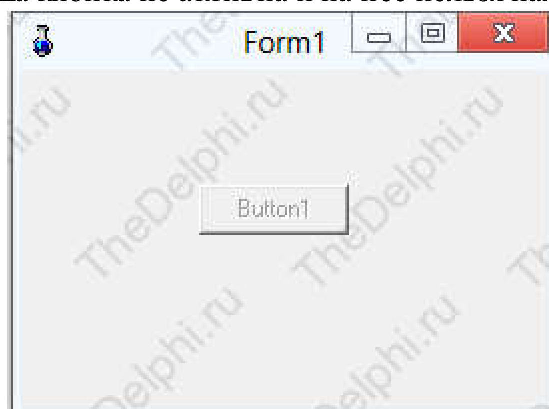
В свойстве Name пишется имя формы, по которому мы будем обращаться в окне редактора кода.

Свойство Position отвечает за позицию формы после компиляции. Поэкспериментируйте со значениями.

Далее свойство Top, оно аналогично свойству Left, только отсчет пикселей идет от верхнего края экрана и до верхнего края формы.

Свойства Width и Height отвечают за ширину и высоту формы в пикселях.

Свойство Enabled отвечает за активность. Давайте сейчас кинем на форму компонент Button и изменим у него свойство Enabled, выставив значение False. Компилируем программу и видим, что наша кнопка не активна и на нее нельзя нажать.

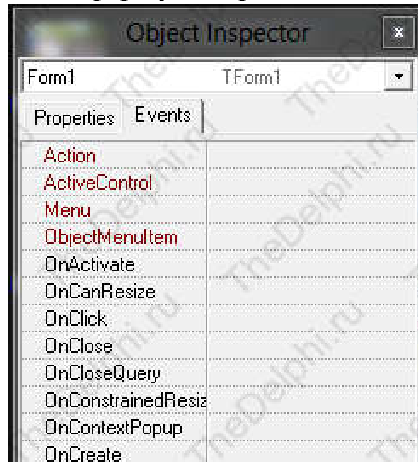


Ну и последнее свойство, которое мы разберем в этом уроке, свойство Visible. Это свойство отвечает за видимость компонента. Измените это свойство у кнопки, выставив значение False и скомпилируйте программу, кнопки вы не найдете.

На последок хочу отметить, что большинство свойств у компонентов одинаковое. В этом уроке я рассказал не про все свойства, а только самые основные. В дальнейших уроках, если мы будем сталкиваться с новым свойством, то я обязательно про него расскажу.

Урок 9 - События. Программное изменение свойств

В этом уроке я расскажу, что такое событие, реакция на событие и мы научимся программно изменять свойство компонентов. Создаем новый проект, на форму кидаем компонент Button с закладки Standard. Все события выделенного компонента находятся в Object Inspector на вкладке Events. Выделяем форму и переходим к списку ее событий.



Список довольно большой и по этому мы рассмотрим самые важные события.

Событие OnClick. Вы уже знакомы с этим событием из прошлых уроков и знаете, что оно происходит тогда, когда мы нажимаем на ЛКМ один раз.

Далее событие OnCreate, оно возникает тогда, когда форма только начинает создаваться. Формы еще нет на экране, а событие уже произошло.

Событие OnDblClick, это событие похоже на OnClick, только нажать на ЛКМ нужно 2 раза (двойной клик).

Событие OnKeyDown. По названию можно понять, что оно происходит, когда пользователь нажимает на клавишу клавиатуры.

Событие OnKeyUp, схоже с событием OnKeyDown, только в этом случае клавиша отпускается.

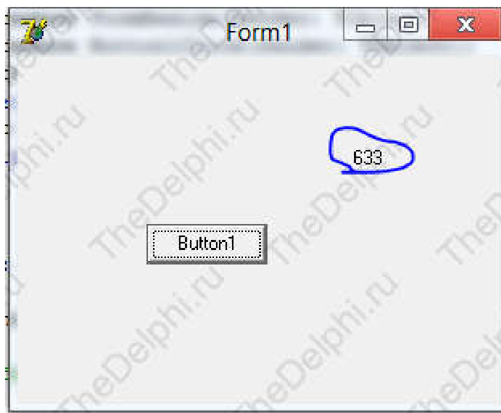
Событие OnMouseDown, оно происходит при нажатии на левую кнопку мыши, на правую и на колёсико.

Событие OnMouseUp тоже самое что и OnMouseDown, только в этом случае кнопка отпускается.

Следующие событие OnMouseMove, оно происходит тогда, когда мы двигаем мышку в зоне данного компонента. Давайте попробуем поработать с этим свойством. Для этого нам нужно добавить компонент Label с вкладки Standard на форму и изменить его свойство Caption на 0. Теперь выделяем компонент Button и находим там событие OnMouseMove, щелкаем по нему 2 раза и Delphi создаст процедуру Button1MouseMove. В этой процедуре между ключевыми словами Begin и end пишем код:

```
1 label1.Caption:=IntToStr(StrToInt(Label1.Caption)+1);
```

Объясняю что будет происходить. Когда мы водим мышкой по кнопке, выполняется событие OnMouseMove. В этом событии мы присваиваем свойству Caption у лейбла значение на единицу больше чем было. Первоначально у нас в лейбле стоит ноль, когда мы водим по кнопке, то прибавляется к нулю единица, потом дальше шевелим мышку и уже к единице прибавляется еще одна единица и так далее. Вы, наверное заметили, что в коде присутствуют функции IntToStr и ей обратная StrToInt они нужны для того, что бы складывать значения в числовом формате.



Если бы мы их не использовали и прибавляли бы единицу в строковом виде, то получилась бы длинная цепочка из единиц.

Идем далее и следующие свойство формы, которое мы рассмотрим называется OnResize оно возникает при изменении размера формы. Кликаем по этому событию и заполняем процедуру всё тем же кодом:

```
1 label1.Caption:=IntToStr(StrToInt(Label1.Caption)+1);
```

Теперь компилируем программу и наводим мышь на уголок формы. Теперь изменяем размер и видим, что событие происходит и выполняется код, который мы там написали.

С рассмотрением основных событий мы закончили, теперь давайте научимся изменять свойства компонентов программно. В событии кнопки OnClick пишем код:

```
1 form1.caption:='Programm';
```

Этот код изменит заголовок формы после нажатия на кнопку. Как вы видите, сначала пишется имя компонента (в данном случае форма), далее через точку его свойство и через оператор присваивания пишется сам заголовок в кавычках, так как свойство Caption имеет строковый тип.

Далее давайте программно изменим ширину формы, кликом по кнопке:

```
1 procedure TForm1.Button1Click(Sender: TObject);
2 begin
3 form1.width:=500;
4 end;
```

Я думаю вы поняли, как программно изменять свойства компонентов. Экспериментируйте и всё получится.

Здравствуйте, дорогие друзья! В этом, десятом уроке, мы начинаем подробное знакомство с компонентами. Всего будет 12 уроков на эту тему. В этом уроке мы будем знакомиться с компонентами на вкладке Standard.

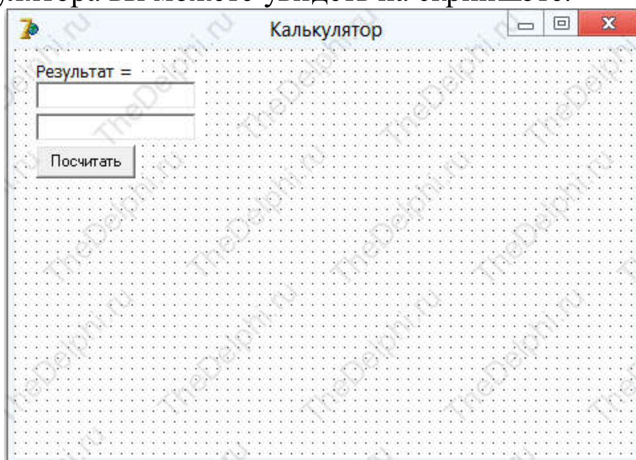
Открываем Delphi, и кидаем на форму компонент Edit. У этого компонента есть свойство Text. Всё, что написано в этом свойстве, отображается в самом Edit'e. Очистим свойство Text.

Следующий компонент, который оказывается у нас на форме прямо над Edit'ом, называется Label. Этот компонент является подписью для других компонентов. Так же в него удобно выводить какую-нибудь динамическую информацию. Напишем в свойство Caption значение **Результат =** .

Еще один компонент, который нам нужен - кнопка, то есть Button. Располагаем его под Edit'ом.

Теперь давайте напишем простенькую программу, которая будет работать с этими компонентами. Программа называется "Калькулятор". Вы можете назвать заголовок формы соответствующе. Нашему калькулятору не хватает еще одного компонента Edit, пожалуйста разместите его ниже первого Edit'a.

Общий вид калькулятора вы можете увидеть на скриншоте.



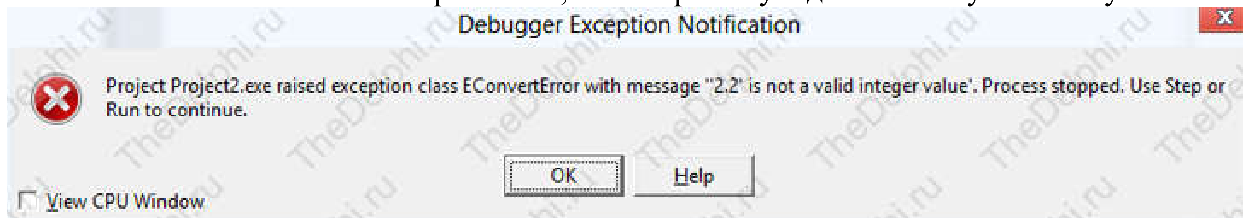
Теперь, когда мы закончили с настройкой компонентов, погружаемся в код программы и создаем обработчик события Button1Click. В нем пишем код, который будет складывать числа первого эдита и второго.

```
1 Label1.Caption:='Результат = '+IntToStr(StrToInt(Edit1.Text)+StrToInt(Edit2.Text));
```

Я надеюсь вам не нужно объяснять, что такое IntToStr и с чем его едят.

После компиляции, мы вводим в первый эдит число 10, а во второй 5 и после нажатия на кнопку видим, что в лейбле отображается Результат = 15. Это значит, что программа работает и правильно складывает числа, а если работает, то требует совершенства.

Попробуйте сложить в нашей программе дробные числа. Ничего не выйдет, так как значение из эдитов мы преобразовываем в числовой тип Integer, а он работает только с целыми числами. Если же вы все таки попробовали, то наверняка увидели похожую ошибку:



Она как раз и сообщает о том, что тип Integer не может работать с дробными числами.

Но как же быть?! Что делать? А вот тот, кто хорошо читает уроки, знает, что с целыми и дробными числами работает тип Real. Что бы преобразовать значение из эдита в тип Real, нужно изменить операторы IntToStr и StrToInt на FloatToStr и StrToFloat соответственно.

В итоге мы получим вот такой код:

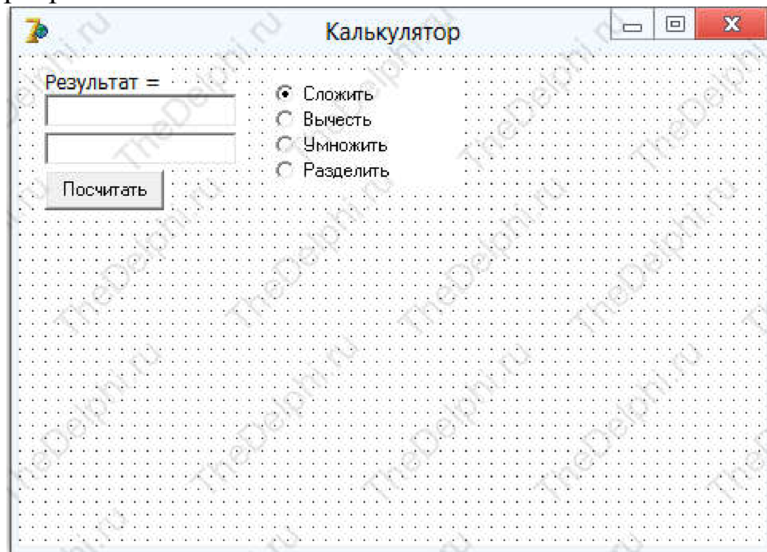
```
1 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)+StrToFloat(Edit2.Text));
```

Далее учим программу вычитать, умножать и делить. Для этого поместим на форму компонент RadioButton в количестве четырех штук и изменим у них свойство Caption на

Сложить, Вычесть, Умножить, Разделить.

Компонент RadioButton вы могли видеть на сайтах или в других программах, он представляет из себя кружочек (темный или светлый, в зависимости от выбора) и рядом подпись к этому кружочку. У этого компонента есть свойство Checked. Оно может принимать значения True или False, что определяет выбор радиокнопки. Поставьте в свойстве Checked значение True у радиокнопки с именем Сложить.

Общий вид программы:



Теперь нам нужно изменить код, в событии Button1Click. Работать он будет так: если выбрана радиокнопка 'сложить' - складываем, если выбрана радиокнопка 'вычесть' - вычитаем. И дальше по аналогии.

Код:

```
01 procedure TForm1.Button1Click(Sender: TObject);
02 begin
03 if (RadioButton1.Checked = True) Then //если выбрано 'сложить'
04 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)+StrToFloat(Edit2.Text)); //
складываем
05
06 if (RadioButton2.Checked = True) Then //если выбрано 'вычесть'
07 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)-StrToFloat(Edit2.Text)); //
вычитаем
08
09 if (RadioButton3.Checked = True) Then //если выбрано 'умножить'
10 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)*StrToFloat(Edit2.Text)); //
умножаем
11
12 if (RadioButton4.Checked = True) Then //если выбрано 'разделить'
13 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)/StrToFloat(Edit2.Text)); //
делим
14 end;
```

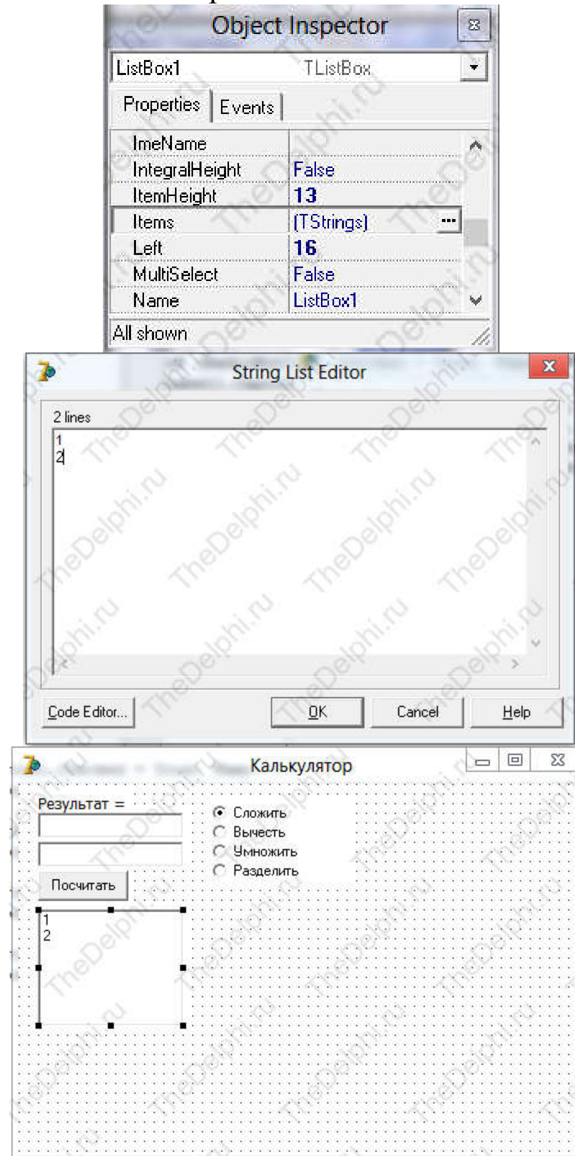
Компилируем и пробуем! У меня работает ;)

В следующем уроке мы будем модернизировать калькулятор, поэтому, пожалуйста, сохраните проект в надежное место.

Ну вот и всё!

Продолжаем знакомство с компонентами. Открываем наш предыдущий проект и начинаем модернизировать калькулятор, как я и обещал.

Кидаем на форму компонент ListBox. Он находится на вкладке Standard правее компонента RadioButton. У компонента ListBox есть свойство Items, если его открыть, то появляется окошко для редактирования строк. Например если написать 1 в первой строке и 2 во второй, то в ListBox сразу появятся эти строки.



Теперь откройте свойство Items еще раз и сотрите всё, что написали в окошке редактирования строк.

После краткого знакомства с компонентом ListBox, нужно его задействовать в нашем калькуляторе. ListBox будет выполнять функцию лога (журнала), он будет записывать результат в новую строку каждый раз, когда будет нажата кнопка "Посчитать".

Чтобы это осуществить, переходим в процедуру Button1Click и пишем код в самом конце:

```
1 ListBox1.Items.Add(Label1.Caption);
```

То есть, когда пользователь нажмет на кнопку, после всех вычислений добавляется новая строка с содержимым лейбла в самый конец компонента ListBox.

Теперь добавим возможность пользователю отключить ведение лога. Для этого добавьте на форму компонент CheckBox. Он находится на вкладке Standard левее компонента RadioButton. CheckBox схож с компонентом RadioButton, только при выделении ставится галочка, а не кружок. Если мы поместим на форме четыре компонента CheckBox, то поставить галочку можно на всех, что нельзя сделать с RadioButton. В этом и есть вся разница этих компонентов.

Изменим свойство Caption у CheckBox на **Вести лог** и теперь научим его управлять

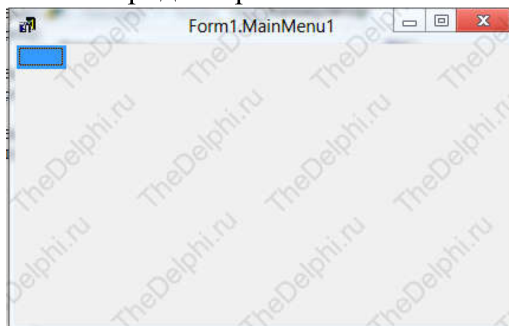
логом. Для этого нужно написать условие: если галочка стоит, то записывать результат в лог, если галочка не стоит, то соответственно не записывать.

```
if (CheckBox1.Checked = True) Then
ListBox1.Items.Add(Label1.Caption);
```

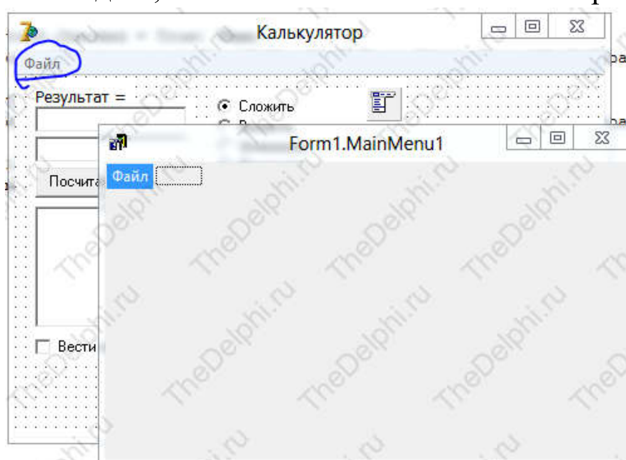
Вот и готово, скомпилируйте и проверьте!

Нашей программе еще не хватает главного меню, которое вы можете наблюдать во многих программах и даже в самом Delphi.

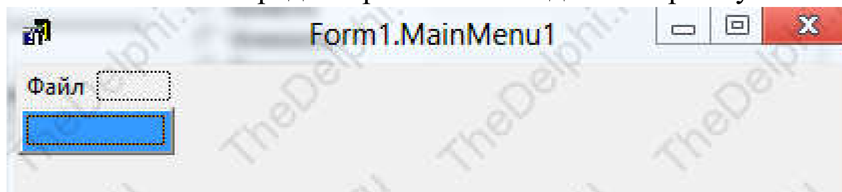
Кидаем на форму компонент MainMenu (он находится на вкладке Standard) и кликаем по нему 2 раза. Должно появится окошко редактирования меню.



Теперь переходим в Object Inspector, меняем свойство Caption на **Файл** и нажимаем клавишу Enter. Мы сразу можем видеть, как появилось меню в нашей программе.



Теперь нажимаем на Файл в редакторе меню и выделяем прямоугольник ниже.



Сейчас опять переходим в свойство Caption, изменяем его на **Посчитать** и кликаем Enter. Снова кликаем на прямоугольник ниже и переименовываем его на **Выход**. Далее кликаем на прямоугольник правее от Файл и изменяем у него Caption на **Помощь**. Теперь нажимаем на Помощь в редакторе меню, выделяем прямоугольник ниже и меняем свойство Caption на **О программе**.

Таким образом, мы создали простейшее меню, но оно пока не работает.

В редакторе кода выделяем кнопку (тот же прямоугольник) **Посчитать** и переходим в Object Inspector на вкладку Events. Выделяем событие OnClick и теперь у нас есть 2 пути:

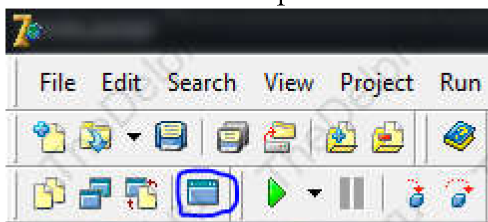
1. Создать процедуру N2Click (N2 - это имя кнопки в меню) и скопировать в нее весь код, который мы написали в процедуре Button1Click;

2. Поступить по умному и ничего не копировать, а просто указать Delphi, что нужно выполнить процедуру Button1Click по клику на кнопке в меню N2.

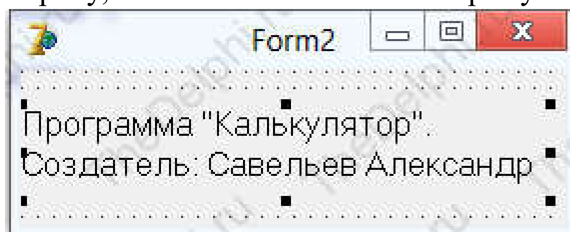
Я выбираю второй вариант, думаю вы тоже. Нажимаем на стрелочку и выбираем из выпадающего списка процедуру Button1Click.

Компилируем и проверяем работу меню. У меня всё отлично, все работает и компилируется без ошибок.

Кнопку Выход (в меню) я думаю вы "оживите" сами. По этому сразу переходим к кнопке **О программе**. Когда мы нажмем на эту кнопку, у нас вылезет окошко с информацией о программе и его авторе. Для этого создадим вторую форму. Создается она очень легко, нажатием на кнопку New Form в главном окне Delphi.

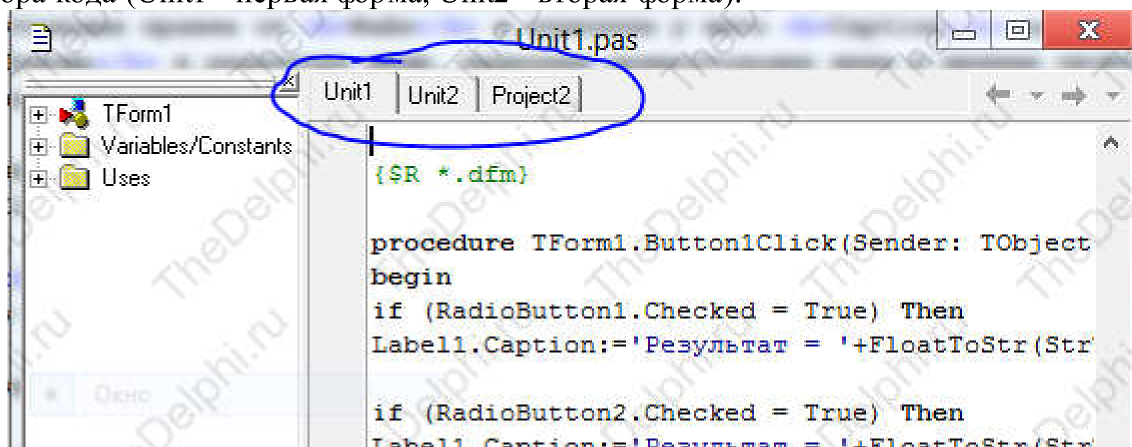


Далее кидаем на вторую форму компонент Label и пишем в него информацию о программе. Кстати, у лейбла есть очень хорошее свойство WordWrap, которое позволяет переносить слова на новую строку, если они не влезают в ширину.



После того, как мы оформили вторую форму, ее надо показать пользователю, когда он нажмет на кнопку **О программе** в главном меню.

Переходим к первой форме, нажатием на закладку соответствующего юнита в окне редактора кода (Unit1 - первая форма, Unit2 - вторая форма).



Открыть вторую форму можно двумя способами:

Первый способ - это когда можно спокойно перемещаться между формами, то есть, если мы открыли вторую форму, то мы можем переключится на первую и поработать в ней.

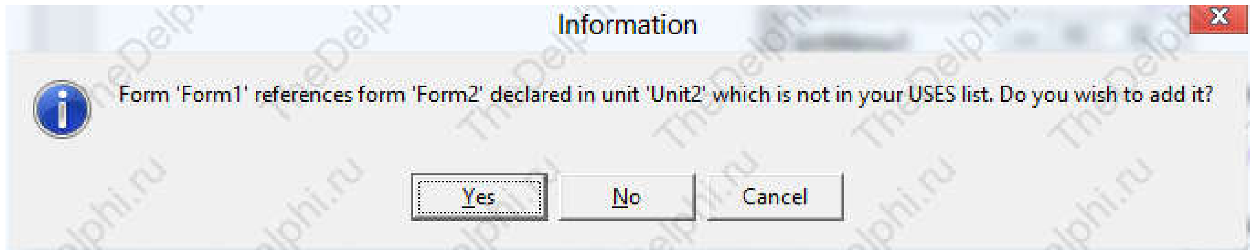
```
1 Form2.Show;
```

Второй способ - это когда мы не можем переключится на первую форму, если открыта вторая.

```
1 Form2.ShowModal;
```

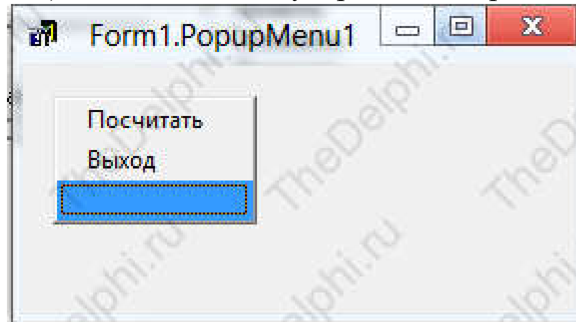
Создаем у кнопки **О программе** событие OnClick и пишем туда код, в зависимости от того, что вы выбрали. Если вы сейчас скомпилируете программу, то при компиляции откроется

окно:



Это окно предлагает добавить вторую форму в юнит первой. Нажмите кнопку **"Yes".** Добавить можно было вручную, дописав в Uses слово Unit2, но зачем, если delphi это может сделать за нас? :)

Идем далее и теперь мы сделаем контекстное меню, оно будет появляться, когда мы нажмем на форму правой кнопкой мыши. Кидаем на форму компонент PopupMenu (он находится на вкладке Standard), кликаем по нему 2 раза и настраиваем его вот так:



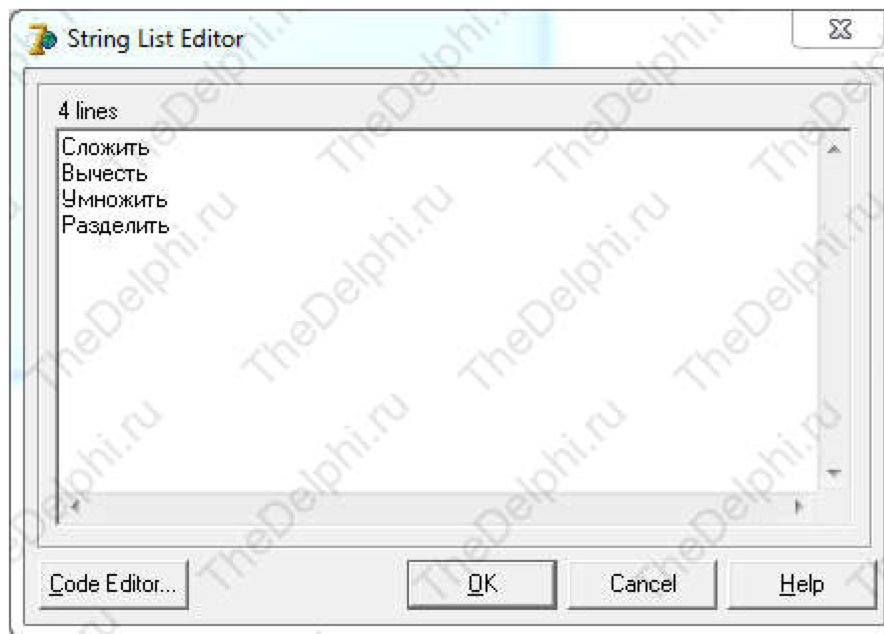
Теперь выделяем первую форму и в свойстве PopupMenu выбираем наш компонент PopupMenu1. Сейчас осталось "оживить" кнопки в контекстном меню. Я надеюсь, что вы с этим справитесь сами. На этом я завершаю этот урок и прошу вас сохранить проект.

Ну вот и всё!

Урок 12 - Знакомство с компонентами (часть 3/12)

В двенадцатом уроке мы продолжаем писать свой калькулятор и параллельно знакомиться с компонентами. Я прошу вас открыть проект из предыдущего урока, чтобы мы продолжили.

И так, начнем с компонента под названием ComboBox, он находится на вкладке Standard. Компонент ComboBox представляет из себя выпадающий список. Давайте сделаем так, чтобы при нажатии на этот компонент у нас выпадал список с функциями сложения, вычитания, умножения и деления. Для этого обратимся к свойству Items и запишем в каждую строку по функции.



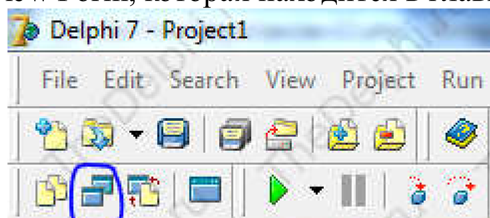
Если сейчас скомпилировать программу, то по нажатию на ComboBox выпадет список с нашими функциями, но пока что они не работают. У компонента ComboBox есть свойство ItemIndex. Оно определяет выбранную строку. Нумерация строк начинается с нуля, то есть сложить - нулевая строка, вычесть - первая строка и т. д. Если в этом свойстве стоит значение -1, то значит ни одна строка не выбрана и в ComboBox записывается значение из свойства Text. Присвоим свойству ItemIndex значение 0. Кинем на форму компонент Button и в событии OnClick пишем:

```
01 If ComboBox1.ItemIndex = 0 then
02 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)+StrToFloat(Edit2.Text)); //
Складываем
03
04 If ComboBox1.ItemIndex = 1 then
05 Label1.Caption:='?Результат = '+FloatToStr(StrToFloat(Edit1.Text)-StrToFloat(Edit2.Text)); //
Вычитаем
06
07 If ComboBox1.ItemIndex = 2 then
08 Label1.Caption:='?Результат = '+FloatToStr(StrToFloat(Edit1.Text)*StrToFloat(Edit2.Text)); //
Умножаем
09
10 If ComboBox1.ItemIndex = 3 then
11 Label1.Caption:='?Результат = '+FloatToStr(StrToFloat(Edit1.Text)/StrToFloat(Edit2.Text)); //
Делим
```

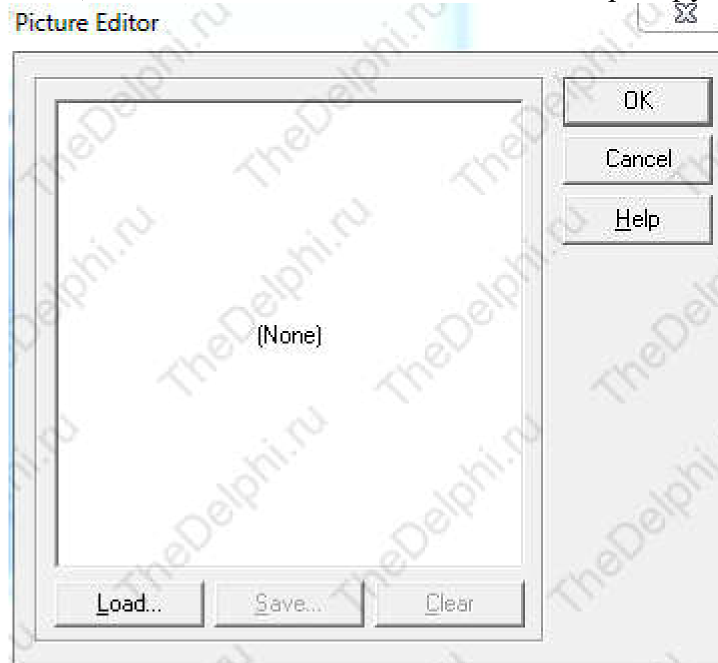
То есть сначала в условии проверяется какая строка в выпадающем списке выбрана, а потом в зависимости от этого мы либо складываем, либо вычитаем...

Наконец-то мы закончили разбирать основные компонента с вкладки Standard. Теперь мы можем перейти на вкладку Additional. Знакомиться с её компонентами в этом уроке мы будем на примере нашей второй формы.

Нажимаем на кнопку View Form, которая находится в главном окне Delphi.

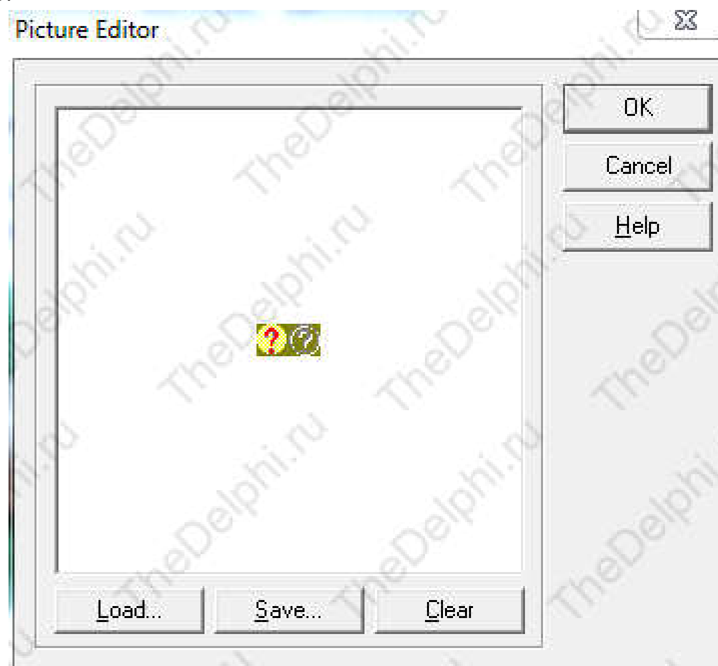


Выбираем Form2 и кидаем на нее компонент Image с вкладки Additional. В этот компонент можно загружать изображения, тем самым создавать свои дизайны для программ. У Image есть свойство Picture, нажимая на него появляется менеджер загрузки изображения.



Нажимаем на кнопку Load... и загружаем любую картинку. Я загрузил картинку HANDSHAK.BMP, которая находится в папке C:\Program Files (x86)\Common Files\Borland Shared\Images\Splash\256Color. Эта папка есть у всех, у кого установлен Delphi. У компонента Image есть свойство AutoSize, если выставить его значение в True, то размеры компонента автоматически будут подгоняться под размеры картинки. Так же есть свойство Center, если его выставить в True, то картинка будет располагаться по центру компонента, не в зависимости от его размеров. У Image тоже есть события, они схожи с событиями кнопки.

Переходим к следующему компоненту с вкладки Additional, который называется BitBtn. Этот компонент похож на обычную кнопку, но он гораздо усовершенствование. В BitBtn можно загружать картинку, которая будет располагаться рядом с текстом. Нажмем на свойство Glyph и выберем любую картинку из папки C:\Program Files (x86)\Common Files\Borland Shared\Images\Buttons.



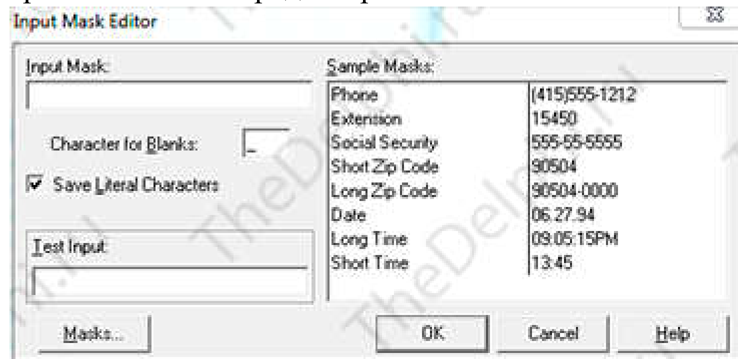
Как вы видите, картинка как бы двойная: в левой части картинка в цвете, а в правой нет. Если кнопка активна (свойство Enabled), то используется левая часть картинки, если не активна, то соответственно правая.

На этом я заканчиваю двенадцатый урок, если вы потеряли исходник, то вы можете найти его в папке Source.

Ну вот и всё!

Давайте же продолжим изучение компонентов Delphi со вкладки Additional.

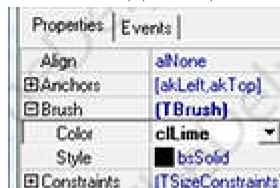
Компонент MaskEdit представляет собой обычный Edit, но у него есть особенность вводить текст по маске. Для этого существует свойство EditMask. После нажатия на это свойство появляется простенькое окно редактирования маски.



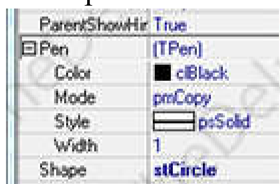
С правой стороны окна находится список, в который записаны маски по умолчанию. Выбираем например маску Short Time, нажимаем кнопку "ОК" и компилируем проект. Теперь в наше поле MaskEdit можно вводить только время, состоящие из четырех цифр. Думаю с компонентом трудностей не возникнет и по этому переходим к следующему компоненту, который называется Shape.

Компонент Shape предназначен для отображения простых геометрических фигур на форме. Кидаем этот компонент на форму и переходим к свойству Shape. Из выпадающего списка мы можем выбрать любую из представленных геометрических фигур.

Так же у компонента Shape есть свойство Brush. Открываем его и видим еще одно свойство под названием Color. Это свойство задает цвет закраски фигуры.



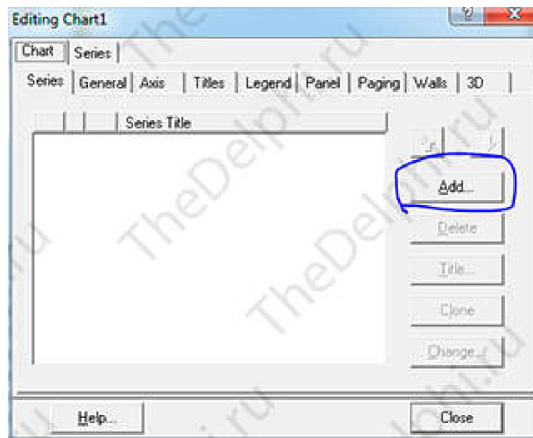
Далее рассмотрим свойство Pen. Открываем его и видим еще несколько свойств.



Свойство Color отвечает за цвет границы нашей фигуры.

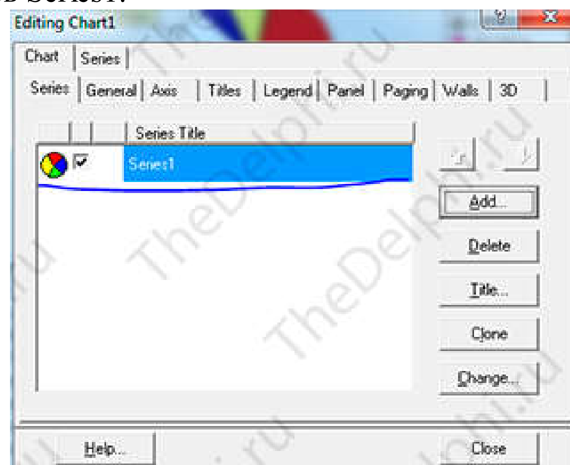
Свойство Style отвечает за стиль границы, то есть можно выбрать пунктир, частый пунктир и т. д. В принципе этот компонент тоже из себя ничего сложного не представляет.

Теперь настало время познакомиться с более мощным компонентом, который называется Chart. Он необходим для построения различных графиков, круговых диаграмм на нашей форме. Переносим этот компонент на форму и кликаем на него 2 раза. Появляется окно редактора диаграмм. Нажимаем на кнопку "Add...".



Нам открылось окно, в котором мы видим список возможных диаграмм.

Выбираем круговую диаграмму и нажимаем кнопку "OK". Имя нашей диаграммы автоматически присвоилось Series1.



Не обращайте внимания на то, что на компоненте сразу выстроилась диаграмма с разными значениями. При компиляции мы увидим пустую область на этом компоненте. Теперь научимся создавать диаграммы. Кликаем по нашей кнопке "Посчитать" и приступаем к изменению кода в обработчике события Button1Click. Усовершенствуем конструкцию if then, добавляя ключевые слова begin и end. Далее пишем код после строчки, где присваиваем лейблу результат:

```
1 Series1.AddPie(strtoint(label1.Caption),'+Сложить',clred);
```

Вообще что будет делать наша диаграмма? После того, как мы будем с числами выполнять какие-то математические операции, у нас будет на круговой диаграмме показываться доля этих операций.

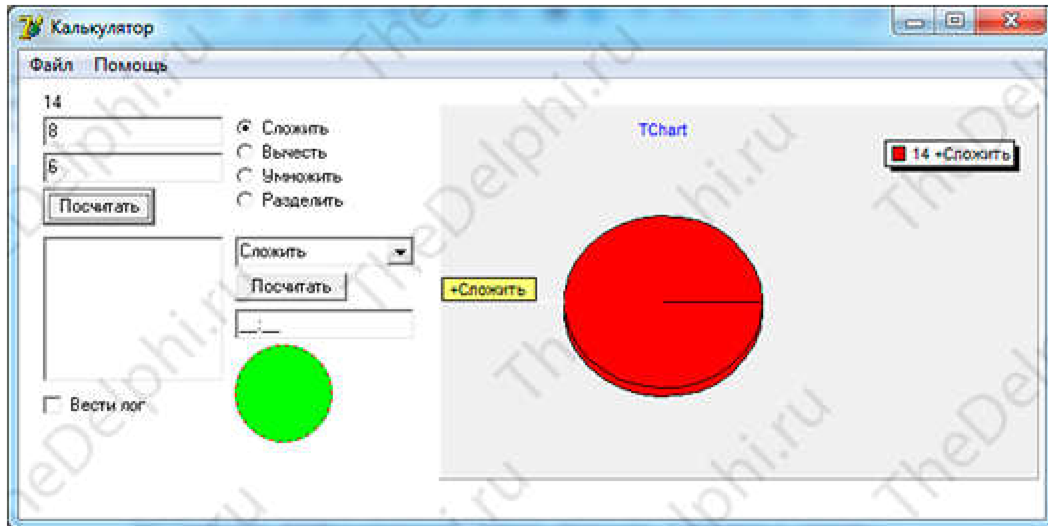
Функция AddPie имеет 3 параметра. Первый это числовое значение, которая долька будет занимать от всей диаграммы. Второй параметр отвечает за подпись, а третий за цвет дольки. В первый параметр нужно занести число, а лейблу у нас присваивается значение "Результат = и тут какое-то число". Дак вот, нам нужно избавиться от "Результат =" и оставить только присваиваемое число. Для этого мы упростим строку:

```
1 Label1.Caption:='Результат = '+FloatToStr(StrToFloat(Edit1.Text)+StrToFloat(Edit2.Text));
```

Вот до такого вида:

```
Label1.Caption:=FloatToStr(StrToFloat(Edit1.Text)+StrToFloat(Edit2.Text));
```

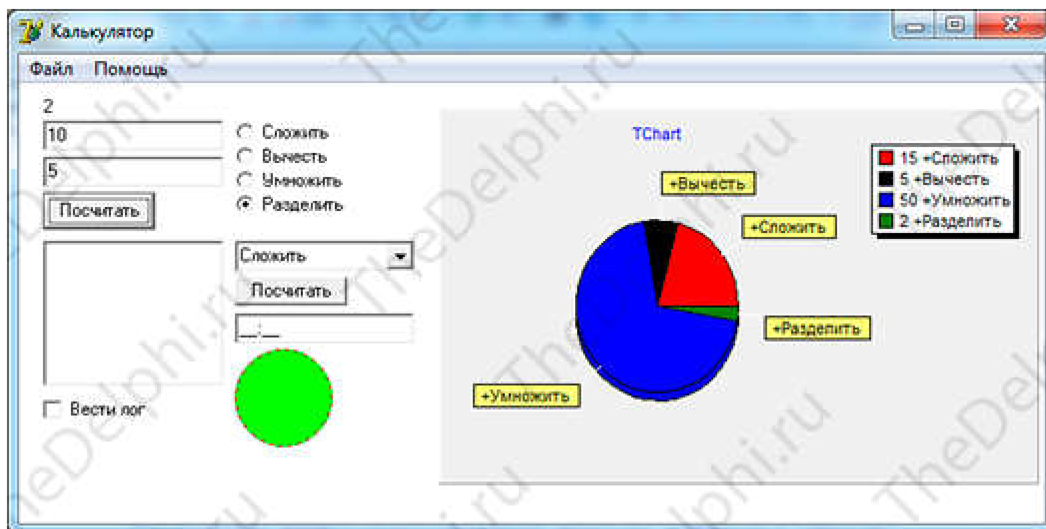
Теперь скомпилируем и сложим два числа. У нас появится долька на диаграмме, с результатом сложения.



Теперь давайте изменим код для вычитания, деления и умножения. В итоге код обработчика события Button1Click будет таковым:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if (RadioButton1.Checked = True) Then
begin
Label1.Caption:=FloatToStr(StrToFloat(Edit1.Text)+StrToFloat(Edit2.Text));
series1.AddPie(strtoint(label1.Caption),'+Сложить',clred);
end;

if (RadioButton2.Checked = True) Then
begin
Label1.Caption:=FloatToStr(StrToFloat(Edit1.Text)-StrToFloat(Edit2.Text));
series1.AddPie(strtoint(label1.Caption),'+Вычесть',clblack);
end;

if (RadioButton3.Checked = True) Then
begin
Label1.Caption:=FloatToStr(StrToFloat(Edit1.Text)*StrToFloat(Edit2.Text));
series1.AddPie(strtoint(label1.Caption),'+Умножить',clblue);
end;

if (RadioButton4.Checked = True) Then
begin
Label1.Caption:=FloatToStr(StrToFloat(Edit1.Text)/StrToFloat(Edit2.Text));
series1.AddPie(strtoint(label1.Caption),'+Разделить',clgreen);
end;

if (CheckBox1.Checked = True) Then
ListBox1.Items.Add(Label1.Caption);
end;
```

Результат выполнения такой программы вы можете видеть на скриншоте



Таким образом, мы научились создавать простейшие диаграммы в таком мощном компоненте. В следующем уроке мы продолжим знакомство с компонентами.

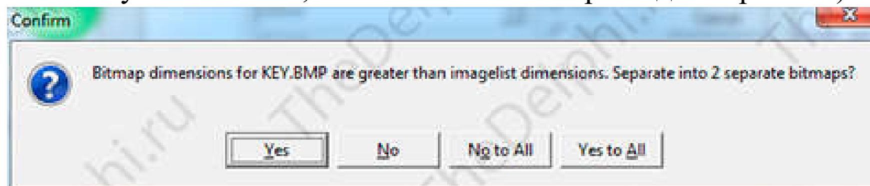
Ну вот и всё!

Урок 14 - Знакомство с компонентами (часть 5/12)

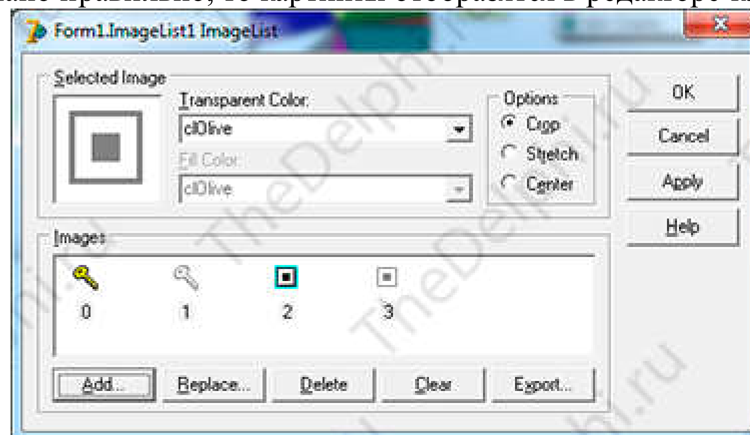
В этом уроке мы переходим на вкладку Win32! Первый компонент, который мы будем рассматривать на этой вкладке называется ImageList или как говорят в народе "хранилище картинок".

Кидаем данный компонент на форму и щелкаем по нему 2 раза. Нам открывается окно редактора картинок. Здесь жмем на кнопку "Add..." и выбираем понравившуюся картинку.

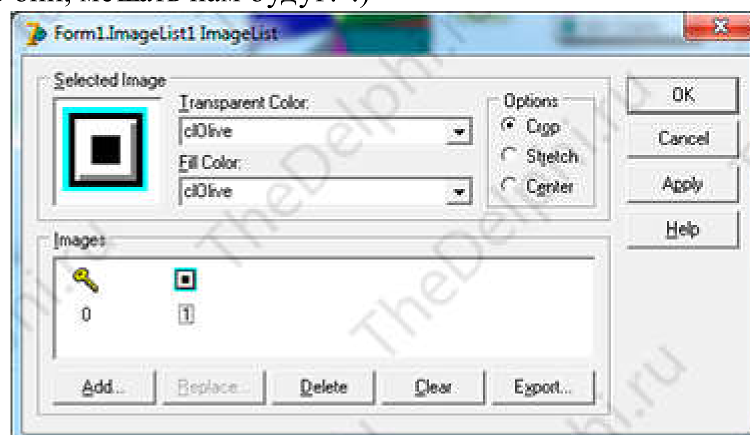
Я выбрал две картинки из папки, которая устанавливается вместе с Delphi и находится вот по такому адресу: C:\Program Files (x86)\Common Files\Borland Shared\Images\Buttons. И так, выбираем картинки и нажимаем кнопку "Открыть". Далее появится окно, в котором Delphi спросит: "Разделить ли картинку на 2 части, то есть на левую половинку и правую)". Нажимаем кнопку "Yes" (или кнопку "Yes to All", если вы как и я выбрали две картинки).



Если все сделано правильно, то картинки отобразятся в редакторе картинок.

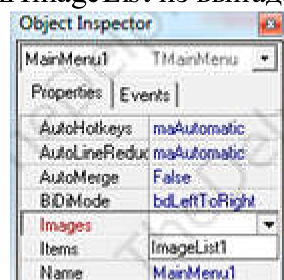


Заметьте, что каждая картинка получила свой индекс, он отображается снизу картинки. Теперь удалим "бледные" картинки (на скриншоте под индексами 1 и 3), выделив их и нажав кнопку "Delete". Что они, мешать нам будут? :)

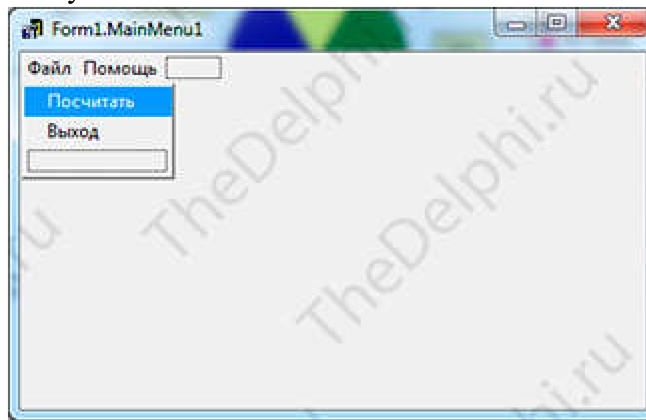


Закрываем редактор картинок, нажимая на кнопку "OK", чтобы все изменения сохранились.

Далее выделяем на форме знакомый нам компонент MainMenu. У него есть свойство Images, в котором нужно выбрать наш ImageList из выпадающего списка.



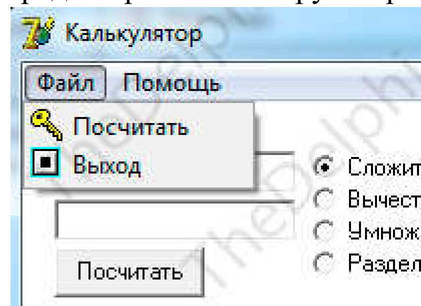
Теперь кликнем два раза по компоненту MainMenu. Откроется окно редактирования меню, в котором выбираем пункт "Файл -> Посчитать".



В инспекторе объектов ищем свойство ImageIndex.

Caption	Посчитать
Checked	False
Default	False
Enabled	True
GroupIndex	0
HelpContext	0
Hint	
ImageIndex	-1
Name	N2

ImageIndex это индекс картинки в нашем "хранилище" (ImageList), по умолчанию оно имеет значение "-1", то есть картинка не выбрана. Выбираем любую картинку из "хранилища", изменяя индекс например на "0" (ноль), а пункту "Файл -> Выход" поменяем индекс на "1" (единица). Всё готово, закрываем редактор и компилируем программу. Наблюдаем результат.

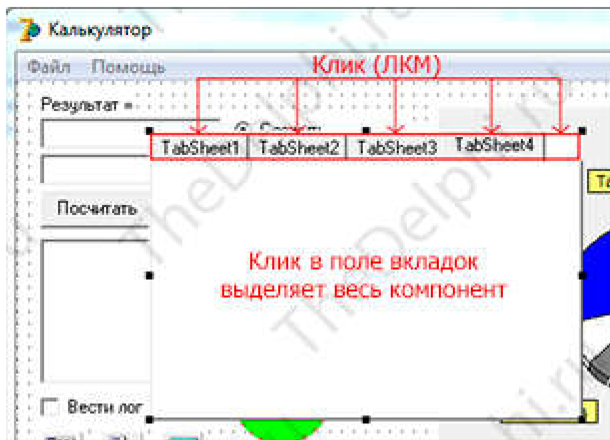


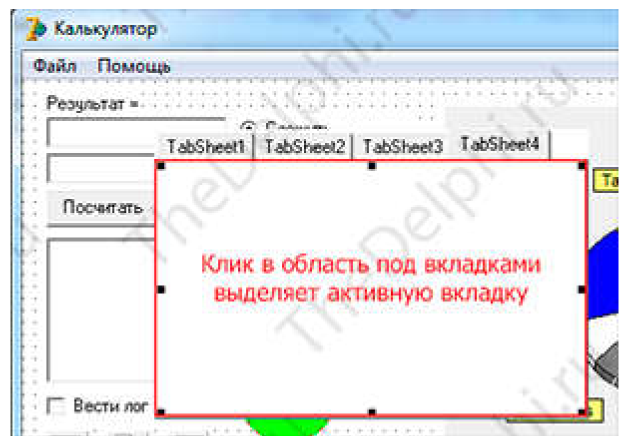
По-моему, всё так, как мы и хотели! Кстати, проделать ту же работу можно и с компонентом PopupMenu.

Переходим к изучению следующего компонента, который откликается на имя PageControl. Он позволяет создавать вкладки, на подобие тех, в которых расположены компоненты в Delphi.

Поместим компонент PageControl на форму, кликнем по нему правой кнопкой мыши и выберем пункт "New Page", чтобы создать новую вкладку. Таким же образом создайте еще три вкладки, чтобы в итоге у вас получилось четыре вкладки.

Сразу объясню один момент, чтобы не было ошибок и недопонимания. Компонент PageControl имеет два выделения: Выделение всего компонента и выделение отдельной вкладки. Если вы выделили весь компонент, то черные квадратики, обозначающие выделение будут располагаться над выбором вкладок. Если же вы выделили отдельную вкладку, то черные квадратики будут находиться под выбором вкладок. Не знаю поняли вы меня или нет, поэтому покажу на скриншоте.





И так, выделите первую вкладку. В инспекторе объектов измените свойство Caption на "Калькулятор". Теперь это же свойство, только у второй вкладки заменяем на "История". Свойство Caption третьей вкладки заменяем на "Диаграмма". Про название четвертой вкладки мы поговорим позже.

У компонента PageControl есть еще такое интересное свойство MultiLine. Если оно принимает значение True, то вкладки, при недостаточном месте, располагаются в несколько строк.

Свойство TabPosition определяет позицию отображения вкладок.

Кстати, компонент PageControl тоже имеет свойство Images, в котором вы можете выбрать наше "хранилище картинок" и назначить вкладкам определенные картинки по индексу. Ну вот и всё, компонент готов!

Приступим к использованию этого компонента по назначению, а именно к экономию места в программе.

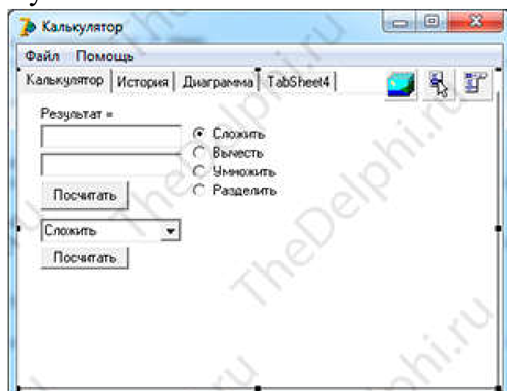
Если сейчас попробовать перетащить Edit1 на первую вкладку компонента PageControl, то наш Edit1 будет находиться как бы под компонентом PageControl. Что бы перенести компоненты, которые находились на форме раньше, чем PageControl, нужно выделить эти компоненты, вырезать их и вставить в PageControl. И так, выделяем мышкой:

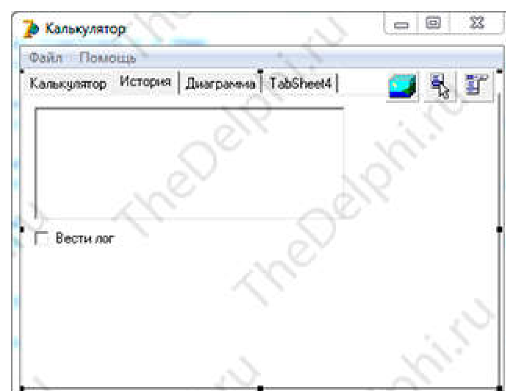
- Label1
- Edit1
- Edit2
- RadioButton1
- RadioButton2
- RadioButton3
- RadioButton4
- Button1
- Button2
- ComboBox1

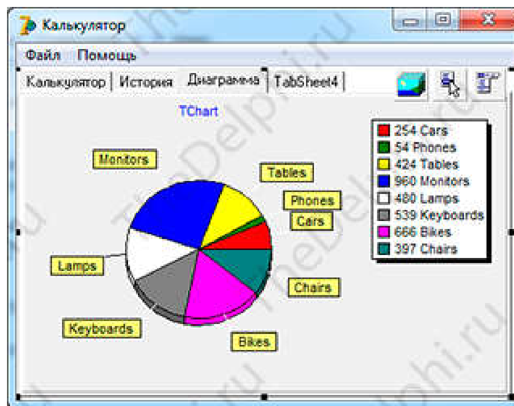
Нажимаем правой кнопкой мыши по одному из выделенных компонентов и выбираем пункт меню "Edit -> Cut" (вырезать).

Теперь выделяем вкладку "Калькулятор" и вставляем в нее вырезанные компоненты, выбирая в пункте меню "Edit -> Paste" (вставить).

То же самое проделайте с остальными вкладками, вставляя в них нужные компоненты. У меня получилась вот так.







Сейчас я предлагаю скомпилировать программу и протестировать наш новый интерфейс! Если всё отлично и результат вас устраивает, то переходим к изучению следующего и последнего на этот урок компонента RichEdit.

RichEdit по сути то же самое, что и Memo, но с более расширенными возможностями. Например, форматирование отдельных абзацев текста и поддержка формата ".rtf".

Разместим RichEdit в четвертой вкладке компонента PageControl, а вкладку переименуем в "RichEdit". Еще нам потребуется парочка кнопок, поэтому кинем их туда же, куда и RichEdit.

Переходим из визуальной части программы во внутреннюю и начинаем печатать код.

Создаем обработчик события Button3Click и пишем код:

```
1 procedure TForm1.Button3Click(Sender: TObject);
2 begin
3 RichEdit1.Lines.Add('Добавленная строка');
4 end;
```

Компонент RichEdit имеет свойство Lines, точно такое же как и в Memo. Этот код, как вы уже наверно догадались, добавляет в самый конец новую строку в RichEdit. Теперь изменим код в обработчике на следующий:

```
1 procedure TForm1.Button3Click(Sender: TObject);
2 begin
3 RichEdit1.Text:='Новый текст';
4 end;
```

Этот код не добавляет новую строку, как предыдущий, а заменяет весь текст в RichEdit на присваиваемый. Кстати, этим способом тоже возможно добавить новую строку:

```
1 procedure TForm1.Button3Click(Sender: TObject);
2 begin
3 RichEdit1.Text:='Новый текст'+#13#10+'Новая строка';
4 end;
```

Сначала мы присваиваем новый текст в RichEdit, потом прибавляем комбинацию "#13#10" и потом еще прибавляем текст новой строки.

Копипаст для ознакомления: Тип данных string представляет собой совокупность одного или нескольких символов, каждый из которых записывается в виде символа "#" и числа от 0 до 255 (в десятичной или шестнадцатеричной форме) - каждая такая комбинация обозначает соответствующий ASCII-символ.

Комбинация "#13#10" - это комбинация возврата каретки и символа новой строки.

"#13" - это ASCII-эквивалент значения CR (carriage return - возврат каретки);

"#10" представляет собой LF (line feed - признак новой строки).

Как я говорил раньше, RichEdit позволяет форматировать текст, давайте это сейчас и сделаем:

```
1 procedure TForm1.Button3Click(Sender: TObject);
2 begin
3 RichEdit1.Paragraph.Numbering:=nsBullet;
4 end;
```

Если сейчас скомпилировать программу, то строка, на которой стоит каретка будет отображаться в маркированном списке. Так же можно выровнять наш текст, по левому краю, по центру или по правому.

Выравнивание по центру:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
RichEdit1.Paragraph.Alignment:=taCenter;
end;
```

Таким же образом можно изменять и выделенный текст. Перейдем в обработчик события четвертой кнопки Button4Click и напишем код:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
RichEdit1.SelAttributes.Color:=clRed;
end;
```

Запускаем программу, выделяем какой-нибудь текст и нажимаем на четвертую кнопку. Текст окрашивается в красный цвет. Вместо clRed можно написать любые другие цвета, например clGreen, clBlue, clYellow и так далее...

Тут же можно изменить стиль текста:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
RichEdit1.SelAttributes.Color:=clRed; // Окрашивание
RichEdit1.SelAttributes.Style:=[fsbold]; // Жирный стиль текста
end;
```

Вот таким образом происходит форматирование текста в компоненте RichEdit.

Как я говорил раньше, компонент RichEdit умеет работать с форматом ".rtf". Давайте научимся сохранять текст в этом формате и загружать его обратно в RichEdit.

Добавим для удобства еще две кнопки на четвертую вкладку компонента PageControl и назовем их "Сохранить" и вторую кнопку "Загрузить".

В обработчике события для кнопки "Сохранить" пишем код:

```
procedure TForm1.Button5Click(Sender: TObject);
begin
RichEdit1.Lines.SaveToFile('MyFile.rtf'); // Save = Сохранить
end;
```

А для кнопки "Загрузить" следующий код:

```
procedure TForm1.Button6Click(Sender: TObject);
begin
RichEdit1.Lines.LoadFromFile('MyFile.rtf'); // Load = Загрузить
end;
```

В одинарных кавычках указываем путь, куда сохраняем файл и откуда загружаем файл. Файл сохранится в папку, где находится сама программа, то есть сам ".exe" файл.

Теперь вы можете открыть сохраненный файл "MyFile.rtf" например в программе Microsoft Office Word.

Сейчас измените текст в RichEdit и загрузите сохраненный файл.

В следующем уроке мы продолжим изучение компонентов с вкладки Win32.

Ну вот и всё с программой «Калькулятор» закончили!

Урок 15 - Знакомство с компонентами (часть 6/12)

В этом уроке мы продолжаем знакомиться с компонентами из вкладки Win32 и сейчас рассмотрим компонент под названием TrackBar. Компонент TrackBar представляет из себя некое поле с засечками и бегунок.

Для чего он нужен? Предположим, что вы пишите программу под кодовым названием медиа проигрыватель и вам нужен компонент для изменения громкости или баланса звука, так вот TrackBar отлично справится с этой задачей.

Открываем Delphi, и кидаем на форму компонент TrackBar. Свойства Min и Max отвечают за минимальную и максимальную позицию ползунка соответственно.

По умолчанию Min = 0, а Max = 10. Давайте изменим максимальное значение, пусть будет 50. И как мы видим, после изменения максимального значения число засечек увеличилось, что вполне адекватно на мой взгляд.

Следующее свойство Orientation. Оно отвечает за ориентацию ползунка (не подумайте ничего плохого). В этом свойстве мы можем задать отображение ползунка: горизонтальное или вертикальное.

Следующие свойства SelStart и SelEnd. Они играют информативную роль в жизни компонента TrackBar. Выставим значения SelStart = 20 и SelEnd = 30. Появилась синяя (возможен другой цвет) область в поле TrackBar'a и оно показывает пользователю диапазон оптимальных значений или наоборот - не желательных.

Следующее свойство ThumbLenght. Оно отвечает за размер TrackBar'a. Выставим значение ThumbLenght = 10 и TrackBar станет у нас узеньким.

Следующее свойство TickMarks. Оно отвечает за положение засечек на компоненте и может принимать следующие значения:
- отображение засечек сверху (справа при вертикальном отображении);
- снизу (слева);
- и по обе стороны, то есть и сверху и снизу одновременно (либо справа и слева, тут ничего сложного нет).

Следующее свойство Position. Оно отвечает за положение самого ползунка на компоненте. Например, у нас стоит максимальное значение 50 (Max = 50), если сейчас изменить значение Position на 25, то ползунок будет по середине TrackBar'a.

С самым основным мы с вами познакомились. Ну а остальные свойства идентичны свойствам других компонентов.

Двигаемся дальше и кидаем на форму уже другой компонент под названием ProgressBar. Он необходим для отображения загрузки в программе.

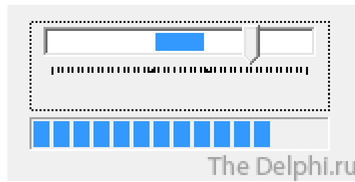
У него так же есть свойство Position и давайте его сразу изменим на 50. Вы наверняка видели много раз этот компонент в других программах. Вообще, свойства компонентов ProgressBar и TrackBar в основном схожи, единственное, что наверное следует выделить свойство Step. Оно изменяет длину шага. То есть, если минимальное значение 0, максимальное 50, а шаг равен 25, то спустя какое-то время наш компонент будет сразу заполнен на половину.

Давайте объединим два этих компонента. У компонента TrackBar есть событие OnChange на вкладке Events, оно возникает при изменении положения ползунка.

В этом событии давайте напишем следующую строку кода:

```
1 procedure TForm1.TrackBar1Change(Sender: TObject);
2 begin
3 ProgressBar1.Position:=TrackBar1.Position; //вот она, строка.
4 end;
```

При перемещении ползунка, будет присваиваться значение позиции ползунка компоненту ProgressBar. В общем скомпилируйте программу и посмотрите что получилось.



Рассмотрим еще один компонент, которого зовут UpDown. Этот компонент взаимодействует с другими компонентами. Лучше всего показать на примере. Для этого разместим на форме несколько компонентов: Edit и UpDown.

У компонента UpDown есть свойство Associate, в нем можно задать имя компонента, с которым будет работать (взаимодействовать) компонент UpDown. Выберем в этом свойстве из выпадающего списка наш Edit. И сразу мы можем созерцать, как UpDown прижался к Edit'y.

После компиляции и запуска программы, нажимая на компонент UpDown, мы автоматически меняем значение в компоненте Edit.

У компонента UpDown имеется полезное свойство Increment, которое задает величину изменения значения при нажатии на стрелочки. То есть, если Increment = 5, то после одного нажатия на стрелочки Up или Down значение изменится на 5 единиц.

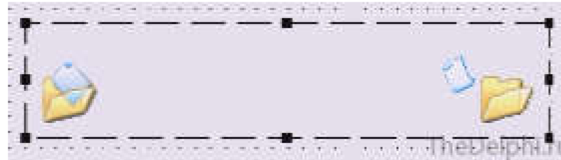
В следующем уроке мы продолжим изучение компонентов с вкладки Win32.

Ну вот и всё!

Продолжаем изучать вкладку Win32 и в этом уроке мы познакомимся с двумя компонентами, а именно: Animate и StatusBar.

Открываем Delphi, и кидаем на форму компонент Animate, он позволяет проигрывать на форме клипы в формате AVI и отображать стандартную анимацию. Но я не рекомендую загружать в этот компонент клипы, потому что размер файла должен быть небольшой и должно стоять определенное сжатие. Лично я использую этот компонент для отображения стандартных анимаций в програме.

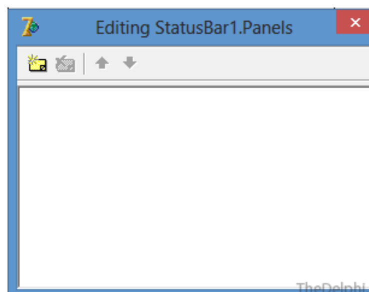
Для этого у компонента Animate есть свойство CommonAVI. Из выпадающего списка значений можно выбрать любую анимацию. Давайте выберем aviCopyFile и изменим свойство Active в значение True. У нас на форме сразу отображается анимация копирования файлов.



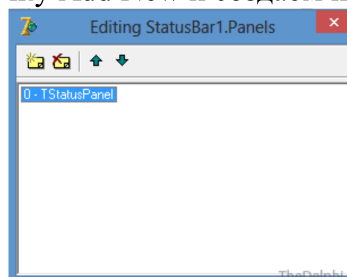
Скажу сразу, если у вас не windows xp, то отображение стандартных анимаций не гарантируется, так как их может не быть в системе.

Свойство Transparent отвечает за прозрачность фона анимации. Если стоит значение True, то фон прозрачен.

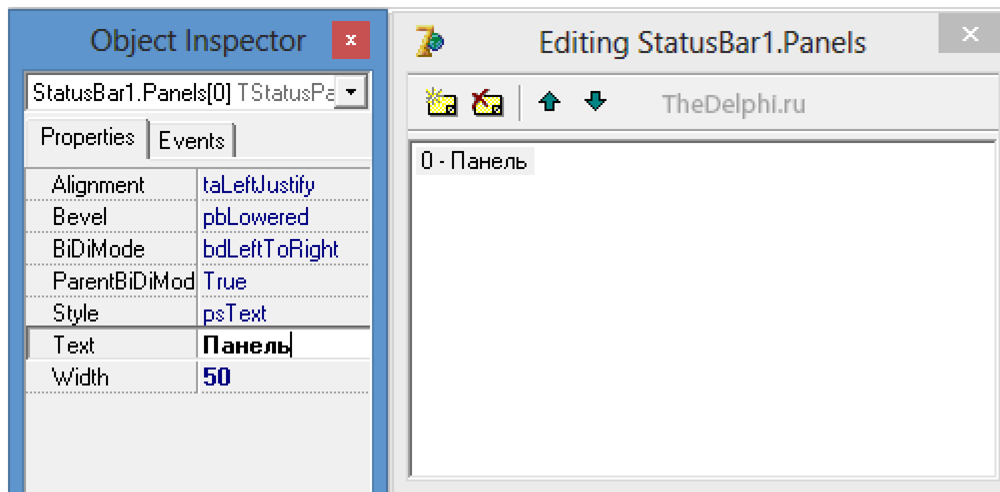
Двигаемся дальше и переходим к следующему компоненту StatusBar. Кидаем его на форму и тут же видим, что он прилипает к нижней части формы (что бы изменить положение компонента воспользуйтесь свойством Align). StatusBar необходим прежде всего для отображения справочной информации и каких либо подсказок. Давайте создадим несколько панелей. Делается это очень просто. Кликаем правой кнопкой мыши по компоненту StatusBar и нажимаем на первый же пункт меню Panels Editor. Появляется окошко редактирования панелей.



Нажимаем на желтую кнопочку Add New и создаем новую панель.



Сразу мы можем ввести сюда какой-нибудь текст, например "Панель", задать ширину панели, ну и так далее...



Давайте еще создадим 2 панели для дальнейшего примера, но не будем у них менять никаких свойств.

В компоненте StatusBar очень легко отображать всплывающие подсказки. Для этого поместим на форму компонент Button и в свойстве Hint (всплывающая подсказка) напишем **"Кнопка"**. Теперь выделяем компонент StatusBar и в свойстве AutoHint меняем значение на True. Компилируем программу, наводим на кнопку и видим, что подсказка отобразилась в первой панели компонента StatusBar.

Если мы хотим вывести текст в определенную панель, то нужно написать следующий код:

```
1 procedure TForm1.Button1Click(Sender: TObject); //событие OnClick у компонента Button
2 begin
3 StatusBar1.Panels.Items[2].Text:='Текст'; //строка кода
4 end;
```

Здесь мы обращаемся к третей по счету панели, которая имеет индекс 2 и изменяем у нее свойство Text.

Вы спросите: "Почему же так? Панель третья, а индекс у нее 2". Все просто, индекс начинается с нуля. То есть первая панель имеет индекс 0, вторая индекс 1, третья индекс 2 и так далее...

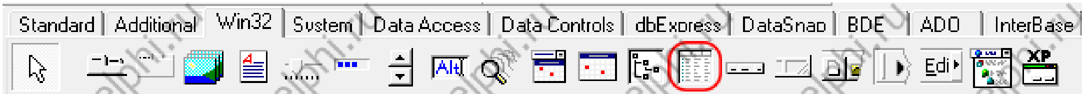
Запустите программу и посмотрите результат.

В следующем уроке мы продолжим изучение компонентов с вкладки Win32.

Ну вот и всё!

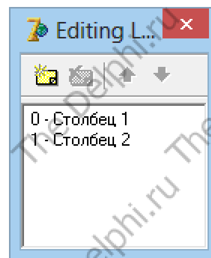
Урок - 17 Знакомство с компонентами (часть 8/12)

В этом уроке мы продолжаем изучать вкладку Win32.
В этот раз мы будем рассматривать компонент под названием ListView.

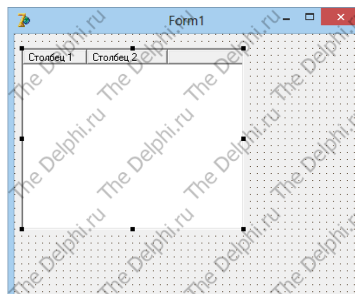


Итак, открываем новый проект и кидаем компонент на форму и сразу ищем свойство Columns в Object Inspector.

Кликаем на многоточие и в появившемся мастере нажимаем 2 раза на жёлтую кнопочку "Add New (Inc)" в верхнем левом углу, появилось 2 новых строки, выделяем каждую из них и изменяем свойство Caption на "Столбец 1" и у второй на "Столбец 2". Вот что должно получится:

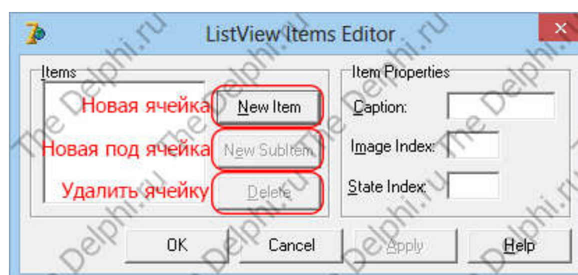


Также можно изменять другие свойства, например Width, но мы пока делать этого не будем. Закрываем редактор, и... ничего не поменялось. Все дело в том что у нас стоит не подходящий стиль отображения, изменить его можно с помощью свойства ViewStyle, поставим на vsReport, и вот, уже что-то получается:



Для удобства просмотра и редактирования таблицы у этого компонента есть замечательное свойство GridLines, по умолчанию False, поменяем его на True, появилась разметка!

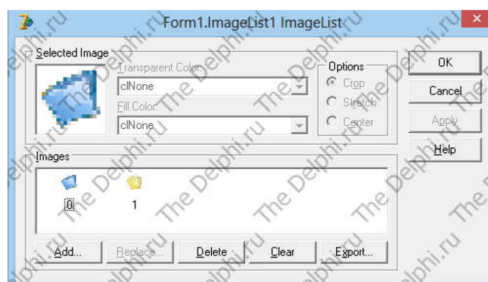
Ну что ж давайте внесем какие-нибудь данные, делается это просто, кликаем правой кнопкой мыши на компонент и выдираем в появившемся меню "Items Editor", должно появится вот такое окно:



Добавим первый Item, кликаем на кнопку "New Item", и в поле Caption напишем "1", далее точно также добавим второй, только Caption = "2" и третий Caption = "3". добавим под-

ячейку в третьем item'e, выделяем третий Item и кликаем на кнопочку "SubItem", Caption задаем "3,1", все, мы добавили, нажимаем "OK".

Все готово, но выглядит как-то мрачновато. Давайте украсим наш ListView, для этого добавим на форму компонент ImageList, о нем мы говорили в предыдущих уроках, и загрузим два изображения.



Теперь надо привязать наш ImageList к ListView, для этого есть свойство SmallImages, в нем указываем наш ImageList1. Вы наверно заметили при добавлении ячеек параметр "Image Index", это индекс (номер картинки в ImageList). Давайте вернемся в Items Editor и установим данное свойство, основным ячейкам зададим "0", а дочерней ячейке (3,1) "1". Не забудем и про заголовки столбцов, вернемся в мастер создания столбцов (клик правой кнопкой мышки по ListView->Columns Editor) и также свойство ImageIndex установим на 0.

Теперь настало время научиться добавлять ячейки динамически (в процессе работы программы). Для этого вытащим 4 кнопки Button и 1 Label. Установим свойство Caption у Button1 равное "Добавить", второй "Удалить", третей "Добавить под-ячейку", четвертой "Изменить".

Для начала нам нужно узнать номер выбранной ячейки. Добавляем глобальную переменную **t: integer**.

Далее создаем обработчик события ListView1 OnChange и пишем код:

```
1  procedure TForm1.ListView1Change(Sender: TObject; Item: TListItem;
2     Change: TItemChange);
3  begin
4  t:= ListView1.ItemIndex;    //Записываем номер текущей ячейки
5  Label1.Caption:= IntToStr(t); //Выводим номер текущей ячейки
6  end;
```

Сейчас пишем код для первой кнопки:

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  begin
3  ListView1.Items.Add.Caption:='Новая ячейка'; //Добавляем ячейку с текстом "Новая ячейка".
4  end;
```

При нажатии на первую кнопку, как мы и хотели, добавляется новая ячейка. Теперь удалим ячейку, как раз для этого нам и понадобится переменная t. Через свойство **ListView1.Items.Item[t].Delete** можно обратиться к любой ячейке, главное, чтоб она существовала.

Создаем обработчик события Button2Click и проверим, чтобы при нажатии исчезала выбранная ячейка.

Добавим под-ячейку. Создаем обработчик события Button3Click:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
ListView1.Items.Add.SubItems.Add('Новая ячейка'); //Добавляем под-ячейку
//с текстом "Новая ячейка".
end;
```

Ну и сразу добавим возможность изменять текст внутри ячеек. Вытащим на форму Edit и создадим обработчик события для Button4Click:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
ListView1.Items.Item[t].Caption:=Edit1.Text; //Изменяем текст в выбранной ячейке.
end;
```

Ну вот и всё, ячейки удаляются и изменяются.

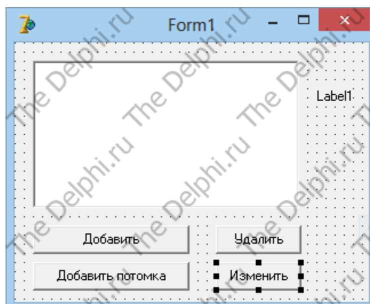
В следующем уроке мы продолжим изучение компонентов с вкладки Win32 и перейдем к компоненту TreeView.

Удачи!

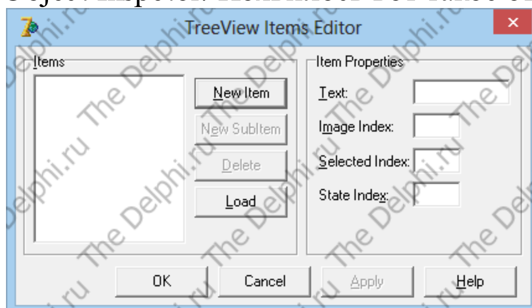
В этот раз мы будем рассматривать компонент под названием TreeView. Мы научимся добавлять, удалять, изменять и сортировать.

Этот компонент предназначен для отображения сложной иерархической структуры данных. Например, в левой области проводника Windows используется этот компонент для быстрой навигации по папкам.

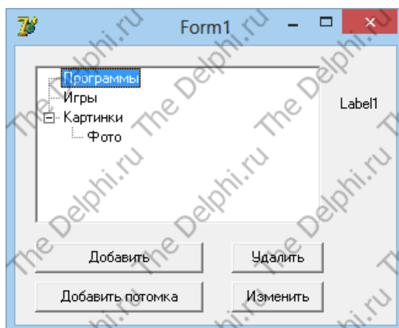
Создаем новый проект. И так перейдём к делу, нам понадобятся компоненты: TreeView, 4 Button, Label и Edit. У первой кнопки Caption давайте сделаем "Добавить", у второй " Добавить потомка ", третьей " Удалить " и для четвертой "Изменить". Разместите все компоненты вот так:



Как и у ListView, у этого компонента есть собственный редактор строк. Вызовем его кликнув по свойству Items в Object Inspetor. Появилось вот такое окно:



Оно похоже на то, что использовалось в ListView, за исключением лишь одной функции, TreeView обладает функцией загрузки/сохранения текущей информации и соответственно добавилась одна кнопка "Load". Добавляем новый Item и Text у него "Программы", добавим еще один с текстом "Игры" и один - "Картинки", в последний добавим потомка, клик по "New SubItem" с текстом "Фото". Скомпилируем и посмотрим что получилось. Нажав на плюсик находящийся перед "Картинки", разворачивается список и в нем наш потомок, а "Картинки" это родитель.



Теперь приступим к добавлению строк. Как и в прошлом уроке нам понадобится текущий выделенный Item. Создадим 2 глобальных переменных t:integer и MyNode:TTreeNode ну и обработчик события TreeView, OnChange:

```
procedure TForm1.TreeView1Change(Sender: TObject; Node: TTreeNode);
begin
MyNode:=Node;
t:=Node.AbsoluteIndex;
Label1.Caption:= IntToStr(t);
end;
```

Проверим... Обратите внимание, казалось бы потомок должен иметь индекс 2.1, но это не так, в TreeView в каком порядке развернуты потомки такие они имеют индексы, то есть в нашем случае потомок "Фото" имеет индекс 3. MyNode здесь нужна для того чтоб узнать какой

родительский Item сейчас выделен.

Ну а теперь можно и добавить. Вытащим Edit в коде, чтоб можно было задавать имя нового элемента.

Создадим обработчик события Button1, OnClick:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
TreeView1.Items.Add(MyNode, Edit1.Text);
end;
```

Ну и сразу же добавим потомка. Создадим обработчик события Button2, OnClick:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
TreeView1.Items.AddChild(MyNode, Edit1.Text);
end;
```

Создадим обработчик события Button3, OnClick:

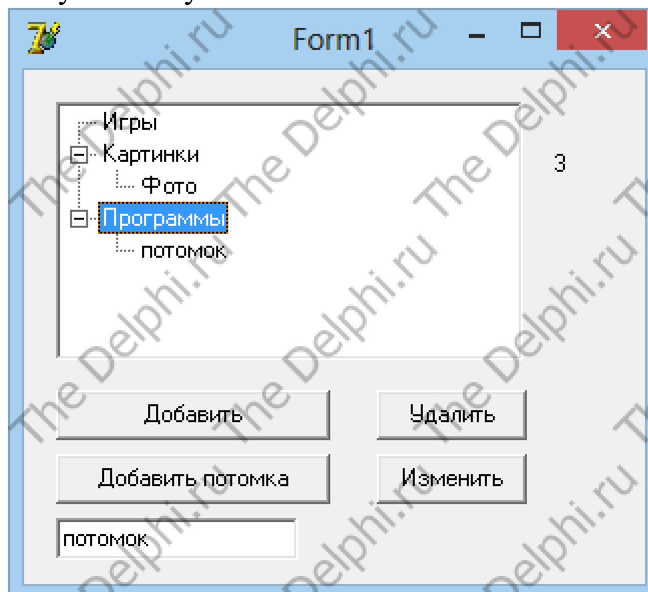
```
procedure TForm1.Button3Click(Sender: TObject);
begin
TreeView1.Items.Item[t].Delete;
end;
```

Изменим элемент. Создадим обработчик события Button4, OnClick:

```
procedure TForm1.Butto4Click(Sender: TObject);
begin
TreeView1.Items.Item[t].Text:=Edit1.Text;
end;
```

Теперь мы можем полноценно редактировать содержимое компонента, но не хватает сортировки. Сортировка управляется свойством SortType, установим сортировку по алфавиту: stText.

Ну вот и все, вот что у нас получилось:



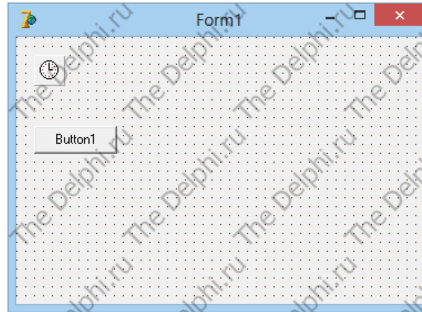
В следующем уроке мы начнём изучение компонентов с вкладки System, первым компонентом будет TTimer.

Удачи!

В этом уроке мы начинаем изучать вкладку System. Сейчас мы будем рассматривать компонент под названием Timer.

Этот не визуальный компонент предназначен для повторения участка кода через определённые промежутки времени. Он имеет всего один обработчик события OnTimer.

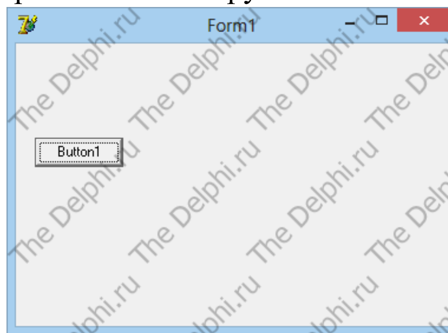
Вытащим на форму Timer и кнопку. Наша цель - двигать кнопочку в право через каждые 100 миллисекунд.



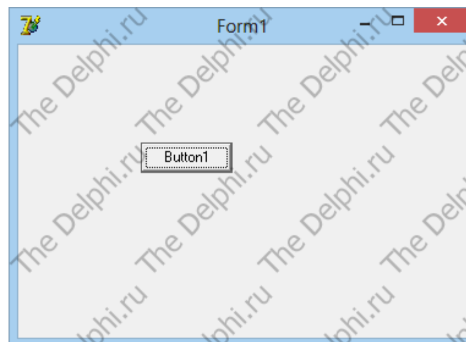
Создаем обработчик события OnTimer и пишем код:

```
1 procedure TForm1.Timer1Timer(Sender: TObject);
2 begin
3 Button1.Left:=Button1.Left + 1;
4 end;
```

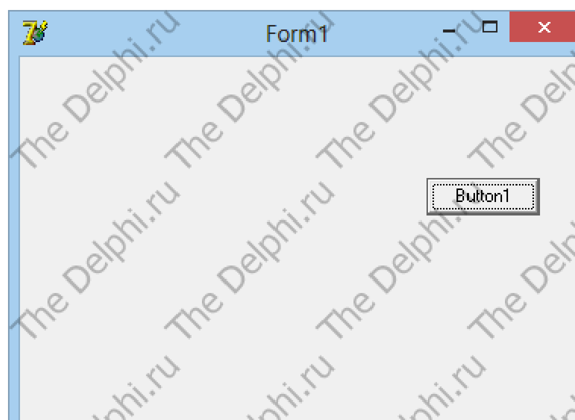
Обратимся к свойствам таймера, свойство Interval установим на 100 (значение указывается в миллисекундах). Свойство Enabled управляет работой таймера. Переместим кнопочку подальше от правого края и скомпилируем. Точка начала:



Спустя 5 секунд:



Спустя 10 секунд:



Кнопочка медленно, но верно ползёт к краю стараясь скрыться за границей формы.

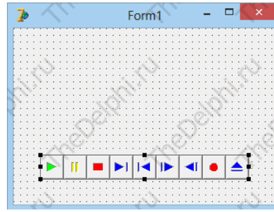
В следующем уроке мы продолжим изучение компонентов с вкладки System и перейдем к компоненту MediaPlayer.

Удачи!

Урок 20 - Знакомство с компонентами (часть 11/12)

В этом уроке мы продолжаем изучать вкладку System. Сейчас мы будем рассматривать компонент под названием MediaPlayer.

Этот компонент производит операции над звуковыми правилами, а как он это делает, сейчас мы и узнаем. Итак, давайте вытащим его на форму.



Как видите он имеет 9 кнопок (Воспроизведение, пауза, Стоп, 4 кнопки перемотки, Запись, Открыть). Рассмотрим некоторые свойства компонента: AutoEnable - это свойство отвечает за активность кнопок, когда эта кнопка не может быть использована, она становится не активной. Например, при воспроизведении файла кнопка "Play" деактивируется, а "Stop" наоборот становится активной и т.д. Аналогичны ему свойства AutoOpen и AutoRewind.

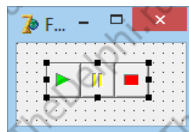
Также не менее интересное свойство FileName, в нем задается путь к файлу. Еще один важный момент, если задать свойству AutoOpen значение True, а свойство FileName оставить пустым, то при запуске программы вы увидите ошибку, поэтому свойству AutoOpen лучше задать False, если не используется FileName. Теперь попытаемся воспроизвести что-нибудь, поместим файл **"20 Урок.mp3"** в корень локального дика "C" ("D").

Создаем обработчик события Fomr1OnCreate и пишем код:

```
1  procedure TForm1.FormCreate(Sender: TObject);
2  begin
3  MediaPlayer1.FileName:='C:\1.mp3'; //Задаем путь к файлу
4  MediaPlayer1.Open;  //Открываем файл
5  end;
```

Наш компонент стал активен, нажмем на "Play", и вот звук воспроизводится, можно его остановить, поставить на паузу и т.д.

Разберём еще 2 свойства: EnableButtons и VisibleButtons, в первом можно устанавливать какие кнопки буду активны, а какие нет, второе свойство аналогично первому, но оно скрывает ненужные кнопки, вот им то мы и воспользуемся, скроем лишние кнопки(bfNext, bfPrev, bfStep, bfBack, bfRecord, bfEject), тоже самое сделаем и в EnableButtons. Все готово, осталось только 3 необходимые кнопки.

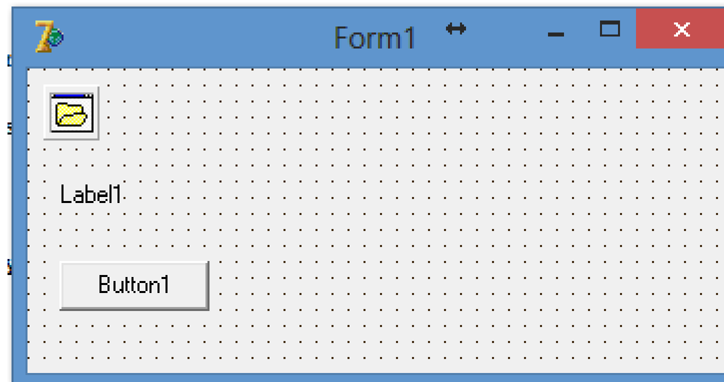


В следующем уроке мы перейдем на новую вкладку под названием Dialogs и перейдем к компоненту OpenDialog.

Удачи!

Урок 21 - Знакомство с компонентами (часть 12/12)

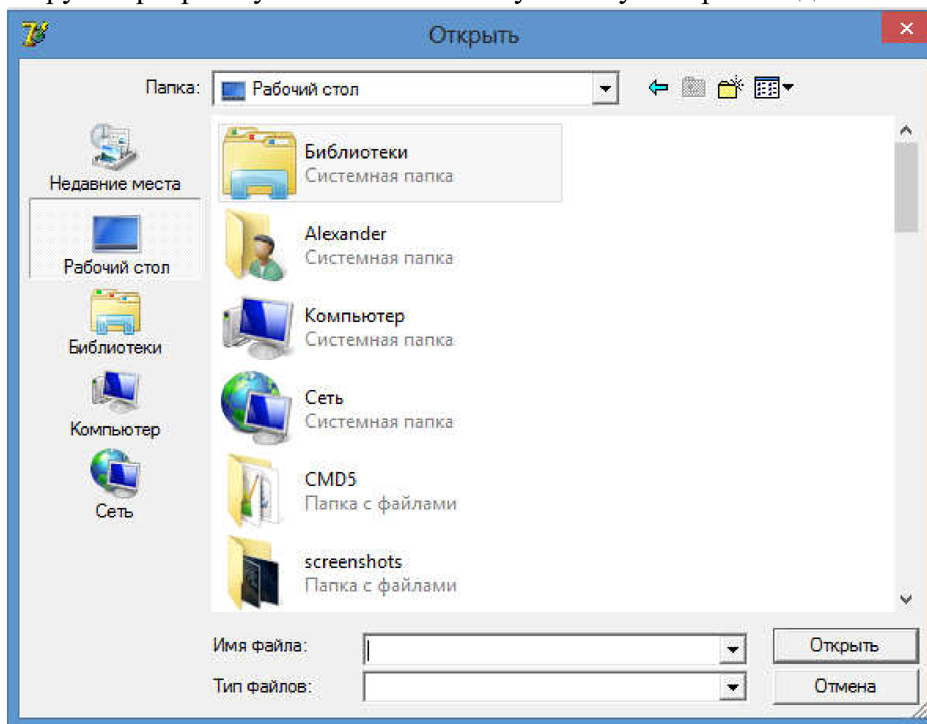
Здравствуйте, дорогие друзья! Сегодня я расскажу вам про вкладку с компонентами - Dialogs. Компонент OpenDialog позволяет инициализировать диалог открытия файла. Для этого, поместим на форму следующие компоненты:
- OpenDialog (вкладка Dialogs);
- Button (вкладка Standard);
- Label (вкладка Standard).



Создаем обработчик события OnClick на кнопке и пишем код:

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  begin
3  OpenDialog1.Execute;
4  end;
```

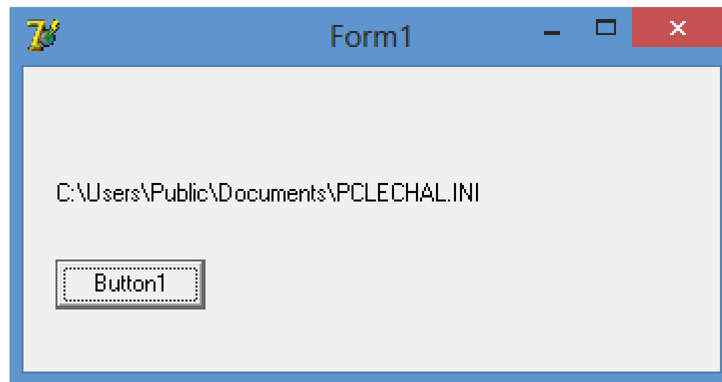
Скомпилируем программу и нажмем на нашу кнопку. Откроется диалог открытия файла:



Усовершенствуем наш код:

```
1  procedure TForm1.Button1Click(Sender: TObject);
2  begin
3  if OpenDialog1.Execute=true then
4  Label1.Caption:= OpenDialog1.FileName;
5  end;
```

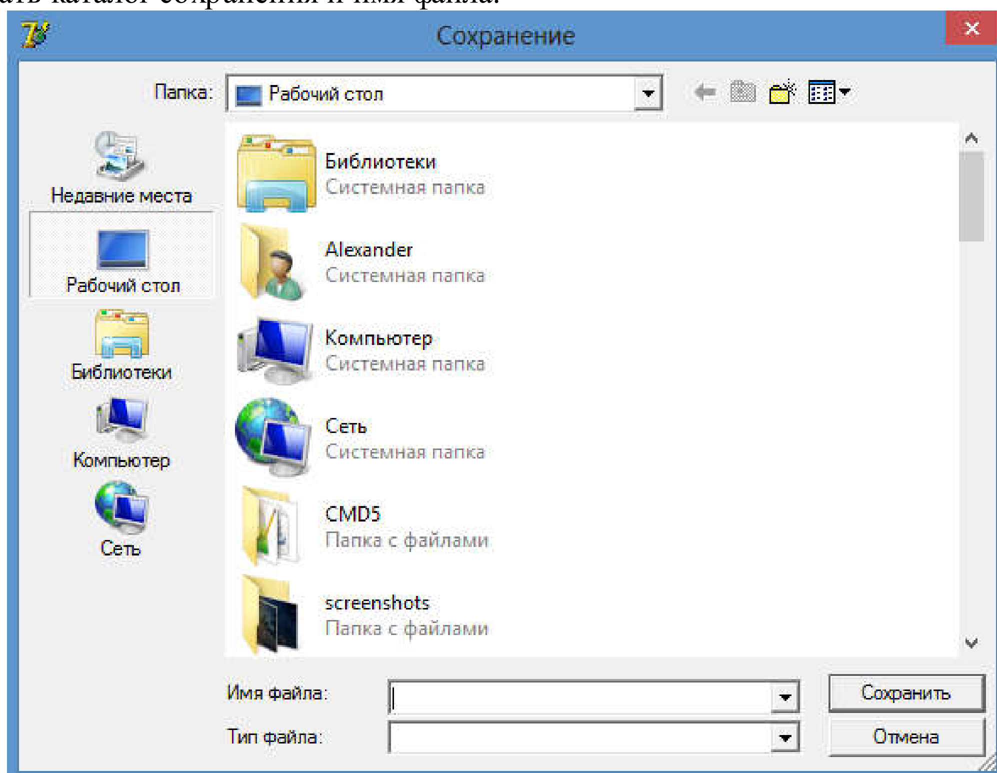
Теперь если пользователь откроет диалог и выберет какой-нибудь файл, то в Label отобразится полный путь до файла. Скомпилируем и проверим.



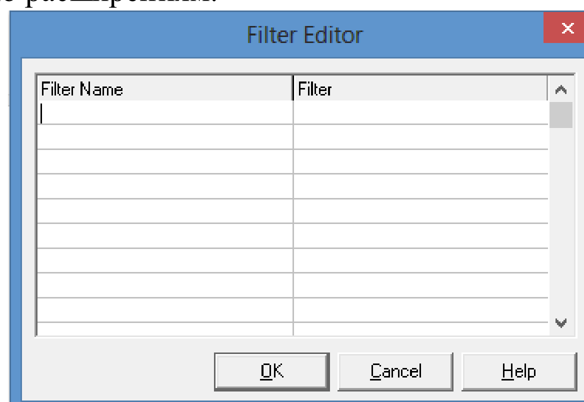
Далее изучим компонент SaveDialog. Поместим его на форму и напишем код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if SaveDialog1.Execute=true then
Label1.Caption:= SaveDialog1.FileName;
end;
```

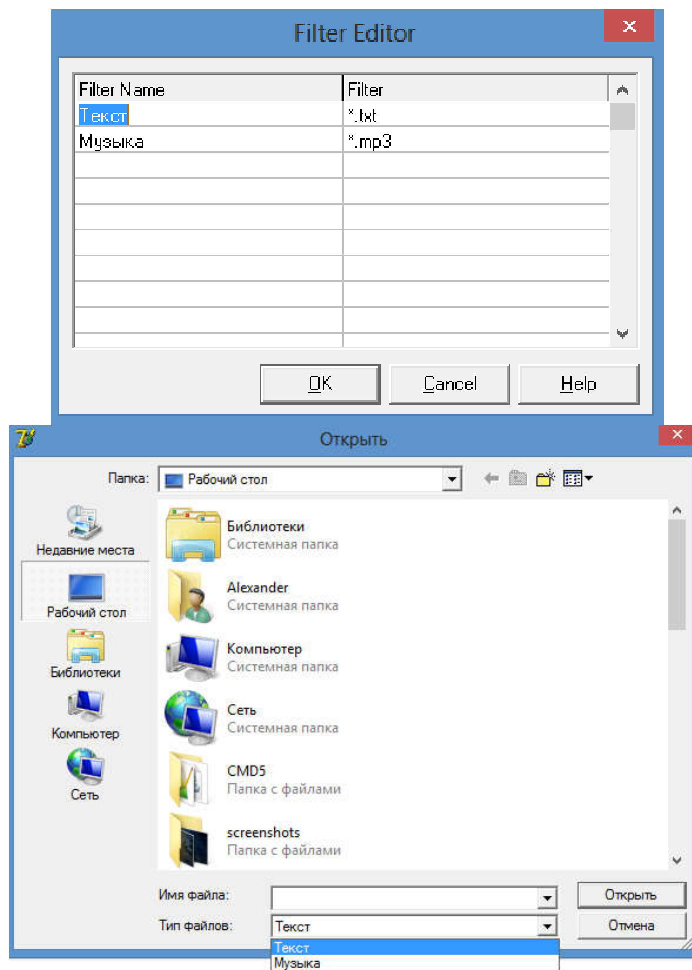
Скомпилируем программу и нажмем на кнопку. Откроется диалог сохранения файла, где нужно указать каталог сохранения и имя файла.



Конечно, физического сохранения файла не произойдет, только лишь в Label запишется полный путь до файла, потому что компонент SaveDialog позволяет только выбрать каталог и имя файла. У этих компонентов есть такое свойство, которое называется Filter. Оно позволяет отфильтровывать файлы по расширениям.



Первый столбик Filter Name задает имя фильтра. Второй столбик Filter задает сам фильтр, например: "*.txt" или "*.exe".



Остальные компоненты на вкладке Dialogs схожи с рассмотренными компонентами. Удачи!

Урок 22 - Принцип работы с файлами

В этом уроке изучим принцип работы с файлами. Давайте ознакомимся с набором функций, необходимых для работы:

- function FileCreate(const FileName: String): integer;

Создаёт файл в указанном пути FileName и возвращает индекс созданного файла. В случае ошибки при создании, она вернет -1.

- function FileOpen(const FileName: String; Mode: LongWord): integer;

Открывает файл в заданном пути FileName с параметрами Mode, она также в случае ошибки вернет -1. Параметр Mode может принимать одно из следующих значений:

- fmOpenRead - открытие только на чтение;
- fmOpenWrite - открытие только на запись;
- fmOpenReadWrite - открытие и на чтение и на запись.

- function FileRead(handle: integer; var Buffer; Count: integer): integer;

Читает открытый или созданный файл. Вместо пути она принимает индекс файла handle. Buffer - куда будет записано содержимое. Count - количество байтов, которое нужно прочитать.

- function FileWrite(handle: integer; var Buffer; Count: integer): integer;

Аналогичная функция чтению, но вместо чтения она записывает содержимое размером Count (в байтах) переменной Buffer в файл.

- function FileClose(handle: integer);

Закрывает файл, чтобы другие программы тоже имели доступ к нему.

- function FileSearch(const Name, DirList: string): String;

Осуществляет поиск файла Name в одной или более папках DirList, отделённых друг от друга точкой с запятой. Необходимо указывать не только название папки, но и полный адрес этой папки. Имя файла может быть, как файловым именем, так и полным адресом файла. Если файл будет найден, то возвращается полный адрес файла, включая имя файла, в случае не нахождения искомого файла будет возвращена пустая строка.

ПРЕДУПРЕЖДЕНИЕ: ВСЕГДА сначала поиск будет проходить в текущей папке, независимо от перечисленных директорий. Если файл обнаруживается там, то путь файла возращён не будет, а только имя файла.

- function FileSeek(Handle, Offset, Origin: Integer): Integer;

Устанавливает позицию для чтения и записи (в байтах) для открытого файла и возвращает новую позицию в файле (байтовое смещение от начала файла). В ниже перечислены возможные значения параметра Origin. Чтобы определить текущую позицию в файле, передайте Origin значение File_Current и Offset, равное нулю.

File_Begin (0) - Смещение (offset) рассчитывается относительно начала файла.

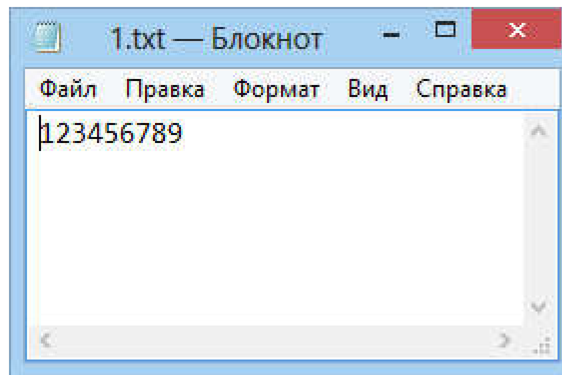
File_Current (1) - Смещение рассчитывается относительно текущей позиции в файле.

File_End (2) - Смещение рассчитывается относительно конца файла.

Все с теорией мы покончили, теперь что-нибудь считаем. Вытащим на форму 2 кнопки и Memo и напишем в обработчике события 1 кнопки OnClick следующий код:

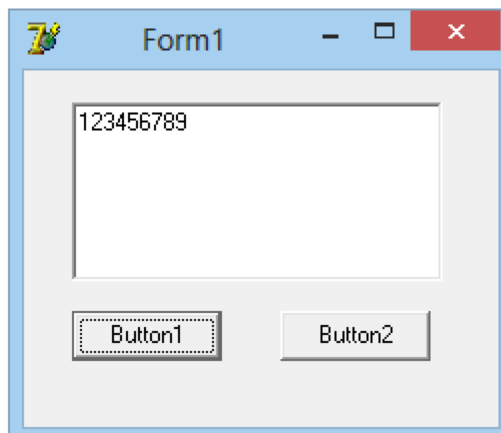
```
1 procedure TForm1.Button1Click(Sender: TObject);
2 var
3   F: integer;  //наш файл
4   Str: string[8]; //Строка которую мы будем писать
5 begin
6   F:= FileCreate('D:/1.txt'); //Создаём файл
7   FileWrite(F, '123456789', SizeOf(str)); //Записываем в него строку "123456789"
8   FileClose(F); //Закрываем файл
9 end;
```

Функция SizeOf возвращает размер типа или самой переменной в байтах, т.к. заранее размер строки неизвестен. Скомпилируем и нажмем кнопочку.



Теперь прочитаем это. Создаём обработчик события 2 кнопки OnClick и пишем код:

```
01 procedure TForm1.Button2Click(Sender: TObject);
02 var
03   F: integer;  //наш файл
04   Str: string[8]; //Строка в которую мы сохранять
05 begin
06   F:= FileOpen('D:/1.txt', fmOpenRead); //открываем файл на чтение
07   FileRead(F, Str, SizeOf(str));  //читаем
08   Memo1.Text:=Str;  //присваиваем memo нашу строку
09   FileClose(F);  //Закрываем файл
10 end;
```



Удачи!

Урок 23 - Функции для работы с мышью

В этом уроке изучим функции для работы с мышью. Итак, начнём! Вытащим на форму 5 кнопок и зададим свойства Caption: "Переместить курсор", "Скрыть курсор", "Показать курсор", "Поменять кнопки", "Вернуть кнопки". Вы, наверное, догадались о чем сейчас пойдет речь.

Первая наша функция: SetCursorPos(x, y: integer), перемешает курсор в указанные координаты. Сразу же создадим обработчик события первой кнопки Button1Click и напишем вот такой код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SetCursorPos(100, 200);
end;
```

При клике на кнопку курсор переместится в координаты X - 100 пикселей, Y - 200 пикселей.

Следующая функция ShowCursor(bShow: longbool), скрывает курсор с параметром False и показывает его с параметром True. Также создадим событие второй и третьей кнопки.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ShowCursor(False);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  ShowCursor(True);
end;
```

При нажатии на вторую кнопку курсор исчезает, а при нажатии на третью появляется.

Ну и наконец последняя функция SwapMouseButton(bShow: longbool), менять местами кнопки мыши (левую на правую и наоборот). Создадим 2 обработчика событий 3 и 4 кнопки.

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  SwapMouseButton(True);
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  SwapMouseButton(False);
end;
```

При нажатии на 4 кнопку происходит смена кнопок. **ВАЖНО**!!!: если не вернуть обратно то при завершении работы программы все настройки сохранятся.

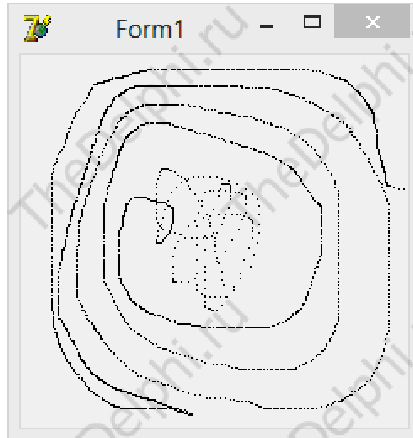
Задание на закрепление: Создайте кнопку, которая заставит курсор исчезнуть, при этом курсор должен стать видимым при клике левой кнопки мыши на любом месте программы.

Урок 24 - Изучаем компонент PaintBox

В этом уроке изучим компонент PaintBox на вкладке System. У этого компонента есть один недостаток, проявляющий себя только на Windows XP: если работающую программу перекрыть окном другой программы, то все содержимое компонента стирается.

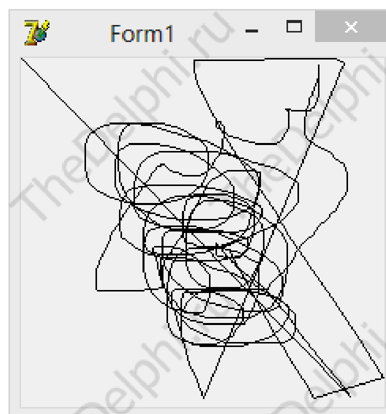
Вытащим компонент на форму и попытаемся нарисовать, для этого сделаем обработчик события OnMouseMove и напишем следующий код:

```
procedure TForm1.PaintBox1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  PaintBox1.Canvas.Pixels[x,y]:=clBlack; //задаем черный цвет пиксела в координатах курсора (X,
Y);
end;
```



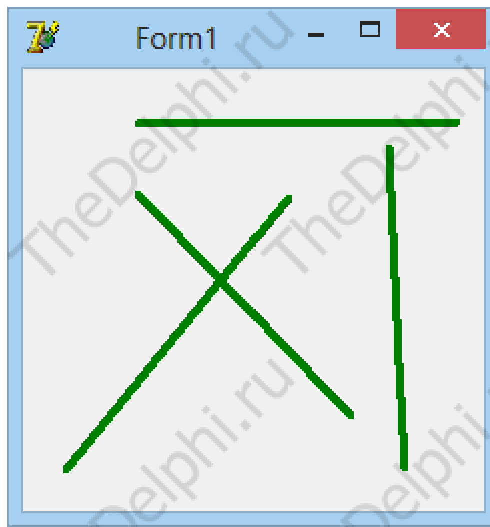
Рисуется! Но линия прерывается, дело в том, что это событие срабатывает не сразу а через определённый промежуток времени (16 - 47 мс зависит от параметров компьютера), что бы не было этого модифицируем код:

```
procedure TForm1.PaintBox1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  PaintBox1.Canvas.LineTo(X,Y); //задаем черный цвет пиксела в координатах курсора (X, Y);
end;
```



Так-то лучше. Теперь нарисуем прямые линии, что бы не мешать уберём прошлый код и создадим новый обработчик события OnMouseDown:

```
procedure TForm1.PaintBox1MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  PaintBox1.Canvas.Pen.Width:= 4; //задаем размер карандаша
  PaintBox1.Canvas.Pen.Color:= clGreen; //цвет
  if button = mbLeft then
  PaintBox1.Canvas.MoveTo(X,Y); //если нажата левая кнопка то ставим 1 точку в ее координаты
  if button = mbRight then
  PaintBox1.Canvas.LineTo(X,Y); //если нажата правая кнопка то ставим 2 точку в ее координаты
и чертим линию
end;
```

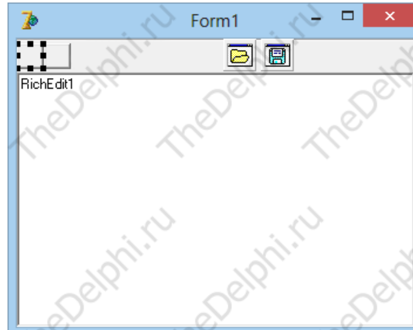


При нажатии на левую кнопку программа запомнит точку начала линии, а при нажатии на правую нарисует линию от 1 точки и до текущих координат курсора.

Урок 25 - Подробное изучение RichEdit'a

В этом уроке изучим компонент RichEdit на вкладке Win32. Кинем на форму компонент RichEdit и ToolBar, свойство Align у RichEdit установим на **alClient** и наш компонент растянется на всю форму. Определимся, что будет уметь делать наш редактор: Загружать, сохранять, выравнивание (по левому краю, по центру, по правому краю), стилизация текста.

Итак, начнем с самого простого, загрузка и сохранение, вытаскиваем 2 компонента OpenDialog и SaveDialog с вкладки Dialogs, и кликаем на ToolBar правой кнопкой, в появившемся меню выбираем New Button, также создадим еще одну кнопочку.

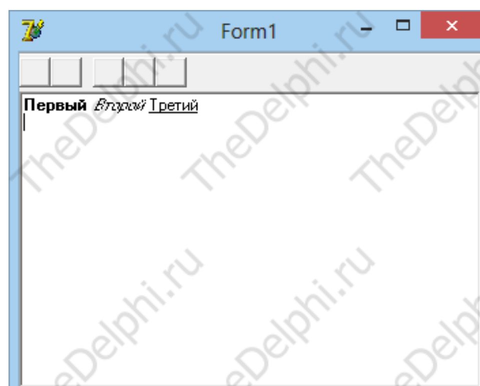


И в первой и во второй создаем событие OnClick и пишем код:

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
 if OpenDialog1.Execute then  //Вызываем диалог открытия
  RichEdit1.Lines.LoadFromFile(OpenDialog1.FileName); //Открываем файл
end;

procedure TForm1.ToolButton2Click(Sender: TObject);
begin
 if SaveDialog1.Execute then //Вызываем диалог сохранения
  RichEdit1.Lines.SaveToFile(SaveDialog1.FileName); //Сохраняем файл
end;
```

Загрузку и сохранение мы реализовали, теперь стилизацию текста, для этого сделаем разделитель в ToolBar'e, кликаем на него правой кнопкой и в меню выбираем New Separator и описанным выше способом создаем 3 кнопки и в их обработчиках событий OnClick пишем код:

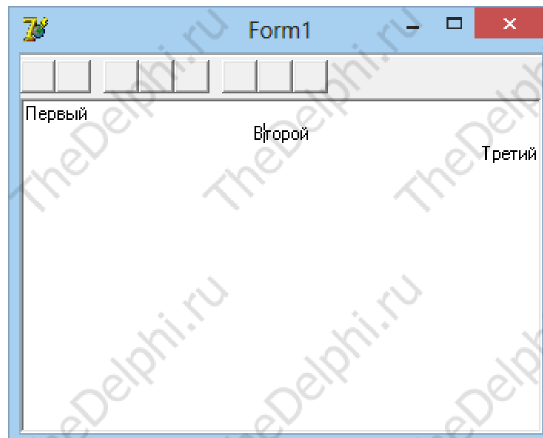
```
procedure TForm1.ToolButton4Click(Sender: TObject);
begin
 RichEdit1.SelAttributes.Style:=[fsBold];  //Жирный
end;

procedure TForm1.ToolButton5Click(Sender: TObject);
begin
 RichEdit1.SelAttributes.Style:=[fsItalic]; //Курсив
end;

procedure TForm1.ToolButton6Click(Sender: TObject);
begin
 RichEdit1.SelAttributes.Style:=[fsUnderline];  //Подчеркнутый
end;
```



Теперь выравнивание, также создаем разделитель и 3 кнопки. В

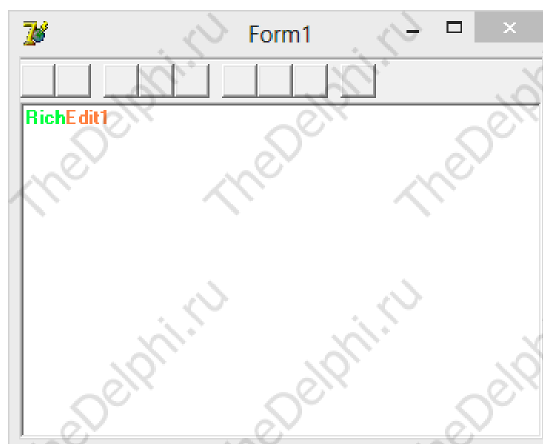
событиях OnClick следующий код:

```
procedure TForm1.ToolButton8Click(Sender: TObject);
begin
 RichEdit1.Paragraph.Alignment:=taLeftJustify;//По левому краю
end;

procedure TForm1.ToolButton9Click(Sender: TObject);
begin
 RichEdit1.Paragraph.Alignment:=taCenter;// По центру
end;

procedure TForm1.ToolButton10Click(Sender: TObject);
begin
 RichEdit1.Paragraph.Alignment:=taRightJustify;// По правому краю
end;
```



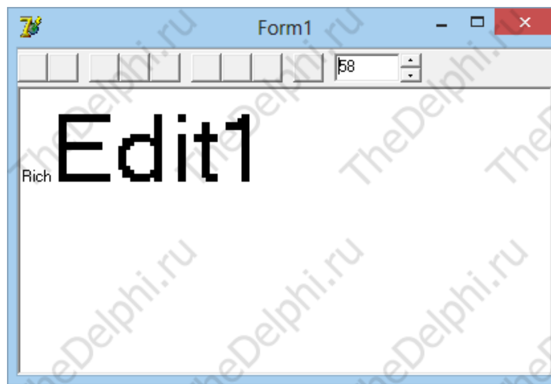
Цвет шрифта, вытаскиваем диалог выбора цвета ColorDialog и делаем кнопку с разделителем. В событии OnClick код:

```
procedure TForm1.ToolButton12Click(Sender: TObject);
begin
 if ColorDialog1.Execute then //Открываем диалог выбора цвета
  RichEdit1.SelAttributes.Color:= ColorDialog1.Color; //Применяем цвет выделенному фрагменту
end;
```



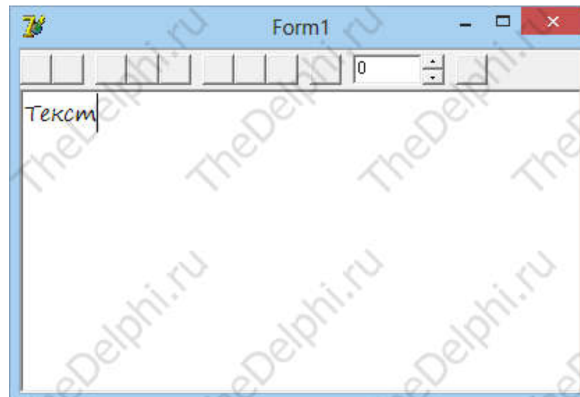
Работает! Теперь научимся изменять размер выделенного текста, опять делаем разделить и вставляем Edit с компонентом UpDown. Свойство Associate в компоненте UpDown установить на Edit1. В обработчике события Edit1 OnChange напишем код:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
 RichEdit1.SelAttributes.Size:=StrToInt(Edit1.Text);  // Изменяем размер
end;
```



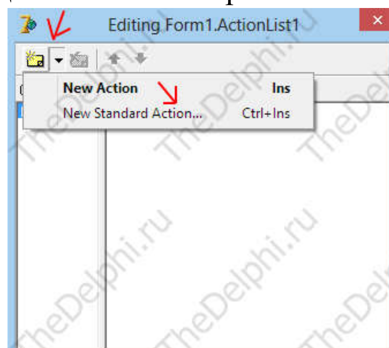
Можно изменять внешний вид шрифта, вытащим FontDialog и создадим еще одну кнопочку, в её событии OnClick напишем:

```
procedure TForm1.ToolButton15Click(Sender: TObject);
begin
  if FontDialog1.Execute then
   RichEdit1.Font:= FontDialog1.Font;
end;
```

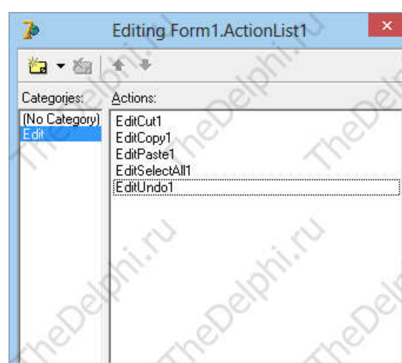


Теперь у нас есть простой редактор текста, но чего-то не хватает. Не хватает стандартных функций вырезки, вставки, копирования и т.д. Урок получился довольно большим и поэтому мы схитрим, не будем писать код, а воспользуемся стандартными функциями, открою вам небольшой секрет, все что мы сейчас сделали можно было не писать, а воспользоваться всё теми же функциями.

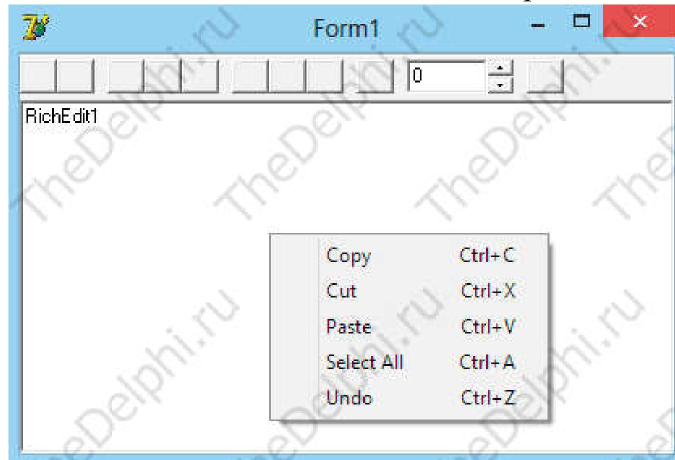
Все что нам надо сделать, это вытащить ActionList и PopupMenu на вкладке Standart. Кликаем два раза на ActionList и дальше как на картинке.



Затем выбираем действие в появившемся меню TEditCut, также добавляем: TEditCopy, TEditPaste, TEditSelectAll, TEditUndo. В конечном итоге у вас должно получиться вот это:



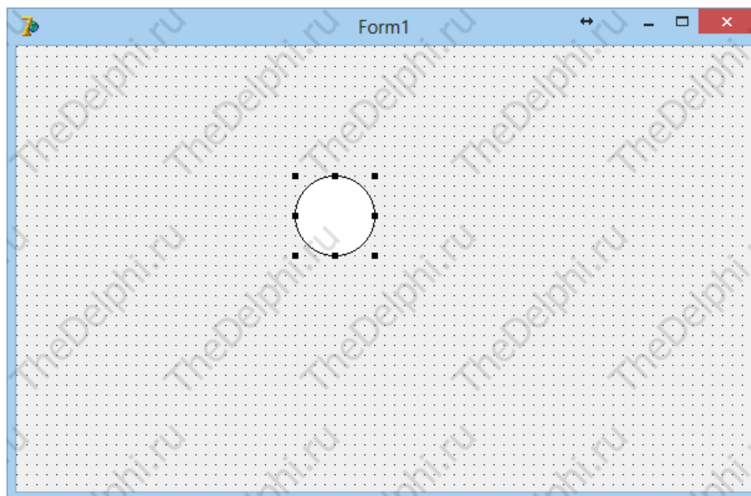
Теперь в PopupMenu создаём 5 пунктов с любым именем, в свежесозданом пункте в свойстве Action выберем наше действие, и Caption задастся сам и так для всех пяти штук, по желанию их можно перевести на русский язык. В конце нужно установить свойство PopupMenu RichEdit'a на PopupMenu1 и все готово можно пользоваться правой кнопкой.



Урок 26 - Создаем игру Ping-pong - часть(1/3)

Начинаем писать простенькую игру Пинг-Понг. Думаю, смысл игры Пинг-Понг давно всем известен. Создание этой игры я разделю на 3 части: простое перемещение и рикошеты от граней формы, возможность отбивать шарик мышкой, искусственный интеллект (Противник).

Итак, все, что нам понадобится - это компонент Shape на вкладке Additional. Вытаскиваем его на форму и меняем свойство Shape на stCircle, у нас вместо квадрата получился кружок.



Для работы нам понадобятся 2 свойства: Left и Top, первое - это позиция по горизонтали в дальнейшем X, второе - по вертикали (Y).

Приступим, создадим 4 глобальных переменных типа Single: PosX, PosY, VelX, VelY. первые 2 переменных - это позиция по X и Y, остальное - скорость по X и Y.

Создадим обработчик события Form1 OnCreate:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 PosX:= 30; // Задаем     позицию
 PosY:= 50; //      начальную
 VelX:= 1;  // Задаем     скорость
 VelY:= 2;  //      начальную
end;
```

Если мы сейчас скомпилируем то ничего не будет, просто зададутся значения, нам нужно обновлять позицию шара через равные промежутки времени, с этой работой хорошо справится таймер. Вытащим его на форму, сразу установим интервал на 1 мс и в обработчике события напишем код:

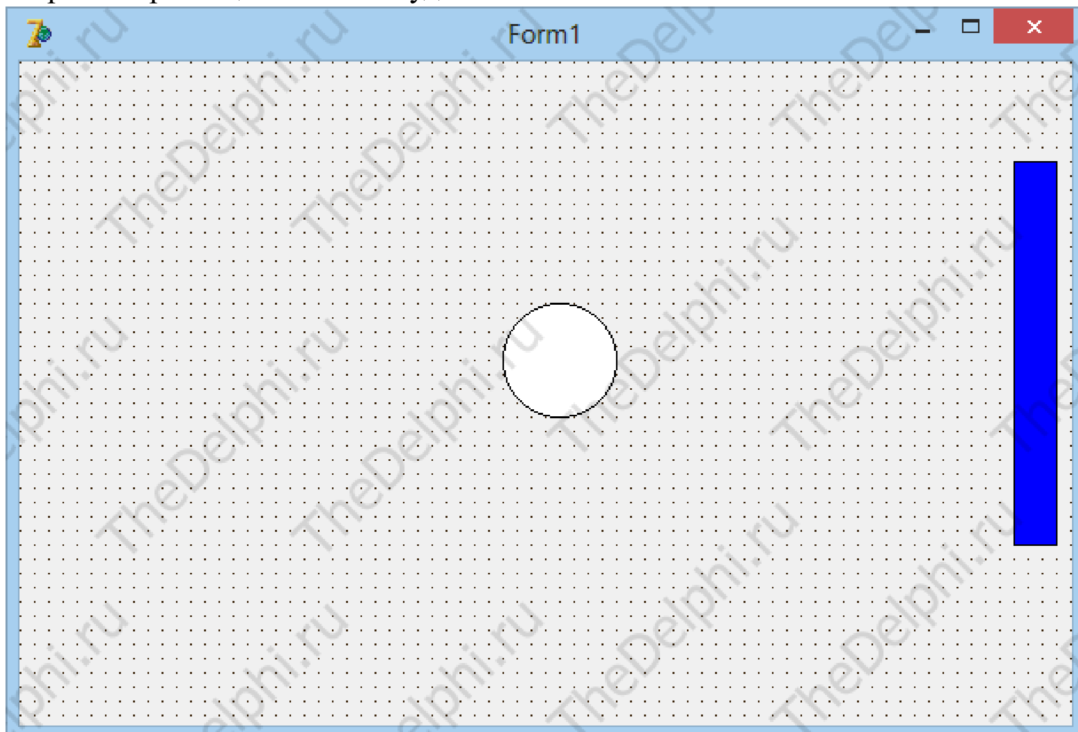
```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  PosX:= PosX + VelX;          // Изменяем
  PosY:= PosY + VelY;          //  позицию
  Shape1.Left:= Round(PosX);  // Обновляем
  Shape1.Top:=  Round(PosY);  // положение шарика
end;
```

Теперь наш шарик двигается, но он выходит за границы формы, модифицируем код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  PosX:= PosX + VelX;           // Изменяем
  PosY:= PosY + VelY;           //  позицию
 if PosX > ClientWidth - Shape1.Width then  //если шарик коснется правого края то направим его в обратную сторону
    VelX:= -VelX;
 if PosY > ClientHeight - Shape1.Height then //если шарик коснется нижнего края то направим его в обратную сторону
    VelY:= -VelY;
  if PosX < 0 then  //если шарик левого правого края то направим его в обратную сторону
    VelX:= -VelX;
  if PosY < 0 then  //если шарик верхнего правого края то направим его в обратную сторону
    VelY:= -VelY;
  Shape1.Left:= Round(PosX);  // Обновляем
  Shape1.Top:=  Round(PosY);  // положение шарика
end;
```

Продолжаем писать игру Пинг-Понг. В этом уроке мы займемся реализацией игрока.

Вытащим еще один Shape и сделаем ему синий цвет. Вот собственно этим мы и будем отбивать шарик. Перемещать его мы будем мышкой.



Для этого создадим событие Form1 OnMouseMove:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  Shape2.Top:= Mouse.CursorPos.Y - Form1.ClientOrigin.Y;
end;
```

Shape2 - это наш прямоугольник, Mouse.CursorPos - позиции курсора, Form1.ClientOrigin - экранные координаты формы. После компиляции прямоугольник двигается вместе с мышкой, но шарик не отбивается.

В Delphi есть такая функция:

- function InterSectRect(var Dst: TRect; const Src1: TRect; const Src2: TRect);

Она определяет пересекаются ли Rect'ы, в нашем случае Shape1 и Shape2. Модифицируем код таймера добавив в конец процедуры вот это:

```
 if InterSectRect(Overlay, Shape2.BoundsRect, Shape1.BoundsRect) then  //если есть пересечение
между прямоугольником и кружком то
  begin                 // кружок отправляем обратно
   VelX:= -VelX;
   VelY:= -VelY;
  end;
```

И здесь нам понадобится локальная переменная Overlay: TRect. Вот что должно получится у вас:

```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  Overlay: TRect;
begin
  PosX:= PosX + VelX;
  PosY:= PosY + VelY;
  if PosX > ClientWidth - Shape1.Width then
  VelX:= -VelX;
  if PosY > ClientHeight - Shape1.Height then
  VelY:= -VelY;
  if PosX < 0 then
  VelX:= -VelX;
  if PosY < 0 then
  VelY:= -VelY;
  Shape1.Left:= Round(PosX);
  Shape1.Top:=  Round(PosY);
   if InterSectRect(Overlay, Shape2.BoundsRect, Shape1.BoundsRect) then
     begin
    VelX:= -VelX;
    VelY:= -VelY;
     end;

end;
```

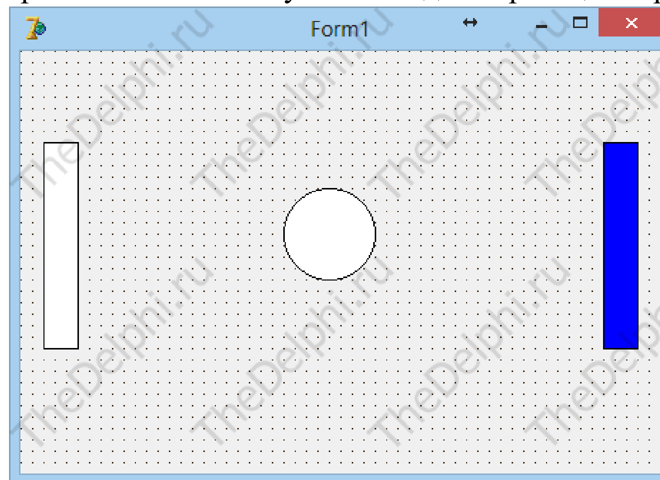
Теперь при столкновении шарик отскакивает! Но он движется по одному и тому же пути, избавимся от этого, добавив немного случайности. В FormCreate допишем событие, добавим в конец слово Randomize, тем самым мы говорим Delphi, что будем использовать улучшенный алгоритм генерации случайных чисел и измени проверку пересечения 2 объектов, добавив Random в OnTimer:

```
if InterSectRect(Overlay, Shape2.BoundsRect, Shape1.BoundsRect) then
begin
 VelX:= -VelX - Random(5);
 VelY:= -VelY - Random(5);
end;
```

Теперь будет лучше. В следующем уроке мы будем делать искусственный интеллект.

Урок 28 - Создаем игру Ping-pong - часть(3/3)

Мы продолжаем простенькую игру "Пинг-Понг". Сделаем искусственный интеллект. Перейдем на вкладку Additional и вытащим третий Shape - это наш противник. Придадим ему ту же форму, что и у нас. Противник должен отбивать шарик и не давать ему ударится о левую границу формы. Но как нам забить противнику, если он постоянно отбивает мяч, неважно с какой скоростью он летит. Нужно замедлить реакцию противника.



Вытащим на форму еще один таймер и сделаем интервал меньше чем у первого таймера, например 16 мс. В обработчике события мы запишем код:

```
procedure TForm1.Timer2Timer(Sender: TObject);
begin
 if Shape3.Top > ClientHeight - Shape3.Height - 40 then  //Не пускаем врага
  Shape3.Top:= Shape3.Top      //за нижний край формы
       else
  begin
        if Shape1.Top > Shape3.Top then // если шарик ниже противника
         Shape3.Top:= Shape1.Top + Shape3.Width; //то отпускаем его
       end;
 if Shape1.Top < Shape3.Top then   // поднимаем если шарик выше
  Shape3.Top:= Shape1.Top - Shape3.Width;
end;
```

Проверим, да, враг ожил, но он не отбивает мяч, нужно в первом таймере дописать проверку столкновения шарика с противником, все что нам нужно это скопировать условие проверяющие столкновение шарика с игроком и заменить Rect игрока на противника и сменить знаки т.к. отскакивать шарик будет в противоположную сторону:

```
if InterSectRect(Overlay, Shape3.BoundsRect, Shape1.BoundsRect) then
begin
 VelX:= -VelX + Random(5);
 VelY:= -VelY + Random(5);
end;
```

Ну вот и все мы написали простенькую игру. Удачи!

Урок 29 - Работа с DLL

В этом уроке мы ознакомимся с динамическими библиотеками Windows. Научимся её использовать и создавать.

Файл динамической библиотеки Windows имеет расширение .dll, вы наверняка сталкивались с такими файлами. Библиотеки имеют ряд плюсов в сравнении с обычным .exe файлом:

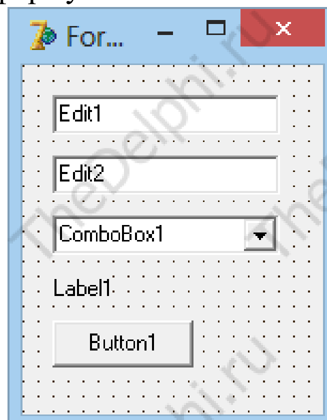
- Экономия системной памяти;
- Можно хранить картинки, файлы и т.д.;
- Быстрое обновление программы.

Например, наша программа работает и использует определённый алгоритм действий, этот алгоритм записан в dll и при обновлении этого алгоритма достаточно заменить библиотеку с этим алгоритмом на новую, а так бы пришлось заново скачивать все программу целиком.

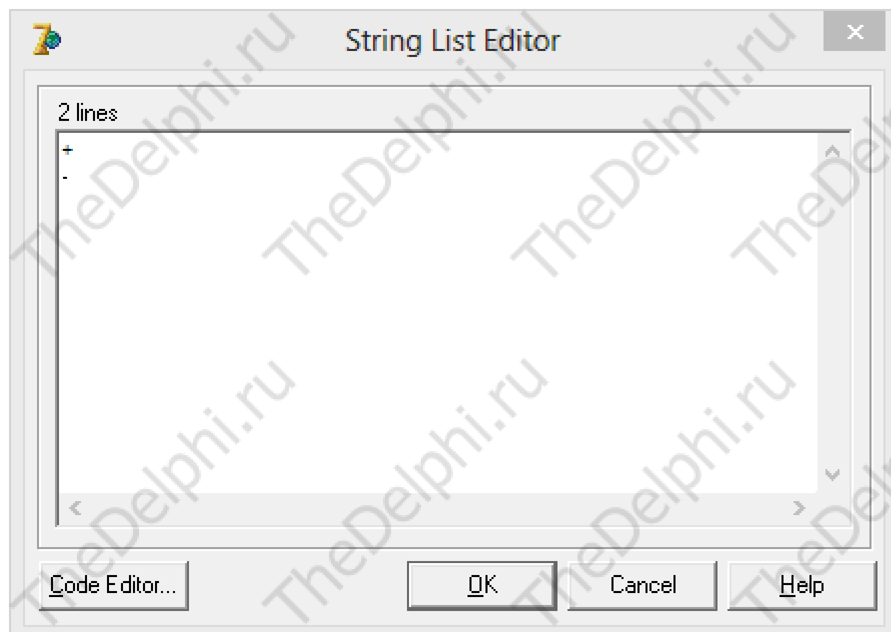
Создается библиотека очень просто, открываем Delphi и выбираем File->New->Other->DLL Wizard и вот у нас появилась область кода, сразу стоит заметить, что нельзя использовать формы и компоненты, только ресурсы, функции и процедуры. Мы будем делать калькулятор, но все действия будут записаны в dll. Итак, определимся с видом нашей функции: **function calc(a, b, index: integer): integer;** a и b это наши числа с которыми необходимо произвести действие, про index я расскажу немножечко позже. Над begin и под {$R *.res} напишем нашу функцию:

```
function calc(a, b, index: integer): integer;
begin
if index = 0 then //если index = 0 то сложим числа
        Result:= a + b;
if index = 1 then //если index = 1 то вычтем из 1 числа 2 число
        Result:= a - b;
end;
exports calc;  //Позволим другим программам использовать эту функцию
```

Теперь скомпилируем библиотеку, но для начала нужно где-нибудь сохранить проект и назовём его "Project2", сохраняем и компилируем... вылезла ошибка. Это говорит о том, что все готово, мы просто компилируем без параметра, в папке куда мы сохранили появилась библиотека "Project2.dll". Все с библиотекой закончили, теперь надо создать сам калькулятор, который будет использовать нашу библиотеку. Создаем новый проект и сохраняем в ту же папку где и dll. Создаем вот такую форму:



Добавим элементы для ComboBox1, кликаем на свойство Items и делаем тоже самое:



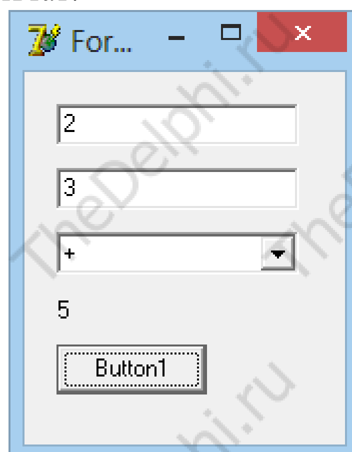
И сразу после implementation объявляем нашу функцию:

```
function calc(a, b, index: integer): integer; external 'Project2.dll';  //Говорим компилятору, что эта функция находится в Project2.dll
```

В событии Button1 OnClick пишем код:

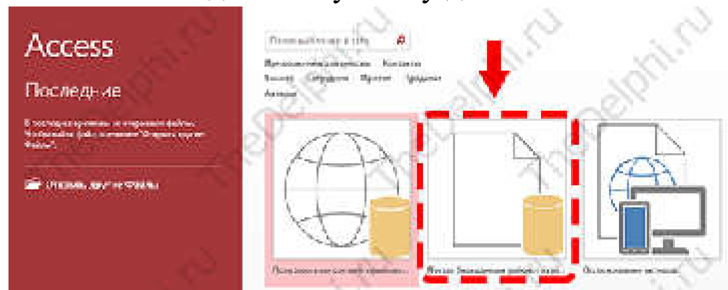
```
Label1.Caption:= IntToStr(calc(StrToInt(Edit1.Text), StrToInt(Edit2.Text), ComboBox1.ItemIndex));
```

Все теперь запускаем программу, вводим числа, выбираем действие и нажимаем на кнопочку, в Label1 выводится результат:

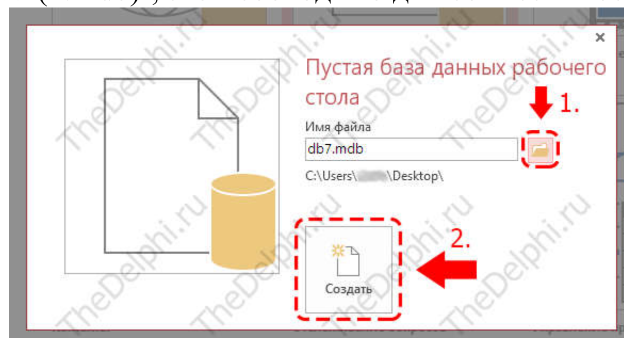


В этом уроке мы создадим простую базу данных. Я буду использовать Microsoft Access 2013. Создавая в современной программе, мы облегчим себе жизнь, потому что создавать стало значительно проще.

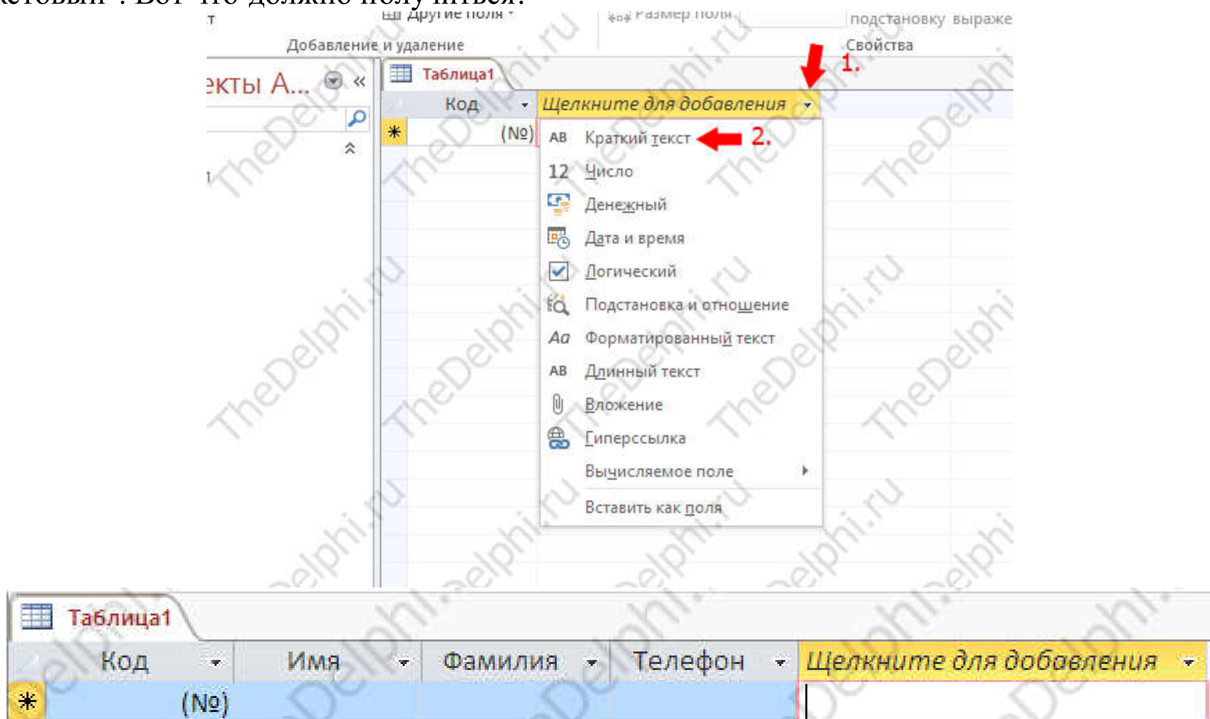
И так открываем Access и создаем новую базу данных:



Далее, сохраняем базу данных, назовём её db7, а тип файла сделаем "Базы данных Microsoft Access 2002-2003 (*.mdb)", это необходимо для совместимости Delphi и Базы данных.



После сохранения создаем таблицу «Сотрудники», с ней нужно провести кое какие действия. Добавляем новый столбец с типом данных "Текстовый", после этих манипуляций появится новый столбец с возможностью задания названия, назовем этот столбец "Имя", точно также создадим второй с названием "Фамилия" и столбик "Телефон" у всех тип данных "Текстовый". Вот что должно получиться:



Ну а теперь остается её заполнить, например вот так:

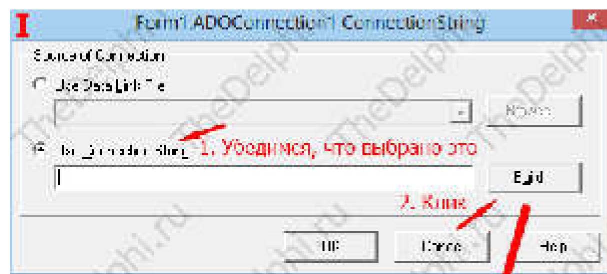
Таблица1

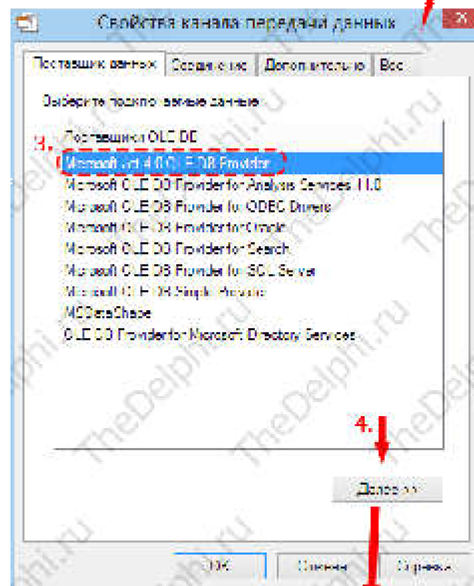
Код	Имя	Фамилия	Телефон	Щелкните для добавления
2	Иван	Петров	79025446154	
3	Петр	Сидоров	79124576321	
4	Михаил	Мухин	78654448846	
* (№)				

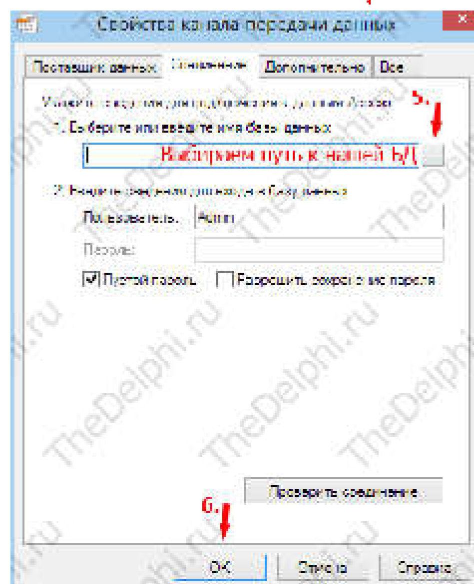
Все, сохраним таблицу, нажав сочетание клавиш Ctrl+S, в появившемся окошке вводим имя таблицы "Сотрудники". Наша таблица готова к использованию, все, что связано с Delphi мы узнаем в следующем уроке.

Урок 31 - Продолжение работы с базами данных

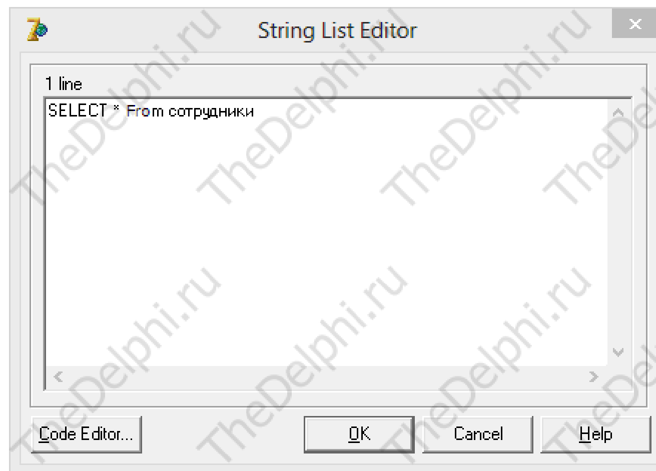
В этом уроке мы откроем нашу ранее созданную базу данных в Delphi. Для того чтобы открыть базу данных нам понадобятся 4 компонента: ADOConnection, ADOQuery с вкладки ADO, DataSource с вкладки Data Access и DBGrid с вкладки Data Controls. Настроим компонент ADOConnection, выбираем свойство ConnectionString, дальнейшие действия изображены на рисунке:



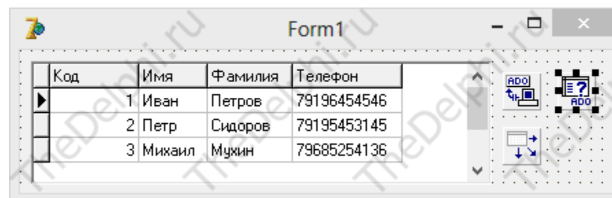




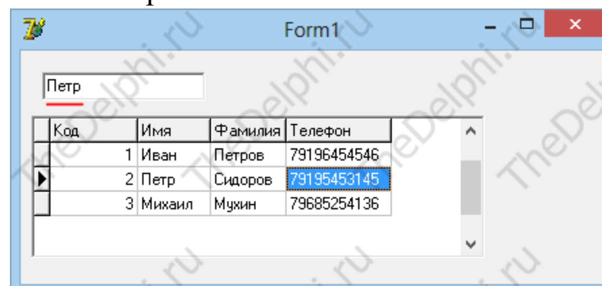
Далее свойство LoginPrompt установить на False, это для того, чтобы не вылезло сообщение о вводе пароля для каждого подключения к БД. ADOConnection мы настроили, теперь перейдём к ADOQuery. Свойство Connection установим на ADOConnection1 и выберем свойство SQL, перед нами появляется менеджер запросов SQL. SQL - это язык запросов к базе данных, для тех, кто незнаком с ним в конце урока будет способ сделать тоже самое, но без SQL запросов. Итак, в менеджере пишем запрос выбора все таблицы базы данных: "SELECT * From сотрудники", мы выбираем все поля для базы данных "сотрудники". Жмем ок.



С ADOQuery готово, теперь осталось DataSource и DBGrid, тут все просто, у DataSource надо установить свойствоDataSet на ADOQuery1, а у DBGrid свойство DataSource на DataSource1. Все, осталось только установить свойство Active в True у ADOQuery, и у нас появилась наша база данных:

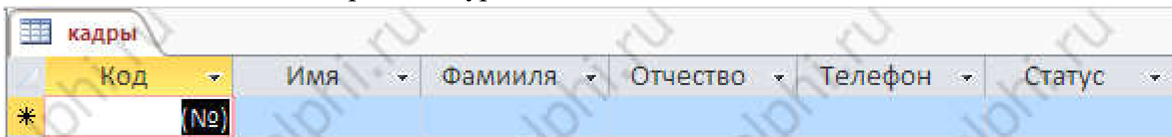


Базу можно легко изменять, все автоматически сохраняется. Также можно привязать какой-нибудь компонент, например DBEdit с вкладки Data Controls, свойство DataSource установить на DataSource1, а в DataField выбрать нужный вам столбец (например "Имя"), тогда при выборе номера из второй строки в DBEdit будет слово из второй строки столбика "Имя", то есть "Петр".

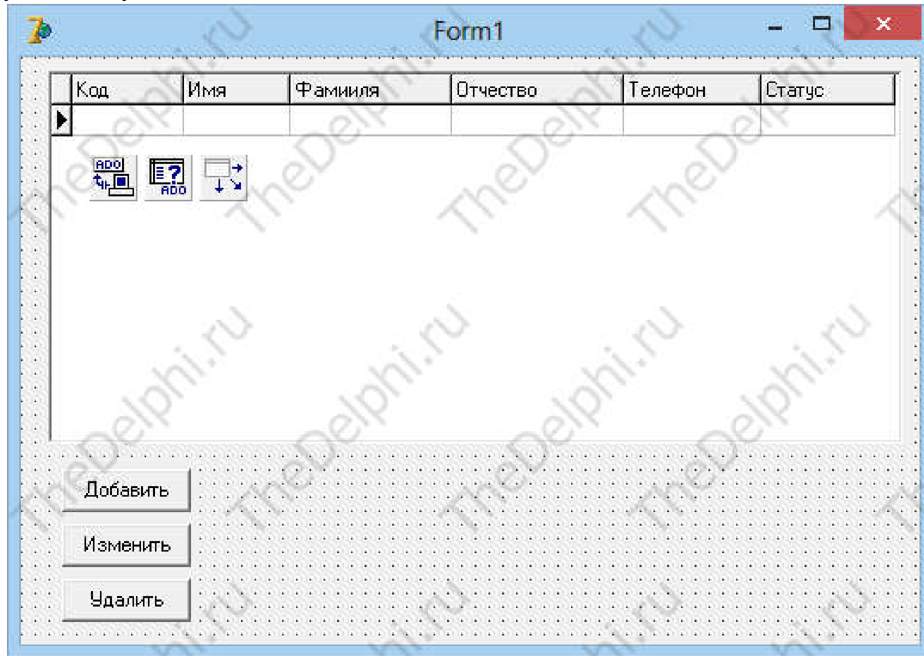


Ну а теперь способ без SQL, нужно заменить ADOQuery на ADOTable и установить свойства Connection и TableName на ADOConnection1 и "Сотрудники" соответственно, у DataSource свойство DataSet на ADOTable1, ну и установить свойство Active в True у ADOTable1. Все тоже самое, но меньше гибкости.

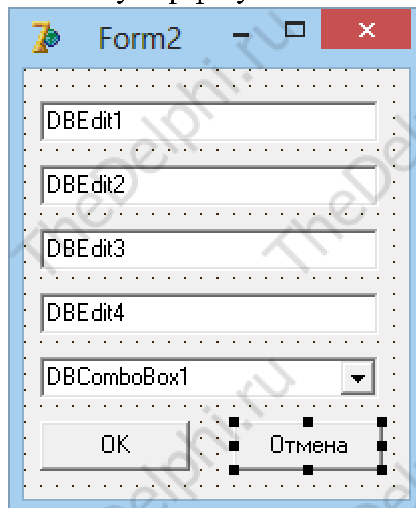
В этом уроке мы объединим все, что прошли ранее. Для начала нам нужно создать пустую таблицу "кадры" с набором полей: "Имя, Фамилия, Отчество, Телефон, Статус". Всё точно также как это делали в прошлом уроке.



Также возьмём за основу пример программы из прошлого урока, добавим 3 кнопки и проверим нашу таблицу:



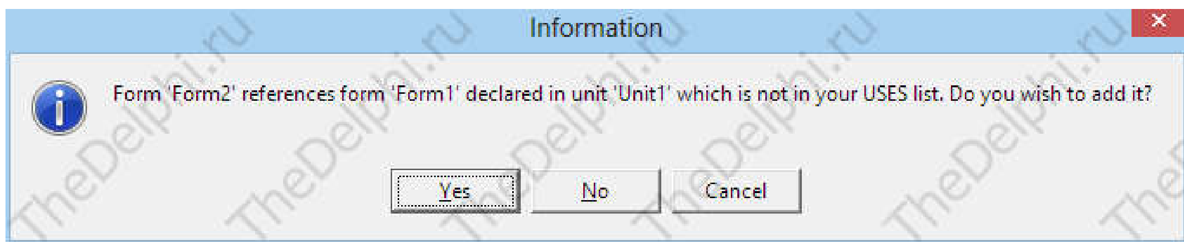
Мы напишем программу для полноценного управления базой данных (добавление, редактирование, удаление). Добавим новую форму и положим на неё следующие компоненты:



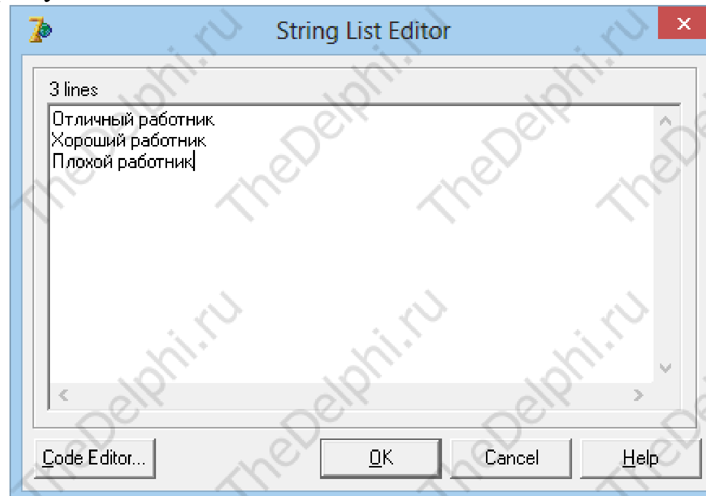
Форма готова и чтоб начать задавать свойства нам нужно подключить 2 юнит к 1 юниту, потому что без этого мы не можем обратиться к первой форме из второй. Делается это очень просто, создаем обработчик события OnClick на 1 кнопке:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
Form1.Show; // Вызываем что-нибудь из 1 формы
end;
```

И у нас вылезет вот такое окно:



Нажимаем **Yes** и первая форма подключается ко второй, можно использовать её свойства. Всем DBEdit'ам и BDComboBox1 свойство DataSource зададим как Form1.DataSource1, а DataField как "Имя", "Фамилия", "Отчество", "Телефон", "Статус" соответственно. Далее настроим еще одно свойство у DBComboBox1, кликаем на Items и заполняем его как на рисунке:



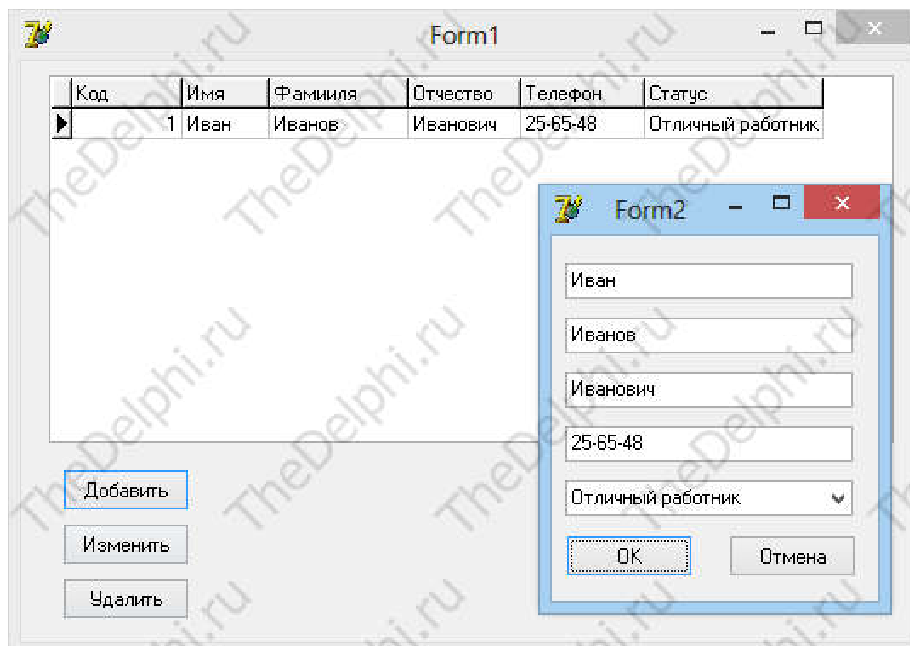
Теперь нам надо изменить наш ранее созданный обработчик события на кнопке "OK", стираем там все что есть и пишем новый код:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
Form1.ADOQuery1.Post; // Фиксируем изменения
Close;  // Закрываем форму
end;
```

И сразу же создадим обработчик события на кнопке "Добавить" на первой форме:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Form2.Show; // Показываем вторую форму
ADOQuery1.Insert; // Добавляем строку
end;
```

У нас опять вылезет сообщение о том что надо добавить первый юнит ко второму, соглашаемся, компилируем и добавляем:



Теперь можно заполнять базу данных. Удаляются значения так, создаем обработчик события кнопки "Удалить" на первой форме:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
ADOQuery1.Delete; // Удаляем элемент
end;
```

Все, выделенная строка исчезает. Также легко можно изменить строку, создаем обработчик события кнопки "Изменить":

```
procedure TForm1.Button2Click(Sender: TObject);
begin
Form2.Show; // Показываем вторую форму
end;
```

У нас просто показывается вторая форма и выделенную строку можно изменять. Теперь представим, что наша база данных состоит из тысячи строки и нам надо найти одну единственную, эта задача решается поиском строки, это мы сейчас и сделаем. Кидаем на форму компонент StatusBar и 2 раза кликаем на неё, в открывшемся окне создаем новую панель и закрываем. Создаем глобальную переменную **fs: String** и в обработчике события OnKeyPress DBGrid1 пишем код:

```
procedure TForm1.DBGrid1KeyPress(Sender: TObject; var Key: Char);
begin
fs:= fs + Key; // Прибавляем символ нажатой клавиши к искомому слову
DBGrid1.DataSource.DataSet.Locate('Имя', fs, [loPartialKey]); // Выполняем поиск в столбце
"Имя"
StatusBar1.Panels.Items[0].Text:= 'Ищем: ' + fs; // выводим что мы ввели
end;
```

И что бы во время печатания слова мы случайно не изменили значение в таблице надо установит свойствоdgRowSelect у DBGrid1, которое находится в списке Options на True. Все поиск у нас есть, давайте его опробуем, запускаем программу и вводим "ми":

И выделяется "Мигель", Вводим "мих", выделяется "Михаил".

Для упрощения процесса составления отчетов в поставку всех версий Delphi входят специальные программные средства - генераторы отчетов. В первых версиях Delphi это был ReportSmith, затем, с Delphi 3 по Delphi 6 - QuickReport, а начиная с Delphi 7 и заканчивая Delphi 2006 - Rave Reports.

ПРИМЕЧАНИЕ

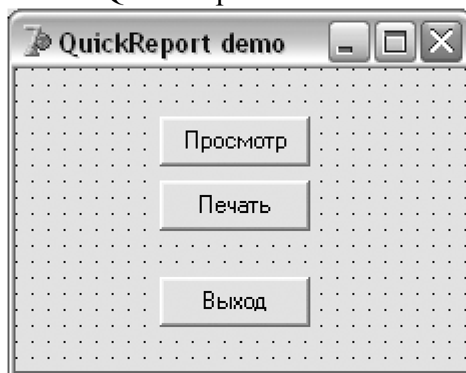
В поставку Delphi 7 так же входит и QuickReport, однако по умолчанию этот инструментарий отсутствует на панели компонентов. Чтобы воспользоваться QuickReport в этой версии Delphi, следует открыть Component - Install Packages, нажать кнопку Add и выбрать файл **dclqrt70.bpl** в подкаталоге ProgramFiles\Borland\Delphi7\Bin.

Генераторы ReportSmith и Rave Reports представляют собой отдельные приложения, при помощи которых можно создавать отчеты, в то время, как QuickReports - это набор VCL-компонентов, которые помещают непосредственно на стандартную форму Delphi.

В простейшем случае работа с QuickReport выглядит следующим образом: создается новая форма, на которую помещают компонент QuickRep. Затем при помощи составного свойства Bands определяют, какие основные составные части отчета требуются:
- HasColumnHeader - заголовки столбцов;
- HasDetail - сами столбцы (собственно данные);
- HasPageFooter - нижний колонтитул страницы;
- HasPageHeader - верхний колонтитул страницы;
- HasSummary - итоговое значение по столбцам данных;
- HasTitle - общий заголовок отчета.

В типичном случае, как минимум, необходимы столбцы и их заголовки. Общий заголовок так же обычно необходим. Кроме того, для потенциально многостраничных отчетов, как правило, предусматривают нумерацию страниц, для чего пригодится тот или иной колонтитул.

Рассмотрим простейшее приложение, использующее отчет. Для этого создадим новый проект в Delphi (если у вас Delphi 7, то сначала подключите QuickReport, как описано в примечании). Затем на главную форму поместите 3 кнопки - "Просмотр", "Печать" и "Выход", а в заголовке окна (Caption) напишите "QuickReport demo"



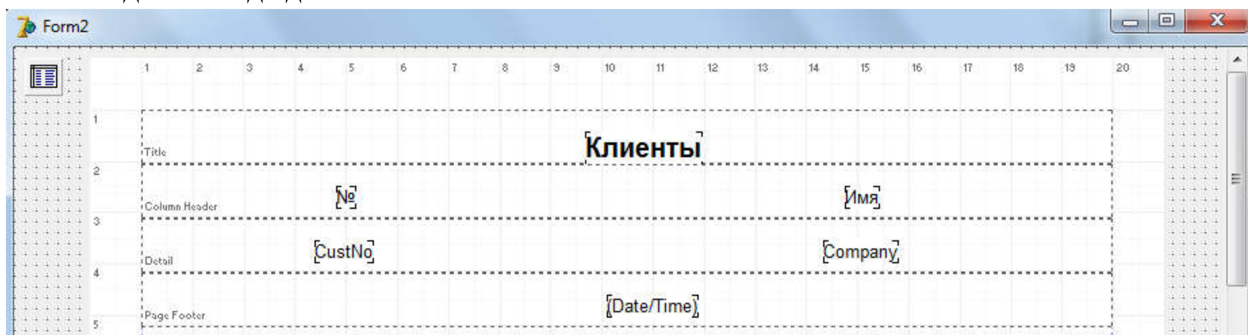
Теперь создадим новую форму, поместим на нее компонент Table, установим свойство DatabaseName в DBDEMOS, а для TableName выберем Customer, после чего установим свойство Active в истину. Таким образом, мы подготовили данные для вывода. Теперь поместим на форму компонент QuickRep, который визуально предствляет собой лист отчета и установим в свойстве Bands флаги HasColumnHeader, HasDetail, HasPageFooter и HasTitle. В результате на "листе" появятся 4 области, представляющие собой соответствующие логические части отчета. Теперь в центре самой верхней области - Title - поместим компонент QRLabel. Это аналог стандартной подписи (Label), предназначенный специально для отчетов QuickReport. В свойстве Caption напишем "Клиенты", после чего можно увеличить размер шрифта, сделать его полужирным или выбрать другую гарнитуру, чтобы заголовок не "затерялся".

Следующим этапом будет собственно подготовка к выводу данных отчета. Поскольку мы выбрали таблицу клиентов, то выводить будем 2 колонки - номер клиента в базе (поле

CustNo) и его имя (Company). Для этого сначала на области заголовков столбцов (Column Header) разместим еще 2 надписи, воспользовавшись QRLabel, в свойстве Caption которых напишем "№" и "Имя". В следующей области, Detail, непосредственно под этими надписями, поместим 2 компонента QRDBText. Эти компоненты, в свою очередь, можно назвать аналогами компонент DBText, предназначенных для вывода данных из БД в строковом формате. Свойство DataSet компонентов QRDBText следует установить в Table1, а DataField - в CustNo у одного и в Company - у другого. Здесь надо отметить, что компоненты, помещенные в области Detail, при печати будут повторяться построчно столько раз, сколько требуется для вывода всех данных из указанного источника. Но следует учитывать, что и у компонентов, используемых для вывода полей, и у основы отчета - компонента QuickRep, должен использоваться один и тот же источник данных. В данном случае это Table1.

Последнее, что остается - это определить содержимое нижнего колонтитула. Для этого в области Page Footer поместим компонент QRSysData. Прямого аналога в VCL у этого компонента нет, можно назвать его "продвинутой" версией текстовой подписи, которая может менять свое содержимое в зависимости от состояния отчета. За тип выводимой информации отвечает свойство Data, которое может принимать одно из следующих значений: qrsDare, qrsDateTime, qrsDetailCount, qrsDetailNo, qrsPageNumber, qrsReportTitle и qrsTime. Соответственно, будет выводиться текущая дата, дата и время, количество записей на странице, номер последней записи на странице, номер страницы, название отчета, или текущее время. Поскольку записей в таблице у нас весьма немного, то остановимся на втором варианте - выводе даты и времени составления отчета. Впрочем, при необходимости всегда можно использовать несколько таких компонентов, чтобы вывести всю необходимую информацию.

В результате, после всех действий по настройке отчета, мы получим заготовку, представляющую собой изображение листа с заданными нами подписями и намеченными местами для вывода данных.



Теперь остается написать код для обращения к этому отчету. Для этого вернемся к первой (главной) форме приложения, в начале секции implementation подключим форму с запросом при помощи ключевого слова uses и создадим обработчики событий для кнопок, как показано ниже.

Исходный код приложения QuickReport demo:

```
unit Unit1; interface uses Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls; type TForm1 = class(TForm) Button1: TButton; Button2: TButton; Button3: TButton; procedure Button1Click(Sender: TObject); procedure Button2Click(Sender: TObject); end; procedure Button3Click(Sender: TObject); var Form1: TForm1; implementation uses unit2; {$R *.dfm} procedure TForm1.Button1Click(Sender: TObject); begin Form2.QuickRep1.Preview; end; procedure TForm1.Button2Click(Sender: TObject); begin Form2.QuickRep1.Print; end; procedure TForm1.Button3Click(Sender: TObject); begin close; end; end.
```

Как видно из кода, вся программная работа заключается в вызове метода Preview для просмотра отчета и метода Print для вывода на печать.

Урок 34 - Пример создания отчета с использованием Rave Reports

Рассмотрим пример создания простейшего отчета с использованием Rave Reports. Пусть он будет выводить ту же самую таблицу и примерно в таком же виде, как и рассмотренный в начале этой главы отчет, созданный на базе VCL-компонентов QuickReport.

Для начала нам потребуется главная форма приложения, подобная той, что была использована в примере с QuickReport. Этой формой и ограничимся, поскольку в данном случае форма отчета создается не средствами VCL-компонент в IDE Delphi, а в специализированной среде Rave Reports. Поэтому единственный компонент доступа к данным - Table - мы так же поместим на этой форме. При этом, как и в предыдущем случае, установим свойство DatabaseName в DBDEMOS, TableName - в Customer, а Active - в истину.

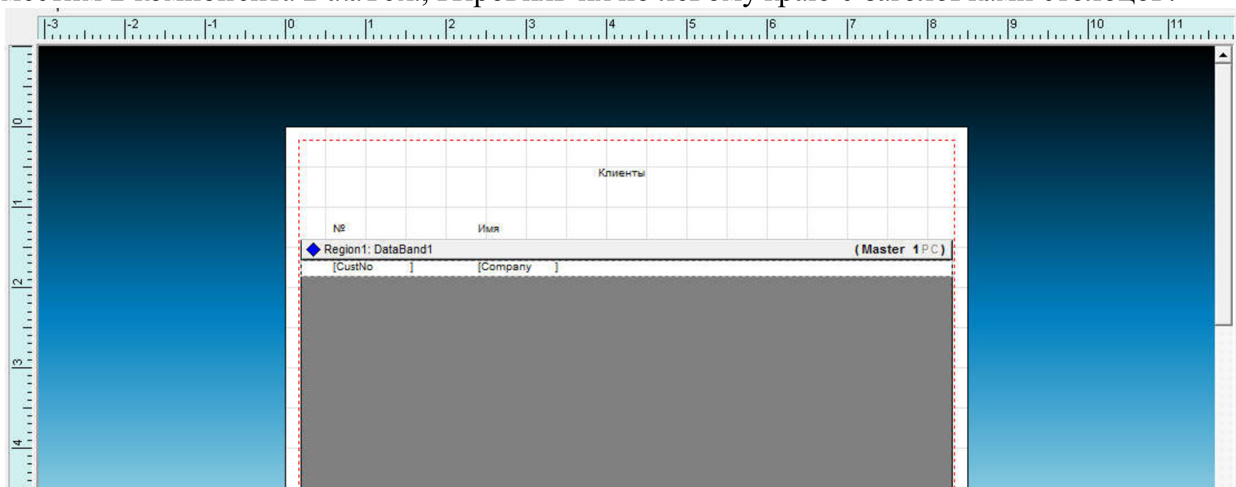
Теперь перейдем к компонентам, расположенным на закладке Rave. Из них нам понадобятся RvProject - основной компонент отчета, RvSystem - для генерации отчетов и вывода их на принтер или в окно просмотра, а так же RvDataSetConnection - для взаимодействия с источником данных, т.е. с компонентом Table. Соответственно, следует связать компонент RvDataSetConnection с Table, установив у него в свойстве DataSet значение Table1. Что касается компонента RvProject, то его следует связать с компонентом RvSystem, для чего в свойстве Engine следует установить значение RvSystem1.

После этого можно приступить собственно к проектированию отчета, для чего нам понадобится загрузить среду Rave Reports. Сделать это можно как из меню Tools, так и при помощи двойного щелчка мышкой по компоненту RvProject.

В среде Rave Reports следует начать новый проект, выбрав пункт New из меню File. При этом автоматически создастся новый проект, уже содержащий пустой одностраничный отчет. Для начала подготовим заголовок страницы отчета, для чего вверху страницы, по центру, расположим текстовую надпись (компонент Text расположен на вкладке Standard), в свойстве Text, при помощи инспектора компонентов, укажем значение "Клиенты". Чтобы выделить эту надпись шрифтом, можно перейти на закладку Fonts и настроить вид надписи при помощи имеющихся на ней элементов управления. Затем наметим заголовки столбцов, для чего несколько ниже основного заголовка разместим еще 2 компонента Text, один из которых будет являться заголовком для колонки с номерами клиентов в базе, а другой - для имен.

Таким образом, можно приступать к подготовке к выводу собственно данных. Но, прежде всего, нам потребуется определить источник данных, для чего следует создать объект данных (File ' New Data Object). В окне диалога Data Connections следует выбрать вариант Direct Data View и нажать на кнопку Next, после чего появится второй экран этого диалога со списком доступных соединений. Впрочем, в данном случае, список будет состоять всего из 1 элемента - RvDataSetConnection1, который мы и выберем. После нажатия на кнопку Finish, новый источник данных появится в дереве проекта под именем DataView1.

После всей проделанной подготовительной работы займемся выводом данных, для чего нам понадобятся компоненты отчета, расположенные на закладке Report. Прежде всего, поместим на лист компонент Region и при помощи мышки растянем его таким образом, чтобы он занял практически все свободное место на листе, поскольку именно размеры этого компонента определяют область, в которую будут выводиться данные. Затем, непосредственно на этот компонент, поместим другой - DataBand. На компонент DataBand, в свою очередь, поместим 2 компонента DataText, выровняв их по левому краю с заголовками столбцов.



Теперь остается связать Data-компоненты с источником данных, для чего как для компонента DataBand, так и для обоих компонентов DataText следует установить свойство DataView в DataView1. Для компонентов DataText помимо этого следует выбрать еще и поля, которые они будут выводить, что определяется при помощи свойства DataField. Соответственно для левого компонента это будет CustNo, а для правого - Company.

Фактически, разработку отчета можно считать завершенной, и можно сохранить проект, причем желательно в том же каталоге, что и приложение Delphi, которое данный отчет использует. После сохранения можно закрыть среду Rave Reports и вернуться к разработке в Delphi.

Прежде всего, нам следует указать имя файла проекта в свойстве ProjectFile компонента RvProject. В том случае, если проект был сохранен в том же каталоге, что и это приложение, то достаточно просто ввести имя файла. В противном случае потребуется указать полный путь, для чего можно нажать кнопку с многоточием напротив этого поля в инспекторе объектов и воспользоваться стандартным диалогом открытия файла. Кроме того, поскольку с одной стороны, при выполнении запроса на вывод отчета компонент RvSystem запрашивает, куда его выводить, а с другой у нас и так имеется 2 отдельных кнопки - для вывода на экран и на принтер, то в свойстве SystemSetups этого компонента следует выключить флаг ssAllowSetup.

Вся подготовительная работа, связанная с созданием отчета и настройке компонент на этом заканчивается и можно приступать к написанию программного кода, которого, собственно, будет весьма немного. В частности, нам надо лишь указать на то, куда выводить отчет и собственно обеспечить вызов процедуры вывода. Делается это двумя строками кода:

RvSystem1.DefaultDest:=rdPreview; RvProject1.Execute;

Здесь в первой строке указывается, что вывод следует производить в окно просмотра на экран монитора, а во второй обеспечивается сам вывод. Для вывода же на печать следует лишь изменить значение, назначаемое свойством DefaultDest с rdPreview на rdPrinter:

RvSystem1.DefaultDest:=rdPrinter; RvProject1.Execute;

Вместе с тем, следует учитывать, что для вывода из проекта будет браться первый попавшийся отчет, поэтому если отчетов будет несколько, то всякий раз следует указывать, какой именно отчет нам требуется. Сделать это можно при помощи метода SelectReport:

RvProject1.SelectReport('Report1',true);

Кроме того, не было бы лишним явно загрузить отчет, для чего можно воспользоваться методом Open:

RvProject1.Open;

В результате, мы получим код, приведенный в листинге.

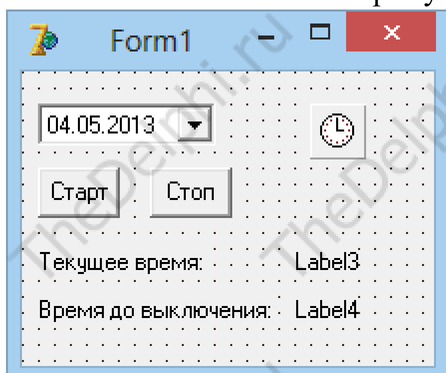
Листинг. Код приложения Rave Reports demo:

```
procedure TForm1.FormCreate(Sender: TObject); begin RvProject1.SelectReport('Report1',true); RvProject1.Open; end; procedure TForm1.Button1Click(Sender: TObject); begin RvSystem1.DefaultDest:=rdPreview; RvProject1.Execute; end; procedure TForm1.Button2Click(Sender: TObject); begin RvSystem1.DefaultDest:=rdPrinter; RvProject1.Execute; end; procedure TForm1.Button3Click(Sender: TObject); begin close; end; end.
```

Урок 35 - Автовыключатель компьютера

В этом уроке мы создадим программу под названием "рубильник", она будет выключать наш компьютер по расписанию.

Итак, нам потребуется: 4 Label, 2 Button, DateTimePicker (Win32) и Timer. Располагаем эти компоненты так как на рисунке:



Определимся со свойствами, свойство Kind у DateTimePicker установим в dtkTime, и свойство Enabled у Timer выставляем в False.

Для выключения компьютера служит функция ExitWindowsEx();, но если её написать вот так просто, то ничего не произойдет, она так работала только в Windows98, в более современных системах она работает только когда есть особые привилегия у использующей её программы. Как сделать эти привилегия мы сейчас и узнаем. Для начала создадим процедуру выключения, в разделе Private Form1 объявим процедуру:

```
...
  private
    { Private declarations }
    Procedure PowerOFF;
...
```

Нажмем комбинацию клавиш Shift + Ctrl + C и Delphi сам создаст шаблон для процедуры, его нужно наполнить кодом:

```
procedure TForm1.PowerOFF;
var
  TTokenHd: THandle;
  TTokenPvg: TTokenPrivileges;
  cbtpPrevious: DWORD;
  rTTokenPvg: TTokenPrivileges;
  pcbtpPreviousRequired: DWORD;
  tpResult: Boolean;
const
  SE_SHUTDOWN_NAME = 'SeShutdownPrivilege';
begin
//=========== Получаем привилегии ==============//
begin
  if Win32Platform = VER_PLATFORM_WIN32_NT then
  begin
    tpResult := OpenProcessToken(GetCurrentProcess(),
      TOKEN_ADJUST_PRIVILEGES or TOKEN_QUERY,
      TTokenHd);
    if tpResult then
    begin
      tpResult := LookupPrivilegeValue(nil,
                                SE_SHUTDOWN_NAME,
                                TTokenPvg.Privileges[0].Luid);
      TTokenPvg.PrivilegeCount := 1;
      TTokenPvg.Privileges[0].Attributes := SE_PRIVILEGE_ENABLED;
```

```
      cbtpPrevious := SizeOf(rTTokenPvg);
      pcbtpPreviousRequired := 0;
      if tpResult then
        Windows.AdjustTokenPrivileges(TTokenHd,
                                      False,
                                      TTokenPvg,
                                      cbtpPrevious,
                                      rTTokenPvg,
                                      pcbtpPreviousRequired);
    end;
  end;
//===============================================//
  ExitWindowsEx(EWX_SHUTDOWN or EWX_POWEROFF, 0); //Выключаем компьютер;
end;
```

Все процедура выключения готова, получилась она довольно громадной, зато везде работает. Теперь займемся таймером, создаем его обработчик события и пишем код:

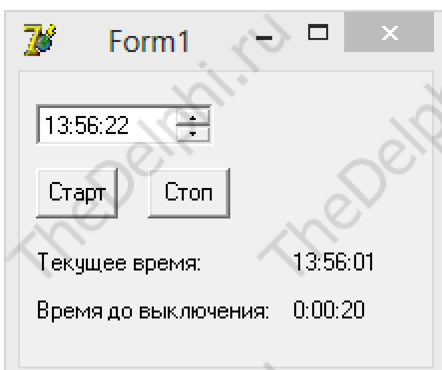
```
procedure TForm1.Timer1Timer(Sender: TObject);
var
  a,b: String;
begin
  Label3.Caption:= TimeToStr(GetTime);  //Получаем текущее время
  Label4.Caption:= TimeToStr(DateTimePicker1.Time - GetTime); //Вычисляем сколько времени осталось до выключения

  a:=  TimeToStr(GetTime); //Присваиваем текущее время
  b:= TimeToStr(DateTimePicker1.Time); //Присваиваем время выключения

  if a = b then  //Если текущее время равно времени выключения то
          PowerOFF;      //выполняем процедуру выключения
end;
```

Помните в начале урока мы выключили таймер, сделано это для того, чтобы можно было сначала настроить время а потом запустить таймер. Давайте сделаем управление таймером. Создадим обработчик события OnClick на кнопке "Старт":

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 Timer1.Enabled:=True; //Включаем таймер
end;
```

Ну и также выключим ,если например ошиблись во времени или передумали, в обработчике на кнопке "Стоп":

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Timer1.Enabled:=False;
end;
```



Вот у нас получилась такая программа, осталось изменить свойство Caption у Form1 на "Рубильник" и готово! Будьте осторожней с ней :)

Урок 36 - Шифрование информации

В этом уроке мы научимся зашифровывать и расшифровывать текст. Рассмотрим простейший пример с использованием стандартных компонентов Delphi7.

Итак, все что нам нужно это 2 кнопочки, 2 memo и еще 2 компонента которые и будут шифровать наш текст, о них по подробнее: 1 компонент - IdEncoderXXE - зашифровывает информацию, 2 компонент - IdDecoderXXE - расшифровывает информацию, эти компоненты находятся на вкладке Indy Misc, кагже там есть аналогичные:

Для зашифровки:
IdEncoderQuotedPrintable
IdEncoderUUE
IdEncoderMIME
Для расшифровки:
IdDecoderQuotedPrintable
IdDecoderUUE
IdDecoderMIME

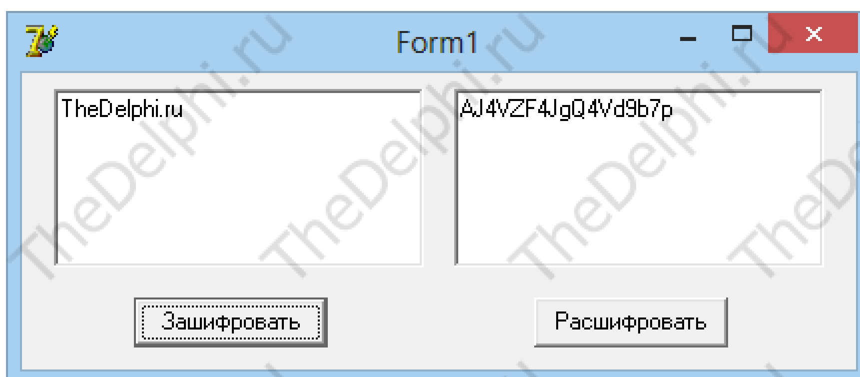
Их устройство одинокого, поэтому мы рассмотрим только один.

Наша программы будет работать следующим образом, при нажатии на первую кнопку мы зашифруем текст в memo1 и выведем шифр в memo2, а при нажатии на вторую кнопку расшифруем из memo2 и пометим слово в memo1. Приступим, создаем обработчик события OnClick на 1 кнопке:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Memo2.Clear;  //Чистим memo2
  Memo2.Text:= IdEncoderXXE1.Encode(Memo1.Text); //Шифруем ...
end;
```

Ну и также расшифруем, с обработчике события OnClick 2 кнопки пишем:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Memo1.Clear;  //Чистим memo1
  Memo1.Text:= IdDecoderXXE1.DecodeString(Memo2.Text); //Расшифровываем ..
end;
```



Также можно шифровать целые файлы.

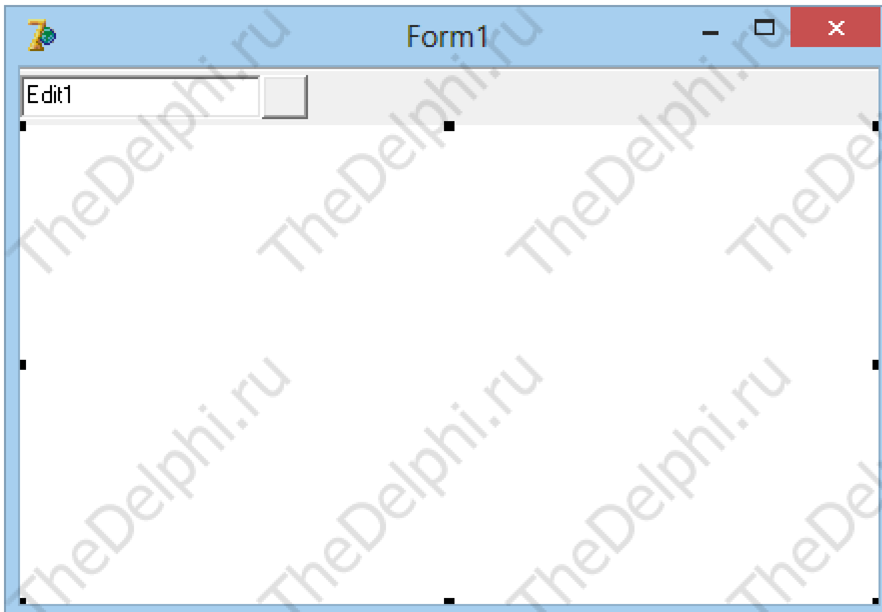
Урок 37 - Создаем Веб браузер

В этом уроке мы напишем свой интернет браузер на основе Internet Explorer.

Нам понадобится ToolBar со вкладки Win32 и WebBrouser на вкладке Internet, кидаем все это на форму и сразу изменим свойство Align на alClient.

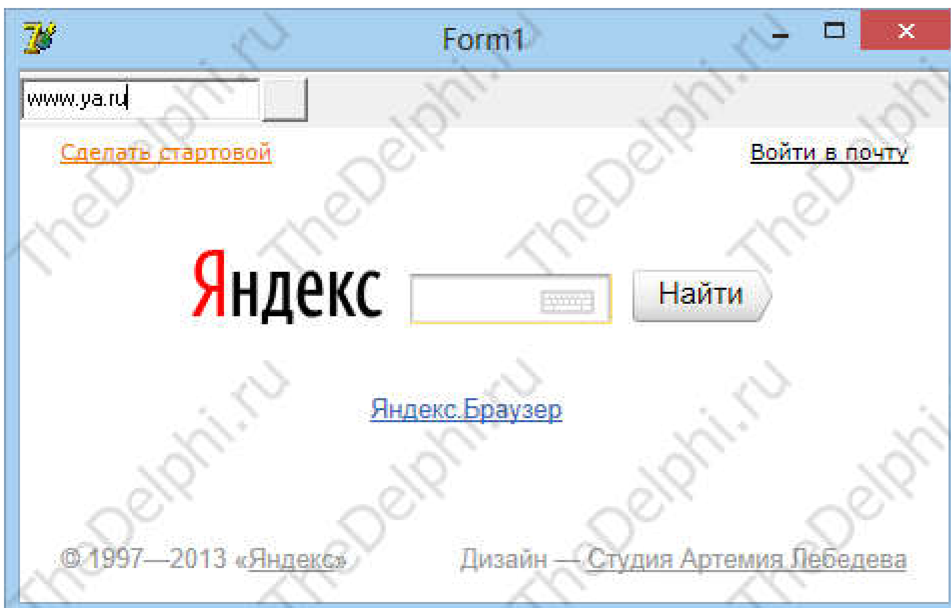
Далее нам понадобится адресная строка и кнопка перехода, кидаем Edit на ToolBar и добавляем кнопку, кликнув правой кнопкой по ToolBar и выбрав New Button. Располагаем ее

вот так:



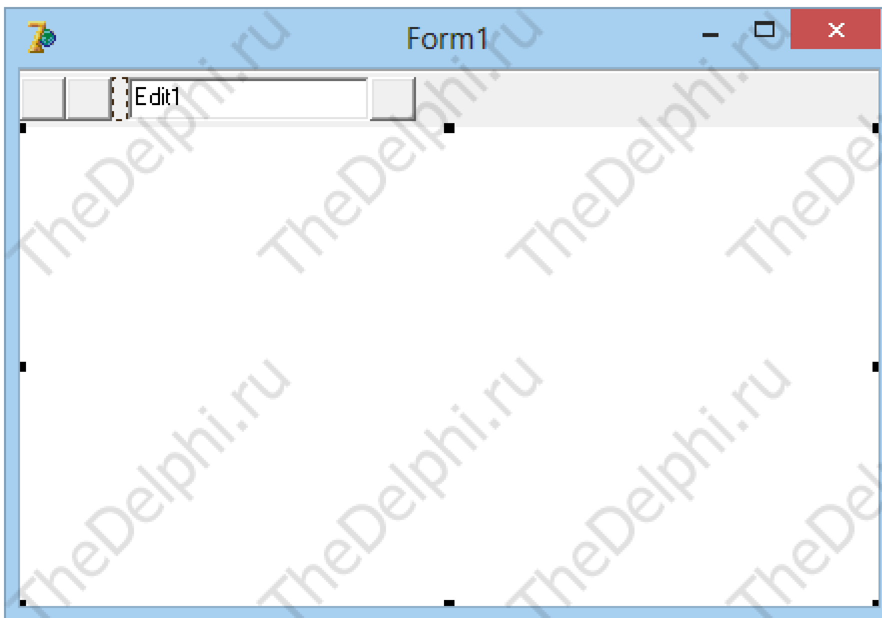
Создаем обработчик события (кликам 2 раза на кнопочку) и пишем код:

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
  WebBrowser1.Navigate(Edit1.Text); //Переходим на новую страницу
end;
```



Вот загрузилась страница, все отображается правильно, попробуйте открыть какую-нибуть большую, сложную страницу, будет полный кавардак. Дело в том что Delphi7 выпущен в 2002 году и в те времена не было HTML5, CSS3 и прочих новшеств, компонент их не понимает. Самое плохое то, что он не обновляется в новых версиях Delphi.

Теперь добавим возможность возврата на предыдущую страницу и вперёд. Добавляем разделитель (New Separator) и 2 кнопочки:



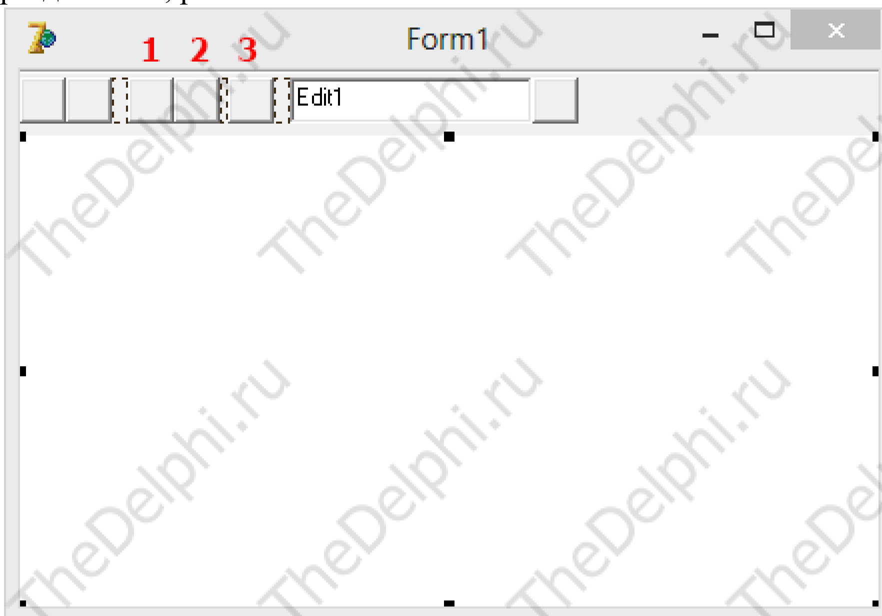
В обработчике события первой кнопки пишем:

```
procedure TForm1.ToolButton2Click(Sender: TObject);
begin
 WebBrowser1.GoBack; //Назад
end;
```

В обработчике события второй:

```
procedure TForm1.ToolButton3Click(Sender: TObject);
begin
 WebBrowser1.GoForward; //Вперёд
end;
```

Ну и добавим кнопочки "Обновить", "Стоп", "Домой". Создадим еще 3 кнопки и 2 разделителя, разместим все вот так:



В первой пишем:

```
procedure TForm1.ToolButton4Click(Sender: TObject);
begin
 WebBrowser1.Refresh; //Обновить
end;
```

Во второй:

```
procedure TForm1.ToolButton54Click(Sender: TObject);
begin
 WebBrowser1.Stop; //Стоп
end;
```

В третьей:

```
procedure TForm1.ToolButton6Click(Sender: TObject);
begin
  WebBrowser1.GoHome; //Домой
end;
```

Ну вот и все, этот компонент больше бы подошёл, например для отображения новостного блока в ваших программах, для более глобальных задач больше подойдет новый компонент TChromium, его нет в числе стандартных компонентов, он устанавливается отдельно и работа с ним выходит за рамки данной статьи.

Урок 38 - Взаимодействие с веб страницей

В этом уроке мы попробуем авторизоваться на сайте yandex.ru.

Мы программно отыщем поля для ввода логина и пароля и нажмем на кнопку входа. Нам понадобится компонент WedBrouser1 и 2 кнопки, кидаем все это на форму.

На первой кнопке сделаем загрузку сайта:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  WebBrowser1.Navigate('www.yandex.ru'); //Открываем яндекс
end;
```

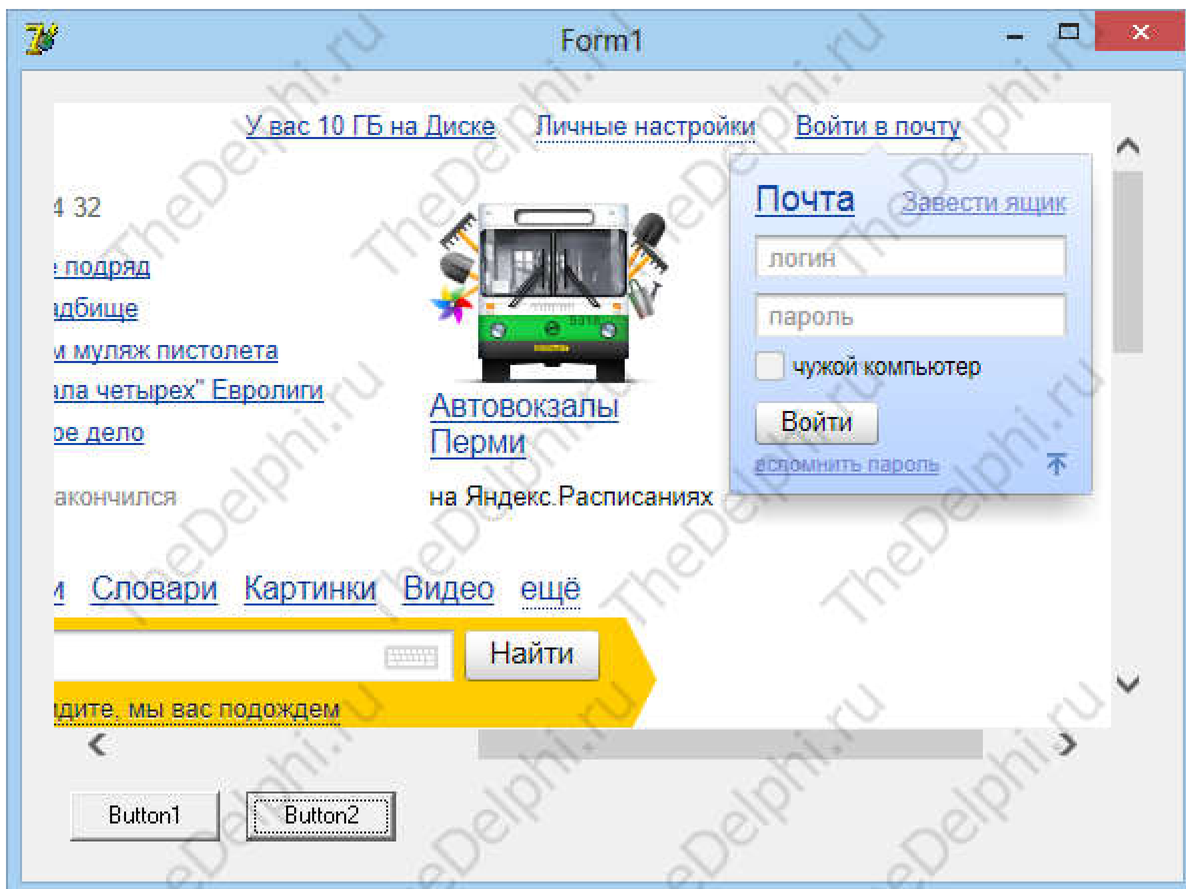
Наша программа будет сканировать страницу на наличие полей и когда имя поля совпадет с искомым она подставит в него текст. Откроем исходный код страницы, и что мы видим, кашу, из всей этой каши нас интересует всего 3 строчки:

Поле для логина:
<input class="b-form-input__input" name="login" id="id001" value="" tabindex="3" />
Поле для пароля:
<input class="b-form-input__input" name="passwd" value="" id="id002" tabindex="4" type="password"/>
Кнопка "Войти":
<input type="submit" hidefocus="true" tabindex="6" class="b-form-button__input" value="Войти">

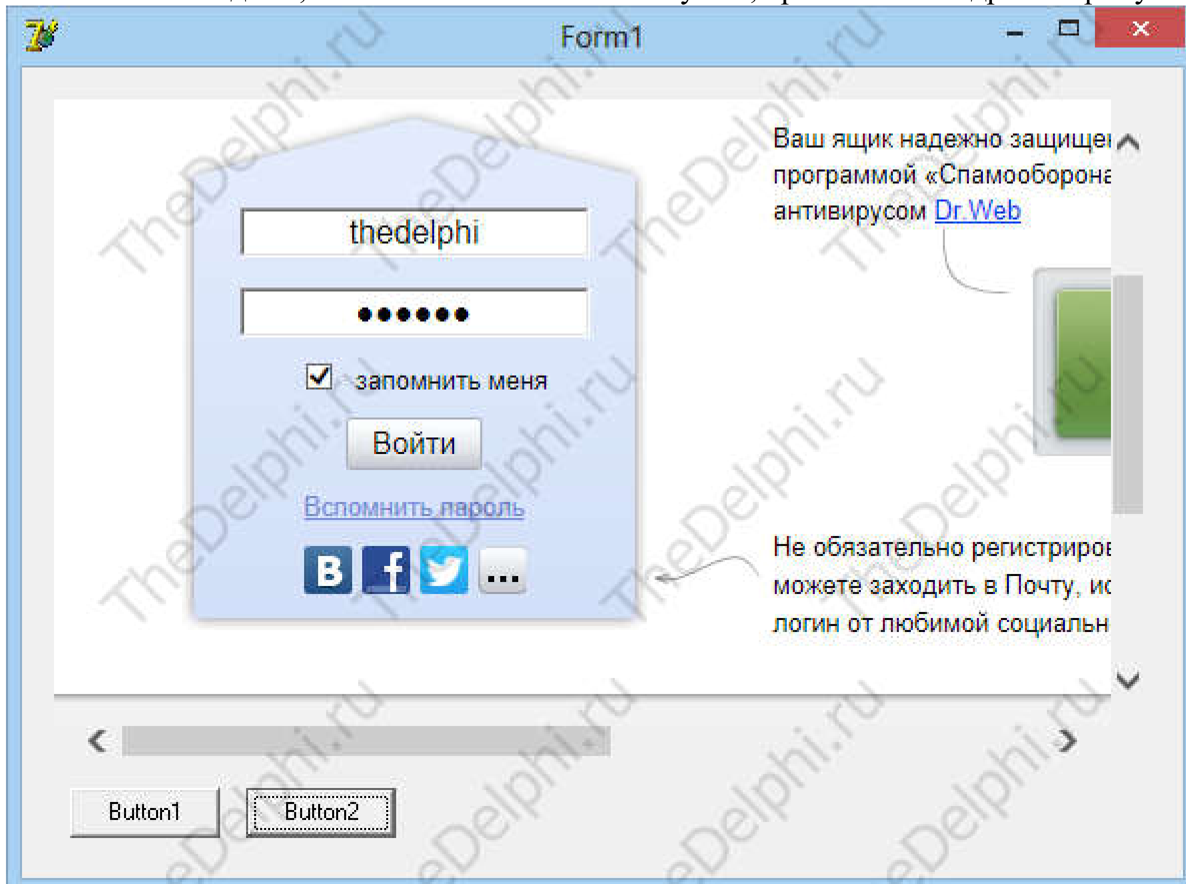
Теперь мы знаем имена полей и кнопки, а значит сможем их найти, создаем обработчик события OnClikc Button2 и заполним его вот таким кодом:

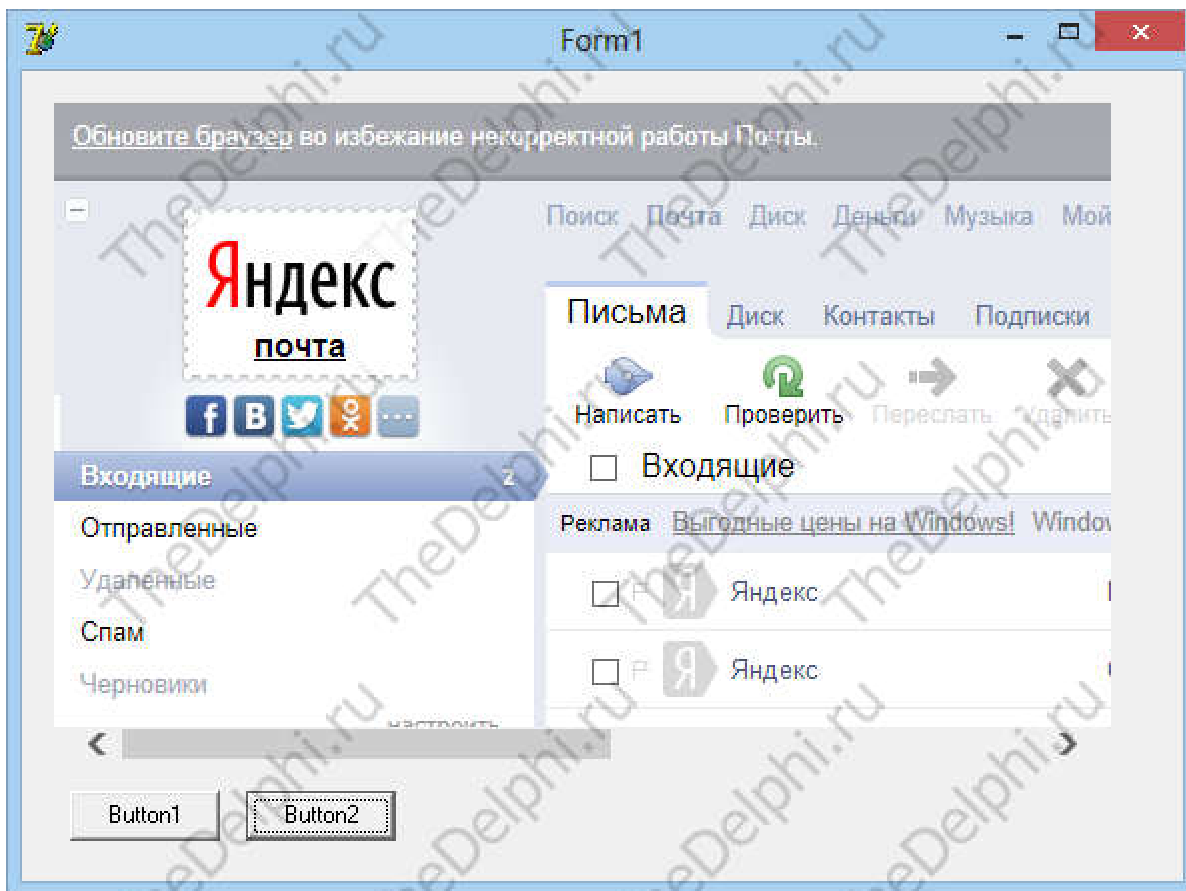
```
procedure TForm1.Button2Click(Sender: TObject);
var
  i:integer;
  s: String;
  html_tag: Variant;
begin
  html_tag:= WebBrowser1.OleObject.Document.forms.item(0).elements;
  for i:=0 to (html_tag.Length-1) do
  begin
    if html_tag.item(i).name = 'login' then  //ищем элемент с именем "login"
      html_tag.item(i).value:= 'thedelphi';  //и присваиваем ему значение

    if html_tag.item(i).name = 'passwd' then  //ищем элемент с именем "passwd"
      html_tag.item(i).value:= 'delphi';  //и присваиваем ему значение

    if html_tag.item(i).value = 'Войти' then //ищем элемент со значением "Войти"
      html_tag.item(i).click;  // и нажимаем на него
  end;
end;
```

Можно проверить, сначала загрузим страницу, нажав на первую кнопку, а потом на вторую, тем самым поля должны заполнится.



Но как видно ничего не произошло, такое случилось из-за той кучи кода, страничка оказалась слишком сложной, но слава богу есть альтернатива по проще: https://mail.yandex.ru/, имена элементов совпадают, так что менять ничего не нужно, кроме самого адреса. Пробуем снова:





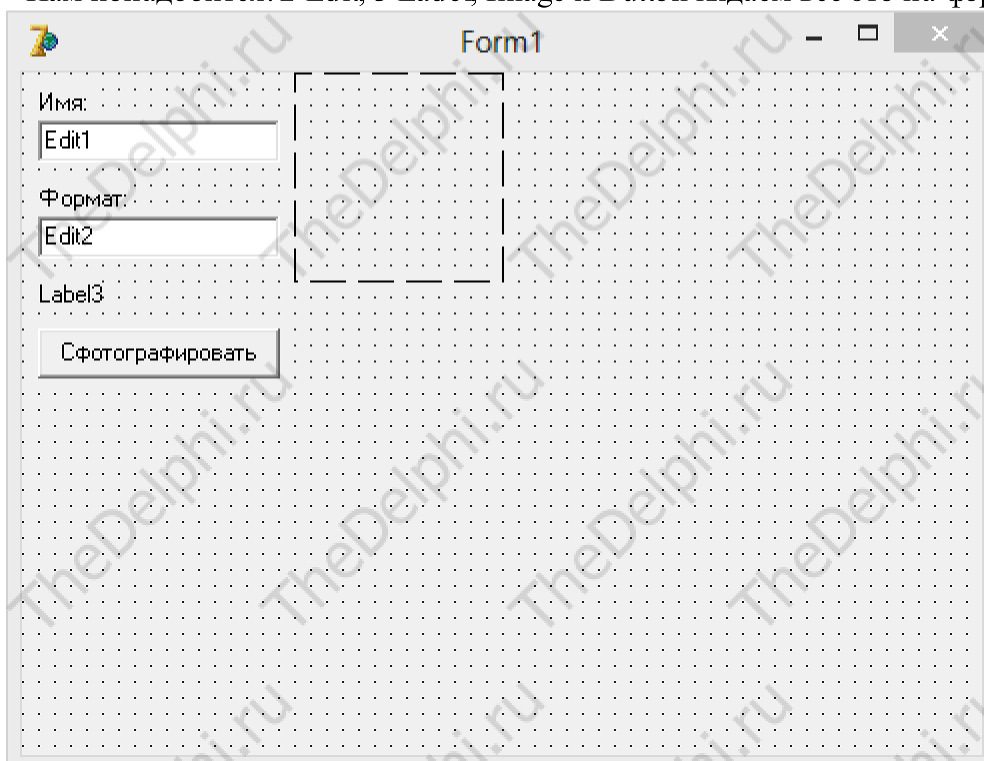
И вуаля, все получилось!!!

Таким способом можно например проверять новости еще что-нибудь.

Урок 39 - Запись рабочего стола

В этом уроке мы будем делать снимки экрана.

Нам понадобится: 2 Edit, 3 Ladel, Image и Button кидаем все это на форму и располагаем так:

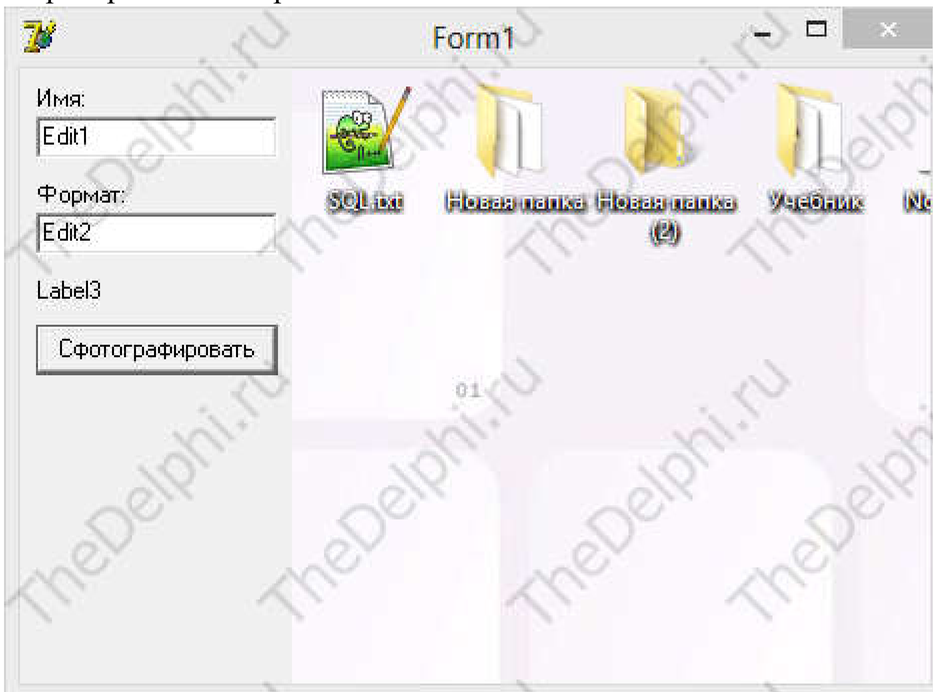


Размер снимка будет напрямую зависеть от размеров компонента Image, сделаем размер равный разрешению моего монитора (1280x1024), то есть свойство Width = 1280, а Heigth = 1024. Появились полосы прокрутки, их нам не надо, в раскрывающемся свойстве VertSrollBar и HorzSrollBar свойство Visible установим на false.

В Delphi вся работа с графикой основывается на понятии класса TCanvas и понятии холста, сейчас мы с этим классом и будем работать, создаем обработчик события OnClick на Button1 и пишем код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Canvas: TCanvas;
  ScreenV: HDC;
begin
  ScreenV:= GetDC(0);
  Canvas:= TCanvas.Create;
  Canvas.Handle:= ScreenV;
  Image1.Canvas.CopyRect(Rect(0, 0, Image1.Width, Image1.Height),
  Canvas, Rect(0, 0, Screen.Width, Screen.Height));
  ReleaseDC(0, ScreenV);
end;
```

Проверим как это работает:



Все отлично, теперь сохраним полученное, модифицировав код создания снимка, просто допишем код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Canvas: TCanvas;
  ScreenV: HDC;
begin
  ...

  Image1.Picture.SaveToFile(Edit1.Text + Edit2.Text); //сохраняем
end;
```

Вот зачем нам были нужны Edit'ы, в первый пишем например "foto", а во второй ".jpg", и в папке с программой появляется наш снимок экрана, не забудьте сохранить программу!

Перейдем к самому интересному, мульти-сьемка. В этом нет ничего сложного, надо просто через равные промежутки времени нажимать на Button1, с этой задачей справится таймер. Вытащим его на форму и создадим глобальную переменную i: integer;, в обработчике события OnTimer пишем код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  i:=i+1;
  Button1.Click; //делаем снимок
```

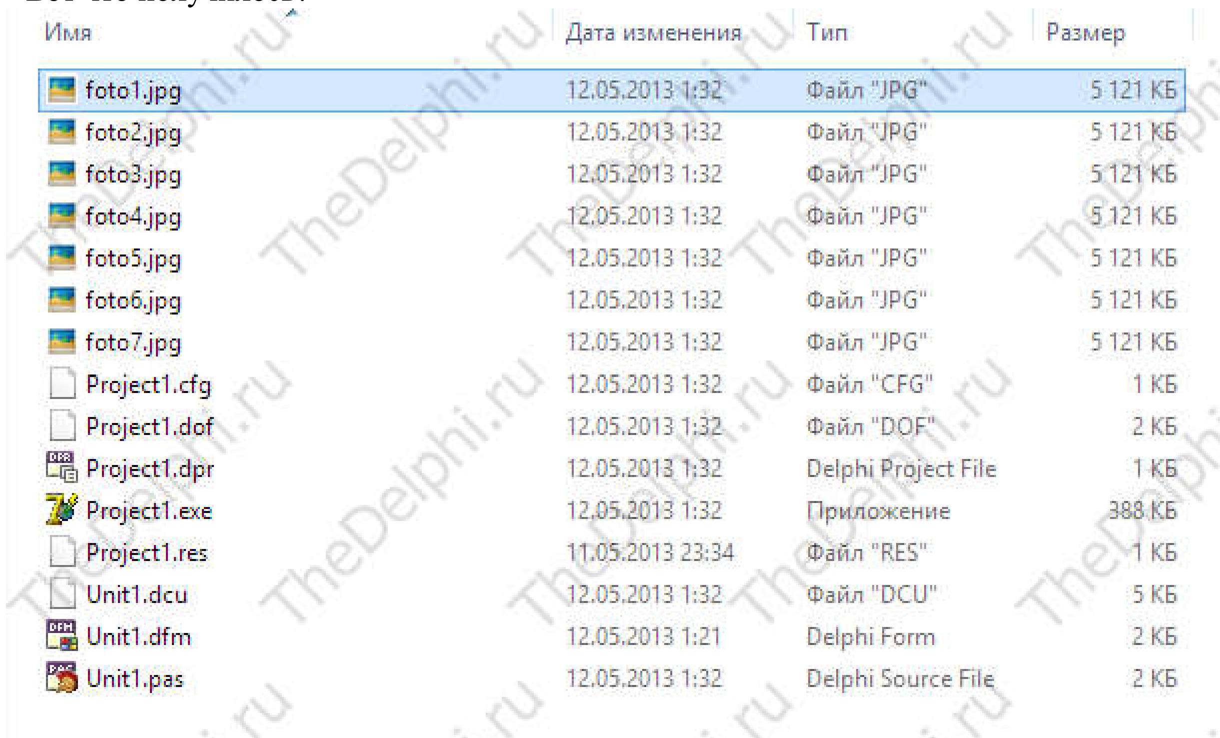
```
end;
```

Если мы сейчас запусти программу, то она будет делать снимки с интервалом в 1 сек. и сохранять все это в один файл, т.е. переписывать его, избавиться от этого нам поможет переменная "i", дополним процедуру создания снимка:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Canvas: TCanvas;
  ScreenV: HDC;
begin
  ...

  Image1.Picture.SaveToFile(Edit1.Text + IntToStr(i) +  Edit2.Text); //сохраняем
end;
```

Таким образом при каждом снимке имя файла будет: foto1.jpg, foto2.jpg, foto3.jpg и т.д. Сразу изменим свойства Textпервого Edit'a на "foto", а второго на ".jpg". После запуска времени чтоб это написать на не хватит и программа будет сохранять файлы с неправильными именами.

Вот что получилось:

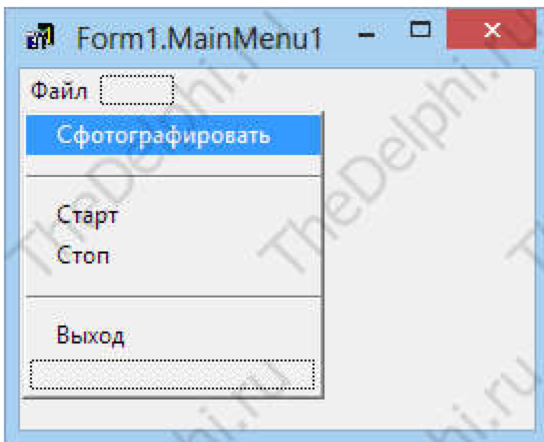


Имя	Дата изменения	Тип	Размер
foto1.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
foto2.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
foto3.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
foto4.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
foto5.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
foto6.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
foto7.jpg	12.05.2013 1:32	Файл "JPG"	5 121 КБ
Project1.cfg	12.05.2013 1:32	Файл "CFG"	1 КБ
Project1.dof	12.05.2013 1:32	Файл "DOF"	2 КБ
Project1.dpr	12.05.2013 1:32	Delphi Project File	1 КБ
Project1.exe	12.05.2013 1:32	Приложение	388 КБ
Project1.res	11.05.2013 23:34	Файл "RES"	1 КБ
Unit1.dcu	12.05.2013 1:32	Файл "DCU"	5 КБ
Unit1.dfm	12.05.2013 1:21	Delphi Form	2 КБ
Unit1.pas	12.05.2013 1:32	Delphi Source File	2 КБ

Урок 40 - Запись рабочего стола, интерфейс

В этом уроке мы доделаем программу и придадим ей божественный вид.

Сначала установим свойство Enabled у Timep1 на False>, таймер мы будем включать в ручную. Затем добавим компонент MainMenu с вкладки Standart. Теперь его нужно заполнить, кликаем 2 раза по компоненту и вылезет окно с выделенным пунктирной линией кусочком в левом верхнем углу, кликам на него и изменяем свойство Caption на "Файл" и видим что у нашей формы появилось меню, а дольше сами, как на рисунке:



Примечание: чтобы сделать разделить нужно в свойство Caption написать "-" (знак - минус).

Теперь кликаем 2 раза на элемент "Старт" и заполняем появившийся обработчик:

```
procedure TForm1.N4Click(Sender: TObject);
begin
  Timer1.Enabled:=true; //Включаем таймер
end;
```

Ну и также для "Стоп":

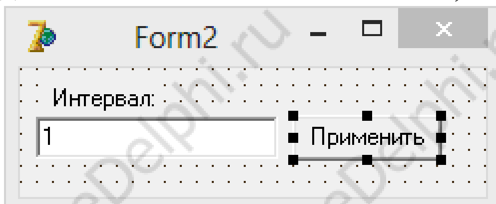
```
procedure TForm1.N5Click(Sender: TObject);
begin
  Timer1.Enabled:=false; //Выключаем таймер
end;
```

Для "Выход":

```
procedure TForm1.N7Click(Sender: TObject);
begin
  Close; // Выходим из программы
end;
```

Можно скомпилировать и проверить как это работает.

Добавим возможность изменения интервала таймера, для этого добавим новую форму и добавим на неё компоненты: Edit, Button, Label. Разместим вот так:



Кликаем 2 раза на кнопке "Применить" и пишем код и компилируем:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
  Form1.Timer1.Interval:= StrToInt(Edit1.Text)*1000; // Задаем интервал таймеру (переводим секунды в миллисекунды)
  Close; // Закрываем вторую форму
end;
```

Вылезет окошко подтверждения, соглашаемся и возвращаемся к MainMenu, там новую вкладку с названием "Опции" и 2 раза кликаем на ней:

```
procedure TForm1.N8Click(Sender: TObject);
begin
  Form2.Show; //Показываем вторую форму
end;
```

Компилируем, соглашаемся и готово!

Урок 41 - Панель быстрого запуска (часть 1/2)

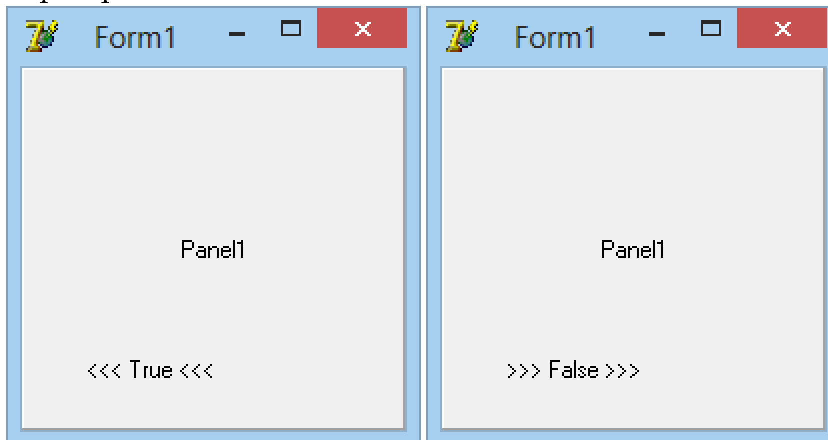
Начинаем создавать панельку быстрого запуска, в этом уроке мы сделаем движение (скрытие\показ).

Панелька будет выполнять простые функции: прятаться за границу экрана и показывается, запускать приложения, добавление приложений будет осуществляться способом "Drag-and-drop". В этом уроке мы сделаем только движение панели.

Наша панель будет располагаться слева, у края монитора и иметь небольшие размеры. Кидаем на форму компонентPanel и для теста Label, у Panel свойство Align установим на alClient. Выдвижение будет происходить при первом клике на панель, а при повторном клике, панель спрячется обратно. Поэтому создаем глобальную переменную Showed: boolean;, благодаря ей мы будем знать в каком состоянии панель, затем кликаем 2 раза Panel и пишем код:

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
 if Showed = false then // Если статут "Спрятана"
 begin
  Label1.Caption:='<<< True <<<';
  Showed:= True;  // Делаем статус "Показана"
 end
 else // Иначе
 begin
  Label1.Caption:='>>> False >>>';
  Showed:= False; // Делаем статус "Спрятана"
 end;
end;
```

Проверим:



Теперь сделаем движение панели. Создаем глобальную переменную S: integer; , это для того чтобы панель не перемещалась вечно, а останавливалась, вытаскиваем таймер, интервал зададим 10 и в обработчике события пишем код:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 if S = 15 then // Если пройдённый путь равен 15
 begin
  Timer1.Enabled:= False; // Выключаемся
 end
 else  // Иначе
 begin
  S:= S+1; // Увеличиваем путь на 1
  Form1.Left:= Form1.Left - 10; // Двигаем панель
 end;
end;
```

Это движение в одну сторону, для движения назад нам понадобится еще один таймер, кидаем его на форму, назначаем интервал 10 и в обработчике пишем код:

```
procedure TForm1.Timer2Timer(Sender: TObject);
begin
 if S = 0 then // Если пройдённый путь равен 0
 begin
  Timer2.Enabled:= False; // Выключаемся
```

```
  end
  else  // Иначе
  begin
    S:= S-1; // Уменьшаем путь на 1
    Form1.Left:= Form1.Left + 10; // Двигаем панель
  end;
end;
```

Теперь вместо присвоения текста Label1 на нужно запускать таймера, модифицируем обработчик OnClick Panel:

```
procedure TForm1.Panel1Click(Sender: TObject);
begin
  if Showed = false then // Если статут "Спрятана"
  begin
    Timer1.Enabled:= True; // Запускаем таймер показа
    Showed:= True; // Делаем статус "Показана"
  end
  else
  begin
    Timer2.Enabled:= True; // Запускаем таймер скрытия
    Showed:= False; // Делаем статус "Спрятана"
  end;
end;
```

Вот и все, в следующем уроке мы поработаем над интерфейсом.

Урок 42 - Панель быстрого запуска (часть 2/2)

В этом уроке мы доработаем панель быстрого запуска и добавим функцию Drag-and-drop.

Первым мы сделаем перетаскивание файлов. Кинем на форму компонент Image и объявим функцию после ключевого слова private в описании класса Form1:

```
...

private
  { Private declarations }
  procedure WmDropFiles( var Msg: TWMDropFiles); message WM_DropFiles;


...
```

Она будет срабатывать когда из системы придет сообщение о перетаскивании какого-нибудь файла на форму программы. Нажимаем комбинацию клавиш Ctrl-Shift-C и появляется шаблон этой функции, заполняем его, но перед этим нужно еще создать глобальные переменные:

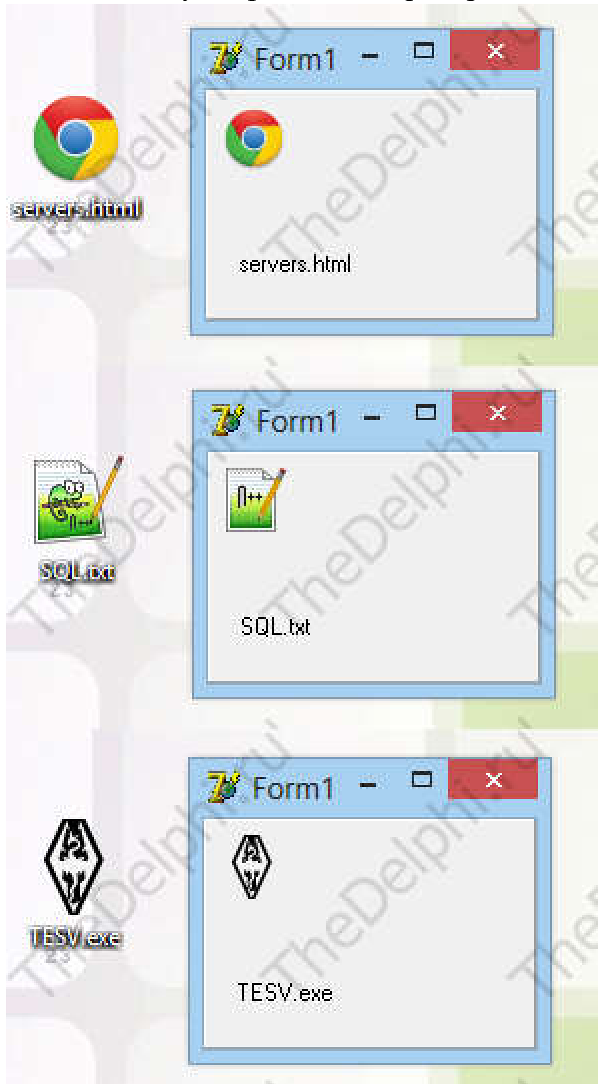
```
  CFileName: array[0..MAX_PATH] of Char; // Переменная с именем перетаскиваемого файла
  F: string; //Это для удобства

procedure TForm1.WmDropFiles(var Msg: TWMDropFiles);
var
  icon: hicon; //Иконка файла
  iconindex: word;
begin
  try
    if DragQueryFile(Msg.Drop,0, CfileName, Max_Path)> 0 then //Если перетащили файл
    begin
      F:=CfileName; // Конвертируем Array of Char -> String
      Label1.Caption:=ExtractFileName(F); // Получаем имя файла из его полного пути
      Msg.Result:=0;
    end;
  finally
    DragFinish(Msg.Drop); //Говорим что приняли файл
  end;
```

```
  iconindex:=1;
  //получаем картинку из файла
  Image1.Picture.Icon.Handle:= ExtractAssociatedIcon(HInstance, Pchar(F), iconIndex);
  DrawIcon(Canvas.Handle,10,10,icon); //Рисуем картинку
end;
```

Еще надо добавит в uses модуль ShellApi и в обработчик OnCreate добавит строчку:

```
        DragAcceptFiles(handle, true); // Включаем режим Drag-and-drop
end;
```

Тогда все будет работать. Проверяем:



Осталось только сделать запуск приложения по клику и убрать огранку формы. Кликаем 2 раза на Image и пишем код:

```
        ShellExecute(Form1.Handle, nil, Pchar(F), nil, nil, SW_RESTORE); //Запускаем программу (путь = F)
```

У Form1 свойство BorderStyle на bsNone, убираем огранку, свойство AlphaBlend True и AlphaBlendValue например 150. А также сделаем нашу форму по верх всех окон, свойство FormStyle на fsStayOnTop.

Вот что получилось:

Урок 43,44 - Структурные типы данных

В этом уроке мы раскроем важную тему собственных типов данных, ее не знание будет очень сильно вас ограничивать в плане удобства програмирования и построения правильной архитектуры приложения.

При создании любой серьёзной программы не обойтись без дополнительных, более сложных, чем числа и строки, типов данных. В Delphi программист может для своих целей конструировать собственные типы данных. Чтобы ввести в программу (описать) новый тип данных, применяется оператор с ключевым словом type:
type название_типа = описание_типа;

Перечислимый тип - это тип данных, диапазоном значений которого является просто набор идентификаторов. Это может применяться в тех случаях, когда нужно описать тип данных, значения которого нагляднее представить не числами, а словами. Перечислимый тип записывается взятой в круглые скобки последовательностью идентификаторов - значений этого типа, перечисляемых через запятую. При этом, первые элементы типа считаются младшими по сравнению с идущими следом. Например, тип, описывающий названия футбольных команд, можно сформировать так:

```
type
   FootballTeam = (Spartak, Dinamo, CSKA, Torpedo, Lokomotiv);
    var

   MyTeam: FootballTeam;
begin

   MyTeam:=Spartak;
end;
```

Вообще, под перечислимыми типами понимают все типы, для которых можно определить последовательность значений и их старшинство. К ним относятся:

все целочисленные типы, для которых всегда можно указать число, следующее за числом N;
символьные типы (Char): за символом ?a? всегда следует ?b?, за ?0? следует ?1?, и так далее;
логические типы - тип Boolean также представляет собой перечислимый тип: type Boolean = (false, true);

Структурные типы данных используются практически в любой программе. Это такие типы, как
массивы
записи
множества

Массив - это структура данных, доступ к элементам которой осуществляется по номеру (или индексу). Все элементы массива имеют одинаковый тип.
Описание массива имеет вид:
1 type

2 имя_типа_массива = array [диапазон] of тип_элемента;

Диапазон определяет нижнюю и верхнюю границы массива и, следовательно, количество элементов в нём. При обращении к массиву индекс должен лежать в пределах этого диапазона. Массив из ста элементов целого типа описывается так:

```
type
  TMyArray = array [1 .. 100] of Integer;
```

Теперь можно описать переменные типа TMyArray:

```
var
  A, B: TMyArray;
```

Вместо присвоения типа можно явно описать переменные как массивы:

```
var
  A, B : array [1..100] of Integer;
```

Для доступа к элементу массива нужно указать имя массива и индекс элемента в квадратных скобках. В качестве индекса может выступать число, идентификатор или выражение, значение которых должно укладываться в диапазон, заданный при описании массива:

```
var
  N: Integer;

begin
  N := 65;

  A[5] := 101;
  A[N] := 165;

  A[N+3] := 200;
  B := A;

end;
```

Иногда требуется узнать верхнюю границу массива. Встроенная функция High() вернёт число, являющееся верхней границей массива. В скобки нужно подставить массив, верхнюю границу которого требуется узнать.

Выражение:

1 B := A

Означает, что каждый элемент массива B равен элементу с таким же индексом массива A. Такое присвоение возможно только если переменные объявлены через некий поименованный тип, или перечислены в одном списке. И в случае:

```
var
  A: array[1..100] of String;

  B: array[1..100] of String;
```

его использовать невозможно (но возможно поэлементное присвоение B[1] := A[2]; и т.д.).

Массивы могут иметь несколько измерений, перечисляемых через запятую. Например, таблицу из четырёх столбцов и трёх строк:

1	2	3	4
5	6	7	8

9	10	11	12

Можно описать в виде массива с двумя измерениями:

```
type
  MyTable = array[1..4, 1..3] of Integer;

var
  X : MyTable;

  Y : Integer;
  begin

  Y:=X[3, 2];
  end;
```

Теперь в результате операции присвоения Y будет равен 7.

Многомерный, например, двумерный массив можно описать как массив массивов:

```
type
  TMyArray = array [1 .. 4] of array [1 .. 3] of Integer;
```

Результат будет аналогичен предыдущему примеру.

Каждое измерение многомерного массива может иметь свой собственный тип, не обязательно целый.

Кроме вышеописанных, так называемых статических массивов, у которых количество элементов неизменно, в Delphi можно использовать динамические массивы, количество элементов в которых допускается изменять в зависимости от требований программы. Это позволяет экономить ресурсы компьютера, хотя работа с такими массивами происходит гораздо медленнее. Описываются динамические массивы аналогично статическим, но без указания диапазона индексов:

```
type
  TDinArray = array of Integer;

var
  A: TDinArray;
```

После создания в динамическом массиве нет ни одного элемента. Необходимый размер задаётся в программе специальной процедурой SetLength. Массив из ста элементов:

```
begin
  SetLength(A, 100);

end;
```

Нижняя граница динамического массива всегда равна нулю. Поэтому индекс массива A может изменяться от 0 до 99.

Многомерные динамические массивы описываются именно как массивы массивов. Например, двумерный:

```
type
   T3DinArray = array of array of Integer;


var
   A : T3DinArray;
```

В программе сначала задаётся размер по первому измерению (количество столбцов):

```
SetLength(A, 3);
```

Затем задаётся размер второго измерения для каждого из трёх столбцов, например:

```
SetLength(A[0], 3);
SetLength(A[1], 2);
SetLength(A[2], 1);
```

Таким образом создаётся треугольная матрица:

A00 A10 A20
A01 A12
A02

Чтобы освободить память, выделенную динамическому массиву, нужно массиву как целому присвоить значение nil:

```
A:=nil;
```

Ключевое слово nil в Delphi означает отсутствие значения.

Записи очень важный и удобный инструмент. Даже не применяя специальные технологии, с его помощью можно создавать собственные базы данных. Записи - это структура данных, каждый элемент которой имеет собственное имя и тип данных. Элемент записи иначе называют поле. Описание записи имеет вид:

```
type
        имя_типа_записи = record
                название_поля : тип_поля ;
                . . .
                название_поля : тип_поля ;
        end;
```

Названия полей, имеющих одинаковый тип, можно, как и в случае описания переменных, указывать в одну строку через запятую. Для обращения к полю записи сначала указывают имя записи, затем точку, затем имя поля. Например, данные о персонале предприятия могут быть организованы таким типом записи:

```
type
        TPers = record
                Fam, Name, Par : String;
                Year : Integer;
                Dep : String;
                end;
        var Pers : TPers;
                begin
                Pers.Fam:='Иванов';
                Pers.Name:='Иван';
                Pers.Par:='Иванович';
                Pers.Year:=1966;
                Pers.Dep:='Цех №1';
end;
```

Теперь осталось записать эти данные в файл, предварительно объявив и его тип как TPers, и база данных готова. С файлом в Delphi также ассоциируется переменная, называемая файловой переменной, которая описывается так:

```
VFile : file of тип_файла;
```

В качестве типа может использоваться любой ограниченный тип Delphi. При этом не допускается тип String, так как он допускает переменный размер до 2 ГБайт. Его необходимо ограничивать: String[N], где N - количество символов. ТипTPers из предыдущего примера должен быть описан, например, так:

```
type
        TPers = record
                Fam, Name, Par : String[20];
                Year : Integer;
                Dep : String[10];
        end;
```

Теперь переменная такого типа занимает строго определённое место в памяти, и может быть записана в файл.

Множество - это группа элементов, объединённая под одним именем, и с которой можно сравнивать другие величины, чтобы определить, принадлежат ли они этому множеству. Количество элементов в одном множестве не может превышать 256. Множество описывается так:

```
type
	имя_множества = set of диапазон_значений_множества ;
```
В качестве диапазона может указываться любой тип, количество элементов в котором не больше 256. Например:
```
type
	TMySet = set of Byte;
```
Конкретные значения множества задаются в программе с помощью перечисления элементов, заключённых в квадратные скобки. Допускается использовать и диапазоны:
```
var
	MySet : TMySet;
begin
	MySet:=[1, 3 .. 7, 9];
end;
```
Чтобы проверить, является ли некое значение элементом множества, применяется оператор in в сочетании сусловным оператором:
```
var
	Key : Char;
	Str: String;
begin
	if Key in [?0? .. ?9?, ?+?, ?-?] then Str:=?Math?;
end;
```
Чтобы добавить элемент во множество, используется операция сложения, удалить - вычитания:
```
var Digit: set of Char=[?1?..?9?];
var Math: Set of Char;
begin
	Math:=Digit+[?+?, ?-?, DecimalSeparator<sup>*</sup>];<br>
end;
```
*Примечание: DecimalSeparator - встроенная в Delphi константа типа Char, имеющая значение символа-разделителя целой и дробной частей, который может быть равен точке ('.') либо запятой (','), в зависимости от текущих настроек Windows.

Урок 45,46 - Динамическое создание компонентов

Динамически создаваемые компоненты - это компоненты, место в памяти под которые выделяется по мере необходимости в процессе работы приложения. Этим они и отличаются от компонентов, которые помещаются на Форму при проектировании приложения. Возможность создавать компоненты динамически это очень большое удобство для программиста. Например, можно создавать в цикле сразу много однотипных компонентов, формируя из них массив, которым в дальнейшем очень просто управлять.

Все компоненты, как объекты, имеют множество свойств, определяющих их работу. При установке компонента на Форму из палитры большинство этих свойств определяются системой Delphi автоматически. При созданиидинамического компонента программист должен описать и настроить их вручную. Посмотрим, как это делается.

Прежде всего, для появления динамически создаваемого компонента нужно выделить под него место в памяти. Выделением места в памяти компьютера под любой компонент занимается конструктор типа объекта этого компонента - метод Create. Для этого сначала нужно описать переменную нужного типа, а затем для выделения памяти воспользоваться методом Create. Метод Create имеет параметр Owner, определяющий так называемого "владельца" для создаваемого компонента.

Хотя на самом деле владелец нужен не для создания, а для уничтожения компонента. То есть, при уничтоженииикомпонента-владельца происходит автоматическое уничтожение всех компонентов, у которых он указан в качестве владельца.

При обычной установке компонента из палитры система делает владельцем этого компонента

Форму. Проще всего поступать так же. Однако можно указать в качестве владельца сам этот компонент, воспользовавшись в качестве параметра ключевым словом Self.

Далее. Когда компонент создан, то есть место в памяти под него выделено, можно задавать значения параметрам этого объекта. Прежде всего, это ещё один компонент, так называемый "родитель". Компонент-родитель будет отвечать за отрисовку нашего динамически создаваемого компонента. Это значит, что новый компонент появится в границах компонента-родителя.

Если компонент-владелец имеет тип TComponent, то есть может быть любым компонентом, то компонент-родитель уже имеет тип TWinControl. То есть это должен быть "оконный" компонент, умеющий принимать и обрабатывать сообщения от системы Windows. Это необходимо, так как компонент должен находиться в некоторой иерархии компонентов, принимающих и передающих сообщения от системы Windows. Нашему динамическому компоненту сообщения будут передаваться через компонент-родитель.

А некоторые компоненты вообще не умеют принимать сообщения от системы, и в процессе работы в этом случае ими также будет управлять компонент-родитель, например, Форма или Панель, на которой они находятся.

Естественно, компонент не может быть родителем для самого себя. Имя компонента-родителя просто присваивается свойству Parent создаваемого динамически компонента.

Вот общая схема "конструирования" динамически создаваемого компонента:

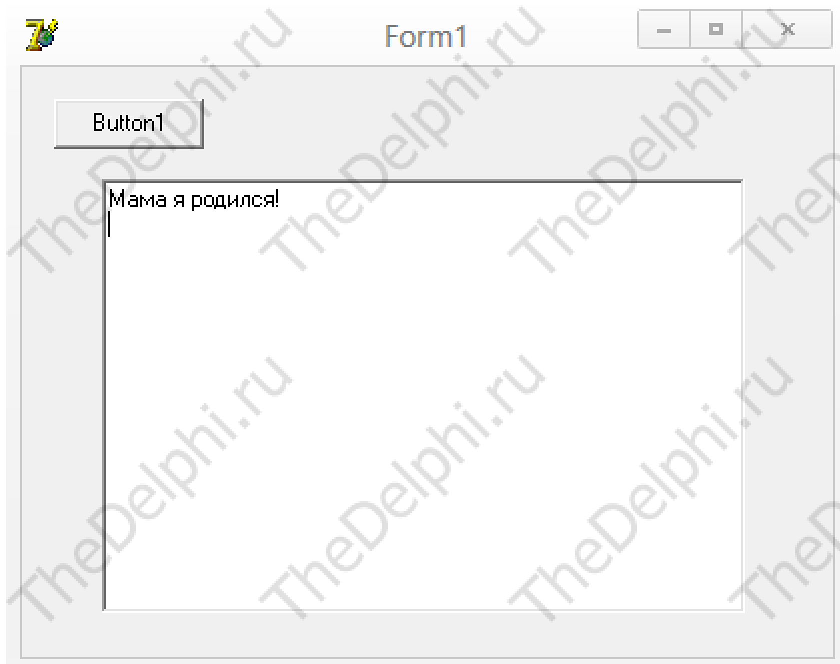
```
var
        Component: TComponent; //Описать переменную для компонента
begin
        Component:=TComponent.Create(Owner); //Задать владельца
        Component.Parent:=Parent; //Задать родителя
end;
```

На этом создание компонента можно считать законченным, и он успешно появляется (или "не появляется", если он не визуальный!) в приложении. Остальные свойства будут присвоены ему по умолчанию самой системой Delphi.

Естественно, для визуальных компонентов значения таких свойств по умолчанию как положение на Форме, ширина, высота "свежесозданного" динамически компонента мало кого устроят. Нужные размеры и положение компонента также придётся задать программисту.

Давайте для примера динамически создадим многострочный редактор, компонент Memo. Пусть он появляется на Форме по нажатию кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
var
        Memo: TMemo;
begin
        Memo:=TMemo.Create(Form1); //Создание Memo
        Memo.Parent:=Form1; // Задаем родителя
        Memo.Left:=50; // Позиция по горизонтали
        Memo.Top:=50;  // Позиция по вертикали
        Memo.Width:=250;  // Ширина
        Memo.Height:=100; // Высота
        Memo.Text:='Мама я родился!';  // Тескт
end;
```



Есть ещё свойство Name! По умолчанию Delphi присвоит ему типовое имя с присвоением очередного порядкового номера: Memo1. Программист при создании компонента также может присвоить свойству Name нужное значение, например:

```
Memo.Name:='DynamicallyCreatedMemo';
```

К данному компоненту можно обращаться как по этому имени, так и с указанием переменной, с помощью которой он был создан: Memo. Естественно, в последнем случае переменная должна быть глобальной.

Урок 47 - Исключительные ситуации

Исключительные ситуации в Delphi встречаются постоянно. Исключительная ситуация это такая ситуация, в результате которой генерируется ошибка, и выполнение программы прерывается. Именно потому такая ситуация и называется исключительной. Например, деление на ноль - классический пример исключительной ситуации.

Как в такой ситуации действует человек? Если он пытается что-то сделать, и это не получается - он идёт другим путём. Так же и компьютер, следующий программе, умеющей обрабатывать исключительные ситуации. Он не бросает выполнение программы, не виснет, а обходит исключительную ситуацию, выполняя альтернативный вариант фрагмента, в которой исключительная ситуация возникла.

Возникает вопрос, почему бы не поставить проверку, например, на равенство нулю знаменателя при делении? Можно и поставить. Но во многих случаях источник исключительной ситуации далеко не так очевиден, а на все случаи жизни проверки не введёшь.

Итак, для контроля исключительных ситуаций программист должен подготовить как основной вариант фрагмента, гдевозможна исключительная ситуация, так и его вариант, в котором она заведомо невозможна, или выводится информационное сообщение для пользователя.

Вот как выглядит оператор контроля исключительных ситуаций:

```
try
    //основные операторы фрагмента;
except
    //альтернативный вариант фрагмента;
end;
```

Вначале производится попытка выполнить операторы секции try/except, содержащие основной вариант программы. При возникновении в каком-либо операторе этой секции исключительной ситуации остальные операторы секции пропускаются, и выполняется секция except/end. Если всё "проходит штатно", то секция except/endпропускается.

Ещё один вариант оператора контроля исключительных ситуаций применяется, когда необходимо, чтобы определённый фрагмент кода выполнился в любом случае, возникла исключительная ситуация или нет:

```
try
	//операторы;
finally
	//заключительные действия;
end;
```

Основные операторы, находящиеся в секции try, могут пройти штатно, или вызвать исключительную ситуацию. Операторы заключительных действий, находящиеся в секции finally, будут выполнены в любом случае.

Есть ещё один способ контроля исключительных ситуаций, касающийся ошибок операций ввода-вывода.

Перед участком программы, где возможны ошибки ввода-вывода (а это, по сути, все операторы ввода-вывода), ставится директива {$I-}, заставляющая компилятор не включать в код автоконтроль ошибок ввода-вывода. Таким образом, в случае ошибки ввода или вывода программа не прерывается. В конце участка с операторами ввода-вывода ставится директива, включающая автоконтроль: {$I+}. Затем анализируется результат вызова функции IOResult. Если функция IOResult (вызывается без параметров) возвращает 0, значит ошибок ввода-вывода на данном участке не было.

Вот какой пример использования директив {$I} и функции IOResult содержит справка системы Delphi:

```
var
	F: file of Byte;
begin
	if OpenDialog1.Execute then //Открываем диалог выбора файла
	begin
		AssignFile(F, OpenDialog1.FileName); // Открываем файл
		{$I-} //Выключаем контроль ошибок
		Reset(F);
		{$I+} //Включаем контроль ошибок
		if IOResult = 0 then //Если ошибок нет
		begin
			//Сообщение с количеством байтов
			MessageDlg(?File size in bytes: ? + IntToStr(FileSize(F)), mtInformation,
[mbOk], 0);
			CloseFile(F); // Закрываем файл
		end
			else //Иначе
				MessageDlg(?File access error?, mtWarning, [mbOk], 0); // Сообщение
об ошибке
	end;
end;
```

Функция IOResult досталась Delphi в наследство от Turbo Pascal. Тот же самый фрагмент можно

составить и с использованием оператора try. На мой взгляд, это удобнее и проще.

При работе программы под управлением Delphi, система будет сама реагировать на исключительные ситуации, мешая работе операторов обработки исключений. Чтобы проверить их действие, можно запускать программу непосредственно, сворачивая Delphi и пользуясь ярлыком, установленном на Рабочем столе. Или можно отключить реакцию системы на исключительные ситуации, тем самым давая возможность отработать специально для этого написанным фрагментам программы - нашим операторам try/except/end. Для этого откроем пункт системного меню Delphi Tools -> Debugger Options.... В появившемся окошке нужно снять галку в чекбоксе Stop on Delphi Exceptions, расположенном на вкладке Language Exceptions. Теперь система Delphi будет предоставлять вашей программе возможность самостоятельно обрабатывать исключительные ситуации, среди которых могут быть и ситуации, возникновение которых прописано специально как удобный инструмент достижения необходимых результатов.

Урок 48,49 - Потоки в Delphi
Создание потока

Потоки в Delphi выполняют функцию имитации псевдопараллельной работы приложения. Как известно, для организации многозадачности операционная система выделяет каждому приложению, выполняющемуся в настоящий момент, определённые кванты времени, длина и количество которых определяется его приоритетом. Поэтому объём работы, который приложение может выполнить, определяется тем, сколько таких квантов оно сможет получить в единицу времени. Для операционной системы каждый поток является самостоятельной задачей, которой выделяются кванты времени на общих основаниях. Поэтому приложение Delphi, умеющее создать несколько потоков, получит больше времени операционной системы, и соответственно сможет выполнить больший объём работы.

Создать дополнительный поток в Delphi поможет объект TThread. Ввести объект TThread в программу можно двумя способами:
С помощью Мастера
Вручную
Мастер создания дополнительного потока в Delphi создаёт отдельный модуль, в рамках которого выполняется поток. Выполним:

File -> New -> Other...

В появившейся табличке выбора найдём TThread Object. Появится окошко, в верхнюю строку которого (Class Name) введём имя нашего будущего потока: MyThread. В результате будет создан модуль, содержащий заготовку кода, реализующего дополнительный поток Delphi:

```
unit Unit2; // Имя модуля, содержащего поток. При сохранении его можно изменить.

interface

uses
        Classes;

type
        MyThread = class(TThread) //MyThread - заданное нами имя потока.
        private
                { Private declarations }
        protected
                procedure Execute; override;
        end;

        implementation
```

```
	{ Important: Methods and properties of objects in visual components can only be
	  used in a method called using Synchronize, for example,

	Synchronize(UpdateCaption);

	  and UpdateCaption could look like,

	procedure MyThread.UpdateCaption;
	begin
		Form1.Caption := 'Updated in a thread';
	end; }

	{ MyThread }

	procedure MyThread.Execute;
	begin
		{ Place thread code here }
	end;

end.
```

В первом способе класс MyThread был создан мастером в дополнительном модуле. Второй способ состоит в том, что мы сами создаём такой класс в рамках одного из уже существующих модулей программы, например, в модулеUnit1:

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
  TMyThread = class(TThread)
	private
		{ Private declarations }
	protected
		procedure Execute; override;
	end;

var
  Form1: TForm1;
  MyThread: TMyThread;

implementation

{$R *.dfm}
```

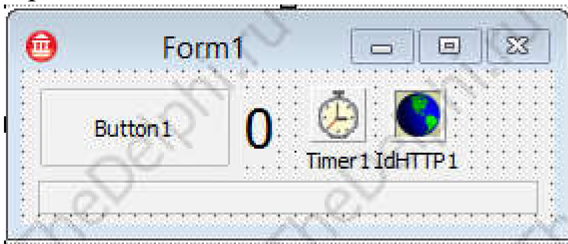
```
procedure TMyThread.Execute;

begin
end;

procedure TForm1.Button1Click(Sender: TObject);

begin
end;

end.
```

Если поток создаётся мастером, т.е. в другом модуле, то не забудьте в основном модуле описать переменную - экземпляр потока, как указано выше. Также, поскольку класс потока описан в другом модуле, имя этого модуля необходимо добавить в секцию uses. Теперь можно запускать поток, даже если в его процедуре Execute нет ни единого оператора.

Использование потока

Сделаем на основе полученных знаний о потоках из предыдущих уроков небольшой пример, мы будем скачивать файл из интернета размером 50 мб.

Создадим новый проект, и вытащим компоненты: Button, Label, Timer, IdHTTP (Indy) и ProgressBar, сначала сделаем без потока, а потом с ним. Разметим эти компоненты так как на картинке:



Скачивать файл мы будем по нажатию на кнопку, но чтобы информировать нас о том зависла ли программа, нам понадобятся таймер и label. Создадим у таймера обработчик OnTimer:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
        Label1.Caption:= IntToStr(StrToInt(Label1.Caption) + 1); //Увеличиваем на 1 число
end;
```

Если программа не зависла то число в label будет увеличивается, ибо нет.

Разберёмся с ProgressBar'ом, он нам нужен для отображения прогресса скачивания. Создадим 2 обработчика:OnWorkOnWorkBegin, в первом:

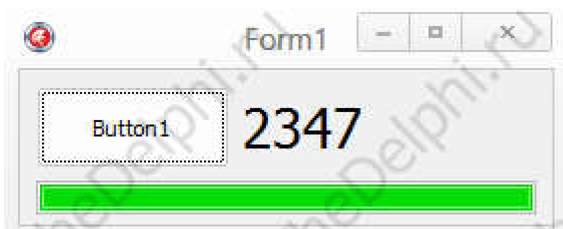
```
procedure TForm1.IdHTTP1Work(ASender: TObject; AWorkMode: TWorkMode;
  AWorkCount: Int64);
begin
        ProgressBar1.Position:= AWorkCount; //Кол-во скаченых байтов
end;
```

Во втором:

```
procedure TForm1.IdHTTP1WorkBegin(ASender: TObject; AWorkMode: TWorkMode;
  AWorkCountMax: Int64);
begin
        ProgressBar1.Max:= AWorkCountMax; //Макс. кол-во байтов в файле
end;
```

Ну а теперь самое главное, процедура скачивания. В обработчике OnClick на кнопке:

```
procedure Button1Click(Sender: TObject);
var
        Stream: TMemoryStream; //Буфер для хранения файла
begin
        Stream:= TMemoryStream.Create; // Создаем буфер
        Form1.IdHTTP1.Get('Адрес до файла', Stream); // Скачиваем
        Stream.Free; // Очищяем буфер
end;
```



Как видите числе замерло во время скачивания. Основной поток был занят файлом и не обращал внимания на компоненты. Даже переместить программу за заголовок нельзя.

Задание

Создайте поток под именем TDownload и поместите выполняемое приложение в поток, так чтобы при нажатии на кнопку «Button1» создавался новый поток и выполнялось в нем приложение, а при завершении работы приложения поток уничтожался.

Урок 51,52 - Создание собственных процедур и функций Delphi
Создание собственных процедур

Подпрограммы - процедуры и функции в языке Delphi служат для выполнения специализированных операций. Delphi имеет множество стандартных подпрограмм, но всё равно приходится создавать собственные для выполнения часто повторяющихся операций с данными, которые могут меняться.

Вообще, существует методика программирования "сверху вниз". Методика программирования "сверху вниз" разбивает задачу на несколько более простых, которые оформляются в виде подпрограмм. Те, в свою очередь, при необходимости также делятся до тех пор, пока стоящие перед программистом проблемы не достигнут приемлемого уровня сложности (то есть простоты!). Таким образом, эта методика программирования облегчает написание программ за счёт создания так называемого скелета, состоящего из описателей подпрограмм, которые в дальнейшем наполняются конкретными алгоритмами. Пустое описание подпрограммы иначе называется "заглушкой".

И процедуры, и функции позволяют добиться одинаковых результатов. Но разница всё же есть.

Процедура Delphi просто выполняет требуемые операции, но никаких результатов своих действий не возвращает. Результат - в тех изменениях, которые произошли в программе в процессе выполнения этой процедуры. В частности, процедура может поменять значения переменных, записать новые значения в ячейки компонентов, сделать запись в файл и т.д.

Функция Delphi также позволяет выполнить всё перечисленное, но дополнительно возвращает результат в присвоенном ей самой значении. То есть вызов функции может присутствовать в выражении справа от оператора присваивания. Таким образом, функция - более универсальный объект!

Описание подпрограммы состоит из ключевого слова procedure или function, за которым следует имя подпрограммы со списком параметров, заключённых в скобки. В случае функции далее ставится двоеточие и указывается тип возвращаемого значения. Обычная точка с запятой далее - обязательна! Сам код подпрограммы заключается в "логические скобки" begin/end. Для функции необходимо в коде присвоить переменной с именем функции или специальной зарезервированной переменной Result (предпочтительно) возвращаемое функцией значение. Пример:

```
function Имя_функции(параметры): тип_результата;
begin
	Код функции;
	Result:=результат;
end;
```

Описанная таким образом подпрограмма должна быть размещена в основной программе до

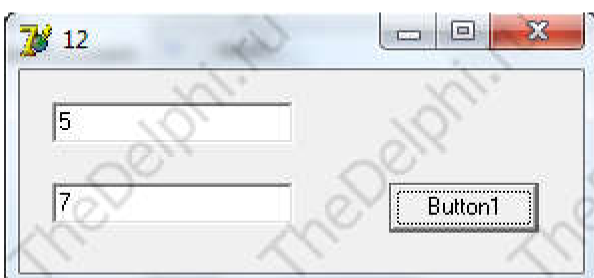
первого её вызова. Иначе при компиляции получите извещение о том, что "неизвестный идентификатор..." Следить за этим не всегда удобно. Есть выход - разместить только заголовок подпрограммы там, где размещают описания всех данных программы.

Параметры - это список идентификаторов, разделённых запятой, за которым через двоеточие указывается тип. Если списков идентификаторов разных типов несколько, то они разделяются точкой с запятой. Всё, как и в случае обычного описания данных. Это так называемые формальные параметры. При вызове подпрограммы они заменяются нафактические - следующие через запятую данные того же типа, что и формальные.
Параметры в описании подпрограммы могут и отсутствовать, тогда она оперирует данными прямо из основной программы.

Теперь нужно ввести понятие локальных данных. Это данные - переменные, константы, подпрограммы, которые используются и существуют только в момент вызова данной подпрограммы. Они так же должны быть описаны в этой подпрограмме. Место их описания - между заголовком и началом логического блока - ключевым словом begin. Именалокальных данных могут совпадать с именами глобальных. В этом случае используется локальная переменная, причём её изменение не скажется на глобальной с тем же именем.
Совершенно аналогично локальным типам, переменным, константам могут быть введены и локальные процедуры и функции, которые могут быть описаны и использованы только внутри данной подпрограммы

Теперь пример. Напишем программу суммирования двух чисел. Она будет состоять из Формы, на которой будет кнопка (компонент Button), по нажатию на которую будет выполняться наша подпрограмма, и двух строк ввода (компоненты Edit), куда будем вводить операнды.

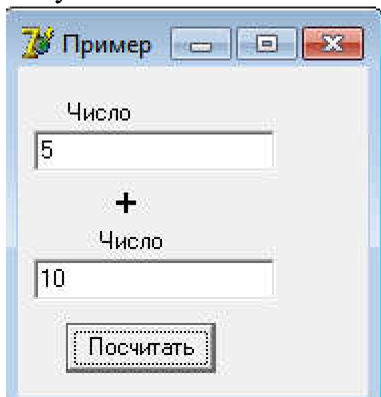
```
var
	Form1: TForm1;
	A, B, Summa: Integer;

procedure Sum(A, B: Integer);

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
	A:=StrToInt(Edit1.Text);
	B:=StrToInt(Edit2.Text);
	Sum(A, B);
	Caption:=IntToStr(Summa);
end;

procedure Sum(A, B: Integer);
begin
	Summa:=A+B;
end;
```

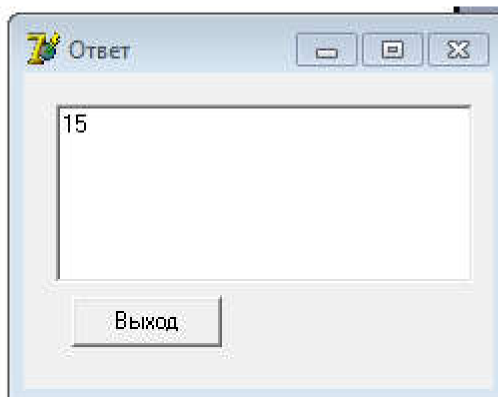


Задание

Переделайте только что созданное вами приложение так, чтобы при нажатии на кнопку «Button1» выводилась новая форма и результат сложения показывался в поле элемента Memo, добавте кнопку которая закрывает вторую форму. Процедуру Sum переделайте в функцию.

Результат





Урок 53 - Классы Delphi

Класс — это тип данных, определяемый пользователем. То, что в Delphi имеется множество предопределенных классов, не противоречит этому определению -ведь разработчики Delphi тоже пользователи Object Pascal.

Класс должен быть объявлен до того, как будет объявлена хотя бы одна переменная этого класса. Т.е. класс не может объявляться внутри объявления переменной.

В любом вашем Делфи-приложении вы можете увидеть строки:

```
type
        TForml = class(TForm)
                Buttonl: TButton;
                procedure ButtonlClick(Sender: TObject);
        end;
var
        Forml: TForml;
```

Это объявление класса TForml вашей формы и объявление переменной Forml -объекта этого класса.

В общем случае синтаксис объявления класса следующий:

```
Type
        <имя  класса> = Class(<имя класса - родителя>)
        public    // т.е. доступно всем
                <поля, методы, свойства, события>
        published // т.е. видны в Инспекторе Объекта и изменяемы
                <поля, свойства>
        protected // доступно только  потомкам
                <поля, методы, свойства, события>
        private // доступно только в этом модуле
                <поля, методы, свойства, события>
        end;
```

Имя класса может быть любым допустимым идентификатором. Но принято идентификаторы большинства классов начинать с символа "Т". Имя класса - родителя может не указываться. Тогда предполагается, что данный класс является непосредственным наследником TObject - наиболее общего из предопределенных классов. Таким образом, эквивалентны следующие объявления:

```
type
        TMyClass = class
        end;
```

и

```
type
        TMyClass = class(TObject)
        end;
```

В приведенном ранее объявлении класса формы TForml видно, что его родительским классом является класс TForm.

Класс наследует поля, методы, свойства, события от своих предков и может отменять какие-то из этих элементов класса или вводить новые. Доступ к объявляемым элементам класса определяется тем, в каком разделе они объявлены.

Раздел public (открытый) предназначен для объявлений, которые доступны для внешнего использования. Это открытый интерфейс класса. Раздел published (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и которые, следовательно, пользователь может устанавливать в процессе проектирования. Раздел private (закрытый), содержит объявления полей, процедур и функций, используемых только внутри данного класса. Раздел protected (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от конечного пользователя. Однако в отличие от закрытых, защищенные элементы остаются доступны для

программистов, которые захотят производить от этого класса производные объекты, причем не требуется, чтобы производные объекты объявлялись в этом же модуле.
Объявления полей выглядят так же, как объявления переменных или объявления полей в записях:

<имя поля>: <тип>;

В приведенном ранее объявлении класса формы вы можете видеть строку

Buttonl: TButton;

Это объявление объекта (поля) Buttonl типа (класса) TButton.
Имеется одно очень существенное отличие объявления поля от обычного объявления переменной: в объявлении поля не разрешается его инициализация каким-то значением. Автоматически проводится стандартная инициализация: порядковым типам в качестве начального значения задается 0, указателям — nil, строки задаются пустыми. При необходимости задания других начальных значений используются конструкторы, описанные далее в 4 части.
Объявления методов в простейшем случае также не отличаются от обычных объявлений процедур и функций.
Поля данных, исходя из принципа инкапсуляции — одного из основополагающих в объектно-ориентированном программировании, всегда должны быть защищены от несанкционированного доступа. Доступ к ним, как правило, должен осуществляться только через свойства, включающие методы чтения и записи полей. Поэтому поля целесообразно объявлять в разделе private - закрытом разделе класса. В редких случаях их можно помещать bprotected — защищенном разделе класса, чтобы возможные потомки данного класса имели к ним доступ. Традиционно идентификаторы полей совпадают с именами соответствующих свойств, но с добавлением в качестве префикса символа F.
Свойство объявляется оператором вида:

```
property <имя свойства>:<тип>
                  read
                 <имя поля или метода чтения>
                  write
                 <имя поля или метода записи>
                 <директивы  запоминания>;
```

Если в разделах read или write этого объявления записано имя поля, значит предполагается прямое чтение или запись данных (т.е. обмен данными непосредственно с полем).
Если в разделе read записано имя метода чтения, то чтение будет осуществляться только функцией с этим именем. Функция чтения — это функция без параметра, возвращающее значение того типа, который объявлен для свойства. Имя функции чтения принято начинать с префикса Get, после которого следует имя свойства.
Если в разделе write записано имя метода записи, то запись будет осуществляться только процедурой с этим именем. Процедура записи — это процедура с одним параметром того типа, который объявлен для свойства. Имя процедуры записи принято начинать с префикса Set, после которого следует имя свойства.
Если раздел write отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь не может задавать его значение.
Директивы запоминания определяют, как надо сохранять значения свойств при сохранении пользователем файла формы .dfm. Чаще всего используется директива default - значение по умолчанию.
Она не задает начальные условия. Это дело конструктора. Директива просто говорит, что если пользователь в процессе проектирования не изменил значение свойства но умолчанию, то сохранять значение свойства не надо.
Приведем пример.

Начните новый проект. Объявление класса, который мы хотим создать, можно поместить непосредственно в файл модуля. Но если вы хотите создать класс, который будете использовать в различных проектах, лучше оформить его в виде отдельного модуля unit, сохранить в каталоге своей библиотеки или библиотеки Delphi, и подключать в дальнейшем к различным проектам с помощью предложения uses. Мы выберем именно этот вариант. Так что выполните команду File | New | Unit (в некоторых более старых версиях Delphi — File | New и на странице New выберите пиктограмму Unit). Сохраните сразу этот ваш модуль в библиотеке под именем, например, MyClasses. А в модуль формы введите оператор, ссылающийся на этот модуль:

```
uses MyClasses;
```

Теперь займемся созданием класса в модуле MyClasses. Текст этого модуля может иметь пока такой вид:

```
unit MyClasses;
interface

uses SysUtils, Dialogs, Classes;

type
	TPerson = class
	private
		FName: string;  // Фамилия, имя и отчество
		FDepl, FDep2, FDep3: string;	// Место работы (учебы)
		FYear: word;  // Год рождения
		FSex: char;	// Пол: "м" или "ж"
		FAttr: boolean;		// Булев атрибут
		FComment: string;	// Комментарий

	protected
		procedure SetSex(Value: char);  // Процедура записи
	public
		property Name: string read FName write FName;
		property Depl: string read FDepl write FDepl;
		property Dep2: string read FDep2 write FDep2;
		property Dep3: string read FDep3 write FDep3;
		property Year: word read FYear write FYear;
		property Sex: char read Sex  write SetSex default 'м';
		property Attr: boolean read FAttr write FAttr default true;
		property Comment: string read FComment write FComment;
	end;

implementation
procedure TPerson.SetSex(Value: char);
// Процедура записи пола
begin
	if Value in ['м', 'ж']
	then FSex := Value
	else ShowMessage('Недопустимый символ "' + Value +
	'" в указании пола');
end;
end.
```

Вглядимся в приведенный код. Интерфейсный раздел модуля interface начинается с предложения uses. Заранее включать это предложение в модуль не требуется. Но по мере написания кода вы будете встречаться с сообщениями компилятора о неизвестных ему идентификаторах функций, типов и т.п. Столкнувшись с таким сообщением, надо посмотреть во встроенной справке Delphi или в справке [3], в каком модуле объявлена соответствующая

функция или класс. И включить этот модуль в приложение uses.

Теперь обратимся к объявлению класса. Объявленный класс TPerson наследует непосредственно классу TObject, поскольку родительский класс не указан. В закрытом разделе класса private объявлен ряд полей. Поле FNameпредполагается использовать для фамилии, имени и отчества человека. Поля FDepl, FDep2, FDep3 будут использоваться под указание места работы или учебы. Поле FYear будет хранить год рождения, поле FSex - указание пола: символ "м" или "ж". Поле FAttr будет хранить какую-то характеристику: штатный — нештатный, отличник или нет и т.п. Поле FComment предназначено для каких-то текстовых комментариев. В частности, в нем можно хранить свойство Text многострочного окна редактирования Memo или RichEdit. Так что это может быть развернутая характеристика человека, правда, без форматирования.

В открытом разделе класса public объявлены свойства, соответствующие всем полям. Чтение всех свойств осуществляется непосредственно из полей. Запись во всех свойствах, кроме Sex, осуществляется тоже непосредственно в поля. А для поля Sex указана в объявлении свойства процедура записи SetSex, поскольку надо следить, чтобы по ошибке в это поле не записали символ, отличный от "м" и "ж". Соответственно в защищенном разделе класса protected содержится объявление этой процедуры. Как говорилось ранее, она должна принимать единственный параметр типа, совпадающего с типом свойства.

В раздел модуля implementation введена реализация процедуры записи SetSex. Ее заголовок повторяет объявление, но перед именем процедуры вводится ссылка на класс TPerson, к которому она относится. Не забывайте давать такие ссылки для методов класса. Иначе получите сообщение компилятора об ошибке; Unsatisfied forward or external declaration: TPerson.SetSex — нереализованная ранее объявленная или внешняя функцияTPerson.SetSex.

Тело процедуры SetSex в особых комментариях не нуждается. В нем проверяется допустимость символа, переданного в процедуру через параметр Value. Если символ допустимый, то его значение заносится в поле FSex. Это поле, как и другие закрытые поля, открыто для методов данного класса. При ошибочном символе пользователю выдается сообщение об ошибке. Правда, лучше было бы в этом случае сгенерировать исключение, но пока мы не обсуждали, как это можно делать.

Вы создали класс в вашем модуле MyClasses. Давайте посмотрим, как можно использовать объекты нашего класса. Создайте в модуле формы Unit1 вашего приложения тест класса TPerson. Введите в модуль операторы:

```
uses Classl;
var Pers: TPerson;
```

Они обеспечивают связь с модулем, описывающим класс, и объявляют переменную Pers, через которую вы будете связываться с объектом класса. Но так же, как при работе с другими объектами и записями, объявление этой переменной еще не создает сам объект. Это указатель на объект, и его значение равно nil.

Создание объекта вашего класса TPerson должно осуществляться вызовом его конструктора Create. Так что создайте обработчик события OnCreate вашей формы, и вставьте в него оператор:

```
Pers := TPerson.Create;
```

Вот теперь объект создан, и переменная Pers указывает на него. Чтобы не забыть очистить память от этого объекта при завершении работы приложения, сразу создайте обработчик события OnDestroy формы, и вставьте в него оператор:

```
Pers.Free;
```

Почему ваш объект воспринимает методы Create и Free? Ведь вы их не объявляли в классе TPerson! Это работает механизм наследования. В классе TObject, являющемся родительским для TPerson, методы Create и Free имеются. А поскольку вы их не перегружали, то ваш класс наследует их.

Теперь перенесите на форму четыре окна Edit, окно Memo и две кнопки. Пусть первая кнопка с надписью Запись заносит в объект Pers данные из окон редактирования:' фамилию с именем и

отчеством, пол, подразделение, в котором работает или обучается человек, год рождения, характеристику из окна Memo. Обработчик щелчка на ней может иметь вид:

```
with Pers do
begin
        Name := Editl.Text;
        if Edit2.Text <> '' then Sex := Edit2.Text[1];
        Depl := Edit3.Text;
        Year := StrToInt(Edit4.Text);
        Comment := Memol.Text;
end;
```

А вторая кнопка с надписью Чтение пусть осуществляет чтение информации из объекта в окна редактирования. Обработчик щелчка на ней может иметь вид:

```
with Pers  do
begin
        Editl.Text:= Name;
        Edit2.Text:= Sex;
        Edit3. Text:= Depl;
        Edit4.Text:= IntToStr(Year);
        Memol.Text:= Comment;
end;
```

Выполните ваше приложение. Занесите в окна редактирования какую-то подходящую информацию и щелкните на кнопке Запись. А потом сотрите тексты всех окон редактирования и щелкните на кнопке Чтение. Информация в окнах должна восстановиться. Проверьте также реакцию на неверный символ, задающий пол.

Урок 54 - INI файлы

INI файлы - это удобный способ хранения настроек программы, ini файлы представляют собой простые текстовые файлы доступных для редактирования.

Мы напишем пример, программа будет запоминать размеры и положения формы и восстанавливать их при запуске.

Для начала добавим в секцию uses модуль IniFiles, и создадим глобальную переменную Settings: TINIFile;

В FormCreate напишем код:

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

  Settings:=  TIniFile.Create(GetCurrentDir + '\config.ini');   // Создаем ini файл
  Form1.Top:= Settings.ReadInteger('Form', 'Top', 200);         // Читаем свойство Top

  Form1.Left:= Settings.ReadInteger('Form', 'Left', 200);       // Читаем свойство Left
  Form1.Width:= Settings.ReadInteger('Form', 'Width', 200);     // Читаем свойство Width

  Form1.Height:= Settings.ReadInteger('Form', 'Height', 200);   // Читаем свойство Height
end;
```

Если ничегго не указать в TIniFile.Create то файл создастся в C:\Windows\.

Разберём функцию: ReadInteger(const Section, Ident: string; Default: Longint): Longint;
Section - Имя секции с параметрами
Ident - Имя параметра

Default - Стандартное значение если параметр не был найден

Как видите при компиляции ничего не появилось, а просто форма изменила свои габариты, чтобы появился файл с настройками надо их записать.

В обработчике OnClose Form1 пишем код:

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

  Settings.WriteInteger('Form', 'Top', Form1.Top);       //Записываем свойство Top
  Settings.WriteInteger('Form', 'Left', Form1.Left);     //Записываем свойство Left

  Settings.WriteInteger('Form', 'Width', Form1.Width);   //Записываем свойство Width
  Settings.WriteInteger('Form', 'Height', Form1.Height); //Записываем свойство Height

end;
```

При закрытии параметры буду сохранятся. Попробуйте по растягивайте форму, по перемещайте ее и закройте, затем откройте и все сохранится.

Урок 55 - Реестр Windows, (часть 1/2)

На прошлом уроке мы с вами рассмотрели методику записи и чтения пользовательских данных и данных состояния компонентов в ini файл. Эта методика хранения параметров программы является устаревшей. Рекомендуется для этих целей использовать реестр.
Для начала краткая информация о реестре. Проще говоря, реестр windows, эта огромная база данных, хранящая в себе всевозможные пользовательские и системные данные. Установленное оборудование, драйвера, установленные шрифты, установленные принтеры и многое другое записано в соответствующих разделах.
Как же все-таки увидеть данные реестра? Очень просто. Надо нажать на кнопку "Пуск", выбрать пункт "Выполнить" и ввести название программы REGEDIT, которая отображает реестр для чтения и редактирования. Эта программа находится в каталоге windows. Помните, что некорректное изменение некоторых системных ключей в реестре может привести к постоянным сбоям в системе, может привести даже к полному краху операционной системы, поэтому изменения и удаление следует производить очень осторожно.
Внешний вид этой программы состоит из двух панелей. Левая древовидная и правая, в которой отображаются параметры ключей. Ключ - это элемент реестра, который может содержать некоторые данные или содержать другие ключи. Это древовидная структура, которая хранит в себе вложенные параметры, объединенные общей тематикой. Не буду рассказывать назначение отдельных ключей, это может занять не одну книгу, скажу только, что программы, которые работают с реестром в качестве ini файла, автоматически записывают и читают данные из глобального ключа HKEY_CURRENT_USER. Он же дублируется в ключе HKEY_USERS\ИМЯ_ТЕКУЩЕГО_ПОЛЬЗОВАТЕЛЯ.
Дальше рассмотрим запись и чтение данных. Тут методика аналогична работе с ini файлами. Для начала в разделе подключаемых модулей Uses нужно указать модуль Registry, который необходим для использования команд работы с реестром.
Объявление реестровой переменной

```
Va
r
```

ПЕРЕМЕННАЯ:TRegIniFile;

Создание реестровой переменной, через которую будем читать и писать данные
ПЕРЕМЕННАЯ:=TRegIniFile.Create(НАЗВАНИЕ_КЛЮЧА);
Пример объявления, создания и удаления.

```
procedure TForm1.FormShow(Sender: TObject);
Var

   RegIniFile:TRegIniFile;// реестровая переменная
begin

   RegIniFile:=TRegIniFile.Create('MySelfRegistryApplication'); // создание реестровой
переменной
   RegIniFile.Free;  // уничтожение вручную созданного объекта

end;
```

Подробно о командах чтения и записи.

Чтение и запись целочисленного значения, типа integer:

```
RegIniFile.ReadInteger(СЕКЦИЯ,ПАРАМЕТР,ЗНАЧЕНИЕ_ПО_УМОЛЧАНИЮ);
RegIniFile.WriteInteger(СЕКЦИЯ,ПАРАМЕТР,ЗНАЧЕНИЕ);
```

Дальше аналогично вышерассмотренной команде следует чтение и запись:
двоичного значения ReadBool и WriteBool;
строкового значения ReadString и WriteString;

Если необходимо сохранить данные не в отдельной секции, а в ключе, то вместо параметра СЕКЦИЯ необходимо указать пустую строку, или две кавычки ''.
Сохранение остальных типов данных осуществляется не в секции, а непосредственно в ключе. Для этих команд при обращении к значениям нужно проверять их наличие. Например, если соответствующие параметры не были созданы, то их чтение приведет к ошибке. проверка их наличия производится командой
RegIniFile.ValueExists(ПАРАМЕТР)
Эта команда является функцией, возвращающей истинно (true) или ложно (false), и ее можно использовать в паре с чтением определенного параметра. Например, чтение числа с плавающей точкой:

```
if RegIniFile.ValueExists('MyFloat') then // если данный параметр существует, то
   Edit2.Text:=FloatToStr(RegIniFile.ReadFloat('MyFloat')); // прочитать параметр
```

Продолжение в следуйшем уроке. Там мы рассмотрим пример.

Урок 56 - Реестр Windows, (часть 2/2)

Пример. В новом проекте помещаем в форму следующие компоненты:
·2компонентаCheckBox
· компонент ComboBox, установите в свойстве Items некоторые строки.
· 2 компонента Edit
· компонент DateTimePicker (страница Win32 палитры компонентов)

Процедура OnShow для окна Form1

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

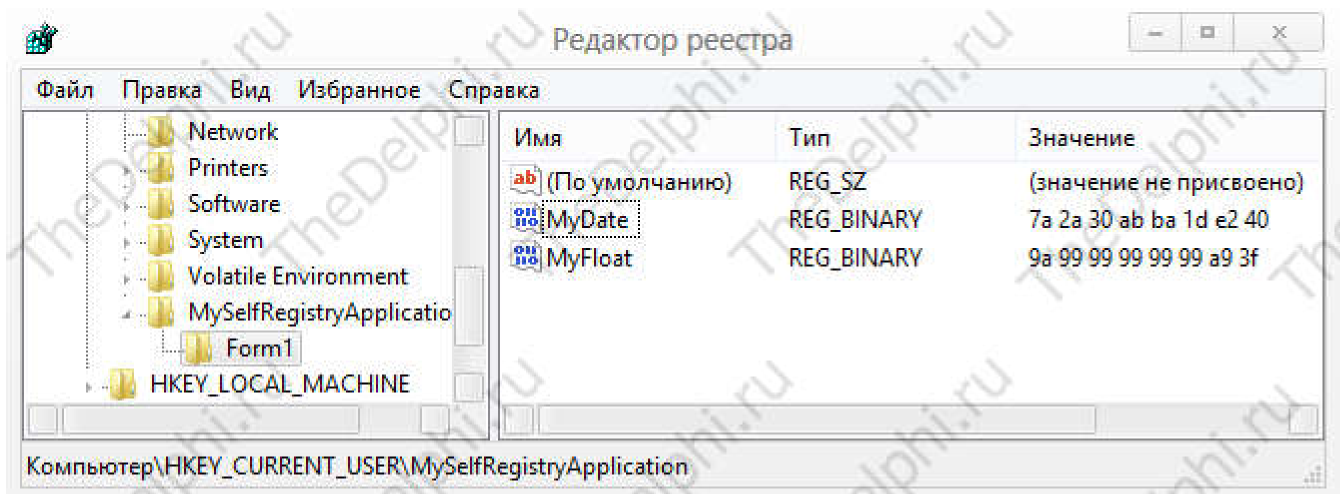
```
Var

   RegIniFile:TRegIniFile; // реестровый объект
begin

   RegIniFile:=TRegIniFile.Create('MySelfRegistryApplication'); // создание реестровой
переменной
   Form1.Left:=RegIniFile.ReadInteger('Form1','Form1Left',Form1.Left); // левая граница окна

   Form1.Top:=RegIniFile.ReadInteger('Form1','Form1Top',Form1.Top); // верхняя граница окна
   Form1.Height:=RegIniFile.ReadInteger('Form1','Form1Height',Form1.Height); // высота окна

   Form1.Width:=RegIniFile.ReadInteger('Form1','Form1Width',Form1.Width); // ширина окна



      // Восстановление состояния компонентов



   CheckBox1.Checked:=RegIniFile.ReadBool('Form1','CheckBox1Checked',CheckBox1.Checked);
   CheckBox2.Checked:=RegIniFile.ReadBool('Form1','CheckBox2Checked',CheckBox2.Checked);

   ComboBox1.ItemIndex:=RegIniFile.ReadInteger('Form1','ComboBox1ItemIndex',ComboBox1.Ite
mIndex);
   Edit1.Text:=RegIniFile.ReadString('Form1','Edit1Text',Edit1.Text);


   if RegIniFile.ValueExists('MyFloat') then // если такой параметр существует, то:

      Edit2.Text:=FloatToStr(RegIniFile.ReadFloat('MyFloat')); // чтение числа с запятой


   if RegIniFile.ValueExists('MyDate') then // если такой параметр существует, то:
      DateTimePicker1.Date:=RegIniFile.ReadDate('MyDate'); // чтение даты


   RegIniFile.Free; // уничтожение вручную созданного объекта

end;
```

Процедура OnClose для окна Form1

```
procedure TForm1.FormClose(Sender: TObject; var Action:
  TCloseAction);

Var
   RegIniFile:TRegIniFile;

begin
   RegIniFile:=TRegIniFile.Create('MySelfRegistryApplication');
```

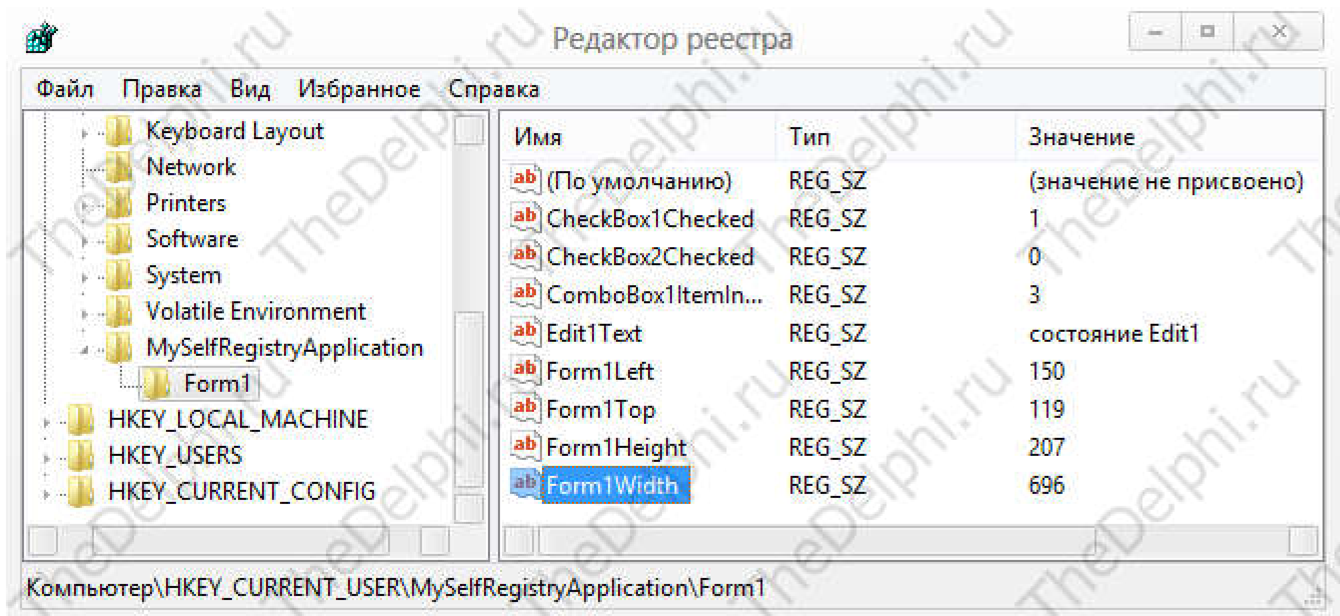
```
    RegIniFile.WriteInteger('Form1','Form1Left',Form1.Left);
    RegIniFile.WriteInteger('Form1','Form1Top',Form1.Top)
;

    RegIniFile.WriteInteger('Form1','Form1Height',Form1.Height);
    RegIniFile.WriteInteger('Form1','Form1Width',Form1.Width);

      // Сохранение состояния компонентов
    RegIniFile.WriteBool('Form1','CheckBox1Checked',CheckBox1.Checked);

    RegIniFile.WriteBool('Form1','CheckBox2Checked',CheckBox2.Checked);
    RegIniFile.WriteInteger('Form1','ComboBox1ItemIndex',ComboBox1.ItemIndex);

    RegIniFile.WriteString('Form1','Edit1Text',Edit1.Text);


    RegIniFile.WriteFloat('MyFloat',StrToFloat(Edit2.Text));
    RegIniFile.WriteDate('MyDate',DateTimePicker1.Date);


    RegIniFile.Free;

end;
```

Обратите внимание, что в этом примере для записи состояния компонента Edit2 используется преобразование строковой величины в число. Если в этом компоненте будут недопустимые символы, то при закрытии окна будет выдаваться ошибка. Для разделения дробной от целой части (запятая), используйте соответствующий символ, установленный в конфигурации windows как символ разделителя.



После запуска и закрытия этой программы можно запустить редактор реестра и просмотреть результат записи данных. Они находятся в ключе HKEY_CURRENT_USER, подключе MySelfRegistryApplication, как было указано в программе. Данные состояния окна, компонентов CheckBox1, CheckBox2 и Edit1 находятся в секции Form1, остальные данные: число в компоненте Edit2, дата в компоненте DateTimePicker1 находятся непосредственно в ключе MySelfRegistryApplication.
Посмотреть на параметры, записанные вышерассмотренной программой автоматически при закрытии, можно на двух рисунках. В первом рисунке вы видите данные даты и дробного

числа, во втором рисунке вы видите содержание секции Form1. Эта секция является одновременно подключом ключа MySelfRegistryApplication.
Удаление того или иного параметра приведет к сбросу сохраняемой величины и при запуске программы будет установлено начальное значение по умолчанию. Удаление всего ключа MySelfRegistryApplication приведет к сбросу всех данных нашей программы. Удалить параметр или ключ в редакторе реестра можно кнопкой Del или выбрав соответствующий пункт из выпадающего меню по правой кнопке мышки.
Программное удаление параметра осуществляется командой
Хочу еще сказать, что с помощью реестра можно сохранять кроме вышерассмотренных типов данных еще формат времени, значение денежной единицы, нетипизированные данные. Последний тип данных может представлять собой шестнадцатиричный код некоторых данных.



Некоторые команды для работы со строками

Вспомним, а для некоторых узнаем некоторые команды для работы со строками.
Строка является набором символов. Строковые переменные можно склеивать, разрезать, копировать, удалять символы.
Для склеивания нескольких строк можно воспользоваться знаком + как в арифметической операции сложения.
Копирование части строки в другую строку производится функцией Copy. Она возвращает результат копирования:

```
1 РЕЗУЛЬТАТ:=Copy(ИСХОДНАЯ_СТРОКА, ПЕРВЫЙ_СИМВОЛ,
  ДЛИНА_ЧАСТИ_СТРОКИ);
```

Удаления части в строке производится процедурой
Delete:

```
Delete(СТРОКА,ПЕРВЫЙ_СИМВОЛ, ДЛИНА);
```

Для вставки части строки в другую применяем процедуру Insert

```
Insert(ВСТАВЛЯЕМАЯ_СТРОКА, РЕЗУЛЬТИРУЮЩАЯ_СТРОКА,
НОМЕР_СИМВОЛА_ВСТАВКИ);
```

В любой момент можно узнать длину строки с помощью функции Length(СТРОКА), а установить длину строки можно процедурой

```
SetLength(СТРОКА, НОВАЯ_ДЛИНА)
```

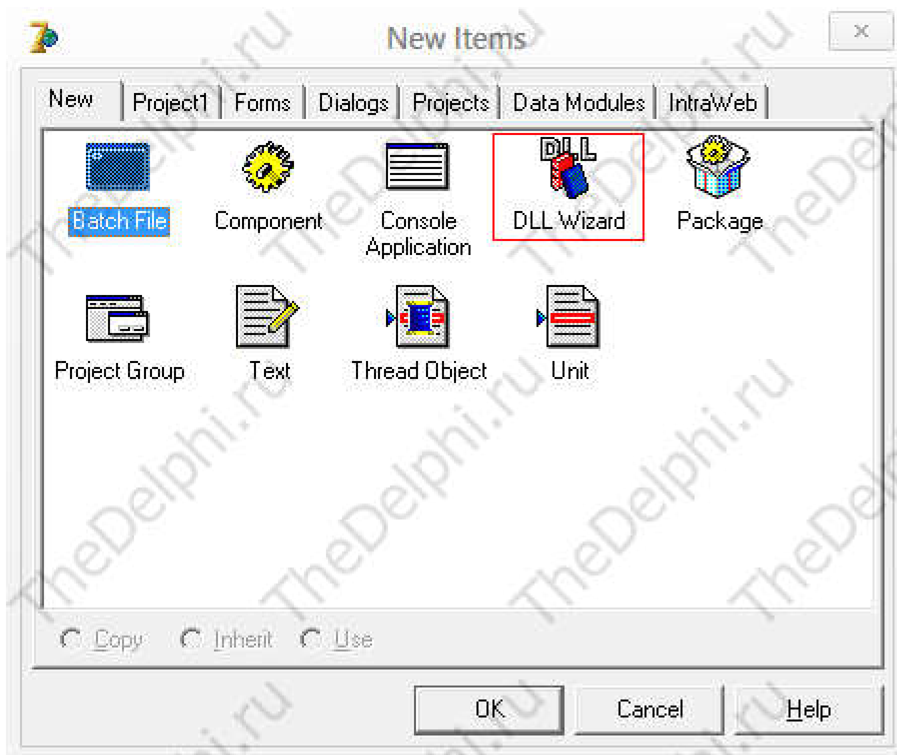
В pascal'е строковая переменная объявляется как тип String. Если объявляем такую переменную, значит мы будем ее использовать для работы со строками. Но в отличие от языка программирования pascal, в таких строках запрещен доступ к нулевому символу (в этом символе хранилось длина строки). Поскольку в операционной системе windows все строки

имеют стандарт PChar (строки, заканчивающиеся символом #0), то тип String здесь оставлен для совместимости. Фактически длина строки String не ограничена 255 символами, как в pascal. Но применяя этот тип, вы незаметно для себя, применяете тип PChar. Все операции перевода одного типа в другой delphi производит автоматически.
Вот пример процедуры обработки строк.

```
procedure StringOper;
Var

   st1,st2,st3,st4:String; // объявление строковых переменных
   i:integer; // целочисленная переменная

begin
   st1:='это 1 строка';

   st2:='это 2 строка';
   st3:=st1+' '+st2; // результат 'это 1 строка это 2 строка'

   st4:=Copy(st3,1,5); // копирование части строки. Результат 'это 1'
   Delete(st4,1,4); // удаление части строки. Результат '1'

   Insert(' строка',st4,2); // вставка части строки в строку st4. Результат '1 строка'
   st4[1]:='2'; // изменение первого символа строки с 1 на 2

   i:=Length(st4); // определение длины строки i=8
   SetLength(st4,7); // установка новой длины строки. Результат '1 строк'

end;
```

Урок 57 - Динамические библиотеки DLL

Приветствую! Сейчас расскажу о библиотеках DLL и о том, как их создавать в Delphi.
В таких языках как C++ библиотеку DLL создать и подключить довольно сложно. Для создания приходится прибегать к WinAPI, а для подключения - к классам.
В Delphi все куда легче. Создается библиотека DLL буквально двумя кликами мыши. Для этого в меню File -> New -> Other выберите DLL Wizard:



Теперь посмотрите на картинку:

Project2.dpr

```
library Project2;

{ Important note about DLL memory management: ShareMem must be the
  first unit in your library's USES clause AND your project's (select
  Project-View Source) USES clause if your DLL exports any procedures or
  functions that pass strings as parameters or function results. This
  applies to all strings passed to and from your DLL--even those that
  are nested in records and classes. ShareMem is the interface unit to
  the BORLNDMM.DLL shared memory manager, which must be deployed along
  with your DLL. To avoid using BORLNDMM.DLL, pass string information
  using PChar or ShortString parameters. }

uses
  SysUtils,
  Classes;

{$R *.res}

begin
end.
```

Тут наш код.

1: 17 Modified Insert

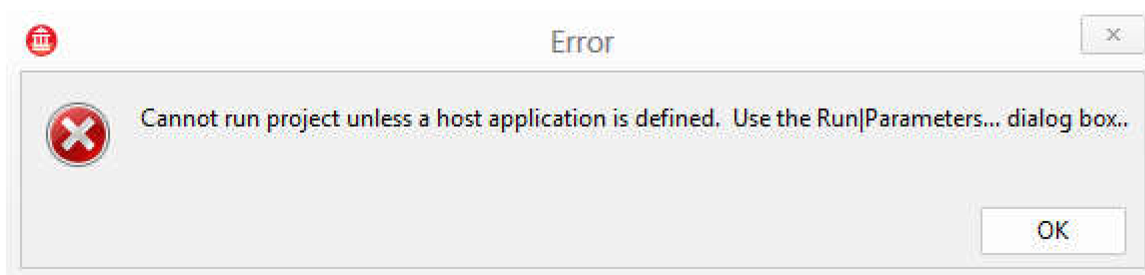
И пропишите такой код (кстати говоря, мы напишем программу, которая проверяет вымышленный пароль):

function checkPassword(password: string): string;

begin

```
  if password = 'admin' then
  begin

    result:= 'correct'
  end

  else
  begin

    result:= 'incorrect'
  end;

end;
exports checkPassword;
```

Теперь сохраните и скомпилируйте. Вы увидите такую ошибку:



Это вполне нормально. Создайте новый формовый проект и киньте на форму 1 Edit, 1 Label и 1 Button. Сохраните проект в той же папке, куда сохраняли DLL.

Потом скопируйте нижеследующий листинг вместо всего вашего кода формовой программы:

```
unit Unit1;


interface


uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

  Dialogs, StdCtrls;


type
  TForm1 = class(TForm)

    Edit1: TEdit;
    Label1: TLabel;

    Button1: TButton;
    procedure Button1Click(Sender: TObject);

  private
```

```
    { Private declarations }

  public
    { Public declarations }

  end;


var
  Form1: TForm1;


implementation


{$R *.dfm}


function checkPassword(password: string): string;

external 'dll.dll';


procedure TForm1.Button1Click(Sender: TObject);
begin

  Label1.Caption:= checkPassword(Edit1.Text);
end;


end.
```

Теперь объясняю:

Здесь мы командой EXTERNAL читаем функцию checkPassword из библиотеки 'dll.dll'. Потом смотрим в Label возвращаемое этой функцией значение (correct или incorrect).

Урок 58 - Работа с сжатыми файлами

В этом уроке мы рассмотрим библиотеку "7z.dll", с её помощью можно распаковывать архивы такого типа:

Zip, BZ2, Rar, Arj, Z, Lzh, 7z, Cab, Nsis, Lzma, Pe, Elf, Macho, Udf, Xar, Mub, Hfs, Dmg, Compound, Wim, Iso, Bkf, Chm, Split, Rpm, Deb, Cpio, Tar, GZip

Практически все существующие архивы. Попытаемся распаковать архив типа Zip. Для этого создадим проект и сам архив, назовем его 1.zip и положим рядом с программой.

На форму вытащим кнопочку и напишем в событии OnClick вот такой код:

```
begin
  // Распаковывает файлы

  with CreateInArchive(CLSID_CFormatZip) do
```

```
  begin

    OpenFile(ExtractFilePath(ParamStr(0)) + '1.zip');
    ExtractTo(ExtractFilePath(ParamStr(0)) + '1');

  end;
end;
```

После, рядом с программой появится папка "1", с содержимым архива. И не забудьте подключить в Uses модуль SevenZip.

Также можно создать архив:

```
begin
 // Добавляет файлы

 with CreateInArchive(CLSID_CFormatZip) do
  begin

    Arch.AddFile('Имя файла');
    Arch.SaveToFile('Имя архива');

  end;
```

Задать уровень сжатия можно функцией SetCompressionLevel(Arch: I7zOutArchive, Level: Integer);. Как видите все просто.

Для работы с другими архивами надо изменить константу CLSID_CFormatZip . Например для .Rar архива, константа будет иметь вид CLSID_CFormatRar, полный список можно увидеть кликнув по константе с зажатым Ctrl.

Урок 59 - Получение хеша файла

Иногда надо проследить, чтобы конечный пользователь не вмешивался в содержимое файла. Например у вас есть программа работа которой напрямую зависит от содержания того или иного файла, пользователь может случайно или намерено изменить файл, те самым вызвав в программе ошибку и получить секретную информацию. Избежать этого нам поможет Хеш файла, это преобразование файла по специальному алгоритму, именуемым "Детерминированным алгоритмом". Преобразовав файл мы получим его уникальный слепок или отпечаток, фиксированной длины, 32 символа. При использовании файла можно проверять его хеш и хеш оригинала, если они совпадают то файл не был изменён и все будет работать нормально.

Приступим, добавим на форму 1 кнопку и 1 Label. Перед тем как писать код подключим в Uses модуль md5. В событии OnClick у кнопки напишем код:

```
begin
  Label1.Caption:= MD5DigestToStr(MD5File('1.txt'));

end;
```

Создадим файл 1.txt и напишем в него "123456789", положим его в папку с проектом и скомпилируем. Нажимаем кнопочку и в Label1 появляется вот такая строка: 25F9E794323B453885F5181F1B624D0B , запомним ее. Теперь допишем в файл еще "0" и опять нажмем кнопку, будет уже: E807F1FCF82D132F9BB018CA6738A19F, строки не совпадают.

E807F1FCF82D132F9BB018CA6738A19F
25F9E794323B453885F5181F1B624D0B

Если бы это был тот файл то программа заметила подмену и оповестила бы об этом.

Урок 60 - Указатели

Как следует из названия, переменные - указатели это особый тип переменных, которые не содержат значения, а указывают на них - на ту ячейку памяти, где они фактически располагаются. И хотя справедливо считается, что использование указателей может приводить к трудно контролируемым ошибкам в программе, всё же переменные - указатели это очень эффективный инструмент для управления объектами в оперативной памяти компьютера.

Конечно, ячейка памяти - это структура размером в один байт. Объекты же, с которыми работает программа, в основном намного большего размера. Соответственно, указатель содержит в себе адрес только первого байта той области оперативной памяти компьютера, где располагается данный объект. Зная тип и соответственно размер объекта, можно прочитать его целиком.

Описание переменных - указателей

Указатель описывается ключевым словом Pointer. По первой букве ключевого слова принято называть переменные - указатели с первой буквы P:

```
var
  PIndexer: Pointer; //не типизированный указатель
```

Ключевым словом Pointer задаётся так называемый не типизированный указатель, по аналогии с не типизированным файлом. Не типизированный указатель содержит просто адрес некой ячейки памяти. Объект, располагающийся начиная с этой ячейки, соответственно может быть совершенно любого типа и размера.

Также в Delphi существуют и типизированные указатели. Они могут указывать на объект соответствующего типа и размера. Именно "могут указывать", потому что это по прежнему адрес одной - первой ячейки области памяти, где располагается объект. И далее его использование в программе зависит от программиста!

Итак, типизированный указатель описывается ключевым словом означающим данный тип, перед которым ставится значок ^:

```
var
  PInteger: ^Integer; //указатель на переменную целого типа

  PText: ^String; //указатель на переменную типа String
```

Также можно описать любой свой тип, и задать переменную-указатель данного типа:

```
type
  TMyType = Record

    X: Integer;
    S: String;

  end;



var
  PMyPointer: ^TMyType;
```

Естественно, можно определить тип, для описания через него переменных - указателей. Делается это в том числе и потому, что, например, в процедурах и функциях в качестве

параметров можно использовать только заранее описанный тип данных. Например, следующее описание задаёт функцию с параметром, являющимся указателем ранее описанного типа, результат которой также является указателем данного типа:

```
type
   TPMyPointer = ^TMyType;


function MyFunc(Point: TMyPointer): TMyPointer;
```

Ну вот, надеюсь теперь вы не боитесь разнообразных вариантов описания таких переменных как указатели.

Использование переменных - указателей

Использование указателей предполагает:

присвоение значения указателю;
изменение значения указателя;
создание области памяти нужного типа и присвоение его адреса указателю;
запись значения в область памяти, адресуемой указателем, и чтение из неё;
освобождение области памяти, адресуемой данным указателем;
применение переменных - указателей;

Описанный указатель без присвоенного значения указывает на совершенно неопределённую ячейку памяти. Попытка использовать такой указатель чревата крахом программы. Поэтому всем указателям нужно явно присваивать значения.

1. Указателю можно присвоить значение другого указателя. В результате оба указателя будут указывать на одну и ту же ячейку памяти. Также указателю можно присвоить пустое значение с помощью ключевого слова nil:

```
var
   P1, P2: Pointer;

begin
   P1:=P2;//Присвоение указателю значения другого указателя

   P2:=nil;>//Присвоение указателю "пустого" значения
end;
```

Указатель со значением nil не адресует никакой ячейки памяти и единственное, что с ним можно сделать - это сравнить с другим указателем или со значением nil.

2. Значение типизированного указателя можно увеличить или уменьшить на размер области памяти, занимаемой объектом данного типа. Для этого служат операции инкремента и декремента:

```
type
   P: ^Integer;

begin
   inc(P); //увеличение значения указателя на 4 байта (размер типа Integer)

   dec(P); //уменьшение значения указателя на 4 байта (размер типа Integer)
end;
```

Попытка выполнить операциии inc либо dec с не типизированным указателем вызовет ошибку на этапе компиляции, так как компилятору неизвестно, насколько именно изменять значение указателя.

3. Процедурой New можно создать область памяти соответствующего типа и присвоить её адрес указателю (инициировать указатель):

```
var
   PInt: ^Integer;

begin
   New(PInt);  //Указатель PInt получает значение адреса созданной области памяти типа Integer

end;
```

Поскольку с областью памяти, созданной с помощью процедуры New, не связана ни одна переменная, но там содержится реальное используемое значение, то можно считать, что это значение связано с некой "безымянной переменной". Обращаться к ней по имени переменной невозможно, а можно оперировать только используя указатель.

Также задать указателю адрес объекта можно с помощью операции, называемой "взятие адреса", которая обозначается значком @. При этом создавать область памяти уже не нужно, так как она создана предварительным описанием данного объекта:

```
var
   MyVar: TMyType;  //Описание переменной, при этом выделяется область памяти
соответствующего размера

   P: ^TMyType;
beginr>

   P:=@MyVar;  //Указатель получает адрес области памяти, занимаемой переменной MyVar
end;
```

4. Если область памяти уже создана и её адрес присвоен указателю, то в ячейку памяти, адресуемую данным указателем, можно записать значение объекта, соответствующего типу указателя. Для этого служит операция, обозначаемая также значком ^, стоящим после имени указателя, например: P^. Эта операция называется "разыменование указателя". Также с помощью этой операции со значением в данной ячейке памяти можно делать всё что нужно:

```
var
   MyVar: Integer;

   P: ^Integer;
begin

   P:=@MyVar;  //Указатель получает адрес области памяти, занимаемой переменной MyVar
   P^:=2;  //В ячейку памяти по адресу переменной MyVar записывается значение 2

   Form1.Caption:=IntToStr(P^+3);  //В заголовке Формы появится число 5
end;
```

С обычными переменными всё просто, но возникает вопрос, как получить значение по адресу указателя, если тип переменной - запись с несколькими полями? Аналогично:

```
type
   TMyRec = Record

      N: Integer;
```

```
      S: String;

    end;


 var
    MyRec: TMyRec;

    PRec: ^TMyRec;


 begin
    PRec:=@MyRec;  //Указатель получает адрес области памяти, занимаемой переменной MyRec

    PRec^.S:='Строка данных';  //С помощью указателя производится изменение строкового поля записи
    PRec^.N:=256;  //С помощью указателя производится изменение числового поля записи

 end;
```

А теперь уберите стрелку возле PRec: PRec.S:='Строка данных'; Вы увидите, что никакой ошибки ни компилятор, ни выполнение программы не показали! Выходит, выражения PRec^.S и PRec.S аналогичны.

Далее, а как получить значение, если указатель это элемент массива, например:

```
 var
    PArray: Array[1..100] of ^Integer;

    X: Integer;
```

Ни PArray^[10], ни PArray[10]^ не являются правильными выражениями. Ну конечно, нужно использовать скобки:

```
 X:=(PArray[10])^;
```

5. Память, выделенную процедурой New, всегда нужно явно освобождать. Освободить область памяти, адресуемую указателем, инициированным с помощью New, можно процедурой Dispose:

```
 var
    MyVar: TMyType;

    P: ^TMyType;
 begin

    P:=@MyVar;
    Dispose(P);  //Освобождение области памяти, адресуемой указателем P

 end;
```

При выполнении процедуры Dispose указатель снова приобретает неопределённое значение, не равное даже nil, и его использование может привести к неопределённым результатам, даже к краху программы.

6. Указатели особенно эффективны для манипуляций с объектами большого размера, например, записями с многочисленными полями. Для сортировки массива таких записей создаётся массив указателей, каждый из элементов которого указывает на соответствующий элемент массива записей. И вместо перемещения элемента массива записей перемещается соответствующий элемент массива указателей, на что требуется гораздо меньше времени.

Урок 61 - Создание и использование интерфейса (часть 1/2)

Интерфейсы играют главную роль в технологиях COM (Component Object Model - компонентная модель объектов),CORBA (Common Object Request Broker Architecture - архитектура с брокером требуемых общих объектов) и связанных с ними технологиях удаленного доступа, т. е. технологиях доступа к объектам, расположенным (и выполняющимся) на другой машине. Их основная задача - описать свойства, методы и события удаленного объекта в терминах машины клиента, т. е. на используемом при разработке клиентского приложения языке программирования. С помощью интерфейсов программа клиента обращается к удаленному объекту так, как если бы он был ее собственным объектом.

Тема интерфейсов достаточно обширна и интересна. В этом уроке даются лишь самые общие сведения об интерфейсах. Сведение этой темы в одну главу с классами не случайно, т. к. интерфейс представляет собой пустой класс, т. е. класс, в котором провозглашены, но никак не расшифрованы свойства и методы.

Интерфейсы представляют собой частный случай описания типов. Они объявляются с помощью зарезервированного слова interface. Например:

```
 type
   IEdit = interface

     procedure Copy; stdcall;
     procedure Cut; stdcall;

     procedure Paste; stdcall;
     function Undo: Boolean; stdcall;

   end;
```

Такое объявление эквивалентно описанию абстрактного класса в том смысле, что провозглашение интерфейса не требует расшифровки объявленных в нем свойств и методов.

В отличие от классов интерфейс не может содержать поля, и, следовательно, объявляемые в нем свойства в разделах read и write могут ссылаться только на методы. Все объявляемые в интерфейсе члены размещаются в единственной секции public. Методы не могут быть абстрактными (abstract), виртуальными (virtual), динамическими (dynamic) или перекрываемыми (override). Интерфейсы не могут иметь конструкторов или деструкторов, т. к. описываемые в них методы реализуются только в рамках поддерживающих их классов, которые называются интерфейсными.

Если какой-либо класс поддерживает интерфейс (т. е. является интерфейсным), имя этого интерфейса указывается при объявлении класса в списке его родителей:

```
 TEditor = class(TInterfacedObject, IEdit)
   procedure Copy; stdcall;

   procedure Cut; stdcall;
   procedure Paste; stdcall;

   function Undo: Boolean; stdcall;
```

```
end;
```

В отличие от обычного класса интерфейсный класс может иметь более одного родительского интерфейса:

```
type
  IMylnterface = interface procedure Delete; stdcall;

  end;
  TMyEditor = class(TInterfacedObiect, lEdit, IMylnterface)

    procedure Copy; stdcall;
    procedure Cut; stdcall;

    procedure Paste; stdcall;
    function Undo: , Boolean; stdcall;

    procedure Delete; stdcall;
  end;
```

В любом случае в разделе реализации интерфейсного класса необходимо описать соответствующие интерфейсные методы. Если, например, объявлен интерфейс

```
IPaint = interface
  procedure CirclePaint(Canva: TCanvas; X, Y, R: Integer);

  procedure RectPaint(Canva: TCanvas; X1, Y1, X2, Y2: Integer);
end;
```

и использующий его интерфейсный класс

```
TPainter = class(TInterfacedObject, IPaint)
  procedure CirclePaint(Canva: TCanvas; X, Y, R: Integers);

  procedure RectPaint(Canva: TCanvas; X1, Y1, X2, Y2: Integer);
end;
```

то в разделе implementation следует указать реализацию методов:

```
procedure TPainter.CirclePaint(Canva: TCanvas; X, Y, R: Integers;
begin

  with Canva do
    Ellipse(X, Y, X + 2 * R, Y + 2 * R);

end;
```

procedure TPainter.RectPaint(Canva: TCanvas; X1, Y1, X2, Y2: Integer); begin with Canva do Rectangle(XI, Yl, X2, Y2) end; Теперь можно объявить интерфейсный, объект класса TPainter, чтобы с его помощью нарисовать окружность и квадрат:

```
procedure TForm1.PaintBoxIPaint(Sender: TObject);
var

  Painter: IPaint;
begin

  Painter := TPainter.Create;
```

```
  Painter.CirclePaint(PaintBoxl.Canvas, 10, 0, 10);

  Painter.RectPaint(PaintBoxl.Canvas, 40, 0, 60, 20);
 end;
```

Урок 62 - Создание и использование интерфейса (часть 2/2)

И так, продолжим. Несмотря на то что интерфейс всегда объявляется до объявления использующего его интерфейсного класса и, следовательно, известен компилятору, его методы обязательно должны быть перечислены в объявлении класса. В нашем случае простое указание

```
 type
  TPainter = class(TInterfacedObject, IPaint)

  end;
```

было бы ошибкой: компилятор потребовал бы вставить описание методов CirclePaint и RectPaint. Подобно тому как все классы в Object Pascal порождены от единственного родителя TObject, все интерфейсные классы порождены от общего предка TInterfacedObject. Этот предок умеет распределять память для интерфейсных объектов и использует глобальный интерфейс lunknow:

```
 type
  TInterfacedObject = class(TObject, lUnknown)private

    FRefCount: Integer;
  protected

    function Querylnterface(
      const IID: TGUID; out Obj): Integer; stdcall;

    function _AddRef: Integer;
 stdcall;
    function _Release: Integer; stdcall;

  public
    property RefCount: Integer read FRefCount;

  end;
```

Если бы в предыдущем примере класс TPainter был описан так:

```
 TPainter = class(IPaint)
  procedure CirclePaint(Canva: TCanvas; X, Y, R: Integer);

  procedure RectPaint(Canva: TCanvas; X1, Y1, X2, Y2: Integer);
 end;
```

компилятор потребовал бы описать недостающие методы Queryinterface, _Add И _Release класса TInterfacedObject. Поле FRefCount этого класса служит счетчиком вызовов интерфейсного объекта и используется по принятой в Windows схеме: при каждом обращении к методу Add интерфейса IUnknow счетчик наращивается на единицу, при каждом обращении к Release - на единицу сбрасывается. Когда значение этого поля становится равно 0, интерфейсный объект уничтожается и освобождается занимаемая им память. Если интерфейс предполагается использовать в технологиях COM/DCOM или CORBA, его методы должны описывать с директивой stdcall или (для объектов Автоматизации) safecall К интерфейсному объекту можно применить оператор приведения типов as, чтобы использовать нужный

интерфейс:

```
 procedure PaintObjects(P: TInterfacedObject)
 var

   X: IPaint;
 begin

   try
     X := P as IPaint;

     X.CirclePaint(PaintBoxl.Canvas, 0, 0, 20)
   except

     ShowMessage('Объект не поддерживает интерфейс IPaint')
   end

 end;
```

Встретив такое присваивание, компилятор создаст код, с помощью которого вызывается метод Queryinterfaceинтерфейса IUnknow с требованием вернуть ссылку на интерфейс IPaint. Если объект не поддерживает указанный интерфейс, возникает исключительная ситуация. Интерфейсы, рассчитанные на использование в удаленных объектах, должны снабжаться глобально-уникальным идентификатором (guiD). Например:

```
 IPaint = interface
   ['{A4AFEB60-7705-11D2-8B41-444553540000}']

   procedure CirclePaint(Canva: TCanvas; X, Y, R: Integer);
   procedure RectPaint(Canva: TCanvas; Xl, Yl, X2, Y2: Integer);

 end;
```

Глобально-уникальные идентификаторы создаются по специальной технологии, гарантирующей ничтожно малую вероятность того, что два guid совпадут. Эта технология включена в Windows 32: чтобы получить guid для вновь созданного интерфейса в среде Delphi, достаточно нажать клавиши Ctrl+Shift+G. Для работы с guid в модуле System объявлены следующие типы:

```
 type
   PGUID = ^TGUID;

   TGUID = record Dl: LongWord;
     D2: Word;

     D3: Word;
     D4: array[0..7] of Byte;

   end;
```

Программист может объявлять типизированные константы типа tguid, например:

```
 const IID_IPaint: TGUID= ['{A4AFEB61-7705-11D2-8B41-444553540000}'] ;
```

Константы guid могут использоваться вместо имен интерфейсов при вызове подпрограмм. Например, два следующих обращения идентичны:

```
 procedure Paint(const IID: TGUID);
```

```
 Paint(IPaint) ;
 Paint(IID_Paint);
```
С помощью зарезервированного слова implements программист может делегировать какому-либо свойству некоторого класса полномочия интерфейса. Это свойство должно иметь тип интерфейса или класса. Если свойство имеет тип интерфейса, имя этого интерфейса должно указываться в списке родителей класса, как если бы это был интерфейсный класс:
```
 type
   IMylnterface = interface procedure P1; procedure P2;

   end;
   TMyClass = class(TObject, IMylnterface)

     FMyInterface: IMylnterface;
     property Mylnterface: IMylnterface

       read FMyInterface implements IMylnterface;
   end;
```
Обратите внимание: в этом примере класс TMyciass не является интерфейсным, т. е. классом, в котором исполняются методы p1 и P2. Однако если из него убрать определение уполномоченного свойства Mylnterface, он станет интерфейсным, и в нем должны быть описаны методы интерфейса IMylnterface. Уполномоченное свойство обязательно должно иметь часть read. Если оно имеет тип класса, класс, в котором оно объявлено, не может иметь других уполномоченных свойств.

Урок 63 - Работа с реестром

Для начала определимся, что же такое реестр Windows. Форумлировка корпорации Microsoft: Иерархически построенная, централизованная база данных в составе операционных систем Microsoft Windows 9x/NT/2000/XP/2003/Vista/7/8, содержащая сведения, которые используются операционной системой для работы с пользователями, программными продуктами и устройствами. Реестр хранит данные о продуках, установленных в вашей ОС. Новая Windows 7 также не осталась в стороне от прошлых систем и обзовелась реестром. Так что если у вас Windows 7 к вам урок также относится.

И так давайте определимся для чего программа написанная на Delphi(необязательно на Delphi 7) может хранить данные в реестре Windows. Самые простые способы использования: Загрузка из реестра: языка(русский например или английский), фоновый цвет программы, хранение данных о версии, об обновлении, о регистрации и о многом другом.

Если Вы программируете, либо программировали на Delphi 7, то я уверен на сто процентов, Вы не пользовалась реестром. В этом уроке я Вам покажу и детально расскажу, как из программы, написанной на Delphi, добавить запись в реестр.

Основной код для добавления в реестр данных(для меня самый лёгкий из все, что я знаю):
Перед тем как начать работать:

Откройте Delphi->Создайте новый проект->В uses проекта пропишите registry.
```
var
RegIniFile:TRegIniFile; //объявляем переменную
begin
edit1.text:='1';//edit1 равен 1
RegIniFile:=TRegIniFile.Create('***');// Можно, конечно, использовать и ***, но рекомендую
```

Вам поменять ***, на любое слово на английском языке, либо цифр

```
RegIniFile.WriteString('***1','***2',edit1.text);
reginifile.free;
end;
```

Вместо edit1.text можно записать и число и слово, но лучше на форму выставить edit1, а edit1 поставить в его настройках visible: true и написать что либо на edit1, в этом случае из edit1 в реестр будут записаны данные.

В данном случае в реестр в ключ HKEY_CURRENT_USER->***->***1->***2->1, в параметр подключа в данном случае будет записана цифра 1, т.к edit1.text:='1', а можно сделать так, чтобы программа записывала тот текст в реестр, который записан в поле. Мы записали в реестр данные, а что теперь. Теперь с помощью записанной единички в реестре, мы можем указать форме, то что будет происходить, например, если ключ полностью совпадет(можно менять форме цвет или язык и много другое). Давайте поменяем цвет. В настройках формы найдите onshow, кликните два раза по пустой клетке

Мы получим следующее:

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

end;
```

Теперь убираем begin и вставляем этот код, этот код будет читать данные из реестра Как Вы наверное понимаете, данные в скобках должны совпадать, с тем, что мы записавали ранее.

```
var // объявляем переменные
RegIniFile:TRegIniFile;
begin
regIniFile:=TRegIniFile.Create('***');
Edit1.Text:=RegIniFile.ReadString('***1','***2',form1.Edit1.Text);
reginifile.Free ;
```

Так данные из реестра мы загрузили. Теперь нам необходимо объявить нашей программе, что ей надо делать после загрузки

После вставляем этот код:

```
if edit1.text='1' then begin
form1.color:=clblue
end
else
form1.color:=clred;
```

Здесь, мы написали программа и объяснили ей, что надо делать, объясню Вам: Если едит равен 1(из реестра), то цвет программы синий, если нет красный.(в нашем коде так будет всегда, хотя модернизировать код совсем не трудно).

Урок 64 - Использование потоков данных (часть 1/3)

Чтобы легче себе представить что такое поток можно представить себе магнитную ленту. Чтобы прочитать что-то с нее мы опускаем считывающую головку и читаем данные определенного объема или пока не дойдем до конца ленты. Тоже самое при записи. Если запись производить несколько раз (блок за блоком), то информация будет ложиться на ленту последовательно блок за блоком. После записи или чтения позиция считывающей головки останется там же где она осталось после предыдущей операции. Вот и все ограничения! Внутри это может быть реализовано совершенно по разному. В перерывах между чтением данных данных хранилище может шириться (например подгружаться из интернета или из диска), можно сделать бесконечную ленту, т.е. при переходе в конец потока подставить ему начало

хранилища, можно реализовать принцип конвейера, т.е. хранилище маленькое, но в него постоянно подкладывают в конец (очередь), а забирают всегда поочереди. Это весьма удобный механизм при работе с одиночными объектами, коллекциями, массивами объектов и даже деревьями. Все начинается именно с TStream. В делфи есть много классов его наследников таких как TMemoryStream, TFileStream, TResourceStream, TCompressionStream, TStringStream и другие. Их объединяют механизмы удобной быстрой загрузки, сохранения, добавления и обрезки данных. Но их объединяет не только логика но и реализация, реализация класса TStream. Забегая наперед скажу что сам класс "недореализован", т.е. нельзя создать объект такого класса (так как он абстрактный), но кой чего он умеет.

Как это ни странно, но начнем именно с того чего он не умеет. Не умеет он то что касается низкоуровневого доступа к самим данным. Оно и понятно, наследников много всем не угодишь, но при этом он "знает" как это должны делать его наследники. Под знает я имею ввиду что часть его методов виртуальные и даже абстрактные (без реализации). Работает это так. Пусть мы создаем собственный класс. Наш класс занимается обработкой данных. Может получать их из некоторого источника и после обработки сохранять или передавать дальше по цепочке. Однако источников данных может быть много, например из файла, из ресурса, или просто из буфера в памяти. Если решать это задачу в лоб, то нам понадобилось бы создавать 3 разных метода, для получения данных из перечисленных источников. Используя класс TStream, нам досрочного сделать всего один. Часто его объявляют так
Procedure TMyClass.LoadFromStream(AStrem : TStream);
Это совсем не значит, что мы ожидаем что нам в функцию передадут объект класса TStream (Это невозможно! Такой класс создать нельзя в принципе). Вместо TStream, нам подойдет любой полноценный класс где реализованы все возможности ввода и вывода (например TFileStream, TResourceStream и т.д.). Это возможно потому, что все эти классы являются наследниками TStream. А мы у себя считая что работаем TStream, на самом деле будем вызывать методы того класса который в действительности был создан (это то что я имел ввиду когда писал, что TStream "знает" как это должны делать его наследники). Это и есть полиморфизм на практике т.е. наследник и предок реагируют похоже на одинаковые события. Чтобы такое было возможно метод должен объявлен как виртуальный или абстрактный. Если метод был объявлен как виртуальный, то производимое методом действие будет заменено действием того объекта, который был реально создан, если же он объявлен как абстрактный, то изначально предок ничего и не делал, но по его названию можно догадаться что бы он хотел сделать. Мы уже говорили, что механизм виртуальных методов позволяет подменять действие, но бывает так что класс предназначается только для подмены своего наследника (именно такой TStream), тогда метод предка никогда не будет вызываться, вот как раз в таком случае виртуальные методы делают абстрактными (т.е. без своей реализации).

После прочтения предыдущего абзаца у вас уже достачно знаний, чтобы разобрать структуру класса

```
TStream = class(TObject)
  private
    function GetPosition: Int64;
    procedure SetPosition(const Pos: Int64);
    procedure SetSize64(const NewSize: Int64);
  protected
    function GetSize: Int64; virtual;
    procedure SetSize(NewSize: Longint); overload; virtual;
    procedure SetSize(const NewSize: Int64); overload; virtual;
  public
    function Read(var Buffer; Count: Longint): Longint; virtual; abstract;
    function Write(const Buffer; Count: Longint): Longint; virtual; abstract;
    function Seek(Offset: Longint; Origin: Word): Longint; overload; virtual;
    function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64; overload; virtual;
    procedure ReadBuffer(var Buffer; Count: Longint);
    procedure WriteBuffer(const Buffer; Count: Longint);
```

```
    function CopyFrom(Source: TStream; Count: Int64): Int64;
    function ReadComponent(Instance: TComponent): TComponent;
    function ReadComponentRes(Instance: TComponent): TComponent;
    procedure WriteComponent(Instance: TComponent);
    procedure WriteComponentRes(const ResName: string; Instance: TComponent);
    procedure WriteDescendent(Instance, Ancestor: TComponent);
    procedure WriteDescendentRes(const ResName: string; Instance, Ancestor: TComponent);
    procedure WriteResourceHeader(const ResName: string; out FixupInfo: Integer);
    procedure FixupResourceHeader(FixupInfo: Integer);
    procedure ReadResHeader;
    property Position: Int64 read GetPosition write SetPosition;
    property Size: Int64 read GetSize write SetSize64;
  end;
```

Начнем анализ класса с методов Read и Write. Они абстрактные, значит реализованы в наследнике. Служат они для чтения данных из текущей позиции потока наследника TStream и записи в него соответственно. Первый параметр это сам буфер данных (внимание не указатель, а сама переменная буфера!). Второй размер буфера в байтах. Результат число переданных данных. Это базовые методы ввода/вывода. На них построена вся передача данных. Их удобство заключается в том, что при правильном использовании вы никогда не прочитаете данных больше чем вам может предоставит источник и не запишите в никуда в случае если приемник данных не готов в данный момент их принять. Если что-то не так то результат функция всегда вернет нуль или значение отличное от того что вы ей передали в поле Count. На этом удобства не заканчиваются. Если вы уже умеете использовать такой гибкий механизм как исключения, то вы должны оценить методы ReadBuffer, WriteBuffer, они вызывают Read и Write для передачи данных, но в случае неудачи вызывают исключения EReadError и EWriteError соответственно. Это нам дает возможность грамотно выйти из непредвиденной ситуации и объяснить пользователю, почему его действия не увенчались успехом.

Урок 65 - Использование потоков данных (часть 2/3)

Пример: Создаем пустой текстовый файл на диске С: с именем "1.txt".

```
procedure SeadFromStream(AStream : TStream);
var
  i         : Integer;
Begin
  try
    AStream.ReadBuffer(i, sizeof(i));
  except
    on EReadError do ShowMessage('Ошибка чтения данных');
    else ShowMessage('Неизвестная ошибка ');
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  F : TStream;
  i : Integer;
begin
  Try
    F := TFileStream.Create('C:\1.txt', fmOpenRead);
    SeadFromStream(F);
    F.Free;
  finally end;
end;
```

Под отладкой все равно вылетают исключения, но так бывает только под отладкой... Кто повнимательнее заметил заметил что открытие файла тоже в защищенной секции. Что поделаешь, отстуствие файла это тоже ошибка, но тут я ее не стал отдельно отрабатывать, так как для исследуемой функции LoadFromStream не имеет значения к какому из потоков принадлежит переданный ей объект. Уточню, что методы ReadBuffer и WriteBuffer

определенны уже в TStream. Т.е. если вы вдруг захотите написать своего наследника TStream, для своего специфического источника данных, то вам не нужно будет их переопределять самому(ой). Говоря об операциях чтения и записи следует уточнить, что не все наследники TStream позволят вам писать/читать данные одновременно. Например, в ресурс нельзя ничего писать, а из файла открытого для записи бесполезно читать. Это значит что если класс имеет метод WriteToStream, то передавать ему объект класса TResourceStream будет неправильно.

Метод Seek устанавливает позицию чтения/записи в потоке, хотя он и реализован в TStream, но его ОБЯЗАНЫ перекрывать все наследники. Он существует в 2х вариациях:
function TStream.Seek(Offset: Longint; Origin: Word): Longint;
function TStream.Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
Перекрыть нужно хотя бы один. Если один из них не будет реализован, то при необходимости всегда будет вызван другой. Если не будет ни одного, то при попытке изменить позицию чтения произойдет исключение. Метод Seekпозволяет производить установку позиции от начала потока (параметр Origin = soBeginning), от текущей позиции (Origin = soCurrent), или от конца(Origin = soEnd). Для получения позиции от начала хранилища удобно пользовать свойством Position. Еще один метод, который часто нужно перекрывать в наследнике
это SetSize
procedure TStream.SetSize(NewSize: Longint);
Он хоть и не абстрактный но пустой. Его сделали пустым на случай если вдруг ваш поток не поддерживает операции записи (это допустимо). В этом случае, было бы нелогичным требовать от наследника TStream реализовывать возможность изменения размера его внутреннего хранилища данных, тогда как он не предусматривает такую операцию. В простых случаях линейных хранилищ Функцию GetSize можно не переопределять. Она самостоятельно использует метод Seek для получения размеров хранилища. При использовании объектов класса следует использовать публичное свойство Size.

Теперь о реализованных высокоуровневых возможностях. Копирование больших фрагментов (более 100кб)
function CopyFrom(Source: TStream; Count: Int64): Int64;
Замечательная функция, она нам позволит скопировать большой блок данных из чужого потока, начиная с текущей позиции в потоке Source, при этом начнет писать данные, начиная с текущей позиции в нашем потоке. Для копирования будет выделен внутренний буфер размером в 60 кб, позволяющий сравнительно большими блоками копировать данные, при этом не занимая надолго сам источник данных (источник может оказаться неспособным к работе с несколькими процессами). И еще если в качестве Count указать 0, то функция поймет что нужно скопировать весь поток от начала (Внимание! Не от текущей позиции) до конца. Весьма удобная функция, при перекачке данных из медленного источника в быстрый или из ReadOnly потока в поток доступный для записи. Сериализация объектов. Сериализация это перевод структур данных (в данном случае классов наследников TComponent) в бинарное представление. В таком виде данные объектов можно передать из программы или сохранить на диске. Тут же уточняю сохраняться лишь published свойства. Потому если вы хотите использовать такой механизм в своей программе, то в published секцию следует перенести все свойства определяющие состояние объекта.
function ReadComponent(Instance: TComponent): TComponent;
procedure WriteComponent(Instance: TComponent);
Можно записывать несколько объектов подряд, сохранять картинки, тесктовые файлы и много чего другого. Кстати, таким же образом создается любая форма, только источником для нее становиться TResourceStream. Файл DFM сохраняется в ресурс, на лету конвертируется из текстового ресурса в бинарный и становиться доступным для загрузки функцией ReadComponent. Однако перед использованием этих функций нестандартные компоненты следует зарегистрировать при помощи функции RegisterClass. (Подробный пример см. в справке по Delphi "TStream.ReadComponent Method")

Дополнительные возможности. Реализация интерфейса IStream. Связь с COM. Наверное некоторые уже заметили созвучность имени класса TStream и COM интерфейса IStream. IStream

- это универсальный интерфейс приема-передачи данных в Windows. Если бы нам удалось подружить TStream и IStream, то задачу передачи данных между двумя сторонними объектами можно было бы считать решенной. Например, передать картинку из потока TFileStream в объект класса Bitmap библиотеки GDI+. Аналогичная задача стоит при обмене данными между приложением и своей собственной Dll. Как известно в Dll нельзя передавать объекты классов, а интерфейсы очень даже можно, при этом идеально сохраняется принцип инкапсуляции класса и объектный подход к разарботке приложений. Ближе к делу. Для того чтобы получить интерфейс IStream имея объект наследник TStream необходимо дополнительно воспользоваться классом TStreamAdapter, который и реализует IStream, но не сам, а с помощью класса наследника TStream. Вот его кноструктор.

```
constructor Create(Stream: TStream; Ownership : TStreamOwnership = soReference);
```

Видно, что при создании объект должен принять экземпляр класса наследника TStream, но кроме того у него есть еще такой замечательный параметр Ownership, который указывает нужно ли уничтожать сам объект потока (тот который передали ему в конструкторе Stream: TStream). Напомню, что после того как интерфейс освобождается (переменной интерфейса присваивают nil) и больше не остается ссылок на этот объект (ссылока значит то, что у этого объекта получали интерфейс), сам объект, который релизовывал интерфейс уничтожается. Уничтожается, ну и пусть, но ведь после уничтожения у нас останеться неуничтоженным сам объект потока (наследник TStream, который и реализовывал передачу данных). Уничтожить самостоятельно мы его не можем, ведь мы не знаем, когда клиент прекратит с ним работать. Вот для это и нужно в качестве Ownership передвать значение soOwned, т.е. адптер владеет экземпляром потока и удалит его, перед тем как самоуничтожиться.

Урок 66 - Использование потоков данных (часть 3/3)

Звучит это все страшно, а используется очень просто. Вот пример.

```
var
  strm : IStream;
  mem : TMemorySteam;
Begin
  mem := TMemorySteam.Create;
  strm := TStreamAdapter.Create(mem, soOwned) as IStream;
```

Все теперь передаем интерфейс strm в Dll ил вообще куда угодно и не думаем о уничтожении ни TStreamAdapter ни TMemorySteam, в нужное время они сами уничтожаться, тогда когда клиент закончит с ними работать.

Теперь рассмотрим случай, когда программа на делфи является не сервером, а клиентов, т.е. ей передают интерфейс IStream, объекта потока созданного неизвестно где. Идеально было бы если бы мы получили не IStream, а TStream, ведь все объекты VCL умеют работать именно с TStream, а работать с IStream они не умеют. Решение в лоб состоит в том, чтобы банально скопировать все данные из IStream например в TMemorySteam. Однако, таким образом мы получаем лишнее копирование. Это ничего, когда данных на 20кб, другое дело если их 1Гб, тогда дополнительное копирование выйдет боком. Для таких случаев предлагаю написать свой собственный адаптер, но уже не адаптер сервера, а клиента. Класс назвал TExternStream (от слова External т.е. внешний). Заодно это будет пример написания своего класса наследника TStream. Основная задача этого класса переадресовать все запросы интерфейсу внешнего объекта IStream. Сам интерфейс нужно передать в конструктор. В деструкторе интерфейс освобождается FSource := nil; в классе перекрыты все абстрактные методы Read, Write, виртуальный Seek, GetSize и SetSize.

```
type
  TExternStream = class(TStream)
  protected
    FSource : IStream;
    procedure SetSize(const NewSize: Int64); override;
  public
    constructor Create(Source : IStream);
    destructor Destroy; override;
    function Read(var Buffer; Count: Longint): Longint; override;
```

```
    function Write(const Buffer; Count: Longint): Longint; override;
    function Seek(const Offset: Int64; Origin: TSeekOrigin): Int64; override;
  end;
procedure TForm1.FormCreate(Sender: TObject);
var
  T : TStreamAdapter;
begin
  RegisterComponents
end;
{ TExternStream }
constructor TExternStream.Create(Source: IStream);
begin
  inherited Create;
  FSource := Source;
end;
destructor TExternStream.Destroy;
begin
  FSource := nil;
  inherited;
end;
function TExternStream.Read(var Buffer; Count: Integer): Longint;
begin
  if FSource.Read(@Buffer, Count, @Result) <> S_OK
  then
    Result := 0;
end;
function TExternStream.Seek(const Offset: Int64; Origin: TSeekOrigin): Int64;
begin
  FSource.Seek(Offset, byte(Origin), Result);
end;
procedure TExternStream.SetSize(const NewSize: Int64);
begin
  FSource.SetSize(NewSize);
end;
function TExternStream.Write(const Buffer; Count: Integer): Longint;
begin
  if FSource.Write(@Buffer, Count, @Result) <> S_OK
  then
    Result := 0;
end;
```

Используется все это счастье следующим образм. Пусть в клиенсткую часть передали интерфейс strm : IStream, тогда создаем свой экземпляр класса TExternStream на базе этого strm и используем как любой другой объект наследник TStream. Напрмер вызываем Image1.Picture.Bitmap.LoadFromStream(aOut);

```
var
  aOut : TExternStream;
try
  aOut := TExternStream.Create(strm);
  Image1.Picture.Bitmap.LoadFromStream(aOut);
....
  aOut.Free; //Тут уничтожиться поток созданный в Dll, так как указатель на aStream занулиться.
```

Урок 67 - Работа с памятью в системе Windows32 (часть 1/3)

Исследуем проблему накопления потоковых данных в специальных потоковых хранилищах. Работа с памятью является одной из важнейших функций любой программы. Выделение участков памяти для структур программы должно быть эффективным, поэтому программист

должен хорошо разбираться в особенностях этого процесса.
В уроке приводятся только основные принципы работы с памятью в системе Windows 32. Для подробного изучения всех тонкостей этого сложного процесса читатель может обратиться к специальной литературе. Особенно хочется отметить книгу: Дж.Рихтер, "Windows для профессионалов".

Как известно, Windows 32 - тридцатидвуразрядная операционная система (число 32 как раз это и обозначает). Прежде всего, из этого следует, что запущенная программа может адресовать линейное адресное пространство размером 2^32 байт = 4 ГБ, при этом адресация производится при помощи тридцатидвуразрядных регистров-указателей. Каждый запущенный в системе процесс обладает своим собственным адресным пространством, каждое из которых не пересекается с адресными пространствами других процессов. Распределение системных областей в адресном пространстве систем Windows 95/98 и Windows NT различно.
Программа может адресовать любую ячейку памяти в диапазоне адресов своего адресного пространства. Однако, это не значит, что программа может записать или считать данные из этой ячейки. Адресное пространство в Windows - виртуально, оно не связано напрямую с физическим пространством оперативной памяти вашего компьютера.
Механизм выделения памяти в Windows состоит из двух фаз. Первая фаза выделения памяти состоит в резервировании (захвате) участка необходимого размера в адресном пространстве процесса. При этом не выделяется ни байта реальной памяти (не считая системных структур ядра). Вы можете спокойно зарезервировать участок адресного пространства размером 100 мегабайт, и это ничего не будет стоить системе. Вы можете указать какие адреса вы хотели бы занять, а можете предоставить выбор участка необходимого размера самой системе. Стоит отметить, что адресное пространство резервируется с гранулярностью 64 кБ. Это значит, что несмотря на указанные вами адреса, базовый адрес реально зарезервированного участка памяти будет кратен 64 кБ. При резервировании участка в адресном пространстве можно указать желаемый аттрибут, который регулирует доступ к этой памяти: запись данных, чтение данных, выполнение кода или комбинацию этих признаков. Нарушение правил доступа к памяти приводит к генерации системой исключения.
Вторая фаза выделения памяти в Windows - это выделение реальной, физической памяти в зарезервированный участок виртуального адресного пространства. На этом этапе системой выделяется реальная память, со всеми вытекающими из этого последствиями. Выделение реальной памяти также гранулярно. Минимальный блок реальной памяти, которым оперирует система, и который можно выделить, называеся страницей памяти. Размер страницы зависит от типа операционной системы и составляет для Windows 95/98 - 4 кБ, а для Windows NT - 8 кБ. Гранулярность при резервировании участка памяти и при выделении реальной памяти призвана облегчить нагрузку на ядро системы.
Выделение реальной памяти происходит постранично, при этом существует возможность выделения произвольного количества страниц в произвольные (кратные размеру страницы) адреса заранее зарезервированного участка адресного пространства процесса. Каждой странице может быть назначен свой собственный атрибут доступа. Желательно указывать тот же самый атрибут, что имеет зарезервированный участок адресного пространства в котором происходит выделение страницы реальной памяти.
Важным моментом в механизме выделения памяти является механизм динамической выгрузки и загрузки страниц памяти. В самом деле, современный компьютер имеет оперативную память объемом 16-256 МБ, а для совместной работы нескольких программ необходимо гораздо больше. Windows, как и большинство современных операционных систем, выгружает страницы памяти, к которым давно не было обращений, на жесткий диск в так называемый своп-файл. При этом размер реальной памяти, доступной для программ, становиться равным суммарному объему оперативной памяти и своп-файла. По возможности, система старается держать все страницы в оперативной памяти, однако когда суммарный размер выделенных всеми процессами страниц превышает ее размер, система выгружает страницы с давним доступом на диск, а на их месте выделяет новые страницы. Если же выгруженная на диск страница затребуется владеющим ею процессом, система освободит для нее место в оперативной памяти путем выгрузки редко используемой страницы, загрузит затребованную страницу на ее место и вернет управление процессу.

Весь механизм динамической загрузки и выгрузки страниц абсолюдно прозрачен для процессов, а реализация этого механизма полностью обеспечивается операционной системой. Процесс может абсолюдно ничего не знать о сущесвовании такого механизма - он обращается к своим ячейкам памяти, а система автоматически выгружает редко используемые страницы и загружает необходимые страницы; процесс может только регулировать некоторые нюансы работы этого механизма.
На основании вышеперечисленного можно сделать следующие выводы:
каждый процесс со всеми своими потоками имеет отдельное и независимое линейное адресное пространство размером 4 Гб; выделение памяти состоит из двух фаз: резервирования адресного пространства и выделение в нем реальной памяти; при резервировании участка адресного пространства существует гранулярность размером 64 кБ; выделение реальной памяти производится постранично, размер страницы записит от типа операционной системы; каждой странице моежт быть назначен свой собственный атрибут доступа, нарушение которого приводит к генерации исключения системой; операционная система динамически выгружает редко используемый страницы памяти на из оперативной памяти на жесткий диск, причем этот механизм прозрачен для всех процессов.

Алгоритмы современных программ используют механизмы выделения и освобождения памяти очень интенсивно. Строки, динамические массивы, объекты, структуры, буфера - выделение и освобождение этих элементов происходит очень часто, при этом оказывается, что все эти элементы имеют небольшой размер.
Выделение у системы большого количества объектов небольшого размера оказывается неэффективным по следующим причинам.
частое обращение на выделение памяти снижает производительность, так как резервирование адресного пространства и выделение реальной памяти происходит на уровне ядра операционной системы;
из-за страничной организации памяти (гранулярности) выделение памяти происходит с большими издержками; запрос на выделение 100-байтного участка приводит к выделению одной страницы памяти с размером 4 или 8 кБ.
Решение этой проблемы организуется следующим образом: у операционной системы выделяется достаточно большой участок памяти, а уже из него для прикладной программы "нарезаются" небольшие участки. Такая организация называется кучей, а механизм, который следит за выделением и освобождением участков памяти называется менеджером кучи. Куча позволяет решить как проблему потери производительности - менеджер кучи может функционировать на уровне прикладной программы, так и проблему гранулярности - менеджер запрашивает у системы один большой участок памяти.
Windows имеет свою собственную реализацию менеджера кучи, который позволяет приложениям создавать, уничтожать кучи, а также производить с ними операции выделения и освобождения памяти в куче. Кроме того, для каждого вновь создаваемого процесса Windows специально создает кучу по умолчанию, которая используется при работе API функций, а также может быть использована прикладной программой. Все создаваемые Windows кучи потоко-безопасны, то есть существует возможность обращения к одной и той же куче из разных потоков одновременно.
Инженеры компании Borland по видимому не доверяют инженерам компании Microsoft, поэтому каждая программа на Delphi имеет свою собственную реализацию менеджера кучи, которая определена в системных модулях. Такое решение аргументируется инженерами Borland тем, что стандартный менеджер кучи Windows не обеспечивает достаточно эффективную работу с памятью. Конечно, как и любая Windows-программа, программа на Delphi имеет стандартную кучу по умолчанию, которую создает для нее система, однако функции New, Release, GetMem, FreeMem и некоторые другие оперируют с собственной реализацией менеджера куч. Менеджер кучи Delphi резервирует блоки адресного пространства размером 1 Мб, а выделяет блоки реальной памяти размером 16 Кб. Также вы можете написать и установить свою реализацию менеджера куч, если не доверяете ни инженерам Borland, ни инженерам Microsoft - для этого имеются все необходимые функции.
Хотя куча совершенно не предназначена для выделения больших участков памяти, запрос выделения большого участка у кучи не приведет к ошибке. Куча просто перенаправит ваш

запрос операционной системе и вернет указатель на выделенный ею участок памяти.

Урок 68 - Работа с памятью в системе Windows32 (часть 2/3)

Опишу базовые функции работы с памятью, которые доступны для программиста на Delphi. Включены описания как API-фукнций, так и Delphi-функций.

Delphi-функции

New(), Dispose()

Фунции работают с менеджером кучи Delphi. Обеспечивают типизированное выделение и освобождение памяти. Используются для динамической работы со структурами.

GetMem(), FreeMem()

Фунции работают с менеджером кучи Delphi. Обеспечивают нетипизированное выделение и освобождение памяти. Используются для динамической работы с небольшими бинарными блоками памяти (буфера, блоки).

API-функции

HeapCreate(), HeapDestroy(), ...

Функции работы со стандартным менеджером кучи Windows. Используются для создания и уничтожения куч, выделения и освобождения большого количества нетипизированных блоков памяти малого размера. Функции позволяют работать со стандартной кучей по умолчанию, которую создает операционная система для каждого процесса.

LocalAlloc(), LocalFree(), ... , GlobalAlloc(), GlobalFree(), ...

Так как в Windows 32 нет разделения на глобальные и локальные кучи, эти две группы функций идентичны. Функции работают со стандартной кучей по умолчанию, которую создает операционная система для каждого процесса. Функции морально устарели и Microsoft не рекомендует их использовать без крайней необходимости. Однако эти функции могут пригодиться, например, при работе с буфером обмена.

VirtualAlloc(), VirtualFree(), ...

"Основополагающие" функции выделения памяти в Windows. Используются как для резервирования адресного пространства, так и для выделения страниц реальной памяти в заранее зарезервированный участок адресного пространства. Позволяют выполнить обе фазы за один вызов функции. Используются для резервирования и выделения больших участков памяти.

Очень часто во многих программах встает проблема накопления потока поступающих данных. Например, это может быть запись звука, запись сигналов с датчиков, накопление данных с модема, коммуникационного порта, прием данных по сети и так далее. Если объем накапливаемых данных небольшой и заранее точно известен, то такая задача решается элементарно - под буфер выделяется блок памяти и эта память постепенно заполняется. Если же размер требуемого буфера достаточно большой, то выделение его полностью в самом начале может быть неэффективным - запись потока может прерваться гораздо раньше, а если же размер его неизвестен заранее, например когда запись данных останавливается по какому-либо внешнему сигналу, то встает проблема о выборе размера выделяемого блока.

В таких случаях используют динамические хранилища. В Delphi такими хранилищами являются динамические массивы, объект TMemoryStream, динамическое перераспределение памяти. Все эти хранилища работают на одном и том же принципе: под хранение данных выделяется блок памяти и поступающие данные последовательно записываются в этот блок, когда этот блок заполняется полностью, он перераспределяется с некоторым запасом (размер его увеличивается, а старые данные остаются). После того как каждый новый блок заполняется полностью он снова перераспределяется по мере поступления новых данных.

Перераспределение памяти занимает много ресурсов само по себе, а так как оно выполняется еще и в куче, то можно считать его вдвойне неэффективным, особенно если размеры перераспределяемых блоков становятся очень большими. Динамические массивы и динамическое перераспределение памяти используют менеджер кучи Delphi, а объект TMemoryStream использует стандартный менеджер кучи Windows.

Кроме того постоянное перераспределение участков памяти с разными размерами приводит к сильной дефрагментации памяти компьютера, что приводит к замедлению работы компьютера, а в конечном счете и к блокировке его работы.

Для решения проблемы накопления данных автором были разработаны два объекта накопления данных, основанные на двух разных принципах и имеющих разные характеристики.

TLinearStorage

Если размер буфера большой, но максимально возможный размер известен (например если есть ограничение на объем записываемых данных), то можно поступить следующим образом. В адресном пространстве процесса резервируется блок памяти необходимого размера, напомним, что такая операция не занимает ресурсов у системы. Затем по мере необходимости в этом зарезервированном участке памяти, по мере необходимости, последовательно выделяются страницы реальной памяти. Достоинством такого метода является линейное расположение ячеек в памяти друг за другом, то есть к такому хранилищу можно обращаться как к обычному линейному массиву. Недостатком - необходимость указания максимального размера.

TSectionStorage

Если размер буфера заранее неизвестен (ограничен лишь размером доступной реальной памяти), то предыдущее решение не подходит. В этом случае можно предложить другое решение. По мере необходимости, у системы можно запрашивать участки памяти одинакового размера и записывать в них поступающие данные. При этом буфер не будет иметь линейного адресного пространства, а будет состоять из одноразмерных "лоскутов" памяти - для вычислительного алгоритма такая организация буфера пожет стать серьезной помехой. Достоинством же такого решения является отсутствие необходимости изначально указывать какой либо размер буфера.

Если предполагается интенсивное увеличение-уменьшение хранилища, причем желательно, чтобы приращения были небольшими, то можно запрашивать память не у системы, а у стандартного менеджера кучи Windows, который создается для каждого хранилища отдельно. При этом, менеджер кучи выполняет роль кэша страниц памяти, увеличивая производительность.

Дополнительно, каждое хранилище имеет функции записи и чтения в стандартные потоки Delphi с упаковкой. Упаковка производится по стандартным алгоритмам библиотеки ZLIB.

TBaseStorage - базовый класс

Оба хранилища, которые будут рассматриваться в дальнейшем, основаны на одном абстрактом базовом классе и имеют схожие свойства и методы.

Item[] - получение указателя на указанный элемент по его индексу.
ItemSize - запрос размера хранимого элемента.
Count - запрос и установка числа хранимых элементов.

Clear - очистка хранилища, установление его размера в нуль. AddItems, GetItems, SetItems - добавление, запрос и установка блока элементов. SaveStream, LoadStream - запись и загрузка хранилища в/из потока. Параметр Compression в этих процедурах означает следующее 0 - компрессия не производится, и хранилище записывается в линейном натуральном виде; 1 - наименьшая степень компрессии; 9 - наивысшая степень компрессии. Число между 1..9 - произвольная степень компрессии.

```
// TBaseStorage
// Базовый класс для хранилищ
type
  TBaseStorage = class(TObject)
  public
    property Item[Ind: Cardinal]: Pointer read GetItem; default;
    property ItemSize: Cardinal read FItemSize;
    property Count: Cardinal read FCount write SetCount;
  public
    procedure Clear; virtual; abstract;
    procedure AddItems(Items: Pointer; Count: Cardinal); virtual; abstract;
    procedure SetItems(Items: Pointer; Index, Count: Cardinal); virtual;
      abstract;
    procedure GetItems(Items: Pointer; Index, Count: Cardinal); virtual;
```

```
    abstract;
  procedure SaveStream(Stream: TStream; Compression: Integer); virtual;
    abstract;
  procedure LoadStream(Stream: TStream; Compression: Integer; Count:
    Cardinal);
    virtual; abstract;
end;
```

Линейное хранилище

Линейное хранилище имеет линейное адресное пространство буфера, однако нуждается в указаниии максимальной емкости, пусть даже и очень большой.

Capacity - запрос и установка максимальной емкости хранилища. При установке емкости хранилища, все ранее хранимые данные теряются.

Memory - запрос указателя на линейный участок памяти, в котором хранятся данные, может быть использован в вычислительных алгоритмах.

Create - конструктор, в котором необходимо указать размер хранимого элемента.

```
// TLinearStorage
// Линейное хранилище
type
  TLinearStorage = class(TBaseStorage)
  public
    property Capacity: Cardinal read FCapacity write SetCapacity;
    property Memory: Pointer read FMemory;
  public
    procedure Clear; override;
    procedure AddItems(Items: Pointer; Count: Cardinal); override;
    procedure SetItems(Items: Pointer; Index, Count: Cardinal); override;
    procedure GetItems(Items: Pointer; Index, Count: Cardinal); override;
    procedure SaveStream(Stream: TStream; Compression: Integer); override;
    procedure LoadStream(Stream: TStream; Compression: Integer; Count:
      Cardinal);
      override;
  public
    constructor Create(AItemSize: Cardinal);
    destructor Destroy; override;
  end;
```

Секционное хранилище

Секционное хранилище хранит данные в кусочно-линейном буфере состоящем из участков одинакового размера. Хранилище не требует указания максимальной емкости, но взамен не позволяет обращаться к элементам как к массиву данных.

Block - список указателей на блоки, из которых состоит хранилище.

BlockSize - размер блоков, измеряемый в числе хранимых элементов.

Create - конструктор, в котором необходимо указать размер хранимого элемента в байтах и размер блока хранения.

```
// TSectionStorage
// Секционное хранилище
type
  TSectionStorage = class(TBaseStorage)
  public
    property Blocks: TList read FBlocks;
    property BlockSize: Cardinal read FBlockSize;
  public
    procedure Clear; override;
    procedure AddItems(Items: Pointer; Count: Cardinal); override;
    procedure SetItems(Items: Pointer; Index, Count: Cardinal); override;
    procedure GetItems(Items: Pointer; Index, Count: Cardinal); override;
```

```
    procedure SaveStream(Stream: TStream; Compression: Integer); override;
    procedure LoadStream(Stream: TStream; Compression: Integer; Count:
      Cardinal);
      override;
  public
    constructor Create(AItemSize: Cardinal; ABlockSize: Cardinal);
    destructor Destroy; override;
  end;
```

Первый пример демонстрирует эффективность менеджера кучи Delphi перед стандатным менеджером кучи Windows. На компьютерах, которые были мне доступны, тест показывал более чем четырехкратное превосходство менеджера кучи Delphi над менеджером кучи Windows.

Следующий пример содержит исходные тексты библиотеки потоковых хранилищ и тест, сравнивающий два потоковых хранилища, а также объекты TMemoryStream и TFileStream. Тест содержит один параметр, который вы можете регулировать - число добавляемых объектов. Увеличивайте этот параметр вдвое при каждом запуске теста и наблюдайте за поведением всех четырех объектов, особенно объекта TMemoryStream. Пока массив данных помещается в оперативной памяти, результаты этого объекта будут прекрасными, однако после того как массив перестанет помещаться в ОЗУ, объект начинает резко сдавать свои позиции, а вскоре перестает работать совсем. Когда же он работает на пределе возможностей, он создает помехи при выделении памяти - именно из-за этого тест желательно перезапускать.

Вообще с объектом TMemoryStream связаны странные, необъяснимые истории. Как-то раз автор имел несчастье использовать этот объект в одной из своих программ для накопления потока данных с модема. Через некоторое время после запуска программа зависала сама и, кроме того, подвешивала Windows NT. Анализ с помощью диспетчера задач показал, что в процессе жизнедеятельности программы, она занимает все новые и новые участки памяти. Поиск ошибок ни к чему ни привел, однако в конце концов пришлось обратить внимание на странности в поведении объекта TMemoryStream. Пришлось создать свой поток THeapStream путем формальной замены функций семейства Global... на функции GetMem, FreeMem, ReallocMem - то есть заменой стандартного менеджера кучи Windows на менеджер кучи Delphi. После этого все странности при работе программы исчезли.

Скорее всего это было связано с очень сильной дефрагментацией памяти, так как заполнение объекта TMemoryStream данными приводит к постоянному перераспределению участков памяти с разными размерами. От такой дефрагментации помогает только перезагрузка компьютера.

Урок 70 - Создание своих компонентов (часть 1/3)

Дельфи имеет открытую архитектуру - это значит, что каждый программист волен усовершенствовать эту среду разработки, как он захочет. К стандартным наборам компонентов, которые поставляются вместе с Дельфи можно создать еще массу своих интересных компонентов, которые заметно упростят вам жизнь (это я вам гарантирую). А еще можно зайти на какой-нибудь крутой сайт о Дельфи и там скачать кучу крутых компонентов, и на их основе сделать какую-нибудь крутую прогу. Так же компоненты освобождают вас от написания "тысячи тонн словесной руды". Пример: вы создали компонент - кнопку, при щелчке на которую данные из Memo сохранятся во временный файл. Теперь как только вам понадобится эта функция вы просто ставите этот компонент на форму и наслаждаетесь результатом. И не надо будет каждый раз прописывать это, для ваших новых программ - просто воспользуйтесь компонентом.

Первым делом нужно ответить себе на вопрос: "Для чего мне этот компонент и что он будет делать?". Затем необходимо в общих чертах продумать его свойства, события, на которые он будет реагировать и те функции и процедуры, которыми компонент должен обладать. Затем очень важно выбрать "предка" компонента, то есть наследником какого класса он будет являться. Тут есть два пути. Либо в качестве наследника взять уже готовый компонент (то есть модифицировать уже существующий класс), либо создать новый класс. Для создания нового класса можно выделить 4 случая:

Создание Windows-элемента управления (TWinControl)
Создание графического элемента управления (TGraphicControl)
Создание нового класса или элемента управления (TCustomControl)
Создание невизуального компонента (не видимого) (TComponent)

Теперь попробую объяснить что же такое визуальные и невизуальные компоненты. Визуальные компоненты видны во время работы приложения, с ними напрямую может взаимодействовать пользователь, например кнопка Button - является визуальным компонентом. Невизуальные компоненты видны только во время разработки приложения (Design-Time), а во время работы приложения (Run-Time) их не видно, но они могут выполнять какую-нибудь работу. Наиболее часто используемый невизуальный компонент - это Timer.

Итак, что бы приступить от слов к делу, попробуем сделать какой-нибудь супер простой компонент (только в целях ознакомления с техникой создания компонентов), а потом будем его усложнять.

Создание пустого модуля компонента шаг я буду исходя из устройства Дельфи 7, в других версиях этот процесс не сильно отличается. Давайте попробуем создать кнопку, у которой будет доступна информация о количестве кликов по ней. Чтобы приступить к непосредственному написанию компонента, вам необходимо сделать следующее:

Закройте проекты, которые вы разрабатывали (формы и модули) В основном меню выберите Component -> New Component...
Перед вами откроется диалоговое окно с названием "New Component" В поле Ancestor Type (тип предка) выберите класс компонента, который вы хотите модифицировать. В нашем случае вам надо выбрать класс TButton В поле Class Name введите имя класса, который вы хотите получить. Имя обязательно должно начинаться с буквы "T". Мы напишем туда, например, TCountBtn В поле Palette Page укажите имя закладки на которой этот компонент появиться после установки. Введем туда MyComponents (теперь у вас в Делфьи будет своя закладка с компонентами!). Поле Unit File Name заполняется автоматически, в зависимости от выбранного имени компонента. Это путь куда будет сохранен ваш модуль. В поле Search Path ничего изменять не нужно. Теперь нажмите на кнопку Create Unit и получите следующее:

```
unit CountBtn;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
 TCountBtn = class(TButton)

  private
   { Private declarations }

  protected
   { Protected declarations }

  public
   { Public declarations }

  published
    { Published declarations }
 end;

procedure Register;
```

```
implementation

procedure Register;
begin
  RegisterComponents('MyComponents', [TCountBtn]);
end;

end.
```

Что же здесь написано? да собственно пока ничего интересного. Здесь объявлен новый класс TCountBtn и процедура регистрации вашего компонента в палитре компонентов.

Директива Private. Здесь вы будете писать все скрытые поля которые вам понадобятся для создания компонента. Так же в этой директиве описываются процедуры и функции, необходимые для работы своего компонента, эти процедуры и функции пользователю не доступны. Для нашего компонент мы напишем туда следующее (запись должна состоять из буквы "F" имени поля: тип этого поля):

```
FCount:integer;
```

Буква "F" должна присутсвовать обязательно. Здесь мы создали скрытое поле Count, в котором и будет храниться число кликов по кнопке. Директива Protected. Обычно я здесь пишу различные обработчики событий мыши и клавиатуры. Мы напишем здесь следующую строку:

```
procedure Click; override;
```

Это указывает на то, что мы будем обрабатывать щелчок мыши по компоненту. Слово "override" указывает на то, что мы перекроем стандартное событие OnClick для компонента предка.

В директиве Public описываются те процедуры и функции компонента, которые будут доступны пользователю. (Например, в процессе написания кода вы пишите имя компонента, ставите точку и перед вами список доступных функций, объявленных в диретиве Public). Для нашего компонента, чтобы показать принцип использования этой директивы создадим функцию - ShowCount, которая покажет сообщение, уведомляя пользователя сколько раз он уже нажал на кнопку. Для этого в директиве Public напишем такой код:

```
procedure ShowCount;
```

Осталась последняя директива Published. В ней также используется объявления доступных пользователю, свойств и методов компонента. Для того, чтобы наш компонент появился на форме необходимо описать метод создания компонента (конструктор), можно прописать и деструктор, но это не обязательно. Следует обратить внимание на то, что если вы хотите, чтобы какие-то свойства вашего компонента появились в Инспекторе Объектов (Object Inspector) вам необходимо описать эти свойства в директиве Published. Это делается так: property Имя_свойства (но помните здесь букву "F" уже не нужно писать), затем ставиться двоеточие ":" тип свойства, read процедура для чтения значения, write функция для записи значения;. Но похоже это все сильно запутано. Посмотрите, что нужно написать для нашего компонента и все поймете: constructor Create(aowner:Tcomponent);override; //Конструктор property Count:integer read FCount write FCount; //СвойствоCount Итак все объявления сделаны и мы можем приступить к написанию непосредственно всех объявленных процедур.

Урок 71 - Создание своих компонентов (часть 2/3)

Начнем с написания конструктора. Это делается примерно так:

```
 constructor TCountBtn.Create(aowner:Tcomponent);
 begin

  inherited create(Aowner);
 end;
```

Здесь в принципе понимать ничего не надо. Во всех своих компонентах я писал именно это (только класс компонента менял и все). Также сюда можно записывать любые действия, которые вы хотите сделать в самом начале работы компонента, то есть в момент установки компонента на форму. Например можно установить начальное значение нашего свойства Count.

Но мы этого делать не будем. Теперь мы напишем процедуру обработки щелчка мышкой по кнопке:

```
procedure Tcountbtn.Click;
begin

  inherited click;
  FCount:=FCount+1;

end;
```

"Inherited click" означает, что мы повторяем стандартные методы обработки щелчка мышью (зачем напрягаться и делать лишнюю работу:)). У нас осталась последняя процедура ShowCount. Она может выглядеть примерно так:

```
procedure TCountBtn.ShowCount;
begin

  Showmessage('По кнопке '+ caption+' вы сделали: '+inttostr(FCount)+' клик(а/ов)');
end;
```

Здесь выводится сообщение в котором показывается количество кликов по кнопке (к тому же выводится имя этой кнопки, ну это я добавил только с эстетической целью). И если вы все поняли и сделали правильно, то у вас должно получится следующее:

```
unit CountBtn;



interface



uses
 Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

 StdCtrls, ExtCtrls;



type
 TCountBtn = class(TButton)

  private
   { Private declarations }

  FCount: integer;
  protected

   { Protected declarations }
  procedure Click;override;

  public
   { Public declarations }

  procedure ShowCount;
```

```
  published

   { Published declarations }
  property Count:integer read FCount write FCount;

  constructor Create(aowner:Tcomponent); override;
 end;


procedure Register;


implementation


procedure Register;

begin
 RegisterComponents('Mihan Components', [TCountBtn]);

end;


constructor TCountBtn.Create(aowner:Tcomponent);
begin

 inherited create(Aowner);
end;


procedure Tcountbtn.Click;

begin
 inherited click;

 FCount:=FCount+1;
end;


procedure TCountBtn.ShowCount;

begin
 Showmessage('По кнопке '+ caption+' вы сделали: '+inttostr(FCount)+' клик(а/ов)');

end;


end.
```

Скорее сохраняйтесь, дабы не потерять случайным образом байты набранного кода:)).

Если вы сумели написать и понять, все то что здесь предложено, то установка компонента не должна вызвать у вас никаких проблем. Все здесь делается очень просто. В главном меню выберите пункт Component -> Install Component. перед вами открылось диалоговое окно Install Component. В нем вы увидите две закладки: Into exsisting Package и Into new Package. Вам предоставляется выбор установить ваш компонент в уже существующий пакет или в новый пакет соответственно. Мы выберем в уже существующий пакет.

В поле Unit File Name напишите имя вашего сохранненого модуля (естественно необходимо еще и указать путь к нему), а лучше воспользуйтесь кнопкой Browse и выберите ваш файл в открывшемся окне.
В Search Path ничего изменять не нужно, Делфьи сама за вас все туда добавит.
В поле Package File Name выберите имя пакета, в который будет установлен ваш компонент. Мы согласимся с предложенным по умолчанию пакетом.
Теперь нажимаем кнопочку Ok. И тут появиться предупреждение Package dclusr30.dpk will be rebuilt. Continue? Дельфи спрашивает: "Пакет такой-то будет изменен. Продолжить?". Конечно же надо ответить "Да". И если вы все сделали правильно, то появиться сообщение, что ваш компонент установлен. Что ж можно кричать Ура! Это ваш первый компонент.

Создание свойств своего типа Теперь мы попробуем создать свойство нестандартного типа. Рассмотрим это на примере метки - TLabel. У этого компонента есть такое свойство: Alignment. Оно может принимать следующие значения: taLeftJustify, taCenter, taRightJustify. Приступаем к созданию свойства. Ничего интересного мне придумать не удалось, но тем не менее я вам покажу это на примере того свойства, которое я придумал. Оно очень простое и поможет вам разобраться. Свойство будет называться ShowType (тип TShowTp), в нашем компоненте оно будет отвечать за отображение свойства Count. Если пользователь установит свойство ShowType в Normal, то кнопка будет работать, как и работала. А если пользователь присвоит этому свойтсву значение CountToCaption, то количество кликов, будет отображаться на самой кнопке.
Для начале нам необходимо объявить новый тип. Описание типа нужно добавить после слова Type. Вот так это выглядело вначале:

```
 type
 TCountBtn = class(TButton)
```

Вот так это должно выглядеть:

```
 type


  TShowTp = (Normal, CountToCaption);
  TCountBtn = class(TButton)
```

Здесь мы объявили новый тип TShowTp, который может принимать только два значения. Все значения, которые вы хотите добавить перечисляются через запятую. Теперь нам понадобиться создать поле этого типа. Это мы уже умеем и делать и поэтому не должно вызвать никаких сложностей. В директиву Private напишите:

```
 FShowType:TShowTp;
```

Мы создали поле ShowType, типа TShowTp. Конечно же необходимо добавить это свойство в инспектор объектов:

```
 property ShowType: TshowTp read FshowType write FShowType;
```

Ну и наконец, чтобы наш компонент реагировал на изменение этого свойства пользователем надо слегка изменить обработчик события OnClick. После небольшой модификации он может иметь примерно такой вид:

```
 procedure Tcountbtn.Click;
 begin
```

```
 inherited click;
 FCount:=Fcount+1;

 if ShowType = Normal then
   Caption:=Caption;

 if ShowType =
CountToCaption then
   Caption:='Count= '+inttostr(count);

end;
```

Урок 72 - Создание своих компонентов (часть 3/3)

Объясню что произошло. Вначале мы увеличиваем счетчик на единицу. Затем проверяем какое значение имеет свойство ShowType. Если Normal, то ничего не делаем, а если CountToCaption, то в надпись на кнопке выводим количество кликов. Не так уж и сложно как это могло показаться с первого раза. Имплантируем таймер в компонент Очень часто бывает, что вам необходимо вставить в компонент, какой-нибудь другой компонент, например, таймер. Как обычно будем рассматривать этот процесс на конкретном примере. Сделаем так, что через каждые 10 секунд значение счетчика кликов будет удваиваться. Для этого мы встроим таймер в нашу кнопку. Нам понадобиться сделать несколько несложных шагов.

После раздела uses, где описаны добавленные в программу модули, объявите переменную типа TTimer. Назовем ее Timer. Приведу небольшой участок кода:

```
unit CountBtn;



interface



uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

  StdCtrls, ExtCtrls;



var Timer: TTimer;
type
```

Дальше в директиву Protected необходимо добавить обработчик события OnTimer для нашего таймера. Это делается так:

```
procedure OnTimer(Sender: TObject);
```

Поскольку наш таймер это не переменная, а компонент, его тоже надо создать, для этого в конструктор нашей кнопки напишем:

```
constructor TCountBtn.Create(aowner:Tcomponent);
begin

 inherited create(Aowner);
 Timer:=TTimer.Create(self);

 Timer.Enabled:=true;
```

```
  Timer.OnTimer:=OnTimer;

  Timer.Interval:=10000;
 end;
```
Здесь создается экземпляр нашего таймера и его свойству Iterval (измеряется в миллисекундах) присваивается значение 10000 (то есть 10 секунд если по простому). Собственно осталось написать саму процедуру OnTimer. Я сделал это так:
```
 procedure TCountBtn.OnTimer(Sender: TObject);
 begin

  FCount:=FCount*2;
 end;
```
Вот примерно то, что у вас должно получиться в конце:
```
 unit CountBtn;


 interface


 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

  StdCtrls, ExtCtrls;


 var Timer: TTimer;
 type

  TShowTp = (Normal, CountToCaption);
  TCountBtn = class(TButton)


  private

   { Private declarations }


  FCount:integer;
  FShowType:TShowTp;


  protected

   { Protected declarations }
  procedure OnTimer(Sender: TObject);

  procedure Click;override;
```

```
 public
  { Public declarations }

 procedure ShowCount;


 published
  { Published declarations }

 property Count:integer read FCount write FCount;
 constructor Create(aowner:Tcomponent);override;

 property ShowType: TshowTp read FshowType write FShowType;
end;


procedure Register;


implementation


procedure Register;

begin
 RegisterComponents('Mihan Components', [TCountBtn]);

end;


constructor TCountBtn.Create(aowner:Tcomponent);
begin

  inherited create(Aowner);
  Timer:=TTimer.Create(self);

  Timer.Enabled:=false;
  Timer.OnTimer:=OnTimer;

  Timer.Interval:=1000;
end;


procedure Tcountbtn.Click;

begin
  inherited click;
```

```
  FCount:=Fcount+1;
  Timer.Enabled:=true;

  if ShowType = Normal then
  Caption:=Caption;

  if ShowType =
CountToCaption then
    Caption:='Count= '+inttostr(count);

end;


procedure TCountBtn.ShowCount;
begin

  Showmessage('По кнопке '+ caption+' вы сделали: '+inttostr(FCount)+' клик(а/ов)');
end;


procedure TCountBtn.OnTimer(Sender: TObject);

begin
  FCount:=FCount*2;

end;



end.
```

Если у вас что-то не сработало, то в начале проверьте все ли у вас написано правильно. Затем проверьте может у вас не хватает какого-нибудь модуля в разделе Uses.

Переустановка компонента

Очень часто бывает необходимо переустановить ваш компонент. Если вы попробуете сделать это путем выбора Component->Install Component, то Дельфи вас честно предупредит о том, что пакет уже содержит модуль с таким именем. Перед вами открывается окно с содержимым пакета. В нем вы должны найти имя вашего компонента и удалить его (либо нажать кнопочку Remove). Теперь в пакете уже нет вашего компонента. Затем проделайте стандартную процедуру по установке компонента.

Урок 73 - Оператор Case

В 3 уроке я рассказал об условном операторе IF. В этому уроке я расскажу о другом аналогичном по функционалу, но более эффективном и удобном для большого количества вложенных условий операторе CASE.

Оператор CASE более изящен, более эффективен, и его проще обслуживать чем множество вложений IF.

Сначала разберем, как выглядит конструкция CASE:

```
case a of
```

```
    1 : ShowMessage('a=1');

    2 : ShowMessage('a=2');
    3 : ShowMessage('a=3');

    4 : ShowMessage('a=4');
    else ShowMessage('no');

 end;
```

В данном случае, мы можем назначать каждому значению свой результат. Конструкция ELSE в данном случае опциональна, т.е. не обязательна.

Оператор CASE позволяет удобно и эффективно работать с большим количеством условных вложений, в отличии от оператора IF. Чтобы понять преимущества новой конструкции, рассмотрим как будет выглядеть код, представленный выше, но теперь вместо CASE мы будем использовать оператор IF:

```
 If a=1 then ShowMessage('a=1') else
 If a=2 then ShowMessage('a=2') else

 If a=3 then ShowMessage('a=3') else
 If a=4 then ShowMessage('a=4') else

 ShowMessage('no');
```

Согласитесь, такой код гораздо неудобнее, чем с использованием CASE.

Стоит отметить, что IF и CASE хоть и являются условными операторами с одинаковыми возможностями, используются они для разных целей исключительно по удобству. В каких-то случаях гораздо удобнее использовать IF, например когда условия имеют сложную конструкцию, в других случаях удобно использовать CASE, например когда нам нужно создать условие, имеющее несколько одинаковых по структуре условных вложений.

Урок 74 - Оператор GOTO

Инструкции if и case используются для перехода к последовательности инструкций программы в зависимости от некоторого условия. Поэтому ихиногда называют инструкциями условного перехода. Помимо этих инструкций управления ходом выполнения программы существует еще одна — инструкция безусловного перехода goto. В общем виде инструкция goto записывается следующим образом:

```
 goto Метка;
```

где метка — это идентификатор, находящийся перед инструкцией, которая должна быть выполнена после инструкции goto. Метка, используемая в инструкции goto, должна быть объявлена в разделе меток, который начинается словом label и располагается перед разделом объявления переменных. В программе метка ставится перед инструкцией, к которой должен быть выполнен переход в результате выполнения инструкции goto. Сразу после метки ставится двоеточие.

В листинге приведен вариант процедуры проверки числа, в которой ин струкция goto используется для завершения процедуры в том случае, если пользователь введет неверные данные.

Листинг:

```
 procedure TForml.ButtonlClickfSender: TObject);
 label // раздел объявления меток
```

```
bye;
var

n: integer; // проверяемое число
d: integer; // делитель

r: integer; // остаток от деления п на d
begin

n:=StrTo!nt(Edit1.text);
if n <= 0 then

begin
MessageDlg('Число должно быть больше нуля.',

mtError,(mbOkl,Q);
Editl.text:= '';

goto bye;
end;

// введено положительное число
d: = 2; // сначала будем делить на два

repeat
r:= n mod d;

if r o 0 // n не разделилось нацело на d
then d:= d + 1;

until r = 0;
Iabel2.caption:=Editl.text;

if d = n
then Iabel2.caption:=label2.caption+ ' — простое число.'

else Iabel2.caption:=labe!2.caption+ ' — обычное число.';
bye:

end;
```

В литературе по программированию можно встретить суждения о недопустимости применения инструкции goto, поскольку она приводит к запутан ности программ. Однако с категоричностью таких утверждений согласиться нельзя. В некоторых случаях применение инструкции goto вполне оправдано. Приведенный пример, где инструкция goto используется для аварийного завершения процедуры, относится именно к таким случаям.

Урок 75 - Рекурсия

Рекурсия - это такой способ организации вычислительного процесса, при котором

подпрограмма в ходе выполнения составляющих ее операторов обращается сама к себе.

Рассмотрим классический пример - вычисление факториала. Программа получает от компонента edinput целое число n и выводит в компонент lboutput значение N!, которое вычисляется с помощью рекурсивной функции Factorial.

При выполнении правильно организованной рекурсивной подпрограммы осуществляется многократный переход от некоторого текущего уровня организации алгоритма к нижнему уровню последовательно до тех пор, пока, наконец, не будет получено тривиальное решение поставленной задачи. В нашем случае решение при n = 0 тривиально и используется для остановки рекурсии.

```
procedure TfmExample.bbRunClick(Sender: TObject);
  function Factorial(N: Word): Extended;

  begin
    if N = 0 then

      Result: = 1 else
      Result := N * Factorial(N - 1)

  end;
var

  N: Integer;
begin

  try
    N := StrToInt(Trim(edinput.Text));

  except
    Exit; end;

  IbOutput.Caption := FloatToStr(Factorial(N))
end;
```

Рекурсивная форма организации алгоритма обычно выглядит изящнее итерационной и дает более компактный текст программы, но при выполнении, как правило, медленнее и может вызвать переполнение стека (при каждом входе в подпрограмму ее локальные переменные размещаются в организованной особым образом области памяти, называемой программным стеком).

Рекурсивный вызов может быть косвенным. В этом случае подпрограмма обращается к себе опосредованно, путем вызова другой подпрограммы, в которой содержится обращение к первой, например:

```
procedure A(i: Byte);
begin

  B(i);
end;
```

```
procedure B(j: Byte);

begi
n
  a(j);

end;
```

Если строго следовать правилу, согласно которому каждый идентификатор перед употреблением должен быть описан, то такую программную конструкцию использовать нельзя. Чтобы такого рода вызовы стали возможны, вводится опережающее описание:

```
procedure B(j: Byte); forward;



procedure A(i: Byte);
begin

  B(i);
end;



procedure B; begin

  A(j);
end;
```

Как видим, опережающее описание заключается в том, что объявляется лишь заголовок процедуры в, а ее тело заменяется стандартной директивой Forward. Теперь в процедуре а можно использовать обращение к процедуре в - ведь она уже описана, точнее, известны ее формальные параметры, и компилятор может правильным образом организовать ее вызов. Обратите внимание: тело процедуры в начинается заголовком, в котором уже не указываются описанные ранее формальные параметры.

Урок 76 - Множества

Введение

От рассмотрения простых типов данных переходим к более сложным наборам данных. Следует вспомнить, что простые типы данных позволяют хранить одно-единственное значение в одной переменной. Структуры данных позволяют хранить сразу несколько значений, причём некоторые такие структуры теоретически вообще не имеют ограничения на количество значений - этот предел определяется лишь объёмом доступной памяти. Сегодня мы поговорим о множествах.

Математическое понятие множества

Вспомним, как определяется множество в математике. Множество - это конечный или бесконечный набор определённых объектов, мыслимый как единое целое. Множество характеризуется своими элементами, а элементы имеют лишь одно свойство - принадлежность к данному множеству. Таким образом, мы можем только сказать, принадлежит элемент данному множеству или не приналдежит. Порядок расположения элементов в множестве никакой роли не играет.

Множества в Delphi

Понятие множества в языке программирования несколько отличается от математического определения этого понятия, но смысл сохраняется. Основное отличие в том, что в программировании множество может содержать только конечное число элементов, т.е. не может состоять из бесконечного числа объектов. В математике же последнее допустимо. Например, мы можем определить множество натуральных чисел, которое бесконечно: N = {1, 2, 3, ...}

Следует понимать, что множество не предназначено для хранения каких-либо значений (чисел, символов и т.д.) - оно лишь может дать нам ответ на вопрос: присутствует конкретный элемент в множестве или его там нет.

Перейдём ближе к делу. Множество может быть построено на основе перечислимого типа данных (кто забыл - открываем предыдущий урок). Например, на основе символьного типа Char. По-английски множество называется set (набор) и именно этим словом описывается в Delphi:

```
var A: set of Char;
```

В данном примере мы объявили множество A на основе символьного типа Char.

Запомните: множество не может состоять более чем из 255 элементов!

Например, следующее описание:

```
var N: set of Integer;
```

приведёт к ошибке "Set base type out of range".

Задание множеств

В математике множество обычно записывается в виде фигурных скобок, в которых через запятую перечисляются элементы, принадлежащие множеству. В Delphi вместо фигурных скобок используются квадратные.

Чтобы задать множество, мы можем воспользоваться операцией присваивания, где слева стоит переменная-множество, а справа - нужный нам набор. Например, в описанное выше множество A мы хотим поместить элементы-символы A, B, C, D. Тогда это запишется так:

```
A:=['A', 'B','C','D'];
```

Теперь множество A содержит 4 элемента.

Если вспомнить, что тип данных Char упорядочен, то данную запись можно сократить следующим образом:

```
A:=['A'..'D'];
```

Мы просто указали диапазон значений, который должен находиться во множестве. Результаты одинаковый, но вторая запись короче и красивее. Допустимы, конечно же, комбинации диапазонов и отдельных значений:

```
A:=['A','B','K'..'N','R','X'..'Z'];
```

Помните, что множество - это виртуальный набор элементов: множество нельзя ввести с клавиатуры и точно так же нельзя вывести на экран. Поэтому добавление элементов во множество делается только программным путём. Безусловно, вы каким-либо образом можете связать элементы интерфейса программы и операцию добавления элементов во множество, но напрямую ввести множество нельзя. Аналогично, вы можете показать множество на экране с помощью каких-либо элементов (например, флажков TCheckBox), но само множество "в чистом виде" вывести нельзя.

Операции над множествами

В программировании, как и в математике, над множествами допустимы некоторые операции. Рассмотрим их.

Находится ли элемент во множестве?

Самая простая операция, для понятия смысла которой даже не нужно задумываться. Чтобы проверить, входит ли элемент во множество, следует использовать специальную конструкцию с оператором in. Слева от него указывается элемент, справа - множество. Результатом, как несложно догадаться, является логичное значение - истина или ложь. True - элемент принадлежит множеству, False - не принадлежит:

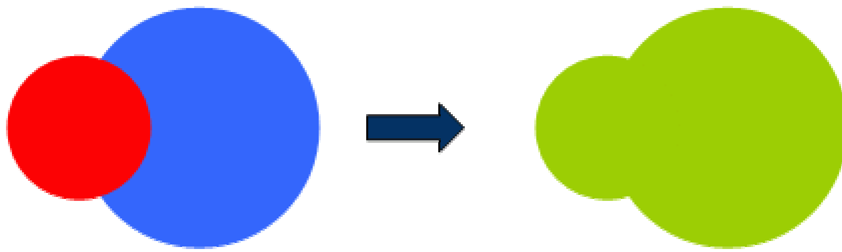
```
var A: set of Char;
{...}


A:=['A'..'E','X'];
if 'D' in A then ShowMessage('Элемент B находится во множестве A.');
```

Несложно проверить, что сообщение в данном случае появится на экране.

Объединение множеств

Если есть два множества, определённые на одном и том же типе данных, то их можно объединить и получить таким образом новое множество.

Если изобразить множества в виде кругов, причём круги пересекаются в том случае, если у множеств есть одинаковые элементы, то объединение можно изобразить следующим образом:



В словесном описании операция объединения - результирующее множество содержит все те элементы, которые есть хотя бы в одном из двух исходных множеств.

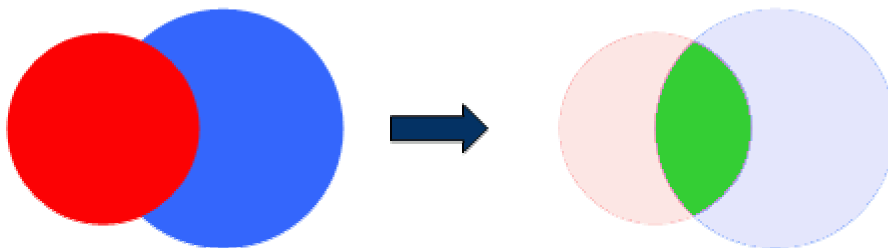
Объединение записывается знаком плюс "+". Пример:

```
var A,B,C: set of Char;
{...}

A:=['A','B','C'];
B:=['X','Y','Z'];

C:=A+B;
//C = ['A','B','C','X','Y','Z']
```

Включение одного элемента во множество делается точно таким же образом, просто в этом случае включаемое множество содержит всего один элемент.

Пересечение множеств

Операция пересечения формирует множество только из тех элементов, которые одновременное присутствуют как в первом, так и во втором исходном множестве. Операция пересечения графически:

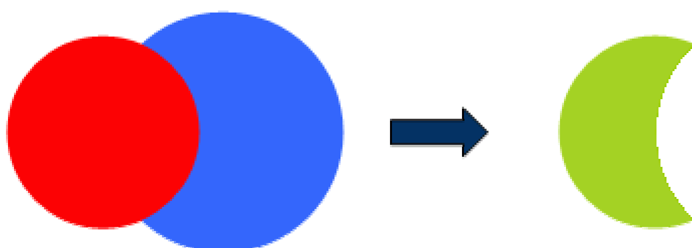


Пересечение обозначается звёздочкой "*". Пример:

```
var X,Y,Z: set of Byte;
{...}

X:=[1,2,3,4,5];
Y:=[4,5,6,7,8];

Z:=X*Y;
//Z = [4,5]
```

Разность множеств

Операция вычитания удаляет из первого множества те элементы, которые есть во втором множестве:



Пример:

```
var X,Y,Z: set of Char;
{...}

X:=['A'..'D'];
Y:=['D'..'F'];

Z:=X-Y;
//Z = ['A'..'C']
```

Следует обратить внимание, что порядок множеств в данном случае важен, т.е. X-Y и Y-X - это разные множества.

Применение множеств

Множества находят широкое применение. С помощью множеств удобно задавать набор опций, каждая из которых либо включена, либо выключена. К примеру, поместите на форму кнопку (TButton), перейдите в инспектор объектов, разверните свойство Font (шрифт) и найдите свойство Style. Вот это свойство как раз и реализовано множеством. Во множестве 4 элемента: fsBold, fsItalic, fsUnderline и fsStrikeOut, каждый из которых отвечает за стиль шрифта. Принадлежность элементов ко множеству задаётся указанием значения True или False для каждого из этих пунктов. В строке "Style" находится описание данного множества. Попробуйте изменять стиль и посмотреть, как меняется описание множества Style.

А теперь давайте сделаем простенький интерфейс для доступа к этому свойству. Пусть будет меняться стиль шрифта у этой кнопки (Button1). Поместим на форму 4 TCheckBox - для доступа ко всем значениям и дадим им соответствующие имена. Изменение стиля будем делать при нажатии на саму эту кнопку. Пример реализации:

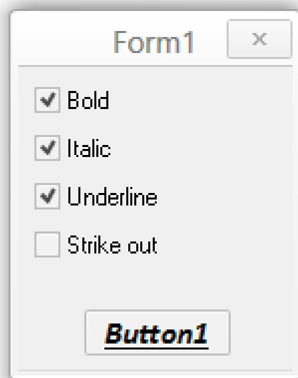
```
procedure TForm1.Button1Click(Sender: TObject);
begin

  Button1.Font.Style:=[]; //Сделали множество пустым
  //Теперь смотрим состояния флажков и добавляем нужные стили

  if CheckBox1.Checked then Button1.Font.Style:=Button1.Font.Style+[fsBold]
;
  if CheckBox2.Checked then Button1.Font.Style:=Button1.Font.Style+[fsItalic];

  if CheckBox3.Checked then Button1.Font.Style:=Button1.Font.Style+[fsUnderline];
  if CheckBox4.Checked then Button1.Font.Style:=Button1.Font.Style+[fsStrikeOut];

end;
```

Чтобы не повторять везде одно и то же "Button1.Font.", эту часть кода можно, что называется, вынести за скобку при помощи специального оператора with. Ранее речь о нём не шла, однако этот оператор очень удобен. Смысл его прост: то, что вынесено вперёд, автоматически применяется ко всему, что находится внутри данного блока. В нашем случае будет так:

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

  with Button1.Font do
  begin

    Style:=[]; //Сделали множество пустым
    //Теперь смотрим состояния флажков и добавляем нужные стили
```

```
    if CheckBox1.Checked then Style:=Style+[fsBold]
;
    if CheckBox2.Checked then Style:=Style+[fsItalic];

    if CheckBox3.Checked then Style:=Style+[fsUnderline];
    if CheckBox4.Checked then Style:=Style+[fsStrikeOut];

  end
 end;
```

Согласитесь, так гораздо удобнее. Используйте оператор with как можно чаще - с его помощью и код по объёму становится меньше и скорость работы увеличивается.



У большинства компонент среди свойств можно найти множества. Например, у диалога открытия файла TOpenDialog (вкладка Dialogs) множеством представлено свойство Options, которое содержит приличное число элементов

Урок 77 - Создание компонентов для Delphi

Для чего нужны компоненты

Дельфи имеет открытую архитектуру - это значит, что каждый программист волен усовершенствовать эту среду разработки, как он захочет. К стандартным наборам компонентов, которые поставляются вместе с Дельфи можно создать еще массу своих интересных компонентов, которые заметно упростят вам жизнь (это я вам гарантирую). А еще можно зайти на какой-нибудь крутой сайт о Дельфи и там скачать кучу крутых компонентов, и на их основе сделать какую-нибудь крутую прогу. Так же компоненты освобождают вас от написания "тысячи тонн словесной руды". Пример: вы создали компонент - кнопку, при щелчке на которую данные из Memo сохранятся во временный файл. Теперь как только вам понадобится эта функция вы просто ставите этот компонент на форму и наслаждаетесь результатом. И не надо будет каждый раз прописывать это, для ваших новых программ - просто воспользуйтесь компонентом.

Шаг 1. Придумывание идеи

Первым шагом нужно ответить себе на вопрос: "Для чего мне этот компонент и что он будет делать?". Затем необходимо в общих чертах продумать его свойства, события, на которые он будет реагировать и те функции и процедуры, которыми компонент должен обладать. Затем очень важно выбрать "предка" компонента, то есть наследником какого класса он будет являться. Тут есть два пути. Либо в качестве наследника взять уже готовый компонент (то есть модифицировать уже существующий класс), либо создать новый класс.

Для создания нового класса можно выделить 4 случая:
Создание Windows-элемента управления (TWinControl)
Создание графического элемента управления (TGraphicControl)
Создание нового класса или элемента управления (TCustomControl)
Создание невизуального компонента (не видимого) (TComponent)

Теперь попробую объяснить что же такое визуальные и невизуальные компоненты. Визуальные компоненты видны во время работы приложения, с ними напрямую может взаимодействовать пользователь, например кнопка Button - является визуальным компонентом.

Невизуальные компоненты видны только во время разработки приложения (Design-Time), а во время работы приложения (Run-Time) их не видно, но они могут выполнять какую-нибудь работу. Наиболее часто используемый невизуальный компонент - это Timer.

Итак, что бы приступить от слов к делу, попробуем сделать какой-нибудь супер простой компонент (только в целях ознакомления с техникой создания компонентов), а потом будем его усложнять.

Шаг 2. Создание пустого модуля компонента

Рассматривать этот шаг я буду исходя из устройства Дельфи 3, в других версиях этот процесс не сильно отличается. Давайте попробуем создать кнопку, у которой будет доступна информация о количестве кликов по ней.

Чтобы приступить к непосредственному написанию компонента, вам необходимо сделать следующее:

Закройте проекты, которые вы разрабатывали (формы и модули)

В основном меню выберите Component -> New Component...

Перед вами откроется диалоговое окно с названием "New Component"

В поле Ancestor Type (тип предка) выберите класс компонента, который вы хотите модифицировать. В нашем случае вам надо выбрать класс TButton

В поле Class Name введите имя класса, который вы хотите получить. Имя обязательно должно начинаться с буквы "T". Мы напишем туда, например, TCountBtn

В поле Palette Page укажите имя закладки на которой этот компонент появиться после установки. Введем туда MyComponents (теперь у вас в Делфьи будет своя закладка с компонентами!).

Поле Unit File Name заполняется автоматически, в зависимости от выбранного имени компонента. Это путь куда будет сохранен ваш модуль.

В поле Search Path ничего изменять не нужно.

Теперь нажмите на кнопку Create Unit и получите следующее:

```
unit CountBtn;
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

  StdCtrls;



type
 TCountBtn = class(TButton)



  private

   { Private declarations }



  protected
   { Protected declarations }



  public

   { Public declarations }
```

```
  published
   { Published declarations }

  end;


procedure Register;


implementation


procedure Register;
begin

  RegisterComponents('MyComponents', [TCountBtn]);
end;


end.
```

Шаг 3. Начинаем разбираться во всех директивах
Что же здесь написано? да собственно пока ничего интересного. Здесь объявлен новый класс TCountBtn и процедура регистрации вашего компонента в палитре компонентов.
Директива Private. Здесь вы будете писать все скрытые поля которые вам понадобятся для создания компонента. Так же в этой директиве описываются процедуры и функции, необходимые для работы своего компонента, эти процедуры и функции пользователю не доступны. Для нашего компонент мы напишем туда следующее (запись должна состоять из буквы "F" имени поля: тип этого поля):

```
FCount:integer;
```

Буква "F" должна присутсвовать обязательно. Здесь мы создали скрытое поле Count, в котором и будет храниться число кликов по кнопке.
Директива Protected. Обычно я здесь пишу различные обработчики событий мыши и клавиатуры. Мы напишем здесь следующую строку:

```
procedure Click; override;
```

Это указывает на то, что мы будем обрабатывать щелчок мыши по компоненту. Слово "override" указывает на то, что мы перекроем стандартное событие OnClick для компонента предка.
В директиве Public описываются те процедуры и функции компонента, которые будут доступны пользователю. (Например, в процессе написания кода вы пишите имя компонента, ставите точку и перед вами список доступных функций, объявленных в диретиве Public). Для нашего компонента, чтобы показать принцип использования этой директивы создадим функцию - ShowCount, которая покажет сообщение, уведомляя пользователя сколько раз он уже нажал на кнопку. Для этого в директиве Public напишем такой код:

```
procedure ShowCount;
```

Осталась последняя директива Published. В ней также используется объявления доступных пользователю, свойств и методов компонента. Для того, чтобы наш компонент появился на форме необходимо описать метод создания компонента (конструктор), можно прописать и деструктор, но это не обязательно. Следует обратить внимание на то, что если вы хотите, чтобы какие-то свойства вашего компонента появились в Инспекторе Объектов (Object Inspector) вам необходимо описать эти свойства в директиве Published. Это делается так: property

Имя_свойства (но помните здесь букву "F" уже не нужно писать), затем ставиться двоеточие ":" тип свойства, read процедура для чтения значения, write функция для записи значения;. Но похоже это все сильно запутано. Посмотрите, что нужно написать для нашего компонента и все поймете:

```
constructor Create(aowner:Tcomponent);override; //Конструктор
property Count:integer read FCount write FCount; //СвойствоCount
```

Итак все объявления сделаны и мы можем приступить к написанию непосредственно всех объявленных процедур.

Шаг 4. Пишем процедуры и функции.

Начнем с написания конструктора. Это делается примерно так:

```
constructor TCountBtn.Create(aowner:Tcomponent);
begin

  inherited create(Aowner);
end;
```

Здесь в принципе понимать ничего не надо. Во всех своих компонентах я писал именно это (только класс компонента менял и все). Также сюда можно записывать любые действия, которые вы хотите сделать в самом начале работы компонента, то есть в момент установки компонента на форму. Например можно установить начальное значение нашего свойства Count. Но мы этого делать не будем.

Теперь мы напишем процедуру обработки щелчка мышкой по кнопке:

```
procedure Tcountbtn.Click;
begin

  inherited click;
  FCount:=FCount+1;

end;
```

"Inherited click" означает, что мы повторяем стандартные методы обработки щелчка мышью (зачем напрягаться и делать лишнюю работу:)).

У нас осталась последняя процедура ShowCount. Она может выглядеть примерно так:

```
procedure TCountBtn.ShowCount;
begin

  Showmessage('По кнопке '+ caption+' вы сделали: '+inttostr(FCount)+' клик(а/ов)');
end;
```

Здесь выводится сообщение в котором показывается количество кликов по кнопке (к тому же выводится имя этой кнопки, ну это я добавил только с эстетической целью).

И если вы все поняли и сделали правильно, то у вас должно получится следующее:

```
unit CountBtn;



interface



uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

  StdCtrls, ExtCtrls;
```

```
type
 TCountBtn = class(TButton)

  private
   { Private declarations }

  FCount: integer;
  protected

   { Protected declarations }
  procedure Click;override;

  public
   { Public declarations }

  procedure ShowCount;
  published

   { Published declarations }
  property Count:integer read FCount write FCount;

  constructor Create(aowner:Tcomponent); override;
 end;


procedure Register;


implementation


procedure Register;

begin
 RegisterComponents('Mihan Components', [TCountBtn]);

end;


constructor TCountBtn.Create(aowner:Tcomponent);
begin

 inherited create(Aowner);
end;


procedure Tcountbtn.Click;
```

```
begin
 inherited click;

 FCount:=FCount+1;
end;


procedure TCountBtn.ShowCount;

begin
 Showmessage('По кнопке '+ caption+' вы сделали: '+inttostr(FCount)+' клик(а/ов)');

end;


end.
```

Скорее сохраняйтесь, дабы не потерять случайным образом байты набранного кода:)).

Шаг 5. Устанавливаем компонент

Если вы сумели написать и понять, все то что здесь предложено, то установка компонента не должна вызвать у вас никаких проблем. Все здесь делается очень просто. В главном меню выберите пункт Component -> Install Component. перед вами открылось диалоговое окно Install Component. В нем вы увидите две закладки: Into exsisting Package и Into new Package. Вам предоставляется выбор установить ваш компонент в уже существующий пакет или в новый пакет соответственно. Мы выберем в уже существующий пакет.

В поле Unit File Name напишите имя вашего сохранненого модуля (естественно необходимо еще и указать путь к нему), а лучше воспользуйтесь кнопкой Browse и выберите ваш файл в открывшемся окне.

В Search Path ничего изменять не нужно, Делфьи сама за вас все туда добавит.

В поле Package File Name выберите имя пакета, в который будет установлен ваш компонент. Мы согласимся с предложенным по умолчанию пакетом.

Теперь нажимаем кнопочку Ok. И тут появиться предупреждение Package dclusr30.dpk will be rebuilt. Continue? Дельфи спрашивает: "Пакет такой-то будет изменен. Продолжить?". Конечно же надо ответить "Да". И если вы все сделали правильно, то появиться сообщение, что ваш компонент установлен. Что ж можно кричать Ура! Это ваш первый компонент.

Создание свойств своего типа

Теперь мы попробуем создать свойство нестандартного типа. Рассмотрим это на примере метки - TLabel. У этого компонента есть такое свойство: Alignment. Оно может принимать следующие значения: taLeftJustify, taCenter, taRightJustify. Приступаем к созданию свойства. Ничего интересного мне придумать не удалось, но тем не менее я вам покажу это на примере того свойства, которое я придумал. Оно очень простое и поможет вам разобраться. Свойство будет называться ShowType (тип TShowTp), в нашем компоненте оно будет отвечать за отображение свойства Count. Если пользователь установит свойство ShowType в Normal, то кнопка будет работать, как и работала. А если пользователь присвоит этому свойтсву значение CountToCaption, то количество кликов, будет отображаться на самой кнопке.

Для начале нам необходимо объявить новый тип. Описание типа нужно добавить после слова Type. Вот так это выглядело вначале:

```
type
TCountBtn = class(TButton)
```

Вот так это должно выглядеть:

```
type
 TShowTp = (Normal, CountToCaption);
```

```
  TCountBtn = class(TButton)
```

Здесь мы объявили новый тип TShowTp, который может принимать только два значения. Все значения, которые вы хотите добавить перечисляются через запятую.
Теперь нам понадобиться создать поле этого типа. Это мы уже умеем и делать и поэтому не должно вызвать никаких сложностей. В директиву Private напишите:

```
 FShowType:TShowTp;
```

Мы создали поле ShowType, типа TShowTp.
Конечно же необходимо добавить это свойство в инспектор объектов:

```
 property ShowType: TshowTp read FshowType write FShowType;
```

Ну и наконец, чтобы наш компонент реагировал на изменение этого свойства пользователем надо слегка изменить обработчик события OnClick. После небольшой модификации он может иметь примерно такой вид:

```
 procedure Tcountbtn.Click;
 begin

  inherited click;
  FCount:=Fcount+1;

  if ShowType = Normal then
    Caption:=Caption;

  if ShowType =
 CountToCaption then
    Caption:='Count= '+inttostr(count);

 end;
```

Объясню что произошло. Вначале мы увеличиваем счетчик на единицу. Затем проверяем какое значение имеет свойство ShowType. Если Normal, то ничего не делаем, а если CountToCaption, то в надпись на кнопке выводим количество кликов. Не так уж и сложно как это могло показаться с первого раза.
Имплантируем таймер в компонент
Очень часто бывает, что вам необходимо вставить в компонент, какой-нибудь другой компонент, например, таймер. Как обычно будем рассматривать этот процесс на конкретном примере. Сделаем так, что через каждые 10 секунд значение счетчика кликов будет удваиваться. Для этого мы встроим таймер в нашу кнопку. Нам понадобиться сделать несколько несложных шагов.
После раздела uses, где описаны добавленные в программу модули, объявите переменную типа TTimer. Назовем ее Timer. Приведу небольшой участок кода:

```
 unit CountBtn;


 interface


 uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

   StdCtrls, ExtCtrls;
```

```
 var Timer: TTimer;
 type
```

Дальше в директиву Protected необходимо добавить обработчик события OnTimer для нашего таймера. Это делается так:

```
 procedure OnTimer(Sender: TObject);
```

Поскольку наш таймер это не переменная, а компонент, его тоже надо создать, для этого в конструктор нашей кнопки напишем:

```
 constructor TCountBtn.Create(aowner:Tcomponent);

 begin
  inherited create(Aowner);

  Timer:=TTimer.Create(self);
  Timer.Enabled:=true;

  Timer.OnTimer:=OnTimer;
  Timer.Interval:=10000;

 end;
```

Здесь создается экземпляр нашего таймера и его свойству Iterval (измеряется в миллисекундах) присваивается значение 10000 (то есть 10 секунд если по простому).
Собственно осталось написать саму процедуру OnTimer. Я сделал это так:

```
 procedure TCountBtn.OnTimer(Sender: TObject);

 begin
  FCount:=FCount*2;

 end;
```

Вот примерно то, что у вас должно получиться в конце:

```
 unit CountBtn;


 interface



 uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,

  StdCtrls, ExtCtrls;


 var Timer: TTimer;
 type

  TShowTp = (Normal, CountToCaption);
```

```
 TCountBtn = class(TButton)

 private
  { Private declarations }

 FCount:integer;
 FShowType:TShowTp;

 protected
  { Protected declarations }
 procedure OnTimer(Sender: TObject);
 procedure Click;override;

 public
  { Public declarations }
 procedure ShowCount;

 published
  { Published declarations }
 property Count:integer read FCount write FCount;
 constructor Create(aowner:Tcomponent);override;
 property ShowType: TshowTp read FshowType write FShowType;
end;

procedure Register;

implementation

procedure Register;
begin
 RegisterComponents('Mihan Components', [TCountBtn]);
end;
```

```
constructor TCountBtn.Create(aowner:Tcomponent);
begin

  inherited create(Aowner);
  Timer:=TTimer.Create(self);

  Timer.Enabled:=false;
  Timer.OnTimer:=OnTimer;

  Timer.Interval:=1000;
end;


procedure Tcountbtn.Click;

begin
  inherited click;

  FCount:=Fcount+1;
  Timer.Enabled:=true;

  if ShowType = Normal then
  Caption:=Caption;

  if ShowType =
CountToCaption then
    Caption:='Count= '+inttostr(count);

end;


procedure TCountBtn.ShowCount;
begin

  Showmessage('По кнопке '+ caption+' вы сделали: '+inttostr(FCount)+' клик(а/ов)');
end;


procedure TCountBtn.OnTimer(Sender: TObject);

begin
  FCount:=FCount*2;

end;
```

end.

Если у вас что-то не сработало, то в начале проверьте все ли у вас написано правильно. Затем проверьте может у вас не хватает какого-нибудь модуля в разделе Uses.

Переустановка компонента

Очень часто бывает необходимо переустановить ваш компонент. Если вы попробуете сделать это путем выбора Component->Install Component, то Дельфи вас честно предупредит о том, что пакет уже содержит модуль с таким именем. Перед вами открывается окно с содержимым пакета. В нем вы должны найти имя вашего компонента и удалить его (либо нажать кнопочку Remove). Теперь в пакете уже нет вашего компонента. Затем проделайте стандартную процедуру по установке компонента.

Урок 78 – Написание интерфейса для БД

Задание

Задание: Создать проект в Borland Delphi 7 и обеспечить доступ к ранее созданной БД Microsoft Access “Детская поликлиника”, используя компоненты среды Delphi, получить информацию из базы данных и представить ее в форме таблицы.

В Microsoft Access создана база данных “Детская поликлиника” и наполнена информацией (приложение).

Информационная система “Детская поликлиника” хранит информацию о врачах, пациентах, заболеваниях, детских учреждениях и предоставляет следующие данные:

информацию о враче (фамилия, специализация, стаж, оклад, совместительство);

данные о больном (фамилия, возраст, адрес, детское учреждение, место работы родителей, хронические заболевания, прививки, последнее обращение к врачу);

данные о детском учреждении (наименование, адрес, количество детей, наличие карантина, выявленные инфекционные заболевания, дата последнего профилактического обследования).

список детских учреждений, в которых зафиксированы инфекционные заболевания.

Среда разработки проекта в Delphi

Delphi – это среда быстрой разработки, в которой в качестве языка программирования используется язык Delphi. Язык Delphi – строго типизированный объектно-ориентированный язык, в основе которого лежит Object Pascal. Интегрированная среда позволяет создавать, компилировать, тестировать и редактировать проект в единой среде программирования [4].

Работа над новым проектом, так в Delphi называется разрабатываемое приложение, начинается с создания стартовой формы.

Форма (Form) – основа разработки в нее помещают необходимые компоненты, создают интерфейс программы. Свойства формы определяют ее внешний вид: размер, положение на экране, текст заголовка, др.

Для просмотра и изменения значений свойств формы и ее компонентов используется окно Object Inspector (Инспектор Объектов). В верхней части окна Object Inspector указано имя объекта, значения свойств которого отображается в данный момент. В левой колонке вкладкиProperties (Свойства) перечислены свойства объекта, а в правой – указаны их значения (рис.1). Инспектор Объектов является дизайнером формы.

Исходный текст модуля разработки содержится в Окне редактора и имеет первоначальный заголовок Unit1.pas (рис.1). Проводник кода отображает объекты модуля формы (*pas), что позволяет быстро обращаться к объектам и создавать новые классы.

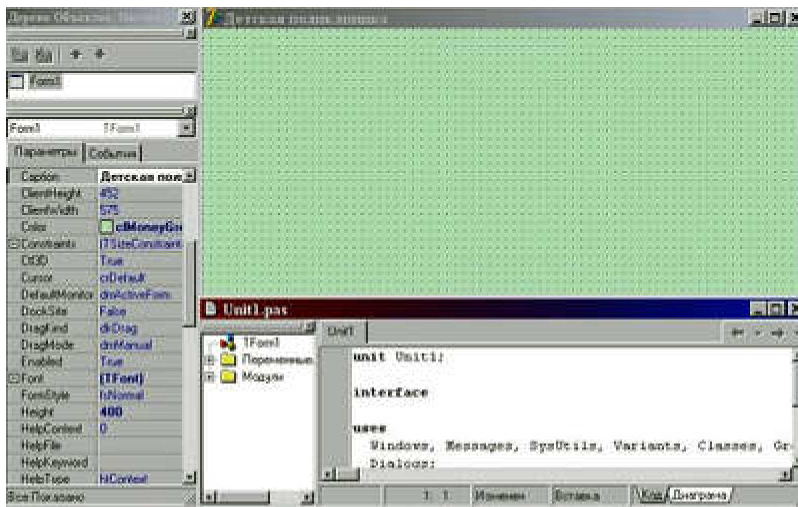


Рис. 1. Основные элементы среды разработки в Delphi

Файл модуля формы Unit1.pas (рис.1) содержит описание класса формы (размещение, поведение компонентов и функционирование обработчиков событий) и создается автоматически.

Любой проект имеет не менее шести файлов (табл.1).

Таблица 1

Основные файлы проекта

Файл	Назначение
Главный файл проекта (код проекта) – PROGECT.DPR	Основной, представляет собой программу на языке Pascal.
Первый модуль программы (модуль формы) UNIT.PAS	Автоматически появляется вначале работы его можно назвать любым другим именем. Содержит текст программы на языке Pascal.
Файл главной формы (описания формы) – UNIT.DFM	Используется для сохранения информации о внешнем виде главной формы.
Файл описание ресурсов – PROGECT.RES	Содержит иконку для проекта, создается автоматически.
Файл параметры проекта – PROGECT.OPT	Текстовый файл для сохранения установок, связанных с данным проектом.
Модули - *.PAS	Дополнительные Модули. Содержат текст программы на языке Pascal.

Разработка приложения состоит из двух этапов:

1. Создание интерфейса приложения.
2. Определение функциональности приложения.

Для создания интерфейса необходимо определить внешний вид проекта, выбрать нужные компоненты по функциональным возможностям и расположить на форме.

Для обеспечения функциональности приложения необходимо задать в Инспекторе Объектов значения свойств и процедур объектов событий, написать программный код обработки событий.

Установить измененные параметры проекта необходимо с помощью команды меню Project/Построить Project. Если требуется объединить несколько форм под единое начало – выполнить командуProject/Options.

Компиляция является обязательным процессом – процессом перевода всей программы с последующим исполнением. Данный процесс может быть выполнен на любой стадии разработки проекта. Компиляция создает:

- готовый к выполнению файл (*exe),
- динамически загружаемая библиотека (*DLL).

Запускать проект можно из среды Delphi командой Run/Run (Выполнить), из среды Windows (Название приложения.exe).

Создание приложения для работ с БД Access

Работа над новым проектом, так в Delphi называется разрабатываемое приложение, начинается с создания стартовой формы.
Создание закладок в конструкторе форм.
1. В Инспекторе Объектов изменить параметры некоторых свойств: Align —> alClient, color—>clMoneyGreen, caption—>Детская поликлиника, font, name—>Form (рис.1).

2. Создать закладки в конструкторе форм при помощи компонента PageControl на вкладке Win32 (таб.2) (Win32—>PageControl).
3. В окне Дерево Объектов создать Страницу (Контекстное меню—>Новая страница). В Инспекторе Объектов изменить параметры названия (Caption—>Больной) (рис.2).

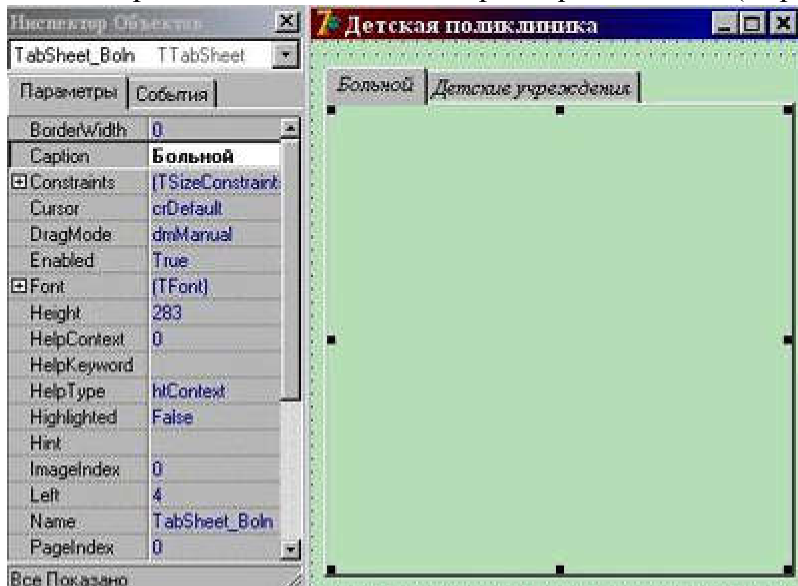


Рис. 2. Создание закладок в конструкторе форм
4. Аналогичным способом создать закладки, названия которых соответствуют названию таблиц в схеме данных БД MS Access (рис.7).

5. Поместить на форму компонент ADOConnection (вкладка ADO> ADOConnection) и выполнить его настройку (табл.2; рис.3).
Таблица 2
Значения свойств компонента ADOConnection

Свойство	Примечание
PageControl	Набор панелей с закладками. Каждая панель может содержать свой набор интерфейсных элементов и набирается щелчком по связанной с ней закладке.
Вкладка Win32	Содержит интерфейсные элементы для 32 разрядных операционных систем Windows 2000.
LoginPromt	False
ConnectionString	Сделать щелчок на кнопке с тремя точками (находится в поле значения свойств). Сделать щелчок на кнопке Build. На вкладке поставщик данных выбрать Microsoft Jet 4.0 OLEDB Provider. На вкладке Подключение указать файл базы данных “Детская поликлиника”. Дополнительно указать Права доступа: ReadWtite.

Для включения асинхронного режима необходимо установить свойство ConnectOptions компонента TADOConnection в значение coAsyncConnect. В этом случае новые запросы будут выполняться, не ожидая ответа от предыдущих запросов.

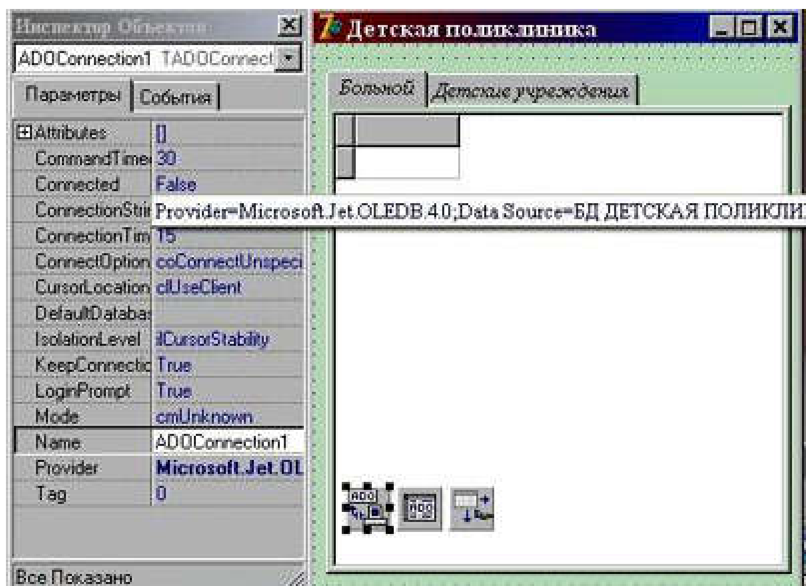


Рис.3. Настройка ADOConnection в параметрах Инспектора Объектов

6. Поместить на форму компонент ADOTable(ADODB) (Вкладка ADO—>ADOTable) и выполнить его настройку (табл.3; рис.3).

Таблица 3

Значения свойств компонента ADOTable(ADODB)

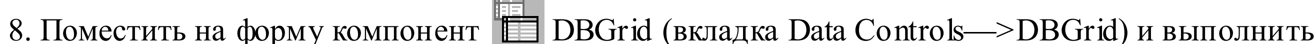
Свойство	Значение	Примечание
Connection	Имя компонента ADOConnection, обеспечи-вающего соединение с БД	Раскрыть список, находящийся в поле значения свойства, и выбрать имя компонента
TableName	Команда, обеспечивающая получение информации из определенной таблицы БД	Раскрыть список, находящийся в поле значения свойства, и выбрать название нужной таблицы
Active	True	
Технология ADO		Это синхронное/асинхронное выполнение операций с хранилищем данных. При помощи свойства connectoptions можно задать режим соединения с хранилищем данных
TADODataSet TADOCoommand		Общие компоненты для работы с технологией ADO, позволяют выполнять команды на языке провайдера данных.
DataSource Object		Хранилище данных.

7. Поместить на форму компонент DataSource (вкладка Data Access —> DataSource) и выполнить его настройку (табл. 4; рис.3)

Таблица 4

Значения свойств компонента DataSource

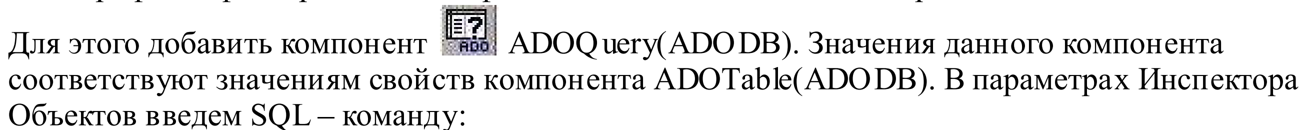
Свойство	Значение	Примечание
Connection	Имя компонента ADODataSet	Раскрыть список, находящийся в поле значения свойства, и выбрать имя компонента
DataSource Object		Хранилище данных.

8. Поместить на форму компонент  DBGrid (вкладка Data Controls—>DBGrid) и выполнить

его настройку (табл.5; рис.3)
Таблица 5
Значения свойств компонента DBGrid, Edit, Button

Свойство	Значение	Примечание
DataSource	Имя компонен-та DataSource	Раскрыть список, находящийся в поле значения свойства, и выбрать имя компонента
Align	alClient	
DBGrid		Компонент предназначен для визуализации данных, их ввода и редактирования.
Edit		Строка ввода. Предназначена для ввода, отображения или редактирования одной текстовой строки.
Button		Командная кнопка. Обработчик события OnClick этого компонента обычно используется для реализации некоторой команды.

9. Аналогично создать другие графы таблицы (проекта “Детская поликлиника”). На рисунке 4 представлен результат правильно выполненных действий задания.

10. В графе “Зафиксированные инфек.заболевания” выполним запрос по датам обследования.

Для этого добавить компонент ADOQuery(ADODB). Значения данного компонента соответствуют значениям свойств компонента ADOTable(ADODB). В параметрах Инспектора Объектов введем SQL – команду:

```
ADOQuery1.Clear := True;
ADOQuery1.Add('SELECT DU.DU, DU_Adres.DU, DU.Inf_Zab, DU.Data_obs');
ADOQuery1.Add('FROM DU');
ADOQuery1.Add('WHERE ((DU.Data_Obs) BETWEEN (DateValue("' + Edit1.Text + '")) AND
(DateValue("' + Edit2.Text + '")) );');
ADOQuery1.Active := True;
```

11. Поместить на форму компонент Edit1 и Edit2. В параметрах этих компонентов укажем дату по умолчанию: Edit1—>Text—>01.01.2009; Edit2—>Text—>31.12.2009.
12. Поместить на форму компонент Button (рис.5).
13. Создать процедуру обработки события Click, обеспечивающую выполнение SQL – команды.Процедура обработки события Click на кнопке “Обновить запрос” (Button1):

```
procedure TF.OtClick(Sender: TObject);
var
sqlfile: TextFile;
i: Integer;
begin
ADOQuery1.SQL.Clear;
ADOQuery1.SQL.Add('SELECT DU.DU, DU.Adres_DU, DU.Inf_Zab, DU.Data_obs');
ADOQuery1.SQL.Add('FROM DU');
ADOQuery1.SQL.Add('WHERE ((DU.Data_Obs) BETWEEN (DateValue("' + Edit1.Text +'")) AND
```

```
(DateValue(''' + Edit2.Text + ''')));');
ADOQuery1.Active := True;
AssignFile(sqlfile,'sql_instructions.txt');
Rewrite(sqlfile);
for i := 0 to ADOQuery1.SQL.Count-1 do Writeln(sqlfile, ADOQuery1.SQL.Strings[i]);
CloseFile(sqlfile);
end;
```

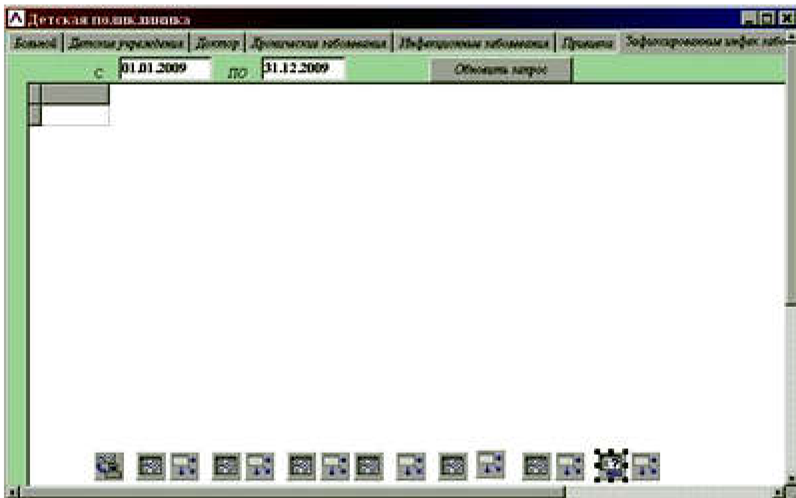


Рис.5. Окно конструктора формы. Закладка “Зафиксированные инфекционные заболевания

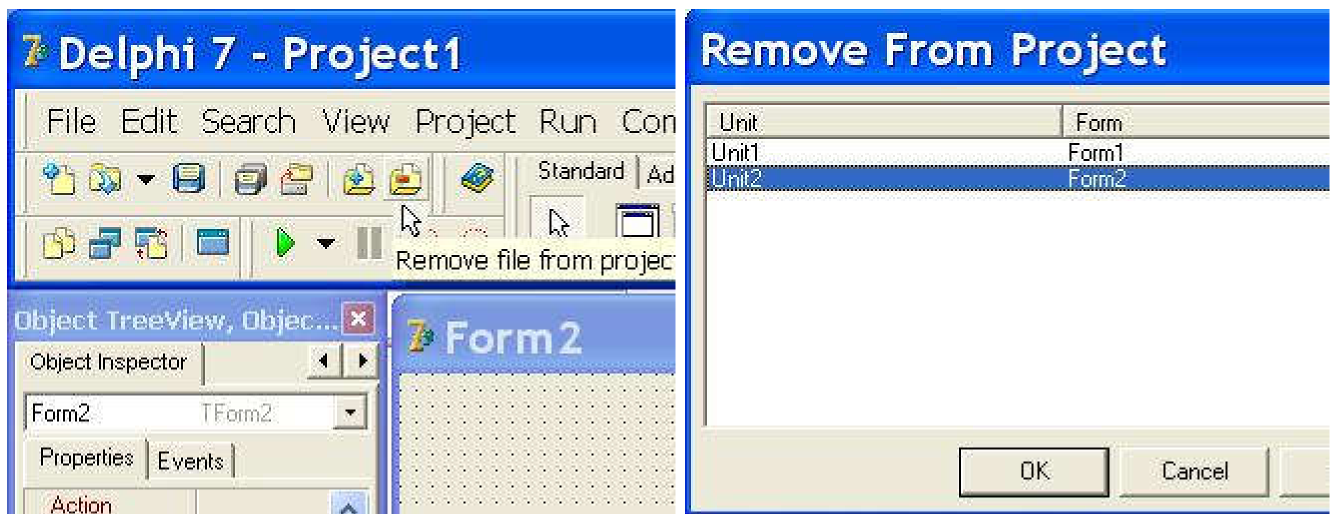
14. Последний этап. Создание исполняемый файл – приложение Project1.exe при помощи процесса компиляции (Project/Compile<Project1>).

Урок 79- Соеденение двух форм

Редкая программа обходится одной формой. Мы можем с лёгкостью создавать дополнительные формы Delphi, предоставляющие возможность, например, вести диалог с пользователем, принимать и выводить любую необходимую информацию. В этом уроке научимся создавать несколько форм Delphi дополнительно к основной.

Ввести новую форму в программу проще всего нажатием на кнопочку на главном окне Delphi. Также есть и соответствующая команда меню File -» New -» Form

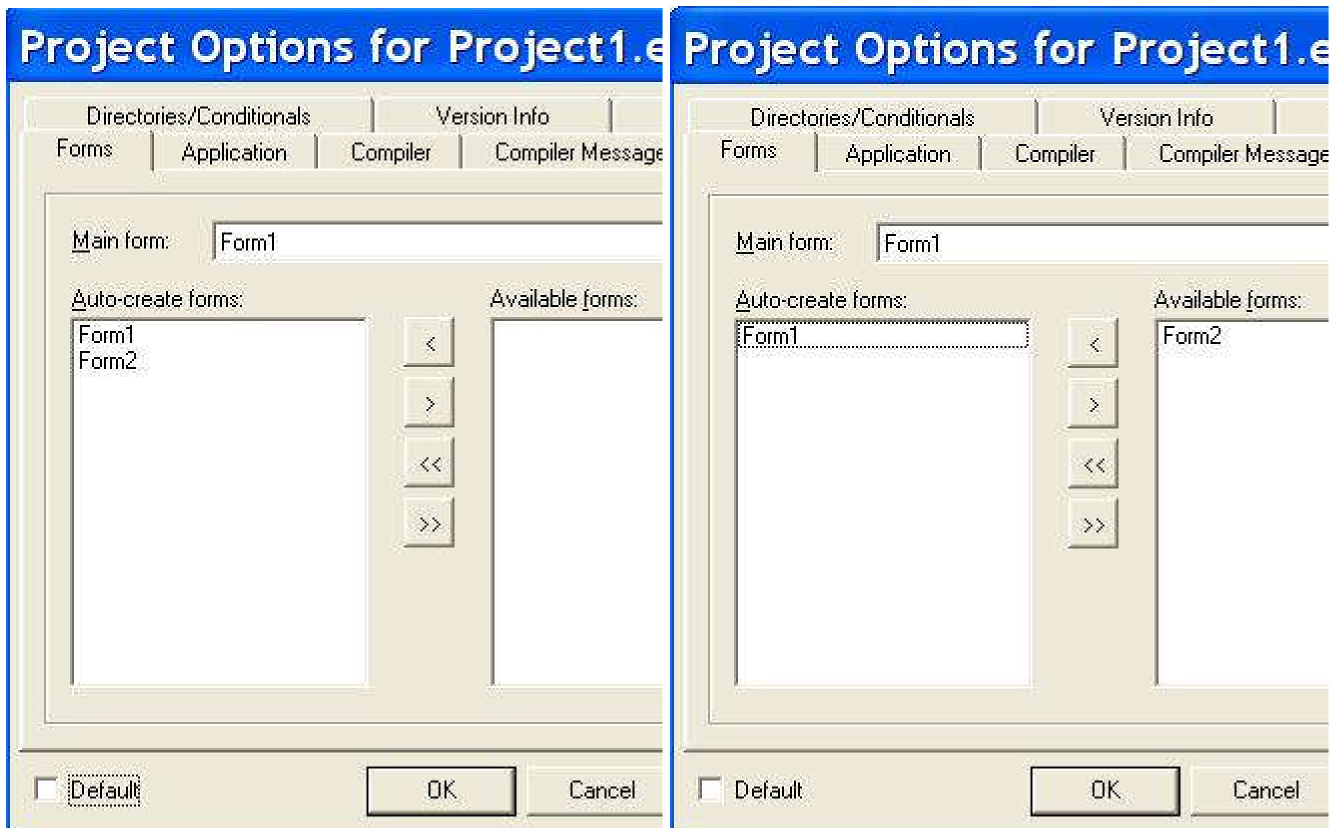
Форма создаётся вместе с новым модулем, описывающим её работу. Сразу же покажем, как **удалить Форму из программы**. Для этого также есть кнопочка, и команда меню Project -» Remove from project.... Так как Форма создаётся вместе с её модулем, то в появившемся окошке нужно выбрать модуль, который и будет удалён из проекта вместе с Формой:



Сразу возникает вопрос, а что будет, если попытаться удалить и модуль Unit1, что останется?. Выполните команду Project -» View Sourse. В редакторе кода появится новая вкладка, на которой будет отображён код Главного Окна программы (не Главной Формы, а именно Окна. В Windows ведь все объекты рассматриваются как окна.) Главное окно невидимо, но управляет всем проектом, и может работать вообще без Форм. Туда можно вставлять свой код, и писать нехилые программы, как в классическом Паскале.

Все новые формы автоматически заносятся в разряд автосоздаваемых, то есть в начале работы программы они будут автоматически создаваться сразу, одновременно с первой, главной Формой проекта. Нам не придётся этим заниматься в программе, но одновременное создание многих форм занимает ресурсы и время программы. Поэтому предлагаю сразу научиться создавать нужные формы только в нужный момент.

Во-первых, нужно отменить автосоздание форм. Выполните команду меню Project -» Options... В появившемся окошке отображаются **Main form** (главная Форма), **Auto-create** (автосоздаваемые) и**Available** (доступные) формы проекта. Нам нужно перевести Форму из разряда **автосоздаваемых** в разряд **доступных** Форм. Делается это нажатием кнопки "> " (обратная операция - нажатием кнопки "<"):



Таким образом, главная Форма **Form1** создаётся **сама**, а дополнительную форму Form2 **мы** создадим в программе при необходимости.

Если этого ничего мы не предприняли (что вполне допустимо при малом количестве дополнительных форм), то для появления новой Формы на экране достаточно написать:

```
Form2.Show; // в случае обычной Формы
Form2.ShowModal; // в случае модальной Формы
```

Если же мы перевели дополнительные Формы в разряд доступных, то перед каждым вызовом такой Формы необходимо делать проверку на существование Формы (оператором **Assigned**) таким образом:

```
if (not Assigned(Form2)) then   // проверка существования Формы (если нет, то
   Form2:=TForm2.Create(Self);    // создание Формы)
Form2.Show; // (или Form2.ShowModal) показ Формы
```

Теперь разберёмся в разнице между обычными и **модальными** Формами. **Обычные** Формы позволяют свободно переходить между всеми Формами, находящимися в данный момент на экране. **Модальная**Форма в момент вызова блокирует переход между Формами проекта до тех пор, пока не будет закрыта, и работа возможна только в ней.

При попытке компилирования программы, содержащей вызов второй формы, **Delphi** выдаст такой запрос:

Information

Form 'Form1' references form 'Form2' declared in unit 'Unit2' which is not in your USES list
wish to add it?

Yes | No | Cancel

означающий:

Форма Form1 содержит вызов формы Form2, которая объявлена в модуле Unit2, но который отсутствует в списке используемых модулей. Вы хотите добавить его?

Нужно разрешить, и в начале модуля перед директивой
{$R *.dfm}
будет добавлена фраза
uses Unit2;
В принципе, можно добавить её туда перед компиляцией "ручками", и тогда запроса не будет. Но смысл? Отвечаем "Yes" и снова жмём **F9**.

Первым делом введём в форму операцию её закрытия! Сделать это можно несколькими способами. Возьмём кнопку, напишем "Закрыть" и в обработчике **OnClick** напишем:

Form2.Close; // В принципе, достаточно просто **Close;**

Этот же оператор работает при вызове его из **меню** формы, если **меню**, конечно, туда ввести (компонент **MainMenu** на вкладке **Standard**), о чём в дальнейшем обязательно поговорим!

Теперь же необходимо рассмотреть способ закрытия Формы, который относится именно к **модальным** формам. Он используется диалоговыми окнами с вопросом, требующим подтверждения, и в других аналогичных случаях. На Форме нужно расположить несколько кнопок, нажатие которых предполагает соответствующий ответ: "Да", "Нет", "Отмена", и т.д.
У каждой кнопки есть свойство **ModalResult**, среди значений которой **mrYes**, **mrNo**, **mrCansel** и другие (посмотрите!). Значение **ModalResult** выбранной кнопки передаётся свойству **ModalResult** Формы. Это свойство отсутствует в списке свойств Формы, которые можно увидеть в Инспекторе Объектов, но программно оно доступно (напишите " Form2 ", поставьте точку и поищите в появившемся списке!).
Нажатие кнопки со значением свойства **ModalResult**, отличного от **mrNone**, приводит к закрытию Формы, даже если у кнопки отсутствует обработчик нажатия! Затем можно проанализировать свойство**ModalResult** Формы и выяснить, какой же ответ на поставленный вопрос дал пользователь:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Form2.ShowModal;
if Form2.ModalResult=mrYes then   // Этот оператор будет доступен только после закрытия Form2
  Form1.Caption:='Пользователь ответил положительно!';
end;
```

Как видно из этого примера, для доступа из одной Формы как к свойствам другой Формы, так и к свойствам её компонентов необходимо указывать имя Формы, к которой мы обращаемся. Мы также имеем доступ к данным, использующимся в модуле, описывающим её работу. Для этого необходимо указывать уже имя модуля. Например, для обращения к переменной **X** из модуля **Unit2** пишем так:**Unit2.X**.

Имеющейся возможности укрыть данные от использования в других модулях касаться пока не будем.

В момент закрытия Формы часто в программе необходимо выполнить определённые операции. Делается это в обработчике события **OnClose** Формы. А теперь рассмотрим блокировку закрытия Формы. Если вдруг понадобится заставить пользователя выполнить определённые действия перед закрытием Формы (это касается как дополнительных форм, так и основной Формы программы), нужно воспользоваться обработчиком события **OnCloseQuery**. В этом обработчике определена переменная логического типа **CanClose**. Форма будет закрыта только в том случае, когда

```
CanClose:=True;
```

Например, если создать такой обработчик **OnCloseQuery** основной Формы программы:

```
procedure TForm1.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  CanClose:=False;
end;
```

то пользователь просто не сможет закрыть программу иначе как через Диспетчер задач Windows!