В.Ю. Ачкасов

Программирование на Lazarus

ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Программирование на Lazarus

2-е издание, исправленное

Ачкасов В.Ю.

Национальный Открытый Университет "ИНТУИТ"
2016

Программирование на Lazarus/ В.Ю. Ачкасов - М.: Национальный Открытый Университет "ИНТУИТ", 2016

В курсе рассматривается среда программирования Lazarus и компилятор FPC.
Lazarus – свободно распространяемая среда программирования на Объектом Паскале, бесплатный аналог Delphi. По этой причине в настоящее время в школах, университетах, гос. учреждениях переходят на Lazarus. В курсе подробно рассматриваются все аспекты создания Windows-приложений.

(c) ООО "ИНТУИТ.РУ", 2014-2016
(c) Ачкасов В.Ю., 2014-2016

Введение в Lazarus

На этой лекции вы познакомитесь с великолепной бесплатной средой разработка программ - Lazarus. Узнаете об истории развития языков программирования, о бесплатной лицензии GNU, о том, где взять последнюю версию Lazarus и как установить ее, напишите свое первое приложение.

Цель лекции

Знакомство с историей развития языков программирования и диалектов Pascal; первое знакомство с IDE Lazarus и компилятором Free Pascal; изучение компонентов TForm, TLabel, TButton; создание первого проекта.

Исторический экскурс

Историю развития языков программирования, пожалуй, можно начать с первой в мире программистки Ады Лавлейс (Августа Ада Кинг, графиня Лавлейс, математик). Ада Лавлейс родилась 10 декабря 1815 г. в Лондоне, была известна описанием вычислительной машины (механическая машина Ч. Бэббиджа), в разработке которой она принимала участие, и созданием первой программы для нее. Ввела в употребление термины "цикл" и "рабочая ячейка". В честь Ады Лавлейс в 1975 году был назван язык программирования Ада.

Реально языки программирования получили развитие в 1945-1955 гг., когда появились первые ЭВМ (Электронные Вычислительные Машины), для которых программы составлялись вначале на машинном языке, а затем и на Ассемблере - мнемоническом представлении машинного языка. И если "чистым" машинным языком уже давно никто не пользуется, то Ассемблер все еще применяется там, где требуется либо сверхмалый размер программы, либо большая скорость ее работы, то есть, в основном, для создания критических участков ОС (Операционных Систем) или драйверов, для программирования микропроцессоров в различных платах, устройствах. Написать большую современную программу на Ассемблере - невероятно сложная, а то и невыполнимая задача.

В 1954 году появился первый язык программирования высокого уровня Фортран, и началась новая эра развития программирования.

Язык высокого уровня (или высокоуровневый язык) - это язык программирования, наиболее приближенный к человеческому языку. Он содержит смысловые конструкции, описывает структуры данных, выполняет над ними различные операции.

Современные языки высокого уровня оперируют уже целыми объектами - сложными конструкциями, обладающими определенным состоянием и поведением.

Для обучения программированию и для решения задач общего назначения наибольшее распространение получил язык программирования высокого уровня Паскаль, созданный в 1968-1969 гг. профессором Никлаусом Виртом, и названный в честь выдающегося французского математика Блеза Паскаля (между прочим, создателя первой в мире механической машины, складывающей два числа). Этот язык выгодно отличается от других языков программирования более строгими правилами в описании и использовании данных различного типа. Паскаль - структурный язык, небольшой и эффективный, способствующий выработке у программиста хорошего стиля программирования. В школах и ВУЗах всех стран в мире по сей день изучают ту или иную реализацию Паскаля.

Поскольку Lazarus основан на Паскале (точнее, на Объектном Паскале), историю других высокоуровневых языков в рамках данного курса мы рассматривать не будем.

В 1983 г. фирма Borland, известная разработкой Delphi - платного предшественника Lazarus, выпустила Turbo Pascal - интегрированную среду разработки программ на языке Паскаль. Turbo Pascal - это компилятор, компоновщик, редактор кода и отладчик в одном окне. Он подобен швейцарскому ножу, где множество разных инструментов вмонтировано в единое устройство. Для программистов Turbo Pascal примечателен тем, что он стал своеобразным прародителем сред быстрой разработки программ.

В 1986 г. появился язык Object Pascal (Объектный Паскаль), разработанный в фирме Apple Computer. Этот диалект Паскаля уже мог

оперировать объектами.

В 1989 г. объектное расширение Паскаля было добавлено и в Turbo Pascal фирмы Borland.

В 1994 г. была выпущена первая версия Delphi - Графическая интегрированная среда быстрой разработки программ для Windows. Этот факт дал невероятный толчок развитию таких сред, в которых разработка интерфейса программы для программиста вместо нудной рутины, превращалась в забавный конструктор форм. В современных средах можно создать программу, даже не дотрагиваясь до клавиатуры - исключительно с помощью мыши. Правда, подобной программе вряд ли можно будет придать сколь-либо полезные функции.

Все эти языки и среды были платными, часто оказывались недоступны образовательным учреждениям в силу своей дороговизны. В 1993 г. начались работы над проектом Free Pascal (FPC - Free Pascal Compiler). Первая версия FPC появилась лишь в июле 2000 г., она была полностью бесплатная и поддерживала множество платформ: Windows, Linux, FreeBSD, Mac OS X и т.п. FPC - это бесплатный открытый проект, его исходные коды для изучения или модификации доступны каждому! Чуть позже появился Lazarus - единственная в мире бесплатная графическая среда для быстрой разработки программ, использующая компилятор FPC. Как и FPC, Lazarus распространяется на условиях лицензии GNU GPL (General Public License). Если не особо вдаваться в юридические подробности, то GNU GPL - это лицензия, предоставляющая пользователю права свободно и бесплатно копировать, модифицировать и распространять (в том числе и на коммерческой основе) данный продукт. По этой же лицензии распространяются все версии ОС (Операционной Системы) Linux - бесплатном и довольно серьезном конкуренте Windows.

Итак, мы с вами будем говорить о последней (на момент написания курса) версии Lazarus - 1.0.10, работающей с компилятором FPC 2.6.2. Lazarus - молодой и бурно-развивающийся проект, новые версии выходят довольно часто, так что вы, вероятно, будете пользоваться более свежей версией. Тем не менее, на курсе рассматриваются фундаментальные вопросы программирования, которые едва ли будут пересматриваться. Так что можете изучать предоставленный материал,

пользуясь версией Lazarus 1.0.10 или любой более свежей.

В силу того, что подавляющее большинство пользователей по-прежнему работает в операционной системе Windows, мы будем рассматривать работу с Lazarus именно в этой среде (автор использовал ОС Windows XP SP3). Рамки курса небезграничны, а описание особенностей работы в других платформах может лишь запутать учащихся на начальном этапе.

Впрочем, разработка программ под другие платформы имеет не так уж много отличий, чтобы создать вам непреодолимые трудности при переходе на другую платформу.

Что такое Lazarus?

Lazarus - это IDE (Integrated Development Environment) - Интегрированная Среда Разработки программ, использующая компилятор FPC (Free Pascal Compiler), редакторы кода, форм, Инспектор Объектов, отладчик и многие другие инструменты.

Еще говорят, что среда Lazarus - это RAD (Rapid Application Development) - среда Быстрой Разработки Приложений.

До сих пор среды разработки программ, подобные Lazarus, были исключительно платными. Lazarus же стал первой (и пока единственной) IDE, доступной образовательным и государственным учреждениям совершенно бесплатно. Более того, Lazarus является проектом Open Source - проектом с открытым исходным кодом. Многие программисты по всему миру принимают участие в его развитии, исходный код Lazarus доступен для изучения и модификации. Lazarus имеет поддержку множества языков, в том числе и русского, что выгодно отличает его от других IDE.

Где взять?

Lazarus, как уже говорилось, - бесплатный и свободно распространяемый продукт. Благодаря этому, Lazarus все чаще используют для изучения программирования в школах и ВУЗах, а также

на многих предприятиях. Но где его взять? На официальном сайте производителя: ссылка: http://lazarus.freepascal.org

В правой верхней части сайта вы увидите следующую картинку:

Рис. 1.1. Выбор и закачка необходимой реализации

Здесь вы сможете выбрать реализацию именно под вашу платформу, от Windows до Mac OS X, как 32-х так и 64-х разрядную. При написании курса использовался 32-х разрядный Lazarus для платформы Windows.

Нажав кнопку "Download Now" вы скачаете последнюю версию Lazarus. Кроме того, выбрать последнюю необходимую реализацию и скачать ее вы можете по адресу: ссылка: http://sourceforge.net/projects/lazarus/files/

В этом случае, перейдя по ссылкам, вы получите доступ к закачке нескольких файлов, например,

- lazarus-1.0.10-fpc-2.6.2-win32.exe
- lazarus-1.0.10-fpc-2.6.2-cross-arm-wince-win32.exe
- README.txt

Нам нужен только первый файл из этого списка. Второй файл является расширением для разработки программ под Windows CE (она же WinCE) - это вариант операционной системы Microsoft Windows для наладонных компьютеров, смартфонов и встраиваемых систем. На данном курсе эту возможность мы рассматривать не будем. Последний файл - простой текстовый файл с информацией о версии, он нам тоже не нужен.

Как установить?

Lazarus устанавливается достаточно просто. Собственно, ничего менять нам не придется, оставим все параметры, предложенные установщиком по умолчанию. Для начала выберем русский язык установки, затем все время будем нажимать кнопки "Далее". Лишь в предпоследнем окне установщика при желании можно поставить флажок "Создать значок на Рабочем столе". Когда укажем все параметры, начнется установка Lazarus. Придется подождать пару минут, пока распакуются и скопируются множество файлов. И, наконец, кнопка "Завершить" для закрытия окна установщика. Все, Lazarus у нас есть! Мы можем его загрузить.

В самом начале Lazarus выглядит несколько неопрятно:

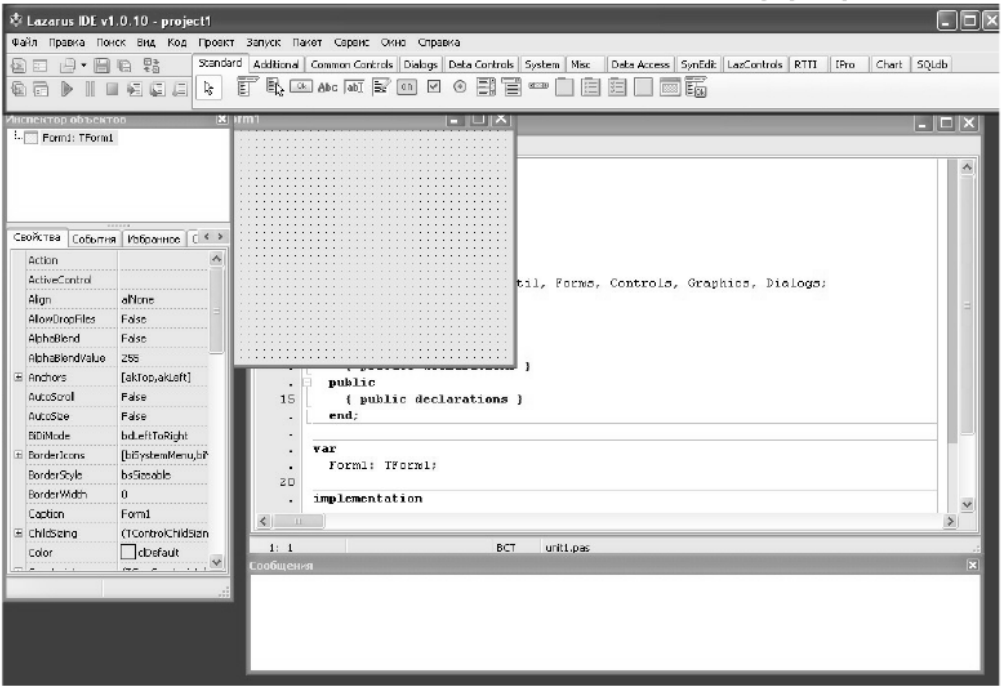


Рис. 1.2. Первый запуск Lazarus

Lazarus состоит из нескольких окон (которые стоит подравнять, чтобы они занимали весь рабочий стол и не мешали друг другу):

1. Главное окно
2. Инспектор объектов
3. Редактор форм
4. Редактор кода
5. Окно сообщений

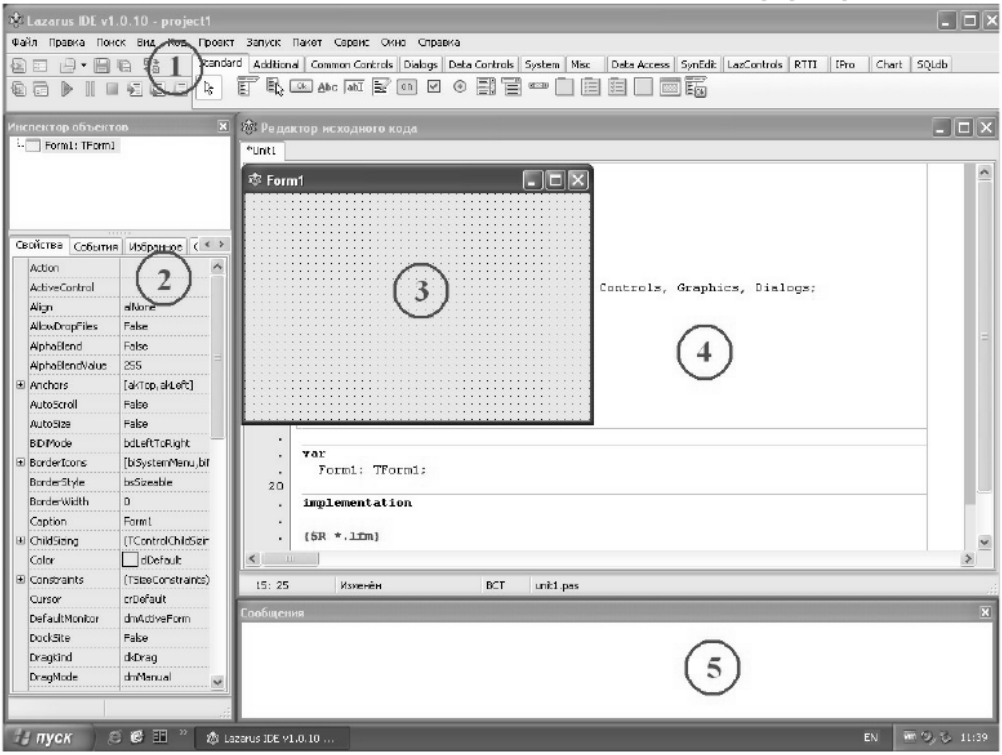


Рис. 1.3. Окна Lazarus

Главное окно

Главное окно состоит из следующих элементов:

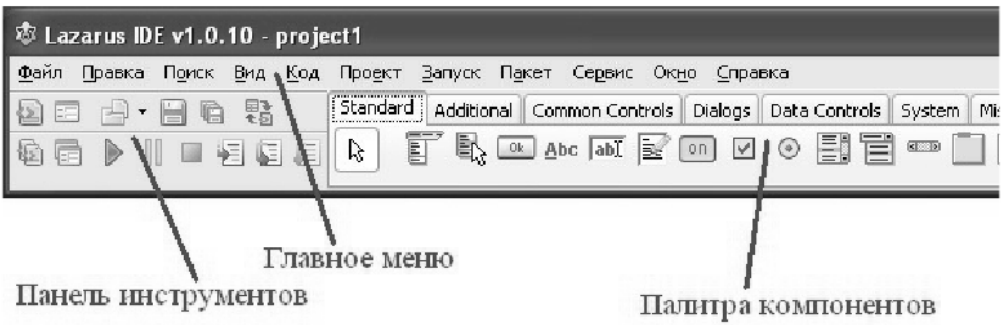


Рис. 1.4. Главное окно Lazarus

1. Главное меню содержит все команды, необходимые для правки, компиляции, отладки программы, для запуска различных вспомогательных утилит.
2. Панель инструментов содержит кнопки чаще всего применяемых команд (эти же команды можно выполнить и с помощью Главного меню).
3. Палитра компонентов содержит множество вкладок, на которых содержится богатый выбор компонентов из собственной библиотеки компонентов Lazarus - LCL (Lazarus Component Library).

Инспектор объектов

Окно Инспектора объектов состоит из двух частей:

- Дерево объектов, в котором в древовидной форме располагаются все объекты, используемые в текущей форме.
- Окно с вкладками, в котором можно настраивать различные свойства текущего объекта. Несмотря на то, что имеется 4 вкладки (Свойства, События, Избранное, Ограничения), чаще всего используются только первые две. О свойствах и событиях мы поговорим подробней в следующих лекциях.

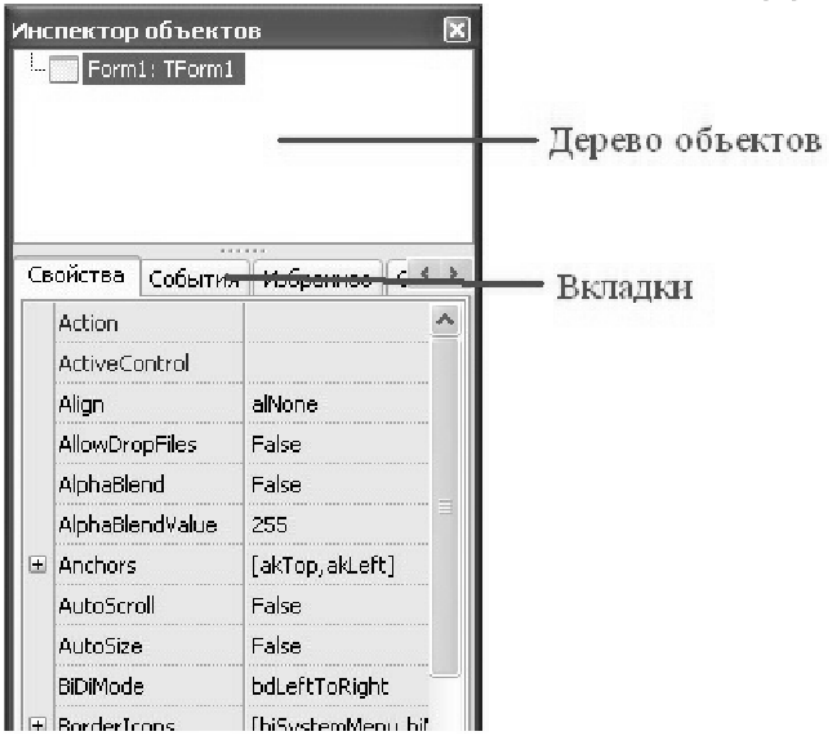


Рис. 1.5. Инспектор объектов

Редактор форм, Редактор кода и Окно сообщений

Последние три окна проще. Редактор форм предназначен, соответственно, для редактирования формы - положения и размеров компонентов, размещенных на этой форме.

Несмотря на явную схожесть, форма и окно приложения - не одно и то же. Форма - это то, что видит программист в процессе разработки проекта, а окно - это то, что увидит пользователь, когда загрузит нашу программу.

Редактор кода содержит исходный код, который нам придется вводить и модифицировать. Редактор обладает рядом полезных умений: подсвечивает синтаксис команд, делает авто-отступ и авто-завершение команд, выводит необходимые подсказки, в общем, сильно облегчает жизнь программисту.

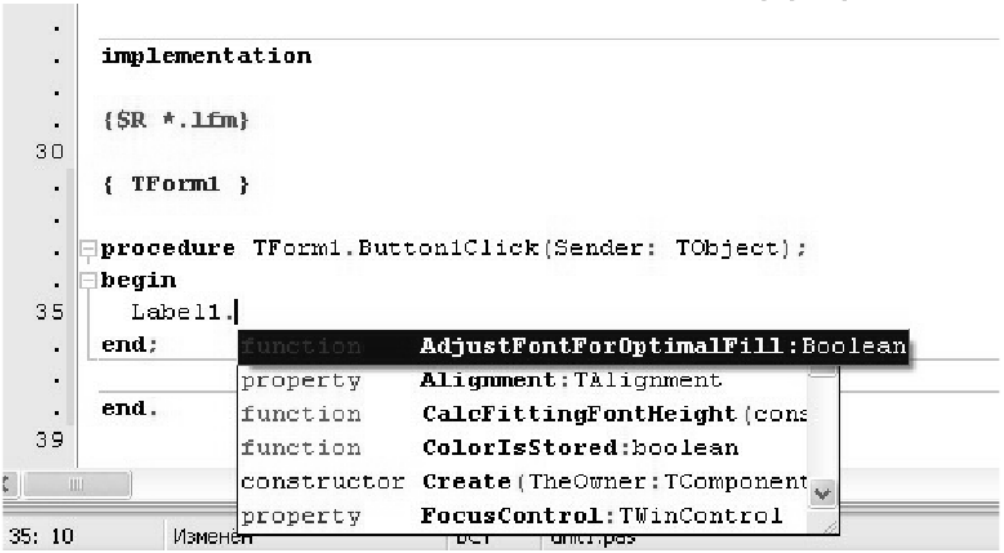


Рис. 1.6. Редактор кода

Нам придется часто переключаться между Редактором форм и Редактором кода. Проще всего это делать кнопкой <F12>.

И, наконец, Окно сообщений выводит различные сообщения: о найденных ошибках, о завершении компиляции, о наличии объявленных, но неиспользуемых переменных и т.п.

Схожесть и отличия с Delphi

Я не мог не упомянуть о Delphi, который является платным аналогом Lazarus, и его предшественником. И Lazarus, и Delphi поддерживают код Объектного Паскаля, используют визуальные и невизуальные компоненты, имеют схожий интерфейс. Взгляните на рисунок:

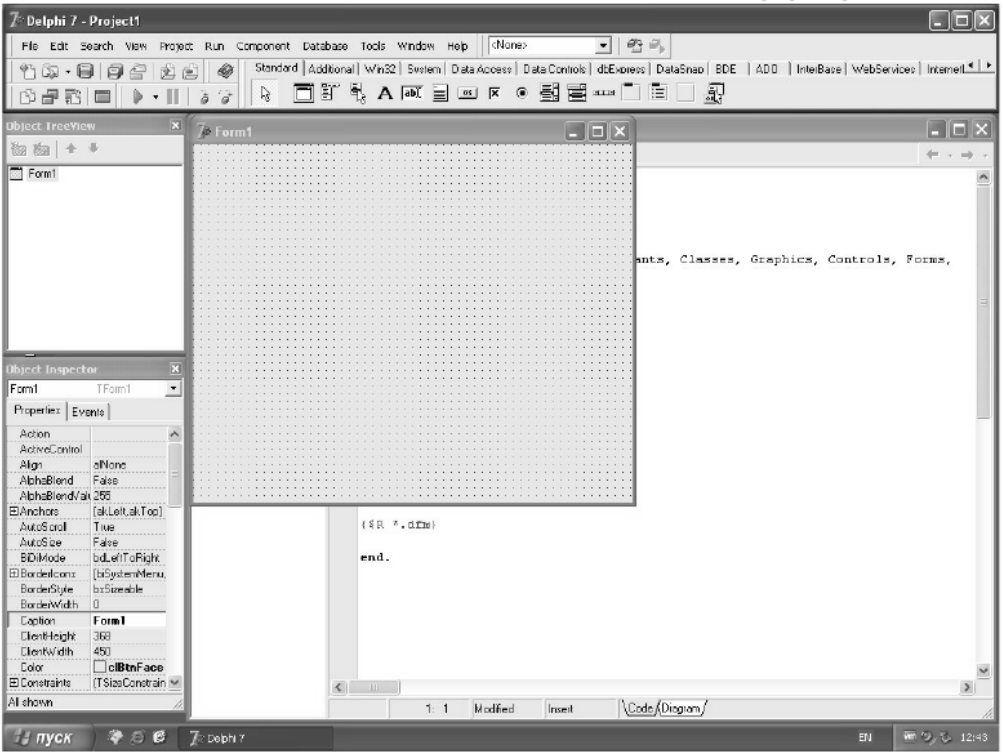


Рис. 1.7. Окна Delphi 7

Как видите, Lazarus очень похож на Delphi, однако есть и отличия.

- Lazarus, в отличие от Delphi, бесплатен, и может свободно и легально применяться в любом учебном, государственном или производственном учреждении, или дома.
- Lazarus имеет собственную библиотеку компонентов LCL (Lazarus Component Library), а Delphi - VCL (Visual Component Library). Однако VCL и LCL во многом так похожи, что программист при работе с компонентами почти не ощущает разницы. Часто (но не всегда) проекты, написанные на Delphi можно без потерь компилировать на Lazarus.
- Lazarus кросс-платформенная IDE, то есть, поддерживает различные операционные системы. Существует, правда, Kilyx - реализация Delphi для Linux, однако Lazarus имеет реализации для гораздо большего списка операционных систем, причем как 32-х, так и 64-х разрядных версий.

- Lazarus, в силу того, что моложе Delphi, пока имеет меньшую поддержку: дополнительные компоненты сторонних разработчиков, книги на русском языке, сайты, посвященные языку и т.п.
- Lazarus имеет менее развитые средства для работы с Базами Данных. Будем надеяться, это временный недостаток.
- Некоторые компоненты LCL в Lazarus еще "сырые" - иногда попадаются свойства, которые не работают. Чаще всего, это второстепенные свойства, так что можно смело использовать компоненты и без них.

Несмотря на все отличия, эти IDE так похожи, что можно смело утверждать - Delphi-программист почти без усилий сможет пользоваться Lazarus, и наоборот. А бурное развитие молодого Lazarus гарантирует, что в будущем его немногочисленные недостатки будут исправлены.

Первая программа

Прежде всего, давайте создадим где-нибудь общую папку, в которую затем будем складывать все наши проекты. Пусть это будет

C:\Education\

(education - образование)

Каждый проект по правилам, должен сохраняться в отдельную папку. Чтобы не запутаться, будем давать этим папкам имена, соответствующие номеру лекции, и номеру проекта в ней (лекция-проект). То есть, в первой лекции первый проект будет сохранен в папку

C:\Education\01-01

Вы можете выработать и собственные правила наименования папок - суть от этого не изменится, лишь бы вы сами потом в них не запутались. Мы несколько раз упомянули слово проект.

Проект - это то, что разрабатывает программист. Когда проект готов и скомпилирован, получается программа, с которой может работать пользователь. Проект - это набор связанных файлов различного типа, а

программа - это полученный в результате компиляции исполняемый файл. Подробней о проектах мы поговорим в следующей лекции.

По традиции, первая созданная программа должна просто выводить сообщение "Hello, world!" ("Привет, мир!"). Не будем отступать от традиций, и сделаем это тремя разными способами. Итак, создадим на диске указанные выше папки, куда сохраним наш первый проект. Загружаем Lazarus, если он еще не загружен, и выделяем редактор форм. В левой части, если вы не забыли, находится Инспектор объектов, и в нем выделена вкладка "Свойства" - это нам и нужно. Среди свойств найдите `Caption`, и вместо текста

Form1

который там находится по умолчанию, впишите

Hello, world!

По окончании этой процедуры нажмите <Enter>. Как только вы это сделаете, текст в заголовке формы изменится:

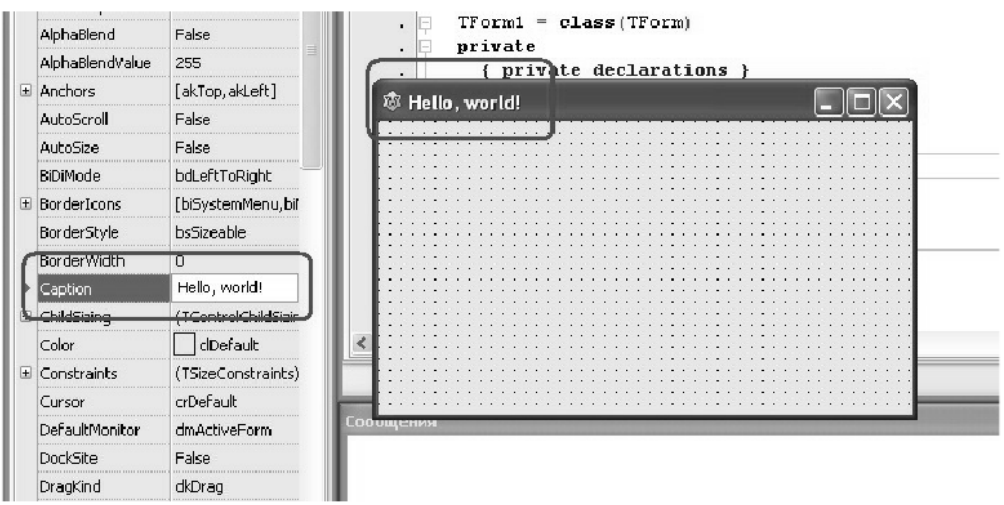


Рис. 1.8. Первые шаги

Теперь обратите внимание на Палитру компонентов. На вкладке Standard, четвертый значок изображает кнопку с надписью "Ok" на ней. Эта кнопка нам и нужна.

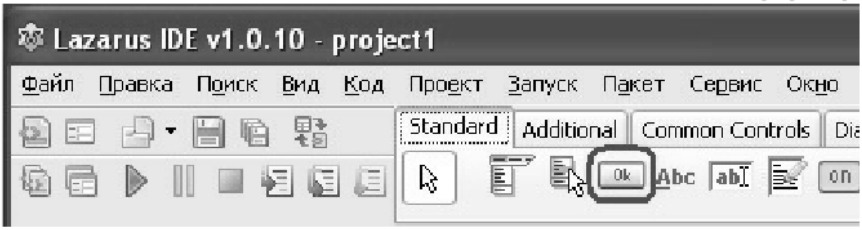
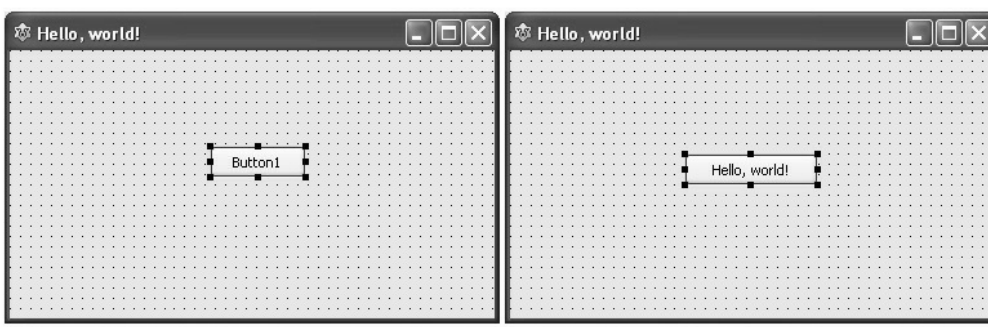


Рис. 1.9. Кнопка TButton в Палитре компонентов

Щелкните по ней мышью, а затем щелкните уже по форме, примерно по центру. На форме тут же появится кнопка, обрамленная рамочкой, с надписью Button1 - такое имя Lazarus дал кнопке по умолчанию. Рамочка вокруг кнопки говорит о том, что ухватившись мышью за одну из ее сторон или углов, мы сможем менять размеры кнопки. Ухватившись за саму кнопку, мы сможем перемещать ее по форме.

Слева, в Инспекторе объектов, список свойств также изменился - некоторые остались прежними, другие добавились. Поступим, как и в прошлый раз - в свойстве `Caption` вместо Button1 впишем Hello, world!. Затем мышью изменим размеры и расположение кнопки, чтобы у нас получилось примерно следующее:



а) кнопка до обработки *б) кнопка после обработки*

Рис. 1.10. Кнопка до и после обработки

Обратите внимание: перемещать кнопку по форме можно не только мышью, но и клавишами <Ctrl> + <Кнопки управления курсором> (стрелки вверх, вниз, влево, вправо). А изменять ее размеры - клавишами <Shift> + <Кнопки управления курсором>. Этот способ удобней для более точной настройки положения и размеров любых компонентов, не только кнопки.

Мы попробовали два способа вывести нужный текст. Теперь поработаем с исходным кодом. Щелкните дважды по кнопке, и Lazarus откроет Редактор кода, создав обработчик для этой кнопки, и установив курсор внутрь него. Здесь нам пока ничего понимать не нужно, просто впишите текст, прямо туда, где находится курсор:

```
ShowMessage('Hello, world!');
```

Чтобы получилось так:

Рис. 1.11. Код обработчика кнопки

На этом наш проект закончен. Осталось сохранить его и скомпилировать в программу. Выберите команду меню Файл -> Сохранить все, или (что проще) нажмите соответствующую кнопку на Панели инструментов:

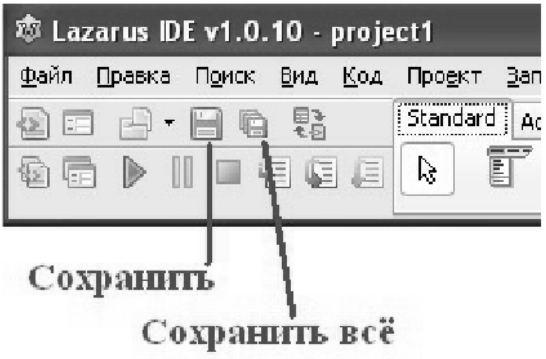


Рис. 1.12. Кнопки сохранения

Кнопка "Сохранить" сохраняет изменения текущей формы, кнопка "Сохранить всё" - изменения всех форм и модулей проекта. Поскольку форма у нас одна, то можно нажать любую из них. Как только вы это сделаете, выйдет окно сохранения проекта. Вы помните, в какую папку мы будем сохранять первый проект первой лекции? Туда и сохраняйте. Название проекта оставьте без изменений, подробней об этом мы поговорим в следующей лекции.

Как только вы сохраните проект, выйдет еще один запрос - на сохранение файла модуля Unit1. Здесь мы тоже оставляем название по умолчанию, нажмем лишь кнопку "Сохранить". Обратите внимание - кнопки "Сохранить" и "Сохранить всё" стали неактивными - это означает, что ни в форме, ни в проекте в целом у нас изменений нет.

Хорошо, проект мы сохранили, однако работающей программы у нас пока нет. Чтобы ее получить, нужно скомпилировать проект. Причем сделать это можно тремя командами Главного меню:

- Запуск -> Компилировать
- Запуск -> Собрать
- Запуск -> Запустить

Последняя команда не только компилирует проект и создает загрузочный файл программы, но и сразу запускает его на выполнение, так что в большинстве случаев предпочтительней именно эта команда. Удобнее всего воспользоваться соответствующей кнопкой на Панели

инструментов:

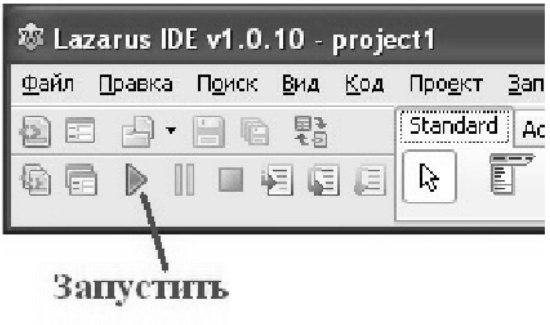


Рис. 1.13. Кнопка запуска

Нажмите эту кнопку, проект скомпилируется и сразу же запуститься. Как только мы нажмем кнопку "Hello, world!" в окне программы, выскочит записанное в Редакторе кода сообщение:

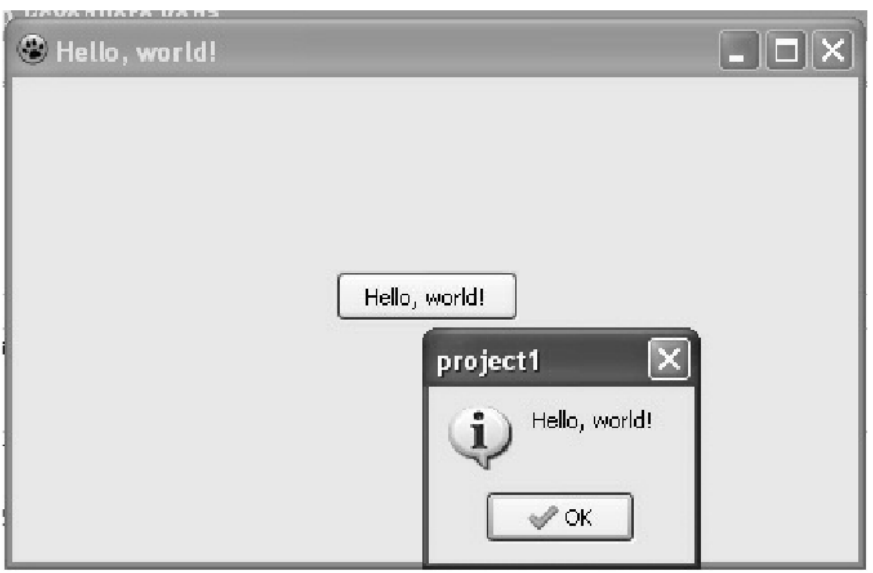


Рис. 1.14. Работа нашей программы

Нажмите кнопку "Ок", закрыв сообщение, после чего закройте саму программу (не проект!). Если выйдет подобное сообщение:

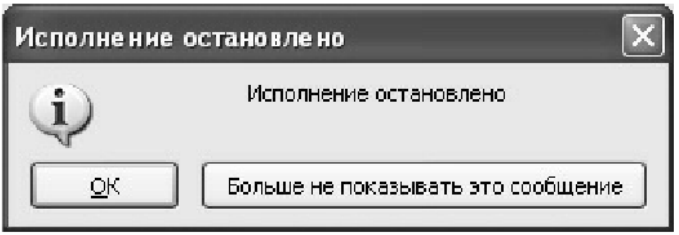


Рис. 1.15. Сообщение об остановке выполнения

то советую нажать кнопку "Больше не показывать это сообщение", чтобы в дальнейшем оно вам не надоедало. После этого можете закрыть и Lazarus. Исполняемый файл project1.exe с нашей программой вы найдете в папке, куда сохраняли проект. Это уже вполне работоспособная программа, её можно запустить прямо из этой папки или переслать другу, чтобы он позавидовал вашим новым знаниям (для выполнения программы требуется только один файл project1.exe, остальные файлы не нужны - это файлы проекта, которые нужны только программисту).

Полезные ссылки

В заключение лекции дам вам несколько ссылок, которые могут оказаться полезными при изучении материалов курса.

- ссылка: http://lazarus.freepascal.org/ - Официальный сайт Lazarus. Здесь вы сможете и новые версии найти, и новости почитать.
- ссылка: http://wiki.freepascal.org/Main_Page/ru - Русскоязычная документация по Lazarus и Free Pascal. Если постараться, тут можно найти ответы почти на все вопросы, которые непременно у вас будут возникать.
- ссылка: http://www.cyberforum.ru/lazarus/ - Большой форум по Lazarus. Форумы вообще очень полезны и информативны, а форумы о программировании особенно. Даже если вы и не смогли найти в Интернете решение вашей проблемы, на форуме непременно найдется какой-нибудь "гуру", который вам поможет советом.
- ссылка: http://www.freepascal.ru/ - Русскоязычный информационный портал о Free Pascal и Lazarus. Тоже весьма полезный сайт.
- ссылка: http://lazarus.16mb.com/ - А это уже мой сайт, который задумывался для поддержки данного курса. В разделе "Скачать" вы

сможете найти исходные коды всех примеров из всех лекций курса на случай, если у вас что-то не будет получаться, или будет просто лень набивать код самостоятельно. Там же вы найдете и сможете скачать все инструменты программиста, которые рассматриваются на данном курсе, за исключением самого Lazarus - программа очень часто обновляется, а на сайте разработчика всегда самая свежая версия, поэтому скачивать Lazarus разумней там. Так же, на сайте я постараюсь почаще выкладывать дополнительный материал, не вошедший в данный курс.

Анатомия проекта

На этой лекции вы узнаете, как настроить IDE Lazarus для более удобной работы, познакомитесь с составом проекта, напишите интерактивную программу. Также вы изучите новые компоненты TLabel и TEdit.

Цель лекции

Знакомство с настройками IDE Lazarus, с составом проекта, изучение новых компонентов TLabel и TEdit.

Настройка IDE

Чтобы облегчить дальнейшую работу с Lazarus, выполним некоторые настройки его IDE. Загружайте Lazarus. По умолчанию включена опция загрузки последнего проекта, поэтому у вас должен был загрузиться проект из первой лекции. Это очень удобно, если вы работаете над каким-нибудь большим проектом, на разработку которого уходит не один день. Однако это будет неудобно при изучении Lazarus - ведь мы будем делать множество небольших программ, выполняющих разные задачи, и для этого нам нужно будет вначале закрыть старый проект, а затем создать новый. А это неудобно.

Выберите в главном меню команду "Сервис -> Параметры", откроется окно "Параметры IDE". Нам нужен раздел "Файлы", расположенный в ветке "Окружение":

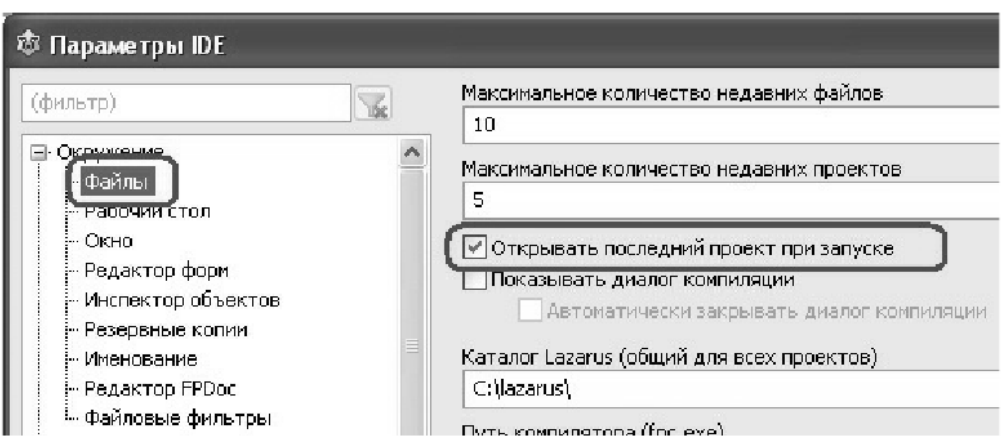


Рис. 2.1. Параметры - Файлы

В этом разделе имеется флажок "Открывать последний проект при запуске". Если этот флажок включен, то каждый раз при запуске будет открываться последний проект. Если выключен - будет создаваться новый проект, что нам и нужно. Выключим его, нажмем кнопку "OK", закроем Lazarus и снова запустим. Ну вот, совсем другое дело, создался новый проект. Ничего в проекте не меняя, просто сохраним его в папку

C:\Education\02-01

(вы помните о нашем договоре по поводу наименования папок с проектами?). Нажмем кнопку "Запустить" на Панели инструментов (зеленая стрелочка вправо), чтобы проект скомпилировался, и полученная программа сразу же загрузилась в память. Появится простое окошко нашей программы, ведь в ней ничего, кроме формы, нет. Закройте полученную программу, оставив Lazarus с проектом открытым.

Затем откройте папку с проектом с помощью Проводника Windows или привычного для вас файлового менеджера (Total Commander, Far, и т.п.). В Проводнике установите вид отображения "Таблица". Вы увидите примерно такую картинку:

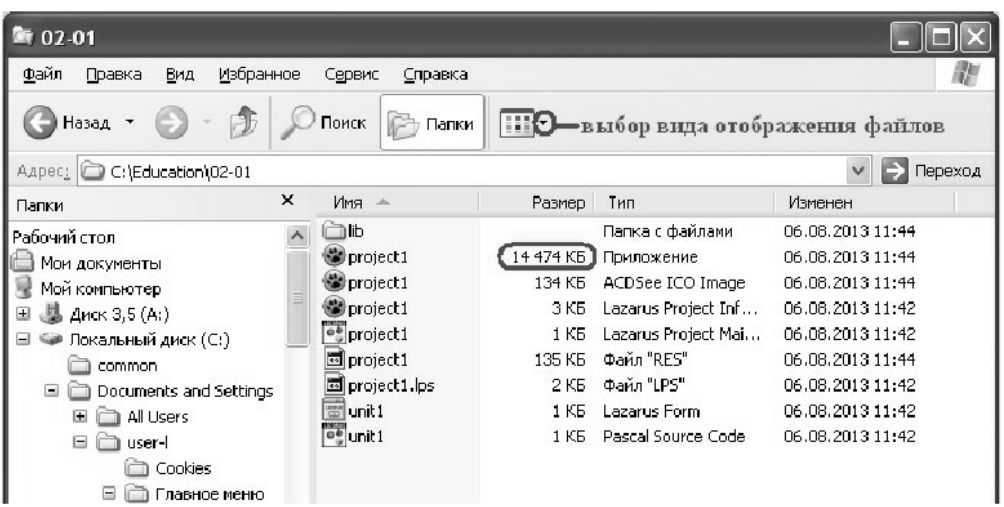


Рис. 2.2. Папка проекта с пустой формой

Наша программа получила название project1.exe, в колонке "Тип"

значится "Приложение". А вот в колонке "Размер" указано 14 474 килобайта (у вас может чуть отличаться, это не страшно). Ого! Программа с пустым окошком, которая совсем ничего не делает, "весит" больше 14 мегабайт!!! Как такое может быть?!

Дело в том, что при компиляции Lazarus по умолчанию добавляет в программу всевозможную отладочную информацию, в результате чего и получается такой большой размер. Это полезно, когда вы отлаживаете программу (об отладке и тестировании программ мы будем говорить в другой лекции), но лишнее в нашем случае. К сожалению, отключить эту функцию можно только для текущего проекта. Делается это так: выберите команду Главного меню "Проект -> Параметры проекта":

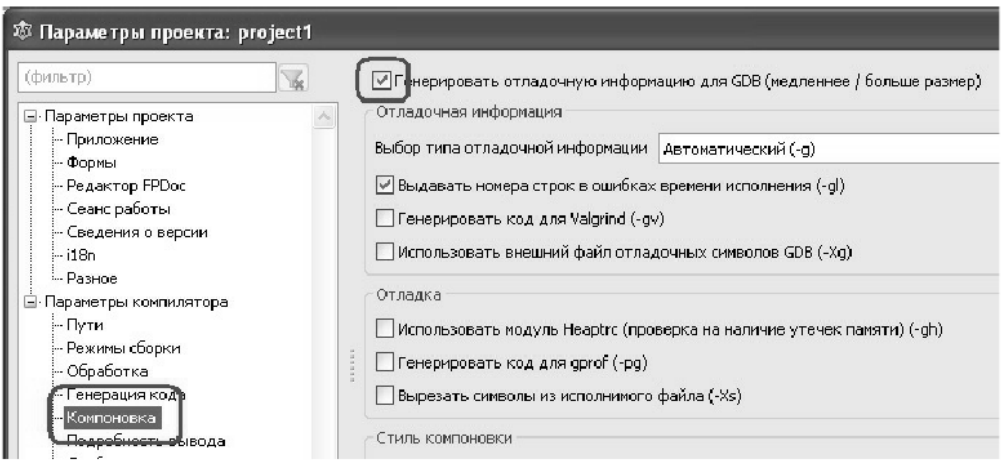


Рис. 2.3. Настройки проекта

В разделе "Компоновка" снимите флажок "Генерировать отладочную информацию для GDB", нажмите "OK" и заново запустите программу. После того, как программа запустится, закройте ее и снова посмотрите на полученный размер. Ну вот, теперь порядок, размер чуть больше полутора мегабайт. Тоже немало, если разобраться, но вполне терпимо, учитывая подключенные к проекту стандартные модули с поддержкой кроссплатформенности. В дальнейшем вы можете либо не обращать внимания на размер учебных программ - снимать этот флажок (после отладки, если она нужна) только для своих готовых реальных проектов, либо снимать его в каждом новом проекте. Это на ваше усмотрение. А если вам хочется получить совсем уж малый размер программы, то можно дополнительно воспользоваться каким-нибудь программным

упаковщиком. ASPack, например, позволяет сжимать *.exe и *.dll файлы до 80%! Такой упаковщик, кстати, не только сжимает, но и шифрует вашу программу, создавая ей дополнительную защиту. Сильного хакера это, правда, не остановит, но лишней головной боли ему прибавит.

Также в настройках можно изменить размеры ячеек в сетке из точек, которая отображается в Редакторе форм. Эта сетка нужна для того, чтобы программист мог выравнивать компоненты на форме относительно друг друга. Выберите команду Главного меню "Сервис -> Параметры" и в разделе "Окружение" выберите "Редактор форм". В правой верхней части расположен раздел "Сетка", где можно изменять ее размеры (по умолчанию, установлено по 8 пикселей для оси X и для оси Y), разрешать или запрещать показ сетки, и т.д. Некоторые программисты предпочитают делать ячейки сетки меньше - 6 или даже 4 пикселя по обеим осям, мотивируя это тем, что чем чаще сетка, тем точнее можно подровнять компоненты. Может быть, они и правы, однако имейте в виду, что разработчики Lazarus выставляли по умолчанию наиболее удобные параметры. В общем, и тут установите настройки на свой вкус, или оставьте их по умолчанию.

Из чего состоит проект

Мы то и дело повторяем слово "Проект". В первой лекции мы говорили, что проект - это набор связанных файлов различного типа, из которых, в конце концов, после компиляции, получается программа.

Из каких же файлов состоит проект?

Выберите команду Главного меню "Сервис -> Параметры", и в ветке "Окружение" перейдите на раздел "Файловые фильтры". Вы увидите 6 основных типов файлов, которые могут встречаться в проекте:

- Модуль Lazarus (*.pas;*.pp)
- Проект Lazarus (*.lpi)
- Форма Lazarus или Delphi (*.lfm;*.dfm)
- Пакет Lazarus (*.lpk)
- Исходный код проекта Lazarus (*.lpr)
- Иной файл Lazarus (*.inc;*.lrs;*.lpl)

Если мы перейдем в папку с нашим проектом, то увидим, что он состоит из восьми файлов:

- project1.exe (Исполняемый файл программы).
- project1.ico (Файл с "иконкой" проекта - изображением в виде лапы гепарда, которое появляется в верхнем левом углу окна программы).
- project1.lpi (Информационный файл проекта). Если вы желаете открыть данный проект в Lazarus, то запускать нужно именно этот, информационный файл.
- project1.lpr (Исходный файл проекта). Запуск этого файла также приведет к запуску Lazarus с загрузкой данного проекта.
- project1.lps (Конфигурация проекта в виде xml-кода)
- project1.res (Файл ресурсов, используемых в проекте)
- unit1.lfm (Файл формы модуля. В нем в текстовом виде отображены настройки всех компонентов, используемых в модуле. Редактировать этот файл в ручную настоятельно не рекомендуется, для редактирования этих данных нужно использовать Редактор форм).
- unit1.pas (Исходный код модуля на языке Object Pascal).

Файлы с именем project1 - это файлы всего проекта в целом, файлы с именем unit1 - это файлы модуля.

Модуль> - это отдельная единица исходного кода, выполненная в виде файла с расширением *.pas. Совокупность таких единиц составляет программу.

Когда мы создаем окно, то для него создается два файла: модуль - файл *.pas с исходным кодом, и файл *.lfm, в котором содержатся настройки используемых на форме компонентов. Текст модуля мы можем видеть в Редакторе кода. Однако модуль не всегда имеет окно, это может быть и просто текстовый файл с исходным кодом. О модулях и их разделах мы поговорим подробней в одной из следующих лекций. В нашем проекте всего один модуль, но вообще их может быть сколько угодно. И каждый модуль будет представлен этой парой файлов.

Кроме того, в папке проекта находится папка lib, в которой располагаются подключаемые к проекту данные и информация о

компиляции. Если же вы изменяли проект, и сохраняли эти изменения, то появится также папка backup, в которой будут храниться резервные копии старых вариантов проекта.

Нередко программист добавляет в проект и свои типы файлов. Например, в проекте можно использовать базу данных, какой-нибудь текстовый файл или ini-файл для сохранения пользовательских настроек. Разумно располагать эти файлы также в папке с проектом.

Теперь пару советов по поводу наименования проекта и модулей. Проект следует называть так, как мы хотим, чтобы называлась наша программа. Например, проекту из первой лекции было бы уместней дать имя "Hello" вместо нейтрального "project1".

Модули же нужно называть, исходя из их значения. Всегда в проекте есть главный модуль. В наших проектах пока что было по одному окну. Модуль, созданный для этого окна, и будет главным. В учебной литературе есть множество рекомендаций, как обозначать модули, остановимся на одной из них. Давайте договоримся в будущем главный модуль называть Main (англ. main - главный), а другим модулям давать смысловые названия, например, Options, Editor и т.п. Форму этого модуля (точнее, свойство `Name` формы) будем называть также, но с приставкой f-, обозначающей форму. То есть, `fMain`, `fOptions`, `fEditor` и так далее. Закрепим этот материал на практике.

Откройте Lazarus, если он у вас закрыт, или закройте старый проект и начните новый. В данный момент в Редакторе форм у нас пустая форма, в заголовке которой написано Form1 - это имя формы по умолчанию. Мы договорились называть главный модуль Main, а его форме добавлять приставку f-. В Инспекторе объектов найдите свойство `Name`, и вместо Form1 впишите туда `fMain`. Как только вы нажмете <Enter>, заголовок формы изменится на новый. Теперь сохраним проект и модуль главной формы. Нажмите кнопку "Сохранить все" на Панели инструментов, или выберите команду Главного меню "Файл -> Сохранить все".

В запросе вместо имени проекта project1 укажите новое имя Hello, не забывайте, что мы договорились сохранять проекты в папки с именем по номеру лекции, и номеру проекта в ней. В нашем примере это будет

Как только вы нажмете кнопку "Сохранить", выйдет запрос на сохранение главного модуля. Форму мы назвали fMain, значит, модулю дадим название просто Main. В Lazarus строчные и заглавные буквы не различаются, однако для удобочитаемости кода лучше приучиться использовать заглавные буквы, чтобы выделять названия. Например, FileEdit, SaveAll и т.п.

В свойстве `Caption` формы впишем слово "Приветствие" (разумеется, без кавычек), это будет более понятным заголовком для окна. Не забывайте после ввода новых значений свойств в Инспекторе объектов нажимать <Enter>, чтобы изменения вступили в силу. Теперь установим на форму компонент `TLabel` (метку), который позволит выводить на форме текст. Компонент находится на вкладке Standard:

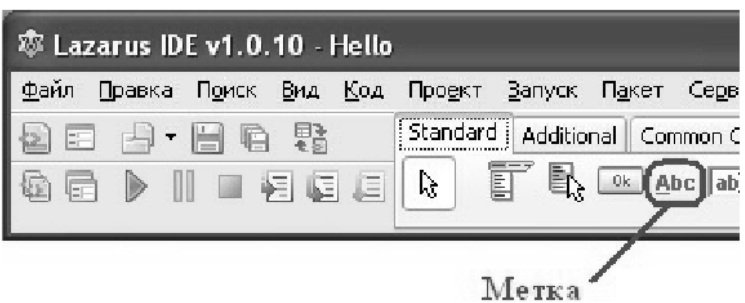


Рис. 2.4. Метка TLabel

Подсказка: если подвести указатель мыши к компоненту и какое то время не нажимать кнопку, выйдет всплывающая подсказка с именем компонента.

Щелкните мышкой по метке, затем по форме, в верхней части окна. Поскольку метка у нас одна, то можно оставить ей имя (свойство `Name`) по умолчанию - `Label1`. А вот в свойстве `Caption` метки напишите:

Как вас зовут?

Ниже метки поместите компонент `TEdit` - редактируемое текстовое поле, в котором пользователь сможет написать что-то:

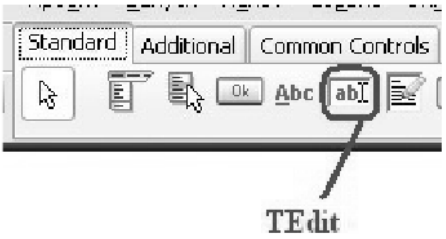


Рис. 2.5. Текстовое поле TEdit

У этого компонента свойство `Name` также оставим по умолчанию - `Edit1`. Как вы можете заметить, у этого компонента свойства `Caption` нет, зато появилось свойство `Text` - именно тут и содержится текст, отображенный в поле. По умолчанию, он совпадает с именем компонента. Просто очистим это свойство, удалив из него старый текст (не забывайте про <Enter>).

Еще ниже установим кнопку TButton. Оставим ее имя по умолчанию, а в свойстве `Caption` напишем

Выполнить

Изменим положение и размеры компонентов и самой формы так, чтобы форма приняла примерно такой вид:

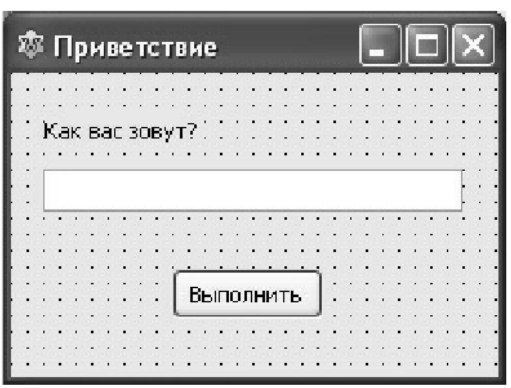


Рис. 2.6. Окончательный вид главной формы

Теперь запрограммируем нажатие кнопки. Щелкните по ней дважды, чтобы сгенерировалось событие, и автоматически открылся Редактор кода. В месте, где мигает курсор, впишем следующий код:

```
ShowMessage('Привет, ' + Edit1.Text + '!');
```

Редактор кода должен выглядеть так:

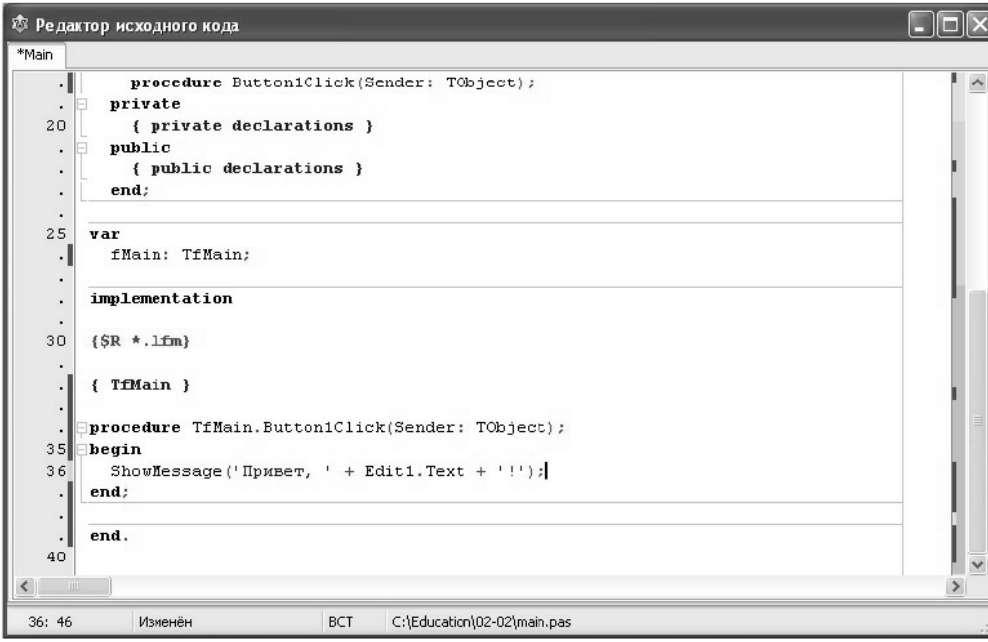


Рис. 2.7. Введенный код

Сохраните проект и запустите его. Когда программа загрузится, впишите в окне Edit1 свое имя и нажмите кнопку "Выполнить". Вы должны получить примерно такой результат:

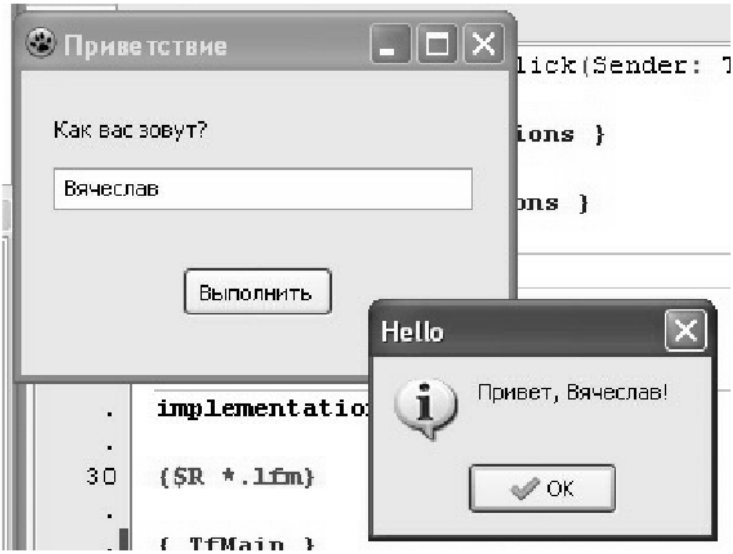


Рис. 2.8. Программа Hello.exe в действии

Вы можете сколько угодно раз менять имя в текстовом поле и снова нажимать кнопку "Выполнить", получая приветствие уже с другим текстом. У нас получилась настоящая интерактивная программа, то есть, программа, взаимодействующая с пользователем.

В дальнейших лекциях мы не будем так подробно останавливаться на том, как сохранять проект или модуль, ограничиваясь кратким

Сохраните проект под именем …

Как и куда сохранять проект, вы уже должны знать.

Работа с компонентами

Эта лекция посвящена работе с компонентами. Подробно рассмотрены свойства таких компонентов, как TForm, TPanel, TButton, TBitBtn и TSpeedButton. Рассмотрена возможность изменения размеров панелей с помощью границы-разделителя TSplitter.

Цель лекции

Изучение изменения свойств компонентов и работы с событиями. Знакомство с общими для многих компонентов свойствами, с различными типами кнопок.

Форма

Мы много раз употребляли слово компонент, но что это такое?

Компонент - это некий виртуальный объект, который обладает своими свойствами и событиями, находится на Палитре компонентов, и который можно установить на форму.

Форма не находится в Палитре компонентов, однако, это тоже компонент! Ведь и у формы есть свойства и события, а мы имеем возможность менять первые и программировать вторые. Познакомимся с основными свойствами формы. Откройте Lazarus и новое приложение в нем. Поскольку ничего, кроме формы, в приложении пока нет, в Инспекторе объектов отображаются свойства формы. Причем все свойства нам не видны, придется прокручивать Инспектор объектов, чтобы рассмотреть остальные свойства или найти нужное.

Свойство `Name` нам уже знакомо - это имя компонента, в нашем случае, формы (англ. name - имя). По этому имени мы можем программно обращаться к его различным свойствам. Мы договорились, что главную форму мы будем называть fMain. Другими словами, свойству `Name` главной формы мы присвоим значение `fMain`. Так и сделаем. Как только мы переименовали форму, автоматически изменился и ее заголовок.

Свойство `Caption` - заголовок формы (англ. caption - заголовок). В это свойство впишем новый текст:

Моя форма

Как только мы впишем новое значение и нажмем <Enter>, текст заголовка изменится.

Поскольку нам придется сегодня много экспериментировать, будет нелишним сохранить новый проект. Сохраните его под именем Proba в папку

C:\Education\03-01

Вы еще не забыли, что модуль мы называем так же, как форму, но без начального "f"? В нашем случае, это будет Main.

Теперь давайте прогуляемся по основным свойствам формы, от начала к концу.

`AlphaBlend` - прозрачность. Это свойство не назовешь основным, однако в некоторых случаях оно может придать нашему приложению дополнительное изящество. Это логическое свойство, в нем мы можем выбрать либо `False` (англ. false - ложь) - значение по умолчанию, либо `True` (англ. true - истина). Другие значения в это свойство мы вписать не сможем. Если в этом свойстве установлено `False`, то форме запрещено быть прозрачной, если же `True`,- то разрешено. Давайте для пробы установим `True`.

Совет. Поменять `False` на `True`, или наоборот, можно либо выбрав нужное значение, либо дважды щелкнув по изменяемому значению.

`AlphaBlendValue` - значение прозрачности. Просто установить `AlphaBlend` в `True` недостаточно, чтобы сделать форму прозрачной. Нужно еще указать значение прозрачности в `AlphaBlendValue`. Это значение может изменяться от нуля (полная прозрачность) до 255 (полная непрозрачность). Другие значения вы просто не сможете установить. Имейте в виду, что если вы сделаете форму полностью прозрачной, то увидеть ее в рабочей программе не сможете. Закрывать

придется либо с помощью кнопки программы в строке состояния (справа от кнопки "Пуск"), либо с помощью Диспетчера задач Windows, либо (если вы скомпилировали и загрузили программу командой "Запустить" или аналогичной кнопкой на Панели инструментов) выполнив команду Lazarus "Запуск -> Сбросить отладчик". Но лучше до этого не доводить. Установите значение в 125 (полупрозрачность), нажмите кнопку "Запустить" и посмотрите, что получилось:

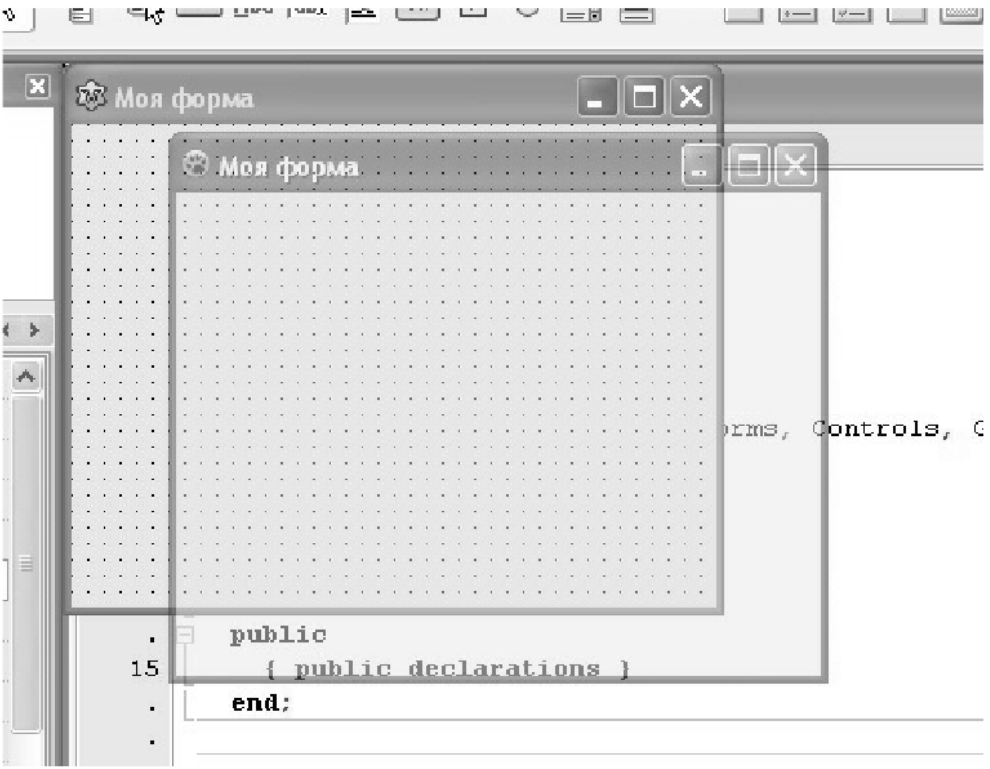


Рис. 3.1. Полупрозрачное окно

Верните в свойство `AlphaBlend` значение `False`, так как для дальнейших экспериментов прозрачность окна будет нам мешать.

`AutoScroll` - автоматическая прокрутка формы. Также логическое свойство. Если установлено в `True`, то прокрутка окна формы разрешена. При этом, если какие то компоненты на форме стали невидны, например, при уменьшении размера формы, то в правой части формы появляется полоса прокрутки, позволяя прокручивать

форму. Оставим здесь значение по умолчанию `False` - прокрутка запрещена.

`AutoSize` - автоматический размер. Если установлено в `True`, то форма автоматически будет принимать размер, достаточный только для установленных компонентов. Имейте в виду, что увидеть этот эффект вы сможете только в работающем приложении, не в форме. Если, например, установить на форму кнопку, и запустить приложение, то окно сожмется по размеру кнопки. Не очень предсказуемый эффект, поэтому данное свойство используется редко. Можете поэкспериментировать с этим свойством, но в конце установите в нем значение по умолчанию `False`.

`BorderStyle` - стиль обрамления формы. Очень интересное свойство, сильно влияющее на вид окна. Может принимать следующие значения:

- `bsDialog` - Диалоговое окно с невозможностью изменять размеры.
- `bsNone` - Окно без строки заголовка. Невозможно изменить размеры, невозможно перемещать по экрану.
- `bsSingle` - Окно, размеры которого нельзя менять, но которое можно свернуть/развернуть с помощью кнопок строки заголовка.
- `bsSizeable` - Обычное окно Windows.
- `bsSizeToolWin` - Строка заголовка более узкая, кнопки свернуть/развернуть отсутствуют, но можно менять размеры окна мышью, ухватившись за край или угол окна.
- `bsToolWindow` - Как bsSizeToolWin, но нельзя мышью менять размеры окна.

Установите значение `bsDialog` и запустите программу, посмотрите на внешний вид полученного окна.

`Font` - шрифт формы и всех компонентов на ней. Когда вы щелкаете по свойству, в его правой части появляется кнопка с тремя точками:

Рис. 3.2. Раскрывающее свойство Font

Если щелкнуть по этой кнопке, откроется диалог выбора шрифта. Если вы выберите, например, шрифт Times New Roman, обычный, размером 11 пикселей, то это скажется на всех компонентах, расположенных на форме. Они все получат этот шрифт "в наследство", хотя для отдельных компонентов затем можно будет выбрать и другой шрифт.

`Height` - Высота формы. Размер в пикселях от строки заголовка до нижней границы формы. Когда вы мышью меняете размеры формы, автоматически меняется и это значение. Изменить высоту окна можно и вручную, указав в этом свойстве нужный размер. Не работает, если окно показывается развернутым.

`Left` - Левая граница формы. Расстояние в пикселях от левой границы рабочего стола до левой границы формы. Не работает, если установлено положение формы по центру рабочего стола (экрана), или окно развернуто.

`Position` - Положение формы. Может принимать следующие значения:

- `poDefault` - положение окна определяет Windows.
- `poDefaultPosOnly` - размер окна соответствует проекту, а его положение определяет Windows.
- `poDefaultSizeOnly` - положение окна соответствует проекту, а его размер определяет Windows.
- `poDesigned` - значение по умолчанию, и размер, и положение определяет разработчик.
- `poDesktopCenter` - окно будет располагаться по центру рабочего стола.
- `poMainFormCenter` - данное значение применяют для дополнительных (неглавных) форм проекта. Окно будет располагаться по центру главного окна.

- `poOwnerFormCenter` - это значение также используется для дополнительных окон, располагая его по центру родительского окна, то есть, окна, из которого произошел вызов данного окна.
- `poScreenCenter` - окно располагается по центру экрана.

Совет: главное окно располагайте по центру рабочего стола, а остальные окна проекта - по центру главной формы. Такие настройки будут предпочтительней для подавляющего большинства проектов.

`ShowHint` - показ подсказок. Если вы подведете указатель мыши к какому-нибудь компоненту в Палитре компонентов (или к кнопке на Панели инструментов), но не будете нажимать кнопку, то через некоторое время появится всплывающая подсказка (хинт, от англ. hint - намёк, намекать) с именем компонента (кнопки). Свойство логическое, если установлено в `True`, то показ таких подсказок разрешен для всех установленных на форме компонентов, либо для самой формы.

`Hint` - текст самой подсказки. Попробуйте разрешить показ подсказок, а в этом свойстве напишите какой-нибудь текст, например, "Наша всплывающая подсказка". Запустите программу, наведите на форму указатель мыши, но не нажимайте на нее:

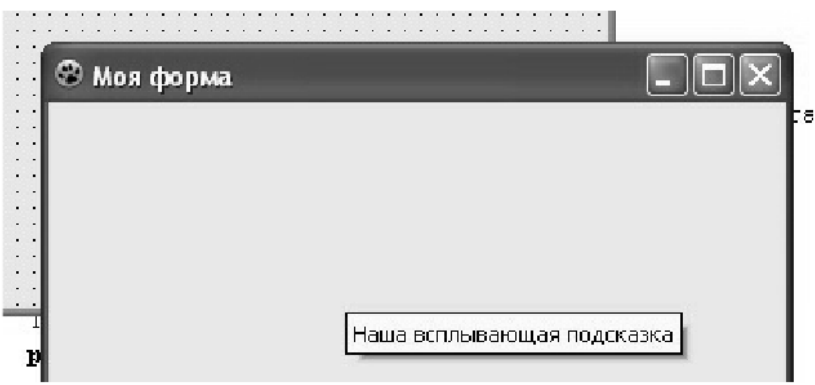


Рис. 3.3. Всплывающая подсказка

`Top` - Верхняя граница формы. Расстояние в пикселях от верхней границы рабочего стола до строки заголовка формы. Не работает, если установлено положение формы по центру рабочего стола (экрана), или окно развернуто.

`Width` - Ширина формы. Размер в пикселях от левой до правой границы формы. Когда вы мышью меняете размеры формы, автоматически меняется и это значение. Не работает, если окно показывается развернутым.

`WindowState` - Состояние окна. Может принимать следующие значения:

- `wsFullScreen` - во весь экран.
- `wsMaximized` - развернуто.
- `wsMinimized` - свернуто.
- `wsNormal` - обычное состояние. Это значение по умолчанию, чаще всего используется именно оно.

Для дальнейших экспериментов давайте установим следующие параметры формы:

```
BorderStyile = bsSizeable
Height = 400
Position = poDesktopCenter
Width = 600
WindowState = wsNormal
```

Примечание. Мы рассмотрели далеко не все свойства формы, и совсем не затрагивали ее события. Одни из них мы и вовсе не будем рассматривать, так как в практическом программировании они обычно не используются, другие мы рассмотрим в следующих лекциях.
Примечание. Мы рассматривали свойства достаточно подробно. Но дело в том, что многие компоненты имеют схожие свойства. Так, у всех компонентов есть свойство `Name`, у многих их них есть такие свойства, как `Caption, Left, Top, Height, Width` и т.п. Изучив эти свойства у одних компонентов, мы с легкостью сможем ими пользоваться и в других.

Панель и TSplitter

Панель - это своего рода контейнер, для размещения на ней связанных

по какому-то признаку компонентов. Размещенные на панели компоненты "привязываются" к ней. При перемещении панели по форме, будут перемещаться и эти компоненты. Панель нередко используют и для украшения формы. На форме может быть несколько панелей.

После изучения предыдущего раздела, у вас должен быть открыт проект с пустой формой, настроенной определенным образом. Если это так, установим на форму панель. Для этого щелкните по компоненту TPanel в палитре компонентов, а затем щелкните по форме, установив на нее панель:

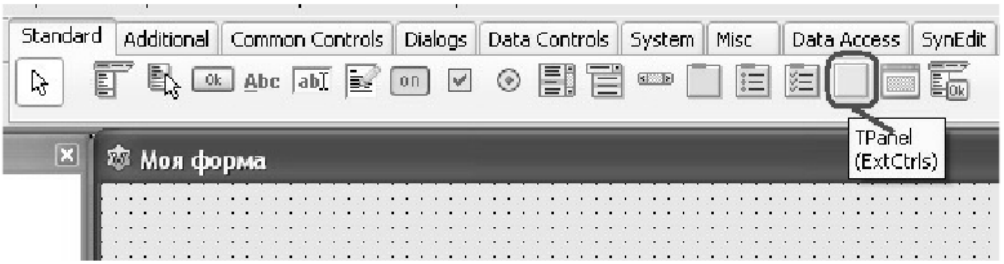


Рис. 3.4. Компонент TPanel

У вас на форме появится некое прямоугольное образование, в центре которого будет написано Panel1 - имя компонента, которое Lazarus присвоил по умолчанию. Если мы будем устанавливать другие панели, Lazarus назовет их Panel2, Panel3 и т.д. Если на форме у нас будет только одна панель, то цифру в названии (свойство `Name`) и вовсе можно будет убрать. В данном случае у нас будет несколько панелей, так что оставим панели имя, которое ей присвоил Lazarus по умолчанию. Рассмотрим некоторые свойства панелей, которыми нам часто придется пользоваться (компонент `Panel1` при этом должен быть выделен).

`Name`	- имя панели. Если будем ее переименовывать, то в этом свойстве.
`Caption`	- текст в центре панели, в нашем случае, это Panel1. Этот текст очень редко используется, обычно его удаляют, чтобы панель была пустой. Мы тоже очистим этот текст.
`Left`,	- расстояние в пикселях, соответственно, от левой и верхней границ формы до панели. Установим в обоих

свойствах значение 10.

`Height` - высота панели в пикселях, установим ее в 320.

`Width` - ширина панели в пикселях, установим ее в 580.

Теперь сохраните изменения в проекте и запустите его. Вы увидите окно с выпуклой панелью, под которой осталось место для кнопки. Так обычно оформляют диалоговые окна. Однако выпуклость панели можно и убрать, превратив ее в аккуратную рамочку. Закройте окно и вернитесь к проекту. Рассмотрим три свойства, влияющих на выпуклость панели.

`BevelInner` - определяет стиль внутренней границы панели. Может иметь значения:

- `bvNone` - никакого стиля. (по умолчанию для `BevelInner`)
- `bvSpace` - пустая отрисовка границы, визуально не отличается от `bvNone`.
- `bvRaised` - приподнятая граница. (по умолчанию для `BevelOuter`)
- `bvLowered` - утопленная граница.

`BevelOuter` - определяет стиль внешней границы панели. Может иметь такие же значения, как и `BevelInner`.

`BevelWidth` - ширина окантовки в пикселях. По умолчанию, имеет значение 1, что обычно вполне достаточно. Чтобы еще больше подчеркнуть границу, можно установить значение 2, однако дальнейшее увеличение ширины не сделает панель красивей, скорее, наоборот. Впрочем, это зависит от вашего вкуса и конкретного дизайна.

Совет: чтобы панель сделать вогнутой, свойство `BevelOuter` установите в `bvRaised`, а чтобы сделать красивую окантовку, сделайте `BevelInner = bvLowered`, а `BevelOuter = bvRaised`. Для ширины окантовки вполне хватит 1 пикселя.

Попробуйте "поиграть" со свойствами панели: меняйте их, запускайте проект на выполнение, и смотрите, как будет меняться внешний вид панели. А когда наиграетесь и запустите программу в последний раз,

панели. А когда наиграетесь и запустите программу в последний раз, разверните окно во весь экран. Вот это уже неприятно: окно программы увеличилось, а панель осталась прежней:

Рис. 3.5. Искажение дизайна окна при изменении размеров

Дело в том, что панель "привязывается" к верхней и левой границе формы. А при изменении размеров окна, правая и нижняя границы панели не изменяются. Чтобы "привязать" к форме все стороны панели, служат "якоря":

`Anchors` (англ. якоря) - свойство, разрешающее или запрещающее привязку сторон к внешнему контейнеру. Если панель располагается прямо на форме, то и привязывается она к форме. А если панель располагается на другой панели, то привязывается уже к ней. Подобное свойство имеет большинство визуальных компонентов. `Anchors` - раскрывающееся свойство. Если щелкнуть по кнопке "+" слева от него, то свойство раскроется, показывая привязки сторон, а кнопка "+" превратится в "-":

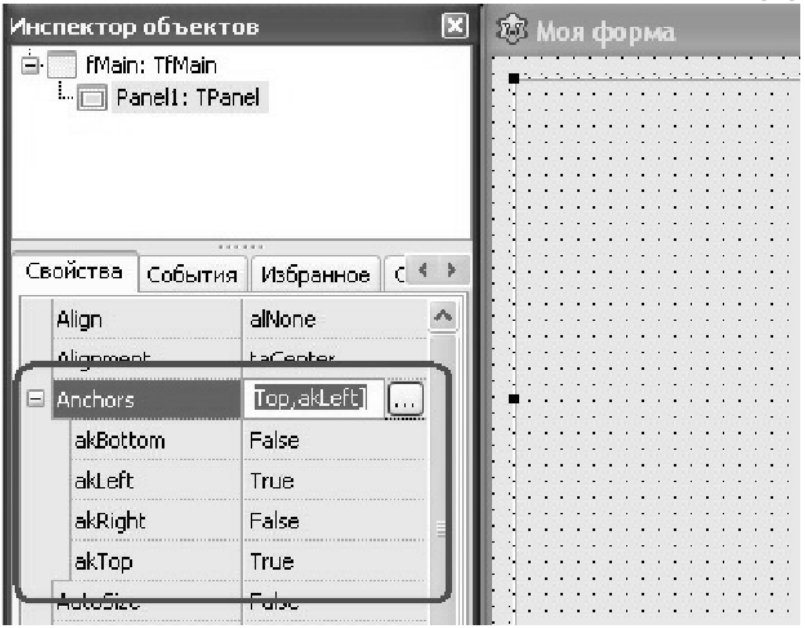


Рис. 3.6. Раскрытое свойство Anchors

Как мы видим, по умолчанию "привязаны" только левая и верхняя границы (`akLeft=True, akTop=True`). Чтобы разрешить привязку правой (`akRight`) и нижней (`akBottom`) границ панели, их тоже нужно установить в `True`. Сделайте это, запустите программу на выполнение и попробуйте изменять размеры окна. Теперь вместе с окном изменяет размеры и панель, что нам и требовалось.

Однако есть и другой способ привязки компонентов к различным сторонам формы или контейнера, на котором компонент находится. Поскольку этот способ применяется довольно часто, разберем его подробней. Для начала удалите панель - выделите ее на форме и нажмите <Delete>. Затем установите на форму новую панель. Таким образом, мы просто сбросили все изменения в настройках панели, ведь новая панель имеет все значения по умолчанию. А имя у нее такое же - Panel1. Очистим свойство `Caption`, и займемся выравниванием.

`Align` (англ. выравнивание) - свойство, позволяющее "прилепить" панель к одной из сторон внешнего контейнера, или ко всему контейнеру. Может иметь следующие значения:

- `alBottom` - Панель занимает весь низ контейнера (в нашем случае контейнером является форма).
- `alClient` - Панель занимает весь контейнер.
- `alCustom` - Пользовательское выравнивание, то же самое, что `alNone`.
- `alLeft` - Панель занимает всю левую часть контейнера.
- `alNone` - Нет выравнивания. Это значение по умолчанию.
- `alRight` - Панель занимает всю правую часть контейнера.
- `alTop` - Панель занимает весь верх контейнера.

Если мы установим это свойство в `alClient`, то наша панель займет всю форму. При изменении размеров формы будут изменяться и размеры панели. Если установим, к примеру, в `alTop`, то панель займет всю верхнюю часть формы. Если затем на этой панели установим кнопки, то получим Панель инструментов. А совместно с другим компонентом, `TSplitter`, который находится на вкладке Additional Палитры компонентов и представляет собой перемещаемую границу-разделитель, мы получим различные части окна, размеры которого пользователь сможет менять мышью во время выполнения программы:

Рис. 3.7. Разделитель TSplitter

Давайте сделаем так. У нашей `Panel1` установим `Align = alTop`. Затем добавим на форму разделитель `TSplitter`, и также установим у него `Align = alTop`.

Теперь добавим на свободное место формы еще одну панель. Очистим у нее свойство `Caption`, и установим `Align = alLeft`. Добавим еще один `TSplitter`, убедимся, что у него также `Align = alLeft`. Теперь весь верх и всю левую часть формы занимают панели, отгораживаясь от свободной части сплиттерами. На эту свободную часть формы мы установим третью панель. Очистим у нее свойство

`Caption` и установим `Align = alClient`. Панель заняла всю оставшуюся часть формы. Теперь запустите проект на выполнение и убедитесь, что мышью можно перемещать границы-разделители, меняя размеры панелей. При изменении размеров окна изменяются и размеры панелей:

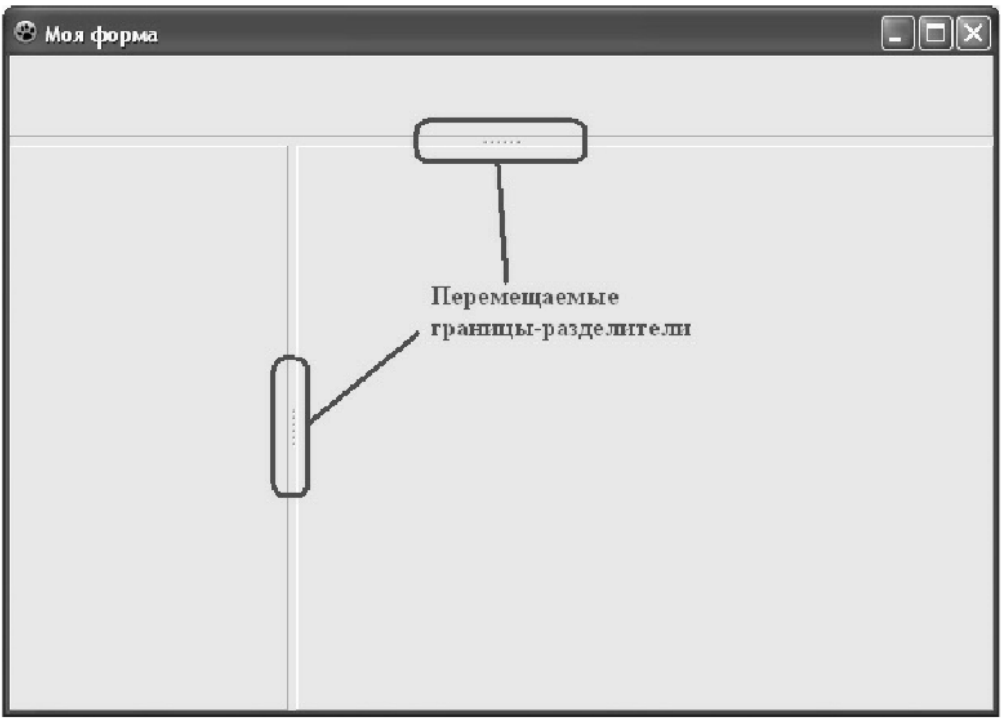


Рис. 3.8. Три панели и два разделителя на форме

Не забудьте сохранять проект время от времени.

Мы упустили еще одно полезное свойство панели - автоматическую настройку размера.

`AutoSize` - автоматическая настройка размера, логическое свойство. По умолчанию, установлено значение `False` - автонастройка запрещена. Если установить свойство в `True`, то автоматическая настройка будет разрешена.

Установите на верхнюю панель кнопку TButton. Свойства `Left` и `Top` кнопки установите в 1. Теперь выделите панель, на которой находится

кнопка, и переведите свойство `AutoSize` в `True`. Верхняя панель по-прежнему занимает всю верхнюю часть формы, но теперь ее высота уменьшилась по высоте кнопки! Таким способом можно создавать простые панели инструментов для приложения. Правда, если вы запустите программу на выполнение, то увидите, что теперь верхним разделителем невозможно менять высоту верхней панели, ведь она у нас автоматически принимает размер под расположенную на ней кнопку. Так что и надобность в первом сплиттере в этом случае отпадает.

Кнопка TButton

Кнопки, пожалуй, самый используемый компонент в программах Windows. Они повсюду: в диалоговых окнах, в панелях инструментов, в сообщениях. Познакомимся поближе с этими элементами.

Итак, простая кнопка TButton. Находится на вкладке Standard Палитры компонентов. Мы уже использовали несколько раз этот компонент, и сейчас в проекте одна кнопка находится на верхней панели. Выделите кнопку, и посмотрим на ее свойства. С большинством из них вы уже знакомы, познакомимся с некоторыми другими.

`Cancel` - логическое свойство, по умолчанию равно `False`. Если равно `True`, то нажатие на кнопку <Esc> будет равносильно нажатию на эту кнопку.

`Default` - логическое свойство, по умолчанию равно `False`. Если равно `True`, то нажатие на кнопку <Enter> будет равносильно нажатию на эту кнопку. Однако если фокус находится на какой-нибудь другой кнопке, то нажатие на <Enter> нажмет ее.

`ModalResult` - полезное в некоторых случаях свойство, особенно в диалоговых окнах. Может иметь значения:

- `mrNone` (по умолчанию) - нет результата
- `mrOk` - OK
- `mrCancel` - Отмена
- `mrAbort` - Прервать

- `mrRetry` - Повторить
- `mrIgnore` - Игнорировать
- `mrYes` - Да
- `mrNo` - Нет
- `mrAll` - Все
- `mrNoToAll` - Нет для всех
- `mrYesToAll` - Да для всех
- `mrClose` - Закрыть

Любая форма также имеет свойство `ModalResult`. Если у кнопки в этом свойстве присвоить одно из этих значений, и этой кнопкой закрыть форму, то и у формы свойство `ModalResult` станет таким, как у кнопки. Предположим, у вас имеется диалоговое окно с тремя кнопками: "Да", "Нет" и "Отмена". И каждой из этих кнопок в свойстве `ModalResult` вы присвоили соответственное значение. Если каждая из этих кнопок будет закрывать окно, то при его закрытии можно посмотреть свойство `ModalResult` формы и определить, какой кнопкой пользователь закрыл окно. И в соответствии с результатом, выполнять дальнейшие действия.

Примечание. Просто присваивание свойству `ModalResult` кнопки какого то значения не закроет окно, это действие нужно программировать отдельно.

`Hint` - текст всплывающей подсказки. Эта подсказка всплывет, когда пользователь подведет указатель мыши к кнопке. Но подсказка появится, только если `ShowHint = True`.

Теперь будем разбираться с событиями. Однако прежде измените свойство `Caption` кнопки, и впишите туда слово `Выход`. Свойство `Name` оставьте по умолчанию.

Двойной щелчок по кнопке приведет к тому, что Lazarus сгенерирует событие - нажатие на эту кнопку. Сразу же Редактор кода окажется наверху, и вы увидите мигающий курсор там, где нужно запрограммировать это событие. Код, который мы туда впишем, будет выполняться всякий раз, когда пользователь нажмет на эту кнопку. Впишем туда короткое

```
  Close;
```

Именно так, с точкой с запятой в конце. Это оператор закрытия окна. А если это окно главное, то значит, закроется вся программа:

```
 .
35  { TfMain }
 .
 .  procedure TfMain.Button1Click(Sender: TObject);
 .  begin
39    Close;
40  end;
 .
 .  end.
43
```

Рис. 3.9. Программирование события нажатия на кнопку

С процедурами мы будем разбираться позже, пока обратите внимание на имя процедуры: `Button1Click`. `Button1` - это имя нашей кнопки, а событие, которое мы программировали, называется `OnClick` - событие нажатия на кнопку. В Инспекторе объектов перейдите на вкладку События, и найдите событие `OnClick`. Как видите, там уже прописана именно эта процедура:

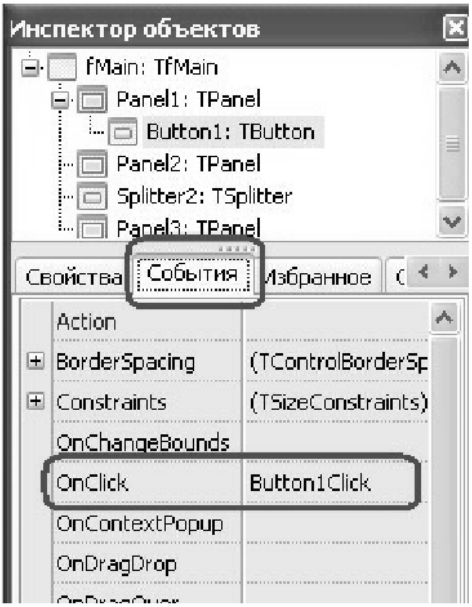


Рис. 3.10. Событие OnClick кнопки

Если мы выделим это событие, то справа от него появится кнопка "…", нажатие на которую приведет к генерированию этого события. Поскольку у нас оно уже запрограммировано, то просто откроется эта процедура и внутрь нее установится курсор. Вот вам второй способ генерировать нужное событие, выбирайте, что вам удобней. Нам не раз придется переключаться в Инспекторе объектов между вкладками Свойства и События.

Другие события кнопок нам пока не нужны. Сохраните проект и запустите его. Убедитесь, что нажатие на кнопку приводит к закрытию программы.

Кнопкой <F12> верните на передний план Редактор форм, а в Инспекторе объектов верните вкладку Свойства. В Палитре компонентов откройте вкладку Additional. Первые два компонента в этой вкладке - это кнопки TBitBtn и TSpeedButton:

Рис. 3.11. Кнопки TBitBtn и TSpeedBtn

Поставьте две кнопки TBitBtn и одну TSpeedButton куда-нибудь на нашу форму, например, на верхнюю панель, правее нашей Button1. Будем разбираться, чем они отличаются от обычной кнопки.

Кнопка TBitBtn

Это более продвинутая кнопка, имеющая некоторые дополнительные свойства. Прежде всего, вы заметили, что она несколько более высокая, чем обычная кнопка. TBitBtn помимо текста может содержать и изображение. Для этого используются несколько свойств.

`Glyph` - Изображение. С помощью этого свойства можно найти и загрузить изображение, которое будет отображаться на кнопке.

Примечание: Большинство изображений хранится в папке Images там, куда вы установили Lazarus. Если вы оставили адрес по умолчанию, это будет

C:\lazarus\images\

Там хранится несколько папок, где изображения рассортированы по тематикам. Однако необязательно использовать только эти изображения. В Интернете можно найти огромное количество бесплатных коллекций изображений, которые можно использовать в своих проектах.

Выделите свойство `Glyph` кнопки BitBtn1 и щелкните по кнопке "..." справа от него. Откроется диалог выбора изображения. В этом диалоге

нажмите кнопку "Загрузить" и найдите папку

C:\lazarus\images\menu

(если вы указывали другой путь при установке Lazarus, то укажите его). Откроется список с картинками, которые хранятся в этой папке. При выборе она будет отображаться в правой части окна. Найдите картинку menu_exit и нажмите кнопку "Открыть", затем "OK". Выбранное изображение окажется на кнопке, слева от надписи. Кстати, чтобы уж придать кнопке соответствующий вид, измените свойство `Caption` на `Выход`. А заодно, сгенерируйте событие нажатия на эту кнопку, и впишите код закрытия, как это делали для простой кнопки. Сохраните проект и запустите его; убедитесь, что нажатие на эту кнопку также закрывает программу.

Однако с этой кнопкой мы еще не закончили. Выделите ее в Редакторе форм и обратите внимание на свойства кнопки.

`GlyphShowMode` - политика показа или сокрытия изображения. Может быть:

- `gsmAlways` - всегда показывать изображение.
- `gsmNever` - никогда не показывать изображение.
- `gsmApplication` - изображение будет показано во время выполнения программы. Значение по умолчанию для этого свойства.
- `gsmSystem` - будет показано изображение или нет, зависит от текущих настроек системы.

Данное свойство лучше не изменять.

`Layout` - место расположения изображения. Может быть:

- `blGlyphBottom` - изображение ниже текста
- `blGlyphLeft` - изображение слева от текста (значение по умолчанию)
- `blGlyphRight` - изображение справа от текста
- `blGlyphTop` - изображение над текстом

Данные настройки целиком зависят от конкретного стиля интерфейса, который вы выбрали для вашей программы. Но в большинстве случаев, значение по умолчанию - то, что нужно.

`Spacing`	- расстояние в пикселях от изображения до текста. Это свойство меняют также редко.
`TabOrder`	- порядковый номер выделения компонентов. Имеется у многих визуальных компонентов. Определяет, в каком порядке компоненты будут получать фокус ввода при нажатии клавиши <Tab>. Нумерация начинается с 0, установить одинаковый номер для двух компонентов невозможно, если они находятся на одном контейнере.
`Enabled`	- доступность компонента. Также присутствует у многих визуальных компонентов. Логическое свойство. Если `True`, то компонент доступен (кнопку можно нажимать), если `False`, то недоступен (кнопка серого цвета, нажать на нее невозможно). Измените это свойство, запустите программу и посмотрите, как изменилась кнопка. Затем верните значение по умолчанию.
`Visible`	- видимость компонента. Если `True`, то компонент видимый, иначе компонент спрятан. "Поиграйте" и с этим свойством, не забудьте вернуть его в конце в значение по умолчанию.

Мы не зря установили на форму две кнопки TBitBtn. Для изучения следующего свойства выделите `BitBtn2`.

`Kind` - разновидность кнопки. При выборе какого-то значения на кнопке появляется соответствующие текст и картинка, а кнопка сразу обладает нужными свойствами, зависящими от ОС - закрывает окно (в случае если `Kind = bkClose`), присваивает форме нужное значение `ModalResult`. Правда, в работающей программе (по крайней мере, для Windows XP Professional SP3) текст на кнопке будет на английском языке. Может быть:

- bkAbort - Прервать
- bkAll - Все

- bkCancel - Отмена
- bkClose - Закрыть
- bkCustom - Пользовательские настройки - текст и надпись устанавливает программист. Это значение по умолчанию.
- bkHelp - Справка
- bkIgnore - Пропустить
- bkNo - Нет
- bkNoToAll - Нет для всех
- bkOK - OK
- bkRetry - Повтор
- bkYes - Да
- bkYesToAll - Да для всех

Установите `Kind = bkClose`, сохраните проект и запустите его. Убедитесь, что теперь и кнопка BitBtn2 закрывает окно. Для этого нам даже не пришлось ее программировать!

Кнопка TSpeedButton

Напоследок поработаем с этим типом кнопок. Они чаще всего применяются для организации панелей инструментов. Дело в том, что эта кнопка не имеет фокуса ввода! Что я имею в виду? Запустите программу на выполнение и клавишей <Tab> перемещайтесь по кнопкам. Вы увидите, что можно выделить любую кнопку, кроме SpeedButton1. На такую кнопку можно нажать только мышью, при этом, если окно не закроется, фокус вернется к тому компоненту, который был выделен ранее.

Как видите, кнопка квадратная, имеет малый размер и рассчитана только на то, чтобы с помощью свойства `Glyph` в нее загрузили изображение. Однако она имеет такие же свойства, как `TBitBtn`: `Caption, Layout, Space` и т.д., поэтому в данной кнопке также можно использовать текст совместно с изображением, или только текст, или только изображение.

Поскольку кнопка не имеет фокуса ввода, свойства `TabOrder` у нее также нет. Отсутствует и свойство `Kind`. Но в остальном она похожа на TBitBtn.

С помощью свойства `Glyph` загрузите картинку menu_exit, как в прошлом примере. Однако программировать событие `OnClick` мы не будем - зачем писать один и тот же код? Можно поступить проще. Перейдите на вкладку "События" Инспектора объектов, и, не нажимая в событии OnClick на кнопку "…", откройте доступный список:

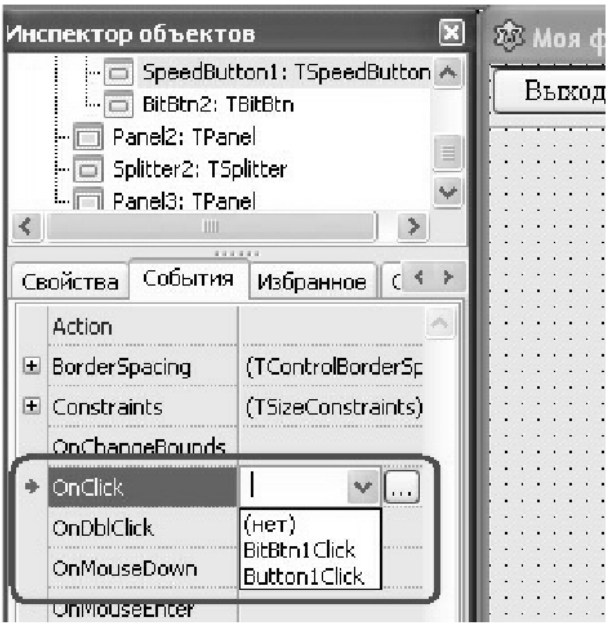


Рис. 3.12. Выбор события OnClick

Выберите в списке `Button1Click`, сохраните проект и запустите его на выполнение. Убедитесь, что наша `SpeedButton1` закрывает программу - это срабатывает событие первой кнопки.

Напоследок исправим один недостаток. Если вы обратили внимание, у нас на форме 4 кнопки, но при установке указателя на любую из них всплывает та самая подсказка, которую мы писали в свойстве `Hint` формы: "Наша всплывающая подсказка". А это не очень хорошо. Нужно в свойстве `Hint` кнопок написать соответствующие тексты, например:

- Кнопка Button1
- Кнопка BitBtn1
- Кнопка BitBtn2
- Кнопка SpeedButton

Конечно, в реальных приложениях в свойстве `Hint` вы будете писать подходящий текст подсказки, например, "Выход из программы". Но у нас все кнопки выполняют выход из программы, и чтобы отличить их друг от друга, мы укажем соответствующий текст. Сохраните проект и запустите его на выполнение; убедитесь, что текст всплывающих подсказок меняется.

Совет. Текст в свойстве `Hint` формы нам, вообще говоря, и не нужен, его можно очистить. Однако чтобы получить доступ к свойствам формы, нам нужно выделить саму форму, а это теперь проблематично, ведь ее полностью перекрывают три панели! Выделить форму можно либо в Дереве объектов в верхней части Инспектора объектов, либо выделив какую-либо панель, и нажав клавишу <Esc>. При этом выделение перейдет на внешний компонент, то есть, на форму.

Основы кода

Эта лекция посвящена основам кодирования на Паскале. Рассмотрены такие понятия, как Типы данных, Переменные, Константы, Комментарии, Значения, Выражения, Ключевые слова и так далее. Для закрепления материала приводятся подробные практические примеры по темам лекции.

Цель лекции

Знакомство с типами данных. Применение переменных, констант, комментариев.

Типы данных

Наверное, вы понимаете, что умение пользоваться компонентами - это еще не программирование? Разработка пользовательского интерфейса - настройка формы, размещение на ней компонентов, организация меню, придание всему этому привлекательного вида и удобства использования - всё это, скорее, дизайн. Программирование - это знание языка, умение писать исходный код. Этим мы сегодня и займемся.

Любые данные, которыми приходится оперировать программисту, имеют свой тип. Паскаль, который лежит в основе многих языков и сред разработки, в том числе и в основе Lazarus, отличается более строгими правилами использования данных различного типа. В частности, в Паскале любую переменную любого типа нужно предварительно объявить - указать имя переменной и ее тип, присвоить начальные значения (во многих других языках переменная объявляется прямо там, где она потребовалась, а в некоторых языках обходятся и вовсе без объявления и указания типа). Всё это приучает программиста к более качественному стилю программирования, делает программу более читаемой и понятной компилятору.

В Паскале существует всего несколько стандартных типов, с которыми может работать программист:

- Целые числа (числа без запятой: 5; -12; 0)
- Вещественные числа (числа с запятой: 5,3; -3,14; 0,0)
- Символы (отдельные буквы или другие символы: "Б", "z", "&", "/")
- Строки (группы символов: "Это строка", "This is a text")
- Логический тип (может принимать значение либо Ложь - `False`, либо Истина - `True`)

Lazarus основан на Object Pascal, который имеет множество дополнительных типов данных. Например, тип Дата-Время, коллекции строк, файлы, записи, классы и так далее. Со многими типами мы будем знакомиться в течении курса, в порядке возрастания сложности материала.

Элементы программы

Программа на языке Object Pascal может содержать следующие стандартные элементы:

- Служебные (зарезервированные) слова
- Идентификаторы
- Типы
- Переменные
- Константы
- Комментарии
- Метки
- Подпрограммы

Служебные (зарезервированные) слова - это такие слова, которые являются частью языка и применяются для обозначения различных инструкций или элементов программы. Примеры служебных слов: `as`, `class`, `if`, `for`, `var`, `array`, `unit`, `interface`, `uses`.

Служебные слова в Редакторе кода всегда выделяются жирным шрифтом. Программист не может использовать служебные слова в качестве идентификаторов: для обозначения переменных, констант, записей, пользовательских констант и т.п.

Идентификаторы - это слова (имена), которыми программист

обозначает любой другой элемент языка, кроме служебных слов, комментариев и собственно, идентификаторов. Например, когда мы даем имя переменной или константе, мы идентифицируем ее.

Правило. Для идентификаторов разрешено использовать только символы латинского алфавита, цифры и знак подчеркивания. Причем первым символом идентификатора не может быть цифра. Строчные и заглавные символы не различаются, то есть `myvar, MyVar` и `MYVAR` - это один и тот же идентификатор.

Примеры правильных идентификаторов:

- `MyVar`
- `_perem`
- `a2`
- `MinimalnayaZarplata`

Примеры неправильных идентификаторов:

- `Зарплата` (не латинские символы)
- `2a` (первый символ - цифра)
- `My perem` (в идентификаторе неразрешенный символ - пробел)

Типы - это специальные конструкции языка, указывающие компилятору, сколько памяти нужно зарезервировать для объявляемого элемента, например, переменной, массива, константы. Типами могут быть стандартные типы Паскаля, расширенные типы Объектного Паскаля или типы данных, объявленные пользователем. Мы будем знакомиться с разными типами данных на протяжении всего курса.

Примеры типов: `integer` (целое число), `real` (число с запятой), `string` (строка) и т.п.

Переменные

Переменные являются элементом языка, который используется повсеместно. Нам неоднократно придется использовать переменные различных типов, различного назначения, различной области

видимости. Поэтому данный раздел стоит рассмотреть подробней.

Все данные, с которыми работает программа, располагаются в ОЗУ (Оперативное Запоминающее Устройство, Оперативная память) компьютера. Когда пользователь загружает программу, то в оперативную память считываются все инструкции программы и данные, которые в ней используются. Данные располагаются в памяти по различным адресам, которые фактически, представляют собой цифры. Так, первый байт памяти располагается под номером 1, второй под номером два и т.д. Однако не все данные занимают только один байт, есть и двухбайтовые, и четырехбайтовые данные, и даже больше. Именно поэтому так важно указывать тип данных - компилятор должен знать, сколько байт в памяти будет занимать ваш элемент, сколько места ему нужно выделить. Допустим, нам нужен некий элемент данных в 2 байта, и компьютер поместит его в ОЗУ по адресу, например, 1000000. Значит, когда потребуется узнать содержимое этого элемента, компьютер должен обратиться к ячейке памяти под номером 1000000, и считать оттуда два байта.

В процессе работы программы эти данные могут меняться. Поэтому нам нужен некий контейнер, ящик определенного размера, где мы будем хранить данные. Эти данные будут меняться, а адрес ящика в памяти останется неизменным, пока работает наша программа. Такой ящик и называется переменной - элементом, чьи данные могут меняться. Итак,

Переменная - это ячейка оперативной памяти, которая занимает соответствующее её типу количество байт. Данные в этой ячейке могут меняться, а адрес размещения в ОЗУ остается неизменным, пока работает программа.

Чтобы программист не мучался с адресацией этих ячеек, в Паскале переменным присваивается имя - идентификатор. Когда мы объявляем переменную с определенным именем, компьютер запоминает это имя и ассоциирует его с реальным адресом в памяти, и количеством занимаемых переменной байт. А когда мы считываем данные переменной `MyPerem`, то компьютер сам находит нужный участок памяти и считывает оттуда нужное количество байт.

Переменные объявляются в разделе `var` (англ. variable - переменная).

Объявление имеет следующий синтаксис (конструкцию):

```
var
  Имя_перем_1: Тип_1;
  Имя_перем_2: Тип_2;
  …
  Имя_перем_n: Тип_n;
```

То есть, в объявлении мы указываем имя создаваемой переменной, затем символ "двоеточие", а затем тип данных этой переменной. Если же нам нужно объявить несколько переменных одного типа, то мы можем перечислить их в одном операторе через запятую:

```
var
  перем_1, перем_2,… перем_n: Тип;
```

Объявление переменных в разных строках - это не правило языка, всё описание можно уместить и на одной строке:

```
var перем_1: Тип_1; перем_2: Тип_2;
```

Но такой код будет сложно читать, он труден для восприятия. Поэтому предыдущие примеры все же предпочтительней, когда каждый оператор описывается на своей строке, программа становится более "читабельной".

Совет. Давайте переменным и другим элементам и конструкциям, которые вы описываете, осмысленные имена, а не просто непонятные наборы символов!

Например, если мы хотим хранить в переменной минимальную заработную плату, то разумно назвать такую переменную `MinZarPlat`, для переменной с вашим именем - `MyName` или `MoeImya` - зависит от вашей фантазии, лишь бы вам самим было понятно, для чего вы вообще создавали эту переменную. Ведь это только поначалу вы будете помнить, что за переменные вы используете. А если вы откроете свой проект через год? Или его откроет ваш друг

или коллега? Что хранится в переменной `a`? Для чего создана переменная `d3`? Такие непонятные идентификаторы затрудняют восприятие программы и категорически не рекомендуются к использованию.

Исключение могут составить переменные, которые обычно создаются в качестве счетчиков для циклов, например, или для каких-то коэффициентов. Такие переменные традиционно называют `i`, `k`, `l` (или др. символами английского языка).

Давайте попробуем использовать переменные на практике. Откроем второй проект из второй лекции, тот, где пользователь вписывает свое имя, а программа выводит ему приветствие. Если вы следовали моим рекомендациям, у вас этот проект должен храниться по адресу:

C:\Education\02-02\

Загрузите файл проекта Hello.lpi - это приведет к загрузке Lazarus и открытию в нем данного проекта. На переднем плане должен оказаться Редактор кода (если это не так, нажмите <F12>, чтобы переключиться с формы на код). Вы можете сразу заметить, что одна переменная у нас уже объявлена:

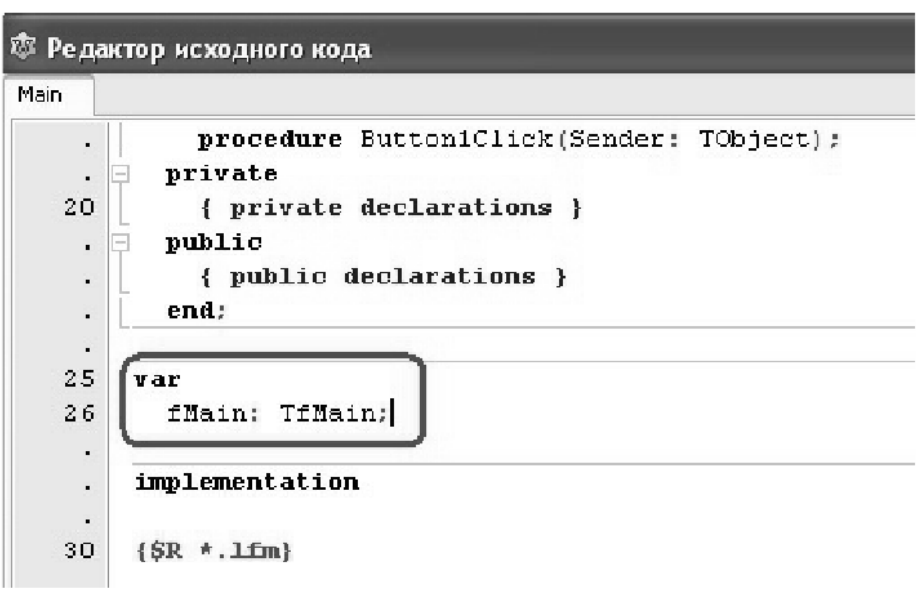


Рис. 4.1. Переменная fMain

Вот ведь любопытно: у нас есть форма fMain, да еще и переменная с таким же именем! Дело в том, что компилятор FPC рассматривает нашу форму, как переменную. Обращение к этой переменной - это обращение к самой форме.

Теперь опустим курсор до обработки события нажатия на кнопку Button1 и немного изменим код этого события:

```
25  var
      fMain: TfMain;

    implementation

30  {$R *.lfm}

    { TfMain }

    procedure TfMain.Button1Click(Sender: TObject);
35  var
      MyName: String;
    begin
      MyName:= 'Привет, ' + Edit1.Text + '!';
      ShowMessage(MyName);
40    MyName:= 'Рады вас видеть, ' + Edit1.Text + '!';
41    ShowMessage(MyName);
    end;

    end.
45
```

Рис. 4.2. Новый код события OnClick кнопки Button1

В дальнейшем, чтобы не загромождать лекции излишними иллюстрациями, я буду указывать исходный код простым текстом с наклонным шрифтом, например, так:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  MyName: String;
begin
  MyName:= 'Привет, ' + Edit1.Text + '!';
  ShowMessage(MyName);
  MyName:= 'Рады вас видеть, ' + Edit1.Text + '!';
```

```
  ShowMessage(MyName);
end;
```

Как видите, здесь мы использовали переменную `MyName` с типом `String` (строка). Мы дважды присваивали ей разные значения, а затем считывали их, чтобы вывести в сообщении. Где именно нужно объявлять переменные, мы поговорим в лекции №8 (Подпрограммы), там же обсудим такое важное понятие, как область видимости переменной. Пока лишь учтите, что при использовании событий, переменные нужно объявлять до служебного слова `begin`, а использовать саму переменную нужно после этого служебного слова.

Оператор присваивания значения

На примере вышеприведенного кода мы могли заметить, что значения переменным присваиваются с помощью оператора присваивания

```
:=
```

То есть, двоеточие и знак "равно", следующий сразу за ним, без пробелов. Присвоение происходит справа-налево: вначале вычисляется значение, указанное ПОСЛЕ этого оператора, затем оно присваивается переменной, указанной ДО оператора присваивания. Например:

```
var
  i: integer;  //объявили переменную с типом integer - целое число
begin
  i:= 3;  //присвоили этой переменной значение 3
```

Значение, которое мы присваиваем переменной, может быть и сложным, составным. Мы можем указать не просто значение, а целое выражение, причем использовать в нем предыдущее значение самой переменной! Взгляните на пример:

```
var
  i: integer;
```

```
begin
  i:= 3;
  i:= 2 * 5 + 36 * i;  //сложное выражение
```

Тут мы объявили целочисленную переменную `i`. Этой переменной мы присвоили значение 3. Потом мы присвоили новое значение, которое сначала высчитывается компьютером из сложного выражения, а затем только присваивается переменной. Какое же значение теперь в `i`? Давайте считать. Из уроков математики мы знаем, что сначала вычисляются такие операторы, как умножение и деление, а только потом - сложение и вычитание. То есть, сначала высчитывается `2 * 5 = 10` (для компьютера знак "*" - это умножение, если кто-то не знает). Затем вычисляется `36 * i`, а `i` - это 3, мы делали это присваивание строчкой выше. Так что получаем `36 * 3 = 108`. Затем складываем эти значения: `10 + 108 = 118`. Именно столько и попадет, в конце концов, в переменную `i`.

Здесь у кого-то может возникнуть вопрос: а почему автор советовал давать переменным осмысленные имена, а сам назначил простое имя `i`? Вопрос справедливый, отвечу так. Это - не реальная программа со множеством подпрограмм и переменных, это простой пример с одной переменной, и мы в любом случае, не запутаемся. Кроме того, переменная была назначена не для какой-то конкретной цели, а лишь для демонстрации присваивания ей значения. Поэтому и имя у переменной простое. В более сложных примерах и в лабораторных работах я постараюсь придерживаться своих же рекомендаций.

Если вы заметили, то в нашей программе мы также присваивали переменной не просто значение, а составное выражение:

```
MyName:= 'Привет, ' + Edit1.Text + '!';
```

Здесь мы строку "`Привет, `" соединяли с тем, что хранилось в поле ввода `Edit1`, добавляли восклицательный знак, а результат уже присваивали переменной `MyName`. Обратите внимание, что после слов "`Привет`" мы вписали не просто запятую, а запятую с последующим пробелом. Если бы мы не указали этот пробел, то в результате текст

сливался бы, а это не очень красиво:

```
Привет,Дмитрий!
```

Константы

Константы используются реже, чем переменные, однако в некоторых случаях они могут быть очень полезными.

В математике константа (лат. constanta - постоянная, неизменная) - некоторая величина, не изменяющая своё значение в рамках рассматриваемого процесса. В программировании константа имеет то же значение. Причем константы бывают двух типов: простые неименованные) и именованные.

Простая константа представляет собой готовое значение, например,

```
i:= 3; //здесь 3 - это числовая константа
s:= 'строка'; // 'строка' - строковая константа
```

Конечно, число 3 всегда и везде будет равно 3, поэтому данное значение - константа. Такие простые константы мы применяем повсеместно, не особо задумываясь.

Именованная константа - несколько более сложное понятие. Фактически, это такая же ячейка оперативной памяти, как и переменная, но значения в этой ячейке не могут меняться. Применение таких констант полезно в двух случаях:

1. Когда не хочется запоминать реального значения, или когда оно довольно длинное. Например, число ПИ равно 3,141 592 653 589 793 238 462 643 383 279 502 88. Хочется вам запоминать такое значение? Не проще ли определить константу `pi` и записать в нее это число? Тогда в дальнейшем, вместо указания этого числа мы будем указывать константу `pi`. Компилятор сам подставит нужное значение вместо имени этой константы.
2. Когда какое-то значение используется в программе без изменений,

но со временем оно может стать другим. Например, минимальная зарплата (МРОТ - Минимальный Размер Оплаты Труда) на 2013 год принята в размере 5205 руб. Наша программа, к примеру, вычисляет зарплату сотрудников, для этого МРОТ может умножаться на различные коэффициенты, например, коэффициент выслуги лет. Поскольку такое вычисление будет многократным - для каждого сотрудника отдельно, да еще с различными условиями (отпуск, больничный, премия и т.п.), то разумно объявить константу `mrot` со значением 5205, и в дальнейшем использовать ее вместо реального значения. Почему такой подход удобен? Во-первых, наша программа может попасть в другой регион, а там может оказаться свой размер МРОТ. Например, в Ленинградской области принят МРОТ в размере 6800 руб., а в г. Москве и вовсе 12200 руб. Во-вторых, на 2014 год размер МРОТ наверняка увеличится, а значит, его значение изменится. В обоих этих случаях достаточно будет изменить значение константы в начале модуля, и во всех местах кода, где эта константа применяется, автоматически будет использовано новое значение. Нам не придется кропотливо выискивать по всему коду, где же мы использовали при расчетах размер МРОТ, чтобы вписать туда новое значение.

Итак, правило:

Константа - область оперативной памяти, значение в которой остается неизменным, пока программа работает.

Стоит отметить, что когда программа завершает свою работу, все ячейки, выделенные под переменные и под константы, очищаются. Вернее, перестают учитываться операционной системой. То есть, если другая программа захочет их использовать, Windows это разрешит. Но до тех пор, пока эта программа не запишет в ячейки уже свои значения, там будет "мусор" - набор непонятных данных. Вот почему очень важно не только объявлять переменные (константы), но и присваивать им начальные значения.

Константы объявляются в разделе `const` (англ. constants - постоянные, константы), причем этот раздел должен располагаться до раздела `var`. Объявление константы и присвоение ей значения в отличие от

переменных, происходит одновременно:

```
const
  mrot = 5205;
```

Здесь мы создали раздел констант, в котором объявили константу `mrot`, и сразу же присвоили ей значение 5205. Как видите, вместо оператора присваивания ":=", как это было у переменных, здесь используется простой знак "равно", а тип константы не указывается. Компилятор сам, в зависимости от указанного значения, присваивает константе наиболее подходящий тип.

Вернемся к нашему примеру, и снова доработаем код события нажатия на кнопку:

```
procedure TfMain.Button1Click(Sender: TObject);
const
  priv = 'Привет, ';
var
  MyName: String;
begin
  MyName:= priv + Edit1.Text + '!';
  ShowMessage(MyName);
  MyName:= 'Рады вас видеть, ' + Edit1.Text + '!';
  ShowMessage(MyName);
end;
```

В этом примере мы объявили константу `priv` и присвоили ей текст - начало приветствия. Причем раздел `const` мы поместили до раздела `var`. В строке

```
MyName:= priv + Edit1.Text + '!';
```

компилятор вместо "`priv`" подставит значение "`Привет,  `" и мы получим прежний текст приветствия.

Комментарии

Когда кода очень много, программист легко может в нем запутаться. Чтобы этого не произошло, в Паскале предусмотрены комментарии.

Комментарий - это пояснительный текст, который добавляется программистом к исходному коду и игнорируется компилятором.

То есть, программист сам себе оставляет комментарии> - заметки, указывающие, что происходит в данном участке кода, для чего создан данный элемент, что будет происходить дальше и т.п. При сборке программы компилятор FPC игнорирует весь этот текст и не включает его в программу. Комментарий остается лишь в исходных кодах.

Комментарии улучшают восприятие кода программистом, а если проект разрабатывается не одним программистом, а группой, тут без комментариев и вовсе не обойтись. Говорят, в компании Microsoft, когда принимают на работу нового программиста, смотрят на стиль его кода. И если там комментариев меньше, чем 1/3 от всей программы, ему отказывают в вакансии. Это и понятно - если кто-то другой будет подключать ваш модуль к общей программе, без комментариев ему будет трудно разобраться, что у вас к чему. Даже если вы сами вернетесь к вашей программе через какое-то время, без комментариев вам будет трудно вспомнить, где и что у вас хранится, иной раз проще написать такую программу заново. Поэтому использование комментариев обязательно для каждого программиста, уважающего свое собственное время, и время коллег.

Комментарии бывают однострочными и многострочными.

Однострочный комментарий начинается с символов "//" и может располагаться как на отдельной строке, так и после действующего оператора. Примеры мы уже видели:

```
i:= 3;  //присвоили этой переменной значение 3
```

Здесь текст после "//" - однострочный комментарий. Мы могли бы расположить его и на отдельной строке:

```
//присваиваем переменной новое значение:
i:= 10;
```

Имейте в виду, что если вы используете какой-то оператор после однострочного комментария на той же строке, для FPC это будет считаться продолжением комментария, и оператор не будет выполнен:

```
//присваиваем переменной новое значение:  i:= 10;
```

Иногда однострочные комментарии применяются для визуального отделения одной смысловой части кода от другой:

```
//************************************************
…какой-то код
//************************************************
Или так:
//------------ НАЧАЛО БЛОКА КОДА ----------------------------
…какой-то код
//------------ КОНЕЦ БЛОКА КОДА -----------------------------
```

В общем, для улучшения читабельности программы, после "//" вы можете использовать собственные разделители.

Многострочный комментарий. Когда комментарий содержит много пояснительного текста, его делают многострочным. Для этого его помещают в фигурные скобки "{…}". Всё, что находится внутри таких скобок, считается комментарием:

```
{Пример комментария}

{Пример длинного
 многострочного комментария}

{
Пример длинного
многострочного комментария
}
```

Также в Паскале вместо фигурных скобок для многострочного комментария можно применять символы комментариев языка C: "(*...*)":

```
(*
Пример многострочного комментария
в стиле C/C++
*)
```

Какие комментарии применять - дело вкуса. Их можно и комбинировать между собой, в серьезных проектах часто так и делают.

Совет: комментарии - это хорошо, однако не переусердствуйте и не давайте совсем уж очевидных комментариев.

Выше я приводил такие очевидные комментарии, но там это было необходимо, чтобы пояснить новую тему:

```
var
  i: integer;  //объявили переменную с типом integer - целое число
begin
  i:= 3;  //присвоили этой переменной значение 3
```

Теперь вы и без комментариев сможете разобраться в коде этого примера, а значит, комментарии здесь можно и не давать.

Символы и строки

Лекция посвящена работе с символами и строками. Подробно рассмотрен механизм строк в Lazarus, символьные и строковые типы данных. Подробно рассмотрены компоненты для работы со строками.

Цель лекции

Знакомство с символьными и строковыми типами данных, использование компонентов для работы со строками.

Понятия "символ" и "строка"

Для рядового пользователя эти понятия весьма абстрактны. Когда он вводит с клавиатуры "A", то считает это символом, буквой. Когда вводит "4", то это - цифра. А про всякие там пробелы, знаки препинания или арифметические знаки он и вовсе не думает. Но программист должен понимать, как все эти знаки воспринимаются компьютером. Поэтому наберитесь терпения, мы изучим эти понятия достаточно подробно.

На самом деле, всё, что мы вводим с клавиатуры - это символы. Буква "z", цифра "3", пробел, знак умножить, знак процента и т.д. - всё это символы. Компьютер же может оперировать только цифрами, причем двоичными - такими, которые содержат лишь 0 или 1. Все эти буквы, десятичные цифры и прочие знаки для него не значат ровным счетом, ничего. И для того, чтобы мы могли как-то обрабатывать текст, цифры и прочую информацию, нужно было придумать специальную систему для перевода информации в понятную компьютеру, и обратно. Так появились кодовые страницы.

Кодовая страница (англ. code page) - специальная таблица, сопоставляющая каждому значению байта некоторый символ.

Не очень понятно? Давайте разбираться. Мы знаем, что информация измеряется байтами, и что в одном байте 8 бит. Бит - это минимальная единица информации, с которой может работать компьютер. В бите может храниться либо 0, либо 1.

На заре развития компьютеров была разработана кодовая страница ASCII (англ. American Standard Code for Information Interchange - Американский кодовый стандарт для обмена информацией). Первая версия этого стандарта появилась в 1963 г. Эта страница содержала 7-ми битные символы, в каждом байте один бит был не задействован. Минимальное двоичное число, которое могло храниться в 7-ми битах - это ноль. Максимальное - 1111111.

Нажмите кнопку "Пуск", выберите команду "Выполнить" и в окне "Запуск программы" укажите

```
calc
```

и нажмите <Enter>. Загрузится стандартный калькулятор Windows. В главном меню программы, в разделе "Вид" выберите "Инженерный". В левой части калькулятора, ниже поля ввода чисел, вы увидите переключатели различных систем исчисления:

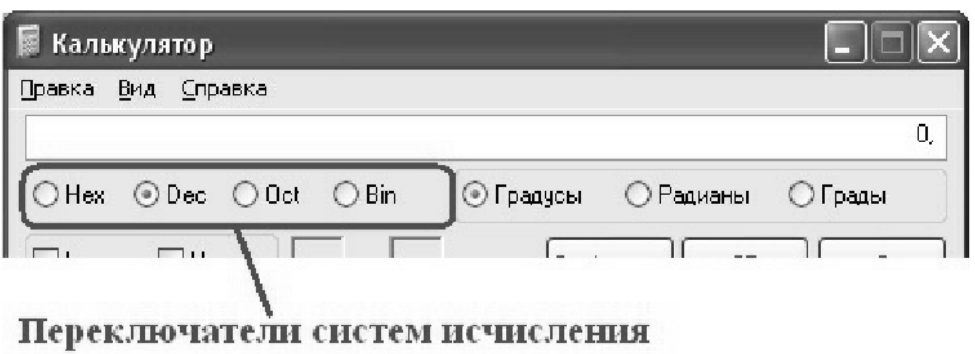


Рис. 5.1. Переключатели калькулятора

Здесь мы имеем возможность переключаться на четыре системы исчисления:

- Hex - 16-ричная. Используется в основном, в языке Ассемблере или при отладке программ.
- Dec - 10-тичная. Наша обычная система, используется по умолчанию.
- Oct - 8-ричная. Почти не используется.

- Bin - 2-ичная. Хоть она и является для компьютера единственно понятной, но вследствие громоздкости записей, в программировании её обычно не применяют.

Давайте посмотрим, сколько символов могло содержаться в кодовой странице ASCII. Переключитесь на двоичную систему (Bin), введите 7 единиц, затем переключитесь обратно на десятичную систему (Dec). У нас получилось 127. Именно столько символов содержалось в первой ASCII таблице. Помимо латинских букв, таблица содержала и другие символы - цифры, арифметические знаки, знаки препинания, символы пробела, скобки и т.п. Каждому символу соответствовал собственный номер в таблице. Если мы вводили английскую букву "A", то в компьютер попадал номер этого символа в таблице - 65. Или, в двоичном виде, 100 0001. Таким образом, символы можно было сравнивать между собой. Английское "B" находилось под номером 66 и, следовательно, было больше, чем "A". Строчные буквы имели другие номера, например, "a" шла в таблице под номером 97 и считалась большей, чем "A". Мы вводим символы, которые автоматически преобразовываются в цифры, с которыми уже оперирует компьютер.

Тут нужно сделать одно важное пояснение. Если мы вводили число "65", то для ПК это не было числом 65, или латинской буквой "A", это было двумя символами "6" и "5". Символу "6" соответствует номер 54 кодовой таблицы, а символу "5" - номер 53. Таким образом, числа, которые мы вводим в ПК, на самом деле не числа, а текстовые символы! Преобразования таких символов в числа и обратно обычно производятся программой автоматически. Такие преобразования, например, постоянно выполняет стандартный калькулятор Windows. А когда мы будем изучать числа, нам самим придется выполнять такие же преобразования.

Всё бы хорошо, но помимо английского языка, в мире существует множество других языков! Мы, например, пишем кириллицей - русским начертанием символов. 127 символов было явно недостаточно, чтобы можно было вводить текст и на других языках. Поэтому таблица ASCII развивалась, из года в год появлялись новые стандарты. Каждый символ стал уже 8-ми битным. Посмотрите на калькуляторе - восемь бит могут содержать максимальное число 1111 1111, при переводе в десятичную систему мы получим 255 символов. Первая половина

таблицы оставалась неизменной, зато вторую половину таблицы можно было задействовать для символов других языков и псевдографики, с помощью которой программисты времен MS-DOS рисовали окошки, панельки, таблицы и меню. Однако и этого было слишком мало, чтобы закодировать символы всех языков на Земле. Для каждого языка приходилось разрабатывать свой стандарт, несовместимый с другими. Причем, для одного языка могло быть разработано несколько стандартов! Для русского языка, например, имеются стандарты CP866 (Code Page 866), KOI8-R, KOI8-U, ISO-8859-5, и это только самые распространенные! В хаосе стандартов надо было как-то разбираться, совершенствовать их.

На смену ASCII пришла кодировка ANSI (англ. American National Standards Institute - Американский Национальный Институт Стандартов). Так, в MS Windows кодовая страница ANSI, содержащая кириллицу - это Windows-1251 (или CP1251), которая появилась в 1990-1991 гг.

Однако и этого было недостаточно, ведь для каждого языка по-прежнему требовалась собственная кодировка, а языков на Земле много. Назрела необходимость переходить к "широким" стандартам, в которых символ занимает более одного байта. Так, в 1991 г. был предложен стандарт Юникод (англ. Unicode) - универсальная система кодирования символов, представляющая знаки практически всех языков. В этом стандарте в одном документе можно использовать символы кириллицы, китайские или японские иероглифы, знаки математических формул, музыкальные знаки и т.п.

Первая версия Юникода имела фиксированный размер символов 2 байта (16 бит). В одной кодировке уже можно было использовать 65 535 символов! Однако оказалось, что и этого недостаточно. Юникод получил дальнейшее развитие, и из года в год стали появляться новые версии и стандарты, основанные на Юникоде. Имеются такие стандарты, как UTF-8 (англ. Unicode Transformation Format, 8 bit - 8-ми битный формат преобразования Юникода), UTF-16, UTF-32.

В Lazarus, в основном, используется формат UTF-8, так что разберем его. UTF-8 появился 2 сентября 1992 года. Основное его отличие от первоначального Unicode в том, что в UTF-8 символы имеют не фиксированный размер! Если используются символы с номером

меньшим, чем 128, то они занимают 1 байт, как обычный ASCII-текст. Символы с номером от 128 и больше могут занимать от 2 до 6 байт (реально максимальный размер символа - 4 байта, т.к. в Юникоде нет символов с большим номером). Символы кириллицы занимают, например, по 2 байта. Так что цепочка символов в 5 байт в UTF-8 не всегда означает строку из пяти символов.

Такой подход делает UTF-8 самой экономичной кодировкой для совместимости со старыми стандартами, однако есть и минусы. К сожалению, для русскоязычных (и вообще для всех не англоязычных) пользователей Windows в Lazarus придется столкнуться с некоторыми проблемами применения различных кодировок: в Lazarus используется кодировка UTF-8, ОС Windows использует UTF-16, а в консольных приложениях Windows используется системная кодировка CP866 (то есть, стандарт ANSI). Её же используют некоторые функции компилятора FPC. Так что в некоторых случаях нам придется пользоваться функциями преобразования кодировок, например, `UTF8ToConsole()`, `CP866ToUTF8()` и т.п. Мы познакомимся с ними позже, в свое время.

Итак, подведем некоторые итоги.

- Символ - это графическое изображение буквы, цифры, арифметического знака, знака препинания или какого-либо другого знака, отвечающее какому-либо стандарту кодировки символов.
- Каждому символу соответствует его номер в кодовой таблице символов.
- Кодировок (code page) существует множество.
- Строка - это цепочка символов.
- Цифры, которые мы набираем на клавиатуре - это символы. Чтобы обрабатывать их, как цифры, программа выполняет автоматическое преобразование из символов в цифры, и обратно, когда выводит нам результаты вычислений.
- UTF-8 - это одно из представлений Юникода, используемое в Lazarus.
- В одной строке могут встречаться символы, которые занимают 1, 2 или даже больше байт.

Символьные типы данных

В Lazarus имеется символьный тип `Char`. Переменная такого типа занимает ровно один байт памяти и может содержать любой ASCII - символ. Русскому языку с этим типом не повезло - поскольку символы кириллицы занимают по 2 байта, использовать этот тип с русскими буквами нельзя. Давайте откроем Lazarus с новым проектом. Если Lazarus у вас уже загружен, закройте старый проект командой Файл -> Закрыть, и создайте новый проект командой Проект -> Создать проект -> Приложение. Сразу же сохраните проект в папку 05-01 там, где вы храните все проекты лекций, проекту дайте имя `MyStrings`, и не забудьте назвать модуль главной формы `Main`, а саму форму `fMain`. Пусть это войдет у вас в привычку. Как всё это делается, вы должны знать по предыдущим лекциям. Не вздумайте давать своим проектам имена, используя ключевые (зарезервированные) слова. Например, нельзя давать проекту или модулю имя `Strings`, поскольку это слово зарезервировано, но `MyStrings` можно.

Прямо посреди формы установите кнопку TButton, дважды нажмите на нее, чтобы сгенерировать обработчик `OnClick`. В обработчике создадим раздел переменных var, укажем там одну переменную типа `Char`, и запрограммируем следующий код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  ch1: Char; //переменная символьного типа
begin
  ch1:= 'Z';
  ShowMessage(ch1);
end;
```

Как видим, здесь мы переменной `ch1` присвоили значение - символ "`Z`". Запомните правило:

Символы и строки в Lazarus должны быть заключены в одинарные кавычки. Если требуется ввести в текст одинарную кавычку (например, в слове can't), то следует указать две таких одинарных кавычки, которые также заключены в одинарные кавычки.

Например:

```
Mystring:= 'I can' + '"' + 't';
```

Этот пример выполнять не нужно, он просто демонстрирует использование кавычек в строковых выражениях, и возможность в выражении соединять несколько строк в одну с помощью знака "+". Однако вернемся к нашему проекту. После того, как мы присвоили символьной переменной большую английскую букву "Z", мы вывели содержимое переменной с помощью строки

```
ShowMessage(ch1);
```

Сохраните проект, скомпилируйте его и выполните. Когда программа с кнопкой появится на экране, нажмите нашу единственную кнопку - выйдет сообщение с буквой "Z":

Рис. 5.2. Вывод символа

Таким образом, мы можем работать с отдельными символами. Однако если вы исправите код, и вместо буквы "Z" в строке

```
ch1:= 'Z';
```

укажете русский символ, то при попытке скомпилировать и запустить проект выйдет ошибка "`Error: Incompatible types: got "Constant String" expected "Char"`" - несовместимость типов `String` (строка) и `Char` (символ). Поскольку русские буквы занимают по два байта, Lazarus их считает строкой символов. Однако это не значит, что мы совсем не имеем возможности работать с

отдельными символами кириллицы.

Ранее упоминалось, что в Lazarus используется формат UTF8. Для поддержки этого формата были разработаны расширенные типы данных, в том числе и символьные. Так, имеется тип `TUTF8Char`, который позволяет работать с любыми отдельными символами, в том числе и русскими. Имеет смысл всегда использовать его вместо стандартного `Char`. Но для этого нам нужно будет подключить модуль `LCLType`, где этот тип описан.

Перейдите в редактор кода и пролистайте код модуля вверх. Там вы увидите раздел `uses` (англ. use - использовать), где перечислены подключенные модули. Нам нужно поставить после последнего модуля запятую и добавить наш подключаемый модуль:

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

Если строка получается длинной, то после запятой можно переносить текст на другую строку.

Теперь вернемся к обработчику нажатия на кнопку и переделаем его:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  ch1: Char;
  ch2: TUTF8Char;
begin
  ch1:= 'Z';
  ch2:= 'Я';
  ShowMessage(ch1);
  ShowMessage(ch2);
end;
```

Теперь всё сработает как надо, и русский символ "`Я`" выйдет так же, как английский "`Z`".

Однако, это еще не всё. Символы можно вводить, указывая их номер в таблице символов. Для этого перед номером нужно указать символ "#", например:

```
ch1:= #90;
```

Под номером 90 находится буква Z в таблице символов, так что результат не изменится. Также в строковых выражениях можно указывать и специальные символы. Например, символ под номером 13 - это символ перехода на другую строку. Давайте снова немного изменим код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  ch1: Char;
  ch2: TUTF8Char;
begin
  ch1:= #90;  //буква Z
  ch2:= 'Я';
  ShowMessage(ch1 + #13 + ch2); //вывод двухстрочного сообщения
end;
```

Сохраните проект, скомпилируйте и запустите на выполнение. Теперь содержимое двух символьных переменных выходит в одном сообщении, но каждое на своей строке.

Строковые типы данных

В Lazarus основным строковым типом является `String` (англ. string - строка). На форму нашего проекта разместите еще одну кнопку. Её обработчик будет выглядеть так:

```
procedure TfMain.Button2Click(Sender: TObject);
var
  s: String;
begin
  s:= 'Это строка из пяти слов!';
```

```
  ShowMessage(s);
end;
```

Как видите, поскольку строка в Lazarus имеет формат UTF8, то никаких особых ухищрений для работы со строками тут не нужно, в переменную типа `String` можно записать любой, в том числе и русский текст. Размер строки `String` неограничен, однако имеется возможность жестко задать размер. Такой способ используется, когда вы точно знаете, что больше данного размера строка не будет. В этом случае размер указывается после ключевого слова `String` в квадратных скобках, например:

```
var
  MyStr: String[50];
```

В переменную `MyStr` можно записать до 50 символов. Максимальный размер строки, который можно жестко задать таким способом - 255 символов. Однако имеются в виду символы ASCII, то есть английские, однобайтовые. Пример:

```
var
  s: string[7];
begin
  s:= 'Привет!';
  ShowMessage(s);
end;
```

Данный пример ошибки не вызовет, однако сообщение выйдет не полностью, а обрезанным: "При". То есть, три первых буквы заняли 6 байт, четвертая уже не уместилась. В данном случае будет правильным указать размер не 7, а 14 - по удвоенному количеству букв. Впрочем, на практике обычно применяют тип `String` без ограничений, в этом случае строка обрабатывается правильно.

Имейте в виду, что тип `String` является динамичным, то есть, заранее память для него не выделяется. Строго говоря, выделяется память на

указатель строки, а не на саму строку. Физически строке выделяется необходимая память только в момент присвоения ей какого-то значения. Однако нередко возникает необходимость очистить такую строку. Для этого достаточно присвоить ей пустые кавычки, без пробелов и без каких-либо других символов:

```
s:= '';
```

В этом случае, строке присваивается значение `Nil`, то есть ноль, ничего, и строка становится пустой.

Строковой переменной можно присваивать значения символьных переменных или констант, например:

```
var
  c: Char;
  s: String;
begin
  c:= 'Z';
  s:= c;
```

Помимо `String` в Lazarus есть и другие строковые типы.

`ShortString`	Короткая строка, которая может содержать максимум, 255 ASCII-символов. То есть, `ShortString` - это `String[255]`.
	Строка неограниченной длины из ANSI-символов. Фактически всё, что мы говорили про `String`, относится именно к `AnsiString`. Отдельного типа `String` как такового, не существует, это просто сокращенная запись `AnsiString`. Однако это зависит от настроек. Есть в Паскале такие переключатели, которые еще называют директивы компилятора. Такие директивы подобно комментариям, заключают в фигурные скобки, а первым символом должен быть символ доллара. Тип, который будет использован в качестве `String`,

`AnsiString`	зависит от директивы `{$H}`. Если она выключена `{$H-}`, то при указании типа `String` будет подразумеваться тип `ShortString`. Если она включена `{$H+}` - то `AnsiString`. По умолчанию переключатель включен, вы сможете это увидеть в третьей строке модуля, пролистав его код вверх: unit Main; {$mode objfpc}{$H+} На практике короткую строку используют крайне редко, ведь если нужна короткая строка, то ее размер можно указать явно. Поэтому данный переключатель исправлять вручную обычно не приходится. Давайте считать, что `String` и `AnsiString` - это одно и то же.
`PChar`	Строка в стиле C/C++ с нулевым символом в конце. Такой тип строк используется в функциях Windows API, поэтому имеет поддержку и в Lazarus. Что это за строка такая? Дело в том, что обычные строки динамические, то есть фактически переменная типа `String` - это указатель на строку. Эта переменная хранит физический адрес строки в памяти, и количество занимаемых ею байт. Строка типа `PChar` устроена несколько иначе. Это тоже указатель на строку, но он не хранит её размер. А как тогда компилятору знать, сколько байт нужно считывать при обращении к такой переменной? Дело в том, что в строках типа `PChar` завершающим всегда будет нулевой символ, то есть, символ `#0`. Если такой символ поместить внутри строки, то компилятор не будет читать всю строку. Встретив символ `#0`, он решит, что строка закончена. Мы будем использовать `PChar`, когда дойдем до функций `WinAPI`.

Как правило, строкам одного типа можно присваивать значения строк другого типа, компилятор автоматически преобразует текст. Исключение составляют строки `PChar`, тут нам придется делать преобразование вручную, с помощью функции `PChar()`. Исправим обработчик второй кнопки:

```
procedure TfMain.Button2Click(Sender: TObject);
var
  s1: String; //он же AnsiString
  s2: ShortString;
  s3: PChar;
begin
  s1:= 'Это строка из пяти слов!';

  //Теперь присвоим тот же текст в ShortString:
  s2:= s1;

  //Тот же текст в PChar. Для преобразования строки в этот тип
  //используем функцию PChar():
  s3:= PChar(s1);

  //Соберем сообщение из трех разнотиповых строк. Каждый тип
  //выведем в отдельной строчке:
  ShowMessage(s1 + #13 + s2 + #13 + s3);
end;
```

Сохраните проект, скомпилируйте и запустите. При нажатии на вторую кнопку у вас выйдет сообщение из трех строк

Имеются еще типы `UnicodeString` и `WideString`, оба типа содержат двухбайтовые символы. Но работать с такими типами неудобно - возникают проблемы с кириллицей, для нас гораздо удобней простой `String`. Поэтому рассматривать данные типы мы не будем.

Резюмируем некоторые положения:

- Три основных строковых типа: `AnsiString`, `ShortString` и `PChar`.

- Обычно указывают тип `String`. Если переключатель `{$H}` включен (по умолчанию), то используется тип `AnsiString`. Иначе - `ShortString`.
- Строке можно присвоить либо текст, заключенный в одинарные кавычки, либо содержимое другой строковой переменной, например:

```
s1:= 'Текст';
s2:= s1;
```

- Строке типа `PChar` можно присвоить либо текст, либо содержимое другой строковой переменной. Но если тип этой переменной отличается, то нужно сделать преобразование функцией `PChar()`:

```
var
  s: String;
  pc: PChar;
begin
  s:= 'Текст';
  pc:= 'Текст'; //присвоили текст
  pc:= PChar(s); //присвоили преобразованное значение переменной s
```

Компоненты для ввода строк

Теперь мы знаем, что такое строки, самое время разобрать способы предоставления пользователю возможностей эти самые строки вводить в нашу программу. Помните второй проект второй лекции? Там мы предлагали пользователю ввести свое имя, которое затем использовали в приветствии. Для этого мы использовали компонент `TEdit`. Разберем его возможности, и познакомимся с другими аналогичными компонентами.

Для этого закройте старый проект и создайте новый. Как всегда, главную форму назовите `fMain` (свойство `Caption`, если кто забыл), и сохраните проект. При этом проекту дайте имя `EditControls`

(контролами программисты между собой называют компоненты, с помощью которых программа взаимодействует с пользователем), а модуль этой формы назовите `Main`. Проект сохраните в папку 05-02 там, где у нас хранятся все остальные проекты.

Компонент TEdit

На форму установите компонент `TEdit` (если не помните, где он находится, смотрите рис. 2.5 в лекции №2). По умолчанию, он имеет ширину 80 пикселей (свойство `Width`), увеличьте его до 200. Теперь давайте разбираться с основными свойствами этого компонента, которые могут нам понадобиться, для этого он должен быть выделен.

Начнем с самого начала. Рекомендую прочитать пояснения к свойству, а затем "поиграть" с его значениями, меняя их и запуская программу на выполнение, чтобы посмотреть результат. Затем вернуть значение по умолчанию, если не будет предложено иное, после чего переходить к следующему свойству. Таким образом, вы познакомитесь с компонентом более подробно.

`Align`	Выравнивание компонента. С этим свойством мы уже знакомы, оно позволяет компоненту занять весь верх, низ, левую или правую сторону, или же всю форму или панель, на которой компонент находится. В компоненте TEdit это свойство тоже есть, но применяется очень редко - трудно придумать ситуацию, в которой этот компонент потребуется таким образом выровнять. Обычно это свойство не трогают. Значения этого свойства должны быть вам знакомы по лекции №3.
	Выравнивание текста. Вот это свойство используется чаще. Оно позволяет выровнять текст в компоненте по центру, по левому или правому краю. Как выравнивать текст, обычно каждый решает для себя сам, но есть некоторые рекомендации: простой текст выравнивают обычно по левому краю, цифры -

`Alignment`	по правому, а такие данные, как дата-время можно выровнять по центру. Итак, значения этого свойства: • `taCenter` - по центру. • `taLeftJustify` - по левому краю. Это значение используется по умолчанию. • `taRightJustify` - по правому краю.
`Anchors`	Привязка компонента к краям формы или панели, на которой компонент находится. С этим свойством вы также знакомы по третьей лекции.
`AutoSelect`	Автоматическое выделение текста. Если свойство равно `True` (по умолчанию), то когда компонент становится активным, выделенным, или как говорят, получает фокус ввода, текст в компоненте, если он есть, автоматически выделяется. Иначе текст будет невыделен.
`CharCase`	Регистр символов текста в компоненте. Может иметь следующие значения: • `ecLowerCase` - Все символы строчные (обычные), даже если набирать текст с нажатой клавишей <Shift> или с включенной <Caps Lock>. • `ecNormal` - Обычный текст, в котором могут быть и строчные, и прописные символы. Используется по умолчанию. • `ecUpperCase` - Все символы прописные (заглавные).
`Color`	Позволяет выбрать цвет фона в компоненте.
`Font`	Позволяет выбрать шрифт, используемый в компоненте. Вообще каждый решает сам, как будет выглядеть дизайн его программы, однако желательно для всех компонентов использовать один и тот же шрифт. Если вас не устраивает шрифт по умолчанию, поменяйте его для всей формы, и он автоматически

Font	будет использоваться для всех компонентов, которые вы на этой форме установите! Если же в вашем проекте будет много форм, то выработайте для них какие-то одни стандарты (шрифт, цвет фона, обрамление и т.п.), и придерживайтесь их во всех формах.
Hint	Текст подсказки, который будет всплывать, когда пользователь подведет к компоненту указатель мыши.
MaxLength	Максимальный размер текста в символах. По умолчанию равен нулю, что снимает ограничение в размерах. Если текст не умещается в поле ввода, он будет прокручиваться. Ограничение на количество символов иногда бывает необходимо, например, если текст будет записан в базу данных или запись, которые имеют фиксированный размер строки. Во всех остальных случаях свойство можно не изменять.
PasswordChar	Символ, скрывающий пароль при вводе. Очень интересное свойство. Должно быть, всем приходилось когда-то вводить пароль. И обычно вместо нужных букв и цифр в этом поле отображались звездочки или кружочки. На самом деле, текст остается неизменным, но он прячется от тех, кто может оказаться у вас за спиной и подсмотреть ваш пароль. Реализуется эта возможность как раз данным свойством. Здесь нужно указать номер символа в таблице символов. По умолчанию, это #0 - нулевой символ, означающий, что текст выводится, как есть. Но можно поставить и другой символ, причем указать именно символ, а не его номер. Давайте в этом свойстве укажем символ "*". Как только вы нажмете <Enter>, сразу же изменится и свойство EchoMode - оба эти свойства взаимосвязаны. Мы разберем его чуть ниже, а пока сохраните проект и запустите на выполнение. Теперь любой текст, включая пробелы

	и другие знаки, будет выводиться на экран в виде звездочек. Однако это не все.

Мы имеем возможность маскировать пароль и другими символами. Выделите компонент `TEdit` и измените его свойство `Font`, выбрав там шрифт Wingdings (если у вас ОС Windows). Это тот редкий случай, когда имеет смысл выбрать другой шрифт для отдельного компонента. Теперь нам нужно выбрать символ. Откройте редактор MS Word. Там выполните команду Вставка -> Символ:

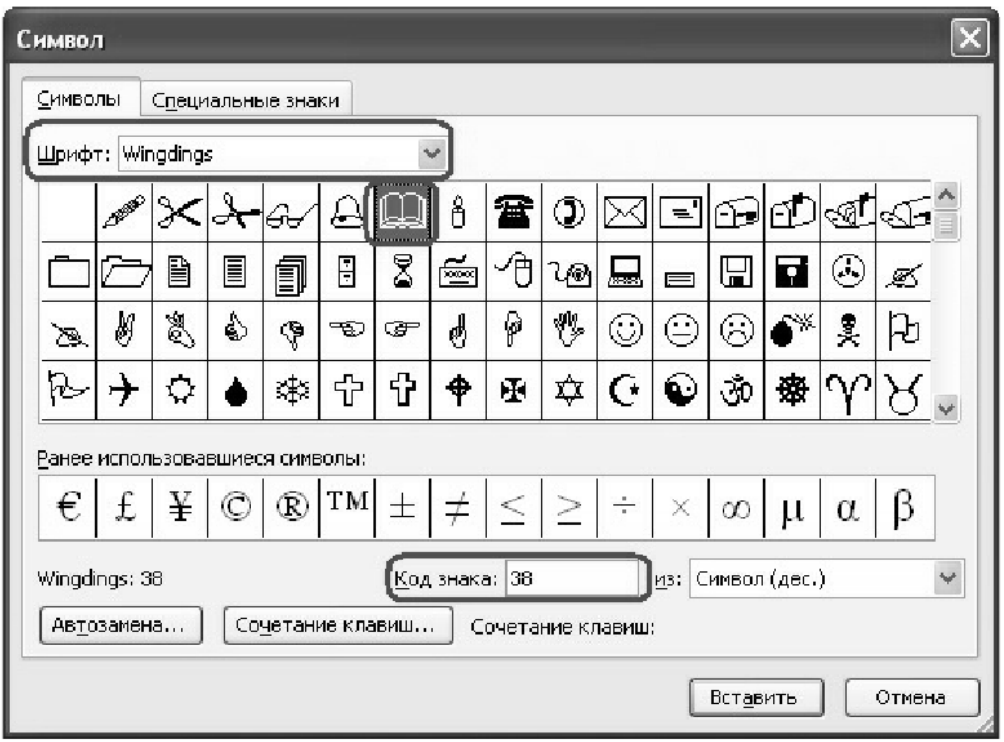


Рис. 5.3. Определение кода символа

В поле "Шрифт" выберите Wingdings, затем выберите подходящий символ, например, книгу. Потом посмотрите код выбранного знака, в данном случае это 38. Нажмите "Отмена" и закройте MS Word - он нам больше не нужен. Теперь вернитесь к проекту и в свойстве `PasswordChar` укажите `#38` - код выбранного нами знака. Когда нажмете <Enter>, то значение `#38` изменится на `&` - не обращайте внимание. Сохраните проект и запустите его на выполнение - текст

будет скрываться за пиктограммой книги. Используя различные шрифты, вы можете подобрать для себя и другие подходящие символы.

`EchoMode`	Тип сообщения. Связан с `PasswordChar`, и обычно устанавливается автоматически, но можно установить его и вручную. Имеет следующие значения: • `emNone` - вместо символов выводятся пробелы. Сам текст не меняется. • `emNormal` - обычный тип, используется по умолчанию. Устанавливается, если в `PasswordChar` установлено значение `#0`. • `emPassword` - тип пароля. Устанавливается, если в `PasswordChar` вы укажете другое значение, не `#0`. Сам текст не меняется.
`ReadOnly`	Только для чтения. Разрешает или запрещает пользователю ввод текста. Если `False`, то пользователь может и читать, и вводить текст, иначе он может только читать его. Однако в любом случае вы можете менять там текст программно. Позже мы разберем такую возможность.
`Text`	Основное свойство этого компонента. Содержит текст, введенный пользователем, или установленный в компоненте программно. Причем даже если вы его скроете, как пароль, сам текст меняться не будет. К этому свойству можно относиться, как к строковой переменной, которую может изменять пользователь. По умолчанию, в это свойство сразу же прописывается имя компонента, в нашем случае, `edit1`. На практике такой текст вряд ли понадобится, поэтому программист обычно очищает это свойство. Очистите его и вы. Как только для подтверждения нажмете <Enter>, текст в компоненте на форме исчезнет.

Остальные полезные свойства должны быть вам знакомы по предыдущим лекциям. Итак, установите для компонента `Edit1` шрифт Times New Roman, размер шрифта пусть будет 11. В свойстве

Times New Roman, размер шрифта пусть будет 11. В свойстве `PasswordChar` пусть будет значение по умолчанию - `#0`. Свойство `Text` очищено, а свойство `ReadOnly` установлено в `True`. Теперь ниже `TEdit` установите кнопку, сгенерируйте для нее обработчик нажатия, и введите там следующий код:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  Edit1.Text:= 'Привет!';
end;
```

Здесь мы программно меняем текст в компоненте. Для этого мы обращаемся к нему по имени - `Edit1`, после чего ставим точку. Обратите внимание - если поставить точку и чуть подождать, всплывет окошко со всеми свойствами, методами и событиями, которые мы можем использовать. Нам нужно свойство `Text`. Как только мы введем первую "T", список отфильтруется, оставив там только команды на букву "T". Как только мы введем следующую букву "e", то в списке останется только одно подходящее свойство - "`Text`". То, что нужно. Можно просто нажать <Enter>, не вводя остальных символов, и свойство само вставится в код. Причем вместе с последующим оператором присваивания. Мелочь, а приятно. Останется только указать строку, которую мы хотим присвоить.

Таким образом, можно менять программно многие (но не все) свойства компонентов, их размеры и положение, текст и цвет, и многое другое. Сохраните проект и запустите его на выполнение. Если вы все сделали правильно, то пользователь не сможет вписывать текст в `Edit1`, а при нажатии кнопки Button1, текст там поменяется программно. Теперь давайте чуть изменим команду присваивания:

```
Edit1.Text:= Edit1.Text + 'Привет! ';
```

С выражениями мы уже работали, так что тут трудностей быть не должно. Мы просто берем текст, который уже был в свойстве `Text` (при первом нажатии там будет пустая строка), и прибавляем к нему строку `'Привет!  '` - не забудьте после восклицательного знака

проект и запустите его. Нажимая кнопку, мы сможем многократно добавлять слово 'Привет! ' в компонент.

Компонент TLabelEdit

На вкладке Additional Палитры компонентов есть странный компонент TLabelEdit:

Рис. 5.4. Компонент TLabelEdit

Установите его на свободное место на форму, и посмотрите на его свойства в Инспекторе объектов. В общем-то, это гибрид метки TLabel и текстового поля TEdit. Свойства почти все знакомы, но есть и некоторые различия.

EditLabel - раскрывающее свойство, внутри которого находится свой набор подсвойств для метки. Слева от этого свойства мы видим кнопку "+", щелкнув по которой раскроем подсвойства. Здесь нам понадобится, прежде всего, свойство Caption, в котором мы можем записать текст метки. Этот текст будет пояснять назначение текстового поля. Напишем в этом свойстве Фамилия, текст метки изменится, а пользователь сразу поймет, что сюда он должен будет вписать свою фамилию:

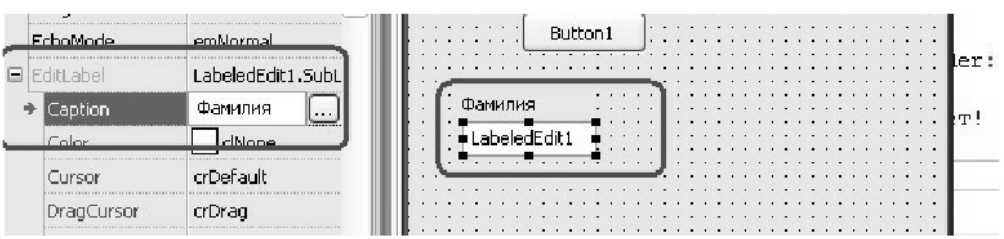


Рис. 5.5. Компонент TLabelEdit в Инспекторе объектов и в Редакторе форм

Можете щелкнуть по кнопке "-" слева от свойства, закрыв список подсвойств, больше ничего интересного там нет. Посмотрим, какие свойства еще нам понадобятся.

`LabelPosition` - местонахождение метки. Может быть:

- `lpAbove` - сверху текстового поля. Это значение по умолчанию.
- `lpBelow` - снизу текстового поля.
- `lpLeft` - слева
- `lpRight` - справа

`LabelSpacing` - расстояние в пикселях между меткой и текстом. По умолчанию равно 3 пикселя. Изменяя это значение, можно приближать метку к полю или отодвигать ее.

В остальном, этот компонент почти полностью идентичен обычному `TEdit`.

`TLabelEdit` будет полезен при создании форм, где пользователь должен вписывать различные данные, для этого раньше применялись отдельно `TLabel` и `TEdit`. Данный гибрид, безусловно, удобней.

Компонент TMaskEdit

На той же вкладке Additional находится еще один полезный компонент для ввода строк - `TMaskEdit`:

Рис. 5.6. Компонент TMaskEdit

Этот компонент предназначен для ввода текста по определенным шаблонам - маскам. Когда пользователю нужно будет ввести в этот

шаблонам - маскам. Когда пользователю нужно будет ввести в этот компонент какой-то текст, он будет вынужден вводить его именно в том виде, в котором нам нужно.

Установите `TMaskEdit` на форму и посмотрите на его свойства в Инспекторе объектов. В принципе, все свойства нам знакомы. Интерес вызывает лишь одно свойство `EditMask`. Это сложное раскрывающееся свойство, то есть, щелкнув по кнопке с тремя точками правее свойства, можно его раскрыть:

Рис. 5.7. Раскрывающееся свойство EditMask

Когда оно раскрыто, мы видим следующую картину:

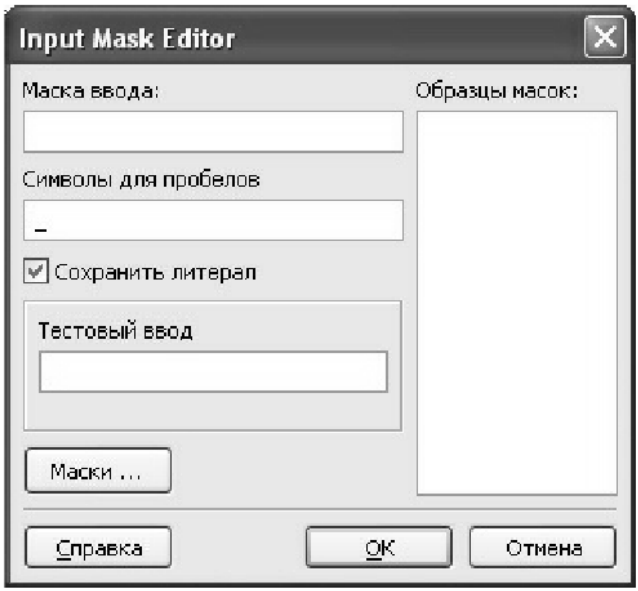


Рис. 5.8. Редактор масок

В строке "Маска ввода" мы можем указать маску, которая будет влиять на формат заполняемой пользователем строки. В маске можно применять

добавляемые самой маской. Маска может иметь следующие символы:

Таблица 5.1. Символы маски компонента TMaskEdit

0	В данной позиции должна быть цифра. Если пользователь введет не все цифры маски, будет создана исключительная ситуация, и он не сможет продолжать работу, не заполнив все положенные цифры. Если такая ситуация возникла у вас во время тестового выполнения программы, можете закрыть программу командой "Запуск->Сбросить отладчик".
9	В данной позиции может быть либо цифра, или ничего.
#	В данной позиции может быть знак "+", "-", цифра, или ничего.
L	В данной позиции должна быть буква.
l	В данной позиции должна быть буква или ничего.
A	В данной позиции должна быть буква или цифра.
a	В данной позиции должна быть буква или цифра, или ничего.
C	В данной позиции должен быть любой символ.
c	В данной позиции должен быть любой символ, или ничего.

Кроме того, в маске можно использовать различные разделители:

Таблица 5.2. Символы-разделители маски компонента TMaskEdit

:	Обычно применяется для разделения часов, минут, секунд.
/	Обычно применяется для разделения года, месяца, дня.
_	Обычно применяется для разделения цифр, например, в телефонных номерах.

Примеры масок:

- `999-99-99` (телефон)
- `99/99/9999` (дата)
- `00:00:00` (время)
- `999999,00 р.` (денежный формат)

Давайте укажем в строке "Маска ввода" маску телефона:

```
000-00-00
```

Использование нулей в качестве маски означает, что в этом месте обязательно должна быть цифра. Это разумно, ведь иначе номер телефона будет неполный, и мы не сможем дозвониться до абонента. Символы "–" служат разделителями, и добавляются автоматически. Введите маску, нажмите "OK", сохраните проект и запустите его на исполнение. Теперь мы сможем вводить в маску только цифры. Причем если мы введем не все цифры, и попытаемся выйти из строки редактирования, сгенерируется ошибка. Нам придется либо вернуться и заполнить номер до конца, либо остановить программу, сбросив отладчик.

В Редакторе масок помимо строки "Маска ввода" есть и другие строки. В строке "Символы для пробелов" мы можем указать символ, который будет выводиться маской в том месте, где пользователь должен будет что-то ввести. По умолчанию, это символ подчеркивания "_". Пользователь будет видеть пустую маску телефона, как

```
___-__-__
```

Мы можем использовать и другие символы для заполнения пробелов.

Ниже в Редакторе масок есть флажок "Сохранить литерал". Этот флажок включает или выключает возможность сохранения в тексте символов-разделителей. Если флажок включен, то в тексте сохранятся символы маски вместе с символами-разделителями, например:

```
123-45-67
```

Если флажок выключен, то в конечном тексте останутся только символы маски:

```
1234567
```

В строке "Тестовый ввод" мы можем опробовать полученную маску, не

запуская программу на выполнение.

Итак, для нашего примера в строке "Маска ввода" укажите

```
000-00-00
```

В "Символы для пробелов" оставьте "_". Флажок "Сохранять литерал" пусть будет включен. Как только нажмете "OK", свойство `EditMask` изменится, теперь в нем будет текст:

```
000-00-00;1;_
```

Как видите, значение этого свойства состоит из трех частей, разделенных точкой с запятой. Первая часть - это указанная нами маска. Во второй части может быть либо `0` (флажок "Сохранять литерал" выключен), либо `1` (флажок включен). В третьей части маски указывается символ для пробелов.

Таким образом, мы могли бы и не пользоваться Редактором масок, а указать маску вручную. Например, введя значение

```
00-00-0000 г.;1;_
```

мы создадим маску, где пользователь будет обязан ввести все цифры месяца, дня и года, где в тексте сохранятся символы-разделители "-", и где на месте пробелов будет знак подчеркивания. В результате, мы получим дату, например в таком виде: "`11-11-2013 г.`".

Стандартные строковые функции и сообщения

В лекции приведены основные стандартные функции для обработки строк, а также функции-сообщения и функция-запрос.

Цель лекции

Расширение возможностей обработки строк с помощью стандартных функций, вывод сообщений в окнах различного типа, получение от пользователя данных с помощью функции-запроса.

Функции для работы со строками

В прошлой лекции мы достаточно подробно познакомились со строковыми и символьными типами данных, изучили три компонента, с помощью которых пользователь может вводить в программу строки. Однако этих инструментов зачастую бывает недостаточно для обработки строк. В практике программирования над строками то и дело приходится совершать какие-то действия: сравнивать строки между собой, преобразовывать строку в строчные или в прописные буквы, находить часть строки и т.п. Сегодня мы разберем наиболее востребованные строковые функции.

А для того, чтобы посмотреть работу функций на практике, создадим новое приложение. Это будет простое приложение с единственной кнопкой посреди окна. Сохраните проект в папку 06-01, поскольку проект будет совсем простой, имена модуля и проекта можно оставить без изменений. Сгенерируйте событие нажатия на кнопку, в этом событии мы и будем экспериментировать над строками.

Объединение (конкатенация) строк

Программисту довольно часто приходится в коде объединять несколько строк в одну. Мы уже делали подобное объединение в прошлых лекциях. Для объединения служит оператор "+":

```
string_1 + string_2 + …string_n;
```

В качестве строки можно использовать и отдельные символы, например, символ перехода на другую строку. Напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Строка №1' + #13 + 'Строка №2' + #13 + 'Строка №3';
  ShowMessage(s1);
end;
```

Здесь, в переменной `s1` мы собрали строку из 5 кусочков, причем два из них - символы перехода на новую строку. Сохраните проект, скомпилируйте его и убедитесь, что полученное сообщение будет трехстрочным.

Функция `Concat()` делает то же самое - объединяет строки. Её синтаксис такой (в квадратные скобки принято помещать необязательные параметры функций):

```
Concat(S1 [, S2,…Sn]);
```

Здесь S1, S2 и т.д. - строки. Предыдущий пример можно было бы запис

```
s1:= Concat('Строка №1', #13, 'Строка №2', #13, 'Строка №3');
```

Длина строки

Поскольку мы предоставляем пользователям возможность вводить различные строки, нередко возникает необходимость выяснить длину этих самых строк. Для этого существует функция `Length()`. Её синтаксис:

```
Length(S);
```

где `S` - строка. Функция возвращает нам размер количество символов в этой строке. Однако в Lazarus не все так просто. Давайте изменим предыдущий код обработчика кнопки на следующий:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Hello';
  ShowMessage(IntToStr(Length(s1)));
end;
```

Поскольку результатом работы функции `Length()` будет целое число - количество символов, то для вывода этих данных на экран нам приходится преобразовать полученное число в строковый тип, что мы и делаем в коде

```
ShowMessage(IntToStr(Length(s1)));
```

Подобные преобразования мы будем изучать позже, пока лишь нужно понять вот что: данная строка кода реализует сложное для начинающих программистов выражение, которое выполняется компьютером в три этапа:

1. `Length()` - вычисление размера строки `s1`.
2. `IntToStr()` - преобразование полученного в №1 результата из целого числа в строку.
3. `ShowMessage()` - вывод преобразованного в №2 результата на экран.

Если вы откомпилируете проект и запустите его на выполнение, то получите результат: 5. Столько символов в строке "`Hello`". Однако измените эту строку на "`Привет`" и снова выполните программу:

```
s1:= 'Привет';
ShowMessage(IntToStr(Length(s1)));
```

Мы ожидаем получить число 6 - количество символов в слове "Привет", но неожиданно получаем 12! Дело в том, что функция работает с ANSI-строками, то есть с теми, что занимают по 1 байту на символ. А кириллица - это не ANSI а UTF8, и требует по 2 байта на символ! Как быть? В подобных случаях используйте UTF8-аналоги строковых функций. То есть, вместо `Length()` используйте `UTF8Length()`. Эта функция гарантированно вернет количество символов в строке, на каком бы языке эти символы не вводились. Но для использования `UTF8Length()` нужно подключить модуль `LCLProc`, где и реализованы все UTF8-функции.

Пролистайте код выше, на начало модуля, и добавьте `LCLProc` в раздел uses:

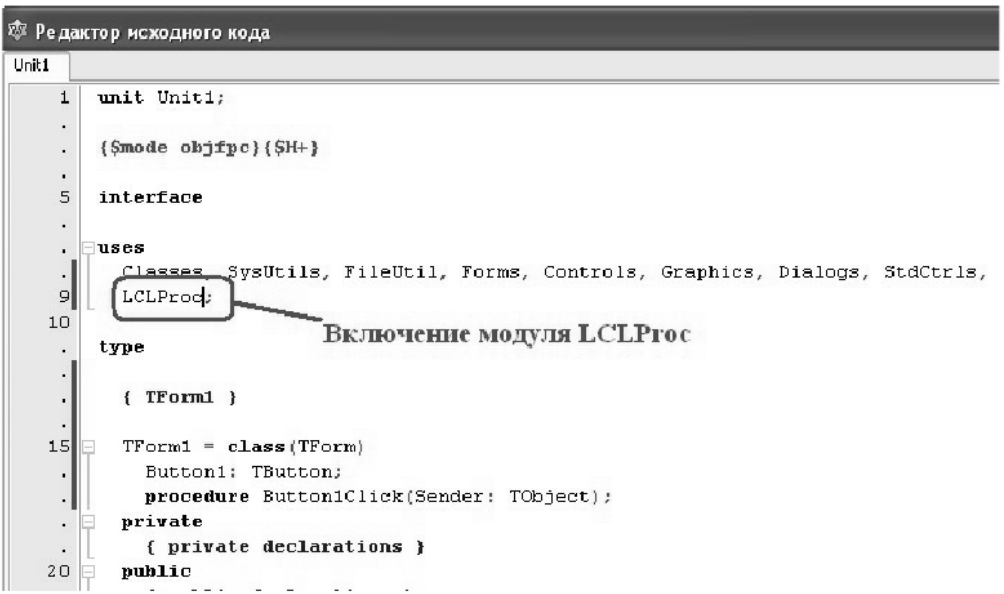


Рис. 6.1. Добавление модуля в раздел uses

Теперь мы можем изменить наш обработчик:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Привет';
```

```
  ShowMessage(IntToStr(UTF8Length(s1)));
end;
```

Вот теперь мы получим в результате число 6.

Таким образом, если вы точно знаете, что будете иметь дело только с английскими символами, то для определения количества символов в строке, и занимаемых этой строкой байт, используйте функцию `Length()`. Во всех остальных случаях `Length()` можно использовать для получения размера строки в байтах, а `UTF8Length()` - для получения количества символов в строке.

Поиск в строке

Иногда необходимо выяснить, есть ли в строке нужный текст. Для этого служат функции

- `Pos()` - для ANSI-символов
- `UTF8Pos()` - для кириллицы (UTF8-символы) - требует подключения модуля `LCLProc`

Их синтаксис:

```
Pos(Substr, Str);
UTF8Pos(Substr, Str);
```

Здесь `Substr` - искомый текст; `Str` - строка, в которой ищется текст. Функции возвращают в качестве результата целое число. Если искомый текст в строке отсутствует, обе функции вернут ноль. Иначе - номер символа строки, с которого начинается найденная подстрока. Переделаем код обработчика кнопки (модуль `LCLProc` у нас уже подключен):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
```

```
  s1:= 'Привет';
  ShowMessage(IntToStr(UTF8Pos('ив', s1)));  //результат: 3
end;
```

Запустив программу на выполнение, убедимся, что функция

```
UTF8Pos('ив', s1)
```

возвращает результат 3 - номер первого символа искомой подстроки в строке "`Привет`".

Если в строке содержится несколько искомых подстрок, функции вернут номер первого такого вхождения. А если для строки с русскими буквами использовать функцию `Pos()` вместо `UTF8Pos()`, результат будет неверный.

Получение подстроки

Чтобы из строки получить ее часть (подстроку), применяют функции

- `Copy()` - для ANSI-символов
- `UTF8Copy()` - для кириллицы - требует подключения модуля `LCLProc`

Их синтаксис:

```
Copy(Str, StartCharIndex, Count);
UTF8Copy(Str, StartCharIndex, Count);
```

Здесь `Str` - строка исходного текста; `StartCharIndex` - номер первого символа подстроки; `Count` - количество символов подстроки. Снова переделаем обработчик кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
```

```
begin
  s1:= 'Привет';
  ShowMessage(UTF8Copy(s1, 3, 2));
end;
```

В результате выполнения программы вы убедитесь, что

```
UTF8Copy(s1, 3, 2)
```

из строки "Привет" вернет "ив" - подстроку, начиная с третьего символа, размером два символа.

Если для кириллицы вместо `UTF8Copy()` использовать `Copy()`, результат будет неверным.

Удаление части строки

Для удаления части строки применяют функции `Delete()` и `UTF8Delete()`.

Их синтаксис:

```
Delete(Str, StartCharIndex, Count);
UTF8Delete(Str, StartCharIndex, Count);
```

Здесь `Str` - строка исходного текста; `StartCharIndex` - номер первого символа удаляемой подстроки; `Count` - количество удаляемых символов. Эта функция отличается от других тем, что она изменяет исходную строку `Str`, обрезает ее. Другие функции оставляли эту строку без изменений. Итак, наш новый код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Привет';
```

```
  UTF8Delete(s1, 3, 2); //обрезаем строку
  ShowMessage(s1);
end;
```

Функцией `UTF8Delete(s1, 3, 2)` мы ампутировали строку `s1`, вырезали из нее, начиная с третьего символа, подстроку размером 2 символа. В результате у нас осталась строка "`Прет`". Как и с предыдущими функциями, применение с кириллицей не UTF8-варианта исказит результат.

Преобразование символов строки в строчные и в заглавные

Иногда требуется сравнить две строки, или найти какую-то подстроку. Но не всегда известно, в каком регистре пользователь ввел текст. К примеру, нам нужно узнать, ввел ли пользователь слово "Москва", или что-то другое. Пользователь знает, что он должен ввести название нашей столицы, однако он может полениться нажать <Shift>, и в результате введет "москва". Или нажмет <Caps Lock> и введет "МОСКВА". А поскольку слово он написал без ошибок, то уверен, что все сделал правильно. Но мы-то ожидаем не просто слово, а слово со строго определенным описанием! К сожалению, пользователи далеко не всегда вводят то, что нужно, и программистам приходится идти на всевозможные хитрости, чтобы добиваться нужного результата. Например, мы можем заранее преобразовать введенное пользователем слово в верхний регистр (то есть, сделать все буквы заглавными), и сравнить его со словом "МОСКВА". И какие бы буквы пользователь первоначально не вводил, результат все равно будет верным.

Точно также, мы можем все буквы сделать строчными, и сверить строку с текстом "москва". Для подобных преобразований служат следующие функции:

`UpperCase(Str)`	- Преобразует ANSI-строку `Str` в верхний регистр
`UTF8UpperCase(Str)`	- Преобразует UTF8-строку `Str` в верхний регистр

`LowerCase(Str)`	- Преобразует ANSI-строку `Str` в нижний регистр
`UTF8LowerCase(Str)`	- Преобразует UTF8-строку `Str` в нижний регистр

Изменим код обработчика:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s1: String;
begin
  s1:= 'Москва';
  ShowMessage(UTF8UpperCase(s1));
  ShowMessage(UTF8LowerCase(s1));
end;
```

Сохраните проект и запустите его на выполнение. Мы получим два сообщения: заглавными и строчными буквами.

Функции-сообщения

Нам то и дело требуется выводить пользователю различные сообщения. И в этой лекции, и в предыдущих лекциях, мы уже неоднократно пользовались одной такой функцией - `ShowMessage()`. Её синтаксис очень простой:

```
ShowMessage('Текст сообщения');
```

В результате будет выведено окно с сообщением, которое не даст пользователю продолжать работу, пока он не нажмет кнопку "OK":

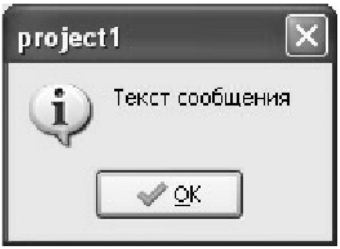


Рис. 6.2. Сообщение ShowMessage()

Application.MessageBox()

Еще один вариант сообщения - функция `Application.MessageBox()`. Данная Windows API-функция позволяет вывести сообщение с разными настройками. Синтаксис функции следующий:

```
Application.MessageBox(Text: PChar; Caption: PChar; Flags: LongInt): Integer;
```

В качестве аргументов мы должны указать текст сообщения `Text`, заголовок окна с сообщением `Caption` и флаги - значок сообщения и используемые кнопки. И `Text`, и `Caption` имеют тип `PChar`. Флаги имеют целочисленный тип, однако нам нужно запомнить только символьные обозначения этих флагов:

Таблица 6.1. Флаги функции Application.MessageBox()

Значки сообщения	
`MB_ICONERROR`	Белый крестик в красном круге. Такой значок обычно используют в сообщениях об ошибке.
`MB_ICONHAND`	
`MB_ICONSTOP`	
`MB_ICONQUESTION`	Синий знак вопроса в белой выноске. Таким способом помечают вопрос, обращенный к пользователю.
`MB_ICONWARNING`	Черный восклицательный знак в желтом треугольнике. Таким знаком привлекают внимание пользователя к вероятной

	опасности, которая может последовать в результате действий пользователя.
MB_ICONINFORMATION	Синяя буква "i" в белой выноске. Таким способом помечают какую-то информацию для пользователя.
MB_ICONASTERICK	
Кнопки сообщения	
MB_OK	Кнопка "OK" в середине окна.
MB_OKCANCEL	Кнопки "OK" и "Cancel".
MB_ABORTRETRYIGNORE	Кнопки "Abort", "Retry" и "Ignore".
MB_YESNOCANCEL	Кнопки "Yes", "No" и "Cancel".
MB_YESNO	Кнопки "Yes" и "No".
MB_RETRYCANCEL	Кнопки "Retry" и "Cancel".

Примечание: Чтобы пользоваться функцией `Application.MessageBox()`, нужно в разделе uses подключить модуль `LCLType`.

Как видно из таблицы, некоторые значки сообщений совпадают. Указывать нужно какое-то одно из них. Переделаем код обработчика кнопки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.MessageBox('Сообщение', 'Заголовок',
                MB_ICONINFORMATION + MB_ABORTRETRYIGNORE
end;
```

В результате получим такое окно сообщения:

Рис. 6.3. Сообщение Application.MessageBox()

Вы заметили, что в качестве флага используется сумма значка и кнопок? И то, и другое, на самом деле, целое число. Но нам удобней запомнить эти значения в символьном варианте.

А как мы узнаем, какую кнопку нажал пользователь? Эта функция возвращает значение - кнопку, нажатую пользователем, которая может быть:

- `IDOK`
- `IDCANCEL`
- `IDABORT`
- `IDRETRY`
- `IDIGNORE`
- `IDYES`
- `IDNO`
- `IDCLOSE`
- `IDHELP`

То есть, если пользователь нажал кнопку "Abort", функция вернет значение `IDABORT`. А раз так, мы можем делать проверку на нажатую кнопку. Снова изменим код обработчика:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Application.MessageBox('Сообщение', 'Заголовок', MB_ICONQUESTIC
    MB_YESNOCANCEL) = IDYES then ShowMessage('Вы нажали кнопк
end;
```

Логическую конструкцию `if` мы будем изучать в следующей лекции, здесь лишь заметим, что сообщение `ShowMessage()` будет выведено только в том случае, если пользователь нажал кнопку "Yes". Во всех остальных случаях это сообщение не выйдет.

MessageDlg()

Похожим образом действует Windows API-функция `MessageDlg()`, которая описана в модуле `Dialogs`. Этот модуль включается в раздел `uses` автоматически, поэтому нам не нужно даже что-то туда добавлять. Синтаксис этой функции несколько отличается от предыдущей:

```
MessageDlg(Caption, Text, MessageType, MessageButton, HelpKeyword): Inte
```

Здесь `Caption` и `Text` соответственно, текст заголовка и самого сообщения, тип `PChar`. Далее идет тип сообщения `MessageType`, - значок сообщения, который может быть:

`mtWarning`	- Восклицательный знак, подобен `MB_ICONWARNING` функции `Application.MessageBox()`
`mtError`	- крестик
`mtInformation`	- буква "i"
`mtConfirmation`	- вопросительный знак
`mtCustom`	- пользовательское окно без значка.

Далее следует тип кнопок `MessageButton`, который может быть:

```
mbYes
mbNo
mbOK
mbCancel
mbAbort
mbRetry
mbIgnore
mbAll
mbNoToAll
mbYesToAll
mbHelp
mbClose
```

Названия кнопок говорят сами за себя, комментарии тут излишни. Кнопки указывают в квадратных скобках, отделяя друг от друга запятыми, например: `[mbYes, mbNo, mbIgnore]`.

Последний параметр `HelpKeyword` определяет экран контекстной справки, которая будет появляться, если пользователь нажмет <F1>. Обычно в этом параметре указывают значение `0` - нет справки.

Функция `MessageDlg()` возвращает значение - нажатую кнопку, которая соответствует типу кнопок, указанному выше. Возвращаемое значение вместо "`mb`" начинается на "`mr`", то есть, если пользователь нажал кнопку "Yes" (тип `mbYes`), то будет возвращено значение `mrYes`.

Снова изменим код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if MessageDlg('Подтверждение', 'Вы действительно хотите закрыть про
        mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then Close;
end;
```

В данном случае пользователю выводится запрос на подтверждение закрытия программы, и если он нажимает кнопку "Yes", программа закрывается (выполняется оператор `Close`). Так как функции `MessageDlg()` и `Application.MessageBox()` похожи, выбирайте любую, на свой вкус.

Функция-запрос

Иногда требуется получить от пользователя какие-то данные (вспомните программу Hello из второй лекции). В этом случае можно использовать компонент `TEdit`, но можно поступить проще - воспользоваться функцией `InputQuery()`.

Функция `InputQuery()` выводит окно запроса. Синтаксис функции такой:

```
InputQuery(Caption, Message, StrVar);
```

Здесь, `Caption` и `Message` - соответственно, текст заголовка и текст сообщения внутри окна. `StrVar` - переменная строкового типа, которая должна быть объявлена заранее. Если в этой переменной есть текст, он выводится в строке редактирования как текст по умолчанию. Если переменная пуста, то пуста будет и строка. Если пользователь что-то введет в строку и нажмет кнопку "OK", этот текст запишется в переменную `StrVar`, а функция вернет значение `True` (Истина). В противном случае функция вернет значение `False` (Ложь). Переделаем код обработчика:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  YourName: String;
begin
  YourName:= 'Неизвестный';
  if InputQuery('Кто вы?', 'Укажите ваше имя', YourName) then
    ShowMessage('Привет, ' + YourName + '!');
end;
```

Напомню, что с логической конструкцией `if` мы познакомимся в следующей лекции, а пока разберем код. Вначале мы присваиваем строковой переменной `YourName` текст "`Неизвестный`". Затем мы вызываем функцию `InputQuery()`. Мы указываем заголовок окна "Кто вы?", текст сообщения "Укажите ваше имя" и текстовую переменную `YourName`. Текст, который мы ранее поместили в эту переменную, будет выходить в строке редактирования:

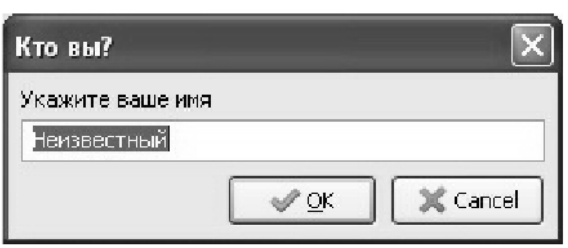


Рис. 6.4. Результат работы функции InputQuery()

Если пользователь введет какой-то другой текст, он попадет в переменную `YourName`, функция вернет `True` и в результате будет выведено сообщение с приветствием

```
ShowMessage('Привет, ' + YourName + '!');
```

Иначе приветствие не будет выведено. Функция `InputQuery()` часто бывает полезна, так что запомните её получше.

Логические типы, конструкции и компоненты

В лекции представлен исчерпывающий материал по работе с логическими данными. Существующие логические типы, операции над ними, а также компоненты TCheckBox, TCheckGroup, TRadioButton, TRadioBox. Примеры программ демонстрируют все возможности работы с логическими типами.

Цель лекции

Изучение способов работы с логическими типами данных. Изучение основных компонентов для работы с этими типами.

Логический тип данных

В предыдущих лекциях мы уже встречались с таким типом - некоторые свойства компонентов могли принимать только одно из двух значений: либо `True` (Истина), либо `False` (Ложь). Это и есть логический тип данных. В Паскале это тип `Boolean`, который занимает один байт оперативной памяти. В Lazarus, кроме того, есть типы `ByteBool` (тоже 1 байт), `WordBool` (2 байта) и `LongBool` (4 байта). Все эти типы могут принимать лишь одно из двух значений: `true` или `false`. На практике, правда, такое многообразие не применяется, программисты вполне обходятся только одним стандартным типом `boolean`. Вы можете создать переменную такого типа и присвоить ей `true` или `false`. Кроме того, нередко встречаются выражения, которые в результате тоже дают либо Истину, либо Ложь, они также считаются логическими. Например, если вы присвоите логической переменной выражение `3>2`, то в переменную попадет `True`, так как три действительно, больше двух.

Итак, что же можно делать с логическими данными, помимо присваивания им значений `True` или `False`? Их можно сравнивать между собой, используя для этого операторы сравнения:

= (Равно)

<> (Не равно)

> (Больше) - Здесь следует иметь в виду, что компилятор, на самом деле, вместо `False` записывает двоичное значение 0, а вместо `True` - 1. Таким образом, True всегда будет больше, чем `False`.

< (Меньше)

>= (Больше или равно)

<= (Меньше или равно)

Операции сравнения можно усложнить, применив следующие дополнительные логические операторы:

- `NOT` (Логическое НЕ)
- `AND` (Логическое И)
- `OR` (Логическое ИЛИ)
- `XOR` (Логическое исключающее ИЛИ).

Эти операторы сравнивают между собой два логических значения (операнда) и в зависимости от этих значений возвращают в качестве результата `true` или `false`. Следующая таблица демонстрирует результат этих операций (сравниваются логические переменные `a` и `b`):

Таблица 7.1. Логические операции

A	B	not A	A and B	A or B	A xor B
`true`	`true`	`false`	`true`	`true`	`false`
`false`	`true`	`true`	`false`	`true`	`true`
`true`	`false`	`false`	`false`	`true`	`true`
`false`	`false`	`true`	`false`	`false`	`false`

Давайте разберем действия этих операций подробней.

Операция `NOT` (НЕ), в отличие от других логических операций, работает только с одним операндом. Это - операция-перевертыш: если операнд содержит `True`, то возвращаемый результат будет `False`, и наоборот. Эту операцию очень часто применяют для того, чтобы поменять значение логической переменной (или свойства) на противоположное, например,

```
A:= not A;
```

Операция AND (И) возвращает True только в том случае, если оба сравниваемых операнда содержат True. Если первый операнд содержит False, дальнейшая проверка уже не проводится (смысла нет), и AND возвращает False. Если первый операнд содержит True, то AND делает проверку второго операнда - если он также содержит True, то и возвращаемый результат будет True, иначе возвращается False.

Операция OR (ИЛИ) действует похожим образом, но возвращает True в том случае, если хоть один из операндов содержит True. OR проверяет первый операнд. Если он True, дальнейшая проверка уже не производится, и OR возвращает True. Иначе OR проверяет второй операнд - если он True, то и возвращается True, в противном случае возвращается False.

Операция XOR (Исключающее ИЛИ) всегда проверяет оба операнда, и возвращает True только тогда, когда один из операндов True, а другой обязательно False. Правда, на практике XOR обычно не используют, поскольку всегда проще выполнить проверку A <> B, которая возвращает такой же точно результат.

Итак, примеры работы с логическими данными (в Lazarus их выполнять не нужно):

```
var
  A, B, C: Boolean;
begin
  A:= True;   //результат - True
  B:= not A;   //результат - False
  C:= A and B;  //результат - False
  C:= A or B;   //результат - True
  C:= A <> B;   //результат - True
  C:= 3 > 5;   //результат - False
  …
```

Управляющая конструкция IF

От логического типа данных было бы мало пользы, если бы мы не могли применять различные действия в зависимости от результатов проверки - если в переменной находится Истина, тогда выполнить один код, иначе выполнить другой код. Именно такую возможность предоставляет управляющая конструкция `IF`.

Простейший синтаксис этой конструкции такой:

```
if Условие then Действие;
```

Английское IF переводится как ЕСЛИ, а THEN - ТОГДА. Тут все довольно понятно: если какое-то условное выражение (логическая переменная или логическое свойство компонента) имеет значение `True`, тогда выполняется указанное действие. Теперь мы можем изучать примеры на практике. Загрузите Lazarus с новым проектом. Посреди окна установите кнопку TButton. Сохраните проект в папку 07-01 под именем MyBool; не забудьте главную форму назвать fMain, а сохраняемый модуль - Main. Затем сгенерируйте событие нажатия на кнопку, которое оформите следующим образом:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  b: boolean;
begin
  b:= True;
  if b then ShowMessage('Истина');
end;
```

Здесь все предельно просто: объявляем логическую переменную `b`, затем присваиваем ей значение `True` (Истина). В строке

```
  if b then ShowMessage('Истина');
```

мы проверяем: если `b` имеет значение `True` (то есть, если `b` истинна),

тогда мы выводим сообщение "Истина". Сохраните и запустите проект. Нажатие на кнопку вызовет вывод сообщения. Можете для примера присвоить `b` значение `False` - в этом случае нажатие на кнопку не вызовет никакого сообщения.

Кстати, в данном случае в качестве условия мы указали просто имя переменной `b`. Ведь она уже содержит значение `True`! Так что строка

```
if b then ShowMessage('Истина');
```

будет полностью аналогична строке с полным условием

```
if b = True then ShowMessage('Истина');
```

Если мы будем расширять условие с помощью условных операций, то все операнды следует заключить в скобки. Исключение составляет операция `NOT`, которая работает только с одним операндом. Изменим код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  a,b: boolean;
begin
  a:= False;
  b:= True;
  if (a = b) or b then ShowMessage('Истина');
end;
```

Здесь мы проверяем уже два условия. И если первое `(a = b)` вернет нам `False`, то второе `b` вернет `True`. И так как мы используем операцию `OR`, нам достаточно, чтобы хоть один из операндов был `True`. Значит, сообщение будет выполнено. Если бы во втором операнде мы сравнивали `b` еще с чем-то, то его тоже следовало бы поместить в скобки. Например,

```
if (a = b) or (b > a) then ShowMessage('Истина');
```

В этом примере сообщение "Истина" будет выведено, так как во втором операнде `b` действительно больше, чем `a`. Если вы забудете про скобки, компилятор не станет указывать вам на ошибку, и программа будет запущена. Однако результат может оказаться неверным. Например, если мы уберем скобки в первом примере:

```
if a = b or b then ShowMessage('Истина');
```

То сообщение "Истина" не будет выполнено. Ведь оператор `IF` делает только одну проверку, значит, он проверит только `a = b`. Часть условия с `OR` будет пропущена, потому что мы не заключили операнды в скобки. Так что если вы в условии применяете операции `AND`, `OR` или `XOR`, не забывайте заключать операнды в скобки. Скобки можно не использовать, если в качестве операндов вы указываете по одной условной переменной или свойству, например, код:

```
if a  or b then ShowMessage('Истина');
```

выведет сообщение "Истина", так как в `b` содержится `True`.

Управляющий оператор `IF` не зря называют конструкцией, его синтаксис может состоять из нескольких частей:

```
if Условие then Действие1
else Действие2;
```

Английское ELSE переводится как ИНАЧЕ. Здесь, ЕСЛИ какое-то условие вернет `True`, то будет выполнено Действие1, ИНАЧЕ будет выполнено Действие2. Обратите внимание, что перед `ELSE` точку с запятой НЕ СТАВЯТ! Изменим код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  b: boolean;
begin
  b:= True;
  if b then ShowMessage('Истина')
```

```
  else ShowMessage('Ложь');
end;
```

Нажатие на кнопку вызовет сообщение "Истина", однако если затем вы исправите код, и в переменную `b` вместо `True` поместите `False`, то выйдет сообщение "Ложь" - проверьте это сами.

Однако конструкция `IF` может быть и сложней. Что, если нам нужно проверить не одно, а два условия? Или больше? Полный синтаксис этой конструкции такой:

```
if Условие1 then Действие1
else if Условие2 then Действие2
…
else if УсловиеN then ДействиеN
else ДействиеИНАЧЕ;
```

Давайте разбираться. В этой конструкции, если `Условие1` вернет `True`, то будет выполнено `Дествие1`, и на этом конструкция закончится. Если же оно вернет `False`, проверка будет продолжена. Если следующее `Условие2` вернет `True`, то будет выполнено `Дествие2`, и так далее, подобных проверок может быть сколько угодно. Если же все проверки условий вернут `False`, то будет выполнено `ДействиеИНАЧЕ`. Изменим код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  a,b: boolean;
begin
  a:= False;
  b:= True;
  if a then ShowMessage('Действие №1')
  else if b then ShowMessage('Действие №2')
  else if a or b then ShowMessage('Действие №3')
  else ShowMessage('Действие №4');
end;
```

Как вы думаете, какое сообщение выйдет в данном случае? И сколько сообщений выйдет? Если вы сказали, что выйдет только одно сообщение "Действие №2", то вы усвоили конструкцию IF. Иначе советую вернуться назад, и внимательней изучить всё вышесказанное.

Операторские скобки BEGIN…END

Логический оператор `IF`, как и многие другие операторы, которые нам еще предстоит изучить, выполняет только одно действие. Взгляните на следующий код:

```
if a then
  ShowMessage('Действие №1');
  ShowMessage('Действие №2');
  ShowMessage('Действие №3');
```

В этом примере сообщение "Действие №1" выйдет только в том случае, если `a = True`. Остальные сообщения выйдут в любом случае, так как они к оператору `IF` уже не относятся, а выполняются как самостоятельные операторы. Однако что делать, если в рамках конструкции `IF` нам нужно выполнить не один оператор, а целый блок кода? В таком случае, этот блок помещают в операторские скобки `begin…end`, в результате чего блок операторов выполняется, как единый оператор. Изменим код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  a: boolean;
begin
  if a then begin
    ShowMessage('Действие №1');
    ShowMessage('Действие №2');
    ShowMessage('Действие №3');
  end; //конец if
end;
```

В данном примере все три сообщения заключены в скобки `begin…end`

и выполняются, как один оператор. Значит, они все будут выведены, только если `a = True`, иначе не будет выведено ни одно сообщение.

Здесь следует сказать пару слов о стиле программирования. Единых стандартов тут нет, есть только рекомендации.

Во-первых, код должен быть удобочитаем. Сравните:

```
if a then ShowMessage('Действие №1') else ShowMessage('Действие №2');
```

Синтаксис здесь правильный, компилятор ошибок не найдет. И код этот будет выполнен верно, однако как же его сложно читать! Все части конструкции `IF` сливаются в одну строку, сразу и не разберешь, что тут к чему, а ведь конструкция еще очень простая! Что, если бы мы применяли несколько операций `AND` или `OR`, да еще с множеством операндов? Совсем иначе смотрится такой код:

```
if a then
ShowMessage('Действие №1')
else
ShowMessage('Действие №2');
```

Здесь конструкция разбита на несколько небольших логических частей, каждой части отведена отдельная строка. Код стал явно более понятным, как говорят, более удобочитаемым. Однако его можно еще улучшить, если сдвинуть вложенные операторы вправо на 2-3 пробела, чтобы показать зависимости:

```
if a then
   ShowMessage('Действие №1')
else
  ShowMessage('Действие №2');
```

Хороший программист именно так и оформляет свой код. Тут сразу видно, что от чего зависит, конструкция становится понятней.

Во-вторых, если вы применяете операторские скобки, то в учебной

литературе вы можете встретить два способа оформления кода. Такой:

```
if a then begin
  ShowMessage('Действие №1');
  ShowMessage('Действие №2');
  ShowMessage('Действие №3');
end; //конец if
```

и такой:

```
if a then
begin
  ShowMessage('Действие №1');
  ShowMessage('Действие №2');
  ShowMessage('Действие №3');
end; //конец if
```

Оба этих способа считаются классическими, оба хорошо смотрятся, какой из них применять - ваш выбор. Лично я не люблю переносить `BEGIN` на отдельную строку (разве что строка получается слишком длинной), поэтому предпочитаю первый способ. Вы, если захотите, можете выбрать и второй вариант, он тоже правильный.

Флажки и радиокнопки

В программах, особенно в настройках, нередко используют флажки (которые иначе называют галочками) и радиокнопки. Флажки `TCheckBox` имеют логическую зависимость включен/выключен, поэтому изучим их в этой лекции. Радиокнопки TRadioButton также имеют такую зависимость. И флажок, и радиокнопка находятся на вкладке Standard Палитры компонентов:

Рис. 7.1. TCheckBox и TRadioButton

Установите на форму один флажок. Посмотрим на его свойства.

`Name` задает имя компонента. `Caption` - пояснительный текст справа от флажка. Самое главное свойство для нас - `Checked` (англ. checked - проверенный). Оно имеет логический тип. Если в этом свойстве значение `True`, то флажок включен, иначе он выключен. Значение меняется автоматически, хотя его можно изменить и программно.

Свойство `Name` изменим на `ChB1`, чтобы проще к нему обращаться. В свойстве `Caption` укажите текст

```
Наш флажок
```

Вы можете заметить, что по мере ввода текста в свойство `Caption` ширина компонента увеличивается, чтобы уместился весь текст. Теперь изменим код обработки нажатия на кнопку:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  if ChB1.Checked then
    ShowMessage('Наш флажок включен!')
  else
    ShowMessage('Наш флажок выключен!');
end;
```

Вы и без комментариев должны понять, что если в свойстве `Checked`

нашего флажка `ChB1` содержится значение `True`, то выйдет одно сообщение, иначе - другое. Сохраните проект и запустите его на выполнение. Включайте или выключайте флажок, после чего нажимайте кнопку. Сообщение скажет, включен флажок, или выключен.

`State` (состояние) - еще одно полезное свойство флажка. Это свойство определяет первоначальное значение флажка, кроме того, можно проверить его и в процессе работы программы. Может иметь следующие значения:

- `cbChecked` - включен
- `cbGrayed` - неопределен (от англ. gray - серый)
- `cbUnchecked` - выключен

Еще нас могут интересовать два события, которые доступны на вкладке События Инспектора объектов.

`OnChange` - При изменении. Событие возникает каждый раз, когда пользователь включает или выключает флажок. Вы можете сгенерировать это событие так же, как и событие нажатия на кнопку - дважды щелкнув по нему в Инспекторе объектов. Затем можете скопировать код из события кнопки, а событие кнопки очистить:

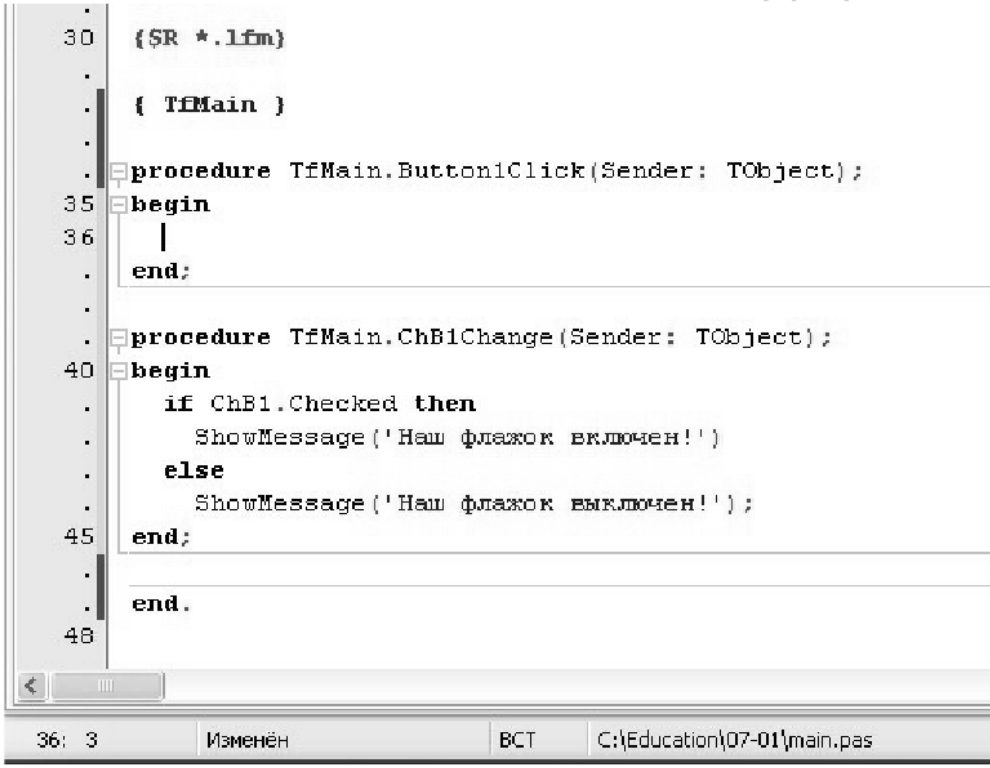


Рис. 7.2. Событие OnChange флажка

Теперь сообщения будут выходить сразу же при изменении состояния флажка. Кнопка же никаких действий совершать не будет. Однако не спешите ее удалять, она еще понадобится.

`OnClick` - событие также возникает, когда пользователь щелкнет мышью по компоненту, включив флажок, или выключив его.

Разница между этими событиями в том, что `OnChange` будет выполнен в любом случае - если пользователь мышью включил/выключил флажок, или если мы сделали это программно, например, вписав в событие кнопки `Button1Click` (между `begin` и `end`) код:

```
ChB1.Checked:= not ChB1.Checked;
```

Тогда при нажатии на кнопку состояние флажка изменится на противоположное - сработал наш перевертыш `NOT`. `OnChange` при

этом сработает, а `OnClick` - нет, ведь мышью по компоненту не щелкали!

Теперь займемся радиокнопками. Если флажок можно использовать в единичном варианте, то радиокнопок должно быть как минимум две. Радиокнопки позволяют пользователю выбрать один из возможных вариантов, но для выбора одной такой кнопки будет явно недостаточно.

Давайте сымитируем выборы президента, установив 5 таких радиокнопок одну под другой. Свойства `Name` этих кнопок переименуем, соответственно, в `RB1`, `RB2`, `RB3`, `RB4` и `RB5`. Ну, а в свойствах `Caption` этих кнопок запишем, соответственно

Путин
Медведев
Жириновский
Зюганов
Прохоров

Теперь нам нужно включить одну из этих радиокнопок. Поскольку нынешний президент у нас Путин, переведем в `True` свойство `Checked` радиокнопки `RB1`. В результате у нас получится такая форма:

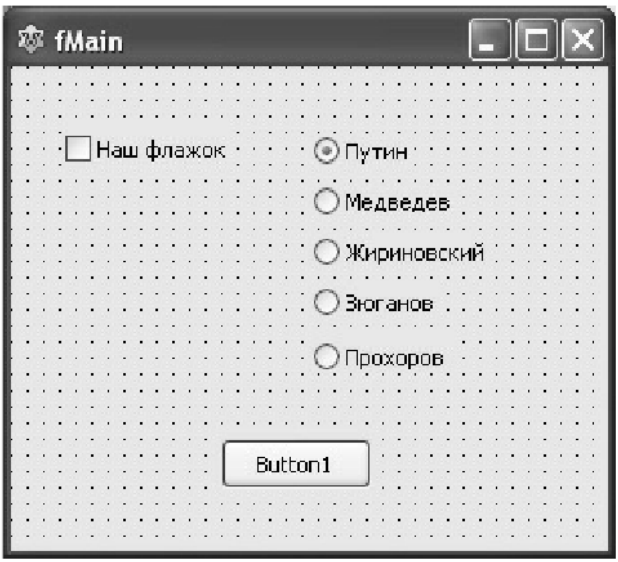


Рис. 7.3. Флажок и радиокнопки

Теперь изменим код нажатия на кнопку:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  if RB1.Checked then
    ShowMessage('Вы выбрали Путина')
  else if RB2.Checked then
    ShowMessage('Вы выбрали Медведева')
  else if RB3.Checked then
    ShowMessage('Вы выбрали Жириновского')
  else if RB4.Checked then
    ShowMessage('Вы выбрали Зюганова')
  else
    ShowMessage('Вы выбрали Прохорова');
end;
```

Тут все достаточно прозрачно, обработка радиокнопок практически не отличается от обработки флажков. Единственное различие: пользователь не сможет отметить несколько радиокнопок, только одну из них. А флажки могут быть включены все, некоторые, или ни один. Раз уж мы создаем имитацию выборов, неплохо бы и в свойстве `Caption` формы написать Выборы. Сохраните проект, запустите на выполнение и играйте на здоровье в выборы президента.

Контейнеры для флажков и радиокнопок

Можно конечно, пользоваться и единичными компонентами `TCheckBox` и `TRadioButton`, однако куда проще использовать для этого контейнеры, особенно когда флажков и радиокнопок много. Для флажков используется контейнер `TCheckGroup`, а для радиокнопок - `TRadioGroup`. Оба компонента располагаются на вкладке Standard Палитры компонентов. Чтобы не мучаться с изменением дизайна и кода нашего проекта, будет проще этот проект закрыть, и создать новый.

Свойство `Name` формы переименуйте сразу в `fMain`, в свойстве `Caption` впишите текст

Свойство `BorderStyle` имеет смысл перевести в `bsDialog`, а `Position` - в `poDesktopCenter`. Далее, сохраните проект под именем election в папку 07-02, модулю главной формы дайте имя Main.

Теперь можем заняться контейнерами:

Рис. 7.4. Контейнеры для радиокнопок и флажков

Слева на форму установите контейнер для радиокнопок `TRadioGroup`. Выделите его, и измените свойство `Name` на `RG1`. Свойство `Caption` контейнера формирует общий заголовок, впишем там слово Кандидаты.

Теперь обратите внимание на свойство `Items`. Это сложное свойство, содержит текст всех радиокнопок (набор строк). Щелкните по кнопке "..." правее свойства:

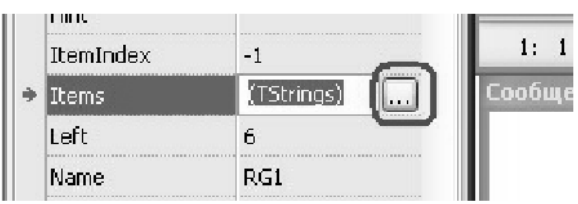


Рис. 7.5. Кнопка Диалога ввода строк

Откроется простой текстовый редактор, где вы можете указать текст ваших радиокнопок. Текст каждой радиокнопки должен располагаться на отдельной строке. Как и в прошлом случае, введите такие строки:

Путин
Медведев
Жириновский
Зюганов
Прохоров

Обратите внимание: если вы будете увеличивать высоту контейнера, расстояние между полученными радиокнопками будет увеличиваться. И наоборот. Подберите наиболее оптимальное расстояние между ними.

Правее этого контейнера установите контейнер для флажков `TCheckGroup`. Его свойство `Name` переименуйте в `ChG1`, а в свойстве `Caption` напишите Города. Этот контейнер также имеет сложное свойство `Items`, которое содержит текст всех флажков. Щелкните по кнопке "..." правее этого свойства, и в Диалоге ввода строк запишите такие строки:

Москва
Санкт-Петербург
Воронеж
Саратов
Уфа

В нашем проекте это будут проголосовавшие города. Ниже контейнеров установите кнопку, в свойстве `Caption` которой напишите Результат. Окончательный вид окна у вас должен быть примерно таким:

Рис. 7.6. Окно программы election

Теперь, прежде чем мы приступим к программированию события нажатия на кнопку, разберем еще кое-что. Контейнер флажков `TCheckGroup` имеет свойство `Checked`, которое будет доступно при работе программы. Это свойство логического типа, оно указывает, включен ли флажок. А чтобы разобраться, какой именно флажок включен, после свойства, в квадратных скобках, указывают индекс флажка. Индексы начинаются с нуля, поэтому, чтобы узнать, включен ли первый флажок, нужно выполнить код

```
if ChG1.Checked[0] then …
```

Не забывайте - индексы флажков всегда начинаются с нуля!

Контейнер радиокнопок `TRadioGroup` имеет свойство `ItemIndex`. Это свойство указывает, какую именно радиокнопку пользователь выбрал, индексация у них также начинается с нуля. По умолчанию, свойство равно -1. Это означает, что никакая радиокнопка не выбрана. Таким образом, чтобы выполнить действие для первой радиокнопки, мы должны проверить, выбрана ли она:

```
if RG1.ItemIndex = 0 then …
```

Чтобы повторить предыдущий пример, укажем 0 в свойстве `ItemIndex` (здесь индексы тоже начинаются с нуля) - радиокнопка Путин окажется выбранной по умолчанию. Не забывайте после ввода в свойства каких-то значений нажимать <Enter>, чтобы подтвердить изменения.

Теперь, когда мы знаем, как определить, какие флажки включены и какая радиокнопка выбрана, займемся программированием события нажатия на кнопку.

```
procedure TfMain.Button1Click(Sender: TObject);
var
  s: String; //для формирования строки с проголосовавшими городами
begin
  //сначала укажем победителя:
  if RG1.ItemIndex = 0 then
    ShowMessage('Вы выбрали Путина')
  else if RG1.ItemIndex = 1 then
    ShowMessage('Вы выбрали Медведева')
  else if RG1.ItemIndex = 2 then
    ShowMessage('Вы выбрали Жириновского')
  else if RG1.ItemIndex = 3 then
    ShowMessage('Вы выбрали Зюганова')
  else
    ShowMessage('Вы выбрали Прохорова');

  //теперь нужно что-то присвоить переменной s, чтобы инициализиро
  //ее. присвоим просто пустую строку:
  s:= '';
  //теперь будем собирать строку s, в зависимости от включенных флажк
  //после каждой строки будем вставлять знак перехода на новую строку
  if ChG1.Checked[0] then s:= s + 'Москва проголосовала' + #13;
  if ChG1.Checked[1] then s:= s + 'Санкт-Петербург проголосовал' + #13;
  if ChG1.Checked[2] then s:= s + 'Воронеж проголосовал' + #13;
  if ChG1.Checked[3] then s:= s + 'Саратов проголосовал' + #13;
  if ChG1.Checked[4] then s:= s + 'Уфа проголосовала' + #13;

  //Если хоть один флажок был включен, то строка s не пуста.
  //покажем ее в этом случае:
```

```
  if s <> '' then ShowMessage(s);
end;
```

Комментарии исчерпывающие, каких то затруднений с кодом у вас возникнуть не должно. Единственное, замечу, что если не проинициализировать вначале переменную `s`, то есть, не присвоить ей какое-то значение, хоть и пустую строку, в окне сообщений при компиляции может выйти напоминание:

```
Warning: Local variable "s" does not seem to be initialized
```

Дело в том, что дальше мы пытаемся использовать значение строки `s`:

```
… s:= s + 'Москва проголосовала' + #13;
```

А компилятор не знает, имеется ли там что-то, или просто мусор? Ошибки не будет, но все равно, переменные желательно вначале инициализировать: строковым присваивать пустую строку, числовым - значение 0 (обнулять). Логические по умолчанию сами принимают значение `False`.

Сохраните проект и запустите его на выполнение.

Числа

В лекции подробно рассмотрена работа с числами - целыми и вещественными, знаковыми и беззнаковыми. Дан материал по различным операциям с числами, их преобразовании в другие типы данных, вывод на экран в нужном формате. Материал закрепляется практикой - созданием программы для определения Индекса Массы Тела.

Цель лекции

Получение знаний по работе с числами всех типов, преобразование чисел в другие типы данных, закрепление материала на практике.

Целые числа

В Lazarus (а точнее, в Free Pascal), как и в любом другом языке программирования, числа играют довольно важную роль. Трудно представить себе программу, в которой не использовались бы числа. Даже когда вы просто установите какой-то компонент на форму, автоматически начинают действовать множество настроек. `Left`, `Top`, `Height`, `Width` - все эти свойства есть в любом визуальном компоненте, и они содержат числа. Числа бывают целые и вещественные, знаковые и беззнаковые.

В этом разделе поговорим о целых числах, как знаковых, так и беззнаковых. Что такое целое число? Это число без запятой, то есть, без десятичной части. Знаковым называют число со знаком: -1, например. Беззнаковое число - это число от нуля и больше.

В программировании базовым целым числом является `integer`, который мы уже не раз использовали. Но вы, вероятно, догадались, что это не единственный возможный целый тип? Есть разные типы целых чисел, они могут быть со знаком и без него, имеют разный диапазон возможных значений и, соответственно, занимают разный размер оперативной памяти. Разберем эти типы:

Таблица 8.1. Целые числа

Тип	Диапазон	Размер в байтах
`Byte`	0…255	1
`ShortInt`	-128…127	1
`Word`	0…65 535	2
`Smallint`	-32 768…32 767	2
`LongWord`	0…4 294 967 295	4
`Cardinal`	0…4 294 967 295	4
`LongInt`	-2 147 483 648…2 147 483 647	4
`Integer`	-2 147 483 648…2 147 483 647	4
`Int64`	$-2^{63} \ldots 2^{63}$	8

Обратите внимание, здесь диапазон и размер `Integer` совпадает с `LongInt`. Вообще-то, это зависит от режима компилятора FPC. Проект можно скомпилировать в разных режимах, с поддержкой Delphi, например, или TP (Turbo Pascal). По умолчанию, выставлен режим Object Pascal, это можно проверить, выполнив в среде Lazarus команду меню Проект -> Параметры проекта, затем в разделе Параметры компилятора выбрать Обработка. В верхней части там указан Режим синтаксиса, по умолчанию это Object Pascal, но при необходимости его можно и поменять.

Так вот, если там выставлен режим Object Pascal или Delphi, тогда `Integer` имеет размер 32 бита, или 4 байта. Если же выставлен старый режим Turbo Pascal или Free Pascal, то `Integer` будет иметь размер в 16 бит или 2 байта, и будет соответствовать типу `Smallint`.

Зачем нужно такое разнообразие целых типов? В былые времена оперативная память была довольно маленькой. Если сейчас у меня на ПК установлена оперативная память 4 Гб, то когда-то давно я был вынужден обходиться компьютером с процессором 486 и оперативкой аж в 16 мегабайт, и можете поверить, это был далеко не самый худший компьютер! В те времена программисты сражались за каждый байт памяти, переписывая и минимизируя код, выбирая самые маленькие из возможных типы данных. Это называлось оптимизацией кода. Допустим, вам нужно выполнить какой-то цикл 10 раз. Для подсчета шагов цикла вам придется создать переменную целого типа. Но зачем

использовать переменную `Integer` в 4 байта, когда вполне можно обойтись однобайтовым `Byte`? Сейчас конечно, это не играет такой большой роли, как прежде, но всё равно, оптимизация кода - это признак хорошего программиста, это хороший тон в программировании. Так что старайтесь не тратить понапрасну лишнюю память.

Рекомендации тут следующие: если вы знаете, что число будет без знака, то и выбирайте беззнаковые типы. Если вы точно знаете, что максимальное число в переменной будет маленьким, выбирайте типы поменьше. Если вам неизвестно, какого размера число попадет в переменную, то выбирайте `Integer` - это универсальный тип, годный для большинства случаев. Ну а если вы уверены, что число будет очень большим, то используйте 4-х или даже 8-ми байтовые типы.

Вещественные числа

Вещественными называются числа с дробной частью, причем, если дробная часть равна нулю, её все равно нужно указать. Например:

$$3.5$$
$$-10.427$$
$$8.0$$

Такие числа еще называют числами с плавающей точкой, поскольку количество цифр после точки может быть различным. Записываются вещественные числа по определенным правилам. Если в математике мы дробную часть отделяем запятой, в Lazarus для этого используют точку. При указании очень большого числа можно выбрать сокращенную форму. Если в математике для этого число умножают на десятичную степень, например,

$$0,25 * 10^5$$

то в Lazarus вместо 10 указывают букву E (от англ. exponent - показатель степени):

$$0.25E5$$

Степени могут быть и отрицательными:

$0.7E-23$

Вещественных типов тоже много. В характеристике вещественных чисел роль играет не только размер, занимаемый в памяти, но и количество значащих цифр:

Таблица 8.2. Вещественные числа

Тип	Диапазон	Количество значащих цифр	Размер в байтах
`Single`	1.5E-45…3.4E38	7-8	4
`Real`	5.0E-324…1.7E308	15-16	8
`Double`	5.0E-324…1.7E308	15-16	8
`Comp`	-2E64+1…2E63-1	19-20	8
`Currency`	-922 337 203 685 477.5808 … 922 337 203 685 477.5807	19-20	8
`Extended`	1.9E-4932…1.1E4932	19-20	10

Как видите, вещественные числа куда больше целых, процессорного времени на обработку таких чисел тратится тоже больше. Поэтому вещественные числа имеет смысл применять только по необходимости, когда целыми числами явно не обойтись. Не слушайте тех, кто предлагает на все случаи жизни использовать тип `Real` - и для целых, и для вещественных чисел.

Рекомендации тут такие же, как и для целых чисел - выбирайте типы по необходимости. Особо выделю тип `Currency` - его создали специально для финансовых расчетов, поэтому для всякого рода бухгалтерских расчетов лучше выбирать именно этот тип, как наиболее точный. Но чаще всего обходятся типом `Real` (или `Double`).

Операции над целыми и вещественными числами

Целые числа можно складывать (+), отнимать (-) и умножать (*) друг на друга. С делением дело обстоит сложней. Допустим, нам нужно 10 разделить на 3. Получится 3,33333…, а это уже не целое число. Поэтому для целых чисел в Паскале предусмотрено деление нацело. Операция `div` обеспечивает деление нацело, и возвращает целую часть,

отбрасывая дробную. Например, 10 разделить на 8 будет равно 1,25. Если применить целочисленное деление, то `10 div 8 = 1`. Чтобы узнать остаток от такого деления, применяют операцию `mod`. `10 mod 8 = 2`.

Арифметика над вещественными числами еще проще, здесь применяют следующие стандартные операции: + (сложение), - (вычитание), * (умножение), / (деление).

Кроме того, как целые, так и вещественные числа можно сравнивать между собой, используя для этого логические операторы: = (равно), <> (не равно), > (больше), < (меньше), >= (больше или равно), <= (меньше или равно).

Очень часто приходится использовать большие и сложные выражения, где вместе с арифметическими используются и логические операторы. Здесь главное - не забывать о приоритетах. Возьмем выражение

$$r := 3 + 4 * 2;$$

Что попадет в переменную `r`? Если вы ответили 11, то вы правы. Чтобы сначала выполнить сложение, его нужно поместить в скобки, которые имеют высший приоритет:

$$r := (3 + 4) * 2;$$

В этом случае, в переменную `r` попадет число 14.

Таблица 8.3. Приоритеты

Порядок	Операции
1	( ) - То, что в круглых скобках, вычисляется в первую очередь
2	`NOT`
3	`*, /, DIV, MOD, AND`
4	`+, -, OR, XOR`
5	`=, <>, >, <, >=, <=`

Примечание: Если в выражении встречаются операции одинакового приоритета, они выполняются слева - направо.

Примечание: Ни целое, ни вещественное число НЕЛЬЗЯ ДЕЛИТЬ НА НОЛЬ! Если такое произойдет, возникнет Исключительная ситуация, и будет выведена ошибка. Программисты обычно всегда проверяют, что и на что пользователь делит. Если он пытается делить на ноль, выводится понятное сообщение об ошибке, а само деление не производится. Это называется "защита от дурака".

Преобразования типов

Нередко возникают ситуации, когда нужно преобразовать один тип данных в другой. Например, целое число в вещественное или в строку, и обратно.

Целое число можно сразу же присвоить вещественному - преобразование произойдет автоматически. Например,

```
var
  r: real;
begin
  r:= 3;  //результат: 3.0
  …
```

Если есть желание, вы можете поступить так же, как в прошлых лекциях: создать новый проект с одной кнопкой, сгенерировать для кнопки событие нажатия, и все примеры пробовать в этом событии. Раздел переменных и начало события `begin` указаны, не забудьте в конце события оставить `end`;

Вещественное число присвоить целой переменной нельзя - компилятор выдаст ошибку. В этом случае мы можем поступить двумя способами: либо просто отбросить дробную часть, либо округлить вещественное число до ближайшего целого. Математическая функция `Trunc()` отсекает дробную часть, и возвращает целое число:

```
var
  i: integer;
begin
  i:= Trunc(3.65); //результат = 3
```

```
  …
```

Математическая функция `Round()` округляет вещественное число до ближайшего целого:

```
var
  i: integer;
begin
  i:= Round(3.65); //результат = 4
  …
```

Если целое число требуется преобразовать в строку, используют функцию `IntToStr()`, нам уже приходилось это делать. В скобках указывают целое число любого целочисленного типа, а функция возвращает это же число в виде строки. Пример:

```
var
  i: integer;
  s: string;
begin
  i:= 10;
  s:= IntToStr(i); //теперь в s строка '10'
  ShowMessage(s);  //содержимое s можно вывести на экран
  …
```

Для обратного преобразования используют функцию `StrToInt()`, программист должен следить, чтобы в эту функцию попадало действительно целое число в виде строки. Пример:

```
var
  i: integer;
  s: string;
begin
  s:= '35';  //загрузили число в виде строки
  i:= StrToInt(s);  //преобразовали строку '35' в число 35
  i:= i * 2;  //теперь здесь 70
  s:= IntToStr(i);  //получили строку '70'
```

```
  ShowMessage(s);  //содержимое s выводим на экран
  …
```

Если строку требуется преобразовать в вещественное число, используют функцию `StrToFloat()`:

```
var
  r: real;
  s: string;
begin
  s:= '3,14';
  r:= StrToFloat(s);
```

Обратите внимание - здесь в строковую переменную мы внесли вещественное число в виде строки. При этом десятичную часть мы разделили не точкой, а запятой: "3,14". Так положено вносить числа в операционной системе, если установлена русская версия Windows. В английской версии разделителем является точка. Данная настройка зависит от глобальной переменной `DecimalSeparator`. Переменную объявлять не нужно - она уже объявлена, и содержит символ системного разделителя дробной части.

Для преобразования вещественного числа в строку можно использовать две функции. Проще всего воспользоваться функцией `FloatToStr()`:

```
var
  r: real;
  s: string;
begin
  r:= 3.14;  //здесь мы вводим число, а не строку, поэтому разделитель - т
  s:= FloatToStr(r); //теперь в s строка '3,14'
  ShowMessage(s);
```

Если мы попытаемся ввести в вещественную переменную `r` число, используя в качестве разделителя запятую, то произойдет ошибка. Числа нужно разделять точкой, а числа в виде строки - запятой (в русской Windows).

Но лучше всего для преобразования числа в строку использовать функцию `FormatFloat()`. Эта функция не только преобразует вещественное число в строку, но и выводит его в нужном формате. Желаемый формат задается маской - строкой со специальными символами. Синтаксис функции:

```
FormatFloat(Формат, Значение);
```

Формат - это строка с маской, Значение - исходное вещественное число. Функция возвращает в качестве результата указанное вещественное число в виде строки, отформатированной согласно маске.

Таблица 8.4. Символы формата

Символ	Описание
0	В данную позицию записывается цифра. Если в исходном числе в данной позиции была цифра - записывается она. Иначе будет записан '0'.
#	В данную позицию записывается цифра. Если в исходном числе в данной позиции была цифра - записывается она. Иначе в данную позицию ничего не записывается.
.	Данный символ, встречающийся в маске первый раз, определяет расположение десятичного разделителя. Любой последующий символ '.' игнорируется. Символ, который будет использован в качестве десятичного разделителя в результирующей строке, хранится в глобальной переменной `DecimalSeparator`.
,	Если маска содержит данный символ, то в отформатированной строке, между каждой группой из трех цифр слева от десятичной точки, будут вставлены разделители тысяч. Положение и количество символов ',' в маске на результат не влияет. Символ, который будет использован в качестве разделителя тысяч, определяется глобальной переменной `ThousandSeparator`.
E+	Экспоненциальный формат. Если любая из строк: 'E+', 'E-', 'e+', 'e-', содержится в строке формата, то результирующее значение будет представлено в экспоненциальном виде. За одной из

	этих строк могут следовать символы '0', определяющие минимальное число цифр в экспоненте.
;	Отделяет в маске варианты форматирования для положительного, отрицательного и нулевого значения.

Примечание: Если в маске после запятой должны выходить две цифры, а в вещественном числе их больше, дробная часть будет округлена.

Пример:

```
var
  r: real;
  s: string;
begin
  r:= 123.789;
  s:= FormatFloat('#.##', r);
  ShowMessage(s);  //результат - '123,79' - дробная часть округлена
  r:= 789.123;
  s:= FormatFloat('#.##', r);
  ShowMessage(s);  //результат - '789,12'
```

Для денежных расчетов можно применить маску `'#.00'` - это гарантирует две цифры после запятой. Если цифр нет, будут выведены нули, например

```
123,00
```

Практика

Теперь, когда вы знаете про цифры почти всё, самое время закрепить материал на практике. Мы напишем программу, которая будет определять нормальный ли у нас вес, избыточный, или наоборот, недостаточный.

Общее признание получил так называемый индекс массы тела (ИМТ). Его расчет таков: разделите свой вес в килограммах на рост в метрах в квадрате. Пример: `ИМТ = 68 кг / (1,72 м * 1,72 м) = 23.`

Эта формула хороша тем, что работает и для "малышей", и для "гулливеров", и для женщин, и для мужчин. Нормой считается ИМТ от 19 до 25. ИМТ меньше 19 - дефицит веса, 25-30 - избыточный вес, 30-40 - ожирение, более 40 - сильное ожирение.

Загрузите Lazarus с новым проектом. Если он у вас уже загружен - закройте проект и создайте новый. Сохраните проект в папку 08-01 под именем IMT, модуль главной формы назовите Main, а свойство `Name` формы переименуйте в `fMain`. Сделаем предварительные настройки формы. В свойстве `Caption` напишите:

ИМТ - Индекс массы тела

В свойстве `BorderStyle` установите `bsDialog`, а в `Position` - `poDesktopCenter`.

От пользователя нам нужно получить вес и рост, причем рост должен быть вещественным числом, а вес - целым. Например, вес 68, рост 1,72. Чтобы гарантировать правильность ввода, используем компонент `TMaskEdit` с вкладки Additional Палитры компонентов. Нам понадобятся два таких компонента, две метки `TLabel` и одна кнопка для вывода результатов. Оформите форму похожим образом:

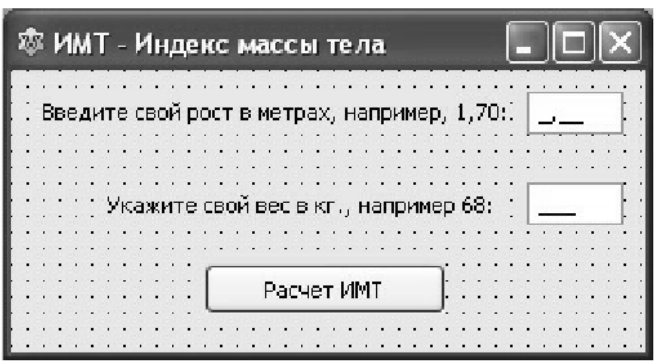


Рис. 8.1. Внешний вид программы IMT

Измените свойство `Name` верхнего `TMaskEdit` на `ME1`, нижнего - на `ME2`. Имена остальных компонентов можно оставить по умолчанию. Нам еще понадобится изменить свойства `EditMask` обоих

компонентов. Для ME1 установите EditMask:

```
0,00;1;_
```

Это нам гарантирует правильный ввод пользователем своего роста. Однако не забывайте, что на самом деле, пользователь будет вводить не число, а строку! К примеру, он введет "1,82" - это не вещественное число, как можно было бы подумать, а строка из цифровых символов. И нам ещё придётся преобразовать её в настоящее число. Для ME2 маска будет

```
###;0;_
```

Учитывая, что вес может быть и трехзначным целым числом.

Теперь нам осталось только сделать расчет. Правда, пока мы не будем делать проверку - ввел ли вообще пользователь в ME1 и ME2 что-нибудь? Оставим это на совести пользователя. О том, как делать "защиту от дураков", мы ещё поговорим в одной из следующих лекций.

Сгенерируйте событие нажатия на кнопку, и оформите его следующим образом:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  s: string; //для формирования отчета
  rost: real; //для получения роста
  ves: Byte; //для получения веса
  imt: real; //для расчета ИМТ
begin
  //сначала преобразуем рост из строки в вещественное число:
  rost:= StrToFloat(ME1.Text);
  //теперь вес:
  ves:= StrToInt(ME2.Text);
  //теперь рассчитываем ИМТ:
  imt:= ves / (rost * rost);
  //в зависимости от результата формируем строку отчета:
  s:= 'Ваш ИМТ = ' + FormatFloat('#.##', imt) + #13;
```

```
  if imt < 19 then s:= s + 'У вас дефицит веса!'
  else if (imt >= 19) and (imt <= 25) then s:= s + 'У вас нормальный вес!'
  else if (imt > 25) and (imt <= 30) then s:= s + 'У вас избыточный вес!'
  else if (imt > 30) and (imt <= 40) then s:= s + 'У вас ожирение!'
  else if (imt > 40) then s:= s + 'Кошмар! У вас сильное ожирение!'
  else s:= 'Что-то пошло не так, результат не удалось рассчитать';
  //выводим результат на экран:
  ShowMessage(s);
end;
```

Программа получилась небольшая, но довольно полезная. Комментарии тут достаточно подробные, чтобы вы поняли, что тут к чему. Если вы планируете передавать программу друзьям, не забудьте отключить отладочную информацию для уменьшения размера файла программы IMT.exe (см. Лекцию №2).

Подпрограммы

В данной лекции рассматривается работа с подпрограммами - процедурами и функциями. Подробно рассматриваются аргументы, передаваемые в подпрограммы, параметры по значению, параметры по ссылке. Рассматриваются различные способы реализации подпрограмм, а также такое важное понятие, как область видимости переменных и других объектов.

Цель лекции

Освоение работы с подпрограммами, с параметрами по ссылке, параметрами по значению, с досрочным выходом из программ и подпрограмм, с областью видимости переменных.

Подпрограммы

Вначале языки программирования были проще, они выполнялись строго сверху-вниз, один оператор за другим. Такие языки еще называли линейными. Типичный пример линейных языков - Бейсик. Единственную возможность организовать хоть какую-то логику в таких языках предоставлял оператор безусловного перехода `GOTO`, который в зависимости от условия, "перепрыгивал" на заранее расставленные метки. В современных языках программирования `GOTO` тоже остался, наверное, для любителей антиквариата. Но его применение может привести к трудно обнаруживаемым логическим ошибкам времени выполнения (`run-time errors`). Использование `GOTO` в современном программировании считается дурным тоном. Мы не будем изучать эту возможность, поскольку для организации логики есть куда более "продвинутые" средства! Одним из таких средств являются подпрограммы.

Подпрограмма - это часть кода, которую можно вызвать из любого места программы неопределенное количество раз.

Другими словами, подпрограммы подобны строительным кирпичикам, из которых, в конце концов, получается здание - программа. Без подпрограмм можно обойтись, если вы пишете небольшую учебную

программу на пару десятков строк кода. А если это серьезное приложение, с парой сотен модулей, в каждом из которых могут быть тысячи строк кода? Как такую программу написать, не разбивая задачу на отдельные части? Подпрограммы помогают улучшить код, структурировать его. Поэтому языки высокого уровня, которые позволяют использовать подпрограммы, называют ещё процедурно-ориентированными языками. И наш компилятор FPC тоже относится к таким языкам.

Подпрограммы бывают двух типов: процедуры и функции - первые просто выполняют свою работу, вторые еще и возвращают результат этой работы.

Процедуры

На самом деле, мы уже неоднократно использовали процедуры. Например, когда генерировали событие нажатия на кнопку. Это событие - процедура. Процедура начинается с ключевого слова `procedure` и имеет следующий синтаксис:

```
procedure <имя процедуры>(<список параметров>);
const
  <объявление констант>;
type
  <объявление новых типов>;
var
  <объявление переменных>;
<описание вложенных процедур и функций>;
begin
 <тело процедуры>;
end;
```

Большая часть указанного синтаксиса является необязательной - процедура может не иметь параметров, констант, пользовательских типов данных, переменных и т.п. - она может быть очень простой, например:

```
procedure ErrorMessage;
```

```
begin
  ShowMessage('Ошибка!' +#13 + 'На ноль делить нельзя!');
end;
```

Такую процедуру можно вызвать из любого места программы, но процедура обязательно должна быть описана выше - ведь иначе компилятор не будет знать о ней. Есть еще возможность предварительно объявить процедуру, но об этом чуть позже. Итак, если эта процедура описана выше, то мы можем вызвать её, просто указав её имя:

```
ErrorMessage;
```

Компилятор перейдет к процедуре и выполнит её код (в данном случае - выведет сообщение об ошибке). После этого компилятор вернется назад, и выполнит следующий за вызовом процедуры оператор.

Параметры

Параметры подпрограмм - достаточно важная тема, поговорим об этом подробней. В процедуру (или в функцию) можно передать какие то исходные данные, чтобы процедура их обработала. Такие данные называются параметрами, или формальными параметрами. Пример - пользователь ввел какое-то число (а на самом деле, строку из цифровых символов), нам нужно удвоить его, а результат сообщить пользователю. Описать подобную процедуру можно следующим образом:

```
procedure Udvoenie(st: string);
var
  r: real;
begin
  //полученную строку преобразуем в число:
  r:= StrToFloat(st);
  //теперь удвоим его:
  r:= r * 2;
  //теперь выведем результат в сообщении:
  ShowMessage(FloatToStr(r));
end;
```

Этот пример уже сложнее, правда? На самом деле, всё просто. Давайте разберем, что тут к чему. Итак, строка объявления процедуры:

```
procedure Udvoenie(st: string);
```

объявляет процедуру `Udvoenie` с параметром строкового типа `st`. Это означает, что теперь мы можем вызвать процедуру, передав ей в качестве параметра какую-то строку. Параметр `st` условно можно считать внутренней переменной процедуры, в которую компилятор скопирует передаваемую процедуре строку. Этот способ передачи данных в подпрограмму называется параметром по значению. Допустим, в дальнейшем мы вызвали процедуру таким образом:

```
Udvoenie('123.4');
```

Компилятор сделает вызов процедуры, передав в параметр `st` указанное значение `'123.4'`. Или же мы можем вызвать процедуру иначе, передав в неё значение, которое хранится в какой то другой строковой переменной:

```
myst:= '123.4';
Udvoenie(myst);
```

Результат будет таким же. Тут важно помнить, что тип передаваемого значения обязательно должен совпадать с типом параметра. Если параметр у нас `string`, то и передавать ему нужно значение типа `string`. Компилятор копирует это значение в параметр. Другими словами, если внутри процедуры мы изменим значение параметра `st`, это никак не отразится на переменной myst, поскольку мы изменим копию данных, а не сами данные.

Пойдем дальше. А дальше мы объявляем вещественную переменную `r`:

```
var
  r: real;
```

Здесь она необходима, ведь нам нужно умножить значение параметра на два, поэтому мы вынуждены будем преобразовать строковое представление числа в настоящее число - ведь строку на два не умножишь! Результат поместим в `r`:

```
begin
  //полученную строку преобразуем в число:
  r:= StrToFloat(st);
```

Служебным словом `begin` мы начинаем тело процедуры. Стандартной функцией `StrToFloat(st)` мы преобразуем строковое значение параметра `st` в число, и присвоим это число переменной `r`. Далее всё просто:

```
  //теперь удвоим его:
  r:= r * 2;
  //теперь выведем результат в сообщении:
  ShowMessage(FloatToStr(r));
end;
```

Мы удваиваем значение `r`, результат этого помещаем снова в `r`, затем стандартной функцией `FloatToStr(r)` преобразуем полученное число в строку, и выводим эту строку в сообщении `ShowMessage()`. Вот, собственно, и всё.

Теперь мы сможем вызывать эту процедуру, когда необходимо, и передавать ей различные числа в виде строки. А уж функция сама позаботится обо всех необходимых преобразованиях, об удвоении числа и выводе результатов на экран.

Кстати, сами данные, которые мы передаём в подпрограмму, называются аргументами или фактическими параметрами. В примере вызова процедуры

```
myst:= '123.4';
Udvoenie(myst);
```

переменная `myst` - аргумент.

В качестве параметров в процедуре можно использовать не одну, а множество переменных. Если они имеют одинаковый тип, то их имена разделяют запятыми, а тип указывается в конце сразу для всех параметров. Например:

```
procedure MyStrings(st1, st2, st3: string);
```

Если параметры имеют разные типы, их разделяют точкой с запятой:

```
procedure MyProc1(st: string; r1:real);
procedure MyProc2(st1, st2, st3:string; r1:real);
```

Однако, разбавим теорию практикой, и поработаем с процедурами на реальном примере. Откройте Lazarus с новым проектом. Как всегда, назовем главную форму (свойство `Name`) `fMain`, сохраним проект в папку 09-01, при этом назовем проект, например, MyPodprog, а модулю дадим имя Main.

В свойстве `Caption` формы напишем

```
Примеры работы с подпрограммами
```

Наша задача: получить от пользователя вещественное число, удвоить его, и результат вывести на экран. Пользователь может ввести и целое число, но процедура обработает его как вещественное (помните о преобразовании типов в прошлой лекции?), например, если пользователь введет 3, то процедура получит 3.0. В результате вычисления получится 6.0, но `FloatToStr()` конечные нули не выводит, так что пользователь увидит на экране просто 6.

Ладно, сейчас нужно решить, как получить у пользователя число. Для этого используем компонент `TEdit`, который нам уже знаком по прошлым лекциям. Для начала установим метку `TLabel` с поясняющим

текстом

Введите любое число:

а рядом установим `TEdit`. Имена у `TLabel` и `TEdit` оставим по умолчанию, `TEdit` будет называться `Edit1`. Не забудьте очистить у него свойство `Text`.

Ниже установите простую кнопку TButton, в `Caption` которой напишите текст:

Пример удвоения №1

Разумеется, будут и другие примеры. Подровняйте компоненты, при необходимости измените их размеры. Наша форма должна выглядеть примерно так:

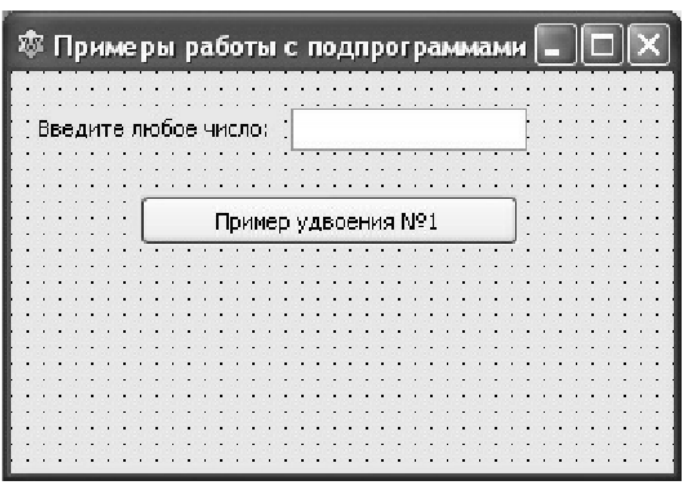


Рис. 9.1. Окно программы MyPodprog

Пока что мы будем вынуждены доверять пользователю, что он введет в поле Edit1 число, и ничего более. Но на прошлой лекции вам было обещано показать реализацию "защиты от дураков", так что чуть позже мы и это сделаем.

Сгенерируйте событие нажатия на кнопку, оно будет таким:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  Udvoenie(Edit1.Text);
end;
```

Теперь нам нужно создать процедуру `Udvoenie` выше нашего события, сразу после комментария

```
{ TfMain }
```

Текст процедуры приведен выше.

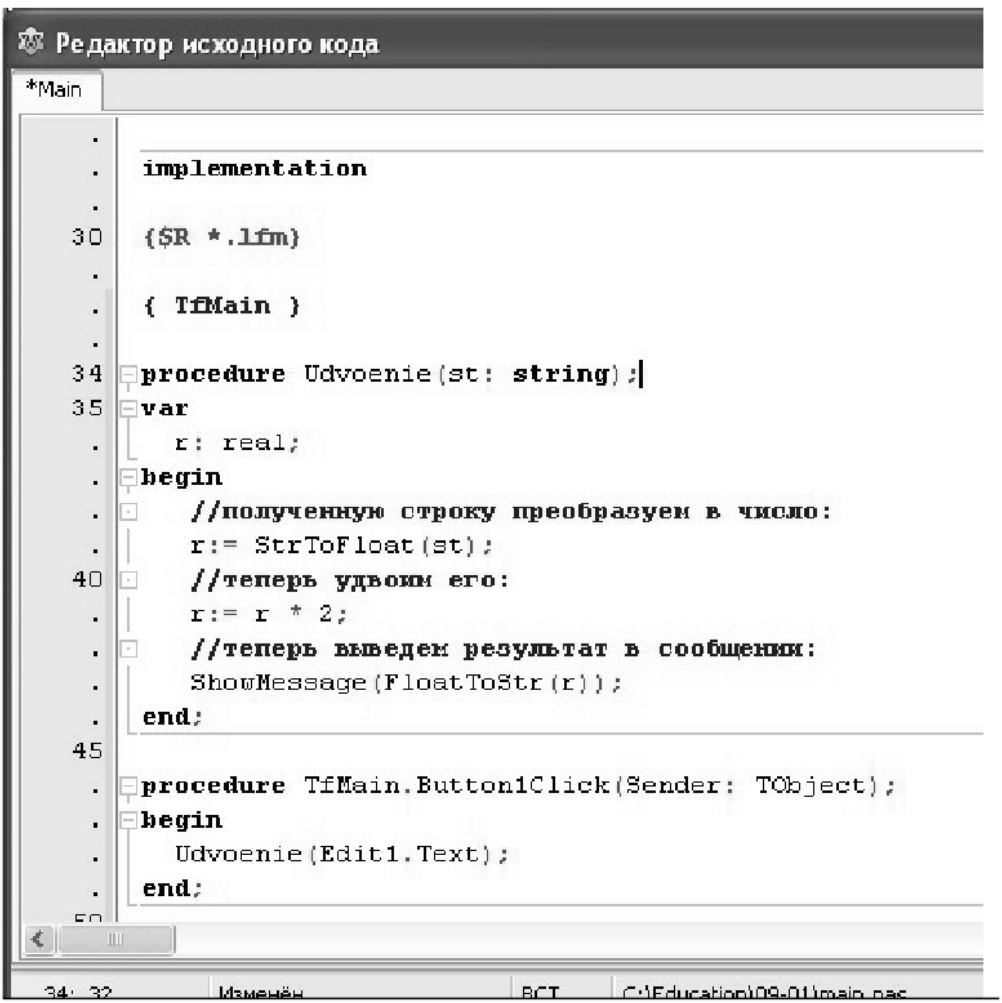


Рис. 9.2. Реализация подпрограммы Udvoenie

Обратите внимание, мы передаем в подпрограмму значение, которое ввел пользователь, и которое хранится в свойстве `Text` компонента `Edit1`:

```
Udvoenie(Edit1.Text);
```

Никаких дополнительных переменных в данном случае создавать не нужно. Сохраните проект и запустите его на выполнение. Попробуйте ввести целое число. Затем вещественное. Обратите внимание: если у вас установлена русская версия Windows, то в качестве разделителя вещественного числа нам нужно вводить запятую, а не точку! Помните про глобальную переменную `DecimalSeparator`?

Если же вы случайно или намеренно ввели точку, то выйдет сообщение об ошибке, подобное этому:

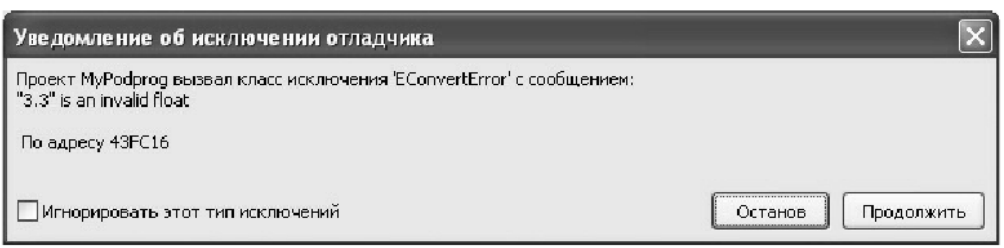


Рис. 9.3. Сообщение об ошибке

Ничего страшного, нажмите кнопку "Останов", затем выберите команду главного меню "Запуск -> Сбросить отладчик". Lazarus закроет зависший проект, и вы сможете запустить его снова. Похожая ошибка возникнет, если вы попытаетесь удвоить пустую строку. Если же вы ввели числа правильно, то программа отработает как нужно в независимости, целое это было число, или вещественное. Не закрывайте пока проект, он нам еще понадобится.

Функции

Функции являются такими же подпрограммами, как и процедуры, с

одной разницей: они возвращают значение указанного типа. Функция начинается с ключевого слова `function` и имеет следующий синтаксис:

```
function <имя функции>(<список параметров>): <тип возвращаемого зна
const
  <объявление констант>;
type
  <объявление новых типов>;
var
  <объявление переменных>;
<описание вложенных процедур и функций>;
begin
 <тело процедуры>;
end;
```

В теле функции должен быть оператор, который присваивает возвращаемое значение стандартной переменной `Result`, что и приводит к возвращению функцией значения. `Result` объявлять не нужно, он уже есть в каждой функции. Разумеется, в `Result` нужно присваивать значение только указанного в заголовке типа. Пример:

```
function MyFunc(i: integer): integer;
begin
  Result:= i * 2;
end;
```

Здесь функция `MyFunc` принимает в качестве параметра какое-то целое число, удваивает его и присваивает результат переменной `Result`. Это приводит к тому, что функция возвращает этот результат. Теперь мы можем удвоить целое значение, вызвав эту функцию, например:

```
myperem:= 5;
myperem:= MyFunc(myperem);
```

Что будет в результате в переменной `myperem`? Если вы ответили 10,

то вы правы. Здесь мы в первом шаге присваиваем переменной целого типа `myperem` значение 5. Во втором шаге мы вызываем функцию `MyFunc`, передавая ей в качестве параметра значение `myperem`. В третьем шаге мы присваиваем полученный от функции результат удвоения снова в переменную `myperem`.

Кстати, вместо системной переменной `Result` можно использовать имя функции:

```
function MyFunc(i: integer): integer;
begin
  MyFunc:= i * 2;
end;
```

Данный пример даст точно такой же результат. Какой способ использовать - дело выбора. Лично я предпочитаю использовать `Result`, это выглядит как-то более стандартно. В любом случае, вы должны знать про оба способа.

В отличие от многих других языков, в Lazarus переменной `Result` (или имени функции) значение можно присваивать неоднократно, если этого требует логика подпрограммы. Например, функция получает два вещественных числа. Первое нужно разделить на второе, и результат вернуть. Но на ноль делить нельзя, поэтому во избежание зависания программы, если второе число - ноль, то и вернуть нужно ноль. Такую функцию можно реализовать следующим образом:

```
function Delenie(r1, r2: real): real;
begin
  if r2 = 0 then Result:= 0
  else Result:= r1 / r2;
end;
```

Хорошо, опробуем функции на практике. Ниже Button1 добавьте Button2 с текстом в `Caption`:

Пример удвоения №2

Сгенерируйте для второй кнопки событие `OnClick`, чуть выше этого события опишите функцию удвоения `FuncUdvoenie`. В результате у вас получится следующее:

```
function FuncUdvoenie(st: string): string;
var
  r: real;
begin
   //полученную строку сначала преобразуем в число:
   r:= StrToFloat(st);
   //теперь удвоим его:
   r:= r * 2;
   //теперь вернем результат в виде строки:
   Result:= FloatToStr(r);
end;

procedure TfMain.Button2Click(Sender: TObject);
begin
  ShowMessage(FuncUdvoenie(Edit1.Text));
end;
```

Обратите внимание, здесь мы пошли немного другим путем. Мы создали функцию, которая лишь возвращает результирующее число в виде строки, а вывод этой строки на экран организовали при вызове функции, в строке

```
ShowMessage(FuncUdvoenie(Edit1.Text));
```

Тут в первом шаге мы вызываем функцию `FuncUdvoenie`, передавая ей в качестве параметра текст из `Edit1`. Во втором шаге отрабатывает функция - преобразует этот текст в число, удваивает его, снова преобразует в строку, и результат возвращает компилятору. А в третьем шаге с помощью функции `ShowMessage()` мы выводим этот результат на экран. Реализовано иначе, но работать будет точно также. Попробуйте. И пока не закрывайте проект.

Параметры по ссылке

Параметры по ссылке позволяют изменять сами аргументы, поскольку в процедуру или функцию передается не копия аргумента, а ссылка на сам аргумент. Чтобы в функцию или процедуру передать параметр по ссылке, нужно перед параметром указать ключевое слово `var`, например:

```
procedure MyProc(var myparam: integer);
```

Если вы передаете в подпрограмму параметры как по значению, так и по ссылке, то параметры по ссылке должны идти в описании последними. Вот пример объявления процедуры с множеством параметров:

```
procedure MyProc2(a,b: integer; c: real; var s1, s2: string);
```

Здесь мы объявили три параметра по значению: `a` и `b` - целые числа, `c` - вещественное; и два параметра по ссылке - `s1` и `s2`, оба строкового типа. Если внутри процедуры мы изменим все эти параметры, то исходные целые числа и вещественное число не изменятся, но изменения в обеих строках будут сохранены и после выхода из процедуры. Давайте попробуем поработать с параметрами по ссылке на практике. Сделайте в проекте третью кнопку, аналогично первым двум, и расположенную ниже. В свойстве `Caption` этой кнопки напишем:

```
Пример удвоения №3
```

Сгенерируйте событие `OnClick` для нее, и чуть выше опишите процедуру, которая будет принимать и изменять параметр по ссылке, следующим образом:

```
procedure UdvoeniePoSsilke(var r: real);
begin
  r:= r * 2;
end;

procedure TfMain.Button3Click(Sender: TObject);
var
```

```
  myReal: real;
begin
  myReal:= StrToFloat(Edit1.Text);
  UdvoeniePoSsilke(myReal);
  ShowMessage(FloatToStr(myReal));
end;
```

Этот пример отличается от предыдущих, но выполняет ту же работу, и с тем же результатом. В процедуре `UdvoeniePoSsilke` всего одна строка

```
  r:= r * 2;
```

Эта строка удваивает не просто параметр-копию, она удваивает сам аргумент! Поскольку перед параметром `r` указано ключевое слово `var`, то этот параметр - не копия аргумента, а ссылка на него. И удваивая `r`, мы удваиваем аргумент. Что и было продемонстрировано следующим событием нажатия на третью кнопку. Здесь, в первом шаге мы преобразуем строковое представление числа из `Edit1` в число, и результат присваиваем вещественной переменной `myReal`. Затем, во втором шаге, мы передаем эту переменную в процедуру `UdvoeniePoSsilke`, которая изменяет значение `myReal`. Ну и, наконец, в третьем шаге, мы выводим результат на экран, предварительно преобразовав его в строковое представление. Надеюсь, вы не запутались в параметрах и аргументах?

Описание подпрограмм с их предварительным объявлением

Применение процедур и функций вышеописанными способами имеет некоторые недостатки. Во-первых, мы вынуждены описывать процедуры и функции выше того места, где будем их применять. Это связано с тем, что компилятор не будет знать про эти подпрограммы, если описать их ниже, и не сможет их выполнить. Собственно, вы не сможете даже скомпилировать такой проект. Во-вторых, в таких подпрограммах мы не сможем обращаться к свойствам компонентов.

Если бы мы попытались из подпрограммы обратиться к той же `Edit1.Text`, то не смогли бы это сделать, хотя в событии нажатия на кнопку мы делаем это без труда. Дело в том, что события принадлежат к самой нашей форме, это видно по названию события:

```
procedure TfMain.Button3Click(Sender: TObject);
```

Видите, компилятор обращается вначале к форме, и только потом к самому событию? А вот в случае подпрограмм мы обращаемся напрямую к процедуре или функции. Но есть способ объявления подпрограммы, как части формы. Такой способ предоставляет нам следующие преимущества:

1. Предварительно объявленную подпрограмму можно описывать в любом месте программы, не обязательно выше места её применения.
2. В такой подпрограмме можно обращаться к свойствам и событиям компонентов.
3. Подпрограмму можно описать как приватную, или как публичную. Приватную можно использовать только в текущем модуле (pas-файле). Публичную можно использовать в любом другом модуле, к которому подключается текущий.

Давайте посмотрим, как это делается на практике. Добавьте четвертую кнопку ниже предыдущих трех, с надписью в `Caption`

Пример удвоения №4

Сгенерируйте для неё событие нажатия на кнопку, в котором просто укажем вызов процедуры `MyPrivat`:

```
procedure TfMain.Button4Click(Sender: TObject);
begin
  MyPrivat;
end;
```

Это ничего, что самой процедуры еще нет, сейчас мы её объявим, а затем создадим. Поднимите курсор на начало модуля. Вы увидите раздел `type`, в котором есть объявление класса `TfMain` - нашей формы. А там объявлены все компоненты и сгенерированные нами ранее события `OnClick`. Ниже располагаются подразделы `private` и `public`, где мы можем объявлять подпрограммы соответственно, приватные и публичные. Давайте объявим процедуру `MyPrivat` в подразделе `private`:

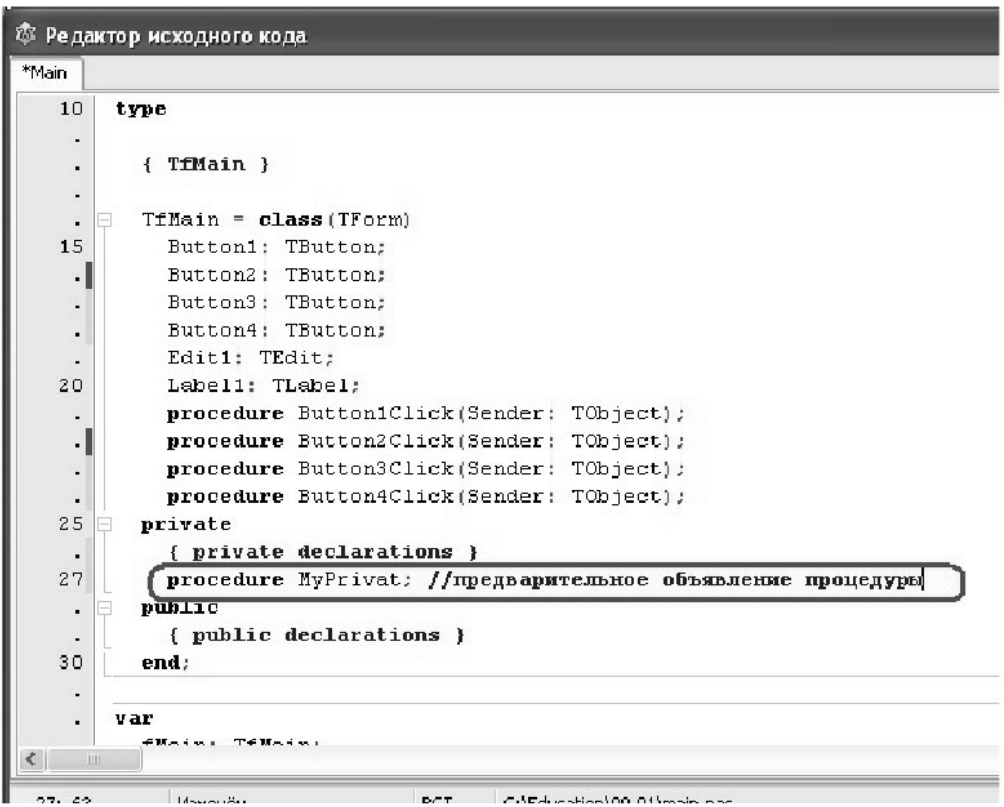


Рис. 9.4. Предварительное объявление процедуры в подразделе private

Теперь, не убирая курсора с этой строки, нажмите клавиши <Ctrl + Shift + C>. Это приведет к тому, что в самом низу модуля (но перед завершающим "`end.`") сгенерируется описание процедуры `MyPrivat`. На самом деле, она могла бы быть сгенерирована где-то и в другом месте, всё зависит от того, в каком порядке указаны объявления всех событий и других подпрограмм. Посмотрите на рис. 9.4 - там

`MyPrivat` объявлена последней, потому и сгенерировалось описание внизу модуля.

Никаких параметров нам в данном случае не нужно, мы будем обращаться к свойству `Text` компонента `Edit1` прямо из нашей подпрограммы. И обратите внимание, в описании перед именем процедуры указано имя формы. Отредактируйте подпрограмму следующим образом:

```
procedure TfMain.MyPrivat;
var
  r: real;
begin
  //преобразуем в число то, что ввел пользователь:
  r:= StrToFloat(Edit1.Text);
  //теперь удвоим его:
  r:= r * 2;
  //теперь выведем результат в сообщении:
  ShowMessage(FloatToStr(r));
end;
```

Комментарии достаточно подробны, чтобы вы смогли разобраться с кодом. Подобный способ применения подпрограмм с предварительным объявлением является наиболее удобным, советую использовать именно его. Подпрограммы без предварительного объявления стоит использовать только в простейших случаях, когда, к примеру, нужно рассчитать какие-то данные, ну например, преобразовать значения температуры из шкалы в Фаренгейтах в шкалу по Цельсию. Тогда можно описать функцию подобного преобразования выше, и передавать в неё различные величины. Но даже и такую подпрограмму можно предварительно объявить!

Область видимости переменных

До сих пор мы не рассматривали переменные в этом аспекте. Просто объявляли их в подпрограммах, и использовали только внутри их. Такие переменные называются локальными, поскольку имеют локальную область видимости. Поясню. Переменная, объявленная внутри

процедуры или функции, физически создается только тогда, когда компилятор обращается к данной подпрограмме. До этого переменной не существует. Только когда происходит вызов процедуры или функции, в оперативной памяти физически выделяется место для объявленных в подпрограмме переменных, констант и других объектов. Когда же подпрограмма завершает свою работу, то все эти переменные (константы и проч.) автоматически уничтожаются. К ним нельзя обратиться из других подпрограмм, их там просто не видно. Вот почему мы можем объявлять переменные с одинаковым именем в различных подпрограммах - это разные переменные, и они не мешают друг другу. Посмотрите на код нашего проекта - мы трижды объявляли переменную `r` вещественного типа, и каждый раз это была другая переменная.

Однако бывают моменты, когда требуется использовать глобальные переменные - переменные, которые видны по всему модулю, и в других модулях, если к ним подключается текущий. Такие переменные мы можем объявить либо в разделах `private` и `public`, до объявления подпрограмм, либо в разделе `interface`, после объявления переменной с именем формы, до ключевого слова `implementation`. Давайте объявим переменную `MyNum`:

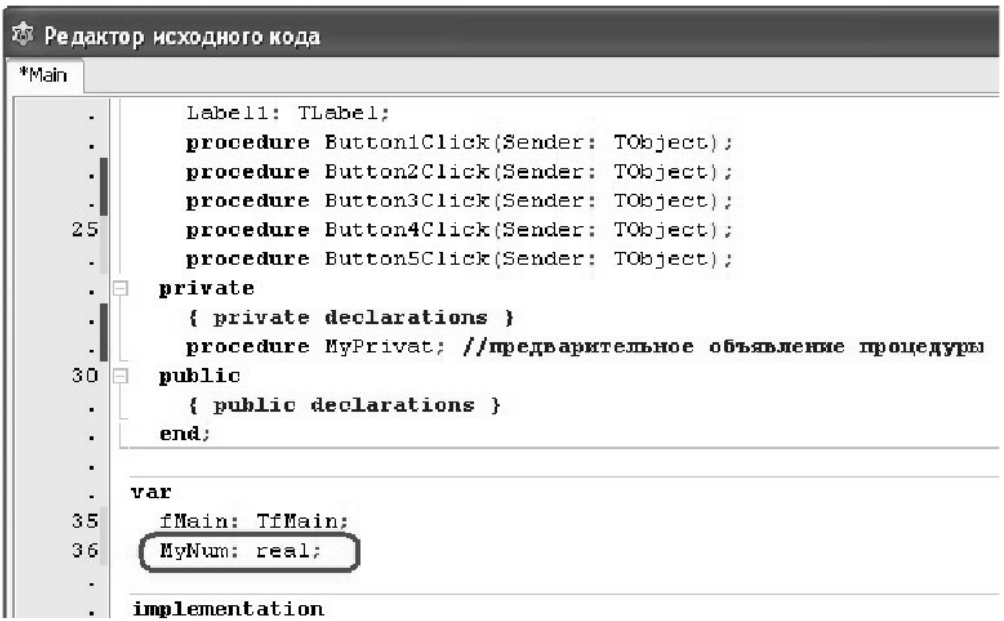


Рис. 9.5. Объявление глобальной переменной

Теперь добавим на форму пятую кнопку, текст в `Caption` будет соответствующий. Сгенерируйте для неё событие `OnClick`. Обратите внимание, оно создалось выше предыдущей процедуры `MyPrivat`, поскольку объявления различных событий располагаются выше разделов `private` и `public`. Опишем событие следующим образом:

```
procedure TfMain.Button5Click(Sender: TObject);
begin
  MyNum:= StrToFloat(Edit1.Text);
  //теперь удвоим его:
  MyDouble;
  //выводим результат на экран:
  ShowMessage(FloatToStr(MyNum));
end;
```

Обратите внимание: переменную `MyNum` мы в событии не объявляем - она уже объявлена глобально, и ей можно пользоваться в любом месте модуля. Процедуры `MyDouble` ещё нет, объявим её в разделе `private` ниже `MyPrivat`:

```
procedure MyDouble; //удвоение глобальной переменной
```

Далее сгенерируем описание этой процедуры (`<Ctrl + Shift + C>`). Само описание будет очень простым:

```
procedure TfMain.MyDouble;
begin
  //удвоим глобальную переменную:
  MyNum:= MyNum * 2;
end;
```

Как видите, здесь мы работаем с той же глобальной переменной, не объявляя её внутри процедуры, поскольку она уже объявлена.

Глобальные переменные очень удобны, однако используйте их только там, где это действительно необходимо. Ведь память для глобальной

переменной выделяется, когда вы загружаете программу, и не освобождается, пока вы программу не закроете.

Досрочный выход из подпрограмм и программы

Иногда бывает необходимо срочно завершить процедуру или функцию. К примеру, в зависимости от каких то условий, вам требуется либо продолжать обработку данных, либо закончить подпрограмму и вывести готовый результат. Для этого существует ключевое слово `Exit`. Запомните: если в подпрограмме встретилось `exit`, подпрограмма досрочно завершает свою работу и компилятор передает управление следующему за вызовом подпрограммы оператору.

Если же встретится ключевое слово `Halt`, то уже вся программа досрочно завершает свою работу. Она закрывается, память, занимаемая программой, освобождается, а управление передается операционной системе. Чаще всего эту команду применяют для аварийного завершения работы.

Сейчас мы не будем применять эти операторы на практике, поскольку лекция итак получилась очень большой. При желании, вы можете опробовать их самостоятельно, в каких то из уже сделанных подпрограмм, или в новых. В дальнейшем мы не раз будем обращаться к этим операторам, по крайней мере, к оператору `Exit`.

Циклы и переключатель case

В данной лекции мы изучим работу с циклами for, while, repeat, и переключателем case. Рассмотрим весь материал на небольших практических примерах, показывающих все особенности работы с циклами.

Цель лекции

Изучение циклов и оператора выбора case. Применение этих инструментов на практике в различных случаях.

Циклы

На этой лекции мы поговорим еще об одном способе организовать логику программы - о циклах. Довольно часто программисту приходится выполнять какой-то участок кода заданное количество раз, или до тех пор, пока не наступит какое-то условие. Такой повтор кода и называется циклом.

Цикл - последовательный повтор части кода от нуля раз до бесконечности.

Действительно, бывают условия, при которых цикл не выполняется ни разу. А бывает и так, что цикл начинает выполняться бесконечно, при этом программа "зависает" - не реагирует на клавиатуру и мышь, не дает продолжать с ней работу. Но это уже ошибка программиста. Циклы бывают двух видов - по счетчику, и по условию. Первые выполняются заданное количество раз, вторые - пока не наступит определенное логическое условие. Именно второй тип циклов начинающие программисты могут "зациклить", не предусмотрев гарантированного наступления условия их завершения. Но давайте-ка по порядку.

Цикл for…to…do

Цикл `for…to…do` выполняется определенное количество раз, по счетчику. Счетчик представляет собой переменную целочисленного

типа - обычно используют `integer`, но если цикл должен выполниться 10-20 раз, вполне можно обойтись типом `byte`, чтобы не тратить впустую лишних 3 байта оперативной памяти. Синтаксис цикла следующий:

```
for <счетчик>:= <начальное значение> to <конечное значение> do <опер
```

Здесь, счетчик, как говорилось выше - переменная целого типа. Начальное и конечное значения - целые числа, например, от 1 до 10. Оператор - та часть кода, которую нужно выполнить нужное количество раз. Нередко бывает так, что нужно выполнить не один оператор, а несколько. В этом случае делают составной оператор, поместив весь нужный код между операторскими скобками `begin…end`:

```
for <счетчик>:= <начальное значение> to <конечное значение> do begin
  Оператор 1;
  Оператор 2;
  …
  Оператор n;
end;
```

При каждом очередном прохождении цикла счетчик автоматически увеличивается на единицу, и как только он достигнет конечного значения, цикл завершит свою работу. Давайте, опробуем этот инструмент на конкретном примере.

Загрузите Lazarus с новым проектом. Сохраните его в папку 10-01 там, где вы храните все учебные проекты. Проект назовите, скажем, MyCycles. Как назвать главную форму, вам уже, наверное, не нужно напоминать? Посреди окна установите простую кнопку TButton, переименовывать её не нужно, но в `Caption` напишите

```
for…do
```

Сгенерируйте для кнопки событие `OnClick`, в котором напишите следующий код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  b: byte;
begin
  for b:= 1 to 10 do ShowMessage('Проход цикла №' + IntToStr(b));
end;
```

Что, по-вашему, произойдет при нажатии на кнопку? Выйдет окошко с сообщением "Проход цикла №1". Как только вы нажмете <OK>, выйдет новое сообщение "Проход цикла №2". И так будет 10 раз, при этом `b` будет автоматически увеличиваться на 1. После того, как `b` станет равно 10, и цикл выполнится в последний раз, процедура `Button1Click` завершит работу и управление перейдет обратно главной форме.

В данном примере цикл `for` наращивал счетчик по возрастающей, от 1 до 10. Но у этого цикла есть ещё одна форма, когда счетчик не увеличивается, а уменьшается на единицу:

```
for <счетчик>:= <начальное значение> downto <конечное значение> do ·
```

Разница здесь только в том, что вместо ключевого слова `to` мы указываем `downto`. И, разумеется, в этом случае начальное значение счетчика должно быть больше конечного значения. Переделайте строку с циклом так:

```
for b:= 10 downto 1 do ShowMessage('Проход цикла №' + IntToStr(b));
```

Запустив программу на выполнение, вы убедитесь, что теперь счетчик уменьшается от 10 до 1.

Инструкции break и continue

Циклы не всегда нужно выполнять от начала, и до конца. Иногда, в зависимости от условий, бывает необходимо пропустить какие-то шаги цикла, или вовсе досрочно завершить цикл. Для этого и служат инструкции `break` и `continue`.

Break - инструкция досрочного завершения работы цикла.

Изменим пример процедуры `Button1Click`. Предположим, нам нужно вывести на экран результаты деления числа 100 на числа от -10 до 10. НО! На ноль делить нельзя, поэтому мы выполним проверку: если второе число - ноль, мы завершим цикл. А чтобы нам не пришлось много раз нажимать на <OK>, как в прошлом примере, результаты мы соберем в одну строку и в конце цикла разом выведем её на экран. Итак, код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  s: string; //для сбора результатов деления
  b: ShortInt; //счетчик
  r: real; //результат деления
begin
  for b:= -10 to 10 do begin  //начало цикла
    //если ноль, не делим, а сразу выходим из цикла:
    if b = 0 then break;
    // делим:
    r:= 100 / b;
    //теперь добавляем результат в строку s:
    s:= s + '100 / ' + IntToStr(b) + ' = ' + FloatToStr(r) + #13;
  end;  //конец цикла

  //теперь разом выводим все полученные результаты:
  ShowMessage(s);
end;
```

Я постарался дать подробные комментарии, но всё же остановимся на некоторых моментах. Прежде всего, бросается в глаза, что переменной `b` вместо предыдущего типа `byte` мы назначили тип `ShortInt`. Ведь нам нужно считать от -10 до 10, а `byte` имеет диапазон от 0 до 255, отрицательные величины этим типом не поддерживаются. Тип `ShortInt` также занимает 1 байт, но он имеет диапазон от -128 до 127, и в данном случае подходит нам больше всего. Если же мы попытаемся оставить счетчику тип `byte`, то компилятор просто выведет ошибку нарушения допустимого диапазона, и не соберет

исполняемую программу, не запустит её.

Далее, для получения результатов деления мы использовали переменную типа `real`. Ведь деление одного целого числа на другое не всегда даст целое число! Например, `100 / -8 = -12,5`. Поэтому и переменная для результата у нас вещественного типа.

Затем у нас начинается цикл. Вначале мы проверяем, не равна ли `b` нулю. Если равна, то сразу же завершаем цикл, при этом управление передается на следующий за циклом оператор

```
ShowMessage(s);
```

Если `b` не равна нулю, то цикл продолжает свою работу, выполняя деление 100 на `b`. Поскольку значение `b` будет изменяться при каждом проходе, то и результат все время будет другой. И так будет, пока `b` не поменяет значение от -10 до 0. Как только это случится, цикл закончится.

Самой сложной строкой цикла является оператор сбора строки `s`:

```
s:= s + '100 / ' + IntToStr(b) + ' = ' + FloatToStr(r) + #13;
```

Смотрите, как формируется эта строка. В начале выражения у нас стоит "`s + `", то есть, предыдущее значение `s` не теряется, а добавляется к новой строке, причем при первом проходе цикла `s` еще пуста, зато потом она уже будет содержать текст! Затем мы собираем строку из разных частей. Сначала идет часть "`100 / `", затем к ней добавляется значение счетчика `b`, преобразованное в строку, затем добавляется часть строки " = ". Потом к строке добавляется результат деления, преобразованный функцией `FloatToStr()` в строковое выражение. И в самом конце мы добавляем символ перехода на новую строку `#13`. Таким образом, у нас каждое выражение будет на новой строке, не смешиваясь в кашу. В результате нажатия на кнопку мы получим следующее сообщение:

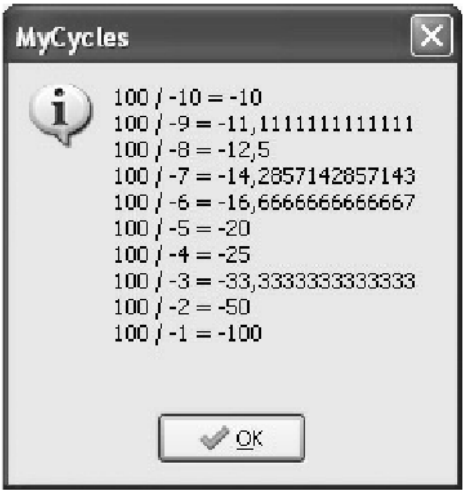


Рис. 10.1. Досрочный выход из цикла

Как видите, строка цикла

```
if b = 0 then break;
```

не позволила нам сделать деление на ноль, и мы избежали ошибки. Но тут сразу бросается незавершенность работы цикла. А как же целая часть, от 1 до 10?! Чтобы исправить положение, замените `break` на `continue`:

```
if b = 0 then continue;
```

Continue - инструкция пропуска оставшейся части цикла и переход к новому шагу цикла. Цикл при этом не завершается.

Нажав на кнопку, вы убедитесь, что теперь цикл выполняется до конца, от -10 до 10, пропустив лишь нулевое значение. Однако и тут дотошный пользователь сможет найти недостаток. Куда делся ноль то? Ну да, на ноль делить нельзя, но что-то ведь нужно вывести в этой позиции! Давайте поступим, как стандартный калькулятор Windows: в случае, если `b` равно нулю, выведем сообщение, что деление на нуль запрещено. Нам нужно только изменить условие `if`, сделав составной оператор:

```
  if b = 0 then begin
    s:= s + 'Деление на нуль запрещено';
    continue;
  end;
```

Как видите, в случае, если `b` равна нулю, мы в данной позиции выводим соответствующее сообщение, после чего пропускаем всю оставшуюся часть данного шага цикла (не делаем деление).

На всякий случай, полный код процедуры:

```
  procedure TfMain.Button1Click(Sender: TObject);
  var
    s: string; //для сбора результатов деления
    b: ShortInt; //счетчик
    r: real; //результат деления
  begin
    for b:= -10 to 10 do begin  //начало цикла
      //если ноль, не делим, а пропускаем шаг цикла с соответствующим со
      if b = 0 then begin
        s:= s + 'Деление на нуль запрещено' + #13;
        continue;
      end;
      //сначала делим:
      r:= 100 / b;
      //теперь добавляем результат в строку s:
      s:= s + '100 / ' + IntToStr(b) + ' = ' + FloatToStr(r) + #13;
    end;  //конец цикла

    //теперь разом выводим все полученные результаты:
    ShowMessage(s);
  end;
```

В результате мы получим такое сообщение:

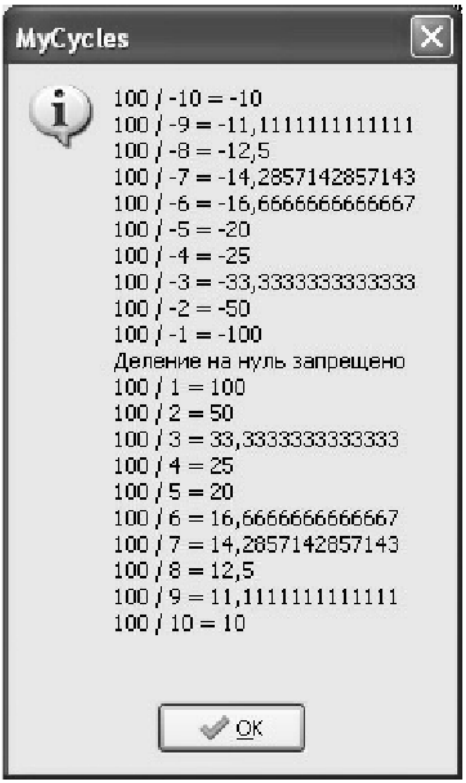


Рис. 10.2. Пропуск шага цикла

Условный цикл while…do

Слово while переводится как пока, а do - как делать. Другими словами, цикл `while…do` выполняется до тех пор, пока какое-то условие верно. Синтаксис:

```
while <условие> do <оператор>;
```

Здесь, условие - какое то условное выражение или логическая переменная. Если переменная или выражение имеют значение `true`, то цикл будет выполняться до тех пор, пока это значение не станет `false`. И только тогда цикл прекратит свою работу.

Если же значение условия изначально будет `false`, то цикл не будет

выполнен ни разу.

Оператор также может быть составным, если находится в скобках `begin…end`.

В цикле `while…do` также можно использовать операторы `break` и `continue`.

В этом цикле очень легко совершить ошибку, не предусмотрев внутри цикла изменения условия. Вот простейший пример:

```
var b: boolean;
begin
  b:= true;
  while b = true do
    ShowMessage('Истина');
```

Здесь цикл `while` будет выполняться до тех пор, пока `b` будет оставаться `true`. А поскольку внутри цикла не предусмотрено изменение значения `b`, цикл получился "зацикленным". Он будет выводить сообщение "Истина" до тех пор, пока у вас не сгорит компьютер, либо пока вы сами его со злости не разобьёте (шучу, у вас всегда есть возможность прервать зависшую программу).

Обратите внимание на строку

```
while b = true do
```

Я так написал, чтобы сразу было понятно, что тут к чему. Для цикла важно, чтобы условие было истинным. Поскольку `b` действительно равно `true`, то условие истинно. Но в этом случае можно было написать проще:

```
while b do
```

Ведь `b` итак содержит `true`! А зацикленный цикл можно написать еще понятней, вообще обойдясь без переменной:

```
while true do
```

Делать, пока истина, то есть, всегда. Все эти три примера будут равноценны. Однако вернемся к нашему проекту, и опробуем цикл на практике. Мы будем 100 делить на любое введенное пользователем число, и выводить результат. Причем будем делать это, пока пользователю не надоест, то есть, заранее неизвестно, сколько раз. Когда же ему надоест вводить числа, он введет ноль. Мы со своей стороны выведем сообщение, что на нуль делить нельзя, и на этом завершим работу цикла. Проверку на корректность введенного числа производить не будем. Пользователь сам будет контролировать введенное значение.

Чтобы не заниматься переделкой кода, добавим на форму еще одну кнопку, в `Caption` которой напишем

```
while…do
```

чтобы было понятно, какая кнопка с каким циклом работает. Для кнопки сгенерируем такое событие `OnClick`:

```
procedure TfMain.Button2Click(Sender: TObject);
var
  s: string; //для получения числа от пользователя
  r: real; //результат деления
  begin
    //начинаем цикл
    while true do begin
      //очистим строку:
      s:= '';
      //получим число от пользователя:
      InputQuery('100 / на...', 'Введите число', s);
      //если ноль, выводим сообщение и выходим из цикла:
      if s = '0' then begin
        ShowMessage('На нуль делить нельзя!');
        break;
      end;
      //если еще не вышли, значит не нуль. делим:
```

```
    r:= 100 / StrToFloat(s);
    //выводим результат:
    ShowMessage('100 / ' + s + ' = ' + FloatToStr(r));
  end;  //конец цикла
end;
```

Комментарии достаточно подробны, но все же проясним некоторые детали. Прежде всего, мы сделали "зацикленный" цикл:

```
  while true do begin
```

Ведь заранее мы не знаем, когда пользователю надоест вводить разные числа, и он введет ноль, чтобы завершить цикл. Но внутри цикла у нас предусмотрен досрочный выход из цикла:

```
    if s = '0' then begin
      ShowMessage('На нуль делить нельзя!');
      break;
    end;
```

Поэтому компьютер не зависнет - стоит пользователю ввести ноль, как цикл прекратится. Как пользоваться функцией `InputQuery()` вы, надеюсь, не забыли? Если забыли, вернитесь к лекции №6. Всё остальное должно быть понятно по предыдущим примерам.

Когда цикл начнется, пользователю выйдет запрос:

Рис. 10.3. Запрос пользователю

При вводе числа не забывайте о десятичном разделителе, который

зависит от языка операционной системы. В русских Windows это запятая, в английских - точка. Если введете неправильное (некорректное) число, программа зависнет. Сбросить отладку просто: <Запуск -> Сбросить отладчик>.

Условный цикл repeat…until

Цикл `repeat…until` - еще один условный цикл. Однако есть и отличия. Если `while…do` - цикл с предусловием, то `repeat…until` имеет постусловие. То есть, если в цикле `while…do` вначале проверяется условие, и в зависимости от этого условия принимается решение - выполнять ли цикл, то в `repeat…until` вначале выполняется цикл, а затем только проверяется условие. Другими словами, цикл `repeat…until` обязательно будет выполнен хотя бы однажды. Причем, опять таки, в отличие от `while…do`, цикл `repeat…until` будет выполняться до тех пор, пока условие ложно, и прекратит свою работу, когда условие станет истинным.

Англ. repeat переводится, как повторять, until - до тех пор, пока… Другими словами - выполнять какие то действия до тех пор, пока не выполнится условие. Синтаксис простой:

```
repeat <оператор(ы)> until <условие>;
```

Обратите внимание: ключевые служебные слова `repeat…until` работают, как операторские скобки `begin…end`, поэтому в тело цикла можно включить сколько угодно операторов. Проверим работу цикла на примере. Добавьте в проект третью кнопку, в `Caption` которой напишите

```
repeat…until
```

Код нажатия на кнопку будет следующий:

```
procedure TfMain.Button3Click(Sender: TObject);
var
  i: byte;
```

```
  s: string;
begin
  //зададим начальное значение счетчика:
  i:= 1;
  //теперь сам цикл:
  repeat
    s:= s + 'Квадрат от ' + IntToStr(i) + ' равен ' + IntToStr(i * i) + #13;
    i:= i + 1;
  until i > 10;
  //отчет:
  ShowMessage(s);
end;
```

Как видите, внутри цикла выполняется два оператора, при этом никаких скобок `begin…end` не используется. Внутри цикла мы обеспечиваем наращивание счетчика - если этого не сделать, программа зациклится, так как `i` никогда не станет больше 10. В нашем примере `i` увеличивается, и как только она станет больше 10, цикл прекращает работу, управление передается дальше, на `ShowMessage()`. В результате мы получим следующий отчет:

Рис. 10.4. Работа цикла repeat…until

Операторы `break` и `continue` также можно использовать в этом цикле.

Переключатель case

Переключатель `case` не относится к циклам, это оператор множественного выбора. Синтаксис:

```
case <Селектор> of
  <Значение 1>: <Оператор 1>;
 <Значение 2>: <Оператор 2>;
 …
 <Значение n>: <Оператор n>
 else <Оператор Иначе>;
end;
```

Здесь, `Селектор` - это переменная или выражение исчисляемого типа. То есть, как правило, это целое число или символ (для ПК символ - число, номер в таблице ANSI, то есть, исчисляемый тип), или выражение, которое возвращает в результате целое число или символ.

Далее, в зависимости от значения селектора, выполняется соответствующий оператор. Если нужно выполнить блок операторов, их заключают в операторские скобки `begin…end`.

Блок `else` не является обязательным, но если он есть, то этот блок выполняется в том случае, если значение селектора не соответствует ни одному проверяемому значению. Как обычно, перед `else` точка с запятой не ставится, а после `else` не нужно ставить двоеточие.

Для примера установим на форму последнюю, четвертую кнопку, в `Caption` которой напишем:

```
case
```

Код события нажатия на кнопку будет таким:

```
procedure TfMain.Button4Click(Sender: TObject);
var
  i: integer;
```

```
  s: string;
begin
  //очистим строку:
  s:= '';
  //получим число от пользователя:
  InputQuery('Квадрат целого числа', 'Введите целое число от 1 до 3:', s);
  //преобразуем строковое представление числа в число:
  i:= StrToInt(s);
  //теперь выбор case
  case i of
    1: ShowMessage('Квадрат от 1 равен 1');
    2: ShowMessage('Квадрат от 2 равен 4');
    3: ShowMessage('Квадрат от 3 равен 9')
    else ShowMessage('Неизвестное число, нет решения');
  end;
end;
```

Код достаточно понятен, чтобы его дополнительно комментировать. Когда пользователь нажмет на кнопку "case", выйдет запрос на ввод целого числа. Если ввести число от 1 до 3, то будет выполнен соответствующий оператор. Если целое число будет другим, выполнится оператор `else`. Если ввести не целое число, а что-то другое, то произойдет ошибка, которую можно будет снять, сбросив отладчик.

Если вам потребуется выполнить более одного оператора в каждом случае, то это можно оформить, например, так (код для примера, выполнять его вам не нужно):

```
case i of
  1 : begin
        ShowMessage('i = 1');
        f:= 10;
      end;
  2 : begin
        ShowMessage('i = 2');
        f:= 20;
      end;
```

```
  3 : begin
        ShowMessage('i = 3');
        f:= 30;
      end;
  else ShowMessage ('i не равно 1, 2, или 3');
end; //конец case
```

Собственно, этот же код можно было бы реализовать и посредством условного оператора `if`:

```
if i = 1 then …
else i = 2 then …
```

и так далее. Однако данный пример больше подходит для инструкции `case` - код получается более компактным, лучше читаемым. На сегодня это всё, данных инструментов будет достаточно, чтобы организовать сколь угодно сложную логику программы.

Экранная заставка

Эта лекция написана в виде лабораторной работы, и посвящена закреплению пройденного материала на практике. Описан весь путь создания полноценного рабочего приложения: экранной заставки в виде часов. Кроме того, в лекции описывается работа с системным таймером TTimer и с функцией-генератором случайных чисел Random().

Цель лекции

Закрепление пройденного материала на практике, изучение компонента `TTimer` и функции-генератора случайных чисел `Random()`.

Постановка задачи

Время от времени для закрепления пройденного материала мы будем выполнять лабораторные работы - реальные программные проекты различной сложности. На этот раз мы с вами сделаем экранную заставку в виде электронных часов. Проект разрабатывается на платформе Windows XP SP3, но и на других версиях Windows должно получиться также.

Прежде всего, экранная заставка занимает весь экран, не имеет системной строки, главного меню и обрамления главного окна. Да и окно у экранной заставки, как правило, одно. Еще одно различие - файл с экранной заставкой имеет расширение *.scr (от англ. screen - экран) а не *.exe, как программа. Тем не менее, экранная заставка - та же программа.

Подумаем, каким образом мы можем реализовать такую заставку в виде часов. Ответ напрашивается сам собой: установить в центре экрана обычную панель, на которой в свойстве `Caption` выводить текущее время. Однако, это скучно. Пусть эта панель еще и перемещается по экрану по случайной траектории! Чтобы реализовать такой проект, нам придется изучить пару вещей: системный таймер `TTimer` и функцию-генератор случайных чисел `Random()`.

Реализация проекта

Вообще-то, вначале положено разработать алгоритм программы.

Алгоритм - набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное число действий (из Википедии).

Но наш проект достаточно прост, поэтому алгоритма, как такового, нет. Начнем проект, при необходимости разрабатывая алгоритм дальнейших действий по ходу дела. Откройте Lazarus с новым проектом, свойство `Name` главной формы сразу переименуйте в `fMain`. Сохраните проект в папку 11-01 под именем MyClock, а модуль главной формы назовите `Main`.

Теперь займемся окном. Поскольку у окна заставки не должно быть системной строки, свойство `BorderStyle` установим в `bsNone`. А так как это окно должно занимать весь экран, свойство `WindowState` переведем в `wsMaximized`. Свойство `Color` установим в `clBlack`, чтобы форма стала черной. И ещё: свойство формы `Cursor` переведем в `crNone`, чтобы в работающей заставке не было видно указателя мыши (зачем она там?).

Пойдем далее. Установим на форму метку `TLabel`. Не обращайте внимания, что на черном фоне её пока не видно на форме. Изменим у неё некоторые свойства:

- `Name = lClock` (так мы будем обращаться к метке в программном коде; `l` - сокращение от `label` - метка, `Clock` - часы)
- `Caption = 00:00:00`
- `Font = Times New Roman`, жирный курсив, 48 пикселей. Цвет шрифта выберите желтый.
- `ParentColor = True` (это заставит метку всегда сливаться с формой, а цвет шрифта будет контрастировать с общим фоном)

Если вы забыли, как настроить свойства шрифта, то напомню: нужно выделить свойство `Font`, затем нажать на кнопку с тремя точками в

правой части свойства, в Инспекторе объектов:

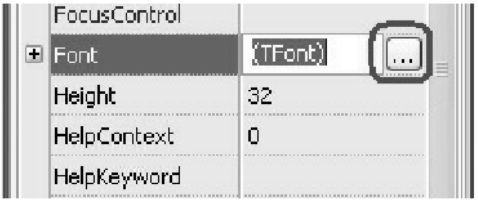


Рис. 11.1. Установка свойств шрифта

Откроется окно, где вы сможете выбрать шрифт, начертание, размер и цвет шрифта. Можете выбрать свои цвета для шрифта и панели, главное, чтобы они контрастировали друг с другом, и хорошо сочетались с черным цветом формы, который мы установим позже.

В результате выполненных действий у вас должно получиться примерно такое окно:

Рис. 11.2. Окно главной формы

Не волнуйтесь по поводу того, что окно пока маленькое, а системная строка и обрамления окна есть - когда программа будет работать, окно займет весь экран, если вы не напутали с параметрами формы.

Компонент TTimer

Пришла очередь таймера. Он нам нужен, чтобы заставить надпись метки меняться - показывать текущее время. Кроме того, благодаря этому компоненту мы сможем обеспечить метке непрерывное движение.

Таймер `TTimer` находится на вкладке System Палитры компонентов:

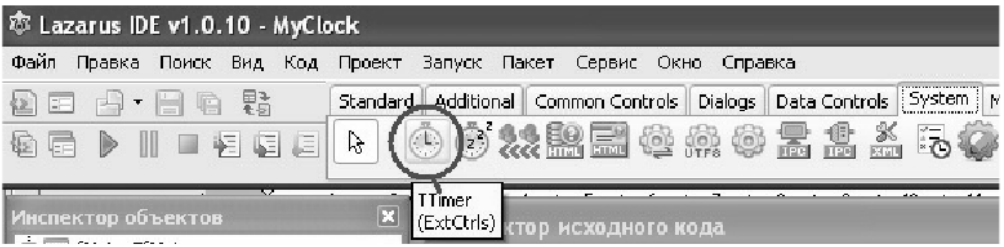


Рис. 11.3. Компонент TTimer

Компонент этот невизуальный - невидимый для пользователя, поэтому его можно установить в любое место на форме, лишь бы он вам не мешал при проектировании формы, не перекрывал другие компоненты. Компонент будет вам виден, несмотря на черный цвет формы.

У компонента есть всего по одному свойству и событию, которые нам могут понадобиться:

`Interval` - это свойство устанавливает интервал времени, когда таймер сработает. По умолчанию, он равен 1000, что соответствует 1 секунде. Это нам и нужно, поэтому оставим значение свойства без изменений. А на будущее запомните: если вам потребуется установить другое время, то умножайте требуемое количество секунд на 1000, и получите то, что нужно.

Далее, переходим в Инспекторе объектов на вкладку События. Нас интересует только одно событие:

`OnTimer` - событие срабатывает всякий раз, когда заканчивается установленный интервал. Сгенерируйте код для этого события так же, как ранее генерировали событие `OnClick` для кнопок, мы заполним

его чуть позже.

Вот теперь нам требуется продумать алгоритм действий - как мы будем двигать надпись. Идея такова: сгенерируем случайное число от 0 до 3, получится 4 варианта. Ведь у нас есть 4 направления, куда двигаться: вверх, вниз, влево или вправо. В зависимости от полученного случайного числа будем выбирать и направление движения. Двигать будем, скажем, на 50 пикселей влево-вправо, или на 25 пикселей вверх-вниз. А как двигать? Да очень просто! `Left` - это расстояние в пикселях от левого края формы до компонента. Допустим, у метки `Left` равен 100. То есть, от края формы до метки 100 пикселей. Если мы прибавим еще 50, то тем самым, сдвинем метку вправо, а если наоборот, отнимем, то сдвинем влево. Для движения вверх-вниз используем свойство `Top` - расстояние от верхней части формы до компонента. Движения будут аналогичные, но на 25 пикселей.

Если получим направление 0, то двигать будем вправо: прибавлять к свойству `Left` метки 50 пикселей. Если направление 1, то двигаем влево, отнимая 50 пикселей от свойства `Left`. Если направление 2, то двигаем вниз: прибавляем 25 пикселей к свойству `Top` метки. Ну и если направление 3, то двигаем вверх, прибавляя 25 пикселей к свойству `Top`.

Причем может случиться и так, что надпись окажется за пределами окна. В этом случае нам нужно будет вернуть её обратно в эти пределы. Сделаем это следующим образом: если метка ушла влево, то свойству `Left` просто присвоим значение 0 - метка вернется в пределы формы и окажется прижатой к левому краю.

Если метка ушла за правый край, то присвоим свойству `Left` ширину формы минус ширину метки, таким образом, метка окажется прижатой к правой части формы и будет в её пределах.

Точно таким же образом будем возвращать метку, если она уйдет вверх или вниз за пределы формы.

Значит, нам потребуется только одна целочисленная переменная, в которую мы будем генерировать направление - 4 позиции. Для этого используем функцию `Random()`, которую ещё называют генератором

случайных чисел.

`Random()` - функция возвращает случайное число (англ. random - случайный) от 0 до указанного количество чисел минус один. Синтаксис очень простой:

```
Random(<количество чисел>);
```

Функция вернет случайное число от 0 до <количество чисел> - 1. То есть, в случае

```
i:= Random(4);
```

в переменную `i` попадет случайное число от 0 до 3. Этим мы и воспользуемся. А чтобы не тратить лишнюю память, переменную `i` объявим с типом `byte`.

Несмотря на амбициозную идею, код события `OnTimer` будет довольно прост:

```
procedure TfMain.Timer1Timer(Sender: TObject);
var
  i: byte; //для получения случайного числа
begin
  //первым делом меняем надпись метки:
  lClock.Caption:= TimeToStr(Now);
  //получаем случайное направление:
  i:= Random(4);
  //теперь в зависимости от направления двигаем метку:
  case i of
    0: lClock.Left:= lClock.Left + 50; //вправо
    1: lClock.Left:= lClock.Left - 50; //влево
    2: lClock.Top:= lClock.Top + 50; //вверх
    3: lClock.Top:= lClock.Top - 50; //вниз
  end;
  //теперь проверяем: не вышла ли метка за пределы формы?
  //если вышла - возвращаем её обратно
  //если ушла влево:
```

```
    if lClock.Left < 0 then lClock.Left:= 0;
    //если ушла вверх:
    if lClock.Top < 0 then lClock.Top:= 0;
    //если ушла вправо:
    if (lClock.Left + lClock.Width) > fMain.Width then
      lClock.Left:= fMain.Width - lClock.Width;
    //если ушла вниз:
    if (lClock.Top + lClock.Height) > fMain.Height then
      lClock.Top:= fMain.Height - lClock.Height;
  end;
```

Вот и весь код! Вам может оказаться непонятной строка

```
  lClock.Caption:= TimeToStr(Now);
```

Таким образом, мы присваиваем свойству `Caption` метки `lClock` текущее время. Работе с датой и временем у нас будет посвящена следующая лекция, так что пока на этом не зацикливайтесь. Весь остальной код должен вам быть понятен.

Собственно, проверить работу заставки можно уже сейчас, только не забудьте предварительно сохранить изменения. Чтобы закрыть окно, у которого нет для этого кнопок и меню, воспользуйтесь стандартными кнопками Windows <Alt + F4>.

Но заставка на этом еще не закончена. Прежде всего, из-за включенной отладочной информации программа имеет большой размер, почти 15 мегабайт! Кроме того, мы получили исполняемый exe-файл, а нам нужна заставка с расширением *.scr. Исправляем эти недостатки:

1. Выбираем команду меню <Проект -> Параметры проекта>.
2. В разделе <Параметры компилятора> выбираем подраздел <Пути>.
3. В поле <Имя исполнимого файла (-o)> вместо MyClock указываем MyClock.scr (то есть, добавляем нужное расширение).
4. Отключаем флажок <Применять соглашения по именованию>, чтобы не было конфликта расширений.

5. В разделе <Параметры компилятора> переходим на подраздел <Компоновка>.
6. Отключаем флажок <Генерировать отладочную информацию для GDB>, чтобы уменьшить размер полученного файла.
7. Сохраняем проект, и заново его запускаем. В результате в папке 11-01 получаем требуемый файл MyClock.scr размером чуть больше 1,5 мегабайта.

Если вас устраивает, что для закрытия заставки приходится нажимать <Alt + F4>, то ничего больше делать не нужно. Но мне бы хотелось, чтобы эту заставку можно было закрыть кнопкой <Esc> (код которой 27 в таблице символов ANSI), как любую другую нормальную заставку. А для этого придется самую малость доработать проект.

Вернитесь в Редактор форм, и выделите саму форму. В Инспекторе объектов перейдите на закладку События и сгенерируйте событие формы `OnKeyPress`, которое срабатывает при нажатии на любую кнопку:

Рис. 11.4. Событие формы OnKeyPress

Его код совсем прост:

```
procedure TfMain.FormKeyPress(Sender: TObject; var Key: char);
begin
  //если нажали Esc, то выходим:
```

```
  if Key = #27 then Close;
end;
```

Сохраните проект и снова его запустите. Заставка теперь должна закрываться при нажатии клавиши <Esc>. Lazarus нам больше не нужен, можете его закрыть.

Установка заставки в Windows

Большинство из вас наверняка знает, как полученную заставку установить в своей Windows. Для тех, кто этого все-таки не знает, расскажу подробно:

1. Найдите файл MyClock.scr, он должен быть в папке 11-01, где вы храните проекты.
2. Скопируйте этот файл в папку C:\Windows
3. Щелкните правой кнопкой по свободному от окон месту Рабочего стола и выберите команду "Свойства".
4. В открывшемся окне "Свойства: Экран" перейдите на вкладку "Заставка".
5. Там выберите нашу заставку и требуемый интервал её появления, после чего нажмите <OK>:

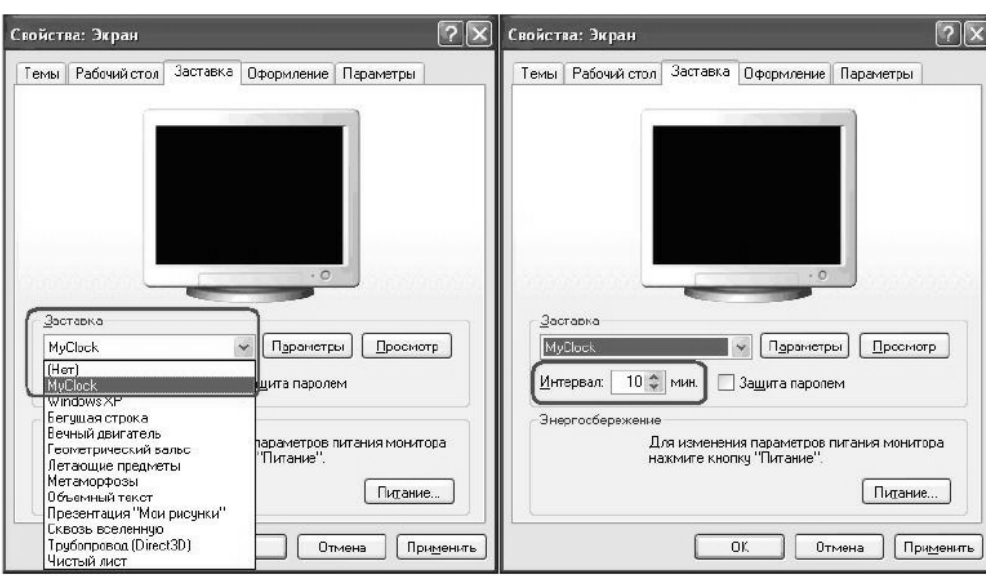


Рис. 11.5. Выбор заставки и интервала

Если пользователь в течение указанного интервала не будет двигать мышью и не нажмет никакой клавиши, то появится наша заставка. На сегодня это всё.

Дата и время

Лекция посвящена изучению работы с типом дата-время TDateTime. Изучены компоненты для работы с этим типом, рассмотрены все основные стандартные функции и процедуры для обработки даты-времени.

Цель лекции

Изучение типа `TDateTime`, компонентов, функций и процедур для работы с ним.

Тип TDateTime

Для работы с датой и временем в Lazarus предусмотрен тип `TDateTime`. Это универсальный тип, он позволяет одновременно определить и дату, и время. `TDateTime` занимает 8 байт оперативной памяти и фактически, является вещественным числом (`TDateTime = Double`): целая часть этого числа содержит дату, а десятичная - время. Можно также для работы с датой и временем использовать типы соответственно, `TDate` и `TTime`, но это просто синонимы того же типа `TDateTime`.

В справочнике Lazarus говорится, что минимальное возможное значения даты-времени равно -693593.0, а максимальное +2958465.99999. Целая часть такого числа обозначает количество суток, а дробная - часть суток, прошедших с 0 часов. Прибавление или вычитание единицы из целой части числа равносильно прибавлению или вычитанию одного дня. Число -693593 соответствует дате 01.01.0001, а число +2958465 соответствует дате 31.12.9999, в этом диапазоне вы и сможете работать с датами в Lazarus.

Компоненты для работы с датой-временем

На прошлой лекции нам уже довелось работать со временем, для вывода текущего времени на экран мы использовали простую метку `TLabel`. Но в Lazarus для ввода и вывода даты-времени имеются

специальные компоненты. Знакомиться с этими компонентами будем сразу на примерах. Загрузите Lazarus с новым проектом, сразу же сохраните его в папку 12-01.

Компоненты для работы с датой и временем располагаются на вкладке Misc Палитры компонентов. Вообще, очень интересные компоненты находятся на этой вкладке, и с некоторыми из них мы обязательно еще познакомимся. Сейчас же нас интересуют всего два компонента - календарь `TCalendar` и строка редактирования даты `TDateEdit`:

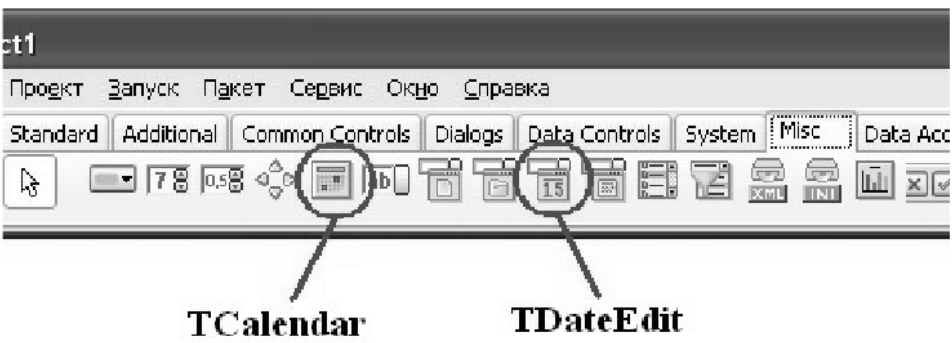


Рис. 12.1. Компоненты для работы с датой-временем TCalendar и TDateEdit

Правда, нужно заметить, компоненты эти еще сырые: некоторые свойства есть, но не работают, некоторые работают лишь частично, некоторых и вовсе нет. Так, в платном аналоге Lazarus - Delphi в компоненте `TDateEdit` есть возможность вводить как дату, так и время. В Lazarus можно вводить только дату. Для ввода времени, вероятно, придется использовать обычный `TMaskEdit`.

Поскольку Lazarus - проект новый, будем надеяться, что эти недоработки в будущем будут исправлены.

TCalendar

Вначале познакомимся с компонентом `TCalendar`. Установите его на форму, и он сразу примет вид календаря, причем текущая дата будет выделена. Выделение будет как красной окантовкой, так в календаре

выделяется текущая дата, так и синей заливкой - так в календаре выделяется выбранная дата. А поскольку выбрать можно любую дату, то они не всегда совпадают. Однако из формы управлять календарем не очень удобно. Сохраните проект и запустите его. В рабочем приложении календарем можно управлять: щелкая по синим стрелкам в левой правой частях заголовка календаря можно листать месяцы, щелкнув по названию месяца можно открыть список, в котором выбрать другой месяц, щелкнув по году, можно выбрать или ввести вручную другой год.

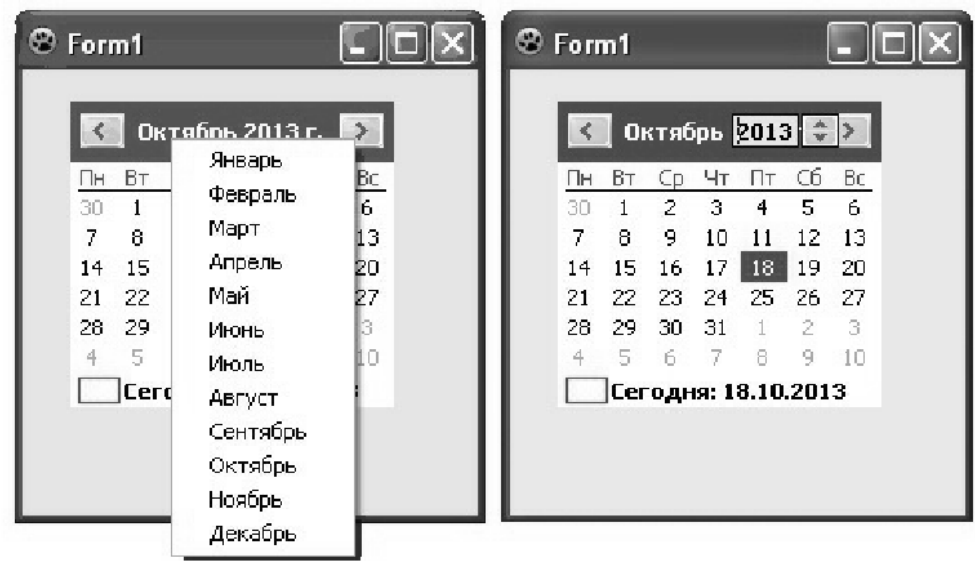


Рис. 12.2. Выбор месяца и года в календаре

Закройте запущенную программу и вернитесь в проект. Познакомимся с некоторыми полезными свойствами компонента `TCalendar`.

`Align`	- выравнивание компонента относительно формы (или другого контейнера). С этим свойством вы уже знакомы, но в случае с календарем применять его не имеет смысла - если еще можно прижать календарь к одной из сторон, то изменить его размеры все равно не удастся.
	- раскрывающееся сложное свойство. По идее,

`BorderSpacing`	оно должно настраивать параметры обрамления: ширину обрамления вокруг календаря или по одной из его сторон, но, увы, это свойство тоже не работает - как я не пытался что-то изменить, вид календарь остался прежним.
`Date и DateTime`	- установленная в календаре дата. При изменении одного из этих свойств, изменится и другое, но есть и отличия: `Date` имеет тип `String`, а `DateTime` - `TDateTime`. Таким образом, если вам нужно получить дату в виде строки, обращайтесь к `Date`. Если надо работать с датой в форме `TDateTime`, а это часто бывает нужно, например, для расчета дат, то обращайтесь к `TDateTime`. Присвоить календарю новую дату можно как через первое, так и через второе свойство, но если вы будете менять дату с помощью `Date`, вы должны быть уверены, что дата в строке написана правильно.
`DisplaySettings`	- свойство должно управлять видом календаря, но тоже почему-то не изменяет его. Имеет следующие подсвойства, разрешающие (при `True`) или запрещающие (при `False`) различные действия: • `dsNoMonthChange` - Не разрешать изменять месяц • `dsShowDayNames` - Показывать имя дня недели • `dsShowHeadings` - Показывать заголовок • `dsShowWeekNumbers` - Показывать номер недели, в году их 52 (это подсвойство установить можно). • `dsStartMonday` - Начинать неделю с Понедельника (в английских и американских календарях неделю принято начинать с Воскресенья).

Перейдем к событиям компонента (переключитесь на вкладку События в Инспекторе объектов). Здесь нам могут понадобиться следующие события:

- `OnChange` - событие наступает при любом изменении в компоненте: изменили год, месяц или день.
- `OnClick` - щелкнули по компоненту мышью.
- `OnDayChanged` - изменился день.
- `OnDblClick` - дважды щелкнули по компоненту мышью.
- `OnMonthChanged` - изменился месяц.
- `OnYearChanged` - изменился год.

TDateEdit

Теперь изучим компонент `TDateEdit`. Установите его на форму, где-нибудь под календарем. Как видите, фактически компонент состоит из двух компонентов: поля для редактирования даты и кнопки, при нажатии на которую появляется календарь:

Рис. 12.3. Компонент TDateEdit

Теперь перейдите на вкладку Свойства в Инспекторе объектов. У компонента имеются следующие полезные свойства:

`ButtonOnlyWhenFocused`	- Показывать кнопку выбора даты только когда фокус ввода находится на компоненте. По умолчанию `False`, то есть, кнопку видно всегда. Честно говоря, не вижу смысла прятать эту кнопку.
`ButtonWidth`	- ширина кнопки. По умолчанию равна 23, но можно и изменить. Правда, опять непонятно зачем.

`CalendarDisplaySettings`	- аналогично свойству `DisplaySettings` календаря, и предназначено для того же: для изменения внешнего вида календаря, который появляется при нажатии на кнопку.
`CancelCaption`	- заголовок для кнопки Отмена в диалоговом окне Календарь. Надо полагать, речь идет о привязанном к компоненту диалоге `TCalendarDialog` (диалоги мы будем изучать в другой лекции). Мне лично не удалось найти, как диалог привязать к компоненту, поиски информации в Интернете также были безуспешны. Будем надеяться, в следующих версиях Lazarus этот компонент заработает, как надо.
`DateOrder`	- формат вывода даты. При выборе формата, отличного от `doNone` (по умолчанию), появляется маска даты. Может иметь следующие значения: • `doDMY` - последовательность даты: День, Месяц, Год. • `doMDY` - последовательность даты: Месяц, День, Год. • `doNone` - последовательность даты соответствует системной. Для русской Windows это обычно День, Месяц, Год. Маски в этом формате не предусмотрено, если пользователь вводит дату сам, не пользуясь выбором в календаре, то легко может ошибиться. Поэтому

	рекомендую использовать значение `doDMY`, соответствующее русскому представлению даты. • `doYMd` - последовательность даты: Год, Месяц, День.
`DefaultToday`	- если `True`, то в работающей программе компонент по умолчанию отображает текущую дату, если `False`, то не отображает ничего.
`OKCaption`	- заголовок для кнопки OK в диалоговом окне Календарь. Проблемы те же, что и у `CancelCaption`.

В Инспекторе объектов почему то не отображаются следующие необходимые свойства:

`Date`	- имеет тип `TDateTime`, содержит выбранную или введенную пользователем дату.
`Text`	- имеет тип `TCaption`, который совместим с обычной строкой. Тут выбранная или введенная дата хранится в виде строки.

Этими свойствами часто придется пользоваться, чтобы узнать, какую же дату выбрал или ввел пользователь, так что запомните их.

Основные события, которые вам могут понадобиться, это `OnChange` (при изменении), `OnClick` (щелкнули мышью), `OnKeyPress` (нажали любую клавишу). События эти работают так же, как в других, уже известных вам компонентах, так что задерживаться тут мы не будем.

Стандартные функции для работы с датой и временем

Ознакомимся с основными функциями, которые помогут нам работать с датой и временем. Для выполнения примеров советую установить на форму простую кнопку, и сгенерировать для нее событие `OnClick`, в

котором и будете пробовать предлагаемые примеры. Итак, начнем.

`Date` - Функция возвращает текущую дату в формате `TDateTime`. Часть числа после запятой (временная часть) будет равна нулю. Пример применения:

```
var  dt: TDateTime;
begin
  dt:= Date;  //получили в dt текущую дату
  …
```

`Time` - Функция возвращает текущее время в формате `TDateTime`. Часть числа перед запятой (часть даты) будет равна нулю. Пример применения аналогичен функции `Date`.

`Now` - Функция возвращает текущие дату и время в формате `TDateTime`. Обе части числа будут заполнены значениями. Пример применения аналогичен совместным функциям `Date` и `Time`.

`DateTimeToStr()` - Функция принимает в качестве параметра дату и время в формате `TDateTime` и возвращает эти данные в виде строки. Синтаксис функции:

```
function DateTimeToStr(DateTime: TDateTime): String
```

Пример вывода текущих даты времени на экран:

```
begin
  ShowMessage(DateTimeToStr(Now));
```

В результате на экран выйдет сообщение с текущими датой и временем, например:

19.10.2013 9:25:57

`DateToStr()` - Функция принимает в качестве параметра дату в формате `TDateTime` и выводит её в виде строки. Время при этом не

выводится. Применение функции аналогично `DateTimeToStr`.

`TimeToStr()` - Функция принимает в качестве параметра время в формате `TDateTime` и выводит его в виде строки. Дата при этом не выводится. Применение функции аналогично `DateTimeToStr`.

Обратите внимание, что форматы вывода даты-времени зависят от того, какая Windows у вас установлена, и какие правила действуют в вашей стране. Так, указанное сообщение сформировалось по правилам России, для русской Windows. А вот в США, например, другие правила. Там эта строка выглядела бы так:

2013.10.19 9:25:57AM

Все эти правила зависят от системных переменных `ShortDateFormat` (краткий формат даты), `LongDateFormat` (длинный формат даты), `ShortTimeFormat` (краткий формат времени) и `LongTimeFormat` (длинный формат времени). Объявлять эти переменные не нужно, они уже присутствуют в системе и содержат строки с соответствующими форматами (о форматах мы поговорим ниже). Так, для русской Windows краткий формат дат установлен, как `dd.MM.yyyy` (то есть, день с ведущим нулем, месяц и год в четырех цифрах). Краткий формат времени будет `h:nn` (час без ведущего нуля и минуты с ведущим нулем), а длинный - `h:nn:ss` (час без ведущего нуля, минуты и секунды с ведущим нулем). При желании, эти системные переменные можно изменить прямо в программе, например:

```
begin
  ShortDateFormat:= 'dd.mm.yy';
```

В результате, год будет выводиться в двух цифрах, а не в четырех, т.е. вместо "2013" будет "13". Однако я не рекомендую это делать. Когда вам понадобится (а это обязательно произойдет) вывести дату и время в желаемом вам, или вашему заказчику, виде, то лучше воспользоваться функциями, реализующими форматный вывод, речь о которых пойдет ниже. Тем более что это гарантирует правильный вывод даты-времени на любой Windows, в любой стране.

`FormatDateTime()` - функция выводит заданную дату и время в требуемом вам формате, который не зависит от того, какая Windows (русская или английская) у вас установлена. Синтаксис функции следующий:

```
function FormatDateTime(const Format: String; DateTime: TDateTime): String;
```

В функцию следует передать два параметра: строку с требуемым форматом, и дату-время (или только дату, или только время) в формате `TDateTime`. Функция вернет эти данные в указанном формате. Например:

```
var
  s: String;
begin
  s:= FormatDateTime('dd mmmm yyyy - hh:nn:ss', Now);
```

В результате в строковой переменной `s` окажется строка типа "`21 Октябрь 2013 - 11:41:56`". Но это в теории. На практике вместо русских букв выходят знаки "`?`" - сказываются проблемы совместимости Lazarus с различными кодировками Windows. Что-ж, решение есть и из этой ситуации: нужно конвертировать полученную строку функцией `SysToUTF8()`, которая получает ANSI-строку, и гарантированно возвращает формат `UTF8`, в котором, как мы знаем, и работает Lazarus:

```
s:= SysToUTF8(FormatDateTime('dd mmmm yyyy - hh:nn:ss', Now));
```

Указанный программистом формат может содержать символы, которые представлены в таблице ниже:

Таблица 12.1. Символы формата функции FormatDateTime()

Символы	Описание
`c`	Отображает короткий формат даты и длинный формат времени, то есть, в русской Windows это будет в виде "дд.мм.гггг чч:мм:сс", например: "21.10.2013 11:54:56".
`d`	Отображает день месяца (число) без ведущего нуля (1-31)

dd	Отображает день месяца (число) с ведущим нулем (01-31)
ddd	День недели в сокращенном виде, например: Пн, Вт для русских Windows.
dddd	День недели в полном виде, например, понедельник, вторник.
ddddd	Короткий формат даты, например: "21.10.2013"
dddddd	Длинный формат даты, например: "21 Октябрь 2013 г." - для вывода даты в документ или на экран самое то.
m	Номер месяца без ведущего нуля (1-12).
mm	Номер месяца с ведущим нулем (01-12).
mmm	Краткое название месяца, например: "окт".
mmmm	Полное название месяца, например: "Октябрь".
y или yy	Год из двух цифр, например: "13"
yyyy	Год из четырех цифр, например: "2013"
h	Час без ведущего нуля (0-23)
hh	Час с ведущим нулем (00-23)
n	Минуты без ведущего нуля (0-59)
nn	Минуты с ведущим нулем (00-59)
s	Секунды без ведущего нуля (0-59)
ss	Секунды с ведущим нулем (00-59)
z	Вывод миллисекунд (0-999). Например, формат "hh:nn:ss:z" выведет время примерно так: "12:36:01:745"
t	Короткий формат времени, например: "12:16"
tt	Длинный формат времени, например: "12:16:53"
am/pm	Время будет выводиться так, как принято на западе: 12-часовое представление, после которого указывается am (до полудня) или pm (после полудня). Эти символы добавляются к требуемому формату, например, формат "t am/pm" выведет время в виде "12:21 pm"
a/p	Аналогично предыдущему, но вместо am (до полудня) или pm (после полудня) будут выходить a или p.

`DateTimeToString()` - процедура, которая преобразует указанные дату-время в строку с использованием указанного формата, и результат помещает в строковую переменную, которая указывается, как параметр по ссылке. Указанный формат использует символы из таблицы 12.1, как и `FormatDateTime()`.

Синтаксис:

```
procedure DateTimeToString(var Result: String; const Format: String; DateTime:
```

Пример применения:

```
var
  s: String;
begin
  DateTimeToString(s, 'dddddd', Now);
  ShowMessage(SysToUTF8(s)); //тут необходимо преобразовать строку в
```

Как видите, для корректного вывода русского текста нам пришлось преобразовать его из ANSI в UTF8 прямо внутри оператора `ShowMessage()`, так как в процедуре `DateTimeToString` использовать `SysToUTF8()` невозможно. Собственно, можно было вначале преобразовать строку, а уж затем вывести ее:

```
  …
  DateTimeToString(s, 'dddddd', Now);
  s:= SysToUTF8(s);
  ShowMessage(s);
```

Но так пришлось бы делать лишнюю строку кода.

`DayOfWeek()` - функция принимает в качестве параметра дату, и возвращает номер дня недели этой даты в виде целого числа. Причем делает это она так, как принято на западе: первым днем недели считается воскресение, последним - суббота. Так что понедельник будет вторым днем недели. Синтаксис:

```
function DateOfWeek(Date: TDateTime): Integer;
```

Если же вам потребуется вывести день недели словом, а не числом, то можно поступить, например, так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: String;
begin
  case DayOfWeek(Now) of
    1: s:= 'Воскресение';
    2: s:= 'Понедельник';
    3: s:= 'Вторник';
    4: s:= 'Среда';
    5: s:= 'Четверг';
    6: s:= 'Пятница';
    7: s:= 'Суббота';
  end; //case

  ShowMessage('Сегодня ' + FormatDateTime('ddddd', Now) + ', ' + s);
end;
```

Вначале с помощью case мы получили в s требуемый день недели. Затем мы вывели сообщение, типа "Сегодня 21.10.2013, Понедельник".

`DecodeDate()` - очень полезная процедура. Она разбивает указанную дату отдельно на год, месяц и число, возвращая эти данные в виде целых чисел. Синтаксис:

```
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);
```

Пример применения:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Year, Month, Day: Word;
  s: String;
```

```
begin
  DecodeDate(Now, Year, Month, Day);
  ShowMessage('Год - ' + IntToStr(Year) + #13 +
                  'Месяц - ' + IntToStr(Month) + #13 +
                  'День - ' + IntToStr(Day));
end;
```

В данном примере мы получили в виде чисел год, месяц и число текущей даты в соответствующие переменные. Затем вывели их на экран, преобразовав с помощью `IntToStr()` в строки, каждое данное на отдельной строке.

`EncodeDate()` - функция, которая является противоположностью `DecodeDate`. Она напротив, собирает дату из отдельных чисел, и возвращает её, преобразовав в формат `TDateTime`. Синтаксис:

```
function EncodeDate(Year, Month, Day: Word): TDateTime;
```

Пример применения:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Year, Month, Day: Word; //для указания года, месяца и дня
  dt: TDateTime; //для получения даты
  s: String;
begin
  //указываем год, число и день:
  Year:= 1999;
  Month:= 3;
  Day:= 9;
  //получаем дату:
  dt:= EncodeDate(Year, Month, Day);
  //теперь выводим ее:
  ShowMessage('Указали ' + SysToUTF8(FormatDateTime('dddddd', dt)));
end;
```

Комментарии достаточно подробны, чтобы вы разобрались с кодом. В

результате выполнения примера будет выведено сообщение "Указали 9 Март 1999 г.". Вы можете указать и другие год, число и день, только помните, что указывать их нужно правильно. Например, нельзя в Феврале указывать 30 дней, а в Апреле 31. И еще: если вы попытаетесь использовать дату, меньшую чем 01.01.0001, или большую чем 31.12.9999, то воспользоваться этими функциями вам не удастся, программа выведет ошибку.

`DecodeTime()` - Процедура разбивает указанное время отдельно на часы, минуты, секунды и миллисекунды, возвращая эти данные в виде целых чисел. Синтаксис:

```
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);
```

Поскольку процедура работает аналогично `DecodeDate`, приводить примеры мы не будем.

`EncodeTime()` - а эта функция работает аналогично `EncodeDate()`, и собирает время из отдельных чисел, возвращая его в формате `TDateTime`. Синтаксис:

```
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;
```

`IsLeapYear()` - функция, которая иногда бывает нужна. Она определяет, был ли год високосным, или нет. Год указывается в виде целого числа. Функция возвращает `True`, если указанный год високосный, и `False` в противном случае. Синтаксис:

```
function IsLeapYear(Year: Word): Boolean;
```

Пример применения:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Year: Word;
begin
  //указываем год:
```

```
  Year:= 1917;
  //високосный или нет?:
  if IsLeapYear(Year) then ShowMessage('Указан високосный год')
  else ShowMessage('Указан невисокосный год');
end;
```

Напоследок остались функции преобразования даты-времени из строки в `TDateTime`, они тоже очень часто бывают нужны.

`StrToDateTime()` - функция преобразует указанные в виде строки дату и время в формат `TDateTime`. Синтаксис:

```
function StrToDateTime(const S: string): TDateTime;
```

Пример:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  dt: TDateTime;
begin
  dt:= StrToDateTime('21.10.2013 14:25:30');
  ShowMessage('Указали ' + FormatDateTime('c', dt));
end;
```

Обратите внимание вот на что. В этой функции строку нужно составлять правильно, в зависимости от языка вашей Windows. Дата должна указываться в кратком формате, от времени дата отделяется пробелом. Сепаратором (разделителем частей даты) может быть как точка - в русской Windows, так и слеш "/" в английской, это зависит от переменной `DateSeparator`. Три части даты (y-год, m-месяц и d-день) могут быть в виде `y/m/d, m/d/y, d/m/y` - порядок этих частей зависит от переменной `ShortDateFormat`.

`StrToDate()` - функция преобразует указанную в виде строки дату в формат `TDateTime`, время при этом не указывается. Синтаксис:

```
function StrToDate(const S: string): TDateTime;
```

`StrToTime()` - функция преобразует указанное в виде строки время в формат `TDateTime`, дата при этом не указывается. Синтаксис:

```
function StrToDate(const S: string): TDateTime;
```

Функции действуют аналогично `StrToDateTime`, так что примеры приводить не буду, на сегодня это все.

Массивы простые, многомерные и динамические

Лекция посвящена изучению массивов. Подробно рассматриваются простые, многомерные и динамические массивы, а также функции для работы с ними. Материал подкрепляется практическими примерами.

Цель лекции

Изучение простых, многомерных и динамических массивов.

Массив

В практике программирования нередко возникает необходимость обработать большое количество однотипных данных. Допустим, распределить недельную прибыль кафе по дням. Что для этого нужно? Использовать переменные типа `currency`, как раз для таких случаев и применяемые. И сколько переменных нам нужно? Ну, ясен вопрос - по переменной на день, всего семь штук. Давайте посмотрим, как такую задачу можно было бы решить на Lazarus (пример академический, выполнять его не нужно):

```
var
  day1, day2, day3, day4, day5, day6, day7: currency;
```

Довольно длинное объявление получилось, правда? А если бы нужно было посчитать не недельную прибыль, а месячную? Или годовую? Теперь предположим, что нам нужно посчитать сумму этой прибыли. Это выглядело бы примерно так:

```
pribil:= day1 + day2 + day3 + day4 + day5 + day6 + day7;
```

А если бы пришлось считать месячную или годовую прибыль? Какой длины оператор тогда бы получился? А ведь часто бывают и более сложные расчеты, которые требуется провести с каждой однотипной переменной.

К счастью, в языках программирования существует такой инструмент, как массивы.

Массив - это формальное объединение нескольких однотипных переменных в единый набор, с одним общим именем.

Фактически, получается цепочка однотипных переменных под общим именем, где у каждой такой переменной имеется собственный номер в ячейке - индекс. Такие переменные называются элементами массива, каждый элемент имеет свой номер. Объявляется массив в разделе переменных следующим образом:

```
var
  a: array[1..7] of currency;
```

Ключевое слово `array` указывает, что это будет массив. Числа в квадратных скобках разделяются двумя точками и обозначают диапазон индексов. Так, [1..7] означает, что индексы массива будут от 1 до 7, то есть, 7 штук. После ключевого слова `of` указывается тип массива, такой же тип будет и у каждого элемента массива. В результате, компилятор создает в оперативной памяти семь ячеек по 8 байт каждая (тип `currency` занимает 8 байт):

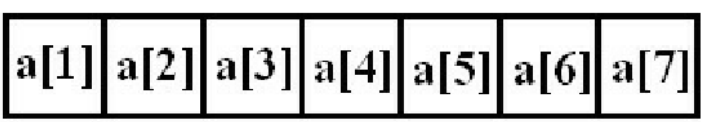


Рис. 13.1. Размещение элементов массива a в памяти

Тип данных, используемый в массивах, так же может быть любым. Во многих других языках программирования диапазон индексов всегда начинается с 0. В языках, основанных на Паскале, диапазон может начинаться с любой цифры: с 0, 1, или 438, если вам так нравится. Хотя программисты обычно начинают диапазон либо с 0, либо с 1, в зависимости от обстоятельств. Пример объявления различных массивов:

```
var
```

```
  name: array[0..30] of string;
  year: array[1..2013] of integer;
  man: array[10..25] of boolean'
```

Хорошо, массивы мы объявили. Что с ними делать дальше? Мы можем обратиться к любому элементу массива, указав после имени в квадратных скобках индекс этого элемента. Так, мы можем присвоить значение элементу или получить его значение. Другими словами, мы можем обращаться с элементами массива, как с отдельными переменными. Примеры:

```
…
a[1]:= 3520.45;
a[2]:= a[1] + 12.4;
ShowMessage('Прибыль за пятницу составила  ' + FloatToStr(a[5]));
…
```

Ну, ладно, мы научились создавать и использовать массивы. Но какие же преимущества у массивов в отличие от простых однотипных переменных? О первом таком преимуществе мы уже говорили, вот как просто можно объявить массив с элементами, рассчитанными на год:

```
var
  a: array[1..365] of currency;
```

Попробуйте-ка объявить 365 отдельных переменных! Второе преимущество гораздо важней: обрабатывать элементы массива намного проще, чем отдельные переменные. Например, нам нужно создать программу, которая бы могла сделать перевод температур из привычных нам Цельсиев в непривычные американские Фаренгейты. И пусть диапазон возможных температур будет от -100 градусов до +200. Идея такова: пользователь вводит нужную температуру, а программа выводит результат на экран. Конечно, проще было бы делать нужный расчет в момент, когда пользователь введет свои данные, и нажмет кнопку "рассчитать". Но мы поступим по-другому: мы сначала сделаем все расчеты и поместим результаты в массив, а затем будем просто выводить нужные данные. Массив сделаем глобальным, чтобы

заполнять его в одном событии, а выводить данные в другом. Открываем Lazarus с новым проектом, сохраняем его сразу в папку 13-01 там, где у нас хранятся все учебные проекты. Проект сохраним под именем MyTemp, а модуль главной формы, как обычно - Main. Займемся главной формой, измените у нее следующие параметры:

- `Name = fMain`
- `Caption = Перевод температурных шкал`
- `BorderStyle = bsDialog`
- `Position = poMainFormCenter`

Далее, перейдите на вкладку Additional Палитры компонентов, найдите и установите на форму компонент `TStaticText`, который предназначен для вывода на экран пояснительного текста, в том числе и многострочного. Если вы забыли, то найти нужный компонент просто: нужно просто подвести указатель мыши к компоненту, и через короткое время появится всплывающая подсказка с именем компонента.

В принципе, для этих целей можно использовать и простую метку `TLabel`, но было бы интересней познакомиться и с другими компонентами. В Инспекторе объектов в свойстве `Caption` нажмите на кнопку "..." справа от свойства:

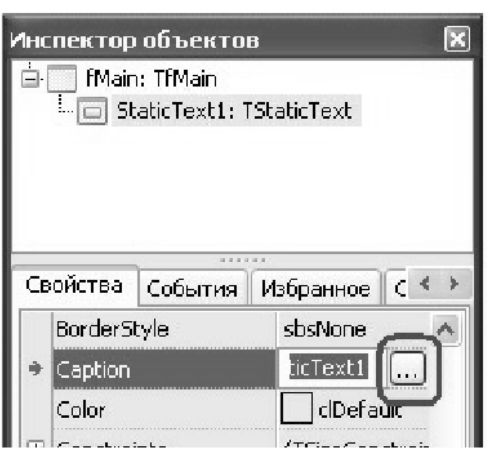


Рис. 13.2. Свойство Caption компонента TStaticText

В открывшемся редакторе напишите следующий текст:

Укажите температуру в Цельсиях
от -100 до 200 градусов:

Именно так, в две строки. Затем измените размеры компонента на форме, чтобы текст отображался точно в таком виде, как мы ввели. И еще: в свойстве `Alignment` выберите `taRightJustify`, чтобы текст выравнивался по правому краю.

Теперь перейдите на вкладку Misc палитры компонентов, найдите там и установите на форму, правее `TStaticText`, компонент `TSpinEdit`, который предназначен для ввода пользователем целых чисел. Этот компонент похож на простой `TEdit`, но справа от него есть стрелки вверх и вниз, которыми пользователь может прибавлять или убавлять указанное в строке число. Использование этого компонента гарантирует нам, что пользователь не сможет ввести в него ничего, кроме целого числа. А чтобы еще гарантировать правильный диапазон этих чисел, в свойстве `MaxValue` (максимальное возможное значение) установите 200, а в `MinValue` (минимальное возможное значение) установите -100 (минус сто). Теперь пользователь не сможет ввести в это поле ничего, кроме целых чисел. Если же он попытается превысить максимальное значение, то компонент автоматически установит число 200 - максимально возможное. И наоборот, если указать число меньше, чем -100, компонент установит -100, так что нам не нужно беспокоиться об ошибках. Кроме того, в этом компоненте позже нам понадобятся ещё два свойства: `Text` (установленное значение в виде строки), и `Value` (установленное значение в виде целого числа).

По умолчанию, компонент недостаточно широк, поэтому установите в его свойстве `Width` значение 75. И еще: нам придется обращаться к компоненту по имени, а `SpinEdit1` - это слишком длинное имя. Измените его свойство `Name` на `SE1`.

Ну и наконец, установите на форму простую кнопку, в свойстве `Caption` которой напишите Рассчитать температуру. Надпись на кнопке не уместится, поэтому кнопку придется удлинить. Расположите компоненты на форме примерно так:

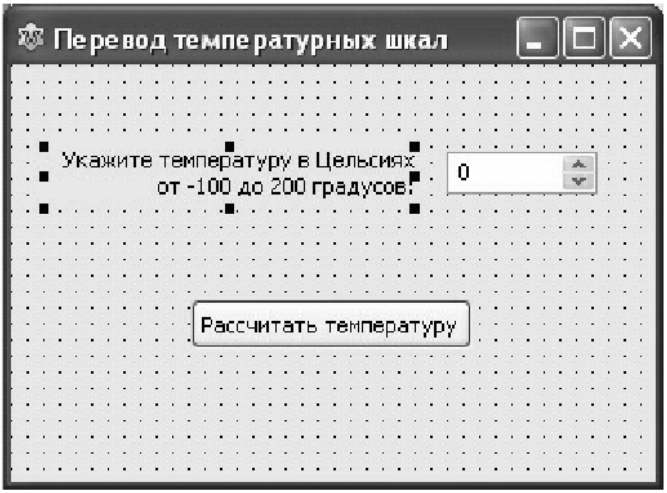


Рис. 13.3. Внешний вид формы

Теперь займемся предварительными расчетами. Для начала нам нужно сделать глобальный массив вещественных чисел - температура по Фаренгейту уже не будет целым числом. Значение по Фаренгейту вычисляются так: берется температура по Цельсию, умножается на 9/5, затем к результату прибавляется 32. Поэтому в глобальном разделе `var`, сразу над ключевым словом `implementation` объявляем массив `atemp`:

```
var
  fMain: TfMain;
  atemp: array[-100..200] of real;  //наш глобальный массив

implementation
```

Поскольку это глобальный массив, он будет создан сразу же при загрузке программы, и будет существовать, пока программа не завершит свою работу. Теперь займемся заполнением этого массива.

Выделите форму, щелкнув по её свободному месту, затем в Инспекторе объектов перейдите на вкладку События. Нам нужно событие `OnCreate` - оно возникает только однажды, при создании формы. Щелкните дважды по этому событию, чтобы сгенерировать его. Код события следующий:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  i: smallint;  //счетчик для цикла
begin
  for i:= -100 to 200 do
    atemp[i]:= i * 9/5 + 32;
end;
```

Вот теперь, наконец, мы сможем оценить всю прелесть использования массивов! Давайте разберем этот код. В разделе `var` мы объявили переменную `i` типа `smallint`. Это - счетчик для цикла `for`, который мы использовали в коде. Так как диапазон нужных нам значений располагается от -100 до +200, тип целых чисел `smallint` - самый подходящий. А теперь обратите внимание на код:

```
for i:= -100 to 200 do
  atemp[i]:= i * 9/5 + 32;
```

Это - самая главная "фишка" массивов - обработка всего массива, каким бы большим он не был, в простом цикле. Вначале счетчик `i` принимает значение -100. Внутри цикла мы присваиваем значение одному элементу массива. Если мы заменим переменную `i` на её значение, как это делает компилятор в процессе работы, то получим следующее:

```
atemp[-100]:= -100 * 9/5 + 32;
```

Таким образом, мы высчитываем перевод температуры из Цельсия в Фаренгейты для конкретной температуры: -100 градусов по Цельсию, и полученный результат присваиваем элементу массива `atemp[-100]`. Затем цикл повторяется, `i` увеличивается на единицу, и уже равна -99. Новое значение присваивается новому элементу массива, и так до конца цикла. Как вы думаете, сколько элементов массива будет заполнено в этом цикле? Кто сказал 300?! А про ноль вы забыли? 301 раз будет работать цикл, и 301 различное значение будет присвоено 301 элементу массива. А теперь представьте, как это выглядело бы для отдельных переменных:

```
t_100:= -100 * 9/5 + 32;
t_99:= -99 * 9/5 + 32;
…
t200:= 200 * 9/5 + 32;
```

Вместо двух строчек цикла мы получили бы 301 строку кода с одинаковыми вычислениями!

Но вернемся к нашему проекту, ведь нам еще нужно вывести на экран результат перевода температур. Сгенерируем для кнопки Button1 событие `OnClick`. Её код будет совсем простой:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  ShowMessage(SE1.Text + ' Цельсия = ' +
                    FloatToStr(atemp[SE1.Value]) + ' Фаренгейта');
end;
```

Здесь у нас всего один оператор - процедура `ShowMessage()`. Она формирует выводимое сообщение из 4 частей: `SE1.Text` - это то значение, которое установил пользователь в компоненте `TSpinEdit` в виде строки. Затем мы вставляем в сообщение текст `' Цельсия = '`. Обратите внимание на пробелы вначале и в конце этого кусочка - без них слова слипались бы, а это некрасиво. Далее, мы добавляем к строке следующий код:

```
FloatToStr(atemp[SE1.Value])
```

Здесь `SE1.Value` - это то значение, которое установил пользователь, но уже в виде целого числа! Допустим, пользователь установил значение 35, тогда этот код будет таким:

```
FloatToStr(atemp[35])
```

В элементе массива `atemp[35]` у нас уже хранится переведенное в Фаренгейты значение температуры 35 по Цельсию. А функция

`FloatToStr()` только переведет это значение в строку. Таким образом, для значения 35 сформируется сообщение: "`35 Цельсия = 95 Фаренгейта`". Сохраните проект, скомпилируйте его и убедитесь, что программа корректно работает в указанных пределах температур, а превысить их пользователь не сможет, если и захочет, как не сможет ввести ничего другого, кроме целого числа.

Многомерный массив

Массивы, которые мы до сих пор разбирали, были одномерными - визуально их можно представить, как цепочку ячеек или одну строку таблицы. Однако бывают ситуации, когда этого недостаточно. Допустим, нужно сохранить какую-то таблицу. Одномерный массив с такой задачей не справится. Для этого существуют многомерные массивы, которые имеют размерность от двух и больше. Например, объявить двухмерные массивы из 5 строк и 10 колонок можно следующим образом:

```
var
  a: array[1..5, 1..10] of string;
  b: array[0..4, 0..9] of integer;
```

То есть, в квадратных скобках через запятую указывается диапазон вначале строк, а затем колонок. Визуально, такой массив можно было бы представить так:

Таблица 13.1. Представление двухмерного массива a

1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10
5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10

Обратиться к отдельному элементу массива можно двумя способами:

```
  a[2, 9]:= 'Строка 1';  //1-й способ
  a[3][10]:= 'Строка 2';  //2-й способ
```

В первом случае мы обращаемся к элементу во второй строке, девятой колонке. Во втором - к третьей строке, десятой колонке. Первый способ обращения к элементу кажется компактней, его мы и будем использовать впредь.

Доработаем наш проект. Что, если мы будем конвертировать нашу температуру не только в Фаренгейты, но и в Кельвины (единица термодинамической температуры в СИ - Международной Системе Единиц)? Пересчет Цельсия в Кельвины еще проще, чем в Фаренгейты: нужно к температуре по Цельсию прибавить 273,15. Для этого нам потребуется вместо одномерного массива объявить двухмерный. Измените код над служебным словом `implementation`:

```
  atemp: array[1..2, -100..200] of real;
```

В первой строке у нас будут Фаренгейты, во второй - Кельвины. А колонки мы отведем под требуемый диапазон. Далее, нам потребуется изменить цикл в событии `OnCreate` формы:

```
  procedure TfMain.FormCreate(Sender: TObject);
  var
    i: smallint;
  begin
    for i:= -100 to 200 do begin
      atemp[1, i]:= i * 9/5 + 32;  //Фаренгейты
      atemp[2, i]:= i + 273.15;  //Кельвины
    end;
  end;
```

Обратите внимание: поскольку нам в цикле требуется выполнить не один оператор, а два, нам пришлось заключить их в программные скобки `begin…end`, сделав составной оператор. Далее, все просто: для строки 1 нашего массива мы рассчитываем Фаренгейты, как в прошлый раз. А для строки 2 уже Кельвины.

Но нам потребуется переделать и событие кнопки `OnClick`:

```
procedure TfMain.Button1Click(Sender: TObject);
begin
  ShowMessage(SE1.Text + ' Цельсия будет равно:' + #13 +
              FloatToStr(atemp[1, SE1.Value]) + ' Фаренгейта' + #13 +
              FloatToStr(atemp[2, SE1.Value]) + ' Кельвина');
end;
```

С кодом, надеюсь, вы разберетесь самостоятельно? Итак, если пользователь введет, например, 36 градусов Цельсия, то нажав на кнопку, получит следующее сообщение:

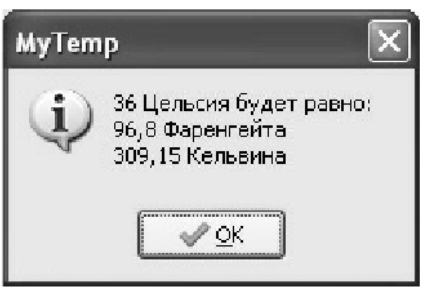


Рис. 13.4. Сгенерированное программой сообщение

Но размерность массива может быть и больше. Трехмерный массив визуально можно представить следующим образом:

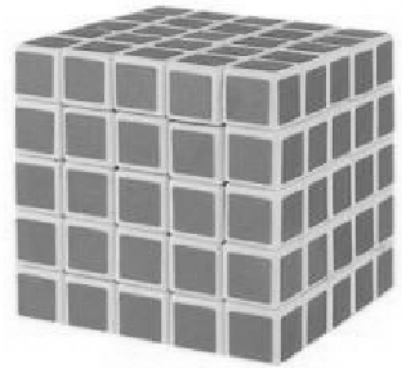


Рис. 13.5. Визуальное представление трехмерного массива

Объявить подобный массив можно было бы так:

```
var
  a: array[1..5, 1..5, 1..5] of integer;
```

А обращаться к отдельным элементам так:

```
a[2, 3, 4]:= 12;
```

Как бы выглядел четырехмерный массив визуально вообще невозможно представить. У вас есть возможность работать хоть с десятимерным массивом, однако в практике программирования обычно используют одно- и двухмерные массивы, и очень редко возникает надобность в трехмерном. Не забывайте про один из основных принципов программирования, который сами программисты называют KISS (Keep It Simple, Stupid - будь проще, дурачок). Этот принцип подразумевает, что код программы не стоит усложнять без нужды - сложный код тяжелее воспринимается самим программистом, больше нагружает процессор, а ошибки времени выполнения (run time errors - ошибки, возникающие во время выполнения программы, обычно это логические ошибки) в таком коде сложнее отслеживать.

Динамический массив

В практике программирования иной раз случается работать с массивами, размерность которых заранее неизвестна - она может зависеть от данных, введенных пользователем, от текущих параметров, от объема обрабатываемой информации, да еще много от чего. Программист в этих случаях не может указать конкретные границы индексов массива. В таких случаях используют динамические массивы.

Динамическими называются массивы, при объявлении которых размер не указывается. А во время выполнения программы размер такого массива можно изменять.

Объявляются динамические массивы также в разделе `var`, следующим образом:

```
var
  da: array of integer;
```

Как видите, мы указываем только тип массива, но не его размерность. Память под объявленный массив при этом не отводится. В дальнейшем с динамическим массивом мы можем производить различные операции: устанавливать размерность (длину) массива, увеличивать или уменьшать эту размерность (потому массив и динамический), узнавать высшие и низшие границы диапазона индексов массива, присваивать значения отдельным элементам или наоборот, считывать эти значения. Познакомьтесь с функциями, которые все это проделывают.

`SetLength()` - устанавливает размер массива. Синтаксис:

```
SetLength(<массив>, <длина>);
```

Пример:

```
SetLength(da, 5);  //установили размер массива в 5 элементов
```

Тут следует оговориться, что в отличие от обычного массива, начальный индекс которого может быть любым, индексация динамического массива всегда начинается с нуля. То есть, индексация элементов массива в нашем примере будет от 0 до 4 - всего пять элементов. Как только мы установили размер массива, в памяти выделяется место под него. В нашем примере будет отведено по 4 байта (тип `integer`) для 5 элементов массива. Обращаться к этим элементам можно обычным образом:

```
da[0]:= 5;
da[4]:= da[0];
```

Теперь мы можем прибавить элемент к массиву, установив для него новый размер:

```
SetLength(da, 6);
```

В этом случае, один пустой элемент будет добавлен в конец массива. Если же напротив, мы уменьшим его размер:

```
SetLength(da, 4);
```

то массив усекается. Если последний элемент содержал какие-то данные, они будут потеряны. Освободить память массива можно, присвоив ему нулевую размерность:

```
SetLength(da, 0);
```

Также освободить память можно, присвоив массиву значение `nil` (ничего, пусто):

```
da:= nil;
```

Впрочем, этого можно и не делать: по окончании процедуры, где данный массив был объявлен, память будет освобождена

автоматически.

Динамические массивы могут быть и многомерными. Например, двухмерный массив целых чисел можно объявить так:

```
var
  a: array of array of integer;
```

Затем такому массиву можно присвоить размерность, например, 4 на 5:

```
SetLength(a, 4, 5);
```

Это будет, как если бы мы объявили простой массив:

```
var
  a2: array[0..3, 0..4];
```

`Length()` - возвращает размер динамического массива, то есть, количество его элементов. Например, нам нужно посмотреть размер массива, и если он пустой, то добавить в него один элемент:

```
if Length(da) = 0 then SetLength(da, 1);
```

`Low()` - возвращает нижний индекс массива, у динамических массивов это всегда ноль.

`High()` - возвращает верхний индекс массива, но это не то же самое, что количество элементов. Если у массива 5 элементов, то `Length()` вернет 5, а `High()` вернет 4, так как индексация начинается с нуля.

Пример обхода массива от первого до последнего элемента:

```
for i:= Low(da) to High(da) do
  da[i]:= i * i;
```

В примере мы каждому элементу присваиваем квадрат его индекса.

Давайте поработаем с динамическими массивами на практике. Чтобы не портить наш конвертер температур, закройте его и откройте новый проект. Мудрить мы не будем - нам нужна только форма и простая кнопка на ней. Переименовывать тоже ничего не будем, это же просто пример. Сохраните новый проект в папку 13-02.

Сгенерируйте процедуру `OnClick` для кнопки, её код будет таким:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;  //для запроса
  da: array of string;  //динамический массив строк
  i: integer;  //счетчик
begin
  //узнаем у пользователя, сколько строк делать:
  s:= '0';
  InputQuery('Привет!', 'Сколько строк желаете создать?', s);
  //если ноль, то ничего не делаем, выходим из процедуры:
  if StrToInt(s) = 0 then Exit;

  //иначе устанавливаем указанную размерность массива:
  SetLength(da, StrToInt(s));
  //теперь обойдем весь массив, устанавливая в его элементы значения:
  for i:= Low(da) to High(da) do
    da[i]:= 'Строка №' + IntToStr(i + 1);

  //теперь снова обойдем массив и создадим сообщение из его строк:
  s:= '';
  for i:= Low(da) to High(da) do
    s:= s + da[i] + #13;
  ShowMessage(s);

end;
```

Код содержит достаточно подробные комментарии. Вначале в строковую переменную s мы получаем количество желаемых строк. Так как функция `InputQuery()` возвращает только строку, нам придется получать целое число в виде строки - никакой проверки мы здесь не

делаем, это же только демонстрация работы с динамическим массивом. Поэтому правильность вводимого числа оставим на совести пользователя.

Далее, мы проверяем - не ноль ли это? Если строк создавать не нужно, мы просто выходим из процедуры, пропуская весь остальной код. Если же пользователь ввел какую-то цифру, причем она должна быть больше нуля, то мы устанавливаем указанную размерность массива:

```
SetLength(da, StrToInt(s));
```

Далее, с помощью цикла `for` мы обходим весь массив от первого до последнего элемента. В каждый элемент мы записываем строку: `"Строка №"`, добавляем номер элемента и переход на новую строку. Обратите внимание, для первой строки `i` будет равно 0, поэтому мы указываем `i + 1`:

```
for i:= Low(da) to High(da) do
  da[i]:= 'Строка №' + IntToStr(i + 1);
```

Затем мы очищаем строковую переменную `s`, чтобы вторично воспользоваться ей, собрать в неё сообщение. Что и делаем в следующем цикле:

```
for i:= Low(da) to High(da) do
  s:= s + da[i] + #13;
```

Получается, в процедуре мы дважды обходили массив: сначала чтобы занести в него значения, затем чтобы считать их. Теперь сохраните проект, скомпилируйте его и запустите на выполнение. Имейте в виду, если вы укажете слишком большое число, то окно сообщения не уместится в экране - в этом случае закрыть его можно будет стандартными клавишами <Alt + F4>.

Коллекции (массивы) строк и компоненты для них

В лекции рассматривается тип TStrings, который является базовым типом массивов строк, а также компоненты TMemo, TListBox и TComboBox, которые используют этот тип. Изучаются основные возможности TStrings. Материал подкрепляется практической работой, в которой учащиеся знакомятся со способами обработки строк: их редактированием, сохранением в текстовый файл, считыванием из файла и прочее.

Цель лекции

Изучение типа `TStrings` и компонентов `TMemo`, `TListBox` и `TComboBox`.

Компонент TMemo

До сих пор для предоставления пользователю возможности ввести текст мы использовали компоненты вроде `TEdit`, с помощью которых можно ввести только одну строку текста. Компонент `TMemo` - это, по сути, целый текстовый редактор! Не верите? Давайте посмотрим. Откройте Lazarus с новым проектом. Сразу же переименуйте форму в fMain (свойство `Name`), в свойстве `Caption` напишите `Мой Блокнот`. Сохраните проект как MyNotebook в папку 14-01, а модулю дайте имя `Main`. Установим некоторые другие параметры формы. Прежде всего, форма будет содержать редактор текстов, поэтому она не должна быть маленькой. Давайте сделаем следующие установки в свойствах формы:

```
Height = 450
Width = 750
Position = poDesktopCenter
```

Далее, на вкладке Standard Палитры компонентов найдите компонент `TMemo`:

Рис. 14.1. Компонент TMemo

Установите его на форму. Теперь давайте рассмотрим, как следует, свойства компонента `TMemo` - уже известные по другим компонентам свойства мы рассматривать не будем, остановимся только на новых и специфичных для `TMemo` (компонент должен быть выделен).

`BorderStyle`	- Включает/отключает обрамление вокруг компонента, и может быть: • `bsNone` - нет обрамления • `bsSingle` - есть обрамление (по умолчанию)
`CharCase`	- задает регистр выводимых в компоненте символов. Может быть: • `ecLowerCase` - все символы строчные • `ecNormal` - символы такие, какими их вводит пользователь (по умолчанию) • `ecUppercase` - все символы прописные
`Color`	- цвет текстового поля. Вы можете задавать цвет как текстового поля, так и шрифта, однако имейте в виду, что серьезные программы не делаются с яркими попугайскими расцветками, поэтому желательно компонентам оставлять их стандартные цвета.
	- шрифт текста в компоненте. Свойство имеет кнопку "…", с помощью которой его можно раскрыть и настроить шрифт: название шрифта, его

`Font`	размер, начертание, цвет и видоизменение. Насчет расцветок шрифта здесь справедливо то же замечание, что и для свойства `Color`. Начертание (простой шрифт, курсив, полужирный, полужирный курсив) тоже без нужды изменять не стоит. А вот что касается названия шрифта и его размера - их изменить можно. Дело в том, что по умолчанию в компоненте установлен мелкий шрифт, а у некоторых пользователей может быть слабое зрение. Чтобы пользователь зря не напрягал глаза, я обычно устанавливаю шрифт Times New Roman (он выглядит привлекательней) 12-го размера. Шрифт получается достаточно крупным. Вы можете поэкспериментировать с разными шрифтами и размерами и подобрать то, что нравится лично вам. Только еще одно замечание: если вы полагаете, что ваша программа будет работать и на других компьютерах, то желательно выбирать стандартные шрифты. Если вы установите какой-то экзотический шрифт, которого у пользователя может и не быть, то программа у него не будет работать нормально. Пользователю придется дополнительно искать и устанавливать ваш шрифт, а они обычно не терпят таких неудобств, и скорее предпочтут какую-то другую, аналогичную программу.
`HideSelection`	- включает или выключает выделение текста, когда компонент теряет фокус. Если равно True, то выделение текста скрывается, когда активным становится другой компонент. При `False` выделение остается. Обычно оставляют значение по умолчанию - `True`.
	- коллекция (массив) строк. Позволяет обращаться как ко всему тексту, так и к его отдельным строкам. Это сложное свойство, которое имеет свои свойства, методы и события. Методы - это то, что

`Lines`	компонент умеет делать. Это те же самые процедуры и функции. `Lines` это свойство, но и у него есть методы, с помощью которых можно сохранять текст в файл, считывать его из файла, делать еще много других полезных вещей. Имеет тип `TStrings`. Мы вернемся к нему чуть позже, фактически, вся эта лекция посвящена работе с типом `TStrings`.
`MaxLength`	- максимальная длина вводимого пользователем текста. По умолчанию, свойство равно нулю. Это означает, что ограничений на размер текста нет.
`ReadOnly`	- переводится как "Только для чтения". По умолчанию, равно `False` - пользователь может как читать, так и редактировать текст. Если установить в `True`, пользователь не сможет изменять текст. Так делают, когда нужно вывести для пользователя многострочный текст, например, текст лицензии или описание программы в окне "О программе". Мы оставим False, так как сейчас нам нужно, чтобы пользователь мог редактировать текст.
	- полосы прокрутки компонента. Может быть: • `ssNone` - нет полос прокрутки. • `ssHorizontal` - горизонтальная полоса прокрутки, располагается по нижнему краю компонента. Неактивна, пока текст строки умещается на экране. Когда он становится больше, полоса становится активной, и пользователь может прокрутить текст по горизонтали. • `ssVertical` - вертикальная полоса прокрутки, располагается по правому краю компонента. Неактивна, пока все строки умещаются на экране. Когда строк больше, полоса прокрутки становится активной, и

`ScrollBars`	пользователь может прокручивать текст вверх-вниз. • `ssBoth` - присутствуют как вертикальная, так и горизонтальная полосы прокрутки. • `ssAutoHorizontal` - отличается от `ssHorizontal` тем, что пока текст умещается в окне, полоса прокрутки невидима. По крайней мере, так задумывалось разработчиками. На деле же, `ssAutoHorizontal` ведет себя так же, как и `ssHorizontal` - полоса видима, но неактивна, пока текст умещается в окне. Будем надеяться, что в будущем разработчики Lazarus исправят эту ошибку. • `ssAutoVertical` - то же, что и `ssAutoHorizontal`, но для вертикальной полосы прокрутки. • `ssAutoBoth` - то же, что и `ssAutoHorizontal`, но для обеих полос прокрутки.
`WantReturns`	- тоже недоработанное свойство. По идее, если его установить в `False`, пользователь не сможет клавишей <Enter> вставлять разрывы строк. Однако новые строки по-прежнему вставляются клавишей <Enter> вне зависимости от установок этого свойства.
`WantTabs`	- вот это свойство работает. Если оно установлено в `False` (по умолчанию), то пользователь не сможет клавишей <Tab> вставлять знаки табуляции - эта клавиша будет переключать фокус с `TMemo` на другой компонент, если он есть. Если же свойство установить в `True`, то пользователь сможет вставлять знаки табуляции. Разрешать ему вставку табуляции или нет, зависит от назначения компонента. Если вам нужно простое поле, чтобы пользователь мог ввести несколько строк текста, то

	табуляцию можно и не разрешать. Если же вам нужно сделать полноценный текстовый редактор, то табуляция необходима.
WordWrap	- разрешает или запрещает перенос текста на новую строку. Если установлено True (по умолчанию), то когда текст достигает края, происходит автоматический переход на новую строку. Если установлено False, то строка будет продолжаться, пока пользователь не нажмет <Enter>. Обычно свойство оставляют в True и устанавливают вертикальную полосу прокрутки. Если вам захочется установить False, позаботьтесь о том, чтобы была установлена и горизонтальная полоса прокрутки.

В Инспекторе объектов отсутствует еще одно важное свойство: Text. Это свойство имеет тип String и содержит весь текст, все строки разом, включая символы перехода на новую строку.

Итак, займемся настройкой компонента TMemo. Имя оставим по умолчанию: Memo1. Свойства Left и Top установите в 5. Свойство Height установите равным 400, а Width - 740. Компонент занял почти всю форму, оставив небольшие бордюры и место для кнопок снизу. Теперь нам нужно привязать компонент к сторонам, чтобы он автоматически изменял размер при изменении пользователем размеров окна. Для этого разверните свойство Anchors (якоря) и установите в True все его четыре подсвойства.

Далее, нам как-то нужно убрать текст "Memo1" из компонента, который был добавлен в поле автоматически. Для этого выделите свойство Lines и щелкните по её кнопке "...". Это приведет к открытию окна диалога ввода строк. Сотрите там весь текст и нажмите "OK" - текст в компоненте пропал.

Теперь установите вертикальную полосу прокрутки (ScrollBars = ssVertical) и убедитесь, что в WordWrap установлено True - компонент будет автоматически переносить текст на новые строки.

Вот, собственно, и все настройки. Теперь нам нужно добиться от компонента трех вещей: чтобы он сохранял текст в файл, чтобы он считывал текст из файла, и чтобы он очищал весь текст. Для этого нам придется воспользоваться некоторыми методами свойства `Lines` компонента. И еще нам потребуются три простых кнопки, для которых внизу как раз осталось место. Давайте первую кнопку назовем `bSave` (в свойстве `Name`), вторую - `bRead`, и третью - `bClear`. Маленькая буква `b` означает кнопку (`button`), хотя вы можете изобрести и свои правила наименования компонентов. Нам потребуется изменить и свойство `Caption` у этих кнопок: в первой кнопке напишите `Сохранить`, во второй - `Читать`, а в третьей - `Очистить`. Чтобы текст умещался на кнопках просторней, свойство `Width` (ширину) кнопок поставьте в 90. Теперь вот еще что: по умолчанию, все компоненты "привязываются" к верхней и левой границам формы. Если пользователь будет изменять размеры окна, то кнопки будут находиться на таком же расстоянии от левого и верхнего края, что и при проектировании. А так как наш `Memo` привязан ко всем сторонам, и будет увеличиваться, то закроет собой кнопки. Чтобы этого избежать, выделите все кнопки, откройте их свойство `Anchors`, `akTop` переведите в `False`, а `akBottom` наоборот, в `True`. Теперь наши кнопки привязаны к левому и нижнему краю, и всегда будут на одинаковом расстоянии от `Memo`.

У вас должна получиться такая форма:

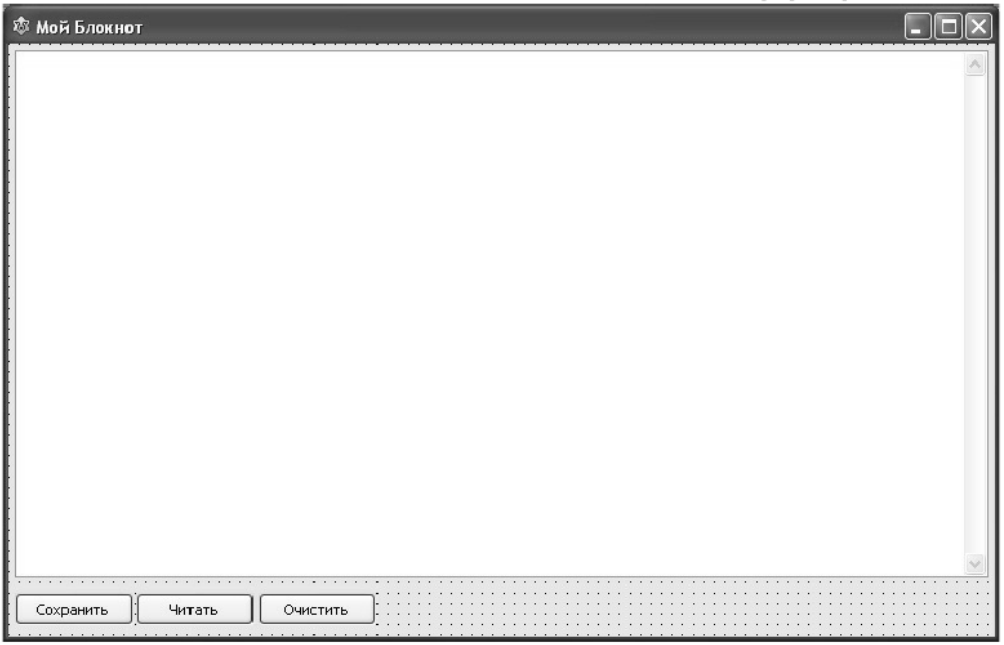


Рис. 14.2. Форма редактора текстов

Осталось запрограммировать события `OnClick` для этих кнопок. Сгенерируйте их, дважды щелкая по кнопкам. Вот код этих событий:

```
procedure TfMain.bSaveClick(Sender: TObject);
begin
  Memo1.Lines.SaveToFile('MyText.txt');
end;

procedure TfMain.bReadClick(Sender: TObject);
begin
  if FileExists('MyText.txt') then
    Memo1.Lines.LoadFromFile('MyText.txt')
  else ShowMessage('Файл MyText.txt не существует');
end;

procedure TfMain.bClearClick(Sender: TObject);
begin
  Memo1.Lines.Clear;
end;
```

Как видите, события очень просты, в двух из них всего по одной строке. Так, в строке

```
Memo1.Lines.SaveToFile('MyText.txt');
```

мы обращаемся к компоненту `Memo1`, его свойству `Lines`, и вызываем метод этого свойства `SaveToFile()`.

Давайте разберемся с методами свойства `Lines` компонента `TMemo`.

`SaveToFile()` - сохранить текст в указанный файл, в нашем случае, это файл MyText.txt. Расширение *.txt традиционно используется для текстовых файлов, так что его лучше указывать, хоть это и не является обязательным. В примере мы указали просто имя файла, без адреса. Файл будет создан в текущей папке, там же, где и программа. Если вы хотите указать какое то конкретное место, то указывайте имя файла вместе с адресом, например:

```
Memo1.Lines.SaveToFile('C:\MyText.txt');
```

Если такого файла не было, он будет создан. Если файл был, он перезапишется. Это будет обычный текстовый файл, который можно открыть любым текстовым редактором.

`LoadFromFile()` - этот метод наоборот, считывает текст из указанного файла в компонент. Однако файл должен существовать на самом деле, иначе программа вызовет ошибку. Именно поэтому вначале мы делаем проверку на существование файла:

```
if FileExists('MyText.txt') then
  Memo1.Lines.LoadFromFile('MyText.txt')
else ShowMessage('Файл MyText.txt не существует');
```

Если указанный файл существует, функция `FileExists()` вернет `True`. Если же файла по какой-то причине нет, то никакого чтения не будет. Вместо этого выйдет сообщение, что файл не существует, и процедура завершит работу.

`Clear` - очищает текст в `Memo`. В примере мы обратились к свойству `Lines`, но можно обратиться и к самому компоненту, так как у него тоже есть метод `Clear`. Вместо

```
Memo1.Lines.Clear;
```

можно написать просто

```
Memo1.Clear;
```

Результат будет одинаков. Сохраните проект, запустите, и опробуйте его в работе. Не забудьте поэкспериментировать с изменением размера окна, чтобы посмотреть, как будут выглядеть "привязанные" к сторонам формы компоненты.

Кроме вышеперечисленных инструментов, `TMemo` имеет и некоторые другие полезные методы. Сейчас мы говорим о методах именно компонента `TMemo`, а не его свойства `Lines`, имейте это в виду!

`CopyToClipboard`	- копирует выделенный в компоненте текст в Буфер обмена. Пример: `Memo1.CopyToClipboard;`
`CutToClipboard`	- вырезает выделенный текст и помещает его в Буфер обмена. Пример: `Memo1.CutToClipboard;`
`PasteFromClipboard`	- вставляет в компонент текст из Буфера обмена в позицию, где находится курсор. Пример: `Memo1.PasteFromClipboard;`

SelectAll	- выделить весь текст. Пример: Memo1.SelectAll;
Undo	- отменить последние изменения в тексте. Пример: Memo1.Undo;

Отдельно стоит сказать и о событиях компонента TMemo. Выделите его, и в Инспекторе объектов перейдите на вкладку События. Нас интересуют только основные события, которые могут нам пригодиться в работе. Примеры я приводить не буду, события генерируются так же, как OnClick, которым мы неоднократно пользовались.

OnChange	- событие происходит, когда текст в компоненте изменен. Например, пользователь ввел или наоборот, удалил символ, загрузил текст из файла или еще каким то образом изменил содержимое Memo.
OnClick	- событие происходит, когда пользователь щелкает по компоненту мышью.
OnDblClick	- событие происходит, когда пользователь дважды щелкает по компоненту мышью.
OnEnter	- событие происходит, когда компонент становится активным. Или, как говорят, получает фокус ввода.
OnExit	- событие происходит, когда компонент теряет фокус ввода (активным становится другой компонент).
OnKeyDown	- событие возникает, когда пользователь нажимает, но еще не отпускает клавишу на клавиатуре. Обработчик обычно используют для

	распознавания нажимаемой клавиши.
`OnKeyPress`	- событие возникает, когда пользователь нажал и отпустил клавишу на клавиатуре. Обработчик обычно используют для распознавания нажатой клавиши. Если пользователь ввёл недопустимый символ, его можно запретить или заменить другим символом.
`OnKeyUp`	- событие возникает, когда пользователь отпускает клавишу на клавиатуре. Обработчик обычно используют для распознавания нажатой клавиши.
`OnUTF8KeyPress`	- то же, что и `OnKeyPress`, но не для ANSI-, а для UTF8-символов. При работе с русскими символами нужно использовать это событие.

Сохраните проект, скомпилируйте и запустите его. Опробуйте редактирование, сохранение текста в файл, чтение его из файла.

Списки выбора TListBox и TComboBox

В Lazarus есть еще два компонента, которые используют тип `TStrings`, или говоря иначе, массивы строк. Это списки выбора `TListBox` и `TComboBox`. Такие списки обычно используют в программах не для редактирования текста, а для выбора пользователем одного или нескольких строк. `TListBox` - это прямоугольный список, похожий на `TMemo`, а `TComboBox` - выпадающий список.

Чтобы познакомиться с ними, закроем текущий проект и создадим новый. Сразу же форму переименуем в fMain, сохраним проект в папку 14-02 под именем MyStrings, а модулю главной формы, как обычно, дадим имя `Main`. В свойстве `Caption` формы напишите Выбор города, свойство `BorderStyle` сделайте `bsDialog`, чтобы пользователь не мог изменять размеры окна. Свойство `Position` сделайте `poDesktopCenter`, чтобы окно появлялось по центру - такие мелочи нужно сразу же устанавливать, чтобы программа хорошо смотрелась. Размеры формы сделаем такими: `Height = 210`, `Width = 340`.

Свойства Name у компонентов в этом проекте мы менять не будем. Так будет наглядней при обращении к этим компонентам внутри кода. Хотя в реальных проектах рекомендуется давать компонентам соответствующие их назначению названия. Ну, вот вам пример. Если на форме всего одна кнопка, то её имя можно оставить по умолчанию - Button1. Итак понятно, о какой кнопке идет речь. А вот если их три, как в прошлом проекте, тогда разумней дать им понятные имена. bSave, bRead, и bClear - ясно, что речь идет о кнопках, раз первая b (button), и что это кнопки "Сохранить", "Читать" и "Очистить". Если у вас слабые знания английского - не беда, можете давать русские названия, но писать латиницей: kSohranit, kChitat и kOchistit (k - knopka). Поскольку в нашем примере все компоненты будут в единственном экземпляре, менять их свойства Name мы не будем.

Вернемся к проекту. На вкладке Standard Палитры компонентов найдите компоненты TListBox и TComboBox:

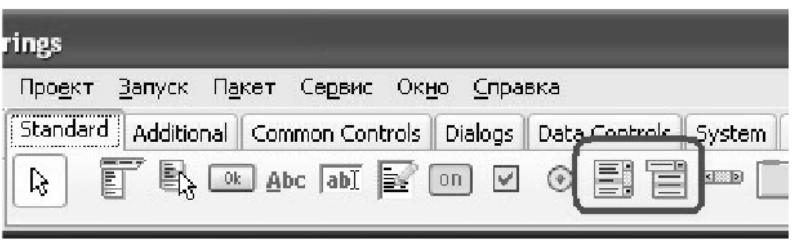


Рис. 14.3. Компоненты TListBox и TComboBox

В левом верхнем углу формы установите TListBox. Обратите внимание на его свойства в Инспекторе объектов. Мы затронем только некоторые, специфичные для TListBox.

Columns	- количество столбцов в списке. Если свойство равно нулю или единице, то список выходит в один столбец. Если больше - то столько же будет и столбцов. Если все строки не умещаются в списке, полоса прокрутки будет появляться автоматически. Обычно это свойство не меняют - строки в один столбец используются чаще всего.

ExtendedSelect	- расширенное выделение. Подразумевает, что пользователь может выделить сразу целый диапазон строк, если, удерживая <Shift>, щелкнет сперва по первой, а потом по последней строке диапазона. По умолчанию, равно True - такое выделение разрешено. В любом случае, пользователь может выделять выборочные строки, щелкая по ним поочередно и удерживая при этом <Ctrl>. Всё вышесказанное работает, только если свойство MultiSelect = True. Не будем изменять это свойство.
Items	- главное свойство компонента, массив (коллекция) строк. То же самое, что и свойство Lines у компонента TMemo, имеет тот же тип TStrings, и соответственно, обладает теми же свойствами и методами. Редактор строк также открывается при нажатии на кнопку "..." справа от свойства.
MultiSelect	- возможность выделения нескольких строк. По умолчанию, равно False - многострочное выделение запрещено. Если вы пожелаете разрешить пользователю многострочное выделение, то вам в коде придется обходить весь список, чтобы выявить выделенные строки.
Sorted	- сортировка списка. Если равно True, список сортируется по алфавиту.

Всё остальное примерно соответствует компоненту TMemo. Итак, установим у нашего ListBox1 следующие свойства:

- Left, Top = 10
- Height, Width = 150
- MultiSelect = True

Теперь в свойстве Items кнопкой "..." откройте редактор строк и внесите следующие строки (каждый город на отдельной строке):

Москва

Санкт-Петербург

Киев

Вильнюс

Будапешт

Вена

Париж

Берлин

Лондон

Мехико

Оттава

Лос-Анжелес

Сан-Франциско

Нью-Йорк

Если есть желание отсортировать список городов по алфавиту, установите `Sorted = True`.

Теперь займемся компонентом `TComboBox`. Установите его правее `ListBox1`. У него тоже не так много специфичных свойств:

`ArrowKeysTraverseList`	- при `True` (по умолчанию) разрешает перемещаться по списку кнопками со стрелками. Хотя у меня и при `False` так можно было перемещаться, может быть, в следующих версиях Lazarus будет иначе.
	- при значении `True` позволяет выбирать элементы из списка, фильтруя их. То есть, если в строке списка вы ввели "А", то выйдут только те элементы, которые начинаются на "А". Если вы ввели следующую "б", останутся только те, что начинаются на "Аб". И т.д.

AutoComplete	К сожалению, в моей версии Lazarus (v1.0.10) эта возможность недоработана. Если в свойстве установить True и попытаться сделать такую фильтрацию, то выходит run-time ошибка. Я не знаю, будет ли эта ошибка проявляться у вас, ведь новые версии Lazarus выходят достаточно часто, а у вас, скорее всего, уже более новая версия.
AutoCompleteText	- переключатели автозавершения ввода, раскрывающее свойство с пятью переключателями. При True переключатель включен, при False - выключен. Имеются следующие переключатели: • cbactEnabled - включение опции AutoComplete. • cbactEndOfLineComplete - выполняет AutoComplete, только если курсор находится в конце строки. • cbactRetainPrefixCase - сохраняет те же символы, которые ввел пользователь. • cbactSearchAscending - при True поиск подходящих строк ведется в восходящем порядке, при False - в нисходящем. • cbactSearchCaseSensitive - при True поиск чувствителен к регистру символов (различает прописные и строчные), при False - нет. Однако, увы, и это свойство пока не работает. Я включал все переключатели,

	но попытка фильтрации строк все равно приводит к ошибкам.
ItemIndex	- это очень важное свойство - индекс выделенной строки. По умолчанию равен -1, то есть, никакая строка не выделена. Индексация строк начинается с нуля, но поскольку, мы не будем вручную заполнять ComboBox1, то оставим -1.
Items	- такой же массив строк, как и у TListBox. Его также можно заполнить в процессе проектирования, нажав на "..." и вызвав Редактор строк. Однако мы не будем сейчас это делать. Заполнять список и изменять ItemIndex мы будем программно, в коде, ведь нужно освоить и эту возможность!
Sorted	- как и у TListBox, это свойство сортирует список по алфавиту.
Text	- текст, который выводится в строке компонента. При изменении ItemIndex меняется и свойство Text. По умолчанию, там написано имя компонента. Но нам это не нужно, поэтому просто очистите это свойство.

Итак, установим следующие параметры компонента ComboBox1:

Left = 177
Top = 10
Width = 150

Строки в свойстве Items мы не заполняли, другие свойства не изменяли.

Далее, нам понадобится простая кнопка TButton. Её параметры:

Left = 10
Top = 170
Width = 150
Caption = Копировать в ComboBox

И напоследок, нам понадобится `TEdit`, чтобы выводить в него выбранную в `ComboBox1` строку. Его параметры:

Left = 177
Top = 172
Width = 150
ReadOnly = True (пользователь не может редактировать эту строку)
Text = '' (то есть, очистить это свойство)

В результате у вас должна получиться такая форма:

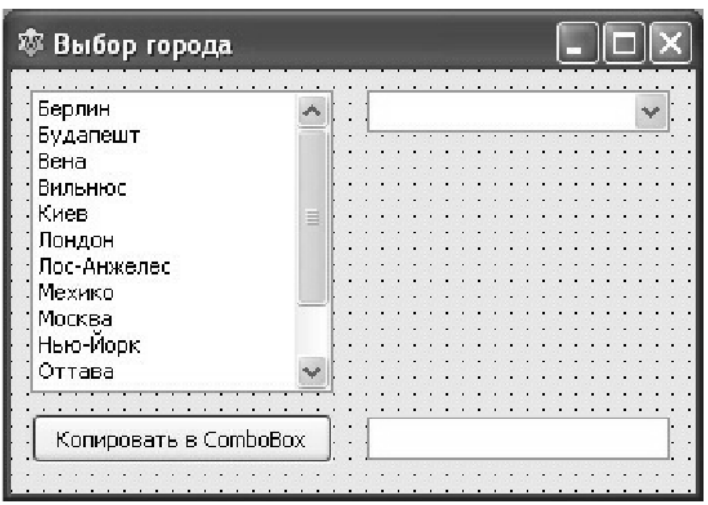


Рис. 14.4. Форма проекта

Суть программы в следующем: пользователь выбирает в ListBox1 один или несколько городов, нажимает кнопку "Копировать в ComboBox". Мы программно обойдем весь список ListBox1, от первой до последней строки, и каждую выбранную пользователем строку добавим в список `ComboBox1`. Если мы что-то в него добавили, нам придется установить `ItemIndex = 0`, то есть, выделить первую строку списка

(иначе сразу будет невидно, добавилось туда что-то или нет). Далее, если в строке `ComboBox` появился какой-то город, нам нужно будет продублировать его в строку `Edit1`.

Пример, в общем, простой, но он демонстрирует почти все возможности списков. Для реализации задуманного нам потребуется всего два события: нажатие `OnClick` на кнопку Button1, и изменение `OnChange` в компоненте `ComboBox1`. Сгенерируйте их, вот код этих событий:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  i: integer; //счетчик
begin
  //очистим ComboBox:
  ComboBox1.Clear;
  //далее в цикле обходим весь ListBox, и если строка
  //выделена - копируем ее в ComboBox:
  for i:= 0 to ListBox1.Count -1 do
    if ListBox1.Selected[i] then
      ComboBox1.Items.Add(ListBox1.Items.Strings[i]);
  //если ComboBox не пустой, выделим первый элемент, иначе
  //не выделяем ничего:
  if ComboBox1.Items.Count > 0 then begin
    ComboBox1.ItemIndex:= 0;
    //копируем в Edit1 выделенную строку из ComboBox:
    Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
  end
  else ComboBox1.ItemIndex:= -1;
end;

procedure TfMain.ComboBox1Change(Sender: TObject);
begin
  //копируем в Edit1 выделенную строку из ComboBox:
  Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
end;
```

Комментарии достаточно подробны, но все же проясним некоторые

моменты. Свойство `Items` у `TListBox` и `TComboBox`, а также свойство `Lines` у `TMemo` имеют тип `TStrings`, поэтому ведут себя похожим образом - что работает у одного компонента, будет работать и у другого. Помните, как мы очищали `Memo1`? То же самое можно сделать и с `TListBox`, и с `TComboBox`:

```
ComboBox1.Clear;
```

А вот, как мы реализовали обход всех строк в `ListBox1`:

```
//выделена - копируем ее в ComboBox:
for i:= 0 to ListBox1.Count -1 do
  if ListBox1.Selected[i] then
    ComboBox1.Items.Add(ListBox1.Items.Strings[i]);
```

Свойство `Count` компонента `ListBox1` возвращает количество строк в компоненте. В Инспекторе объектов его не видно, однако все типы `TStrings` имеют это свойство, а значит, мы сможем таким же образом узнать количество строки ив `Memo`, и в `ComboBox`. Поскольку индексация строк начинается не с единицы, а с нуля, то у последней строки списка номер индекса будет `Count - 1`.

Далее, свойство `Selected` возвращает `True`, если указанная в индексе строка выделена. В принципе, мы можем посмотреть выделена ли какая-то конкретная строка, например, девятая:

```
if ListBox1.Selected[8] then
```

но гораздо удобней, конечно, обойти весь список в цикле `for`, как это было сделано в проекте.

Далее мы смотрим - если текущая строка выделена, то мы копируем её содержимое в список `ComboBox1`:

```
ComboBox1.Items.Add(ListBox1.Items.Strings[i]);
```

Содержимое каждой конкретной строки можно получить через её свойство `Items`, которое содержит непосредственно массив строк `Strings`. Указав индекс нужной нам строки, мы получим её содержимое. Например, получить содержимое третьей строки у `ComboBox` и `Memo` можно было бы так:

```
s:= ComboBox1.Items.Strings[2];
s:= Memo1.Lines.Strings[2];
```

Метод `Add()` типа `TStrings` добавляет указанную строку в конец списка. Точно также мы могли бы добавить строку в конец списка `Memo` или `ListBox`:

```
Memo1.Lines.Add('Новая строка');
ListBox1.Items.Add('Новая строка');
```

Вот таким способом мы заполняем список `ComboBox1` теми строками, которые были выделены в `ListBox1`. Однако, поскольку пользователь мог и не выделять никаких строк, и все же нажать кнопку, мы делаем проверку:

```
  //если ComboBox не пустой, выделим первый элемент, иначе
  //не выделяем ничего:
  if ComboBox1.Items.Count > 0 then begin
    ComboBox1.ItemIndex:= 0;
    //копируем в Edit1 выделенную строку из ComboBox:
    Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
  end
  else ComboBox1.ItemIndex:= -1;
end;
```

Тут все достаточно прозрачно: если список `ComboBox1` пуст, то его свойство `Count` будет равно нулю. В этом случае, нам нужно присвоить свойству `ItemIndex` значение -1, то есть, никакая строка не выделена. Если попытаться установить значение 0, то есть, выделить первую строку, произойдет ошибка, ведь строк-то нет! Если все же

строки есть, то мы, во-первых, выделяем в списке первую строку:

```
ComboBox1.ItemIndex:= 0;
```

и, во-вторых, мы копируем её в `Edit1`:

```
Edit1.Text:= ComboBox1.Items.Strings[ComboBox1.ItemIndex];
```

Тут происходит вот что: `ComboBox1.ItemIndex` вернет текущий индекс, то есть, индекс выделенной строки. А `ComboBox1.Items.Strings[x]` вернет содержимое строки под индексом `x`, как это происходило у `ListBox1` в цикле.

И напоследок, в событии `OnChange` компонента `ComboBox1` мы дублируем код копирования выделенного текста в `Edit1` - это событие будет срабатывать, когда пользователь выберет в списке какой-то другой город.

Свойства и методы типа TSrings

В завершение рассмотрим основные свойства и методы типа `TStrings` - базового класса массива строк, широко используемого в некоторых компонентах. Это может быть как свойство `Lines` компонента `TMemo`, так и свойства `Items` компонентов `TListBox` и `TComboBox`. Контейнер `TRadioGroup`, который мы с вами изучали в лекции №7, также имеет свойство `Items` типа `TStrings`. Имеются различные свойства этого типа и у других компонентов, как стандартных, так и созданных сторонними разработчиками. К сожалению, создавать переменные такого типа нельзя, для этого служит другой класс - `TStringList`, который является наследником `TStrings`, и обладает более расширенными возможностями. Но о нем мы будем говорить уже в другой лекции. А сейчас рассмотрим основные возможности типа `TStrings`.

Таблица 14.1. Свойства и методы типа (класса) TStrings

Наименования	Описание

Свойства	
`Count`	Количество строк в списке
`Strings`	Индексированный массив строк. К отдельной строке обращаются по её индексу, например `Strings[0]` - первая строка массива.
`Text`	Содержит текст всех строк, включая символы перехода на новую строку, в виде единой строки.
Методы	
`Add`	Добавляет строку в конец списка и возвращает индекс добавленной строки. Впрочем, возвращаемое значение можно игнорировать, как это делали мы в примере.
`Append`	Добавляет строку в конец списка, но в отличие от `Add`, не возвращает никакого значения.
`AddStrings`	Добавляет в конец списка другой список строк, также имеющий тип `TStrings`.
`Clear`	Очищает список.
`Delete`	Удаляет из списка строку по её индексу. Например, удалить вторую строку из `Memo` можно так: `Memo1.Lines.Delete(1);`
`Insert`	Вставляет в список строку по указанному индексу. Весь остальной текст сдвигается вниз. Если текста нет, строка будет вставлена по индексу 0.
`LoadFromFile`	Считывает строки из указанного файла и заполняет ими компонент.
`SaveToFile`	Наоборот, считывает строки из компонента, и сохраняет их в указанный файл. Если файла нет, он будет создан, если есть - перезаписан.

Диалоги

Лекция посвящена изучению работы с диалогами. Приводятся многочисленные примеры, а также рассматриваются принципы работы с графическими файлами.

Цель лекции

Изучение стандартных диалогов и компонента-контейнера изображений `TImage`.

Диалоги

В любой большой программе есть диалоги.

Диалоги - это стандартные системные окна, предназначенные для получения информации от пользователя. В Lazarus диалоги - невизуальные (невидимые пользователю) компоненты, работа которых приводит к выводу на экран стандартных диалоговых окон операционной системы.

Информация, которую требуется получить от пользователя, может быть самой разной: какой файл открыть, какой файл создать, куда сохранить, какой выбрать шрифт, цвет, картинку… Можно, конечно, создавать все эти диалоги вручную, но гораздо удобней пользоваться стандартными диалогами - это проще, и эти диалоги используют язык установленной операционной системы. Тем более что от каждого диалога нам с вами потребуется всего пару-тройку свойств, и один метод - легче некуда! Все диалоги находятся на вкладке Dialogs Палитры компонентов.

TOpenDialog и TSaveDialog

Диалоги `TOpenDialog` и `TSaveDialog` предназначены, как нетрудно догадаться из их названия, для открытия и сохранения файлов:

Рис. 15.1. Компоненты TOpenDialog и TSaveDialog

Изучать работу диалогов мы с вами будем сразу на конкретных примерах. Чтобы не делать лишней работы, воспользуемся проектом MyNotebook из прошлой лекции, который вы должны были сохранить в папку 14-01. Если вы помните, в том проекте для хранения текста мы использовали файл MyText.txt, что не очень удобно - а вдруг пользователю захочется работать со многими файлами, а не только с одним? Вот как раз для этого нам и понадобятся два этих диалога.

Итак, откройте Lazarus. Если у вас открылся новый проект (или предыдущий), закройте его командой Файл -> Закрыть. Далее, командой Файл -> Открыть найдем и откроем проект MyNotebook (для этого нужно указать файл MyNotebook.lpi или MyNotebook.lpr). Обратите внимание: только что мы воспользовались стандартным диалогом открытия файла!

Когда проект с редактором текста откроется, установите в любое место формы по одному компоненту `TOpenDialog` и `TSaveDialog`. Эти компоненты будут невидимы для пользователя, поэтому их можно установить в любое место формы, например, прямо поверх `Memo1`. Выделите компонент `TOpenDialog`, разберемся с его свойствами. Как видите, их не очень много, да и нужны нам будут не все.

`DefaultExt`	- расширение имени файла по умолчанию. В зависимости от того, с каким типом файлов нам придется работать, такое у них будет и расширение. Если мы работаем с текстом, то лучше именам файлов давать расширение txt. Это необязательно, но так системе проще будет понять, с каким типом файлов ей придется работать. Так, если вы в Проводнике щелкните дважды по нашему прежнему файлу MyText.txt, то

	откроется стандартный Блокнот Windows, в котором будет загружен текст из этого файла.
`FileName`	- имя файла. Можно сразу же указать имя файла, с адресом и расширением, но обычно это свойство оставляют пустым. После того, как диалог сработает, и пользователь выберет файл для открытия, то в этом свойстве будет и адрес, и имя этого файла. А они нам будут нужны в методе Memo1.LoadFromFile()
`Filter`	- фильтр типов файлов. Здесь можно задать фильтрацию файлов по их типам, используя маску файлов. В маске знак "`*`" означает любое количество любых символов. Например, в фильтре можно указать маски `*.txt`, что означает "любой файл с расширением txt", и(или) *.*, что означает "любой файл с любым расширением или без расширения".
`InitialDir`	- папка (директория, каталог), используемая по умолчанию. Здесь указывается адрес папки, с которой диалог начнет свою работу. Заполнять это свойство имеет смысл лишь тогда, когда нужные файлы у вас будут храниться в каком-то одном, конкретном месте. В нашем случае мы не знаем, где пользователю захочется сохранять свои файлы, поэтому заполнять это свойство не нужно.
`Name`	- имя компонента. С этим свойством вы уже знакомы, но тут нужно сделать одно замечание. Помните, в прошлой лекции я говорил, что если компонент один, то его можно не переименовывать? С кнопками этот совет хорош - к кнопкам в коде нам не приходится обращаться по имени. А вот к диалогам придется обращаться неоднократно. Судите сами, что проще будет набрать на клавиатуре: OpenDialog1.FileName или OD.FileName?

	Вот то-то. Поэтому совет: переименовывайте диалоги, оставляя лишь заглавные буквы имени. Если вам доведется в одной форме использовать два одинаковых диалога (а мне пока такого делать не доводилось), то можно их назвать `OD1` и `OD2`.
`Title`	- заголовок окна диалога. По умолчанию, он уже содержит нужный текст: "Открыть существующий файл". Если же вам захочется установить какое то свое, нестандартное название, например "Открыть мой текстовый файл", можете вписать его здесь.

Со свойствами разобрались, остался метод. Метод этот - `Execute`.

`Execute` вызывает стандартное диалоговое окно, и дает пользователю сделать выбор. Если пользователь сделал свой выбор, то `Execute` вернет `True`. Если пользователь отказался делать выбор и закрыл окно, не указав файла, метод вернет `False`.

Настроим наш `TOpenDialog`:

- `DefaultExt = .txt`
- `Name = OD`

Теперь займемся фильтром. Выделите свойство `Filter` и щелкните по кнопке "…" справа от него. Откроется редактор фильтров, который мы заполним следующим образом:

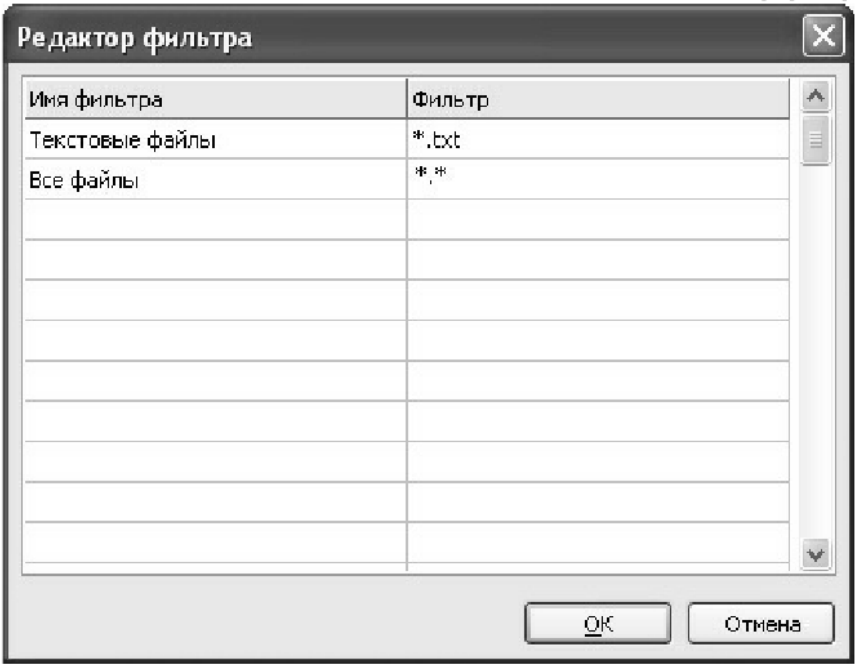


Рис. 15.2. Фильтры для TOpenDialog

Когда вы нажмете "OK", Lazarus в свойстве `Filter` сгенерирует следующий фильтр:

```
Текстовые файлы|*.txt|Все файлы|*.*
```

Эту строку можно было бы написать и вручную, но воспользоваться Редактором фильтров все же удобней.

Теперь, когда мы настроили наш диалог открытия файлов OD, займемся кодом. Если вы помните, то загрузка текста из файла у нас происходила при нажатии кнопки "Читать", то есть, `bRead`. Щелкните дважды по этой кнопке, и вы попадете на код её события `OnClick`. Приведу текст этого события:

```
procedure TfMain.bReadClick(Sender: TObject);
begin
  if FileExists('MyText.txt') then
    Memo1.Lines.LoadFromFile('MyText.txt')
```

```
  else ShowMessage('Файл MyText.txt не существует');
end;
```

Помните, мы вначале проверяли, существует ли этот файл, и затем, в зависимости от результата, предпринимали либо одно, либо другое действие. Так вот, с диалогами этого делать не нужно! Если пользователь выбрал файл, то нет смысла проверять - существует ли он. Конечно, существует, иначе пользователь не смог бы его выбрать. Чувствуете разницу? Новый код события будет таков:

```
procedure TfMain.bReadClick(Sender: TObject);
begin
  if OD.Execute then Memo1.Lines.LoadFromFile(OD.FileName);
end;
```

А теперь разница видна? Мы проверяем - если диалог сработал, то мы вызываем метод `LoadFromFile`, передавая ему в качестве параметра выбранный пользователем файл. Не забывайте, что в свойстве `FileName` диалога содержится имя выбранного файла вместе с его адресом и расширением. Если же диалог не сработал, то и делать ничего не нужно.

Теперь займемся диалогом сохранения файла `TSaveDialog`. А тут нам все уже известно - все свойства и метод `Execute` у него в точности такие, как и у `TOpenDialog`, разве что текст в свойстве `Title` другой. Именно поэтому я и дал эти диалоги парой.

Измените имя компонента на `SD`. Свойства `DefaultExt` и `Filter` настройте так же, как в OD. Все остальное остается без изменений. Теперь нам нужно вписать код сохранения файла. Делается это в событии `OnClick` кнопки "Сохранить". Новый код события такой:

```
procedure TfMain.bSaveClick(Sender: TObject);
begin
  if SD.Execute then Memo1.Lines.SaveToFile(SD.FileName);
end;
```

Код простой, и должен уже быть вам понятен без всяких комментариев. Сохраните проект и запустите его на выполнение. Посмотрите, как работает сохранение и загрузка файлов. Здесь есть один недостаток, если вы заметили: при сохранении уже открытого файла программа все равно запрашивает, куда сохранять текст. Этот недостаток мы исправим в ближайшей лабораторной работе.

TColorDialog

`TColorDialog` - диалог выбора цвета. Он очень прост. Все, что нам от него нужно, это свойство `Color` и метод `Execute`. Если диалог состоялся, то сложное свойство `Color` будет содержать выбранный пользователем цвет. Этот цвет можно присвоить свойству `Color` какого-то другого компонента. Например, свойство `Color` компонента `TMemo` отображает цвет фона теста. Я, правда, говорил, что не стоит использовать в проекте попугайские расцветки, но наша программа учебная, к тому же мы будем устанавливать только те расцветки, которые выберет пользователь. А потому, установите на форму еще одну кнопку `TButton`, свойство `Name` которой переименуйте в `bColor`, а в свойстве `Caption` напишите `Цвет`. Не забудьте снять в свойстве `Anchors` закрепление к верхнему краю, и установить закрепление к нижнему, как мы делали это с остальными кнопками.

Далее, нам потребуется установить на форму компонент `TColorDialog`:

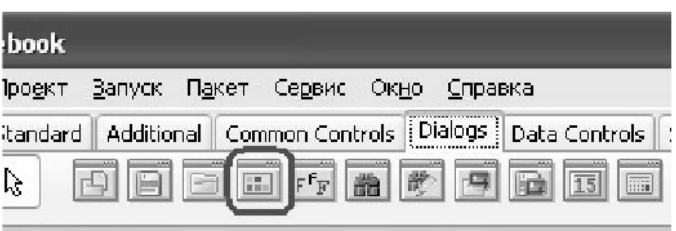


Рис. 15.3. Компонент TColorDialog

Этот компонент, как и все остальные компоненты, невизуальный, поэтому устанавливайте его, куда хотите. Свойство `Name` диалога переименуйте в `CD`. Обработчик `OnClick` для кнопки `bColor` очень

простой:

```
procedure TfMain.bColorClick(Sender: TObject);
begin
  if CD.Execute then Memo1.Color:= CD.Color;
end;
```

Если диалог состоялся, фону редактора присваиваем тот цвет, который выбрал пользователь. Сохраните проект, запустите его и попробуйте менять цвета.

TFontDialog

`TFontDialog` - диалог выбора шрифта. В этом диалоге пользователь может выбрать как сам шрифт, так и его размеры, начертание, эффекты и цвет. Не путайте цвет фона и цвет шрифта! Белый или желтый текст, например, прекрасно читается на синем или черном фоне. Нам потребуется установить на форму один такой компонент:

Рис. 15.4. Компонент TFontDialog

Свойство `Name` компонента переименуйте в `FD`. В коде нам потребуется его сложное свойство `Font`, которое будет содержать тот шрифт, что выбрал пользователь, и которое мы сможем присвоить свойству `Font` другого компонента.

Также нам потребуется кнопка с именем `bFont` и текстом `Шрифт` в свойстве `Caption`. Не забывайте про привязки кнопок к нижней границе формы. Обработчик `OnClick` для кнопки такой:

```
procedure TfMain.bFontClick(Sender: TObject);
```

```
begin
  if FD.Execute then Memo1.Font:= FD.Font;
end;
```

TCalendarDialog

`TCalendarDialog` - диалог выбора даты. В диалоге пользователь может выбрать дату, которую затем можно будет вставить прямо в текст.

Рис. 15.5. Компонент TCalendarDialog

Как и обычный календарь, этот диалог имеет свойство `Date`, которое содержит выбранную пользователем дату в формате `TDateTime`. Свойство `Name` диалога предлагаю переименовать в `DD` (Date Dialog), поскольку `CD` на форме уже есть, это диалог выбора цвета. Нам потребуется кнопка `bDate` с надписью `Дата` в `Caption`. Код обработчика кнопки следующий:

```
procedure TfMain.bDateClick(Sender: TObject);
begin
  if DD.Execute then Memo1.Lines.Add(DateToStr(DD.Date));
end;
```

Не нужно объяснять, что тут мы вставляем в конец текста дату, предварительно преобразовав ее из `TDateTime` в `String`?

TCalculatorDialog

`TCalculatorDialog` - диалог-калькулятор. Очень полезен,

особенно в бухгалтерских программах. Позволяет пользователю вывести калькулятор, сделать необходимые расчеты, и результат вставить в текст:

Рис. 15.6. Компонент TCalculatorDialog

У компонента есть свойство `Value`, которое имеет тип `Double`, то есть, выводит результат в виде вещественного числа. Переименуйте диалог в `CalcD` (так как `CD` уже есть). Для реализации диалога нам также потребуется кнопка `bCalc` с надписью `Считать` в `Caption`. Код кнопки:

```
procedure TfMain.bCalcClick(Sender: TObject);
begin
  if CalcD.Execute then Memo1.Lines.Add(FloatToStr(CalcD.Value));
end;
```

Обратите внимание, диалог считается состоявшимся, если пользователь по окончании расчетов нажал на кнопку с "галочкой":

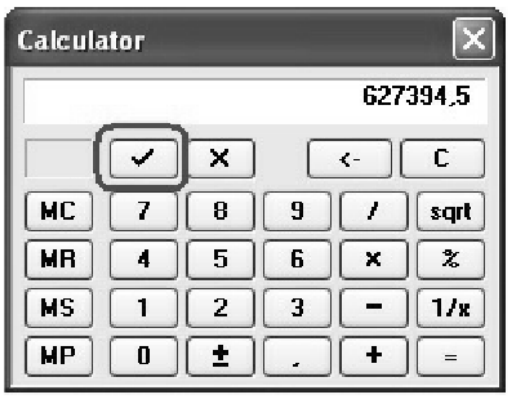


Рис. 15.7. Кнопка завершения диалога TCalculatorDialog

Только в этом случае событие `Execute` вернет `True`, и код выполнится, вставив результат вычислений в текст. Если же пользователь закроет диалог кнопкой с крестиком, клавишами <Alt+F4> или как-то иначе, то диалог не будет выполнен, и результат потеряется.

В результате всех наших действий, наш редактор должен выглядеть и работать примерно так:

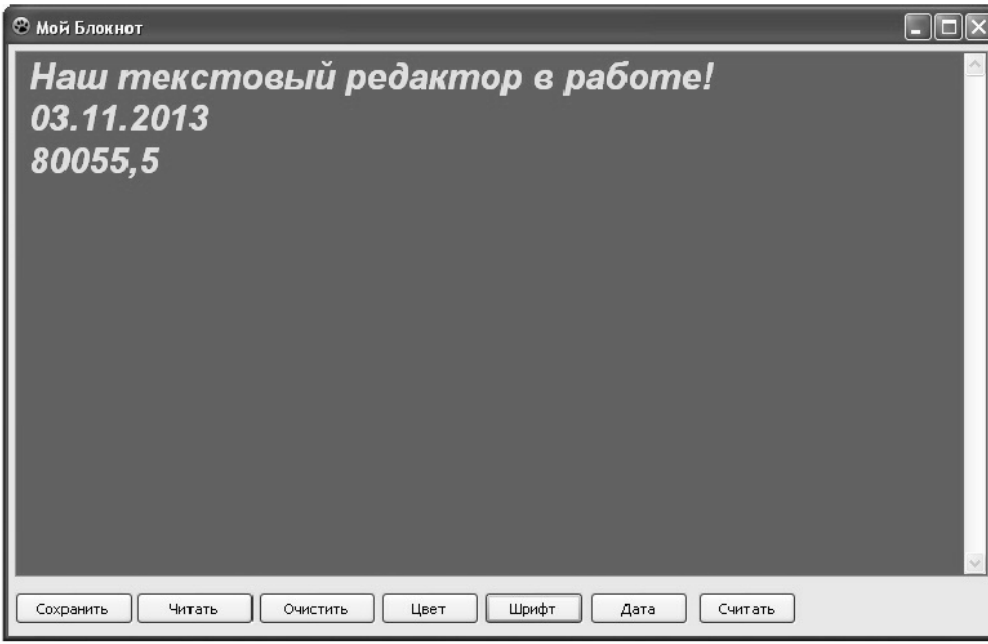


Рис. 15.8. Окончательный вид редактора

На этом данный проект можете закрыть, он нам больше не понадобится.

Компонент TImage и диалоги TOpenPictureDialog, TSavePictureDialog

Последний раздел лекции посвятим работе с графикой, у Lazarus есть и такая возможность. Чтобы не портить наши предыдущие достижения, откроем Lazarus с новым проектом. Как обычно, главную форму назовем fMain, сохраним проект в папку 15-01 под именем MyImageBrowser, а модуль главной формы назовем `Main`. Сделаем следующие настройки свойств у главной формы:

Caption = Просмотр изображений
Height = 450
Position = poDesktopCenter
Width = 700

На вкладке Additional Палитры компонентов найдите компонент `TImage`:

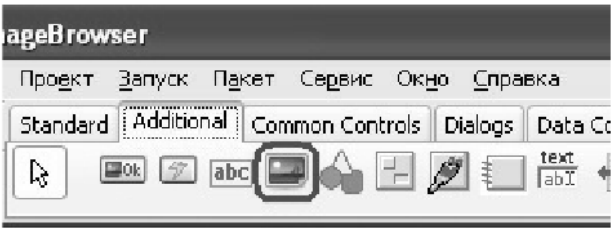


Рис. 15.9. Компонент TImage

Установите его на форму. Компонент `TImage` является контейнером для показа изображений - графических файлов. Сам компонент по умолчанию имеет такой же цвет, как форма, а его границы обозначены пунктирной линией. `TImage` позволяет загружать графические файлы как на этапе проектирования, так и в процессе работы программы. В первом случае изображение считывается, и в дальнейшем становится частью проекта, его ресурсом. Во втором случае изображение не становится частью проекта, оно занимает память с момента его открытия пользователем, и до закрытия программы, либо пока пользователь не загрузит другое изображение. Мы опробуем оба варианта, но для начала настроим положение и размеры компонента. Имя оставим по умолчанию - `Image1`. Обратим внимание на свойства компонента в Инспекторе объектов.

`AutoSize`	- автоматический размер. Если равно `True`, компонент `TImage` будет подгонять свой размер под реальный размер изображения.
`Center`	- при значении `True` выводит изображение по центру компонента.
	- основное свойство, имеет специальный тип `TPicture` и содержит само изображение. С

Picture	помощью этого свойства можно загрузить изображение из файла в компонент и во время проектирования, и во время выполнения программы. Можно также сохранить изображение в файл, хотя это имеет смысл только для графических редакторов, позволяющих создавать и изменять изображения.
Proportional	- при значении True изображение будет изменять размеры, сохраняя пропорции изображения - отношение высоты к ширине. Если значение False, то при изменении размеров изображение может быть искажено: чрезмерно вытянуто в длину или в ширину.
Stretch	- при значении True размер изображения будет подстраиваться под размер TImage. Обычно либо картинку подгоняют под размер контейнера, либо контейнер под размер картинки. В первом случае True будет у свойства AutoSize, во втором - у Stretch.
Transparent	- прозрачность. Действует только на битовые матрицы, на файлы с форматом bmp. Применяется, когда нужно спрятать фоновый цвет рисунка. Допустим, в редакторе Paint вы на белом фоне нарисовали синий круг. Сохранили в bmp-файл, который затем загрузили в TImage. Так вот, если Transparent = False, то вы получите картинку, как есть - синий круг в белом прямоугольном фоне. Если же Transparent = True, то фоновый цвет будет заменяться на цвет компонента под TImage, то есть, станет прозрачным. Фоновым считается цвет самого нижнего левого пиксела - если он белый, то этот цвет отображаться не будет.

Нужными нам методами обладает сложное свойство Picture, которое само является объектом. Подобно массивам строк, свойство Picture

имеет такие методы, как LoadFromFile и SaveToFile, хотя последний метод применяется в основном, в графических редакторах.

Установим следующие свойства компонента Image1:

Left, Top = 5
Height = 385
Width = 690
AutoSize = False
Center = True
Proportional = True
Stretch = True

Остальные свойства изменять не будем. Попробуем загрузить картинку. Выделите свойство Picture и щелкните по кнопке "..." справа от него. Откроется диалог загрузки изображений. Нажав кнопку "Загрузить", вы откроете стандартный диалог "Открыть файл изображений". Здесь вы можете выбрать желаемый файл, просматривая в правой части диалога миниатюрное изображение файла. Выбрав нужный файл, нажмите кнопку "Открыть". Выбранный файл отобразится в Диалоге загрузки изображений, после чего нажмите кнопку "OK". Выбранное изображение попадет в TImage, причем будет отображено по центру контейнера, сохранит пропорции и подгонит свой размер под размеры контейнера.

Однако нам не нужен фон для формы, нам нужна программа для просмотра различных изображений, а такая программа загружает картинки по требованию пользователя. Поэтому удалим изображение. Для этого снова выделите Picture и откройте Диалог загрузки изображений. Там нажмите на кнопку "Очистить", после чего нажмите на "OK". Диалог закроется, контейнер Image1 снова будет пуст.

Теперь нам нужен диалог открытия графических файлов - TOpenPictureDialog, который находится на вкладке Dialogs Палитры компонентов:

Рис. 15.10. Компонент TOpenPictureDialog

Подобно другим диалогам, `TOpenPictureDialog` является невизуальным, его можно установить на любое место, например, прямо посреди `Image1`. Свойств у него также немного, причем все нужные свойства уже заполнены. Откройте, например, Редактор фильтров в свойстве `Filter`, и посмотрите на то обилие графических форматов, с которыми вы можете работать! Однако имя диалога слишком длинное, так что переименуем его свойство `Name` на `OPD`.

Также внизу нам понадобится кнопка TButton, которой мы дадим имя `bOpen`, а в свойстве `Caption` напишем Загрузить изображение. Чтобы надпись поместилась на кнопке, установите её свойство `Width` равным 160, или просто растяните кнопку с помощью мыши или кнопкой со стрелкой вправо, удерживая нажатой кнопку <Shift>.

Давайте воспользуемся примером предыдущего приложения, и подкорректируем свойство `Anchors` у контейнера и кнопки: `Image1` "`привяжите`" ко всем сторонам формы, чтобы он мог менять размер вместе с формой, а кнопку "отвяжите" от верхней границы и "привяжите" к нижней.

Сгенерируйте событие `OnClick` для кнопки, его код:

```
procedure TfMain.bOpenClick(Sender: TObject);
begin
  if OPD.Execute then Image1.Picture.LoadFromFile(OPD.FileName);
end;
```

Как видите, ничего сложного, работа диалога абсолютно похожа на работу других диалогов. Сохраните проект, запустите его, попробуйте

загружать различные графические файлы и изменять размеры формы - размеры картинки также должны меняться, сохраняя пропорции:

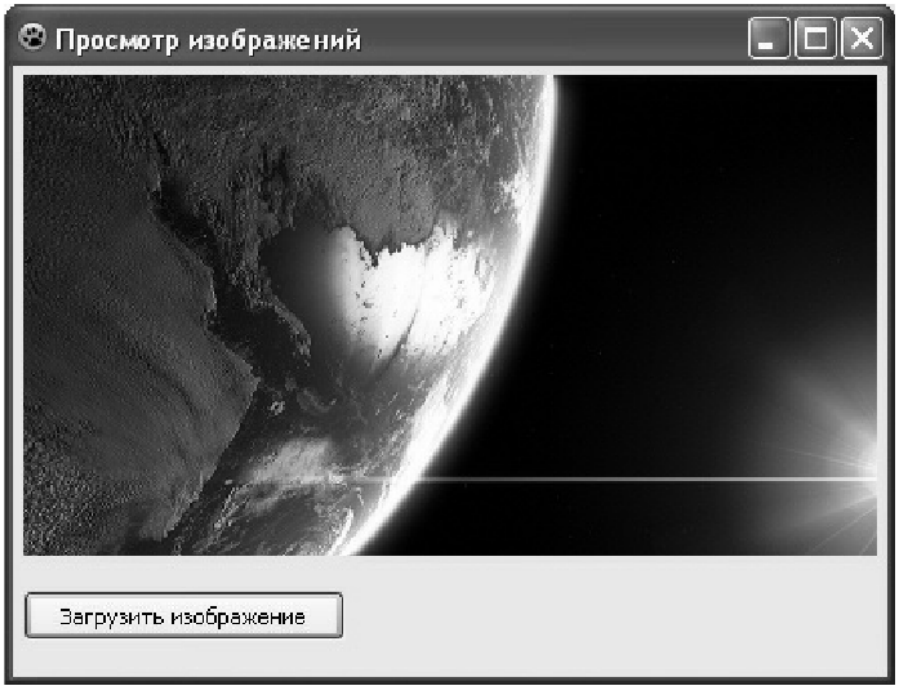


Рис. 15.11. Программа MyImageBrowser с загруженным изображением и уменьшенным размером

Диалог `TSavePictureDialog` имеет те же свойства, что и `TOpenPictureDialog` и предназначен для сохранения изображения в файл. Наша программа только просматривает файлы, а не редактирует их, поэтому здесь надобности в `TSavePictureDialog` нет. Однако на следующей лекции мы все же используем его, чтобы сохранять файл с изображением под другим именем.

Помимо пройденных диалогов вкладка Dialogs содержит и другие, более специфичные диалоги. Например, диалоги поиска и замены текста, настройки принтера и печати. Однако все диалоги работают схожим образом, так что при необходимости, вы сможете самостоятельно разобраться с работой любого из этих диалогов.

Организация меню и панелей инструментов

На лекции рассматриваются способы организации главного меню, всплывающего меню и панели инструментов, рассматривается компонент TImageList и возможность вывести изображение на пункты меню и кнопки панели.

Цель лекции

Организация главного и всплывающего меню и панели инструментов, изучение компонента `TImageList`.

Главное меню

Интерфейс программы может содержать два типа меню: главное и всплывающее. Главное меню представляет собой строку в верхней части экрана. Первой командой меню, как правило, является "Файл", последней - "Справка". Реализуется главное меню компонентом `TMainMenu`, который находится на вкладке Standard Палитры компонентов:

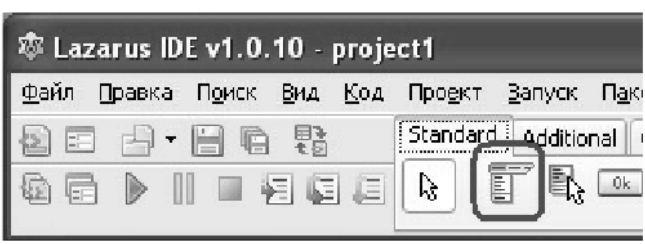


Рис. 16.1. Компонент TMainMenu

Но давайте знакомиться с этим компонентом сразу на практике. Итак, откройте Lazarus с новым проектом. Сразу же переименуйте главную форму в fMain, проект сохраните в папку 16-01 под именем ImageViewer, а модулю главной формы дайте имя `Main`. В `Caption` формы напишите `Просмотр изображений`. А саму форму сделайте побольше, хотя бы 450*700 пикселей.

Теперь установите на форму компонент `TMainMenu` - компонент

невизуальный, поэтому его можно установить в любой место, например, прямо посередине формы. Нам не придется обращаться к компоненту по имени, так что имя можно не изменять. Чтобы создать меню, нужно дважды щелкнуть по компоненту, и откроется Редактор меню. По умолчанию, в Инспекторе объектов окажется выделенным первый (и единственный) пункт меню MenuItem1:

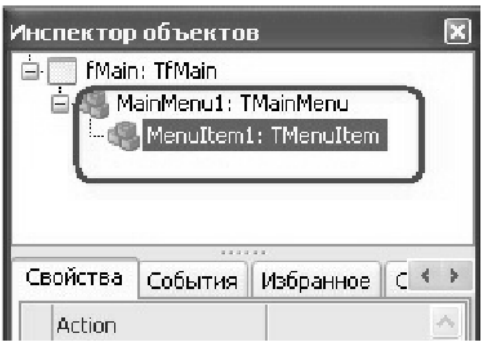


Рис. 16.2. Выделен пункт меню

Вот тут нам придется изменить некоторые настройки, так как имя `MenuItem1` нам ни о чем не говорит. Свойство `Name` переименуйте в `FileMenu`, в свойстве `Caption` напишите `Файл`. Сразу же вы увидите, как изменилось название на кнопке в Редакторе меню, и слово "Файл" появилось в верхней части окна формы. Щелкать по нему не нужно, иначе будет создан обработчик нажатия на пункт меню, а нам это не нужно - для пункта "Файл" обработчик не предусмотрен.

Далее, в Редакторе меню щелкните по кнопке "Файл" правой кнопкой мыши, и в открывшемся всплывающем меню выберите команду "Вставить новый пункт (после)". В Редакторе меню появилась вторая кнопочка с надписью "`New Item2`". Её придется переименовать, как и первую - убедитесь, что именно она выделена в Инспекторе объектов, переименуйте `Name` в `WindowMenu`, а в `Caption` напишите `Окно`. Теперь наше меню содержит два пункта: "Файл" и "Окно":

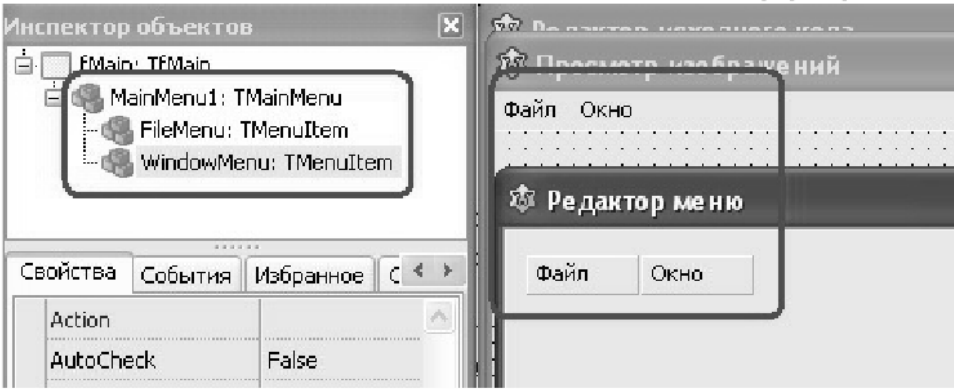


Рис. 16.3. Установка пунктов меню

Научимся создавать подпункты меню. Щелкните правой кнопкой мыши по кнопке "Файл" в Редакторе меню и выберите команду "Создать подменю". Сразу же на кнопке "Файл" появилась стрелка вниз, указывающая, что пункт меню содержит подпункты. И сам подпункт - кнопочку с надписью "New Item3", расположенную ниже "Файл". Убедимся, что в Инспекторе объектов выделен этот пункт, `Name` изменим на `FileOpen`, `Caption` - на `Открыть`.

Тут идея такова: главные пункты меню называть, как FileMenu, WindowMenu, OptionsMenu, HelpMenu и т.п. Последнее - Menu будет указывать на то, что это главный пункт. Подменю именовать иначе - вначале идет имя главного пункта, затем команда. Например, для главного пункта FileMenu мы могли бы создать такие команды, как `FileOpen`, `FileSaveAs`, `FileClose` и т.п. Это лишь рекомендация, правила вы можете придумать и свои, главное, чтобы вы не запутались в названиях главного пункта и его подпунктов.

Щелкните правой кнопкой мыши по кнопке "Открыть", выберите команду "Вставить новый пункт (после)". Полученный пункт переименуйте в FileSaveAs, а в `Caption` напишите `Сохранить как`.

Ниже аналогичным образом вставьте новый подпункт. У этого подпункта имя мы менять не будем, а в свойстве `Caption` мы впишем всего один символ "-". Таким образом в меню вставляются разделители - горизонтальные черточки между командами.

Далее вставим еще один подпункт. Назовите его FileExit, а в Caption напишите Выход. В результате у вас должно получиться нечто подобное:

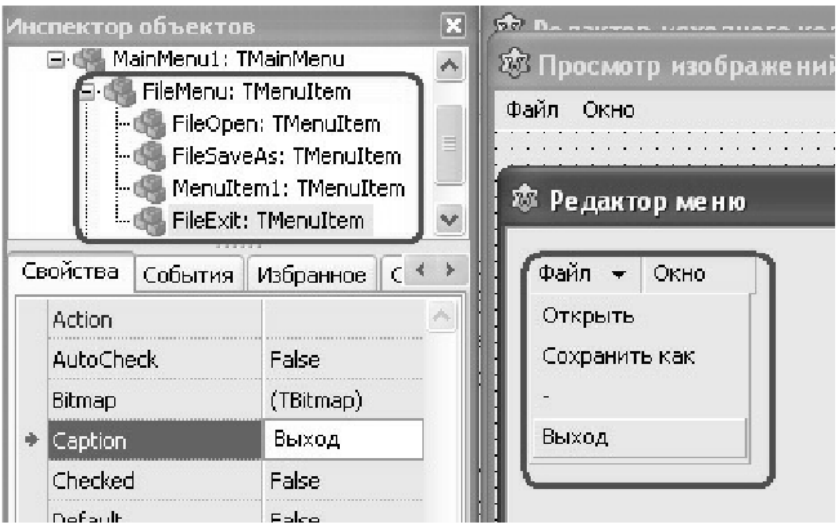


Рис. 16.4. Меню "Файл"

У нас ещё остался необработанным пункт "Окно". Выделите его и создайте следующие подпункты с парой свойств (Name - Caption): WindowNormal - Нормальное, WindowMinimize - Свернуть, WindowMaximize - Развернуть. Как только вы это сделаете, можете закрывать Редактор меню - он нам больше не нужен.

В центральную часть окна поместите компонент TImage, в который мы и будем загружать выбранные пользователем изображения. Настройте компонент следующим образом:

- Align = alClient
- Center = True
- Proportional = True
- Stretch = True

Кроме того, на форме нам понадобятся два диалога: TOpenPictureDialog, который мы переименуем в OPD, и TSavePictureDialog, который назовем SPD. Как и TMainMenu,

диалоги будут невидимы пользователю, поэтому их можно установить на любое место формы.

Теперь займемся программированием пунктов меню. Выберите на форме пункт "Файл -> Выход", и автоматически будет создан обработчик `OnClick` для этого пункта. В обработчике впишите всего одну команду:

```
procedure TfMain.FileExitClick(Sender: TObject);
begin
  Close;
end;
```

Как уже однажды говорилось, оператор `Close` в дочернем (неглавном) окне закрывает это окно, а в главном окне завершает работу программы. Создайте таким же образом обработчик для "Файл -> Открыть", с его кодом вы должны быть знакомы по предыдущему проекту:

```
procedure TfMain.FileOpenClick(Sender: TObject);
begin
  if OPD.Execute then Image1.Picture.LoadFromFile(OPD.FileName);
end;
```

Для "Файл -> Сохранить как" код будет похожий:

```
procedure TfMain.FileSaveAsClick(Sender: TObject);
begin
  if SPD.Execute then Image1.Picture.SaveToFile(SPD.FileName);
end;
```

С пунктом "Файл" разобрались, остался пункт "Окно" и три его подпункта. Здесь мы будем менять состояние окна у `fMain.WindowState`. Код для этих событий следующий:

```
procedure TfMain.WindowMaximizeClick(Sender: TObject);
begin
  fMain.WindowState:= wsMaximized;
end;
```

```
procedure TfMain.WindowMinimizeClick(Sender: TObject);
begin
  fMain.WindowState:= wsMinimized;
end;

procedure TfMain.WindowNormalClick(Sender: TObject);
begin
  fMain.WindowState:= wsNormal;
end;
```

Не перепутайте названия событий. Сохраните проект, запустите его и попробуйте в работе. Программа должна нормально загружать файлы, менять состояние окна, завершать работу. При сохранении файла указывайте не только желаемое имя, но и расширение, например newfile.jpg - компонент `TImage` не умеет угадывать формат и самостоятельно подставлять нужное расширение.

Всплывающее меню

Мы с вами то и дело пользуемся всплывающим меню. Это меню, которое появляется, когда пользователь нажимает правую кнопку мыши. Реализуется оно компонентом `TPopupMenu`, который находится рядом с `TMainMenu` на Палитре компонентов. Поскольку оно также невизуально, устанавливайте его на любое место на форме. Редактор всплывающего меню также вызывается двойным нажатием на компонент (или нажатием на "..." в свойстве `Items`). Редактируется оно примерно также. Выделите первый (и пока единственный) пункт этого меню, в свойстве `Name` напишите `pFileMenu`, в свойстве `Caption` напишите Файл. Далее, щелкните правой кнопкой мыши по кнопке этого пункта в Редакторе меню, и выберите "Создать подменю". Создайте тут два подраздела: `pFileOpen` и `pFileSaveAs` со свойствами `Caption`, соответственно, Открыть и Сохранить как.

Аналогичным образом, создайте пункт `pWindowMenu (Caption = Окно)` с подпунктами `pWindowNormal`, `pWindowMinimize` и `pWindowMaximize (Caption = Нормальное, Свернуть,`

Развернуть).

Далее, создайте главный пункт с Caption = "-" (имя не меняем, это просто разделитель), и еще один пункт Name = CloseMenu, Caption = Выход. У вас должно было получиться такое меню:

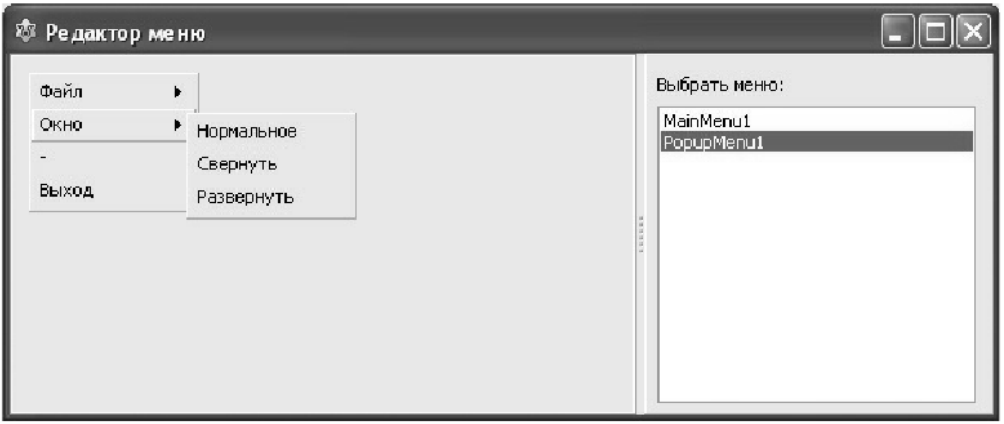


Рис. 16.5. Всплывающее меню

Меню мы сделали, но не спешите закрывать Редактор меню. Если вы думаете, что для всплывающего меню мы снова будем писать тот же самый код, то вы ошибаетесь, все намного проще. Выделите подпункт "Файл -> Открыть" - щелкайте один раз, а не дважды, иначе будет создан обработчик OnClick, а нам это не нужно. Если это всё же случилось, вам придется удалить пустой каркас процедуры этого обработчика, а также объявление этой процедуры в разделе type в верхней части модуля формы. Только не перепутайте названия процедур - все пункты всплывающего меню у нас начинаются с буквы p.

Итак, мы выделили подпункт всплывающего меню "Файл -> Открыть". В Инспекторе объектов перейдите на вкладку События, откройте список в событие OnClick и выберите из этого списка событие FileOpenClick - это событие открытия файла в главном меню, которое мы ранее запрограммировали:

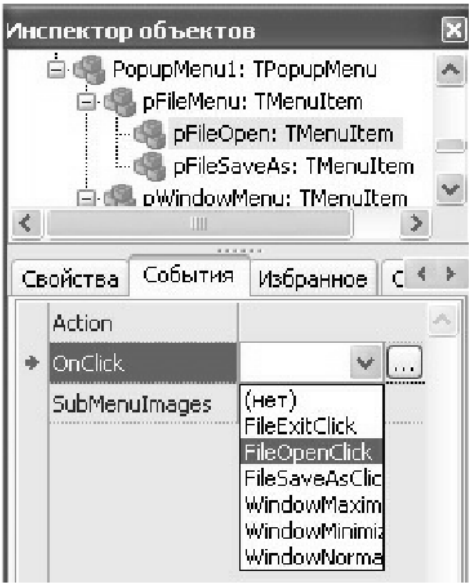


Рис. 16.6. Выбор готового события в OnClick

Таким образом, когда пользователь выберет эту команду в всплывающем меню, будет вызвано соответствующее событие главного меню, так что нам не придется дважды писать один и тот же код. Выберите соответствующие события главного меню для оставшихся команд всплывающего меню "Файл -> Сохранить как", "Окно -> Нормальное", "Окно -> Свернуть", "Окно -> Развернуть" и "Выход". После этого можно закрыть Редактор меню.

Однако и этого недостаточно, чтобы пользователь мог вызвать всплывающее меню - его еще нужно привязать к какому-нибудь компоненту. Обычно его привязывают либо к форме, либо к панели. Если на форме две панели, то для каждой можно сделать своё всплывающее меню, со своими командами. У нас всю часть формы занимает компонент `Image1`, к нему и привяжем всплывающее меню. Выделите `Image1`, перейдите на вкладку Свойства Инспектора объектов, и в свойстве `PopupMenu` выберите наш `PopupMenu1`. Теперь, если пользователь нажмет над `Image1` правую кнопку мыши, выйдет всплывающее меню. Если вы все сделали правильно, то команды всплывающего меню будут вызывать соответствующие команды главного меню, и все будет работать нормально:

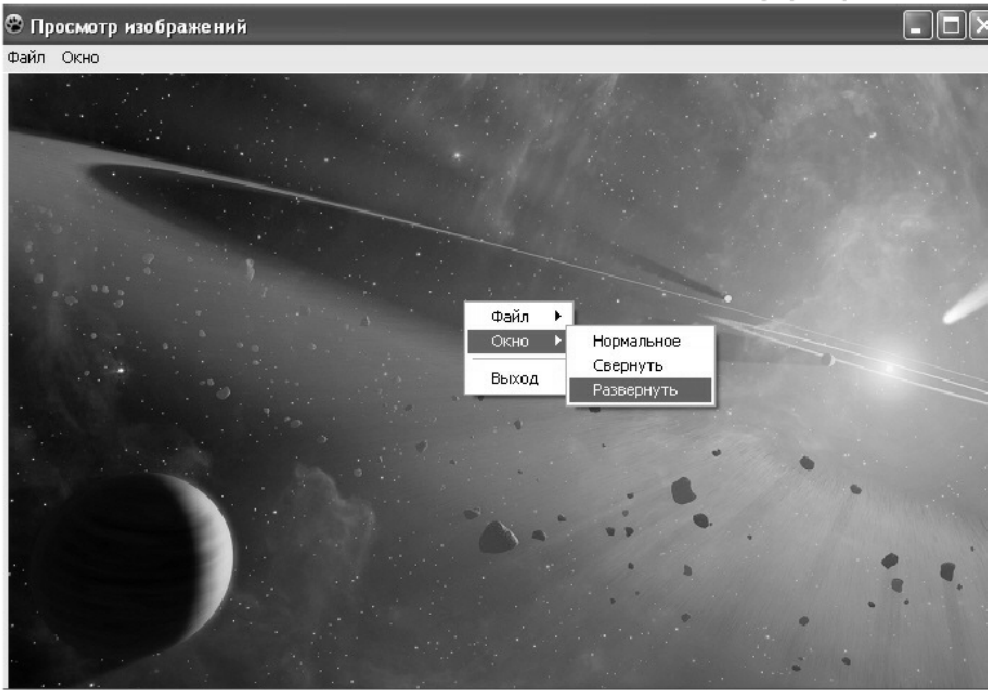


Рис. 16.7. Работа всплывающего меню в загруженном приложении

Компонент TImageList

`TImageList` - это контейнер для хранения списка изображений. Как правило, в `TImageList` хранят изображения для меню и кнопок панелей инструментов. Компонент `TImageList` находится на вкладке Common Controls Палитры компонентов:

Рис. 16.8. Компонент TImageList

Компонент невизуальный, так что его также можно установить на любое свободное место. Щелкните дважды по нему, чтобы открыть

Редактор ImageList. В Редакторе нажмите кнопку "Добавить", откроется диалоговое окно "Добавить изображения". Теперь вот что: для пунктов меню нам нужны маленькие изображения, 16*16 пикселей. По умолчанию, Lazarus устанавливается в папку C:\Lazarus. Если вы не меняли эту папку, то картинки для пунктов меню у вас будут по адресу

C:\Lazarus\images\menu

Изображений там не так, чтобы очень много, гораздо больше интересных изображений, да и целых бесплатных коллекций изображений можно найти на просторах Интернета. Однако на первое время обойдемся тем, что уже есть. Откройте указанный адрес в диалоге выбора изображений. Нам нужна картинка для пункта меню "Файл -> Открыть". Тут лучше подойдет файл menu_project_open.png. Выберите этот файл и нажмите кнопку "Открыть" - файл попадет в список под индексом 0. Как и в других списках, индексация в `TImageList` начинается с 0, индекс -1 означает, что изображения не выбрано. Снова нажмите кнопку "Добавить" в редакторе. Для "Файл -> Сохранить как" подойдет файл menu_project_saveas.png. Изображение встанет под индексом 1.

Впрочем, изображения можно перемещать по списку, менять их местами с помощью кнопок "Переместить вверх" и "Переместить вниз". Под индексом 2 установите файл menu_exit.png - это будет изображение для "Файл -> Выход". Для пунктов меню "Окно" подходящих изображений нет, поэтому мы оставим эти команды без соответствующих картинок.

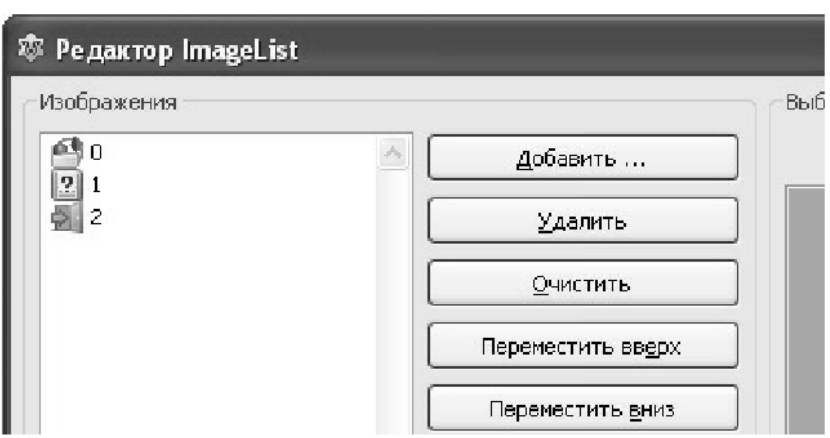


Рис. 16.9. Три установленных изображения в редакторе ImageList

Нажмите кнопку "OK", чтобы закрыть редактор. Теперь нам нужно, чтобы главное меню увидело эти изображения. Выделите компонент `MainMenu1`. В его свойстве `Images` выберите наш `ImageList1`. Таким образом, мы связали главное меню со списком изображений.

Далее, дважды щелкните по `MainMenu1`, чтобы открыть редактор меню. Выделите подпункт "Открыть" пункта "Файл". Вы видите, что его свойство `ImageIndex` равно -1, то есть, изображения не выбрано. Откройте список изображений и выберите первое изображение, с индексом 0. Для подпункта "Сохранить как" установите индекс 1. Для "Выход" - индекс 2. В результате пункты меню "Файл" станут отображаться вместе с изображениями:

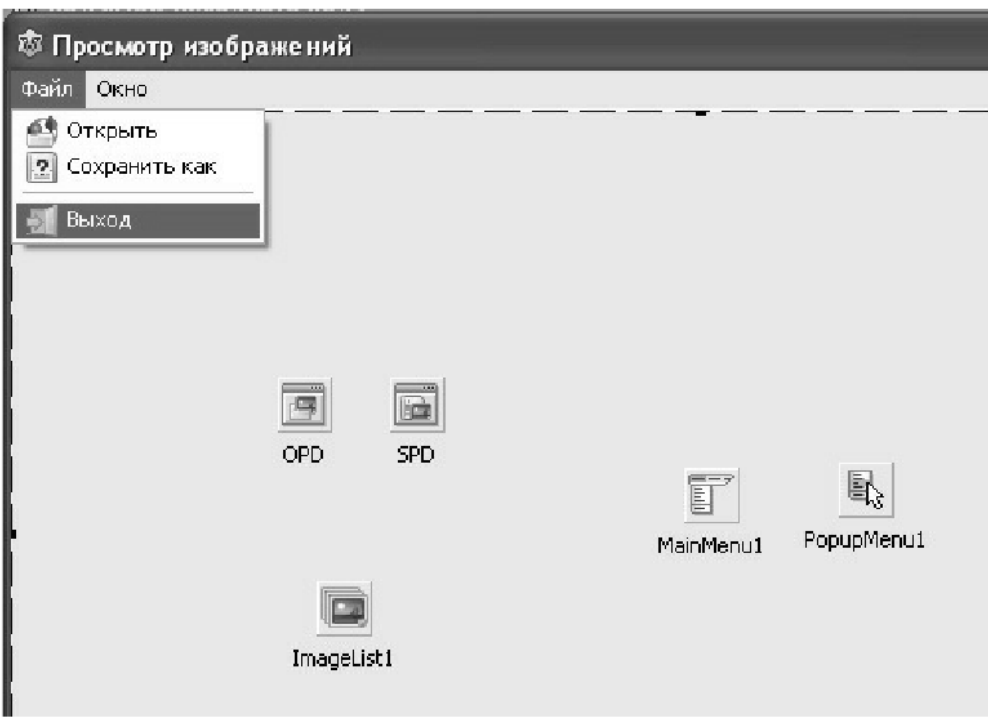


Рис. 16.10. Меню с изображениями

Большой пользы от этого нет, но проект украшает. То же самое сделайте для соответствующих пунктов всплывающего меню: сначала в его свойстве `Images` выберите наш список `ImageList1`. Затем откройте

редактор меню, выделите поочередно пункты "Открыть", "Сохранить как" и "Выход", выберите для них соответствующие `ImageIndex`. Сразу же в редакторе меню вы не заметите изменений, однако закройте его, сохраните проект и запустите программу на выполнение. Щелкните правой кнопкой мыши по окну и, пожалуйста, меню с изображениями:

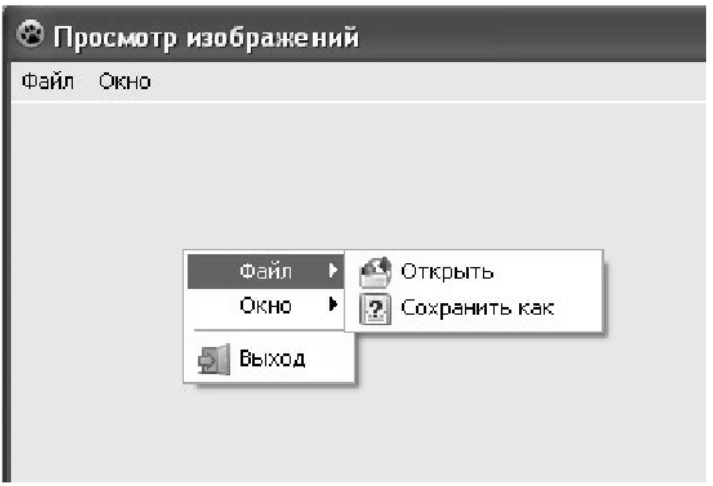


Рис. 16.11. Всплывающее меню с изображениями

Горячие клавиши

Тут самое время вспомнить об одном важном свойстве подпунктов меню, как главного, так и всплывающего - `ShortCut`.

`ShortCut` - свойство, позволяющее выбрать для команды меню сочетание клавиш, так называемые "горячие клавиши". Если какое то сочетание клавиш присвоено одной из команд, то пользователю необязательно будет лезть в меню, чтобы выполнить эту команду, достаточно будет нажать соответствующее сочетание клавиш. Так, для команды меню Файл->Открыть традиционно применяют сочетание <Ctrl+0>; для команды Файл->Сохранить - <Ctrl+S>; для команды Файл->Выход можно использовать <Ctrl+Q>. Откройте редактор главного меню, выделяйте по очереди в меню Файл подпункты "Открыть", "Сохранить как" и "Выход". Выберите для этих команд соответствующие сочетания в свойстве `ShortCut`. В результате меню приобретет такой вид:

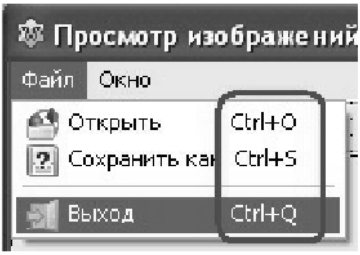


Рис. 16.12. Горячие клавиши в меню

Теперь пользователь сможет выполнять эти команды как с помощью меню, так и с помощью "горячих клавиш". Делать то же самое с пунктами всплывающего меню имеет смысл только в том случае, если его команды отличаются от команд главного меню. В нашем случае все равно будут выполняться команды главного меню, так что указывать эти же горячие клавиши во всплывающем меню не нужно.

Панель инструментов

Самый простой способ организовать Панель инструментов - установить обычную панель TPanel с настройками свойств `Align = alTop`, `AutoSize = True`, и уже на эту панель установить кнопки TSpeedButton. И панель, и кнопки мы с вами изучали в лекции №3, поэтому останавливаться на этом не будем. Изучим другой, более профессиональный способ организации панели инструментов.

Для этого нам потребуется компонент `TToolBar` с вкладки Common Controls Палитры компонентов:

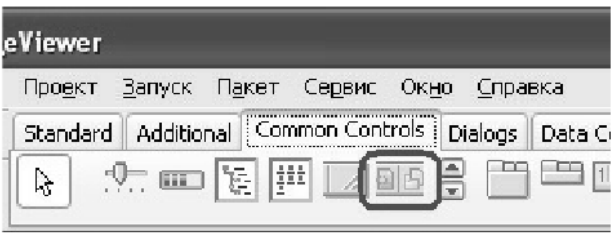


Рис. 16.13. Компонент TToolBar

Просто установите его на форму, и он сам займет нужное место - растянется по всей верхней границе окна. Обращу ваше внимание на

некоторые свойства компонента, которые могут представлять для нас интерес.

`Images`	- ссылка на список изображений для кнопок панели. Выполняет такую же функцию, как соответствующее свойство меню.
`DisabledImages`	- ссылка на список изображений, которые будут привязаны к неактивным кнопкам. Чаще всего это свойство оставляют пустым, в таком случае у неактивных кнопок (у которых `Enabled = False`) будет такое же изображение, как у активных, но только серого цвета. При желании для неактивных кнопок можно подобрать альтернативные изображения.
`HotImages`	- ссылка на список изображений для кнопок, над которыми в данный момент находится указатель мыши. Чаще всего это свойство не заполняют, и тогда изображения у таких кнопок не меняется. При желании, можно подобрать альтернативные изображения, чтобы выделять текущую кнопку.
`EdgeBorders`	- установка бордюра по сторонам панели. Имеет 4 подсвойства, соответствующие границам панели, и если в соответствующем подсвойстве установлено `True`, то данная граница будет выделена. По умолчанию, включена только верхняя граница, отделяющая панель инструментов от строки с главным меню. Если желаете, можете включить отображение всех границ.
`Flat`	- если `True`, то кнопки плоские, как в современных панелях Windows, если `False` - выпуклые, как в старых программах. По умолчанию равно `True`, так лучше и оставить.
	- если `True`, то всплывающие подсказки будут выходить, если `False`, то нет. С таким свойством нам уже доводилось встречаться. На панели

ShowHint	инструментов обычно отображаются кнопки с изображениями, но без текста. Пользователь, если только знакомится с вашей программой, поначалу может и не сообразить, какая кнопка для чего нужна. Поэтому в панели инструментов данную подсказку лучше включать (по умолчанию ShowHint = False). Затем в свойстве Hint кнопок вы пропишите нужные подсказки, и пользователь сможет их прочитать, когда подведет указатель мыши к той или иной кнопке.

Сделаем на панели ToolBar1 следующие установки:

EdgeBorders	- включаем все границы, если есть желание. А вообще это необязательно.
Images	- выберите наш ImageList1. Как видите, один и тот же список изображений мы используем и для главного меню, и для всплывающего, и для панели инструментов.

ShowHint = True

Остальные свойство можно не изменять.

Займемся созданием кнопок. Вначале сделаем кнопку "Выход из программы", затем установим разделитель, затем кнопки "Открыть файл с изображением" и "Сохранить файл как". Для команд пункта "Окно" мы кнопок делать не будем, тем более, что у нас нет для них изображений. Итак, щелкните правой кнопкой мыши по панели инструментов, и выберите команду "Новая кнопка". На панели инструментов появилась кнопка, которой Lazarus автоматически присвоил имя ToolButton1. Измените ее свойство Name на bExit, чтобы название соответствовало действию. В свойстве Hint напишите следующую подсказку:

Выход из программы

А в свойстве `ImageIndex` выберите изображение с индексом 2. Теперь перейдем на вкладку События Инспектора объектов, и в свойстве `OnClick` выберем `FileExitClick` - событие `OnClick` для пункта "Выход" главного меню. Как видите, для панели инструментов нам тоже не придется заново писать код.

Снова щелкните правой кнопкой по панели инструментов, и выберите команду "Новый разделитель". Тем самым вы вставите разделитель - вертикальную черту между предыдущей кнопкой и следующими. Ничего с этим разделителем делать больше не нужно, разве что вам покажется, что разделитель слишком (или наоборот, недостаточно) широк. В этом случае выделите разделитель и измените его свойство `Width`.

Вставьте еще одну кнопку. Назовите ее bOpen, выберите для нее изображение под индексом 0, в свойстве `Hint` напишите следующую подсказку:

```
Открыть файл с изображением
```

В событии `OnClick` этой кнопки выберите соответствующее событие из главного меню.

Вставим еще одну, последнюю кнопку. Назовем ее bSaveAs, в `ImageIndex` выберем картинку с индексом 1, в свойстве `Hint` напишем подсказку:

```
Сохранить файл как
```

Выберем для кнопки соответствующее событие `OnClick`.

В результате у вас должна получиться такая Панель инструментов:

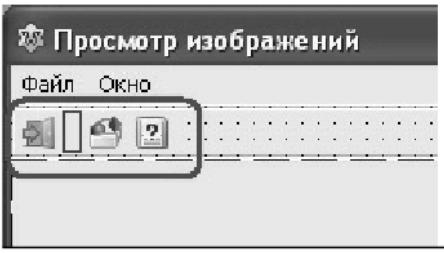


Рис. 16.14. Панель инструментов в проекте

Это все. Сохраните проект и запустите его на выполнение. Если вы все сделали правильно, теперь у вас есть правильно действующая панель управления с тремя кнопками и одним разделителем.

Модули

В этой лекции мы рассматриваем структуру модулей, их создание и подключение к проекту. На конкретном практическом примере рассматривается работа с модулями.

Цель лекции

Изучение внутренней структуры модулей, создание модуля и подключение его к проекту.

Модули

Модуль (англ. Unit) - это автономно компилируемая программная единица, которая может включать в себя такие объекты программы, как типы, константы, переменные и подпрограммы (процедуры и функции).

Каждый раз, когда мы создаем какое либо окно, для него автоматически создаются два файла: файл описаний *.lfm и модуль *.pas. В файле описаний в простом текстовом виде содержится описание формы - какие компоненты на ней находятся, как они настроены, какие параметры содержатся у свойств каждого компонента. Редактировать этот файл вручную крайне не рекомендуется, лучше всё сделать в Lazarus, в Инспекторе объектов и Редакторе форм, так вы гарантированно избежите ошибок в описании формы.

В файле модуля находится исходный код - то, что делает программа, когда пользователь взаимодействует с различными компонентами или с самой формой. Весь тот код, все процедуры и функции, которые мы вводили, находятся в модуле того или иного окна. Однако значение модулей этим не ограничивается. Модули необязательно должны быть связаны с окном, они могут существовать в проекте и сами по себе! Чаще всего это делают с двумя целями:

- Разработка приложений несколькими программистами (каждый пишет свой код, который потом собирается и компилируется в единый проект).
- Программист может собирать собственную библиотеку часто

используемых функций и процедур, помещая их в независимые модули, которые затем он может подключить к любому своему проекту или даже поделиться ими с другими программистами.

Как вы понимаете, больше всего нас интересует вторая цель, с этим рано или поздно сталкивается любой практикующий программист. Кроме того, на просторах Интернета вы можете найти великое множество самых разнообразных инструментов, выполненных в виде модулей, которые авторы выкладывают в общий доступ, их можно скачать и использовать в своих программах.

Как-то раз один мой ученик спросил, не является ли дурным тоном использовать чужой код. Давным-давно, на заре программирования это было и впрямь не очень принято. Однако теперь все изменилось - программы стали гораздо больше, функциональней, простая программа содержит очень много кода. Например, мы с вами не раз применяли функцию `Length()`, которая возвращает длину строки в символах или длину массива в элементах. В принципе, мы могли бы поднапрячься, и самостоятельно реализовать эту функцию стандартными средствами Паскаля. Однако это было бы неоправданной тратой времени, ведь эта функция уже реализована и включена в библиотеку Lazarus! Я ответил тому ученику, что дурным тоном будет изобретать велосипед, то есть, впустую тратить время на разработку того, что давным-давно реализовано. Конечно, если вы будете использовать чужой код без ведома и разрешения автора, то это будет плохо. Однако если автор сам выложил свой код, то почему бы им не воспользоваться? Разумеется, тут остается риск того, что автор - не очень хороший программист, и его код может "глючить" - работать с ошибками. В этом случае могу дать такой совет: ищите исходники на приличных, больших сайтах, которые не заброшены и часто обновляются. В следующей лекции, на лабораторной работе, мы используем один такой чужой модуль.

Сейчас же нас больше интересует, как создать модуль самостоятельно, как поместить в него процедуры и функции, которые могут пригодиться нам в дальнейшем, в различных проектах. Мы реализуем модуль с "защитой от дураков" - с проверкой правильности вводимого пользователем числа. Причем числа могут быть целые беззнаковые (от 0 и выше), целые со знаком (могут быть отрицательными) и вещественные, которые все со знаком.

Структура модулей

Модуль в Паскале имеет следующую структуру:

```
Unit <имя модуля>;
interface
<интерфейсная часть> //открытая часть
implementation
<исполняемая часть (реализация)> //закрытая часть
initialization
<необязательная инициирующая часть>
finalization
<необязательная завершающая часть>
end.
```

Имя модуля

Имя модуля должно точно соответствовать имени файла этого модуля. То есть, если вы создаете модуль

unit MyUnit;

то и сохранять его нужно обязательно в файл MyUnit.pas. При этом старайтесь выбирать для модулей понятные имена, не соответствующие стандартным модулям. Имена стандартных подключаемых модулей вы можете видеть в разделе `uses` в Редакторе кода.

Совет: если при редактировании кода и удерживая нажатой клавишу <Ctrl> вы подведете указатель мыши к имени какого либо подключаемого модуля, его имя превратится в гиперссылку, щелкнув по которой вы откроете этот модуль. Можете посмотреть, как реализуются профессиональные модули, заглянуть "во внутренности" процедур и функций.

Интерфейсная часть

Интерфейсная часть - это открытая часть, начинается со служебного

слова `interface`. Здесь можно объявлять глобальные типы, константы, переменные, функции и процедуры, которые будут доступны для всех программ и модулей, к которым вы подсоедините данный модуль. Здесь же при желании вы можете указать служебное слово uses, после которого перечислить те модули, которые хотите подключить к этому модулю. Пример (выполнять не нужно):

```
Unit MyUnit;
interface
uses windows;
const
  MyPI = 3.1415;
var
  MyGlobalVariable: Integer;
function MyGlobalFunc(a, b: string): boolean;
procedure MyClobalProc;
```

Здесь мы создаем модуль с именем `MyUnit`, который нужно сохранить в файл с именем MyUnit.pas. Далее, мы объявляем интерфейсную часть. Тут с помощью служебного слова `uses` (англ. использовать) мы указываем, что к этому модулю нужно еще подключить модуль `windows`, который входит в стандартную поставку Lazarus.

Далее, мы объявляем константу `MyPI`, которой сразу присваиваем значение, и переменную `MyGlobalVariable`. И константой, и переменной можно будет пользоваться извне - там, где будет подключаться этот модуль.

Далее мы объявляем функцию и процедуру. Обратите внимание: только объявляем! Само описание (код) этих функции и процедуры будет в разделе исполняемой части (или разделе реализации). В дальнейшем этими процедурой и функцией мы также сможем пользоваться извне.

Исполняемая часть

Исполняемая часть (раздел реализации) начинается со служебного слова `implementation`. Это закрытая часть. Тут описываются те процедуры и функции, которые были объявлены в интерфейсной части,

причем заголовки подпрограмм должны в точности соответствовать предыдущему объявлению. Также тут можно объявлять типы, константы и переменные, а также и подпрограммы, которые будут видны только в этом модуле и не смогут использоваться извне. Пример:

```
implementation
uses MyOldUnit;
var
  MyLocalVariable: Real;

function MyGlobalFunc(a, b: string): boolean;
begin
  …
end;

procedure MyClobalProc;
begin
  …
end;
```

Как видно из примера, в исполняемой части также можно подключать внешние модули. Здесь же мы объявили переменную `MyLocalVariable`, она является глобальной внутри этого модуля, то есть, её можно использовать в функции и процедуре, однако извне она не будет видна. Далее у нас идет реализация (описание) объявленных ранее подпрограмм.

Инициирующая и завершающая части

Эти части модуля необязательны, их можно не указывать, как правило, их и не используют.

Инициирующая часть начинается служебным словом `initialization` и содержит код, который будет выполнен только один раз - до передачи управления в основную программу. Тут можно сделать какую то подготовительную работу - открыть нужный файл, присвоить значения каким то переменным и т.п.

Завершающая часть начинается служебным словом `finalization` и содержит код, который будет выполнен тоже один раз - при завершении работы основной программы. Здесь обычно освобождают занятые программой ресурсы, закрывают открытые файлы, сохраняют результаты работы и т.п.

Конец модуля

Модуль завершается служебным словом `end` с точкой в конце. Это единственный случай, когда после `end` ставится не точка с запятой, а точка. Весь дальнейший текст, если он есть, компилятором будет игнорирован. Кроме того, если дальнейший текст не соответствует синтаксису, компилятор выведет сообщение об ошибке, и не даст скомпилировать программу. Так что, если вам нужно указать какие то комментарии, например, с советами как пользоваться модулем, рекомендую делать их между фигурными скобками в самом начале модуля, даже до служебного слова `Unit`. Так обычно и делают. А вот после `end` с точкой уже ничего писать не нужно.

Создание модуля

Ну вот, с теорией покончили, приступим к практике. Как уже говорилось выше, сейчас мы с вами реализуем "защиту от дураков". Допустим, в программе у нас имеются компоненты `TEdit`, в которые пользователь должен вписать какое-то число - целое без знака, целое со знаком или вещественное, которое еще называют действительным числом. Должен то он должен, но кто может предположить, что он туда на самом деле введет? Может быть, слово, скобки, знак процента - пользователь непредсказуем! Как сказал один мудрый программист, "Если есть хоть малейшая возможность ввести в программу неправильные данные, пользователь эту возможность непременно найдет". Наша с вами задача - предусмотреть всё и лишить его этой возможности. Ведь нам в дальнейшем придется преобразовывать это число из текстового формата в настоящие числа, используя `StrToInt()`, `StrToFloat()` и т.п. функции. Если же пользователь введет ошибочные данные, то программа вызовет ошибку.

Поскольку делать такие проверки программисту приходится довольно

часто, имеет смысл реализовать их в виде отдельного модуля, который затем можно будет подключить к любой программе. И этот модуль будет началом нашей библиотеки модулей, которую, как я надеюсь, вы будете постоянно пополнять как своим, так и чужим кодом.

Модуль - это текстовый файл с расширением *.pas, который можно создать в любом текстовом редакторе. Однако будет удобней воспользоваться Редактором кода Lazarus - он подсвечивает синтаксис кода, выводит подсказки, при нажатии <Ctrl+J> выводит список с часто используемым командами. Так что для начала откроем Lazars. Если там загрузился предыдущий или новый проект, то командой меню Файл->Закрыть закройте его. Далее, выберите команду Файл->Создать модуль.

Сразу же появится заготовка модуля со следующим кодом:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
implementation
end.
```

Имя модуля по умолчанию `Unit1` - оно ни о чем нам не говорит, заменим его на `numbers`. Далее, идут директивы компилятору, затем интерфейсная часть с двумя подключенными модулями - их отключать не будем, поскольку там могут содержаться описания глобальных переменных, констант, подпрограмм, которые могут нам понадобиться. В частности, для проверки правильности ввода разделителя в вещественное число, мы используем системную переменную `DecimalSeparator`, которая описана в одном из этих модулей, и которую без него не получится использовать.

Целиком код модуля следующий (можете просто скопировать его отсюда):

```
//------ Начало модуля ---------
{В модуле описаны две функции, которые проверяют правильность ввс
```

```
пользователем символа. Функция TrueIntKeys проверяет целые числа и
параметры: key - введенный пользователем символ; str - строка в TEdit
sign - если True, то число со знаком, если False - без знака. Если пользо-
ватель ввел правильный символ, функция его не изменяет, иначе возвр
символ #0 - то есть, ничего. Функция TrueFloatKeys аналогичным обра
делает проверку вещественных чисел. Пример использования:
Key:= TrueIntKeys(Key, Edit1.Text, True); //целое со знаком
Key:= TrueIntKeys(Key, Edit1.Text, False); //целое без знака
Key:= TrueFloatKeys(Key, Edit1.Text); //вещественное
}

unit Numbers;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

//проверка правильности символа в целом числе:
function TrueIntKeys(key:char; str:string; sign:boolean):char;
//проверка правильности символа в вещественном числе:
function TrueFloatKeys(key:char; str:string):char;

implementation

{проверка правильности символа в целом числе}
function TrueIntKeys(key:char; str:string; sign:boolean):char;
begin
  //сначала укажем, что возвращается тот же символ, что ввел пользоват
  Result:= key;
  //далее делаем проверку на правильность символа. если символ не пра
  //мы его запретим:
  case key of
    //все числа разрешаем:
    '0'..'9': ;
    //backspace разрешаем:
    #8: ;
```

```
    //если знаковое, то разрешаем минус при условии, что минус - первы
    //символ в строке:
    '-': if sign and (Length(str) = 0) then Result:= key
        else Result:= #0; //если беззнаковое, или минус не первый, запреща
  //все остальные символы запрещаем:
  else Result:= #0;
  end; //case
end;

{проверка правильности символа в вещественном числе}
function TrueFloatKeys(key:char; str:string):char;
begin
  //сначала укажем, что возвращается тот же символ, что ввел пользоват
  Result:= key;
  //далее делаем проверку на правильность символа. если символ не пра
  //мы его запретим:
  case key of
    //все числа разрешаем:
    '0'..'9': ;
    //backspace разрешаем:
    #8: ;
    //если разделителя еще нет - выводим правильный разделитель,
    //иначе ничего не выводим
    ',' , '.' : if Pos(DecimalSeparator, str)= 0 then
                  Result := DecimalSeparator
                else Result := #0;
    //разрешаем минус при условии, что минус - первый символ в строке
    '-': if Length(str) = 0 then Result:= key
        else Result:= #0; //если минус не первый, запрещаем
  //все остальные символы запрещаем:
  else Result:= #0;
  end; //case
end;

end.
//------- Конец модуля ---------
```

Листинг .

```
//------ Начало модуля ---------
//------- Конец модуля ---------
```

копировать в модуль не обязательно, они только обозначают границы листинга этого кода.

Первым делом мы изменили имя модуля с `Unit1` на `Numbers`. Теперь этот модуль нужно сохранить в какую-нибудь временную папку, в файл Numbers.pas. Вместе с модулем будет сохраняться и проект, не обращайте на это внимание - проект нам не нужен, его потом можно будет удалить, оставив лишь файл с модулем.

Выберите команду Файл->Сохранить. Сначала будет запрос на сохранение проекта, его имя можно не менять, а папку выбрать временную, например, C:\Temp. Затем будет предложено сохранить модуль, причем Lazarus сам подставит имя `numbers`. Просто нажмите "Сохранить". Теперь откройте ту папку Проводником Windows или файловым менеджером, и увидите файл numbers.pas - это и есть наш модуль, остальные файлы в этой папке нам не нужны.

Прежде, чем двигаться дальше, давайте разберемся с кодом модуля.

В первоначальной заготовке модуля мы только поменяли его имя. Затем вернулись в самое начало, и в виде многострочного комментария описали работу с этим модулем. Делать это весьма полезно, так как модуль прослужит вам не год и не два - в дальнейшем вы можете просто забыть, как он был реализован. Чтобы вновь не изучать его исходный код, проще описать реализуемые тут инструменты и рекомендации по их использованию. А если вы планируете выложить свой модуль в Интернет, имеет смысл также добавить в комментарий строки

```
Автор: ФИО автора
Дата: Дата реализации
```

Затем в интерфейсной части мы объявили две функции:

```
function TrueIntKeys(key:char; str:string; sign:boolean):char;
function TrueFloatKeys(key:char; str:string):char;
```

Исходим вот из чего. У компонента `TEdit` имеется событие `OnKeyPress`, которое возникает ДО того, как введенный пользователем символ попадет в строку `Edit1.Text`. У события имеется переменная `Key`, которая и содержит этот символ. Как правило, в этом событии проверяют корректность введенного пользователем символа. Если символ правильный, то переменную оставляют без изменений. Если неправильный, его можно исправить - в коде мы сравниваем разделитель не с точкой или запятой, а с переменной `DecimalSeparator`, что гарантирует правильный разделитель, невзирая на то, точку нажал пользователь, или запятую. Ну а если уж символ совсем некорректный (например, буква в числе), то переменной `Key` в событии `OnKeyPress` можно присвоить `#0` - нулевой символ, то есть, ничто. В этом случае, пользователь может нажимать на неправильные клавиши хоть неделю, в `TEdit` эти символы не попадут. И, конечно же, в проверке обязательно надо учесть, что пользователю нужно предоставить возможность редактировать неправильно введенные символы. Так, клавиша <Backspace>, которая затирает символ слева от курсора, имеет код `#8` - эту клавишу нужно разрешить.

Реализацию самих функций мы сделали в разделе исполняемой части. Вначале переменной `Result` мы присвоили то значение, что уже содержится в параметре `key`. Ведь если символ правильный, то проще ничего не делать, чем каждый раз писать

```
Result:= key;
```

Затем для целого числа мы устраиваем следующую проверку:

```
case key of
  //все числа разрешаем:
  '0'..'9': ;
  //backspace разрешаем:
  #8: ;
```

```
    //если знаковое, то разрешаем минус при условии, что минус - первы
    //символ в строке:
    '-': if sign and (Length(str) = 0) then Result:= key
        else Result:= #0; //если беззнаковое, или минус не первый, запреща
  //все остальные символы запрещаем:
  else Result:= #0;
  end; //case
```

Комментарии достаточно подробны, но все же разберем код "по косточкам". Переключатель `case` выполняет один из операторов, в зависимости от значения параметра `key`. Строка условий

```
'0'..'9': ;
```

означает, что если в `key` содержится символ любой цифры, от 0 до 9, то ничего делать не нужно, так как `Result` уже содержит символ, введенный пользователем, и этот символ корректен. Если бы мы, к примеру, хотели бы сделать проверку на латинский строчный символ, то указали бы диапазон `'a'..'z'`.

Далее, у нас проверка

```
#8: ;
```

Как мы уже выяснили, код `#8` имеет клавиша <Backspace>, которую мы тоже позволяем, поэтому ничего не делаем.

А вот дальше интересный код. Число может быть знаковым (параметр `sign=True`) - в этом случае нам нужно разрешить символ минуса. Однако все равно, мы должны проверить, что если это минус - он должен быть первым символом в строке, и что он единственный минус в строке. Согласитесь,

```
12--34
```

это неправильное целое число. Если же число беззнаковое, то мы и

вовсе запрещаем этот минус. Всё это реализуется следующей проверкой:

```
'-': if sign and (Length(str) = 0) then Result:= key
    else Result:= #0; //если беззнаковое, или минус не первый, запреща
```

Этот оператор выполняется, если в параметре `key` у нас символ минуса. Условный оператор `if` выполняется, только если в параметре `sign` у нас `True`, и если строка пуста `(Length(str)=0)`. Согласитесь, если в строке уже что-то есть, то минус не будет первым символом! Если же число беззнаковое или вводимый символ - не первый, выполняется часть после `else` - функция возвращает код `#0`, то есть, результат будет такой, будто пользователь ничего и не нажимал.

И в заключение этой проверки мы также возвращаем код `#0`, сюда подпадут все остальные символы. В результате, пользователь не сможет ввести в `TEdit` никакого неправильного символа.

Проверка правильности вещественного числа в функции `TrueFloatKeys` такая же, как и для целого числа, с двумя исключениями: во-первых, все вещественные числа знаковые (могут быть отрицательными), поэтому параметр `sign` тут не нужен, и проверки на знаковость нет. Во-вторых, вещественное число может содержать разделитель целой и дробной части, а в строке не может быть двух разделителей. Это реализуется следующей проверкой:

```
',' , '.' : if Pos(DecimalSeparator, str)= 0 then
              Result := DecimalSeparator
            else Result := #0;
```

Этот оператор выполняется, если в `key` точка или запятая. Функция

```
Pos(DecimalSeparator, str)
```

вернет ноль только в одном случае: если правильный разделитель, который хранится в системной переменной `DecimalSeparator`, не обнаружен в строке `str`. В этом случае мы возвращаем правильный

разделитель, который может быть как точкой, так и запятой, в зависимости от языка операционной системы. Как видите, это универсальная проверка, не зависящая от языка, которая к тому же избавляет пользователя от необходимости следить за тем, точкой или запятой он отделяет целую и десятичную часть числа. Пользователю будет удобно пользоваться правой частью клавиатуры для ввода чисел. Если же в строке правильный разделитель уже есть, то мы возвращаем нулевой код, запрещая второй такой же разделитель.

Теперь о том, как использовать данный модуль в других проектах. Это можно сделать двумя способами. Первый заключается в том, что модуль нужно не только сохранить, но и скомпилировать. В этом случае, в папке с проектом появится папка lib\i386-win32 (для Windows), в которой будут два файла модуля - numbers.o и numbers.ppu. Это - наш модуль в скомпилированном (исполняемом) виде. Теперь нужно найти папку, в которой в других папках хранятся библиотеки поставляемых с Lazarus различных модулей. Для Windows по умолчанию это папка

C:\lazarus\fpc\2.6.2\units\i386-win32

Адрес может быть и другой, если у вас иная версия FPC. Здесь мы можем создать свою папку, например, MyModules или MyUnits, куда и будем копировать наши модули в виде пары файлов, а затем подключать их к программам, включая имя модуля в раздел `uses`. Однако этот вариант не очень хорош - файлы-то двоичные, прочитать их простым текстовым редактором мы не сможем. Если таких файлов много, как узнать, где нужные нам инструменты, как дополнять эти модули новыми функциями и процедурами?

Я предлагаю другой способ. Создайте где-нибудь на диске, где хранятся ваши важные документы, папку с любым именем, куда вы будете собирать ваши (и чужие) модули, и скопируйте туда файл numbers.pas - наш модуль в исходном, не скомпилированном виде. А затем, если в каком то проекте нам нужно будет делать проверку на правильность числа, мы просто будем копировать numbers.pas в папку с этим проектом! Таким образом, он всегда будет доступен в текстовом варианте, по комментариям перед модулем мы легко сможем найти именно то, что нам нужно.

Итак, вы создали папку, которую будете использовать, как библиотеку

ваших модулей, и скопировали туда numbers.pas? Отлично, теперь займемся самим проектом.

Пример включения модуля в проект

Откройте Lazarus с новым проектом. Как обычно, форму назовите fMain, в `Caption` пропишите `Проверка чисел`, сохраните проект в папку 17-01 под именем `CheckNum`, модулю главной формы, как обычно, дайте имя `Main`.

Теперь откройте ваш любимый файловый менеджер, с его помощью найдите наш модуль numbers.pas и скопируйте его в папку с проектом 17-01. Далее, в Редакторе кода найдите раздел `uses` в интерфейсной части, поставьте запятую после последнего указанного включаемого модуля, добавьте туда имя

```
numbers:
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, numbers;
```

Всё, наш модуль мы включили в проект, теперь можем реализовать проверку. Проект будет самый простой, с минимальным набором компонентов - только чтобы проверить работу нашего модуля. Установите на форму три компонента `TLabel`, их имена менять не нужно, а в `Caption` компонентов напишите соответственно, "`Целое со знаком:`", "`Целое без знака:`" и "`Вещественное:`". Правее меток установите три компонента `TEdit`, их имена также оставьте без изменений, в каждом `TEdit` очистите свойство `Text`. Ну и, наконец, в нижней части формы установите кнопку `TBitBtn` с вкладки Additional, в свойстве `Kind` которой выберите значение `bkClose`. Подравняйте компоненты так, чтобы у вас получилась примерно такая форма:

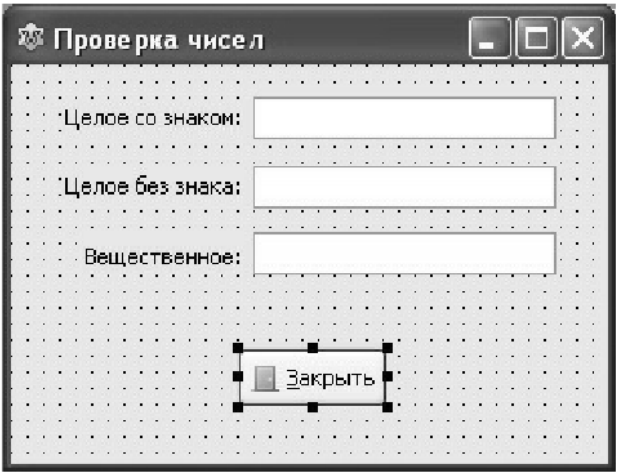


Рис. 17.1. Форма проверочного проекта

Теперь выделите `Edit1`, который предназначен для ввода целого числа со знаком. В Инспекторе объектов перейдите на вкладку События и сгенерируйте для компонента событие `OnKeyPress`. Его код:

```
procedure TfMain.Edit1KeyPress(Sender: TObject; var Key: char);
begin
  //проверка правильности вводимого символа в целое число со знаком:
  Key:= TrueIntKeys(key, Edit1.Text, true);
end;
```

Для `Edit2` код события `OnKeyPress` будет таким:

```
procedure TfMain.Edit2KeyPress(Sender: TObject; var Key: char);
begin
  //проверка правильности вводимого символа в целое число без знака:
  Key:= TrueIntKeys(key, Edit2.Text, false);
end;
```

И, наконец, для `Edit3` код события `OnKeyPress` будет таким:

```
procedure TfMain.Edit3KeyPress(Sender: TObject; var Key: char);
begin
```

```
  //проверка правильности вводимого символа в вещественное число:
  Key:= TrueFloatKeys(key, Edit3.Text);
end;
```

Всё, сохраните проект, скомпилируйте и запустите его на выполнение. Теперь, как бы вы ни старались, ввести некорректные значения в строки ввода у вас не получится:

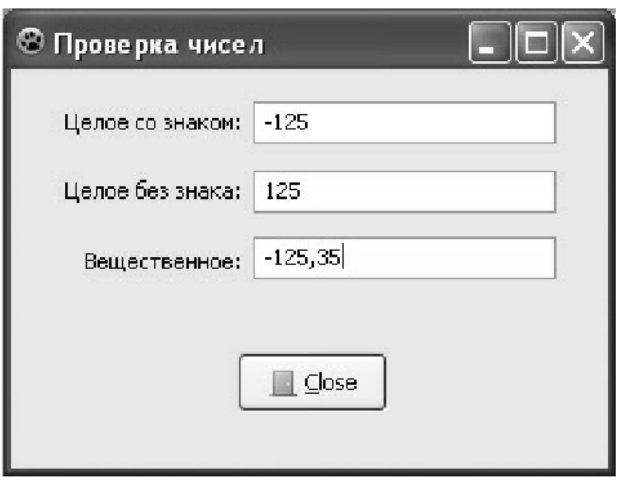


Рис. 17.2. Работающая программа

В дальнейшем, точно таким же образом вы сможете использовать модуль `numbers.pas` для проверки правильности ввода чисел в любых ваших последующих проектах.

Блокнот - шифратор

Данная лабораторная работа закрепляет пройденный материал и показывает на практике реализацию текстового редактора - прототипа стандартного Блокнота Windows. В лекции рассматривается включение модуля стороннего разработчика, практическое применение четырех диалогов.

Цель лекции

Закрепление работы с диалогами, модулями и компонентом `TMemo`.

Постановка задачи

Написать простой Блокнот, пусть даже со всеми возможностями стандартного Блокнота Windows, было бы скучновато. Вот если еще научить его шифровать/дешифровать текст, да еще и с переменным ключом-паролем - совсем другой разговор! Идея такова: делаем Блокнот, добавляем к нему модуль стороннего разработчика, в котором реализованы процедуры шифрации и дешифрации текста. Причем для шифрации/дешифрации требуется ввести пароль - слово или даже фразу. И если хоть один символ пароля будет неверен, текст не расшифруется! Затем вы сможете раздать такой Блокнот всем желающим, но прочитать ваш зашифрованный текст сможет лишь тот, кто знает пароль. Приступим?

Реализация

Подготовка формы очень проста, набор компонентов тоже не очень велик. Для начала откройте Lazarus с новым проектом. Скорее всего, вы эту программу будете передавать друзьям, поэтому имеет смысл сразу же отключить от проекта вставку отладочной информации. Если вы помните, нужно выбрать команду "Проект -> Параметры проекта", в разделе "Параметры компилятора" перейти на подраздел "Компоновка" и убрать флажок "Генерировать отладочную информацию для GDB".

Далее, как обычно, форму называем fMain, сохраняем проект в папку 18-

01 под именем CodeBook, модулю главной формы присваиваем имя Main.

Теперь займемся самой формой. Блокнот Windows имеет возможность менять свой размер, минимизировать и максимизировать окно, оставим такую возможность и нашему Блокноту. Однако по умолчанию, Lazarus выводит слишком маленькое окно. Давайте сделаем окно среднего размера. Пусть будет Height = 500, а Width = 750, и расположите его примерно посреди окна. В свойстве Caption напишите "Блокнот - шифровальщик".

Далее, установим на форму компонент TMemo, имя оставим по умолчанию - Memo1, свойство Align установите в alClient. Текст в окне слишком мелкий, откройте кнопкой "..." редактор свойства Font и выберите шрифт Times New Roman с размером 12 (или любой другой, какой вам понравится больше, только не забывайте, что не у всех пользователей хорошее зрение!). Теперь кнопкой "..." зайдите в редактор свойства Lines и сотрите там весь текст - нам вовсе не нужно, чтобы при открытии программы отображался текст "Memo1", правда?

Займемся остальными компонентами. Нам потребуются диалоги: TOpenDialog, TSaveDialog, TFontDialog и TColorDialog. Поскольку нам придется в коде обращаться к ним по имени, измените их свойства Name соответственно, на OD, SD, FD и CD.

Кроме того, для TOpenDialog и TSaveDialog сделайте следующие настройки:

```
Filter=Текстовые файлы|*.txt|Все файлы|*.*
DefaultExt=.txt
```

Ну, и напоследок, нам потребуется главное меню TMainMenu. Имя самого компонента можно оставить без изменений - MainMenu1, а вот пункты этого меню будем переименовывать. Некоторым подпунктам меню будем присваивать "горячие клавиши". Первым в таблице идут параметры разделов (выделяются жирным шрифтом), затем параметры их подразделов. Подпункты-разделители, в которых будет отражаться разделительная линия, будем называть нейтрально: N1, N2, и т.д. Итак,

создаем следующие пункты и подпункты:

Таблица 18.1. Параметры разделов и подразделов главного меню

Name	Caption	ShortCut
Раздел "Файл"		
FileMenu	Файл	
FileCreate	Создать	Ctrl+N
FileOpen	Открыть	Ctrl+O
FileSave	Сохранить	Ctrl+S
FileSaveAs	Сохранить как…	
N1	-	
FileExit	Выход	
Раздел "Правка"		
EditMenu	Правка	
EditCancel	Отменить	Ctrl+Z
N2	-	
EditCut	Вырезать	Ctrl+X
EditCopy	Копировать	Ctrl+C
EditPaste	Вставить	Ctrl+V
EditDelete	Удалить	
N3	-	
EditSelectAll	Выделить всё	Ctrl+A
Раздел "Формат"		
FormatMenu	Формат	
FormatFont	Шрифт	
FormatColor	Цвет	
N4	-	
FormatWordWrap	Перенос по словам	
Раздел "Кодирование"		
CoderMenu	Кодирование	
CoderCode	Шифровать	

CoderDecode	Дешифровать	
Раздел "Справка"		
HelpMenu	Справка	
HelpAbout	О программе	

В результате у вас должно получиться такое меню:

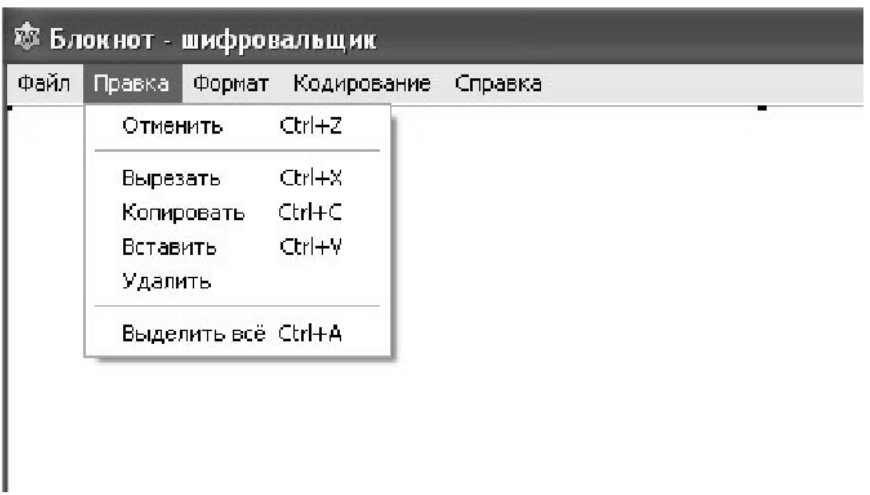


Рис. 18.1. Открытый раздел меню "Правка"

Нам нужно, чтобы по умолчанию, наш Блокнот делал перенос по словам. Для этого понадобится сделать три вещи:

1. Установить в Memo1 вертикальную полосу прокрутки (ScrollBars=ssVertical).
2. Убедиться, что его свойство WordWrap (перенос по словам) установлено в True.
3. Поставить флажок на пункт меню "Формат->Перенос по словам". Для этого в Редакторе меню выделите данный пункт, и установите True в его свойстве Checked. Результат можно будет посмотреть, закрыв редактор меню, и открыв само главное меню:

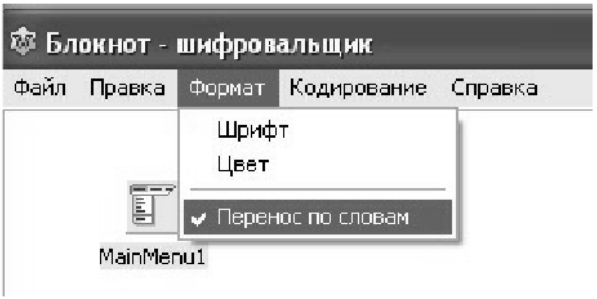


Рис. 18.2. Выделенный флажком пункт меню

Поскольку стандартный Блокнот Windows не имеет картинок в главном меню, мы их тоже привязывать не будем. Впрочем, если есть желание, вы можете самостоятельно добавить `TImageList` и украсить меню картинками. Как это делать, мы обсуждали в лекции №16.

Ну вот, все компоненты мы подготовили, и настроили, как нужно. Но прежде чем перейдем непосредственно к кодированию, подготовим модуль. Я обнаружил этот модуль в проекте `DelphiWorld` много лет назад, и несколько раз использовал, чтобы зашифровать ключи активации программ. Основная часть модуля написана на языке Ассемблер, однако для использования этого модуля знание Ассемблера нам не требуется. Поскольку автор выложил модуль в открытый доступ (спасибо ему большое за это), мы с вами можем использовать его в своих программах с чистой совестью.

Откройте любой редактор текстов, хотя бы тот же стандартный Блокнот Windows. Скопируйте в него нижеследующий листинг, строки

```
//------ Начало модуля ---------
//------- Конец модуля ---------
```

копировать в модуль не нужно они только обозначают границы листинга этого кода:

```
//------ Начало модуля ---------
{ **** UBPFD ********** by delphibase.endimus.com ****
>> Шифрование и дешифрование текстов по принципу S-Coder со скрь
```

После подключения модуля ключевое слово (фраза) будут в MyCript.MyP
Шифровать примерно так:
```
  Memo1.Text:= MyCript.Write(MyCript.Encrypt(Memo1.Text));
```
Дешифровать примерно так:
```
  Memo1.Text:= MyCript.Decrypt(MyCript.Read(Memo1.Text));
```

```
Copyright:   EFD Systems http://www.mindspring.com/~efd
Дата:        23 мая 2002 г.
********************************************************** }

unit MyCript;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

//Это просто интерфейс функций EnCipher и Crypt для шифрования стро
function Encrypt(text: Ansistring): Ansistring;
//Это просто интерфейс функций EnCipher и Crypt для дешифрования ст
function Decrypt(text: Ansistring): Ansistring;

function Write(text: Ansistring): Ansistring;
function Read(text: Ansistring): Ansistring;

var
  MyPassword: AnsiString; //здесь будет пароль

implementation


procedure EnCipher(var Source: AnsiString);

{Low order, 7-bit ASCII (char. 32-127) encryption designed for database use.
 Control and high order (8 bit) characters are passed through unchanged.

 Uses a hybrid method...random table substitution with bit-mangled output.
```

```
  No passwords to worry with (the built-in table is the password). Not industrial
  strength but enough to deter the casual hacker or snoop. Even repeating char.
  sequences have little discernable pattern once encrypted.

  NOTE: When displaying encrypted strings, remember that some characters
        within the output range are interpreted by VCL components; for
        example, '&'.}
begin
{$asmmode intel}
asm
   Push ESI //Save the good stuff
   Push EDI

   Or EAX,EAX
   Jz @Done
   Push EAX
   Call UniqueString
   Pop EAX
   Mov ESI,[EAX] //String address into ESI
   Or ESI,ESI
   Jz @Done
   Mov ECX,[ESI-4] //String Length into ECX
   Jecxz @Done //Abort on null string
   Mov EDX,ECX //initialize EDX with length
   Lea EDI,@ECTbl //Table address into EDI
   Cld //make sure we go forward
@L1:
   Xor EAX,EAX
   Lodsb //Load a byte from string
   Sub AX,32 //Adjust to zero base
   Js @Next //Ignore if control char.
   Cmp AX,95
   Jg @Next //Ignore if high order char.
   Mov AL,[EDI+EAX] //get the table value
   Test CX,3 //screw it up some
   Jz @L2
   Rol EDX,3
@L2:
   And DL,31
```

```
    Xor AL,DL
    Add EDX,ECX
    Add EDX,EAX
    Add AL,32 //adjust to output range
    Mov [ESI-1],AL //write it back into string
  @Next:
    Dec ECX
    Jnz @L1
  // Loop @L1 //do it again if necessary

  @Done:
    Pop EDI
    Pop ESI

    Jmp @Exit
  // Ret //this does not work with Delphi 3 - EFD 971022

  @ECTbl: //The encipher table
    DB 75,85,86,92,93,95,74,76,84,87,91,94
    DB 63,73,77,83,88,90,62,64,72,78,82,89
    DB 51,61,65,71,79,81,50,52,60,66,70,80
    DB 39,49,53,59,67,69,38,40,48,54,58,68
    DB 27,37,41,47,55,57,26,28,36,42,46,56
    DB 15,25,29,35,43,45,14,16,24,30,34,44
    DB 06,13,17,23,31,33,05,07,12,18,22,32
    DB 01,04,08,11,19,21,00,02,03,09,10,20
  @Exit:

  end;//asm
  end;

  procedure DeCipher(var Source: AnsiString);

  {Decrypts a string previously encrypted with EnCipher.}
  begin
  {$asmmode intel}
  asm
    Push ESI //Save the good stuff
    Push EDI
```

```
    Push EBX

    Or EAX,EAX
    Jz @Done
    Push EAX
    Call UniqueString
    Pop EAX
    Mov ESI,[EAX] //String address into ESI
    Or ESI,ESI
    Jz @Done
    Mov ECX,[ESI-4] //String Length into ECX
    Jecxz @Done //Abort on null string
    Mov EDX,ECX //Initialize EDX with length
    Lea EDI,@DCTbl //Table address into EDI
    Cld //make sure we go forward
  @L1:
    Xor EAX,EAX
    Lodsb //Load a byte from string
    Sub AX,32 //Adjust to zero base
    Js @Next //Ignore if control char.
    Cmp AX,95
    Jg @Next //Ignore if high order char.
    Mov EBX,EAX //save to accumulate below
    Test CX,3 //unscrew it
    Jz @L2
    Rol EDX,3
  @L2:
    And DL,31
    Xor AL,DL
    Add EDX,ECX
    Add EDX,EBX
    Mov AL,[EDI+EAX] //get the table value
    Add AL,32 //adjust to output range
    Mov [ESI-1],AL //store it back in string
  @Next:
    Dec ECX
    Jnz @L1
  // Loop @L1 //do it again if necessary
```

```
  @Done:
     Pop EBX
     Pop EDI
     Pop ESI

     Jmp @Exit
  // Ret Does not work with Delphi3 - EFD 971022

  @DCTbl: //The decryption table
     DB 90,84,91,92,85,78,72,79,86,93,94,87
     DB 80,73,66,60,67,74,81,88,95,89,82,75
     DB 68,61,54,48,55,62,69,76,83,77,70,63
     DB 56,49,42,36,43,50,57,64,71,65,58,51
     DB 44,37,30,24,31,38,45,52,59,53,46,39
     DB 32,25,18,12,19,26,33,40,47,41,34,27
     DB 20,13,06,00,07,14,21,28,35,29,22,15
     DB 08,01,02,09,16,23,17,10,03,04,11,05
  @Exit:
  end;//asm
  end;

  procedure Crypt(var Source: Ansistring; const Key: AnsiString);

  {Encrypt AND decrypt strings using an enhanced XOR technique similar to
   S-Coder (DDJ, Jan. 1990). To decrypt, simply re-apply the procedure
   using the same password key. This algorithm is reasonably secure on
   it's own; however,there are steps you can take to make it even more
   secure.

        1) Use a long key that is not easily guessed.
        2) Double or triple encrypt the string using different keys.
           To decrypt, re-apply the passwords in reverse order.
        3) Use EnCipher before using Crypt. To decrypt, re-apply Crypt
           first then use DeCipher.
        4) Some unique combination of the above

   NOTE: The resultant string may contain any character, 0..255.}
  begin
  {$asmmode intel}
```

```
  asm
    Push ESI //Save the good stuff
    Push EDI
    Push EBX

    Or EAX,EAX
    Jz @Done
    Push EAX
    Push EDX
    Call UniqueString
    Pop EDX
    Pop EAX
    Mov EDI,[EAX] //String address into EDI
    Or EDI,EDI
    Jz @Done
    Mov ECX,[EDI-4] //String Length into ECX
    Jecxz @Done //Abort on null string
    Mov ESI,EDX //Key address into ESI
    Or ESI,ESI
    Jz @Done
    Mov EDX,[ESI-4] //Key Length into EDX
    Dec EDX //make zero based
    Js @Done //abort if zero key length
    Mov EBX,EDX //use EBX for rotation offset
    Mov AH,DL //seed with key length
    Cld //make sure we go forward
  @L1:
    Test AH,8 //build stream char.
    Jnz @L3
    Xor AH,1
  @L3:
    Not AH
    Ror AH,1
    Mov AL,[ESI+EBX] //Get next char. from Key
    Xor AL,AH //XOR key with stream to make pseudo-key
    Xor AL,[EDI] //XOR pseudo-key with Source
    Stosb //store it back
    Dec EBX //less than zero ?
    Jns @L2 //no, then skip
```

```
    Mov EBX,EDX //re-initialize Key offset
  @L2:
    Dec ECX
    Jnz @L1
  @Done:
    Pop EBX //restore the world
    Pop EDI
    Pop ESI
  end;//asm
  end;

  function Encrypt(text: Ansistring): Ansistring;
  //Это просто интерфейс функций EnCipher и Crypt для шифрования стро
  begin
    {шифруем текст}
    EnCipher(Text);
    {зашифровываем ключом}
    Crypt(Text, MyPassword);
    Result := Text;
  end;

  function Decrypt(text: Ansistring): Ansistring;
  //Это просто интерфейс функций EnCipher и Crypt для дешифрования ст
  begin
    {расшифровываем ключом}
    Crypt(Text, MyPassword);
    {расшифровываем результат}
    DeCipher(Text);
    Result := Text;
  end;

  function Write(text: Ansistring): Ansistring;
  var
    i: integer;
  begin
    Result := '';
    for i := 1 to Length(text) do
      {получаем hex код из текста}
      Result := Result + InttoHex(ord(text[i]), 2);
```

```
end;

function Read(text: Ansistring): Ansistring;
var
  i: integer;
begin
  Result := '';
  for I := 1 to Length(text) do
    if odd(i) then
      {получаем текст из hex кода}
      Result := Result + Chr(StrtoInt('$' + text[i] + text[i + 1]));
end;

end.
//------- Конец модуля ---------
```

Листинг .

Сохраните модуль в файл MyCript.pas туда, куда вы решили собирать свою коллекцию модулей. Не забудьте скопировать этот файл в папку с нашим текущим проектом.

Также в Редакторе кода нашей программы добавьте модуль `MyCript` в раздел `uses`, в конец списка подключаемых модулей.

Теперь можем приступить непосредственно к кодированию команд. Вы помните, что чтобы сгенерировать команду `OnClick` пункта меню, достаточно выбрать эту команду? Тогда начнем с более простых пунктов меню. Самым первым обработаем пункт "Формат->Перенос по словам":

```
procedure TfMain.FormatWordWrapClick(Sender: TObject);
begin
  //изменяем меню:
  FormatWordWrap.Checked:= not FormatWordWrap.Checked;
  //присваиваем настройку Memo1:
  Memo1.WordWrap:= FormatWordWrap.Checked;
  //если перенос по словам включен, нужна только вертикальная
  //линейка прокрутки, иначе нужны обе линейки:
  if Memo1.WordWrap then Memo1.ScrollBars:= ssVertical
```

```
  else Memo1.ScrollBars:= ssBoth;
end;
```

Мы с вами установили по умолчанию, что текст в `Memo1` переносится по словам, а данный пункт меню отмечен флажком. При выборе этой команды происходит обратное действие: состояние `Checked` пункта меню меняется на противоположное, состояние `WordWrap` компонента `Memo1` становится таким же. То есть, если флажок установлен, то и перенос будет работать, и наоборот. В заключение мы устанавливаем нужное состояние полос прокрутки. Если перенос работает, то горизонтальная полоса не нужна, только вертикальная. Если же переноса нет, то строка может получиться очень длинной, тут понадобятся обе полосы прокрутки.

Далее обработаем команды меню "Правка". Они очень простые, всего на команду придется ввести по одной строке кода, так что дополнительные комментарии не нужны. Меню "Правка->Отменить":

```
procedure TfMain.EditCancelClick(Sender: TObject);
begin
  Memo1.Undo;
end;

"Правка->Вырезать" не сложнее:
procedure TfMain.EditCutClick(Sender: TObject);
begin
  Memo1.CutToClipboard;
end;

"Правка->Копировать":
procedure TfMain.EditCopyClick(Sender: TObject);
begin
  Memo1.CopyToClipboard;
end;

"Правка->Вставить":
procedure TfMain.EditPasteClick(Sender: TObject);
begin
```

```
  Memo1.PasteFromClipboard;
end;

"Правка->Удалить":
procedure TfMain.EditDeleteClick(Sender: TObject);
begin
  Memo1.ClearSelection;
end;

"Правка->Выделить всё":
procedure TfMain.EditSelectAllClick(Sender: TObject);
begin
  Memo1.SelectAll;
end;
```

С пунктом "Правка" разобрались, вернемся к пункту "Формат". У нас остались необработанными два подпункта - "Шрифт" и "Цвет". Процедура `OnClick` для пункта "Формат->Шрифт":

```
procedure TfMain.FormatFontClick(Sender: TObject);
begin
  //сначала диалогу присваиваем шрифт как у Memo:
  FD.Font:= Memo1.Font;
  //если диалог прошел успешно, меняем шрифт у Memo:
  if FD.Execute then Memo1.Font:= FD.Font;
end;
```

Здесь мы сначала свойству `Font` диалога присвоили такой же шрифт, как и у `Memo1`. Зачем нужно было это делать? Если бы мы этого не сделали, диалог не знал бы, какой шрифт считается "текущим", там была бы пустая строка. И пользователю сложно было бы разобраться, какой шрифт, какого размера он сейчас использует. Теперь же, при вызове диалога, сразу будет выделен наш шрифт Times New Roman, 12. Если же пользователь изменит этот шрифт, то свойству `Font` компонента `Memo1` будет присвоен новый шрифт.

Примерно также обрабатываем и пункт "Формат->Цвет", только здесь

мы обращаемся к свойству `Color` и диалогу `CD`:

```
procedure TfMain.FormatColorClick(Sender: TObject);
begin
  //сначала устанавливаем цвет диалога, как у Memo:
  CD.Color:= Memo1.Color;
  //если диалог прошел успешно, меняем цвет у Memo:
  if CD.Execute then Memo1.Color:= CD.Color;
end;
```

Теперь перейдем к более сложному пункту меню "Файл". Сначала самая простая команда, "Файл->Выход":

```
procedure TfMain.FileExitClick(Sender: TObject);
begin
  Close;
end;
```

Команда "Файл->Сохранить как" чуть сложнее:

```
procedure TfMain.FileSaveAsClick(Sender: TObject);
begin
  {Переписываем заголовок окна диалога, иначе он выйдет на английск
   Если сохранение произошло, то свойство Modified у Memo переводим
   так как все изменения уже сохранены}
  SD.Title:= 'Сохранить как';
  if SD.Execute then begin
    Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
    Memo1.Modified:= false;
  end; //if
end;
```

Здесь мы вначале изменили заголовок. Почему-то диалог, хоть и имеет по умолчанию текст в `Title` на русском языке, в работающей программе все равно выходит по-английски, хотя диалоги шрифта и цвета выводят нормальный русский заголовок. Может быть, в будущих версиях Lazarus эта недоработка будет устранена? Пока же будем менять

заголовки внутри кода. Далее мы сохраняем текст из `Memo1` в файл, причем делаем это не так, как обычно:

```
Memo1.Lines.SaveToFile(SD.FileName);
```

а так:

```
Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
```

Зачем такие усложнения? Дело в том, что имя файла (свойство `FileName` диалога) сохраняется в формате UTF8, а процедура `SaveToFile` требует формата ANSI. Это не проблема, если вы будете давать файлу имя исключительно латинскими буквами или цифрами. Но если вы попробуйте сохранить файл с именем под русскими символами (или любыми неанглийскими), то получится то, что называют "кракозябрами" - непонятная абракадабра вместо имени. Такой файл и открыть потом не получится. Если же преобразовать имя файла в нужный формат, это гарантирует правильное сохранение (загрузку) этого файла, на каком бы языке вы не написали его имя.

Далее мы устанавливаем в `False` свойство `Modified` компонента `Memo1`. Это важно! Свойство `Modified` становится `True`, если в `Memo1` есть какие-то изменения. Если же мы эти изменения сохранили в файл, будем считать, что других изменений пока нет. Нам придется еще проверять состояние этого свойства.

Теперь обработаем команду "Файл->Сохранить":

```
procedure TfMain.FileSaveClick(Sender: TObject);
begin
  {если имя файла известно, то не нужно вызывать диалог SaveDialog,
    просто вызываем метод SaveToFile. }
  if SD.FileName <> '' then begin
    Memo1.Lines.SaveToFile(Utf8ToSys(SD.FileName));
    //устанавливаем Modified в false, так как изменения уже сохранили:
    Memo1.Modified:= false;
  end //if
  //иначе имя файла не известно, вызываем Сохранить как...:
```

```
  else FileSaveAsClick(Sender);
end;
```

Смотрите, что тут происходит. Мы смотрим, есть ли какой-нибудь текст в свойстве `FileName` диалога `SD`. Если текст есть (свойство `FileName` не равно пустой строке), значит, диалог `SD` уже вызывался, или же открывался существующий файл. В любом случае, мы знаем имя файла, куда нужно сохранять данный текст. Сохраняем, переводим в `False` свойство `Modified` нашего `Memo1`. Если же там текста нет, значит, текст новый, еще ни разу не сохранялся. В этом случае, нам нужно вызвать метод `OnClick` команды "Файл->Сохранить как". Обратите внимание, как мы это делаем:

```
FileSaveAsClick(Sender);
```

Дело в том, что в качестве параметра в метод нужно передать объект, откуда этот метод был вызван. Переменная `Sender` имеет тип `TObject` и обычно указывает на этот объект. Мы могли бы указать и какой то конкретный объект, например, так:

```
FileSaveAsClick(Memo1);
```

В данном случае не будет никакой разницы, команда все равно сработает одинаково.

Команда "Файл->Создать" чуть сложнее:

```
procedure TfMain.FileCreateClick(Sender: TObject);
begin
  {Если есть изменения текста, спросим пользователя, не хочет ли он сс
  их перед созданием нового текста}
  if Memo1.Modified then begin
    //если пользователь согласен сохранить изменения:
    if MessageDlg('Сохранение файла',
                  'Текущий файл был изменен. Сохранить изменения?',
        mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
          FileSaveClick(Sender);
```

```
    end; //if
    //теперь очищаем Memo, если есть текст:
    if Memo1.Text <> '' then Memo1.Clear;
    //в SaveDialog убираем имя файла. это будет означать, что файл не сохр
    SD.FileName:= '';
  end;
```

Когда пользователь выбирает команду "Файл->Создать", в `Memo` уже мог быть какой-то текст. Он нужен пользователю, или нам просто нужно очистить `Memo`, и подготовить его к новому тексту? Для этого, в случае, если у `Memo` есть изменения, мы выводим запрос-сообщение `MessageDlg()`. Если пользователь желает сохранить старый текст, мы вызываем команду "Файл->Сохранить". И в любом случае, затем мы очищаем `Memo`, если там что-то есть, и очищаем свойство `FileName` у диалога `SD (TSaveDialog)`. Ведь текст новый, и мы еще не знаем, куда пользователь захочет его сохранить, верно?

Код команды "Файл->Открыть" самый длинный, хотя и в нем ничего сложного нет:

```
  procedure TfMain.FileOpenClick(Sender: TObject);
  begin
    //проверка необходимости сохранения файла, как в Файл->Создать:
    if Memo1.Modified then begin //изменения есть
      //если пользователь согласен сохранить изменения:
      if MessageDlg('Сохранение файла',
                    'Текущий файл был изменен. Сохранить изменения?',
          mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
            FileSaveClick(Sender);
    end; //if

    //очищаем имя файла у диалога OpenDialog, изменяем заголовок, и
    //вызываем метод LoadFromFile, если диалог состоялся
    OD.FileName:= '';
    OD.Title:= 'Открыть существующий файл';
    if OD.Execute then begin
      //очищаем Memo, если есть текст:
      if Memo1.Text <> '' then Memo1.Clear;
```

```
    //читаем из файла
    Memo1.Lines.LoadFromFile(Utf8ToSys(OD.FileName));
    //копируем имя файла в диалог SaveDialog, чтобы потом знать,
    //куда сохранять:
    SD.FileName:= OD.FileName;
  end; //if
end;
```

В первой части процедуры мы смотрим, нет ли изменений в `Memo`, ведь пользователь мог редактировать один файл, а затем захотел открыть другой! Нужно ли сохранять эти изменения, если они есть? Как и в прошлом примере, для этого мы выводим запрос `MessageDlg()`. Сохраняем, если пользователь этого хочет.

Далее мы очищаем свойство `FileName` у диалога открытия файла. Зачем? Ладно, если этот диалог вызывается первый раз, тогда это свойство будет пусто. А если это не первый файл, который открывается пользователем? Если свойство не очистить, то при вызове диалога прежний файл будет там по умолчанию. Но пользователь то хочет открыть другой файл! Поэтому мы предварительно очищаем это свойство, а уж потом вызываем диалог. В коде мы изменяем заголовок и у этого диалога, иначе он тоже выйдет на английском языке.

Затем мы чистим `Memo`, если там есть старый текст, и загружаем новый текст из нового файла, после чего копируем имя этого файла в диалог сохранения, чтобы знать, куда сохранять изменения при выборе команды "Файл->Сохранить".

Теперь мы с вами должны предусмотреть еще вот что. Допустим, пользователь что-то записал в блокнот, а потом решил его закрыть. Изменения есть, а надо ли их сохранять? Нужно спросить об этом у пользователя. Это можно было бы сделать в команде "Файл->Выход", но мы этого не сделали. Почему? А если пользователь закроет программу не этой командой, а кнопкой с крестиком в верхней правой части окна? Или кнопками <Alt+F4>? Вот чтобы обработать закрытие программы, каким бы образом пользователь её не закрывал, воспользуемся событием `OnClose` формы главного окна, которое возникает ПЕРЕД закрытием программы. Проблема в том, что компонент `Memo1`

закрывает всю форму, а нам нужно ее выделить. Выделить форму fMain можно либо в верхней части окна Инспектора объектов, где все объекты отображены в виде дерева - форма там расположена в самом верху, как главный, родительский компонент. Либо можно сделать проще - выделите `Memo1` и нажмите <Esc>, при этом выделение перейдет на внешний к `Memo1` компонент, то есть, к форме. Затем перейдите на вкладку События Инспектора объектов, и сгенерируйте событие `OnClose`. Его код вам уже знаком по предыдущему коду:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction: TCloseAction
begin
  {Если есть изменения текста, спросим пользователя, не хочет ли он сс
  их перед созданием нового текста}
  if Memo1.Modified then begin
    //если пользователь согласен сохранить изменения:
    if MessageDlg('Сохранение файла',
                  'Текущий файл был изменен. Сохранить изменения?',
        mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
          FileSaveClick(Sender);
  end; //if
end;
```

Листинг .

Теперь каким бы способом пользователь не закрывал программу, будет проверяться - нет ли изменений в `Memo`, и если есть, выйдет запрос - сохранять ли их. Практически всё, что умеет делать стандартный Блокнот, умеет теперь и наша программа. Самое время заняться шифрованием.

Код "Кодирование->Шифровать" очень простой:

```
procedure TfMain.CoderCodeClick(Sender: TObject);
begin
  //сначала очистим ключ:
  MyCript.MyPassword:= '';
  if InputQuery('Ввод ключа', 'Введите ключевое слово (фразу):',
    MyCript.MyPassword) then
      Memo1.Text:= MyCript.Write(MyCript.Encrypt(Memo1.Text));
```

```
end;
```

Вначале мы очищаем ключ. Делается это на всякий случай, вдруг это уже не первое шифрование, тогда в глобальной переменной `MyPassword` будет храниться предыдущий ключ. Или представьте такую ситуацию: пользователь вводил личный текст, ему понадобилось отлучиться, и он его зашифровал. Если ключ в переменной сохраняется, любой может подойти и расшифровать его. Элементарная безопасность требует, чтобы мы всегда очищали ключ.

Затем, с помощью `InputQuery()` мы выводим запрос, в котором пользователь может ввести пароль - ключевое слово или фразу. Этот ключ мы помещаем в переменную `MyPassword`, после чего вызываем шифрацию текста Memo так, как было указано в рекомендациях модуля `MyCript`. Дешифрация ("Кодирование->Дешифровать") происходит похожим образом:

```
procedure TfMain.CoderDecodeClick(Sender: TObject);
begin
  //сначала очистим ключ:
  MyCript.MyPassword:= '';
  if InputQuery('Ввод ключа', 'Введите ключевое слово (фразу):',
    MyCript.MyPassword) then
      Memo1.Text:= MyCript.Decrypt(MyCript.Read(Memo1.Text));
end;
```

Вот и весь шифратор. Сохраните его, скомпилируйте и опробуйте в работе. Имейте в виду, что если ввести неправильный пароль, то текст будет дешифрован неправильно и не прочитается. Обратного действия не предусмотрено в тех же целях безопасности. И ещё: осталась необработанной команда меню "Справка->О программе", мы вернемся к ней в другой лекции, где будем изучать многооконные приложения, так что не удаляйте пока этот проект. Он нам также понадобится в лекциях 28 и 29, где мы будем создавать для этого проекта справочную систему и инсталлятор.

Деревья

Лекция посвящена работе с древовидными иерархическими структурами данных, и компонентом TTreeView. На практике показаны все основные действия с компонентом.

Цель лекции

Применение компонента `TTreeView` для работы с древовидными иерархическими данными.

Дерево TTreeView

Компонент `TTreeView` расположен на вкладке Common Controls Палитры компонентов, и предназначен для отображения различных древовидных иерархических структур:

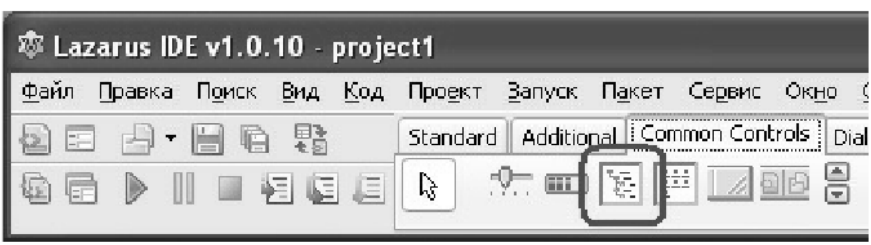


Рис. 19.1. Компонент TTreeView

Мы часто сталкиваемся с иерархической информацией. Это может быть т.н. Дерево каталогов в стандартном Проводнике Windows, которое отображает в виде ветвей дерева внешние и вложенные папки, и файлы, которые там хранятся. Это может быть структура предприятия, библиотечный каталогизатор, структура книги, реферата, курсовой… В общем, самая разная информация, в которой присутствуют родительские и дочерние (вложенные) объекты. Все подобные объекты можно отобразить с помощью компонента `TTreeView`.

Объекты, которые содержаться в данном компоненте называются узлами (англ. node), а сам компонент представляет собой список узлов. Подобно спискам `TListBox` и `TComboBox`, с которыми мы уже

знакомы, TTreeView имеет свойство Items - индексированный список узлов. Каждый узел - это объект, который имеет тип TTreeNode. И родительские, и вложенные в них дочерние элементы - всё это узлы (объекты) типа TTreeNode. Причем дочерний элемент одновременно может быть родителем по отношению к другому узлу, уровень вложенности неограничен.

Но давайте-ка всё по порядку. Рассмотрим работу компонента на примере библиотечного каталога. Откройте Lazarus с новым проектом. Как обычно, форму назовите fMain, проект сохраните под именем MyLibrary в папку 19-01, модулю формы дайте имя Main. В свойстве Caption формы напишите Библиотечный каталог. Саму форму немного растяните, пусть у нас будет высота 350, а ширина 500 пикселей.

Далее, установим на форму простую панель TPanel, из нее мы сделаем своеобразную Панель инструментов программы. У панели очистите свойство Caption, в свойстве Align установите alLeft, в свойстве Width 127 пикселей.

Далее, на полученную Панель инструментов одну за другой установите 7 простых кнопок TButton. У всех кнопок в свойстве Left установите значение 1, а в свойстве Width - значение 125. Верхнюю кнопку расположите повыше (Top = 1), остальные - чуть ниже, чтобы между кнопками было совсем небольшое расстояние. Нам нужно переименовать кнопки и сделать на них соответствующие надписи. Сделайте следующие настройки кнопок:

Таблица 19.1. Настройки свойств кнопок

Name	Caption
bNewNode	Новый раздел
bNewChildNode	Новый подраздел
bDelete	Удалить
bEdit	Переименовать
bSort	Сортировать
bCollapse	Свернуть список

`bExpand`	Развернуть список

Теперь, правее панели, с вкладки Common Controls установите компонент `TTreeView`. Поскольку дерево у нас одно, переименовывать его мы не будем. В свойстве `Align` дерева также установите значение `alLeft`, в свойстве `Width` установите 360 пикселей. Поверх `TTreeView` с этой же вкладки установите список изображений `TImageList`, его тоже переименовывать не будем. В результате у нас должна получиться вот такая форма:

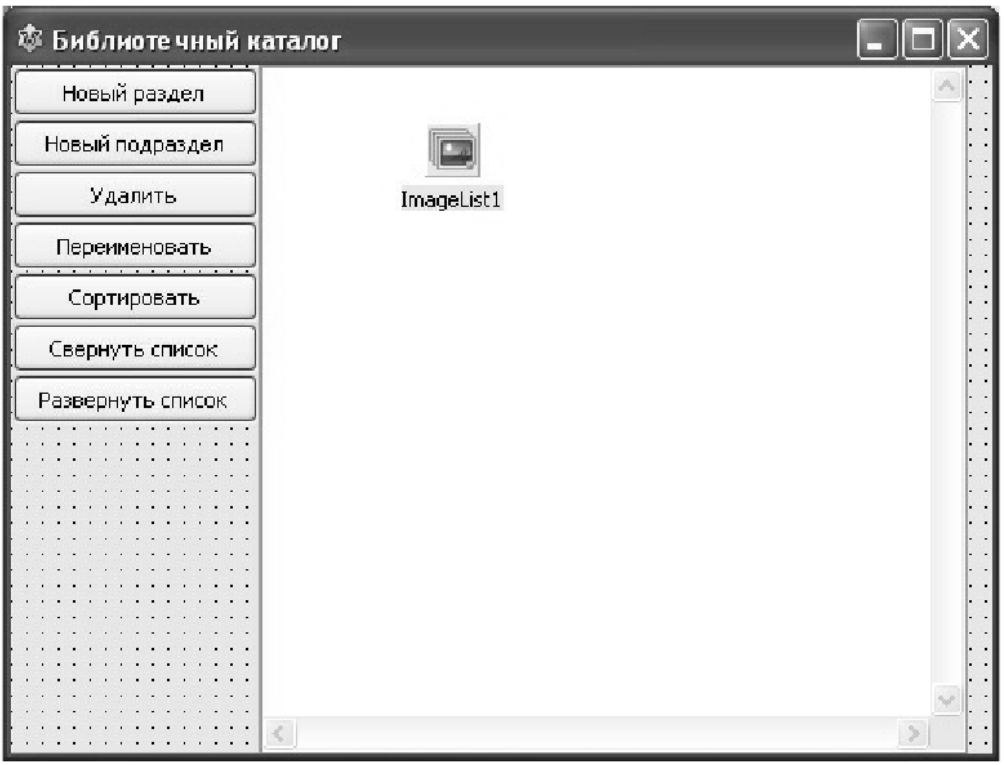


Рис. 19.2. Полученная форма

Как видите, в правой части формы после `TreeView1` осталось немного места. Сейчас мы научимся только производить различные действия с компонентом `TTreeView`, а позже, в другой лекции, мы вернемся к этому проекту, и в правой части сделаем отображение книг, которые относятся к данному разделу библиотеки.

Итак, выделите дерево `TreeView1` и обратите внимание на его свойства. Мы рассмотрим только основные, с которыми еще не сталкивались в других компонентах.

Свойства TTreeView

`AutoExpand`	- если `True`, то дерево автоматически распахивается, когда программа начинает работать, иначе список остается свернутым. Нам желательно, чтобы дерево было распахнуто, поэтому установите тут значение `True`.
`HideSelection`	- если `True`, то когда компонент теряет фокус (активным становится другой компонент), выделение текущего узла будет скрыто. Ну, это дело вкуса, можно оставить значение по умолчанию - `True`.
`Images`	- тут нужно выбрать список изображений, если хотите, чтобы у родительских и дочерних узлов были разные пиктограммы. Выберем тут `ImageList1`, а подходящие картинки подберем позже.
`Indent`	- расстояние в пикселях от левого узла до его подузлов. По умолчанию равно 15 пикселей, и я рекомендую это расстояние не уменьшать. Иначе узлы дерева будут расположены близко друг от друга, как бы слипнуться, что затруднит чтение списка.
`Items`	- сам список, основное свойство компонента. При нажатии на кнопку "..." выходит Редактор элементов дерева:

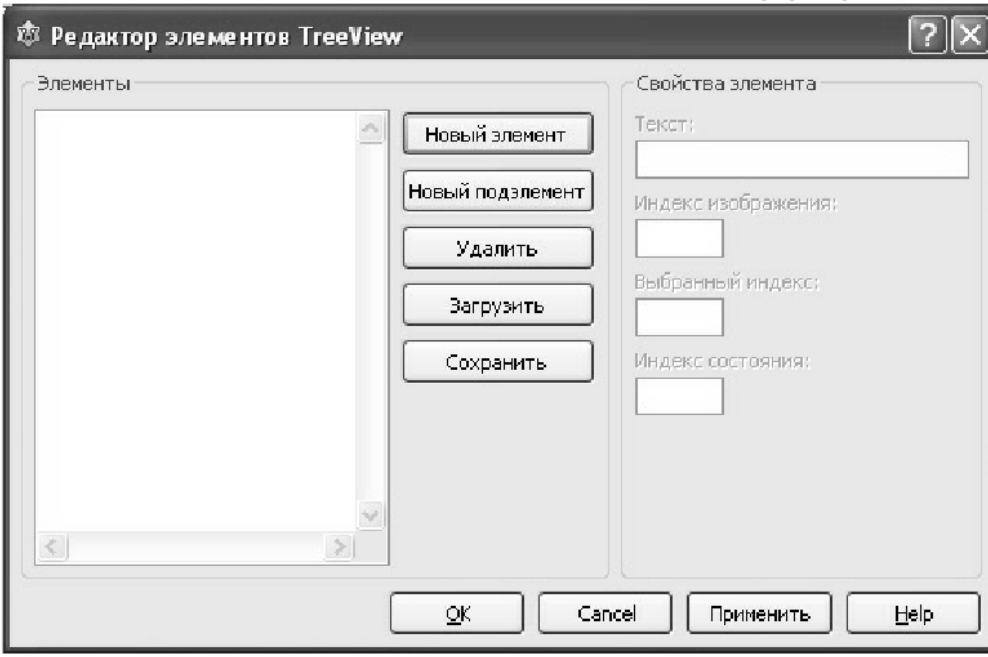


Рис. 19.3. Редактор элементов компонента TTreeView

Как видите, кнопки практически идентичны тем, что мы установили на форму, и работают примерно также. Так что мы не будем рассматривать работу с этим редактором, поскольку почти всегда дерево заполняют не вручную, а программно. Обратиться к отдельному узлу дерева можно через свойство `Items`, указав индекс элемента; индексация начинается с нуля. Например, к первому узлу в списке обращаются так: `TreeView1.Items[0]`.

`MultiSelect`	- если `True`, то разрешает одновременное выделение нескольких узлов; если `False` - запрещает.
	- переключатели, которые определяют способ выделения нескольких узлов. Переключателей четыре, если они имеют значение `True`, то данный способ работает, иначе не работает: • `msControlSelect` - с нажатой и удерживаемой <CTRL>, когда щелкают по

MultiSelectStyle	элементам списка в произвольном порядке. • msShiftSelect - с нажатой и удерживаемой <SHIFT>, когда выбирают сразу диапазон элементов, щелкая сперва по первому, затем по последнему элементу. • msSiblingOnly - как msShiftSelect, но в диапазон включаются только узлы одного уровня. • msVisibleOnly - как msShiftSelect, но в диапазон не включаются нераскрытые дочерние узлы.
RightClickSelect	- при True узлы выделяются как левой, так и правой кнопкой мыши; при False - только левой.
RowSelect	- при True выделяется вся строка элемента, при False - только его имя. Бесполезное свойство, надо сказать.
ScrollBars	- полосы прокрутки, с ними вы знакомы по компоненту TMemo. Рекомендую устанавливать в этом свойстве значение ssAutoBoth.
ShowButtons	- разрешает (при True) или запрещает показ кнопок возле узлов. Обычно разрешает, тогда слева от закрытого узла будет кнопка с плюсом, от открытого - с минусом.
ShowLines	- разрешает показ линий ветвей дерева. Желательно оставлять True.
ShowRoot	- разрешает показ линии, идущей от корня дерева. Если ShowLines = False, то никаких линий не будет показано в любом случае.
	- способ сортировки списка. К этому свойству мы еще вернемся в программе. Может быть:

SortType	• stNone - нет сортировки. • stText - сортировка по тексту. • stData - сортировка по данным. • stBoth - сортировка и по тексту, и по данным.

TreeLineColor и TreeLinePenStyle отвечают за цвет и тип линий ветвей дерева.

Методы TTreeView

AlphaSort	- еще один способ отсортировать список. Вызывается так: TreeView1.AlphaSort; Метод возвращает True, если сортировка прошла успешно, и False в другом случае.
FullCollapse	- сворачивает дерево, скрывая все его дочерние узлы.
FullExpand	- наоборот, разворачивает дерево.
LoadFromFile	- метод считывает информацию об узлах из текстового файла. Файл сохраняется в формате UTF-8. Дочерние узлы находятся ниже родительских, и сдвинуты вправо символом табулятора. Перед использованием желательно проверять реальное наличие файла.
SaveToFile	- наоборот, сохраняет структуру дерева в текстовый файл.

События TTreeView

Компонент имеет несколько специфичных событий, которые могут оказаться полезными.

OnChange	- возникает после перемещения на другой узел.
OnChanging	- возникает перед перемещением на другой узел. При желании, можно сгенерировать это событие, и запретить пользователю этот переход, например, если не выполнено какое-то условие.
OnCollapsed	- возникает после свертывания узла. Пользователь может вызвать это событие, щелкнув по кнопке "-" рядом с узлом.
OnCollapsing	- возникает после свертывания узла.
OnCompare	- возникает, когда обработчик при сортировке списка сравнивает два узла и решает, который должен быть выше.
OnDeletion	- возникает при удалении узла.
OnEdited	- возникает после редактирования пользователем надписи узла.
OnEditing	- возникает во время редактирования надписи узла.
OnExpanded	- возникает после разворачивания узла.
OnExpanding	- возникает после разворачивания узла.
OnGetImageIndex	- возникает при определении индекса изображения у узла из связанного TImageList.
OnGetSelectedIndex	- возникает при определении индекса текущего узла.

Свойства и методы TTreeView.Items

Как уже упоминалось, при программной обработке дерева (а чаще всего, такая обработка и используется) приходится пользоваться свойством Items, которое имеет тип TTreeNodes и само является объектом, а потому имеет собственный набор свойств и методов. Разберем основные из них:

Свойства

`Count`	- Количество узлов, входящих в дерево, включая и вложенные.
`Item[Index:Integer]`	- Индексированный доступ к узлам. Как всегда, индексация начинается с нуля, поэтому, к примеру, изменить текст первого узла дерева можно так: `TreeView1.Item[0].Text:= 'Новый текст';`

Нередко возникает необходимость обойти всё дерево, от первого до последнего узла. Например, изменить текст всех узлов можно так (пример выполнять не нужно):

```
var
  i: integer;
begin
  for i:= 0 to TreeView1.Items.Count-1 do
    TreeView1.Item[i].Text:= 'Узел №' + IntToStr(i+1);
```

В этом случае мы получим дерево с текстом

```
Узел №1
Узел №2
```

и так далее.

Методы

	- добавляет новый узел в конец списка `SiblingNode`, с

`Add(SiblingNode:TTreeNode; const S):TTreeNode`	текстом `S`. Если добавляется корневой (не имеющий родителя) узел, то `SiblingNode = NIL`.
`AddChild(ParentNode:TTreeNode; const S):TTreeNode`	- добавляет новый дочерний (вложенный) узел в конец родительского списка `ParentNode`, с текстом `S`.
`AddFirst(SiblingNode:TTreeNode; const S):TTreeNode`	- добавляет новый узел в начало списка `SiblingNode`, с текстом `S`. Если добавляется корневой (не имеющий родителя) узел, то `SiblingNode = NIL`.
`AddChildFirst(ParentNode:TTreeNode; const S):TTreeNode`	- добавляет новый дочерний (вложенный) узел в начало родительского списка `ParentNode`, с текстом `S`.
`Clear`	- очищает список дерева от всех

	узлов.
`Delete(Node:TTreeNode)`	- удаляет узел `Node`.

Основные свойства узла TTreeNode

К отдельному узлу можно получить доступ через свойство `Item`, например, `TreeView1.Item[0]`.

`ImageIndex`	- содержит индекс пиктограммы из связанного `TImageList`, которая будет отображаться в данном узле.
`Text`	- содержит текст, отображаемый в ветке данного узла.

Пожалуйста, не путайте - свойство компонента `Items` имеет тип `TTreeNodes`, и представляет собой индексированный список узлов. А отдельный узел имеет тип `TTreeNode` - это не одно и то же!

Вернемся к программе. Найдем две подходящих пиктограммы для узлов дерева. Щелкните дважды по `ImageList1`, чтобы открыть его редактор. Нажмите кнопку "Добавить". Если вы устанавливали Lazarus в папку по умолчанию, то различные стандартные картинки будут расположены по адресу

C:\Lazarus\Images

Не будем копаться по вложенным папкам - прямо тут находятся две подходящих пиктограммы. Первым добавьте изображение folder.png, это будет пиктограмма для родительских узлов. Пиктограмма получила индекс 0. Вторым добавьте изображение template.png, это будет изображение вложенных подразделов. Картинка встала под индексом 1. Можно закрыть редактор ImageList кнопкой "OK".

Теперь приступим к программированию кнопок. Сгенерируйте событие `OnClick` для кнопки "Новый раздел". Её код будет следующим:

```
procedure TfMain.bNewNodeClick(Sender: TObject);
```

```
var
  NodeCaption: string; //для получения заголовка нового узла
  NewNode: TTreeNode; //для создания нового узла
begin
  //сначала очистим заголовок:
  NodeCaption:= '';
  //теперь, если пользователь не ввел заголовок нового узла, выходим:
 if not InputQuery('Ввод заголовка', 'Введите заголовок раздела',
                   NodeCaption) then exit;
  //если мы здесь, то заголовок есть. создаем родительский узел:
  NewNode:= TreeView1.Items.Add(nil, NodeCaption);
  //присваиваем ему картинку под индексом 0:
  NewNode.ImageIndex:=0;
end;
```

Как видно из кода и комментариев, эта кнопка добавляет в конец списка узлов новый родительский узел. Функцией-запросом `InputQuery()` мы получаем у пользователя заголовок для будущего узла. Если пользователь закрыл диалог, не введя этого заголовка, то мы просто выходим из события, ничего не предпринимая. Но если он что-то туда ввел, то этот заголовок попадает в переменную `NodeCaption`, и мы приступаем к созданию родительского узла, что и делает код:

```
  NewNode:= TreeView1.Items.Add(nil, NodeCaption);
```

То, что узел родительский, говорит параметр `nil` - ничто, указывающий, что у нового узла нет родителя.

В заключение мы присваиваем этому узлу картинку под индексом 0, если помните, там изображение папки.

Код для кнопки "Новый подраздел" очень похож на предыдущий:

```
procedure TfMain.bNewChildNodeClick(Sender: TObject);
var
  NodeCaption: string;
  NewNode: TTreeNode;
begin
```

```
  NodeCaption:= '';
  if not InputQuery('Ввод заголовка', 'Введите заголовок подраздела',
                    NodeCaption) then exit;
  NewNode:= TreeView1.Items.AddChild(TreeView1.Selected, NodeCaption);
  if NewNode.Parent = nil then NewNode.ImageIndex:=0
  else NewNode.ImageIndex:=1;
end;
```

Разницы тут две. Во-первых, в этот раз мы используем метод `Items.AddChild`, который добавляет именно дочерний узел. В параметре вместо `nil` вы видите уже `TreeView1.Selected`, что означает ссылку на выделенный в данный момент раздел. Именно для этого раздела будет создаваться подраздел.

Во-вторых, мы указываем не конкретный индекс картинки, которая будет тут отображаться, а делаем проверку:

```
  if NewNode.Parent = nil then NewNode.ImageIndex:=0
  else NewNode.ImageIndex:=1;
```

Если у нового узла все-таки нет родителя (`NewNode.Parent = nil`), то присваиваем узлу изображение 0, иначе это будет изображение 1.

Для кнопки "Удалить" код будет следующим:

```
procedure TfMain.bDeleteClick(Sender: TObject);
begin
  if TreeView1.Selected <> nil then
    TreeView1.Items.Delete(TreeView1.Selected);
end;
```

Код очень простенький. Если выделенный узел не равен `nil` (то есть, если вообще какой-то узел выделен), то мы удаляем из списка этот выделенный узел.

Код для кнопки "Переименовать":

```
procedure TfMain.bEditClick(Sender: TObject);
var
  NodeCaption: string;
begin
  NodeCaption:= '';
  if not InputQuery('Ввод заголовка', 'Введите новый заголовок',
                    NodeCaption) then exit;
  TreeView1.Selected.Text:= NodeCaption;
end;
```

Здесь мы точно также пытаемся получить у пользователя новый заголовок. Если он его ввел, то этот новый заголовок мы присваиваем свойству `Text` выделенного в данный момент узла.

Теперь напишем код для кнопки "Сортировать". Её код совсем простой:

```
procedure TfMain.bSortClick(Sender: TObject);
begin
  TreeView1.AlphaSort;
end;
```

Метод `AlphaSort` возвращает истину, если сортировка прошла успешно. Но нам нет смысла проверять эту успешность, поэтому сам метод мы вызываем, а на возвращаемое им значение не обращаем внимания.

Далее на очереди у нас кнопка "Свернуть список". Её код не сложнее:

```
procedure TfMain.bCollapseClick(Sender: TObject);
begin
  TreeView1.FullCollapse;
end;
```

Для "Развернуть список":

```
procedure TfMain.bExpandClick(Sender: TObject);
begin
```

```
  TreeView1.FullExpand;
end;
```

Методы `FullCollapse` и `FullExpand` мы изучали выше.

Кнопки мы запрограммировали. Однако пока толку от нашей программы - ноль. Пользователь потратит время, заполняя список разделов и подразделов библиотеки, но стоит ему только выйти из программы, и вся эта работа потеряется. Нам нужно научить программу этот список сохранять в файл, и загружать его из файла. Файл списка назовем MyLibrary.dat. MyLibrary - потому, что так называется наша программа, dat - такое расширение традиционно имеют файлы с данными. Где лучше всего загружать этот список? Конечно, в событии `OnCreate` главной формы! Это событие возникает однажды, когда загружается программа, но перед ее отображением на экране. Если вам требуется сделать какую то подготовительную работу перед открытием вашей программы, то `OnCreate` для этого - самое место. Выделите форму fMain - это можно сделать, щелкнув по маленькому свободному участку правее компонента `TreeView1`, или выбрав fMain в верхней части Инспектора объектов. Затем перейдите на вкладку "События" Инспектора объектов, найдите и сгенерируйте событие `OnCreate`. Код будет следующим:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  i: integer;
begin
  //если файл существует, загрузим его:
  if FileExists('MyLibrary.dat') then
    TreeView1.LoadFromFile('MyLibrary.dat');
  //теперь пройдемся по списку, и каждому узлу присвоим
  //нужную пиктограмму:
  for i:= 0 to TreeView1.Items.Count-1 do
    if TreeView1.Items[i].Parent=nil then TreeView1.Items[i].ImageIndex:=0
    else TreeView1.Items[i].ImageIndex:=1;
end;
```

Здесь мы сначала с помощью функции `FileExists()` проверяем, есть ли вообще в текущей папке файл MyLibrary.dat? Функция вернет истину, если такой файл есть. В этом случае мы его загружаем в дерево `TreeView1`. Но дерево выйдет без пиктограмм, их еще нужно загрузить. Это мы делаем в цикле `for`, обходя все узлы дерева. Если узел родительский (`TreeView1.Items[i].Parent=nil`), мы присваиваем ему картинку с индексом 0, иначе - картинку с индексом 1.

Осталось научить программу сохранять список. Делать это лучше всего при выходе из программы, в событии `OnClose` главной формы. В этом событии обычно делают все завершающие действия - закрывают открытые ресурсы, сохраняют параметры программы, и т.п. Код события следующий:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction: TCloseAction
begin
  TreeView1.SaveToFile('MyLibrary.dat');
end;
```

Тут мы просто сохраняем список в файл. Сохраните проект, запустите его и попробуйте заполнить разделы и подразделы библиотеки. У меня получилось примерно так:

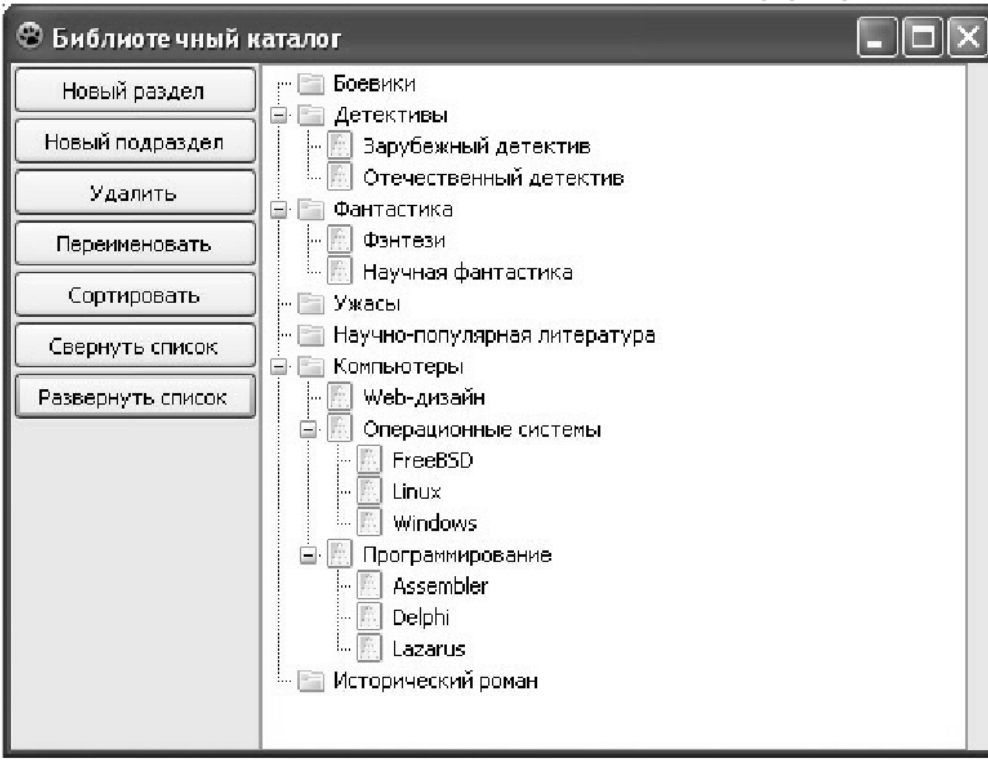


Рис. 19.4. Работающая программа

Поэкспериментируйте с работой всех кнопок. При выходе из программы список должен сохраняться, а при входе - считываться. Не удаляйте проект - мы к нему еще вернемся!

Многооконные приложения. SDI- и MDI-интерфейсы

На этой лекции мы рассмотрим создание многооконных приложений. На практике научимся создавать модальные и немодальные окна. Подробно изучим создание SDI-интерфейсов, немного коснемся принципов работы MDI-интерфейсов.

Цель лекции

Научиться создавать многооконные приложения, применять модальные и немодальные окна.

Многооконные приложения

До сих пор мы с вами все приложения делали с одним единственным окном. А между тем, в современном программировании редко встречаются программы, имеющие только одно окно. Даже простые стандартные утилиты, вроде Калькулятора calc.exe или игры "Сапер" - winmine.exe имеют по нескольку окон. Я недавно закончил проект для одной организации, проект этот можно считать средней сложности, а он содержит 102 окна! В этой лекции мы с вами научимся делать многооконные приложения.

Имеется два типа интерфейсов: SDI (Single Document Interface - однодокументный интерфейс) и MDI (Multi Document Interface - многодокументный интерфейс). SDI-приложения работают одновременно с одним документом, MDI-приложения предназначены для одновременной работы со множеством однотипных документов. При этом все документы располагаются внутри одного контейнера, которым служит, как правило, главная форма. Компания Microsoft не рекомендует использовать MDI-интерфейсы, хотя сама использует их в различных служебных программах, например, в консолях вроде Диспетчера устройств. Кроме того, разработка MDI-приложений в Lazarus пока не реализована, так что подробно рассматривать MDI-интерфейсы мы не будем, хотя вкратце и коснемся этой темы. Но вначале - SDI.

SDI

В SDI-приложениях окна могут быть двух видов - модальные и немодальные. Создаются они одинаково, разница заключается только в способе вывода этих окон на экран. Модальное окно блокирует программу, не даёт с ней работать, пока вы это окно не закроете. Стандартный пример модального окна - окно "О программе", которое присутствует почти в любом приложении. Как правило, такое окно находится в меню "Справка". Пока вы не нажмете "OK", закрыв это окно, вы не сможете работать с основной программой.

Немодальные окна позволяют переключаться между ними, и программой, и работать одновременно и там, и там. Типичный пример - окна Lazarus - вы можете переключаться между Инспектором объектов, Редактором кода, Редактором форм, и другими окнами - они не мешают друг другу, так как все они немодальные.

Изучим работу с различными окнами на примерах.

Модальные окна

Если вы помните, в лекции №18 мы делали лабораторную работу - Блокнот-шифровальщик. Там мы предусмотрели в меню "Справка" подменю "О программе", но само окно не делали. Пришло время исправить это упущение. Вы должны были сохранить проект в папку 18-01 под именем CodeBook. Убедитесь, что Lazarus закрыт, и загрузите файл CodeBook.lpi - это информационный файл проекта. В результате, загрузится Lazarus с этим проектом (с тем же успехом можно было бы загрузить файл CodeBook.lpr).

Выберите команду меню "Файл -> Создать форму" или нажмите одноименную кнопку на Панели инструментов. Появится новая форма с именем по умолчанию Form1. Мы с вами договаривались называть все формы понятными именами, и в начале имени ставить префикс `f`, что означает форму. Поэтому в свойстве `Name` этой формы напишите `fAbout`, затем нажмите кнопку "Сохранить все" (или выберите "Файл -> Сохранить все"), и модулю этого нового окна дайте имя `About`. Переключитесь клавишей <F12> в Редактор кода - вы увидите вкладки

модулей:

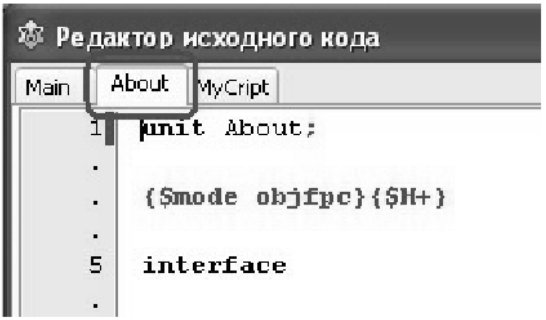


Рис. 20.1. Вкладки модулей в Редакторе кода

Два из этих модулей - `Main` и `About` имеют формы. Переходя по этим вкладкам можно переключаться между модулями. Но нам сначала нужно сделать само окно "О программе". Так что клавишей <F12> переключитесь обратно в Редактор форм. Прежде всего, в свойстве `BorderStyle` формы fAbout выберите значение `bsDialog`, так как нам не нужно, чтобы пользователь имел возможность изменять размеры окна, разворачивать или сворачивать его. Затем в свойстве `Position` выберите `poMainFormCenter`, чтобы окно появлялось по центру главного окна. До сих пор мы не устанавливали это значение у окон, так как все наши приложения содержали единственное, оно же главное окно. Окно "О программе" - не главное, поэтому его желательно выводить на экран по центру главного окна. Главным в проекте считается окно, созданное первым, его мы обычно называем fMain.

В свойстве `Caption` формы напишите "О программе".

Установите на форму простую панель TPanel, очистите ее свойство `Caption`. Чтобы сделать из панели красивую рамку, установите в её свойстве `BevelInner` значение `bvLowered`.

Далее, на панель установите две метки `TLabel` и один `TMemo`. В `TMemo` мы будем выводить многострочный текст с пояснением о назначении программы. Поскольку нам не нужно, чтобы пользователь мог редактировать этот текст, свойство `ReadOnly` компонента `Memo1` установите в `True`. Текст в `Memo1` придется вводить встроенным Редактором через свойство `Lines`.

Ниже панели установите кнопку TBitBtn с вкладки Additional Палитры компонентов, в свойстве `Kind` кнопки выберите значение `bkOK`.

Для экономии места я не буду подробно расписывать, как вводить в метки текст, менять у компонентов шрифты и размеры - вы прекрасно должны уметь делать это сами. В результате у вас должна получиться примерно такая форма:

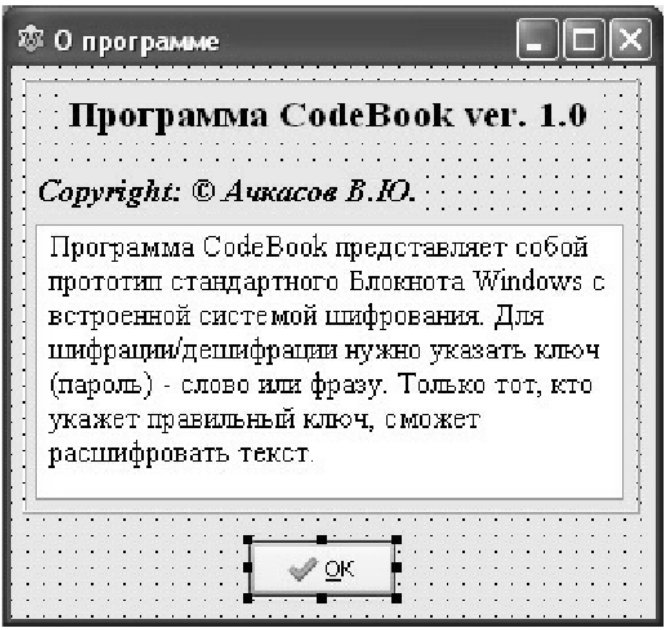


Рис. 20.2. Форма fAbout

Замечу, что во всех компонентах я выбирал свой любимый шрифт - Times New Roman - вы же можете выбрать свой, только подберите подходящие размер и начертание.

Отдельно остановлюсь на строчке `Copyright`. Слово Копирайт (англ. Copyright) означает авторское право. Причем авторское право может быть двух видов - имущественное и неимущественное. Если вы делаете программу на заказ, то имущественное авторское право принадлежит заказчику - он может устанавливать эту программу на сколько угодно компьютеров, продавать или дарить ее. Неимущественное право в любом случае принадлежит автору программы, то есть, вам. Оно подразумевает, что программу нельзя переименовывать или изменять её

код без вашего согласия, и что в программе вы обязательно должны упоминаться, как автор. Таким образом, если вы делаете программу на заказ, вы не обязаны вместе с программой отдавать исходный код вашего проекта! Иначе получится, что вы передаете заказчику не только имущественное, но и неимущественное право, а это уже будет цена продукта на порядок выше.

Так вот, в строчке `Copyright` указывается имущественный правообладатель. Если вы делаете программу на заказ, здесь вы должны указать заказчика. Себя же вы можете упомянуть строчкой ниже, установив еще одну метку, и начав ее текст, как "Автор: ". Но поскольку в данном проекте заказчика у нас нет, то все авторские права принадлежат нам. Кстати, указывайте не мою, а свою фамилию, ведь я добровольно выкладываю этот код для общего использования, а проект по моим описаниям делали все же вы!

И еще. Знак авторского права © имеет в таблице символов код 0169. Чтобы вставить его в `Caption` метки, при вводе текста нажмите <Alt>, и удерживая его, наберите 0169. Затем отпустите <Alt>. Символ должен появиться в метке. Вместо этого знака иногда указывают упрощенный вариант: "`(c)`". Сделайте, как считаете нужным.

Окно мы сделали, нужно теперь научить программу выводить его по требованию пользователя. Кнопкой <F12> перейдите в Редактор кода, затем, щелкнув по вкладке Main, перейдите на модуль главной формы. Вот так, сходу, мы ещё не сможем вызвать форму fAbout, сначала нужно подключить её модуль к главной форме. В разделе uses главной формы, после модуля `MyCript` через запятую добавьте модуль новой формы `About`. Теперь мы сможем вызывать это окно!

Сгенерируйте событие `OnClick` для команды меню "Справка -> О программе" (если вы еще помните, для этого достаточно просто выбрать данную команду). Её код очень простой:

```
procedure TfMain.HelpAboutClick(Sender: TObject);
begin
  fAbout.ShowModal;
end;
```

Метод ShowModal, указанный в коде, вызывает на экран окно fAbout в модальном режиме. Пока окно не закроется, с программой работать будет нельзя. Как только оно закроется, управление передастся обратно в программу. Сохраните проект, запустите его на выполнение и убедитесь, что окно "О программе" вызывается по требованию пользователя и закрывается кнопкой "OK". Однако не спешите закрывать проект, он нам еще понадобится.

Немодальные окна

Как уже упоминалось, немодальные окна могут быть открыты одновременно с основной программой, и не мешают её работе. Чтобы продемонстрировать работу немодальных окон, снабдим наш проект Блокнота-шифровальщика еще одной полезной функцией - статистикой. Создадим еще одно маленькое окно, в котором будем выводить количество строк в тексте, количество слов и символов.

Создайте новую форму командой "Файл -> Создать форму" или одноименной кнопкой на панели инструментов Lazarus. Форму (свойство Name) назовите fStats, а ее модуль сохраните под именем Stats. Далее, в свойстве Caption формы напишите "Статистика". Свойство BorderStyle установим в bsDialog, чтобы пользователь не мог менять размеры формы, а Position - в poMainFormCenter, чтобы окно появлялось по центру главного окна.

Теперь нам понадобятся шесть меток TLabel. Первые три установите в левой части формы, одну под другой. Эти метки мы переименовывать не будем, так как нам не придется обращаться к ним программно. В первой напишите "Строк:", во второй "Слов:" и в третьей "Символов:". Не забывайте ставить двоеточие после этих слов. Затем в правой части, так же одну под другой, установим еще три метки, их уже переименуем. В свойстве Name этих меток напишите, соответственно, LinesCount, WordsCount и CharsCount, а в свойстве Caption всех трех меток установите "0" (ноль).

В центре нижней части формы с вкладки Additional Палитры компонентов установите кнопку TBitBtn, в свойстве Kind которой

выберите значение `bkOK`. В результате, у вас должна получиться примерно такая форма:

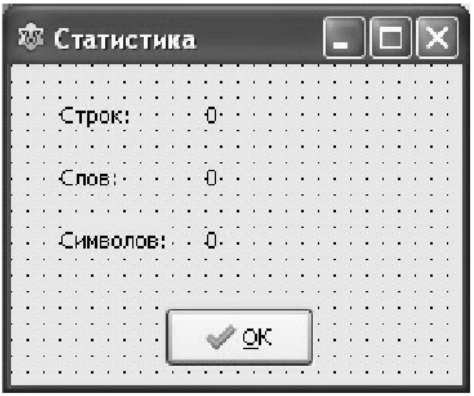


Рис. 20.3. Форма fStats

Сгенерируйте следующее событие `OnClick` для кнопки "OK":

```
procedure TfStats.BitBtn1Click(Sender: TObject);
begin
  Close;
end;
```

То есть, когда пользователь нажмет на эту кнопку, окно статистики закроется.

Далее перейдем в модуль главного окна fMain. Чтобы мы могли оттуда вызвать новое окно, нужно через запятую добавить его модуль `Stats` в конец раздела `uses`, как это мы делали с модулем `About` в прошлом примере. Сюда же добавьте еще два модуля: `LCLProc` и `StrUtils`. В первом реализованы UTF8-функции, одна из них нам понадобится для определения количества символов. Во втором реализовано множество полезных строковых функций, включая и ту, которая нам нужна для подсчета количества слов в тексте:

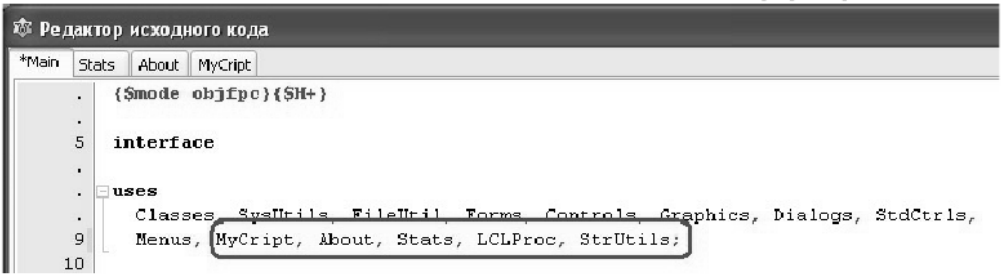


Рис. 20.4. Присоединенные модули

Теперь подумаем, каким образом мы будем вызывать окно статистики. Лучше всего поместить вызов статистики в раздел меню "Файл". Дважды щелкните по MainMenu1, чтобы вызвать редактор меню. В разделе меню "Файл" выделите подраздел "Сохранить как…", щелкните по нему правой кнопкой и выберите команду "Вставить новый пункт (после)". В свойстве `Name` нового подпункта введите `FileStat`, а в свойстве `Caption` напишите "`Статистика`":

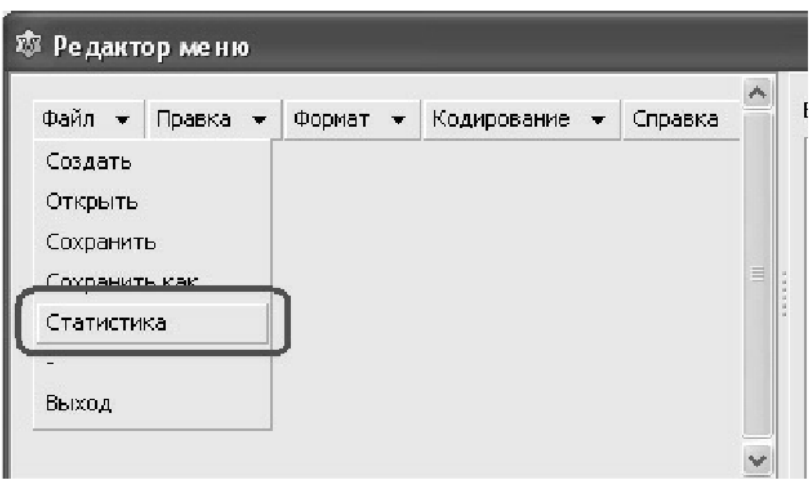


Рис. 20.5. Новый подпункт в меню

Теперь закройте Редактор меню; выбрав команду "Файл -> Статистика" сгенерируйте событие `OnClick` для нового подпункта. Код события следующий:

```
procedure TfMain.FileStatClick(Sender: TObject);
begin
```

```
  fStats.Show;
end;
```

Как видите, здесь мы вызываем форму fStats, но не методом `ShowModal`, как в прошлом примере, а методом `Show`. В результате окно будет показано, как немодальное. Однако этого недостаточно, чтобы окно показывало статистику. Нам нужно еще подсчитать и вывести на экран количество строк, слов и символов.

Выделите компонент `Memo1`. В Инспекторе объектов перейдите на вкладку "События" и сгенерируйте для него событие `OnChange`, дважды щелкнув по нему. Это событие возникает всякий раз при изменении текста в Memo1, тут мы и будем считать статистику. Код события следующий:

```
procedure TfMain.Memo1Change(Sender: TObject);
begin
  //считаем символы:
  fStats.CharsCount.Caption:= IntToStr(UTF8Length(Memo1.Text));
  //слова:
  fStats.WordsCount.Caption:= IntToStr(WordCount(Memo1.Text, StdWordDel
  //строки:
  fStats.LinesCount.Caption:= IntToStr(Memo1.Lines.Count);
end;
```

Тут для нас много нового, так что разберем код подробней. Вначале функцией `UTF8Length` (см. лекцию №6) мы получили количество символов в тексте `Memo1`, включая служебные символы перехода на новую строку. Это - целое число, которое нам пришлось преобразовать в строковую форму с помощью функции `IntToStr`, чтобы мы могли присвоить эту строку свойству `Caption` метки `CharsCount`. Обратите внимание, что метка находится не на этой, а на другой форме, поэтому нам пришлось сначала указать имя этой формы, затем имя метки и уж потом свойство `Caption`.

Слова в тексте считать сложнее, для этого мы воспользовались новой для нас функцией `WordCount`. Функция описана в модуле `StrUtils`,

который мы подключили, она возвращает количество слов в указанном тексте, и имеет следующий синтаксис:

```
function WordCount(const S: String; const WordDelims:TSysCharSet):integer;
```

Константа `S` содержит текст, в котором подсчитываются слова, в нашем случае, это `Memo1.Text`. Константа `WordDelims` содержит список символов-разделителей, которыми может отделяться одно слово от другого. Проще всего в этом параметре воспользоваться системной константой `StdWordDelims`, которая уже описана в том же модуле, и объявлена следующим образом:

```
const StdWordDelims = [#0..' ', ',', '.', ';', '/', '\', ':', '"', '''', '`'] + Brackets;
```

То есть, в этой константе перечислены основные символы-разделители слов. Функция возвращает целое число - количество слов в указанном тексте. Преобразовав его в символьную форму, мы присваиваем это значение свойству `Caption` метки `WordsCount`.

А вот количество строк даже считать не нужно, оно содержится в свойстве `Memo1.Lines.Count`. Нам остается лишь преобразовать его в символьную форму, и присвоить соответствующей метке. Теперь каждый раз, как в `Memo1` изменится текст (пользователь ввел символ, скопировал текст в `Memo` или удалил текст), это событие будет пересчитывать количество слов, символов и строк, и выводить их в форму fStats, даже если ее не видно. Выбрав команду меню "Файл -> Статистика", пользователь сможет вывести окно статистики. Более того, он может, не закрывая этого окна, вернуться в главную форму и продолжать набирать текст. Окно статистики при этом будет параллельно отображать результаты:

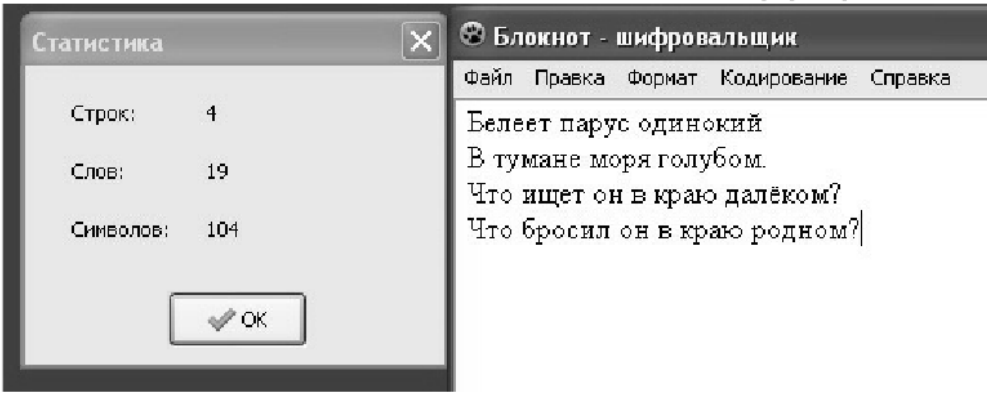


Рис. 20.6. Немодальное окно статистики в действии

Сохраните проект, запустите его на выполнение и убедитесь, что окно статистики не мешает работе главного окна. Проект Блокнота-шифровальщика нам потребуется ещё в лекциях №№ 28 и 29, так что не удаляйте его.

MDI-приложения

В отличие от Delphi, в Lazarus пока не реализована возможность создания MDI-приложений, а поскольку MDI-интерфейсы считаются устаревшими, то возможно, она и не будет реализована. Но знать об этих интерфейсах нужно, поэтому вкратце коснемся этой темы, тем более, что я могу ошибаться, и в следующих версиях Lazarus разработчики эту возможность все же реализуют. Принцип создания MDI-приложений следующий:

1. Вначале вы создаете главное, оно же родительское окно. Это окно будет служить своеобразным контейнером для дочерних окон, поэтому основная, рабочая часть главной формы должна быть свободной. В свойстве `FormStyle` (стиль формы) родительского окна следует выбрать значение `fsMDIForm`.
2. Затем вы создаете дочернее окно. Во время работы программы дочернее окно будет создаваться внутри родительского, и не сможет покинуть его пределы. В свойстве `FormStyle` дочернего окна следует выбрать значение `fsMDIChild`.
3. Форму дочернего окна мы конструируем только один раз, но в

программе это окно можно вызывать сколько угодно много раз, открывая в нем разные документы. Все эти дочерние окна могут быть открыты одновременно.

4. MDI-интерфейс приложения не запрещает вам создавать также модальные и немодальные окна. Например, в MDI-приложении вы с таким же успехом и таким же образом можете создать окно "О программе". Если вы будете создавать отдельные модальные и немодальные окна, то в свойстве `FormStyle` этих форм следует оставить значение по умолчанию `fsNormal`.

Для реализации MDI-интерфейса сами разработчики рекомендуют установить дополнительный компонент `MultiDoc`, который реализует псевдоMDI-интерфейс. Рассматривать работу с нестандартными компонентами мы не будем, желающих отсылаю на сайт разработчиков по адресу: ссылка: http://wiki.freepascal.org/MultiDoc

Русскоязычной страницы этого компонента, к сожалению, нет, так что вам придется воспользоваться каким-нибудь переводчиком, например, Promt.

Консольные приложения и параметры программы

На этой лекции мы научимся создавать консольные приложения, вводить и выводить данные в консольных программах, устанавливать правильную кодировку и использовать параметры программы.

Цель лекции

Освоить работу с консольными приложениями и с параметрами программы.

Консольные приложения

На сегодняшний день о программировании на Lazarus существует очень мало книг на русском языке, и все они описывают работу в основном, с консольными приложениями, хотя особой надобности в консолях сейчас нет. Но и совсем обойти эту тему нельзя. Что же такое консольное приложение?

Консольным приложением называется программа, которая не имеет графического интерфейса - окон, и которая работает в текстовом режиме в черно-белой консоли. Команды в такой программе приходиться вводить с клавиатуры, результаты работы консольные приложения также выводят на экран в текстовом режиме.

Если вы пользуетесь операционной системой Windows, то нажмите на "Пуск" и выберите команду "Выполнить". В открывшемся окне наберите

```
cmd
```

и нажмите <Enter>. Откроется командная консоль, в которой команды нужно вводить в текстовом режиме:

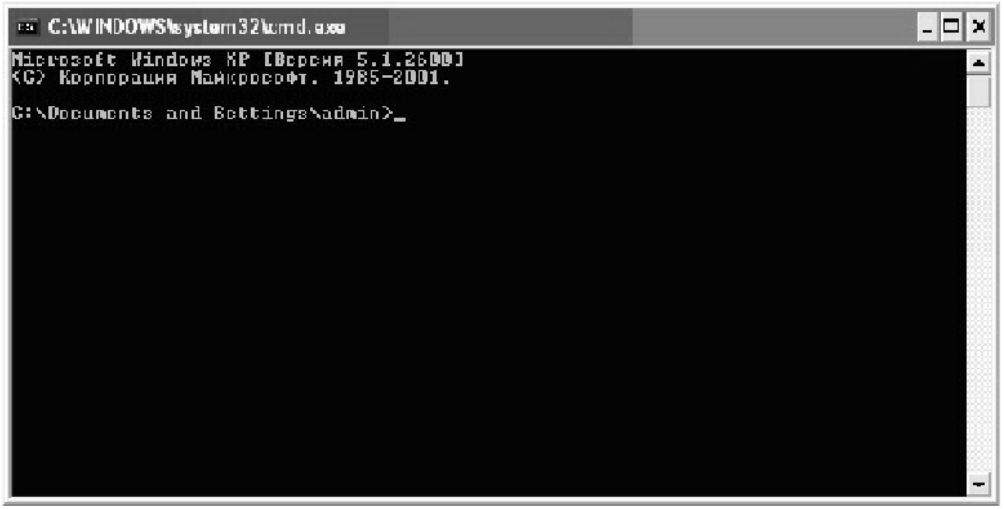


Рис. 21.1. Консоль cmd.exe

Собственно, вы видите окно предка Windows - операционной системы MS-DOS. Именно так выглядел экран с загруженной ОС MS-DOS, и управлять ею приходилось, вручную набивая всевозможные команды. Это уже позже корпорация Microsoft навесила на ядро MS-DOS всевозможные драйверы и утилиты, снабдила его графическим оконным интерфейсом, и появилась сначала полуграфическая ОС Windows 3.10 (русский вариант был 3.11 версии), а затем и целиком графическая Windows 95. Но к консольному ядру системы можно обращаться и в современных ОС, в Windows для этого используется программа cmd.exe, а в ОС Linux - терминал.

Все языки высокого уровня позволяют делать и консольные приложения, другое дело - зачем? Подавляющее большинство современных программ имеет графический интерфейс, который мы с вами создавали с самой первой лекции. А консольные приложения делаются в основном, системными программистами. Несмотря на то, что Object Pascal обладает для этого всеми необходимыми инструментами, системщики обычно пользуются такими языками, как Ассемблер, C, реже - C++. Но все же знать, как создаются консольные приложения нужно, поэтому данную лекцию мы посвящаем им.

Создание консольного приложения

Создать консольное приложение можно разными способами, но проще всего так. Откройте Lazarus. Командой "Проект -> Закрыть проект" закройте текущий проект, автоматически появится окно Мастера создания проекта. В нем нажмем кнопку "Новый проект". Появится окно создания проекта, в котором можно выбрать "Программа" или "Консольное приложение". Если мы выберем "Программа", то будет создан модуль с минимальным кодом. Если же мы выберем "Консольное приложение", то кода будет больше, так как при этом создается программа с новым классом, производным от `TCustomApplication`. `TCustomApplication` обеспечивает хорошую основу и делает программирование утилит командной строки довольно простым. Например, проверку опций командной строки, написание справки, проверку переменных окружения и обработку исключений. Все программы LCL автоматически это используют.

Но нам не нужно использовать возможности класса `TCustomApplication`, мы делаем простую консольную программу, поэтому в окне создания проекта мы выбираем "Программа". Сформируется проект, а в Редакторе исходного кода будет минимум текста:

```
program Project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

begin
end.
```

Обратите внимание, если нам нужно подключить к программе какие то модули, то делать это нужно до комментария

```
{ you can add units after this };
```

В этом случае мы ставим запятую после указанного модуля `Classes` и добавляем свои модули. Завершающая точка с запятой стоит после комментария, нам её ставить не нужно.

Свой код мы будем писать между скобками

```
begin
end.
```

А если нам потребуется указать глобальные константы, переменные или сделать объявления процедур и функций, то все это делается до `begin`.

Для примера мы создадим простое приложение, а по ходу дела, познакомимся с инструментами ввода-вывода информации в консольных приложениях.

WRITE и WRITELN

Процедура `Write` предназначена для вывода информации на экран. Она имеет следующий синтаксис:

```
procedure Write(var F:Text; Args:Arguments);
```

Действует процедура следующим образом. В скобках мы можем указать какой-то текст, вывести содержимое переменных. Например:

```
write('Всем привет!!!');
write('Переменная a = ', a, '; переменная b = ', b);
```

После вывода информации на экран курсор остается на той же строчке, в позиции, где он оказался после вывода последнего символа.

Процедура `Writeln` действует точно также, но после вывода последнего символа курсор переходит на начало следующей строки.

Для ознакомления с процедурами создайте новый проект "Программа". Модуль можно не переименовывать, просто сохраните его в папку 21-01. Полный текст модуля следующий:

```
program project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var a, b: Integer;
begin
  a:= 5;
  b:= 10;
  writeln('Всем привет!!!');
  writeln('Переменная a = ', a, '; переменная b = ', b);
  readln();
end.
```

Обратите внимание: раздел переменных `var` мы указали до служебного слова `begin`, то есть, переменные `a` и `b` в пределах модуля являются глобальными. Далее мы присвоили переменным значения, затем вывели на экран приветствие, а потом содержимое переменных. На процедуру `readln()` пока не обращайте внимания, она нужна только, чтобы программа не закрылась сразу же после вывода текста, а была на экране, пока мы не нажмем <Enter>. Сохраните проект и запустите его на выполнение. И сразу же мы видим недостаток: вместо русских букв выходит нечто, что в Интернете называют "кракозябры":

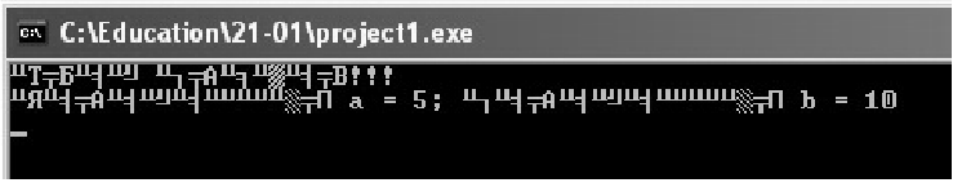


Рис. 21.2. Неправильный вывод кириллицы в консольной программе

В лекции №5 мы упоминали, что в консольных приложениях Windows принято использовать кодировку CP866, тогда как в графических приложениях используется CP1251 и Юникод. Отсюда и "кракозябры". В Linux и Unix таких проблем нет. Однако решить эту проблему совсем несложно. Нажмите <Enter>, чтобы закрыть консоль, и вернитесь в Редактор кода. Щелкните по окну с кодом правой кнопкой мыши, и в открывшемся контекстном меню выберите команду "Параметры файла -> Кодировка -> CP866". Откроется окно смены кодировки, нам надо нажать кнопку "Изменить файл". Снова запустите программу на выполнение - теперь с кодировкой полный порядок:

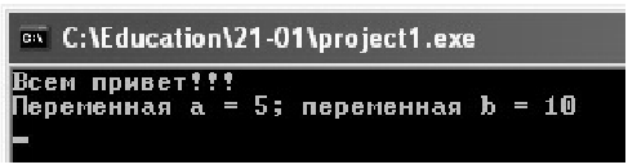


Рис. 21.3. Правильный вывод кириллицы в консольной программе

READ и READLN

Процедура `Read` предназначена для ввода информации в программу, точнее, в переменные. Процедура работает следующим образом:

```
read(Переменная1, Переменная2, …, ПеременнаяN);
```

То есть, в качестве параметров мы указываем одну или несколько разделенных запятыми переменных. Когда программа доходит до этого места, то ожидает от пользователя ввода данных. Пользователь должен ввести данные такого же типа, как у соответствующих переменных. Если переменных несколько, то соответствующие значения нужно

вводить, разделяя их пробелами. Например, если есть переменная целого типа `i`, и вещественная переменная `r`, то при выполнении инструкции

```
read(i, r);
```

произойдет вот что: программа приостановит свою работу, ожидая от пользователя ввода данных. Пользователь должен ввести целое и вещественное числа, разделенные пробелом, и нажать <Enter>. Например, пользователь ввел:

```
12 3.45
```

Как только он нажмет <Enter>, в переменную `i` попадет число 12, а в `r` - число 3,45. Если пользователь введет некорректные данные, не соответствующие типу переменной, то возникает ошибка и программа аварийно завершает работу.

Пользователь может ввести больше данных, чем указано переменных в процедуре `read`. В этом случае, оставшиеся данные будут обработаны следующей процедурой `read`. Например, пользователь ввел:

```
1 23 456
```

и нажал <Enter>. Эти данные могут быть обработаны следующими инструкциями:

```
read(a, b);
read(c);
```

Процедура `Readln` работает также. Отличие заключается в том, что если пользователь ввел больше данных, то `readln` не передает лишние данные в следующую инструкцию `readln`, а попросту обрубает их. То есть, если изменить предыдущий пример:

```
readln(a, b);
```

```
read(c);
```

то в `a` и `b` попадут числа 1 и 23, после чего программа будет ожидать следующих данных, а число 456 потеряется.

Если указать `read` или `readln` без параметров, например, так:

```
readln();
```

то программа просто перейдет в режим ожидания, пока пользователь не введет <Enter>. В конце консольных приложений всегда желательно указывать такую пустую `readln`, иначе программа просто мигнет и выключится, не дав пользователю увидеть результаты работы. Или можно как-то иначе обеспечить работу программы до тех пор, пока пользователь сам не захочет ее закрыть. Например, иногда программисты выводят сообщение типа "Продолжить работу программы? 0-нет, 1-да". Если пользователь введет 0, то программа завершится, а если 1, то продолжит работу. Какой способ использовать - дело ваше.

Чтобы пользователь понимал, чего от него хочет программа, перед каждой инструкцией `read` или `readln` следует с помощью `write` или `writeln` выводить поясняющий текст.

Вернемся к нашей программе, и немного изменим код (между `begin` и `end.`):

```
begin
  write('Введите первое целое число: ');
  readln(a);
  write('Введите второе целое число: ');
  readln(b);
  writeln('Результат:');
  writeln('Переменная a = ', a, '; переменная b = ', b);
  readln();
end.
```

Теперь программа выводит запрос "Введите первое число:", и замирает, ожидая от пользователя ввода данных. После того, как он введет число и нажмет <Enter>, программа таким же образом запросит второе число. Затем она выведет результат, и будет ожидать, пока пользователь не нажмет <Enter>:

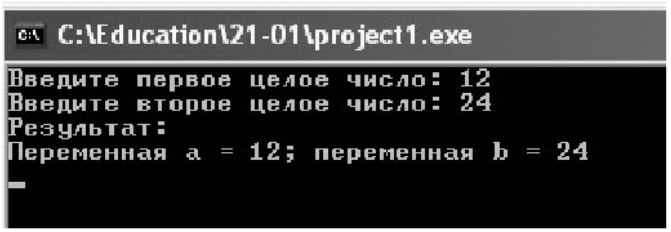


Рис. 21.4. Результат работы программы

Параметры консольного приложения

Нередко системные консольные программы вызываются с какими-то параметрами. Например, в Windows можно нажать "Пуск -> Выполнить" и указать следующую команду:

```
ping 127.0.0.1 -t
```

В результате будет запущена консольная утилита `ping.exe`, которая делает трассировку (обмен данными) с указанным IP-адресом. Таким образом, проверяют, есть ли связь с данным адресом. В примере программа запускается с двумя параметрами. Первым параметром является IP-адрес 127.0.0.1 - это адрес локальной машины, обращаясь к 127.0.0.1, вы обращаетесь к своему же компьютеру. Вторым параметром является

```
-t
```

Этот параметр указывает утилите, что не следует завершать обмен данными, пока пользователь сам не прервет работу программы. Если не указать `-t`, программа сделает 4 попытки отправить данные, после чего завершит свою работу. С параметром `-t` программа будет работать, пока

пользователь не нажмет <Ctrl+C>.

Возможно, когда-нибудь и нам потребуется создать программу (не обязательно консольную), умеющую принимать и обрабатывать параметры. Для обработки параметров используются две функции: `ParamCount` и `ParamStr`.

`ParamCount` возвращает количество параметров, переданных в программу.

`ParamStr(i)` возвращает параметр под индексом `i` - индексация начинается с 1. Например, вывести на экран первый и последний параметры можно так:

```
ShowMessage('Первый параметр: ' + ParamStr(1);
ShowMessage(Последний параметр: ' + ParamStr(ParamCount);
```

Однако имейте в виду, что в любой программе, консольной или обычной, даже запущенной совсем без всяких параметров, доступен и "нулевой" параметр `ParamStr(0)` - в нем всегда находятся адрес и имя файла программы.

Чтобы закрепить материал, давайте снова изменим нашу программу, получая и обрабатывая параметры. Мы сделаем так, что её можно будет использовать без параметров, тогда пользователь будет вводить числа прямо в программе. Можно будет загрузить программу с одним параметром, тогда это значение попадет в переменную `a`, а число для `b` пользователю придется вводить вручную. И, наконец, программу можно будет загружать с множеством параметров, при этом первые два попадут в `a` и `b`, остальные будут проигнорированы. Не забывайте, что тип данных должен соответствовать переменным, то есть один параметр, или первых два параметра должны быть обязательно целыми числами. Иначе программа даже не запустится, а просто выведет сообщение об ошибке.

Имейте в виду, что все параметры, даже числа, вводятся в виде текста, так что для работы с целыми числами нам придется использовать функции `IntToStr` и `StrToInt`. А обе эти функции описаны в

модуле SYSUTILS, который нам придется подключить в разделе uses. Вот полный листинг программы:

```
program project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var a, b: integer;
begin
  //если параметров вообще нет:
  if ParamCount = 0 then begin
    write('Введите первое целое число: ');
    readln(a);
    write('Введите второе целое число: ');
    readln(b);
  end
  //если только один параметр:
  else if ParamCount = 1 then begin
    a:= StrToInt(ParamStr(1));
    write('Введите целое число: ');
    readln(b);
  end
  //если параметров больше одного:
  else begin
    a:= StrToInt(ParamStr(1));
    b:= StrToInt(ParamStr(2));
  end;
  //теперь вывод результатов:
  writeln('Всего параметров: ' + IntToStr(ParamCount));
  writeln('Запущена программа: ' + ParamStr(0));
  writeln('Переменная a = ', a, '; переменная b = ', b);
  readln();
```

```
end.
```

Обратите внимание, после модуля `Classes` мы указали модуль `SysUtils`. Далее, мы обеспечили ввод данных в `a` и `b` в случаях, если параметров вообще нет, если есть только один параметр, и если параметров больше, чем один. В конце мы выводим общее количество указанных параметров, нулевой параметр с указанием адреса и файла программы, и затем указываем значения переменных. В заключение указываем пустой `readln`, чтобы программа не закрылась, пока не захочет пользователь.

Сохраните проект и запустите его. Он загрузится без параметров, вам придется ввести оба числа вручную. Введите числа, полюбуйтесь на результат и закройте программу. Теперь загрузим ее с параметрами. Для этого нажмите "Пуск -> Выполнить", нажмите кнопку "Обзор", найдите и откройте наш проект, допишите после его имени один или два параметра, например, так:

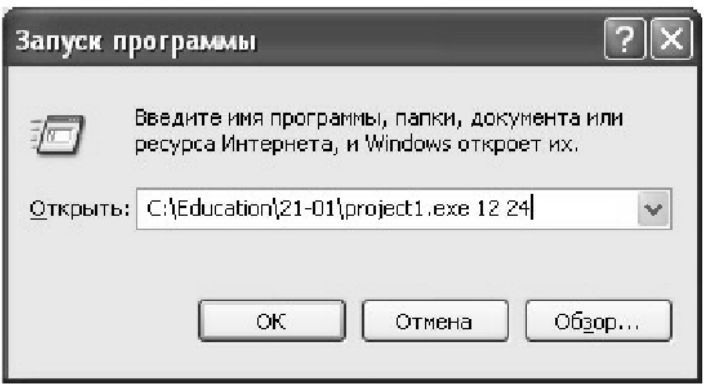


Рис. 21.5. Указание параметров программы

Как видите, в качестве параметров я указал два числа: 12 и 24. Программа не станет запрашивать у меня данные, а сразу выведет результат:

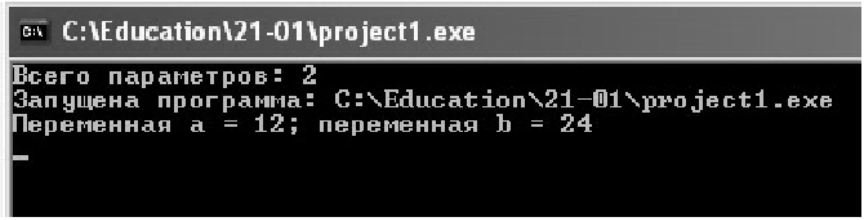


Рис. 21.6. Результат работы программы

Поэкспериментируйте с программой, вызывая с различным количеством параметров.

Как уже говорилось, функции `ParamCount` и `ParamStr` можно использовать в любой программе, не обязательно консольной.

Тип TStringList. Работа с папками

Эта лекция посвящена типу TStringList, который удобно использовать для работы с текстовыми файлами и списками строк. В лекции также рассматриваются инструменты для различной работы с папками - создание, удаление, смена текущей папки, проверка на существование.

Цель лекции

Научиться использовать тип `TStringList`, освоить различную работу с папками.

TStringList

До сих пор мы с вами работали с файлами только с помощью компонентов, например, `TMemo`. Однако для этого есть и другие способы, и самый простой - тип данных `TStringList`. `TStringList` является потомком класса `TStrings`, который нам уже встречался - свойство `Lines` у `TMemo`, и свойство `Items` у компонентов `TListBox` и `TComboBox`. Напрямую работать с типом `TStrings` нельзя, а вот с `TStringList` можно.

`TStringList` позволяет создавать текстовые файлы, записывать в них текст или наоборот, считывать его из файла, добавлять строки и совершать с текстом массу других полезных действий. В переменной типа `TStringList` данные хранятся в виде отдельных строк, и можно обратиться либо ко всему тексту целиком, либо к одной из его строк. Чтобы использовать этот тип данных, мало объявить переменную типа `TStringList`, нужно еще эту переменную проинициализировать (создать), а после работы с ней - уничтожить. Делается это следующим образом:

```
var
  sl: TStringList;  //объявили переменную
begin
  sl:= TStringList.Create;  //проинициализировали
  … ;  //какие-то действия с переменной
```

```
sl.Free;  //уничтожили
```

Когда мы только объявили переменную, компилятор еще не выделил под нее память. А вот когда мы вызвали метод `Create`, то в ОЗУ для переменной `sl` была выделена память. Сама переменная типа `TStringList` является указателем - она указывает на текст, хранящийся в памяти, а также количество занимаемых текстом байт. После инициализации переменной можно пользоваться: создавать с ее помощью файлы, обрабатывать текст и так далее. А вот когда мы вызываем метод `Free`, мы освобождаем память, занимаемую переменной, уничтожаем ее. После этого с переменной работать уже нельзя, попытка обращения к ней вызовет ошибку.

Попробуем использовать этот тип на практике. Откройте Lazarus с новым проектом. Сразу же сохраните его в папку 22-01, переименовывать форму и проект не будем, оставим имена по умолчанию.

В свойстве `Caption` формы напишите `Использование TStringList`. Добавьте на форму один компонент `TEdit` и четыре простых кнопки TButton. У `TEdit` очистите свойство `Text`, а в свойствах `Caption` кнопок напишите, соответственно:

Добавить строку в текст
Вывести текст на экран
Сохранить в файл
Читать из файла

В результате у вас должна получиться примерно такая форма:

Рис. 22.1. Форма программы

Чтобы с переменной можно было работать отовсюду, объявим её, как глобальную, над служебным словом `implementation`:

```
var
  Form1: TForm1;
  sl: TStringList;

implementation

{$R *.lfm}
```

Рис. 22.2. Объявление глобальной переменной

Поскольку переменная глобальная, то выделять под нее память нужно сразу, до нажатий на кнопки. В подобных случаях для подготовительных операций, таких как инициализация глобальных переменных, например, лучше всего подходит событие формы `OnCreate`. А для освобождения памяти перед закрытием программы лучше всего использовать свойство формы `OnClose`. Выделите форму, перейдите на вкладку События Инспектора объектов, и дважды щелкните по событию `OnCreate`, генерируя код для него. Код события очень простой:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //инициализируем переменную:
  sl:= TStringList.Create;
end;
```

Здесь мы просто выделили память для глобальной переменной, и сделали это сразу в начале работы программы, еще до показа формы на экране. Код события `OnClose` не сложнее:

```
procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseActio
begin
  //уничтожаем переменную:
  sl.Free;
end;
```

Таким образом, мы гарантировали выделение и освобождение памяти для переменной. Теперь с переменной `sl` можно будет работать в любой процедуре или функции программы. Код обработки первой кнопки будет таким:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  //если текста нет, сразу выходим:
  if Edit1.Text = '' then exit;
  //добавляем строку в конец текста:
  sl.Add(Edit1.Text);
  //очищаем Edit1:
  Edit1.Text:= '';
end;
```

Как видите, вначале мы сделали проверку - есть ли текст в строке `Edit1`? Если текста нет, то мы выходим из процедуры, ничего не делая. Если текст существует, то вначале мы добавляем его в переменную `sl`, а затем очищаем `Edit1` для нового текста.

Код обработки второй кнопки еще проще:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  ShowMessage(sl.Text); //выводим весь текст на экран
end;
```

Здесь нам не нужно делать проверку на существование текста - даже если текста нет, будет выведено пустое сообщение. Если текст есть, он будет выведен на экран. Код для третьей кнопки:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
  //сохраняем текст в файл:
  sl.SaveToFile('mytext.txt');
end;
```

Тут, я думаю, все понятно - таким же образом мы сохраняли текст в файл с помощью компонента `TMemo`. Если файла нет, он будет создан, если есть - перезаписан. Обратите внимание, мы указали просто имя файла. То есть, файл будет создан в текущей папке - папке с программой. Если вы хотите, чтобы файл сохранялся где-то в конкретном месте, то вместе с именем нужно указывать и адрес, например:

```
  sl.SaveToFile('C:/mytext.txt');
```

Код чтения из файла для последней кнопки чуть сложнее, поскольку нам еще нужно проверить существование файла:

```
procedure TForm1.Button4Click(Sender: TObject);
begin
  //читаем текст из файла, если он существует:
  if FileExists('mytext.txt') then
    sl.LoadFromFile('mytext.txt');
end;
```

Функция `FileExists` возвращает `True`, если указанный файл существует. В этом случае будет выполнено чтение из файла. Если бы такой проверки не было, то попытка обращения к несуществующему файлу привела бы к ошибке программы. Если при сохранении вы указывали имя файла вместе с его адресом, тут также придется указать адрес. Попробуйте работу программы, поэкспериментируйте с кнопками, убедитесь, что файл создается.

Однако, это еще не все. Фактически, переменная типа `TStringList` является массивом строк, из которых состоит текст. Как уже говорилось выше, обратиться можно не только ко всему тексту, но и к его конкретной строке, указав её индекс. Индексация строк начинается с нуля, свойство переменной `Count` содержит общее количество строк. Исправим код для кнопки "Вывести текст на экран" - выведем не весь текст, а только его последнюю строку:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  //если вообще текст есть:
  if sl.Count > 0 then
    //то выведем на экран последнюю строку:
    ShowMessage(sl[sl.Count - 1]);
end;
```

Как видите, теперь нам пришлось сделать проверку - а есть ли текст? Если текста нет, то попытка обращения к несуществующей строке вызовет ошибку программы. Поскольку индексация строк начинается с нуля, то индекс последней строки будет `Count` минус единица. Тут вы можете заметить разницу обращения к тексту. Весь текст содержится в:

```
sl.Text
```

А первая строка это:

```
sl[0]
```

Кроме `Count`, тип `TStringList` имеет и другие, известные по

TMemo инструменты-методы:

Add - добавить новую строку в конец текста, например:

```
sl.Add('Новая строка');
```

Clear - очистить текст, например:

```
sl.Clear;
```

Insert - вставить строку в конкретное место, указав индекс строки. Например, вставить вторую строку:

```
sl.Insert(1, 'Новая вторая строка');
```

Delete - удалить указанную строку. В метод передается индекс удаляемой строки. Например, удалить последнюю строку:

```
sl.Delete(sl.Count - 1);
```

Работа с папками

В Lazarus имеется несколько полезных функций и процедур для различной работы с папками (каталогами), которые вы можете применять в своих программах. Изучим их. При желании, можете создать простое приложение с единственной кнопкой TButton на форме. Сгенерируйте процедуру OnClick для неё, и в этой процедуре поочередно можете проверять рассмотренные далее примеры.

GetCurrentDir - функция возвращает адрес и имя текущей папки. Функция описана в модуле SysUtils, то есть, чтобы использовать функцию, нужно убедиться, что SysUtils подключен в разделе uses. Пример использования:

```
ShowMessage(GetCurrentDir);
```

`GetDir` - функция возвращает текущую папку на указанном в параметре диске. Имеется два параметра: номер диска (целое число) и имя текстовой переменной, куда будет записан результат работы функции. Имейте в виду, что нумерация дисков начинается с 1, это будет диск A:. Первый жесткий диск C: имеет номер 3. Если указать номер диска 0, то будет выводится информация о текущей папке текущего диска. В Linux и других Unix-подобных системах нумерация диска игнорируется. Функция описана в модулях `System` и `SysUtils`, для использования будет достаточно, если подключен один из них. Пример использования:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  GetDir(3, s);  //в переменную s получили текущую папку на диске C:
  ShowMessage('Текущая папка на диске C: - ' + s);
end;
```

`DirectoryExists` - функция проверяет, существует ли указанная в параметре папка. Если папка существует, функция возвращает `True`, если нет - `False`. Если адрес папки не указан, то делается проверка на существование папки внутри текущей папки. Функция описана в модуле `SysUtils`, её следует использовать, если программа будет обращаться к какой-то папке: копировать её или просто сделать текущей. Если папки нет - это вызовет ошибку программы, поэтому проверка на существование в таких случаях обязательна. Пример использования:

```
if not DirectoryExists('C:\Temp') then  ShowMessage('Папки "C:\Temp" не су
else ShowMessage('Папка"C:\Temp" существует!!!');
```

`ChDir` - процедура, делает текущей указанную в параметре папку. Процедура описана в модулях `System` и `SysUtils`, чтобы её использовать, достаточно подключить один из них. Пример использования:

```
ChDir('C:\Temp\111');  //изменили текущую папку
ShowMessage('Теперь текущая папка: ' + GetCurrentDir);
```

`SetCurrentDir` - функция, делает указанную в параметре папку текущей и возвращает `True` в случае успеха, или `False`, если произошла ошибка (например, указанной папки не существует). Функция описана в модуле `SysUtils`. Пример использования:

```
if SetCurrentDir('C:\Temp\1111') then  ShowMessage('Сменили текущую па
else ShowMessage('Ошибка! Сменить папку не удалось');
```

`CreateDir` - функция, создает указанную в параметре папку. В случае успеха возвращает `True`, в случае неудачи - `False`. Неудача может случиться, если отсутствуют права на создание папки, например, диск доступен только для чтения, или такая папка уже существует. Функция описана в модуле `SysUtils`. Следующий пример будет успешным при первом обращении (если конечно, папку вообще можно создать), и неуспешном при последующих, так как такая папка уже будет:

```
if CreateDir('C:\Temp\MyNewDir') then
  ShowMessage('Создали новую папку')
else ShowMessage('Ошибка! Создать папку не удалось');
```

`ForceDirectories` - функция, создает указанную папку и все недостающие папки. В случае успеха возвращает `True`, в случае неудачи - `False`. Функция описана в модуле `SysUtils`. В следующем примере на диске C: будет создана папка MyNewDir, а в ней - папка 123:

```
if ForceDirectories('C:\MyNewDir\123') then ShowMessage('Папки созданы
else ShowMessage('Ошибка создания папок');
```

`RemoveDir` - функция, удаляет указанную в параметре папку. Чтобы удаление было успешным, папка должна быть пустой. Если в параметре указан не абсолютный путь (вместе с адресом), функция пытается удалить указанную папку внутри текущей папки. В случае успеха

возвращает `True`, в случае неудачи - `False`. Функция описана в модуле `SysUtils`. Пример использования:

```
if RemoveDir('C:\MyNewDir\123') then ShowMessage('Папка удалена')
else ShowMessage('Ошибка удаления папки');
```

`RmDir` - процедура, удаляет указанную в параметре папку. В отличие от `RemoveDir`, не возвращает никаких значений и не может удалить текущую папку. При попытке удалить несуществующую папку вызывает ошибку программы, так что применять нужно только совместно с `DirectoryExists`, во избежание неприятных ошибок. А лучше всегда пользоваться `RemoveDir`. Пример использования:

```
RmDir('C:\MyNewDir\123');
```

Файлы

На этой лекции мы изучим работу с файлами: текстовыми, типизированными и нетипизированными. Весь материал подкреплен подробными примерами.

Цель лекции

Научится работать с различными видами файлов.

Файлы

Файл - это область внешней памяти ПК, которая имеет имя.

То есть, файл - это часть ПЗУ - Постоянного Запоминающего Устройства, которым может быть жесткий диск, дискета, флэш-карта, CD или DVD. В операционных системах существуют различные файловые структуры, или, как еще говорят, таблицы размещения файлов. Например, в Windows это может быть FAT32 или NTFS. В этих таблицах имена файлов соотносятся с реальным адресом на диске. На уровне машинных кодов, реальный адрес файла представляет собой целое число - номер байта, с которого начинается файл. Также в таблице содержится общее количество байтов, занятых файлом. Но при программировании на языках высокого уровня, таких, как Объектный Паскаль, мы можем пользоваться символическими именами файлов, которые автоматически преобразуются операционной системой в реальные адреса и размеры этих файлов.

Файлы бывают трех типов:

1. Текстовые файлы. Это файлы, предназначенные для работы с текстом.
2. Типизированные файлы. Такие файлы могут хранить массив записей какого-то одного, указанного типа. Это может быть очень удобно для создания файла с данными.
3. Нетипизированные файлы. Это файлы для побайтовой работы с данными любого типа. Могут использоваться для копирования или переноса файлов, тем более что отсутствие типа делает такие

файлы более универсальными, совместимыми с любым существующим типом. Кроме того, у нетипизированных файлов обмен данными между памятью и диском более скоростной, что важно при обработке файлов большого размера.

Чтобы открыть доступ к файлу, требуется создать файловую переменную одним из трех способов:

```
var
  f1: TextFile;  //текстовый файл
  f2: File of <тип>;  //типизированный файл
  f3: File;  //нетипизированный файл
```

Типизированный файл может работать с любыми типами Объектного Паскаля, кроме файлов. Это может быть целое или вещественное число, символ, строка или даже пользовательский тип данных, например, запись (о записях - в следующей лекции). Например, объявить типизированный файл, работающий с символами, можно так:

```
var
  MyF: File of Char;  //типизированный символьный файл
```

Однако для работы с файлами мало объявить переменную файлового типа, нужно еще эту переменную "привязать" к конкретному файлу. Делается это с помощью процедуры `AssignFile`, в которую передается два параметра - имя файловой переменной, к которой "привязывается" файл, и имя существующего файла, с которым мы хотим работать. Если имя файла указано без адреса, то процедура подразумевает, что файл находится в текущей папке. Если файла нет, программа вызовет ошибку, поэтому рекомендуется всегда перед связыванием файловой переменной с файлом вначале проверить - существует ли файл? Делается это с помощью функции `FileExists`, с которой нам уже доводилось работать, и которая возвращает `True`, если указанный файл существует, и `False` в противном случае. Вот пример связывания файловой переменной с файлом:

```
var
```

```
  f1: TextFile;  //текстовый файл
begin
  if FileExists('c:\01\myfile.txt') then begin
    AssignFile(f1, 'c:\01\myfile.txt'); //связали файловую переменную с файл
    ... //дальнейшая работа с файлом
  end; //if
```

После того, как мы связали файловую переменную с файлом, мы можем обращаться к ней, как к файлу. Мы можем открыть её, считать или записать информацию, закрыть. А после работы с файлом его обязательно нужно закрыть! Делается это процедурой `CloseFile`, в которую передается всего один параметр - имя файловой переменной, например:

```
CloseFile(f1);  //закрыли файл, с которым была связана файловая перемен
```

Во время работы с файлом может возникнуть исключительная ситуация. Внимание! Такой термин мы встречаем впервые, и он очень важен для понимания. Что такое исключительная ситуация? Это любая ошибка программы, которая может произойти во время работы. Например, вы открыли какой-то сетевой файл и начали работу с ним. А другой пользователь в это время взял, да удалил этот файл. Или вы пытаетесь работать с файлом, который содержит вирус. Как только вы обратились к нему, ваш антивирус вмешивается, и удаляет файл, или переносит его в карантин. Или же ваш файл "битый" - содержит ошибку, и не может прочитаться.

При попытке чтения из несуществующего файла, или записи в него, или при работе с "битым" файлом произойдет ошибка и возникнет исключительная ситуация. Если вы не обработаете эту ошибку, то ваша программа, скорее всего, намертво повиснет. Вам придется вызывать Диспетчер задач Windows, и снимать задачу - принудительно закрывать вашу программу. Поэтому в любой ситуации, когда имеется риск возникновения исключительной ситуации, программист ВСЕГДА должен ее обработать. Для этого существует блок `try-finally-end`:

```
try
```

```
  //блок кода, в котором может произойти ошибка
finally
  //код, который должен выполниться в любом случае, например, код за
end;
```

Между служебными словами `try-finally` вы вписываете потенциально опасный код. То есть, код, который может вызвать ошибку. А между `finally-end` указываете тот код, который должен быть выполнен в любом случае, даже при возникновении ошибки. Вот более правильный пример того, как нужно связывать файл с переменной:

```
var
  f1: TextFile;  //текстовый файл
begin
  try
    AssignFile(f1, 'c:\01\myfile.txt'); //связали файловую переменную с файл
    …;   //дальнейшая работа с файлом - чтение/запись
  finally
    CloseFile(f1);  //по окончании закрываем файл
  end;  //try
```

Как видите, здесь попытку связать файловую переменную с файлом, и дальнейшую работу с ним мы заключили в блок `try-finally`. Поскольку блок обрабатывает исключительные ситуации, можно не делать дополнительной проверки на существование файла. Получилась ли открытие файла, и работа с ним, или произошла ошибка, в любом случае будет выполнено закрытие файла, что убережет программу от краха.

Есть и другой способ обработки исключительных ситуаций - блок `try-except-end`:

```
try
  //блок кода, в котором может произойти ошибка
except
  //код, который выполнится в том случае, если в блоке между try-excep
```

```
end;
```

Пример работы с файлом можно представить и так:

```
var
  f1: TextFile;  //текстовый файл
begin
  try
    AssignFile(f1, 'c:\01\myfile.txt'); //связали файловую переменную с файл
    …;   //дальнейшая работа с файлом - чтение/запись
    CloseFile(f1);  //по окончании закрываем файл
  except
    ShowMessage('Внимание! Произошла ошибка открытия файла.');
  end;  //try
```

Каким образом обрабатывать исключительные ситуации - решать вам. В зависимости от ситуации, иногда бывает удобней первый способ, иногда - второй.

После связывания файловой переменной с файлом, этот файл нужно инициировать, то есть, указать направление передачи данных - из файла (чтение), в файл (запись), и в обоих направлениях. Если мы открываем файл для чтения, нам следует воспользоваться стандартной процедурой `Reset`:

```
Reset(f1);
```

При этом указатель устанавливается в начало файла, в нулевую позицию.

Если нам нужно создать новый файл, или перезаписать существующий, используется процедура `Rewrite`:

```
Rewrite(f2);
```

Если файл уже существовал, он будет перезаписан без каких либо

предупреждений. Если же нам нужно открыть для записи существующий файл, не удаляя содержащейся в нем информации, воспользуемся процедурой `Append`:

```
Append(f1);
```

Файл будет открыт для записи, а указатель переместится в конец файла. Однако, процедура `Append` применима только к текстовым файлам, то есть к переменным типа `TextFile`.

Во время работы с файлами можно использовать функции `Bof` и `Eof` - первая возвращает истину, если указатель находится в начале файла, вторая - если в конце. Обе функции имеют параметр - файловую переменную:

```
if Eof(f1) then ShowMessage('Достигнут конец файла.');
```

Есть еще один способ проверки существования файла, и вообще обработки ошибок ввода-вывода в файл. Это - функция `IOResult`, которая возвращает ноль при отсутствии ошибок ввода-вывода, и номер ошибки в противном случае. Но использовать эту функцию просто так не получится. Lazarus автоматически обрабатывает ошибки ввода-вывода. Прежде чем использовать `IOResult`, нужно дать процессору команду отключить автоматическую обработку этих ошибок. Для этого существует директива процессора `{$I-}`. Затем мы выполняем опасный участок кода, где может произойти ошибка, после чего снова включаем автоконтроль операций ввода-вывода директивой `{$I+}`. Затем вызываем `IOResult`. Если операция произошла успешно, то функция вернет ноль. Пример:

```
var
  f1: TextFile;
begin
  AssignFile(f1, 'MyText.txt');  //связали файл с переменной
  {$I-}  //отключили автоконтроль ввода-вывода
  Reset(f1);  //пытаемся открыть файл для чтения
  {$I+}  //снова включили автоконтроль ввода-вывода
```

```
if IOResult <> 0 then
  ShowMessage('Внимание! Произошла ошибка открытия файла.');
```

Текстовые файлы

Эти файлы связываются с переменной, имеющей тип `TextFile`, и предназначены для хранения и обработки текста. Фактически, в Паскале текст - это набор строк различной длины.

В начале файла устанавливается специальный системный признак `BOF` (Begin Of File) - начало файла. В конце каждой строки ставится специальный признак конца строки `EOLN` (End Of Line) - конец линии, строки. Это два системных, невидимых символа - перевод каретки и новая строка, мы их можем выразить, как символы `#13` и `#10`. В конце файла устанавливается признак `EOF` (End Of File) - конец файла.

Для доступа к записям применяются процедуры `Read` и `Readln` для чтения, `Write` и `Writeln` для записи. Мы уже знакомы с этими процедурами по лекции №21, где мы изучали консольные приложения, однако с текстовыми файлами эти процедуры работают несколько иначе. Во всех этих процедурах первым параметром идет файловая переменная, а вторым - список аргументов. Аргументами могут быть одна или несколько переменных строкового, символьного или числового типа. Для примера, создадим простое приложение, которое позволяет работать с текстовым файлом, занося в него построчно текст из строки `TEdit`.

Откройте Lazarus с новым проектом. В `Caption` формы напишите "`Работа с текстовым файлом`". Проект у нас пробный, так что имена формы, проекта и компонентов мы изменять не будем. Просто сохраните проект в папку 23-01. Установите на форму компонент `TEdit` и удлините его, чтобы пользователь мог ввести длинную строку. Очистите у компонента свойство `Text`. Ниже установите две кнопки, в `Caption` первой напишите "`Добавить в файл`", второй - "`Считать из файла`". Растяните их, чтобы текст на кнопках не казался прилипшим к краям. Еще ниже установите `TMemo`. Откройте редактор свойства `Lines` и удалите оттуда текст. Растяните компонент

по размеру `Edit1`.

В результате, у вас должна получиться примерно такая форма:

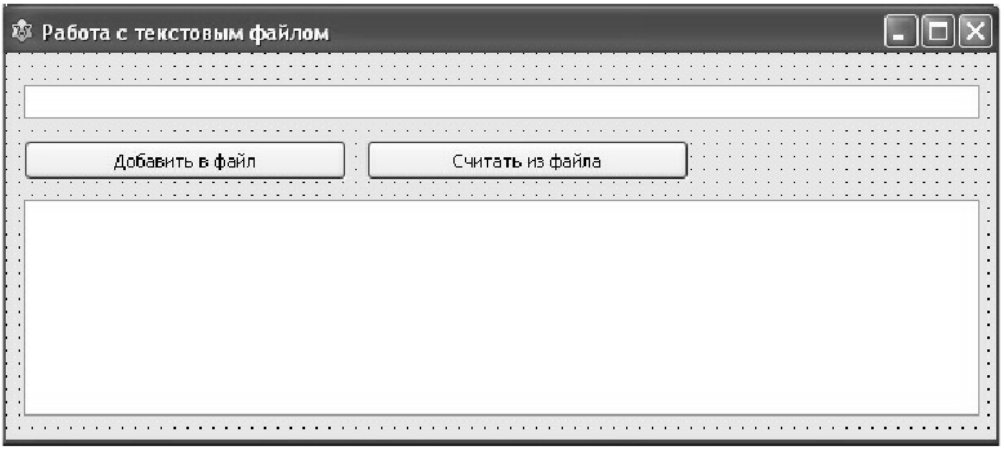


Рис. 23.1. Форма проекта

С подготовительной частью закончили, займемся программированием кнопок. Сгенерируйте событие `OnClick` для первой кнопки. Её код будет таким:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  tf: TextFile;  //текстовый файл
  s: String;
begin
  //если пользователь не ввел строку в Edit1, просто выйдем:
  if Edit1.Text = '' then exit;
  //иначе в s получаем текст:
  s:= Edit1.Text;
  //затем связываем файловую переменную tf с файлом mytext.txt:
  AssignFile(tf, 'mytext.txt');
  //далее может возникнуть исключительная ситуация, поэтому
  //заключим опасный код в блок try-finally-end:
  try
    //если файла нет (кнопку нажали первый раз), создадим его:
    if not FileExists('mytext.txt') then Rewrite(tf)
    //иначе откроем для записи, установив указатель в конец файла:
```

```
    else Append(tf);
    //тут просто записываем строку в файл:
    Writeln(tf, s);
    //очищаем Edit1, чтобы пользователь видел, что событие произошло:
    Edit1.Text:= '';
  finally
    CloseFile(tf); //закрываем файл
  end;
end;
```

Комментарии достаточно подробны, чтобы понять код. Замечу только пару вещей. Если пользователь ничего не ввел в `Edit1`, то нам нет нужды открывать файл и сохранять в него пустую строку, поэтому мы просто выходим из процедуры, ничего не делая:

```
  if Edit1.Text = '' then exit;
```

Когда мы открываем файл, мы вначале делаем проверку на его существование. Файла может не быть в двух случаях: пользователь нажал на кнопку первый раз, либо, поработав с файлом, и не закрывая программы, он удалил этот файл. В этом случае процедурой `Rewrite` мы создаем новый файл. Если же файл уже существует (кнопка нажималась, либо программа открывается не в первый раз, и файл остался с прошлых сеансов), то процедурой Append мы открываем его для записи, перемещая указатель в конец файла. Далее все просто - мы записываем строку текста в файл, очищаем `Edit1` от старого текста и закрываем файл. Код второй кнопки будет похожим:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  tf: TextFile;  //текстовый файл
  s: String;
begin
  //если файла нет, просто выходим:
  if not FileExists('mytext.txt') then exit;
  //иначе сначала очищаем Memo1:
  Memo1.Clear;
```

```
  //связываем файловую переменную tf с файлом mytext.txt:
  AssignFile(tf, 'mytext.txt');
  //далее может возникнуть исключительная ситуация, поэтому
  //заключим опасный код в блок try-finally-end:
  try
    Reset(tf); //открыли файл для чтения, указатель в начале файла
    //делаем, пока не конец файла:
    while not Eof(tf) do begin
      Readln(tf, s); //читаем в s очередную строку
      Memo1.Lines.Add(s);  //добавляем эту строку в Memo1
    end;
  finally
    CloseFile(tf); //закрываем файл
  end;
end;
```

По комментариям понятно, каким образом происходит чтение. Обратите внимание, для записи мы воспользовались процедурой `Writeln`, а не `Write`, а при чтении - `Readln`, а не `Read`. Это избавило нас от необходимости заботиться о вставке и поиске символов конца строки, рекомендую вам всегда использовать такой, более удобный способ. Весь алгоритм считывания строк мы поместили в блок `while-do` и считывали текст строка за строкой, пока функция `Eof` не вернула `True`, то есть, пока файл не закончился.

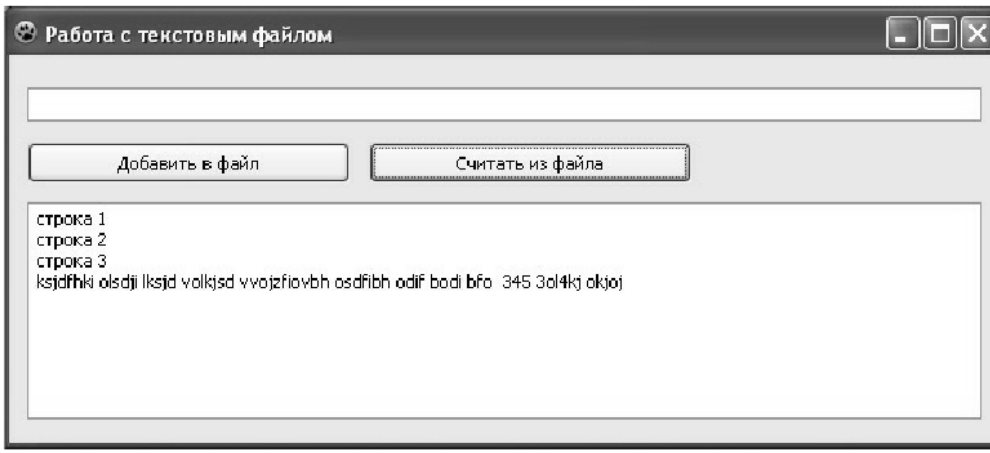


Рис. 23.2. Программа в действии

Типизированные файлы

Такие файлы предназначены для обработки информации какого-то определенного типа. Это может быть и символ, и число. Если это строка, то её нужно объявлять с определенной длиной, например, так:

```
var
  s: string[50];
```

Кроме того, файл может принять пользовательский тип данных, например, запись (записи см. в следующей лекции). Поскольку компилятор знает длину типа в байтах, то он может организовать доступ к любому элементу типизированного файла по его порядковому номеру. Как только файл открыт, указатель устанавливается в начало файла и указывает на первый элемент с номером 0. После чтения или записи указатель сдвигается к следующему элементу. Это было бы невозможно, если бы компилятор заранее не знал размер каждого считываемого или записываемого элемента, поэтому и строка должна иметь строго определенный размер.

К типизированным файлам применимы следующие функции и процедуры:

`FilePos` - функция принимает параметр - файловую переменную, и возвращает позицию указателя, то есть, номер элемента, который будет обрабатываться при последующем чтении/записи. Пример:

```
FilePos(f1);
```

`FileSize` - функция, которая также принимает в качестве параметра файловую переменную, и возвращает количество элементов в файле. Пример:

```
FileSize(f2);
```

`Read` - процедура чтения данных из типизированного файла. В

качестве параметров указывается файловая переменная, и одна или несколько переменных с таким же типом данных, как у файла. В эти переменные будет считан один или несколько (по количеству переменных) элементов, при этом указатель будет сдвигаться к следующему элементу. Пример:

```
Read(f1, MyRealPerem);
```

`Write` - процедура записи в типизированный файл. Как и у `Read`, у `Write` есть параметры - файловая переменная, и одна или несколько переменных с таким же типом данных, как у файла. Из этой переменной (этих переменных) будут браться данные, которые будут записаны в файл. После каждой записи указатель перемещается. Пример:

```
Write(f1, MyRealPerem);
```

`Seek` - процедура смещения указателя. Процедуре нужно указать файловую переменную и номер элемента, к которому нужно переместить указатель. Примеры:

```
Seek(f, 0);  //указатель в начало файла
Seek(f, 1);  //указатель на второй элемент
Seek(f, FileSize(f));   //указатель в конец файла
```

Еще раз замечу, что такие процедуры, как `Readln`, `Writeln` и `Append` работают только с текстовыми файлами, в типизированных или нетипизированных файлах их использовать нельзя.

Изучим работу с типизированными файлами на простом примере.

Откройте Lazarus с новым проектом. В `Caption` формы напишите "`Типизированный файл`" и сохраните проект в папку 23-02, не меняя имен формы, модуля и проекта.

На форму установите компонент `TListBox`. Полученный `ListBox1` немного вытяните по высоте, чтобы в нем уместились 10 чисел. Справа установите одну под другой, две простых кнопки `TButton`, в

Caption которых напишите, соответственно, "Сформировать файл" и "Читать из файла". У вас должна получиться примерно такая форма:

Рис. 23.3. Форма проекта

Идея такая. При нажатии на кнопку "Сформировать файл" мы с помощью функции Random будем формировать 10 случайных чисел, от 0 до 1000, и записывать их в типизированный файл. Файл у нас будет иметь тип Integer, то есть, под каждый элемент в файле будет выделено 4 байта. Таким образом, будет сформировано 10 элементов по 4 байта, в результате получится файл размером 40 байт.

При нажатии на кнопку "Читать из файла" мы будем по одному считывать эти элементы, и добавлять их в ListBox1. Таким образом, мы убедимся, что запись в типизированный файл, и чтение из него происходят без ошибок. Код первой кнопки следующий:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  f: File of integer;  //файл типа integer
  i: integer;  //счетчик для цикла
begin
  AssignFile(f, 'mytypefile.dat');
  Rewrite(f);
  for i:= 1 to 10 do  //делаем 10 раз
    Write(f, Random(1000));  //записываем в файл случайное целое число
```

```
  CloseFile(f);
end;
```

Код во многом вам знакомый, чтобы вы смогли с ним разобраться без надоедливых комментариев. Код второй кнопки:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  f: File of integer;
  k: integer;  //для получения значений элементов
begin
  //выходим из процедуры, если файла нет:
  if not FileExists('mytypefile.dat') then exit;
  ListBox1.Clear;
  AssignFile(f, 'mytypefile.dat');
  //Открываем для чтения:
  Reset(f);
  //делаем от первого до последнего элемента:
  while not eof(f) do begin
    Read(f, k); //считываем очередной элемент в перем. k
    ListBox1.Items.Add(IntToStr(k));  //записываем это значение в ListBox
  end;
  CloseFile(f);
end;
```

Код, я думаю, также не нуждается в дополнительных комментариях. Однако все же кое-что замечу. Поскольку файл типизированный, информация в него будет записана не в текстовом, а в двоичном виде, поэтому, открыв полученный файл простым текстовым редактором, вы не увидите там тех же чисел, что были у вас в `ListBox1`. Собственно, это даже хорошо - простой текстовый файл с данными пользователь может, ради интереса, и испортить неразумной попыткой редактирования, а вот в двоичный файл он лезть побоится.

Нетипизированные файлы

Такие файлы ассоциируются с переменными типа `File`, и могут

обрабатывать побайтово данные любого типа, поэтому их удобно использовать для копирования/переноса файлов. Нетипизированные файлы не поддерживают функции чтения-записи Read, Readln, Write и Writeln. Вместо них типизированным файлам доступны более скоростные функции BlockRead и BlockWrite. Эти функции позволяют производить чтение-запись блоками, используя для этого переменную-буфер.

Открывать файл для чтения или записи нужно функциями Reset и Rewrite, но тут тоже есть отличие: помимо файловой переменной этим функциям можно указывать длину записи в байтах. Если длина записи не указана, принимается длина по умолчанию 128 байт. Однако обычно указывают длину 1 байт, что позволяет обрабатывать любые, даже самые маленькие файлы, например:

```
Reset(f1, 1);
Rewrite(f2, 1);
```

Кроме этих различий, нетипизированным файлам доступны все остальные процедуры и функции, что и типизированным. Изучим работу с нетипизированными файлами на примере, для этого создадим программу, которая выполняет копирование файла.

Откройте Lazarus с новым проектом. Имя формы изменим на fMain (это будет полезная программа, так что имена будем давать осмысленные), в Caption напишем "Копирование файла", BorderStyle установим в bsDialog, а Position - в poDesktopCenter. Сохраним проект в папку 23-03 под именем FileCopier, а модулю главной формы дадим имя Main.

Установите на форму метку TLabel, в Caption которой напишите "Выберите файл, который нужно копировать:". Теперь перейдите на вкладку Misc Палитры компонентов, найдите на ней и установите на форму компонент TFileNameEdit - это гибрид TEdit и диалога открытия файла, он облегчит нам взаимодействие с пользователем. Нам придется обращаться к компоненту в коде, сократите его имя (свойство Name) до FNE.

Ниже установите еще одну метку, в `Caption` которой напишите `"Укажите папку, куда копировать файл:"`. Еще ниже установите компонент `TDirectoryEdit` с той же вкладки Misc - этот компонент облегчит нам выбор папки, куда будем делать копию. Сократите его имя до `DE`.

И напоследок, установите в нижней части формы простую кнопку TButton, в `Caption` которой напишите `"Выполнить копирование"`. Кнопка у нас одна, так что её можно не переименовывать.

Измените размеры формы и компонентов, чтобы у вас получилось примерно следующее:

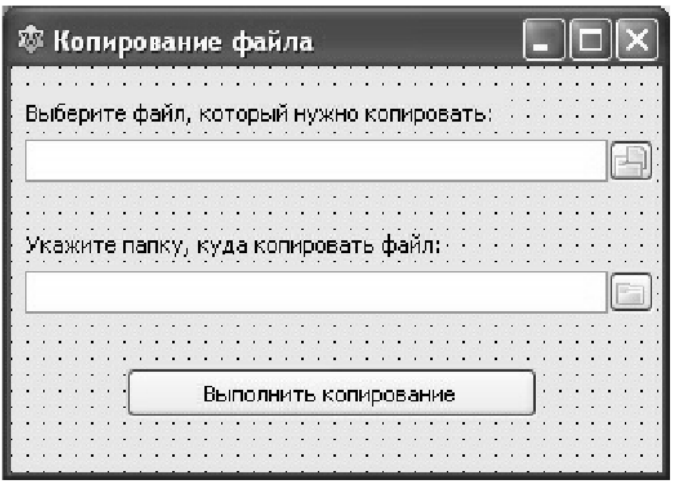


Рис. 23.4. Форма проекта

С подготовительной работой закончили, перейдем к кодированию. Весь код нам нужно будет делать в событии `OnClick` кнопки, сгенерируйте его. Вот этот код:

```
procedure TfMain.Button1Click(Sender: TObject);
var
  fIn, fOut : File; //нетипизированные файлы: исходник и копия
  NumRead, NumWritten : Word; //количество считанных и записанных ба
  Buf : Array[1..2048] of byte;  //буфер
begin
```

```
    //если файл не выбран, делаем сообщение и выходим:
    if FNE.Text = '' then begin
      ShowMessage('Внимание! Требуется указать или выбрать копируемый
      FNE.SetFocus;
      Exit;
    end;
    //если не выбрана папка, делаем сообщение и выходим:
    if DE.Text = '' then begin
      ShowMessage('Внимание! Требуется указать или выбрать папку для ко
      DE.SetFocus;
      Exit;
    end;
    //начинаем работу с файлами:
    try
      AssignFile(fIn, UTF8ToSys(FNE.FileName));
      AssignFile(fOut,UTF8ToSys(DE.Directory + '\' + ExtractFileName(FNE.File
      Reset (fIn,1);
      Rewrite (fOut,1);
      //изменим курсор на песочные часы:
      Screen.Cursor:= crHourGlass;
      Repeat
        BlockRead (fIn,buf,Sizeof(buf),NumRead);
        BlockWrite (fOut,Buf,NumRead,NumWritten);
      Until (NumRead=0) or (NumWritten<>NumRead);
    finally
      closeFile(fIn);
      closeFile(fOut);
      //изменим курсор на обычный вид:
      Screen.Cursor:= crDefault;
      ShowMessage('Копирование завершено!');
    end;
  end;
```

В коде есть кое-что новое, давайте разберем его подробней. Вначале мы объявили переменные `fIn` и `fOut`, которые имеют тип `File`. Эти переменные мы ассоциируем с файлами: `fIn` - с исходным файлом, а `fOut` - с создаваемой копией.

Далее, мы создали переменные-счетчики `NumRead` и `NumWritten` типа `Word`. В эти переменные будет заноситься количество реально считанных и записанных байт.

Также мы создали буфер - массив, в который можно записать до 2048 байт. В принципе, массив можно сделать любого размера, хоть 1 байт, но цифру 2048 я взял из справочной системы самого Lazarus.

Далее идет проверка - указал ли пользователь нужные нам данные? Если он оставил пустой строку с файлом - оригиналом, или с папкой, куда нужно делать копию, мы выводим соответствующее сообщение, и методом `SetFocus` выделяем нужный компонент (тот, который пропустил пользователь), после чего выходим из процедуры.

Если пользователь указал все данные, то дальше начинается работа с файлами. Мы ассоциируем файлы с файловыми переменными:

```
AssignFile(fIn, UTF8ToSys(FNE.FileName));
AssignFile(fOut,UTF8ToSys(DE.Directory + '\' + ExtractFileName(FNE.FileN
```

Обратите внимание: в свойстве `FileName` компонента `TFileNameEdit` находится адрес и имя выбранного пользователем файла. В свойстве `Directory` компонента `TDirectoryEdit` находится адрес и имя выбранной пользователем папки. Но здесь мы дополнительно обработали эти строки функцией `UTF8ToSys`. В принципе, этого можно было и не делать, но тогда программа вызывала бы ошибку, если бы в адресе или имени файла (папки) были неанглийские символы. Функция `UTF8ToSys` преобразует символы UTF8 в системную кодировку, которая требуется процедуре `AssignFile`. Теперь имя файла и адрес может содержать хоть кириллицу, хоть китайские иероглифы - программа не вызовет ошибки. Рекомендую во избежание возможных ошибок всегда обрабатывать ассоциируемые файлы этой функцией. Функция `UTF8ToSys` описана в модуле `FileUtil`, который по умолчанию уже подключен к проекту в разделе `uses`.

Далее, мы открыли оригинал для чтения, и создали копию, указав, что за раз будем обрабатывать по одному байту:

```
Reset (fIn,1);
Rewrite (fOut,1);
```

А дальше идет интересный код, который нам еще не встречался:

```
Screen.Cursor:= crHourGlass;
```

В любом графическом приложении есть системный объект Screen - экран. У этого объекта есть свойство `Cursor`, которое отвечает за внешний вид указателя мыши. Если присвоить этому свойству значение `crHourGlass`, то курсор примет вид песочных часов. Так делают, когда программа выполняет какое то длительное действие, и нужно показать пользователю, что компьютер не завис. Конечно, если файл небольшой, он скопируется мгновенно, вы даже глазом не успеете моргнуть. Но если файл большой, например фильм размером в полтора-два гигабайта, то копирование такого файла займет некоторое время. Поэтому мы и изменили внешний вид курсора. В самом конце строчкой

```
Screen.Cursor:= crDefault;
```

мы возвращаем курсору обычный вид. Если вам интересно, какие еще виды курсора бывают, выделите форму и посмотрите свойство `Cursor` - там довольно внушительный список возможных значений.

Далее мы выполняем чтение-запись:

```
Repeat
  BlockRead (fIn,buf,Sizeof(buf),NumRead);
  BlockWrite (fOut,Buf,NumRead,NumWritten);
Until (NumRead=0) or (NumWritten<>NumRead);
```

Цикл `Repeat…Until` будет выполняться, пока количество считанных байт не станет равным нулю, что означает, что файл закончен, либо пока количество записанных и считанных байт не станет отличаться (произошла какая то ошибка чтения-записи). Внутри цикла мы сначала читаем данные:

```
BlockRead (fIn,buf,Sizeof(buf),NumRead);
```

Как видите, процедура чтения `BlockRead` имеет четыре параметра:

1) Файловая переменная.

2) Буфер, куда будут считаны данные.

3) Количество читаемых байт. У нас указано `Sizeof(buf)`. Функция `SizeOf` возвращает размер переменной, то есть в данном случае это будет 2048 байт.

4) Переменная, куда будет записано количество реально прочитанных байт, ведь их может быть меньше, чем 2048.

Теперь о записи:

```
BlockWrite (fOut,Buf,NumRead,NumWritten);
```

Процедура копирования `BlockWrite` также имеет четыре параметра:

5) Файловая переменная.

6) Буфер, откуда будут браться данные для записи.

7) Переменная с количеством данных, которые требуется записать. Процедура `BlockRead` записала в переменную `NumRead` количество реально считанных байт, это количество мы и будем писать.

8) Переменная с количеством реально записанных байт.

Каждый раз, при чтении или записи, указатель устанавливается в новую позицию, так что нам специально его двигать не нужно. Предположим, что мы уже все байты считали. Тогда в следующий раз при попытке считать новую порцию в переменную `NumRead` попадет 0 байт, поскольку файл кончился, читать больше нечего.

После копирования мы закрываем файлы, делаем указатель мыши обычного вида и выводим сообщение, что файл скопирован. Сохраните проект, запустите его и попробуйте скопировать файл. Если вы все сделали правильно - файл скопируется без проблем. Перенос файла -

это то же самое копирование, только с последующим удалением оригинала.

Записи и вариант. Сетка строк TStringGrid

Данная лекция рассматривает работу с записями (record), типом данных variant и сектой строк TStringGrid. Весь материал подкреплен практическими примерами.

Цель лекции

Изучить типы данных `record` и `variant`, освоить работу с сеткой строк `TStringGrid`.

Запись - пользовательский тип данных

В Паскале предусмотрены не только стандартные типы, можно использовать и новые типы данных, определенные пользователем. Один из таких типов - запись.

Запись (анг. record) - это структура данных, определенная пользователем из различных стандартно-типовых полей.

По сути, запись является мини-объектом, который имеет свойства (поля), но не имеет методов и событий. Еще запись можно сравнить с массивом - в ней также хранятся различные данные, объединенные в одной переменной, только в отличие от массива, эти данные могут быть разнотипными.

Записи часто применяют для сохранения и загрузки различных данных, например, настроек программы. С помощью записей можно даже сделать небольшую базу данных, используя типовой файл, что мы и проделаем в следующей лабораторной работе.

Использование записей происходит в два этапа: вначале объявляется сама запись, затем объявляется переменная, которой присваивают тип этой записи. Кстати, тип записи можно присвоить не только переменной, но и массиву, и типизированному файлу.

Записи объявляются в разделе `type` перед разделом `var`, и могут иметь как локальную область видимости (запись объявлена внутри

подпрограммы), так и глобальную. Синтаксис объявления следующий:

```
type
  <имя_записи> = record
    <поле1>: <тип_поля>;
    <поле2>: <тип_поля>;
    …;
    <полеN>: <тип_поля>;
  end;
```

Здесь, `имя_записи` - имя, которым пользователь называет новый созданный тип данных. Поле - имя поля записи, фактически, это переменная внутри другой переменной. `Тип_поля` - один из стандартных типов. Как уже говорилось, объявлять тип `string` в записи не запрещается, однако использовать такую запись в типизированном файле не получится. Обусловлено это тем, что при создании типизированного файла, компилятор должен точно знать, сколько байт на каждую запись он должен отвести в файле. Строка `string` является динамическим символьным массивом, то есть, заранее узнать её длину невозможно. Однако, можно решить и эту проблему, если поставить ограничитель на строку, например, `string[50]` - строка из 50 символов. В этом случае, типизированный файл можно создать. И если строка будет пустая, или в ней до 50 не будет хватать символов, то часть этой записи в файле просто окажется пустой, хотя место под нее будет выделено. Правда, использование кириллицы в кодировке UTF-8 создает некоторые трудности, но и они решаемы.

После того, как запись объявлена, нужно создать переменную, присвоив ей тип этой записи. Делается это обычным образом, в локальном или глобальном разделе `var`:

```
var
  NewPerem: <имя_записи>;
```

После чего, можно обращаться к отдельному полю этой записи, разделяя его от имени переменной точкой, например:

```
NewPerem.Pole1:= 10;
```

Давайте освоим работу с записями на примере программы, которая сохраняет свои настройки, такие как положение и размеры формы, состояние некоторых компонентов, в типизированном файле, а при последующей загрузке программа считывает эти настройки и подстраивается под них.

Откройте Lazarus с новым проектом. Сразу договоримся, что поскольку проект очень простой и пробный, то имена формы, модуля, проекта и компонентов мы оставим по умолчанию. Сразу же сохраните проект в папку 24-01. Теперь, на форме нам потребуются:

- Метка `TLabel`;
- Строка `TEdit`;
- Кнопка `TButton`;
- Флажок `TCheckBox`.

В `Caption` метки напишите "`Заголовок формы:`". Очистите `Text` у `Edit1`. В `Caption` кнопки напишите "`Применить`". А с флажком можно ничего не делать. В результате у вас должна получиться примерно такая форма:

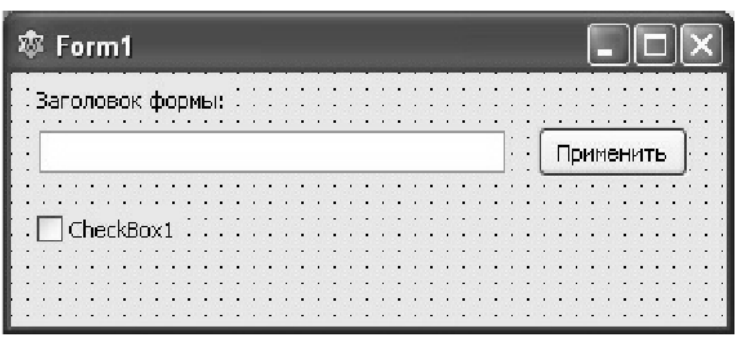


Рис. 24.1. Форма проекта

Теперь самое главное - нам нужно описать запись, в которую затем сохраним параметры формы. Для этого нужно создать раздел `type` ПЕРЕД глобальным разделом `var`, тем самым, в котором описана переменная формы:

```
var
  Form1: TForm1;
```

И не смущайтесь, что выше раздел `type` уже был, таких разделов может быть много. Код нашей вставки следующий:

```
type
  myForm = record  //запись
    Left: integer;  //левая граница формы
    Top: integer;  //верхняя
    Height: integer;  //высота формы
    Width: integer;  //ширина
    Caption: string[100];  //заголовок
    Checked: boolean;  //состояние флажка CheckBox1
    wsMax: boolean;  //окно развернуто?
  end; //record
```

Как видите, внутри раздела `type` мы описали новый, пользовательский (наш) тип данных - запись. Теперь нам нужна переменная этого типа, с которой мы будем работать. Нашей программе нужно сохранять свои настройки при выходе, делать это лучше в событии `OnClose` формы. Но ей также нужно и считывать эти настройки при загрузке, делать это будем в событии `OnCreate` формы. Поскольку переменная нам будет нужна и там, и там, то значит, опишем её в глобальном разделе `var`, сразу ПОСЛЕ описания переменной `Form1`:

```
var
  Form1: TForm1;
  MyF: myForm;  //наша переменная-запись
```

Теперь займемся кодированием. Прежде всего, нам нужно "научить" программу менять свой заголовок. Для этого сгенерируйте событие `OnClick` для кнопки "Применить", его код:

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

```
begin
  if Edit1.Text <> '' then Form1.Caption:= Edit1.Text;
end;
```

Тут всё просто: мы проверяем - не пуста ли строка `Edit1`, и если не пуста, то присваиваем её текст свойству `Caption` формы, тем самым, меняя заголовок окна. Пойдем далее. А далее нам нужно научить программу сохранять свои настройки. Для этого выделите форму, щелкнув по её любому свободному месту. Теперь перейдите на вкладку События Инспектора объектов, и двойным щелчком сгенерируйте событие формы `OnClose`, его код:

```
procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseActio
var
  f: file of myForm;  //типизированный файл для сохранения данных
begin
  //сначала настроим переменную-запись:
  //не сохраняем размеры и положение, если окно wsMaximized:
  if not (Form1.WindowState = wsMaximized) then begin
    MyF.Left:= Form1.Left;
    MyF.Top:= Form1.Top;
    MyF.Height:= Form1.Height;
    MyF.Width:= Form1.Width;
    MyF.wsMax:= False;
  end
  else MyF.wsMax:= True;
  //остальные настройки:
  MyF.Caption:= Form1.Caption;
  MyF.Checked:= CheckBox1.Checked;
  //теперь создаем или перезаписываем файл настроек:
  try
    AssignFile(f, 'MyProg.dat');
    Rewrite(f);
    Write(f, MyF); //записываем в файл данные из записи
  finally
    CloseFile(f);
  end;
end;
```

Давайте разберем этот код. В разделе `var` мы описали переменную `f`, которая будет ассоциироваться с типизированным файлом, имеющим тип нашей записи. Видите, мы присвоили наш тип и переменной, и типизированному файлу!

Далее нам требуется сохранить в переменную-запись настройки нашей формы, но тут есть одно НО: если окно программы было развернуто, то такие данные, как левая и верхняя граница окна, его высота и ширина становятся недействительными. Поэтому мы пошли на хитрость: если окно не было развернуто, то мы сохраняем эти параметры и, кроме того, полю `wsMax` присваиваем `False`, тем самым указывая, что окно не было развернуто. Если же окно было развернуто, то мы присваиваем этому полю `True`, не трогая границы окна:

```
if not (Form1.WindowState = wsMaximized) then begin
  MyF.Left:= Form1.Left;
  MyF.Top:= Form1.Top;
  MyF.Height:= Form1.Height;
  MyF.Width:= Form1.Width;
  MyF.wsMax:= False;
end
else MyF.wsMax:= True;
```

Как видно из кода, мы обращаемся к отдельным полям записи через точку:

```
MyF.Left
```

Остальные параметры (заголовок окна и состояние флажка) нам нужно сохранять в любом случае:

```
MyF.Caption:= Form1.Caption;
MyF.Checked:= CheckBox1.Checked;
```

Далее мы работаем с файлом, создавая или перезаписывая его. В файл

мы записываем одну-единственную запись - нашу переменную `MyF`, в которой уже хранятся все нужные настройки, после чего файл закрываем. Забегая вперед, скажу, что если вы попытаетесь открыть такой файл редактором текстов, то увидите в нем двоичный код:

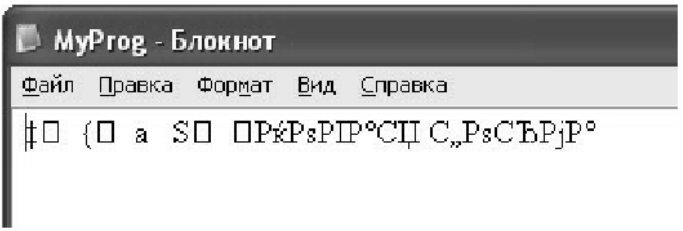


Рис. 24.2. Содержимое файла с настройками

Неискушенный пользователь не рискнет пытаться править такой файл, а если вы случайно чего-то в нем измените, и программа перестанет загружаться, то просто удалите файл - без него программа загрузится с настройками по умолчанию, а при следующем закрытии файл MyProg.dat будет создан снова.

Теперь нужно научить программу считывать настройки из файла, и применять их. Сгенерируйте событие формы OnCreate, его код:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  f: file of myForm;  //типизированный файл для сохранения данных
begin
  //если файла еще нет, просто выходим:
  if not FileExists('MyProg.dat') then exit;
  //иначе открываем файл и считываем из него настройки в запись:
  try
    AssignFile(f, 'MyProg.dat');
    Reset(f);
    Read(f, MyF); //считали данные в переменную типа запись
  finally
    CloseFile(f);
  end;
  //теперь настройки нужно применить к форме:
  //если окно было развернуто, разворачиваем его, иначе настраиваем кр
  if MyF.wsMax then Form1.WindowState:= wsMaximized
```

```
    else begin
      Form1.Left:= MyF.Left;
      Form1.Top:= MyF.Top;
      Form1.Height:= MyF.Height;
      Form1.Width:= MyF.Width;
    end;
    //остальные настройки:
    Form1.Caption:= MyF.Caption;
    CheckBox1.Checked:= MyF.Checked;
  end;
```

Вначале мы проверяем - есть ли файл. Если нет (программа открывается впервые), то мы выходим, ничего не считывая и не меняя настроек. В этом случае, все настройки будут, как при разработке формы.

Комментировать работу с файлом я не буду - всё должно быть предельно ясно по прошлой лекции и по предыдущему коду сохранения файла. В нашу переменную-запись `MyF` считались настройки, сохраненные в файле, и нам осталось только применить их к форме. Делается это как при сохранении, но в обратном направлении - не в запись из формы, а в форму из записи. Также, если окно было развернуто, разворачиваем его, иначе применяем настройки положения и размеров формы.

Теперь сохраните проект, запустите его на исполнение и попробуйте менять положение и размеры окна, разворачивать его - все эти настройки при выходе должны сохраняться, а при следующем запуске - восстанавливаться.

Вариант

В Lazarus существует еще один интересный тип данных - `variant`. Вариантный тип был разработан в Object Pascal для поддержки работы с механизмом OLE (англ. Object Link and Embedding - внедрение и связь объектов). Даже если вы впервые слышите про такой механизм, вы с ним наверняка знакомы, и не раз им пользовались. Каждый раз, когда вы копируете таблицу MS Excel в MS Word, или картинку из Paint в тот же MS Word, или текстовое поле из Word в Excel; другими словами,

любой объект, созданный в одной программе, копируете в другую программу - вы используете OLE.

В контексте Lazarus `variant` может содержать любое число - целое, или вещественное, строку, логическое значение, время-дату, массив, тот же OLE-объект. При этом компилятор сам определяет реальный тип варианта на этапе выполнения программы. Причем сразу же выделяет под переменную этого типа память - если значение не указано, этой переменной присваивается ноль.

Давайте рассмотрим следующий пример:

```
var
  MyV: variant;
begin
  MyV:=10;   //вариант содержит целое число
  MyV:= 2.5;   //вариант содержит уже вещественное число
  MyV:= 'Какой то текст';   //вариант содержит строку
  MyV:= True;   //а теперь вариант содержит логическое значение
```

Как видно из примера, мы присваиваем одной и той же вариантной переменной значения разных типов. При этом компилятор сам определяет наиболее подходящий по значению тип, и переделывает вариантную переменную под него. Например, в строке

```
  MyV:=10;
```

компилятор найдет наиболее подходящий под значение тип - `Byte`, и изменит переменную `MyV` под него. Таким образом, программист может не беспокоится о типах - за него эту работу выполнит компилятор. В вышеописанном примере переменная `MyV` поменяла свой тип 4 раза!

Однако за всё приходится платить. Переменная типа `variant` занимает гораздо больше памяти, чем переменная любого другого типа (за исключением строковой переменной, в которую можно поместить оба тома "Война и мир"). Поэтому, используйте тип `variant` только тогда, когда это действительно необходимо, например, когда на этапе программирования вы не можете заранее определить, данные какого

типа придется сохранять в переменную. В других случаях использование типа `variant` неоправданно.

С переменными типа `variant` можно производить различные действия, например, складывать. При этом различные типы данных будут преобразовываться под нужный тип, если это возможно. Давайте попробуем работу с переменной типа `variant` на практике. Откройте Lazarus с новым проектом. Не меняя имен по умолчанию, сразу сохраните проект в папку 24-02. На форму установите только одну кнопку и сгенерируйте для нее событие `OnClick`. Его код следующий:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  v1, v2, v3: variant;
begin
  v1:= '5';  //присвоили символ
  v2:= '10';  //теперь строку
  v3:= 20;  //теперь целое число
  v1:= v1 + v2 + v3;  //теперь все сложили

  ShowMessage(IntToStr(v1));
end;
```

Не торопитесь запускать эту программу, сначала попробуйте ответить: какое число в итоге выведет сообщение `ShowMessage`? На первый взгляд, результат будет `35 (5 + 10 + 20)`. Однако не спешите с выводами, подумайте. Мы к символу `'5'` прибавили строку `'10'`. Поскольку все однотипные операции производятся слева-направо, то в результате мы получим строку `'510'`. А затем мы к этой строке прибавляем целое число 20. Вариантная переменная, которая сначала имела символьный тип, затем строковый, получила целочисленный тип! А если к числу 510 прибавить 20, то сколько получится? Конечно, 530! Сохраните проект, запустите его на выполнение и убедитесь в этом сами.

Сетка строк TStringGrid

Напоследок познакомимся с еще одним полезным инструментом - сеткой строк `TStringGrid`. Это сетка, похожая на сетку в электронной таблице MS Excel, она позволяет вводить и редактировать табличную информацию в виде строк и колонок.

Загрузите Lazarus с новым проектом. Форму переименуйте в fMain, проект сохраните в папку 24-03 под именем Setka, а модулю формы дайте имя `Main`. Чтобы ваша форма соответствовала моей, установите ширину `width = 625`, а высоту `height = 195`. В `Caption` напишите "`Редактирование сетки TStringGrid`".

Теперь перейдите на вкладку Additional, найдите и установите на форму компонент `TStringGrid`. Рядом с ним вы можете увидеть еще одну сетку, `TDrawGrid`. Разница между ними в том, что сетка `TStringGrid` самостоятельно прорисовывает данные в ячейках, а в `TDrawGrid` это придется делать вручную. Так как `TStringGrid` удобней, её и будем изучать. У сетки установите ширину 615, а высоту 120 пикселей.

Ниже установите одну простую кнопку TButton, в `Caption` которой напишите "`Добавить`". В результате, у вас должна получиться вот такая форма:

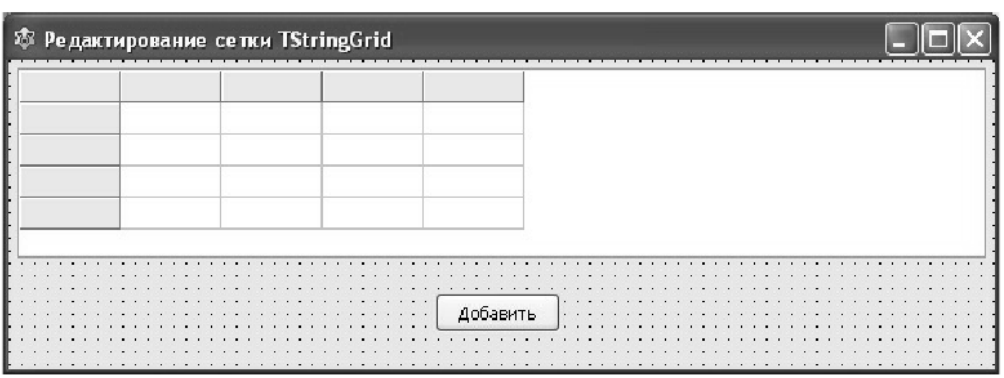


Рис. 24.3. Форма проекта

Прежде, чем приступить к кодированию, выделите на форме сетку и посмотрите в Инспекторе объектов на её свойства. Познакомимся с наиболее значимыми из них.

`BorderStyle`	- стиль обрамления. Может иметь только два значения - с обрамлением и без него.
`ColCount`	- количество колонок в сетке. По умолчанию их 5. При необходимости, это значение можно изменять программно, увеличивая или уменьшая количество колонок.
`DefaultColWidth`	- ширина колонок по умолчанию. К сожалению, всем колонкам устанавливается одинаковая ширина, хотя, как правило, разные колонки должны иметь разную ширину. Во время работы программы ширину каждой колонки можно будет изменить программно.
`DefaultDrawing`	- прорисовка данных по умолчанию. Если стоит `True`, то компонент сам будет отображать введенные данные, иначе это придется делать программисту.
`DefaultRowHeight`	- высота строк по умолчанию. Установлено 20 пикселей, но этот размер немного великоват, рекомендую изменить его на 18. Хотя это на любителя, и зависит от размера шрифта.
`FixedCols`	- количество фиксированных колонок. Они выделяются серым цветом, и всегда первые. Это свойство можно назвать заголовком строк. Практически не бывает необходимости делать более одной такой колонки, а иногда фиксированные колонки и вовсе не нужны. По умолчанию установлена одна колонка.
`FixedRows`	- количество фиксированных строк. По умолчанию тоже одна, и работает так же, как и `FixedCols`. Как правило, эта строка служит заголовком колонок.
`GridLineWidth`	- толщина разделительных линий. Попробуйте поставить ноль - линии исчезнут. Верните единицу.
	- самое главное свойство компонента. Оно

содержит много настроек, которые раскроются, если щелкнуть по плюсу слева от названия свойства. Эти настройки являются выключателями (`True`-разрешено, `False`-запрещено). Все настройки мы рассматривать не будем, их много, и большинство из них обычно не используется, кроме того, в справочнике Lazarus нет описаний этих настроек. Рассмотрим основные:

`goAlwaysShowEditor`	- если `True`, то редактировать ячейку можно сразу при выделении. Если `False`, то для редактирования нужно нажать <Enter> или <F2>.
`goColMoving`	- можно ли мышью перемещать колонки на другое место.
`goColSizing`	- разрешено изменять ширину колонки мышью.
`goFixedVertLine`	- рисовать ли вертикальные линии у фиксированных ячеек? По умолчанию `True`.
`goDrawFocusSelect`	- разрешено выделять ячейку, которая находится в фокусе ввода.
	- можно ли редактировать сетку? То есть,

Options	goEditing	вводить данные с клавиатуры. По умолчанию установлено False, то есть, редактировать нельзя. Игнорируется, если включен элемент goRowSelect. Если вы хотите, чтобы пользователь мог вводить в сетку данные, а обычно это нужно, этот параметр следует установить в True.
	goFixedVertLine	- прорисовка вертикальных линий у фиксированных ячеек.
	goHorzLine	- прорисовка горизонтальных линий у нефиксированных ячеек.
	goRangeSelect	- разрешение выделять несколько ячеек. Не работает, если включен элемент goEdit.
	goRowMoving	- можно ли мышью перемещать строки на другое место.
		- выделяется вся

	goRowSelect	строка. Если равно False, то только одна ячейка.
	goRowSizing	- разрешено изменять высоту строки мышью.
	goTabs	- можно ли переключаться на другие ячейки с помощью клавиши <Tab>.
	goThumbTracking	- разрешена ли прорисовка данных в ячейках при прокрутке. Если нет, то данные будут обновлены после прокрутки.
	goVertLine	- прорисовка вертикальных линий у всех остальных (нефиксированных) ячеек.
RowCount	- количество строк в сетке. По умолчанию их 5, как и колонок.	
Row	- закрытое свойство, содержит индекс текущей (выделенной) строки. Строка может быть выделена целиком, или в ней может быть выделена только одна ячейка. Поскольку свойство закрыто, то в Инспекторе объектов вы его не найдете, однако при кодировании, программно, это свойство доступно. Вы можете узнать индекс текущей строки, но не сможете его изменить с помощью свойства Row.	

Также сетка `TStringGrid` имеет пару полезных методов, с которыми тоже стоит познакомиться.

`DeleteRow` - метод удаляет строку, индекс которой указывается в параметре. Например:

```
StringGrid1.DeleteRow(StringGrid1.Row);  //удалить текущую строку
StringGrid1.DeleteRow(0);  //удалить первую
StringGrid1.DeleteRow(StringGrid1.RowCount-1);  //удалить последнюю
```

`SortColRow` - метод выполняет сортировку данных в сетке как по строкам, так и по колонкам. Имеет два параметра:

```
SortColRow(IsColumn: Boolean;  index: Integer);
```

Если `IsColumn` имеет значение `True`, то сортировка производится по колонке с индексом `index`. Если `False`, то по строке с индексом `index`. Как правило, данные требуется сортировать именно по колонкам, например, по колонке с фамилиями.

О чем еще следует знать? Данные сетки содержатся в отдельных ячейках. К ячейке можно обращаться по её индексу, как к элементу двухмерного массива, например:

```
s:=  StringGrid1.Cells[1, 1];  //считали  значение
StringGrid1.Cells[1, 1]:= 'Новое значение';  //записали значение
```

Как видно из примера, ячейки хранятся в списке ячеек `Cells`, при обращении к ячейке первым индексом является номер строки, вторым - колонки. Индексация начинается с нуля, поэтому самая верхняя левая ячейка будет иметь индекс `[0, 0]`. Если в сетке есть фиксированные строки и(или) колонки, именно они будут иметь нулевой индекс. То есть, если нужно указать в фиксированной строке заголовок третьей колонки, делать это надо так:

```
StringGrid1.Cells[2, 0]:= 'Заголовок';
```

Изменять ширину отдельных колонок нужно программно, используя свойство `ColWidth`:

```
ColWidth[<индекс_колонки>]
```

Например, так:

```
StringGrid1.ColWidths[0]:= 120;
```

Таким образом, можно будет сделать колонки разной ширины, чего нельзя добиться при проектировании формы, в Инспекторе объектов.

Еще в ячейках можно использовать маски ввода для строк и колонок. Для этого нужно сгенерировать событие сетки `OnGetEditMask`, в котором и прописать маску. Как это делается, будет видно из кода примера ниже.

Доделаем сетку. Высоту строк (свойство `DefaultRowHeight`) установите в 18 пикселей. Разверните свойство `Options` и установите в `True` параметр `goEditing`, чтобы пользователь мог редактировать данные сетки. Количество фиксированных строк и колонок оставим по умолчанию - 1. Это будут заголовки столбцов и строк. Их нужно заполнить, для этого выделите форму и сгенерируйте для неё событие `OnCreate`. Код события такой:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //заполняем заголовки строк:
  StringGrid1.Cells[0, 1]:= 'Иванов И.И.';
  StringGrid1.Cells[0, 2]:= 'Петров П.П.';
  StringGrid1.Cells[0, 3]:= 'Николаев Н.Н.';
  StringGrid1.Cells[0, 4]:= 'Бонд Дж.';
  //заполняем заголовки колонок:
  StringGrid1.Cells[1, 0]:= 'Год рождения';
  StringGrid1.Cells[2, 0]:= 'Место рождения';
  StringGrid1.Cells[3, 0]:= 'Прописка';
```

```
  StringGrid1.Cells[4, 0]:= 'Семейное положение';
  //меняем ширину колонок:
  StringGrid1.ColWidths[0]:= 120;
  StringGrid1.ColWidths[1]:= 90;
  StringGrid1.ColWidths[2]:= 150;
  StringGrid1.ColWidths[3]:= 100;
  StringGrid1.ColWidths[4]:= 120;
end;
```

Код, думаю, простой и понятный. Как только программа загрузится, сетка будет настроена. Теперь для колонки "Тод рождения" создадим маску ввода, чтобы пользователю оставалось только вписать цифры. Выделите сетку, в Инспекторе объектов перейдите на вкладку События и сгенерируйте событие `OnGetEditMask`. Код события следующий:

```
procedure TForm1.StringGrid1GetEditMask(Sender: TObject; ACol, ARow: In
  var Value: string);
begin
  if ACol=1 then Value:= '99.99.9999 г.';
end;
```

Как видите, в событии становятся доступными такие переменные, как `ACol` и `ARow`, в которых находится индекс соответственно, текущих колонки и строки. В переменную `Value` следует прописать текст маски, мы это делаем, только если текущей является вторая колонка (индексация с нуля):

```
  if ACol=1 then Value:= '99.99.9999 г.';
```

Теперь запрограммируем кнопку, дав пользователю возможность добавлять новые фамилии. Сгенерируйте для кнопки событие `OnClick`, код события следующий:

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

```
  if InputQuery('Укажите новое имя',
                'Впишите фамилию, имя и отчество в формате "Фамилия И.О.
                s) then begin
    StringGrid1.RowCount:= StringGrid1.RowCount + 1;
    StringGrid1.Cells[0, StringGrid1.RowCount-1]:= s;
  end;
end;
```

С помощью функции-запроса `InputQuerry` мы получаем в переменную s фамилию, имя и отчество. Затем добавляем еще одну строку:

```
StringGrid1.RowCount:= StringGrid1.RowCount + 1;
```

В заключение, в последнюю строку первого (фиксированного) столбца мы вписываем эти новые ФИО:

```
StringGrid1.Cells[0, StringGrid1.RowCount-1]:= s;
```

Причем делается это только в том случае, если пользователь их указал, иначе ничего не происходит. Что же касается редактирования остальных данных, то пользователь будет делать это прямо в программе, в ячейках сетки.

Вот и все. Сохраните проект, запустите его на выполнения и опробуйте сетку в деле:

Рис. 24.4. Приложение в работе

Телефонный справочник

Данная лабораторная работа закрепляет пройденный материал по темам работы с записями, типизированными файлами и сеткой строк TStringGrid. В процессе работы подробно рассматривается создание программы - телефонного справочника.

Цель лекции

Закрепление работы с записями, типизированными файлами и сеткой строк `TStringGrid`.

Постановка задачи

В данной лабораторной работе нам предстоит написать приложение - телефонный справочник. Это приложение должно обладать всеми необходимыми для нашей задачи качествами:

- выводить контактные данные в виде таблицы;
- сохранять контакты в файл и считывать их оттуда;
- добавлять новый или редактировать (изменять) существующий контакт;
- удалять выбранный из списка контакт;
- сортировать список контактов по алфавиту.

Для работы с контактами создадим тип данных запись, будем использовать переменную этого типа и типизированный файл, в котором данные будут храниться в двоичном виде. Сами контакты будем выводить в сетку `TStringGrid`, для добавления и редактирования записей создадим отдельное окно - сетка позволит только просматривать и выбирать контакты. Приступим?

Реализация

Для начала откройте Lazarus с новым проектом. Форму, как обычно, называем fMain, сохраняем проект в папку 25-01 под именем telephones, модулю главной формы присваиваем имя `Main`. Пойдем дальше. Скорее

всего, вы будете делиться этой программой с друзьями, поэтому имеет смысл сразу же отключить от проекта вставку отладочной информации. Если вы помните, нужно выбрать команду "Проект -> Параметры проекта", в разделе "Параметры компилятора" перейти на подраздел "Компоновка" и убрать флажок "Генерировать отладочную информацию для GDB". Это позволит нашей программе сразу "похудеть" - от 15 мегабайт готового продукта, до менее чем 2-х мегабайт.

Теперь займемся формой. В свойстве `Caption` напишите "`Телефонный справочник`", в `BorderStyle` выберите `bsDialog` (нам не нужно, чтобы пользователь менял размеры окна), а в свойстве `Position - poDesktopCenter`. Чтобы ваше приложение в точности соответствовало моему, выберите ширину формы `Width` 700 пикселей, а высоту `Height` 400 пикселей.

Теперь нам на форме потребуется простая панель инструментов. Установите обычную панель TPanel, очистите её свойство `Caption`, в свойстве `Align` выберите `alTop`, чтобы панель заняла весь верх формы, и в свойстве `Height` установите значение 27 - нам не нужна слишком высокая панель инструментов, узкая будет смотреться аккуратней.

На панель нам нужно установить четыре кнопки TSpeedButton с вкладки Additional Палитры компонентов. Кнопки TSpeedButton нам нужны, чтобы на них не переходил фокус ввода, в этом случае, всегда активной будет только сетка. Присвойте свойству `Name` кнопок соответственно, имена `bAdd`, `bEdit`, `bDel` и `bSort`. Первая "`b`" в именах означает, что это кнопка - button. В свойствах `Caption` этих кнопок напишите, соответственно, "`Добавить контакт`", "`Редактировать контакт`", "`Удалить контакт`" и "`Сортировать список`" (без кавычек, разумеется). Картинок на кнопки мы ставить не будем, обойдемся только пояснительным текстом. Свойству `Top` всех кнопок присвойте значение 2, чтобы они почти прижались к верхней границе панели, свойству `Width` присвойте 150, чтобы весь текст помещался на кнопках без проблем. Переместите их относительно друг друга так, чтобы от левого края панели и друг от друга их отделяло тоже примерно по 2 пикселя.

Теперь сетка. На свободное место формы, ниже панели, установите

сетку TStringGrid. Поскольку нам часто придется обращаться к ней в процессе кодирования, укоротим имя в свойстве Name до SG. В свойстве Align сетки выберите alClient, чтобы сетка заняла всё оставшееся место. Теперь нужно изменить некоторые другие настройки. По умолчанию, в сетке установлено по пять колонок ColCount, и по пять строк RowCount. Колонок нам нужно только три: имя контакта, его телефон и примечание, какой это телефон (мобильный, домашний, служебный, соседский). Поэтому уменьшим количество колонок ColCount до 3. А строка нам нужна и вовсе одна, она будет служить заголовком колонок, и она будет фиксированной. Уменьшайте RowCount до 1.

Теперь о фиксированных строках и колонках. Фиксированная строка FixedRows нам нужна только одна, это значение по умолчанию. А фиксированные столбцы FixedCols нам и вовсе не нужны, тут поставьте ноль.

Остальные параметры сетки можно оставить без изменений. Если вы всё сделали правильно, у вас в результате должна получиться такая форма:

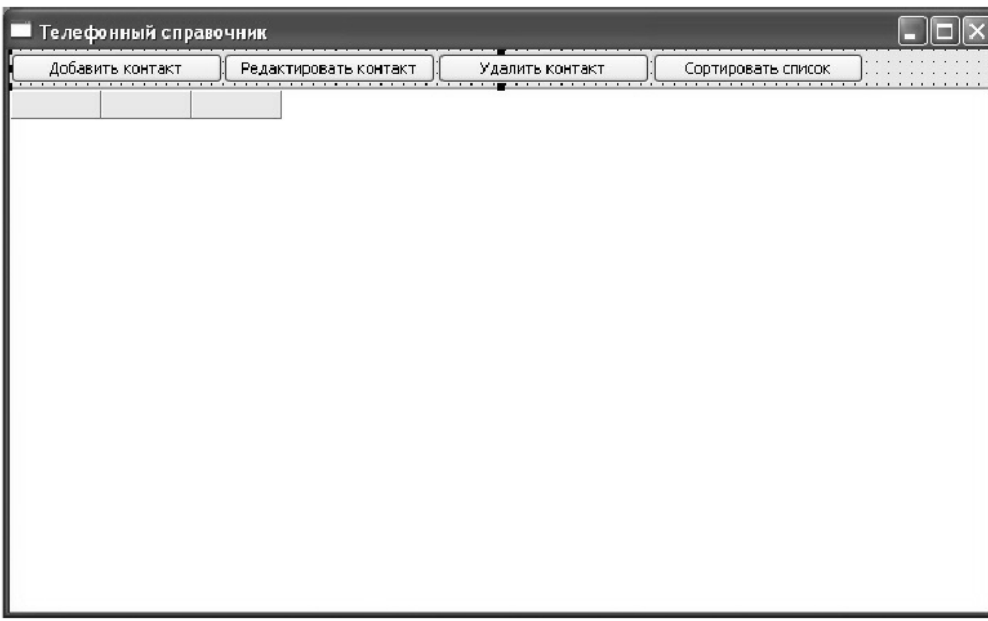


Рис. 25.1. Главная форма проекта

Пойдем дальше. Прежде, чем приступать к самому кодированию, давайте сделаем еще одну форму - окно редактора контактов. Командой Файл -> Создать форму создайте новую форму. Присвойте ей имя fEdit, нажмите кнопку "Сохранить все" и дайте модулю формы имя `Edit`.

В свойстве `Caption` напишите "`Редактор контакта`". Почему не "контактов"? Каждый раз мы будем загружать в это окно по одному контакту, поэтому и использовано единственное число. В свойстве `BorderStyle` также выберите `bsDialog`, в свойстве `Position` - `poMainFormCenter`, чтобы окно появлялось по центру главной формы, где бы она ни находилась. Размеры формы: ширина width = 400, высота height = 225 пикселей.

Теперь разместим на форме компоненты. Установите метку `TLabel`, в `Caption` которой напишите "`Укажите имя контакта:`".

Ниже установите строку `TEdit`. Переименуйте её в `eName`, ширину сделайте 380. Очистите свойство `Text`.

Ниже еще одну метку, с текстом в `Caption` "`Укажите телефон:`".

Ниже установите еще один `TEdit`. Переименуйте её в `eTelephone`, ширину сделайте 210. Очистите свойство `Text`.

Ниже еще одну метку, с текстом в `Caption` "`Выберите тип телефона:`".

Ниже нам потребуется список выбора `TComboBox`. Мы же не хотим, чтобы пользователю каждый раз самому приходилось писать "мобильный" или "служебный"? Переименуйте `TComboBox` в `CBNote`, откройте редактор свойства `Items` и там наберите следующие строки:

Мобильный
Домашний
Служебный
Соседский

Если сможете придумать другие типы, можете дописать их сюда.

В свойстве ItemIndex компонента установите 0, чтобы в списке появилась первая строка "Мобильный". Ширину сделайте 210.

Ещё ниже установите на одном уровне кнопки TBitBtn. Первую переименуйте в bSave, в свойстве Kind выберите bkOK, чтобы на кнопке появилась картинка - зеленая галочка, убедитесь, что свойство ModalResult кнопки перешло в mrOk. В свойстве Caption напишите "Сохранить контакт". Чтобы текст нормально умещался, увеличьте её ширину до 160 пикселей.

Вторую кнопку переименуйте в bCancel, в свойстве Kind выберите bkCancel, а в Caption напишите "Не сохранять". Свойство ModalResult кнопки должно перейти в mrCancel. В результате, у вас должна получиться вот такая форма:

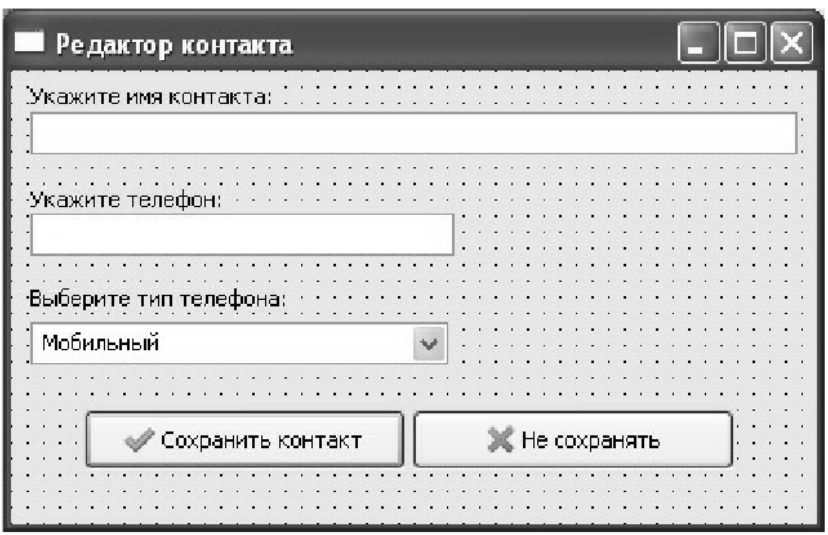


Рис. 25.2. Форма проекта fEdit

Чтобы более не возвращаться к этой форме, давайте сразу введем весь необходимый код формы, тем более что его совсем мало. Для кнопок нам вообще не нужно писать код. Когда мы изменили их свойства Kind, установив там bkOK и bkCancel, сразу изменилось и их свойство ModalResult, установившись, соответственно, в mrOK и mrCancel. И теперь нажатие на любую из этих кнопок приведет к закрытию формы, и к тому, что в ModalResult формы попадет

значение `ModalResult` нажатой кнопки. То есть, мы сможем узнать, какой кнопкой пользователь закрыл окно.

Но нам нужно все же сделать некоторую подготовительную работу. Прежде, чем показывать форму, нам нужно будет перевести фокус ввода в первый `TEdit`, ведь от прежней работы с формой выделенным мог остаться и другой компонент - кнопка или список выбора, например. А это будет неудобно для пользователя, ему придется лишний раз кликать мышью, выделяя первую строку.

Выделите форму, щелкнув по свободному участку, в Инспекторе объектов перейдите на вкладку События и сгенерируйте событие формы `OnShow` - это событие возникает всякий раз перед показом формы - идеальное место для подготовительной работы. Код события совсем простой:

```
procedure TfEdit.FormShow(Sender: TObject);
begin
  eName.SetFocus;
end;
```

И с редактором контактов это все! Вся остальная работа будет происходить в главной форме.

В редакторе форм перейдите на вкладку Main - наша главная форма. Прежде всего, в разделе `uses` главной формы, через запятую, добавьте модуль `edit` - это наш редактор контактов. Если мы не добавим его в раздел `uses`, то не сможем с ним работать из главной формы.

Далее, перед разделом глобальных переменных, тем, где объявлена переменная формы `fMain`, создадим еще один раздел `type`, в котором опишем запись:

```
type
  Contacts = record
    Name: string[100];
    Telephon: string[20];
    Note: string[20];
  end; //record
```

В этой записи три поля. Первое поле `Name` предназначено для имени контакта. Вы можете заметить, что оно ограничено размером 100 символов. Как уже говорилось в лекции № 24, в записи можно указывать неограниченный тип `string`, но тогда создать типизированный файл этого типа не получится. Поэтому у нас все строки имеют фиксированный размер. В поле `Name` пользователь будет вписывать имена контактов, например:

Анатолий Васильевич, директор
Дима, друг
Леночка, секретарь

Может показаться, что 100 символов для имени контакта - это перебор. Однако вспомните, что для текста у нас используется кодировка UTF-8, а в ней каждый символ кириллицы занимает 2 байта. То есть, если писать имена по-русски, реально можно вписать только 50 символов! А фамилии и имена бывают очень длинными, но 50 символов должно хватить. Для телефона мы указали 20 символов. Это обычно, цифры, скобки и знаки тире:

111-11-11
3-(222)-111-11-11

На каждый из этих символов требуется только 1 байт, так что 20 символов хватит за глаза для любого телефона.

Ну и, наконец, поле `Note`, в котором будет одно из значений:

Мобильный
Домашний
Служебный
Соседский

Эти символы тоже занимают по 2 байта, самое длинное слово имеет 9 символов - 18 байт. А мы указали 20, так что должно хватить.

Далее, в разделе глобальных переменных, после описания самой формы, опишем глобальную переменную `adres`:

```
var
  fMain: TfMain;
  adres: string; //адрес, откуда запущена программа
```

Эта переменная понадобится нам позже, в нее мы запишем адрес, откуда была запущена программа, чтобы потом по этому адресу отыскать файл с контактами. Ранее мы говорили, что если не указывать адрес, то компьютер будет искать файл в текущей папке, то есть - в папке с программой. Однако так бывает не всегда. Если вы параллельно работаете с несколькими другими программами, и все они то и дело обращаются к файлам, то текущий адрес может измениться. Вот почему всегда лучше подстраховаться, и указать настоящий адрес - это гарантирует вашу программу от любых ошибок.

Далее, "научим" программу принимать новые данные и выводить их в сетку. Сгенерируйте событие `OnClick` для кнопки "Добавить контакт". Код события такой:

```
procedure TfMain.bAddClick(Sender: TObject);
begin
  //очищаем поля, если там что-то есть:
  fEdit.eName.Text:= '';
  fEdit.eTelephone.Text:= '';
  //устанавливаем ModalResult редактора в mrNone:
  fEdit.ModalResult:= mrNone;
  //теперь выводим форму:
  fEdit.ShowModal;
  //если пользователь ничего не ввел - выходим:
  if (fEdit.eName.Text= '') or (fEdit.eTelephone.Text= '') then exit;
  //если пользователь не нажал "Сохранить" - выходим:
  if fEdit.ModalResult <> mrOk then exit;
  //иначе добавляем в сетку строку, и заполняем её:
  SG.RowCount:= SG.RowCount + 1;
  SG.Cells[0, SG.RowCount-1]:= fEdit.eName.Text;
  SG.Cells[1, SG.RowCount-1]:= fEdit.eTelephone.Text;
```

```
  SG.Cells[2, SG.RowCount-1]:= fEdit.CBNote.Text;
end;
```

Давайте разбираться с кодом. Прежде всего, в редакторе контактов `fEdit` мы очистили текст из строк с именем и номером телефона. Ведь если пользователь уже добавлял или редактировал контакты, в этих строках останутся записи с прошлого сеанса. Затем свойству `ModalResult` формы мы присваиваем значение `mrNone` - нет результата. Делаем это затем, чтобы потом точно знать, каким образом пользователь закрыл окно редактора контактов. Если кнопкой "Сохранить контакт", то в этом случае `ModalResult` формы будет `mrOk`, если кнопкой "Не сохранять", `ModalResult` формы будет `mrCancel`. Если же он просто закрыл окно крестиком в правом верхнем углу окна или клавишами <Alt+F4>, то в этом случае `ModalResult` формы так и останется `mrNone`.

Далее, мы выводим окно редактора контактов, как модальное.

К следующей строке кода программа доберется, когда пользователь введет (или не введет) новый контакт и закроет окно редактора. Прежде, чем что-то добавлять в сетку, мы делаем проверку - а ввел ли пользователь имя нового контакта, и телефон? Ведь если он ничего не ввел, то и сохранять нечего. И если он ввел один только телефон, без имени, или имя без телефона - какой смысл сохранять такой контакт? Поэтому, если нет имени или телефона, мы просто выходим из процедуры:

```
if (fEdit.eName.Text= '') or (fEdit.eTelephone.Text= '') then exit;
```

Однако может случиться, что пользователь ввел и имя, и телефон, но потом передумал сохранять контакт и нажал кнопку "Не сохранять", или просто закрыл окно редактора. Поэтому нам нужно сделать еще одну проверку:

```
if fEdit.ModalResult <> mrOk then exit;
```

Здесь мы смотрим значение свойства `ModalResult` формы. Оно

сможет стать `mrOK` только в одном случае - если пользователь нажал кнопку "Сохранить контакт". И если это не так, то выходим из процедуры, ничего не сохраняя.

Если мы не вышли из процедуры в предыдущих двух проверках, то это означает, что

1. нам есть, что сохранять
2. пользователь нажал "Сохранить контакт"

И если это так, значит, добавляем новую строку в сетку SG и заполняем три её колонки значениями из редактора контактов. Дополнительных проверок на существование текста примечания мы не делали, так как там в любом случае будет указан один из типов телефонов:

```
//иначе добавляем в сетку строку, и заполняем её:
SG.RowCount:= SG.RowCount + 1;
SG.Cells[0, SG.RowCount-1]:= fEdit.eName.Text;
SG.Cells[1, SG.RowCount-1]:= fEdit.eTelephone.Text;
SG.Cells[2, SG.RowCount-1]:= fEdit.CBNote.Text;
```

Сохраните проект и запустите его на выполнение. Если вы всё сделали верно, программа позволит вам добавлять новые контакты. Не обращайте внимания на недоработанный вид сетки - узкие столбцы без заголовков - это мы исправим чуть позже.

Но этого мало, хотелось бы ещё и сохранять эти контакты в файл. Для этого нам лучше всего подойдет событие формы `OnClose`, которое будет выполняться только один раз за сеанс работы программы - перед её закрытием. Форма вся закрыта компонентами, но по прошлым лекциям вы должны помнить, что форму можно выделить в верхнем окошке Инспектора объектов, где компоненты указаны в виде дерева, в самой вершине которого форма fMain. Еще можно выделить компонент верхнего над формой уровня - панель или сетку, и нажать <Esc>. При этом выделение перейдет на более низкий уровень - на форму.

Итак, выделите форму, в Инспекторе объектов перейдите на вкладку События и сгенерируйте событие `OnClose`. Код события такой:

```
procedure TfMain.FormClose(Sender: TObject; var CloseAction: TCloseAction
var
  MyCont: Contacts; //для очередной записи
  f: file of Contacts; //файл данных
  i: integer; //счетчик цикла
begin
  //если строки данных пусты, просто выходим:
  if SG.RowCount = 1 then exit;
  //иначе открываем файл для записи:
  try
    AssignFile(f, adres + 'telephones.dat');
    Rewrite(f);
    //теперь цикл - от первой до последней записи сетки:
    for i:= 1 to SG.RowCount-1 do begin
      //получаем данные текущей записи:
      MyCont.Name:= SG.Cells[0, i];
      MyCont.Telephon:= SG.Cells[1, i];
      MyCont.Note:= SG.Cells[2, i];
      //записываем их:
      Write(f, MyCont);
    end;
  finally
    CloseFile(f);
  end;
end;
```

Как видите, мы объявили переменную типа нашей записи `Contacts`, и типизированный файл с таким же типом. Еще мы объявили целочисленную переменную, которая понадобится нам для организации цикла.

Вначале мы проверяем - есть ли в сетке данные? Если `SG.RowCount = 1`, то это означает, что в сетке есть только одна строка - фиксированная, та, где у нас находятся заголовки колонок (а меньше 1 строки и быть не может, пользователь не сможет удалить фиксированную строку). Другими словами, данных нет. В этом случае мы просто выходим из процедуры.

Если программа пошла дальше, значит, данные в сетке есть, правда? Тогда мы начинаем работу по сохранению данных в файл. Эту работу мы помещаем в блок `try-finally-end`, работу с файлами (в том числе типизированными) мы с вами неоднократно рассматривали, так что подробно останавливаться тут не буду. Но пару замечаний сделать все же нужно. Во-первых, при ассоциации с файлом мы указали не только имя, но и адрес файла, который должен находиться в переменной adres:

```
AssignFile(f, adres + 'telephones.dat');
```

Однако эту переменную мы объявили, но ничего туда пока не внесли. Не обращайте на это внимания - мы сделаем это в следующем коде. Пока считайте, что адрес там есть, однако не спешите запускать программу на выполнение, точнее, не пытайтесь закрыть эту программу, если там есть данные.

Во-вторых, запись данных в файл мы организовали в виде цикла:

```
for i:= 1 to SG.RowCount-1 do begin
```

То есть, мы делаем эту работу от строки с индексом 1 (нулевой индекс у фиксированной строки, она нам для обработки данных не нужна), до строки с индексом `SG.RowCount-1`. Так, если в сетке у нас 10 строк, то индекс последней строки = 9, а `RowCount` вернет 10, потому мы отнимаем единицу.

Внутри цикла мы считываем в переменную типа запись данные из сетки, из текущей строки данных:

```
MyCont.Name:= SG.Cells[0, i];
```

Когда считали все значения, мы записываем эту переменную в типизированный файл. Поскольку типы переменной и файла совпадают, то запись будет успешной, а указатель сместится в конец файла. Затем цикл продолжится. При каждом новом шаге цикла значение `i` будет увеличиваться на единицу, и код будет ссылаться на

новую следующую строку. Записи будут записываться в файл одна за другой, до самого конца. В результате у нас получится нечто вроде базы данных.

Но чтобы от нашей базы был толк, программа при загрузке должна эти данные считывать обратно в сетку. Удобнее всего, как уже говорилось ранее, подготовительную работу помещать в событие формы `OnCreate`. Сгенерируйте это событие, код будет таким:

```
procedure TfMain.FormCreate(Sender: TObject);
var
  MyCont: Contacts; //для очередной записи
  f: file of Contacts; //файл данных
  i: integer; //счетчик цикла
begin
  //сначала получим адрес программы:
  adres:= ExtractFilePath(ParamStr(0));
  //настроим сетку:
  SG.Cells[0, 0]:= 'Имя';
  SG.Cells[1, 0]:= 'Телефон';
  SG.Cells[2, 0]:= 'Примечание';
  SG.ColWidths[0]:= 365;
  SG.ColWidths[1]:= 150;
  SG.ColWidths[2]:= 150;
  //если файла данных нет, просто выходим:
  if not FileExists(adres + 'telephones.dat') then exit;
  //иначе файл есть, открываем его для чтения и
  //считываем данные в сетку:
  try
    AssignFile(f, adres + 'telephones.dat');
    Reset(f);
    //теперь цикл - от первой до последней записи сетки:
    while not Eof(f) do begin
      //считываем новую запись:
      Read(f, MyCont);
      //добавляем в сетку новую строку, и заполняем её:
        SG.RowCount:= SG.RowCount + 1;
        SG.Cells[0, SG.RowCount-1]:= MyCont.Name;
        SG.Cells[1, SG.RowCount-1]:= MyCont.Telephon;
```

```
        SG.Cells[2, SG.RowCount-1]:= MyCont.Note;
      end;
    finally
      CloseFile(f);
    end;
  end;
```

Здесь мы, прежде всего, получаем адрес запускаемой программы в переменную `adres`:

```
  adres:= ExtractFilePath(ParamStr(0));
```

Как вы помните из лекции № 21 про консольные приложения, в любом проекте доступен параметр `ParamStr(0)`. В этом параметре хранится адрес и имя загружаемой программы. Функция `ExtractFilePath` отсекает имя файла, возвращая только его адрес с завершающим "\", например:

C:\Program Files\MyProg\

Адрес, откуда запущена программа, мы и сохраняем в переменную `adres`. Поскольку она глобальная, то становится доступной во всем модуле главной формы. Поэтому мы воспользовались этой переменной для получения адреса в событии `OnClose`, в котором прописали код сохранения данных в файл. Так как событие `OnCreate` выполняется первым, то адрес в переменной `adres` действительно, уже будет. Так что теперь данные будут сохраняться.

Далее мы настраиваем сетку - указываем ширину столбцов, вписываем заголовки колонок. Как это делается, мы разбирали в прошлой лекции.

Затем мы смотрим - есть ли файл? Программа может запускаться впервые, или пользователь мог удалить старый файл, в любом из этих случаев мы выходим, не производя дальнейших действий. Сетка все равно уже настроена, так что программа будет выглядеть, как надо. Да и адрес программы мы уже поместили в переменную `adres`.

Но если файл есть, то нужно считать из него данные, и поместить их в

сетку. Работа с файлом вам знакома, так что описывать процесс не буду. Замечу только, что в данном случае мы воспользовались циклом

```
while not Eof(f) do begin
```

То есть, делать, пока не конец файла. При открытии файла его указатель устанавливается в начало. Когда мы считываем первую запись в переменную `MyCont`, то указатель перемещается к следующей записи. Так, раз за разом, считывая запись в переменную, а затем, копируя из нее данные в сетку, мы двигаемся до конца файла, обрабатывая все записи, какие там есть. Причем при каждом шаге цикла мы сначала добавляем в сетку новую строку:

```
SG.RowCount:= SG.RowCount + 1;
```

а затем в эту, последнюю строку, мы записываем считанные из файла данные.

Итак, на данном этапе, мы научили программу добавлять новые записи (событие `OnClick` кнопки `bAdd`), научили её сохранять эти записи в файл (событие формы `OnClose`) и считывать их из файла (событие формы `OnCreate`). Осталось научить программу редактировать существующие записи - вдруг пользователь что то решит исправить, например, сменился телефон контакта, удалять контакты, ставшие ненужными, а также сортировать список по алфавиту. Ведь пользователь будет вносить контакты, как придется, не по порядку, а нужную запись гораздо проще найти в сортированном списке.

Итак, сгенерируйте событие `OnClick` для кнопки `bEdit` ("Редактировать контакт"). Код:

```
procedure TfMain.bEditClick(Sender: TObject);
begin
  //если данных в сетке нет - просто выходим:
  if SG.RowCount = 1 then exit;
  //иначе записываем данные в форму редактора:
  fEdit.eName.Text:= SG.Cells[0, SG.Row];
  fEdit.eTelephone.Text:= SG.Cells[1, SG.Row];
```

```
    fEdit.CBNote.Text:= SG.Cells[2, SG.Row];
    //устанавливаем ModalResult редактора в mrNone:
    fEdit.ModalResult:= mrNone;
    //теперь выводим форму:
    fEdit.ShowModal;
    //сохраняем в сетку возможные изменения,
    //если пользователь нажал "Сохранить":
    if fEdit.ModalResult = mrOk then begin
      SG.Cells[0, SG.Row]:= fEdit.eName.Text;
      SG.Cells[1, SG.Row]:= fEdit.eTelephone.Text;
      SG.Cells[2, SG.Row]:= fEdit.CBNote.Text;
    end;
  end;
```

Давайте разбираться с кодом. Как обычно, мы выходим из процедуры, если данных в сетке нет - редактировать ведь нечего. Иначе перед показом окна редактора контактов мы проделываем подготовительную работу - заносим в строки `TEdit` этой формы данные контакта - имя и телефон. Также свойству `Text` списка выбора `TComboBox` мы присваиваем текст примечания. Все это позволит представить пользователю данные текущей записи. Вы помните, как узнать индекс текущей строки сетки? Этот индекс хранится в свойстве `Row`:

```
    fEdit.eName.Text:= SG.Cells[0, SG.Row];
```

Далее, `ModalResult` формы редактора мы устанавливаем в `mrNone`, чтобы потом можно было определить, каким образом пользователь закрыл этот редактор.

Потом мы выводим форму, как модальное окно. В заключение мы проверяем - нажал ли пользователь кнопку "Сохранить контакт"? Если да, то мы перезаписываем данные в сетке на новый вариант, из формы редактора.

Нажав на кнопку "Редактировать контакт", пользователь сможет воспользоваться этим кодом. Однако в хороших программах пользователю часто предоставляют различные инструменты для

выполнения одной задачи. Например, хорошим тоном считается редактировать запись сетки при двойном клике по ней. Давайте, реализуем эту возможность, тем более что это совсем просто. Выделите сетку. В Инспекторе объектов перейдите на вкладку События. Только не торопитесь генерировать новое событие, ведь писать новый код нам не придется. В событии `OnDblClick` нажмите кнопку выбора, и выберите там событие `bEditClick` - то есть то, которое мы только что описали:

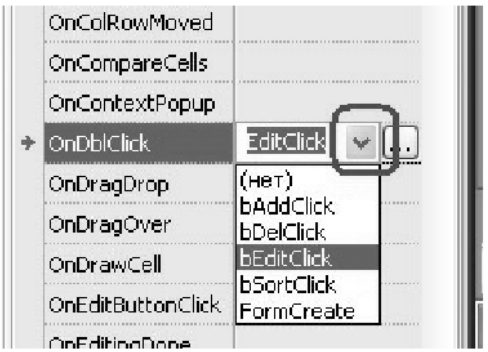


Рис. 25.3. Выбор уже существующего события

Таким образом, и при нажатии кнопки "Редактировать контакт", и при двойном клике по записи сетки будет выполнена одна и та же процедура.

Теперь научим программу удалять контакты. Сгенерируйте `OnClick` для кнопки "Удалить контакт":

```
procedure TfMain.bDelClick(Sender: TObject);
begin
  //если данных нет - выходим:
  if SG.RowCount = 1 then exit;
  //иначе выводим запрос на подтверждение:
  if MessageDlg('Требуется подтверждение',
              'Вы действительно хотите удалить контакт '" +
              SG.Cells[0, SG.Row] + '"?',
     mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
       SG.DeleteRow(SG.Row);
end;
```

Этот код проще, да? Опять мы выходим из процедуры, если данных нет, ведь тогда и удалять нечего. Удалить запись просто, но нужно ли это делать? Пользователь мог нажать на кнопку не подумав, или случайно. Поэтому перед удалением мы запросим подтверждения:

```
if MessageDlg('Требуется подтверждение',
          'Вы действительно хотите удалить контакт '" +
          SG.Cells[0, SG.Row] + "'?',
   mtConfirmation, [mbYes, mbNo, mbIgnore], 0) = mrYes then
     SG.DeleteRow(SG.Row);
```

Программа выведет подобное сообщение:

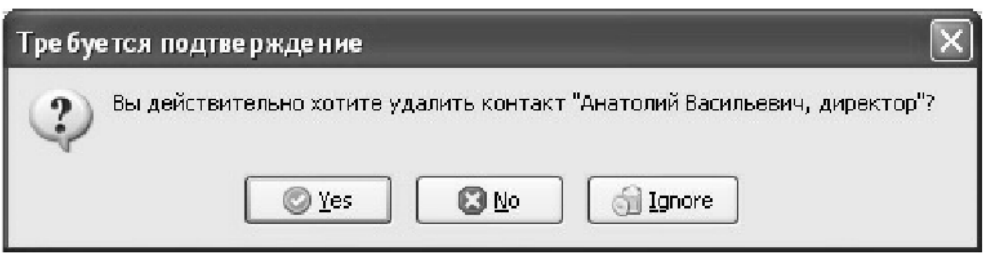


Рис. 25.4. Запрос на подтверждение

И только если пользователь подтвердит удаление контакта, он будет удален:

```
SG.DeleteRow(SG.Row);
```

Нам осталось сгенерировать событие `OnClick` для кнопки "Сортировать список". Его код:

```
procedure TfMain.bSortClick(Sender: TObject);
begin
  //если данных в сетке нет - просто выходим:
  if SG.RowCount = 1 then exit;
  //иначе сортируем список:
  SG.SortColRow(true, 0);
```

```
end;
```

Код достаточно простой, мы выполняем сортировку, если есть данные, по первому столбцу (с именами контактов):

```
SG.SortColRow(true, 0);
```

Сохраните проект, запустите его и попробуйте в работе. Приложение должно работать, и выглядеть примерно так:

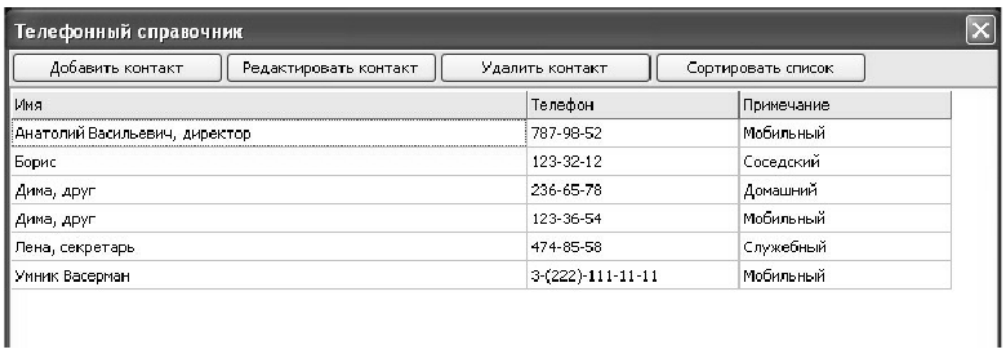


Рис. 25.5. Окно работающего приложения

Если есть желание, можете дополнительно расширить проект - самостоятельно сделать окно "О программе" к этому приложению, реализовать механизм сортировок иначе, где пользователь мог бы выбрать способ сортировки: по именам контактов, по телефонам, или по типам телефонов.

DLL

Лекция посвящена работе с динамически подключаемыми библиотеками - DLL. Рассматривается концепция DLL, создание библиотеки, статическое и динамическое связывание библиотеки с приложением.

Цель лекции

На лекции рассматриваются вопросы создания, а также различные способы использования динамически подключаемых библиотек - DLL.

Что такое DLL

Современное программирование стало достаточно сложным, уже написаны километры строк кода, и некоторые его части время от времени приходится дублировать в разных программах. Если бы каждый программист в каждой новой программе заново писал весь необходимый код, языки программирования никогда бы не получили такого стремительного развития. Однако мы в своих проектах можем использовать инструменты, созданные другими программистами - типы, процедуры, функции… Мы уже неоднократно подключали к нашим проектам различные модули и пользовались описанными в них средствами. И всё было бы хорошо, если бы не одно НО…

Дело в том, что при использовании модульного подхода, мы в момент компиляции проекта внедряем в него весь тот код, который был описан в модуле. И наша программа от такого внедрения "толстеет", добавляет иной раз по несколько мегабайт. Казалось бы, пустяки, современное аппаратное обеспечение достаточно хорошо развито, и мы можем себе это позволить. Но это все же тысячи строк чужого кода, причем далеко не весь этот код задействован в наших проектах! Теперь представьте, что вы пишете одну программу за другой, внедряете в нее одни и те же модули, в результате ваши проекты содержат мегабайты совершенно одинакового кода. И ладно бы только это, но представьте также, что вы одновременно загружаете и используете несколько программ (да здравствует многозадачность!). А эти программы тоже могут иметь в своем составе множество строк дублированного кода. Вот теперь у нас

получается настоящее расточительство - мы впустую переводим не только память жесткого диска, где хранятся программы, но и оперативную память, куда мы их для использования загрузили!

Есть еще один недостаток модульного подхода - мы можем использовать чужой код, только если он написан на таком же языке программирования, каким пользуемся мы. Ну, в Lazarus мы можем использовать модули Delphi, а также код на Ассемблере. Но если код был написан на C++ или Visual Basic, к примеру? Короче говоря, требовался новый подход, позволяющий использовать чужой код, не внедряя его непосредственно в программу, и, кроме того, "языконезависимый". И таким подходом стала DLL.

DLL (англ., Dynamic Link Library - Динамически Подключаемые Библиотеки) - универсальный механизм внедрения в программу функций и процедур, написанных другими программистами, и возможно, на другом языке программирования. Внедрение происходит не в момент компиляции проекта, как в случае модулей, а в момент загрузки готовой программы в оперативную память.

Использование DLL дает нам еще одно очень важное преимущество. Представьте, что у вас десятки программ импортируют какой-то код. И вот, этот код нужно изменить, или дополнить, потому что вы перешли на другую ОС, на другие стандарты или просто потому, что начальство потребовало от вас расширить функциональность ваших программ. Если бы вы использовали модули, вам пришлось бы туго - нужно было бы переделывать не только этот модуль, но и перекомпилировать ваши программы. Но вы ведь их создавали не неделю и не месяц, правда? Что, если часть из них вы писали когда-то на Delphi 7, другую часть на Delphi XE3, а третью - уже на Lazarus? Работа была бы та еще. Но с использованием DLL эта задача сильно упрощается. Вам нужно переделать всего лишь эту самую DLL-библиотеку, а программы, использующие её, переделывать уже не нужно!

Каким же образом используются эти DLL? Фактически, DLL - это библиотека тех инструментов, которые могут быть использованы вами в различных проектах. Причем с учетом того, что вы, может быть, будете использовать различные языки программирования. Итак, первым делом, вы создаете саму библиотеку - файл *.dll. Создается он, как

отдельное приложение, но после компиляции сам по себе работать все равно не сможет, поскольку не имеет собственной точки входа, как выполняемый exe-файл.

Далее, вы создаете приложение. Файл используемой динамической библиотеки вы помещаете в ту же папку, что и проект. В случае, если несколько ваших проектов будут использовать эту библиотеку, то её лучше поместить в какую-либо системную папку, например,

C:\Windows\system32

Тогда все эти программы смогут подгружать данную библиотеку, не указывая её адреса - Windows просматривает все свои системные папки и знает, где находится эта DLL. Функции и процедуры, описанные в DLL, наша программа будет воспринимать, как свои собственные. И когда во время выполнения программы происходит вызов таких функций и процедур, то будет подгружена соответствующая DLL.

Во время компиляции проекта код библиотеки не будет внедрен в программу. Это означает две вещи:

1. ваша программа не будет иметь лишнего кода;
2. при распространении программы между пользователями вам нужно будет также позаботиться, чтобы у них была установлена и эта dll-библиотека.

Когда пользователь загружает вашу программу, то также загружается и используемый dll-файл, на это тратится дополнительное время. Зато библиотека остается в оперативной памяти даже после завершения работы загрузившей её программы. И если затем будет загружена другая программа, использующая ту же DLL, она уже не будет тратить время на повторную загрузку этой библиотеки, а воспользуется уже загруженной в память версией. Более того, если будут одновременно работать несколько программ, использующих одну и ту же DLL, то они не будут загружать для себя по копии этой библиотеки, поскольку Windows гарантирует, что одновременно может быть загружена только одна копия динамической библиотеки. Так что первая из этих программ загрузит DLL в оперативную память, остальные будут использовать её же, не тратя лишнего времени на загрузку. Вот почему такие программы,

как MS Word или MS Excel медленно загружаются в первый раз - они обращаются ко множеству библиотек, которые также подгружаются в память. Зато когда вы загружаете MS Word вторично, он грузится гораздо быстрее, так как нужные библиотеки уже в оперативной памяти! Почувствовали разницу между DLL и модулями?

Кстати, динамическим библиотекам принято давать расширение *.dll, однако это не всегда так. Вы можете дать такой библиотеке любое расширение, или даже вообще не указывать его! Например, драйверы устройств имеют расширение *.drv, но это такие же динамические библиотеки.

У DLL есть один недостаток: поскольку они языконезависимы, а также по некоторым другим причинам, в этих библиотеках нельзя использовать глобальные переменные, которые будут доступны в программе. Поэтому в DLL нельзя использовать объекты (свойства объектов - это глобальные переменные). Так что воспринимайте DLL, как набор процедур и функций, который удобно подгружать к различным программам.

В нашем курсе подразумевается, что мы используем операционную систему семейства Windows, однако механизм динамически подключаемых библиотек существует во всех современных операционных системах. В Windows такая библиотека представляет собой файл *.dll, в Linux и Unix - *.so (Shared Object Library), а в MacOS X - *.dylib (Dynamic Shared Library). И все эти библиотеки можно создавать с помощью Lazarus. Учитывая, что мы используем ОС Windows, мы будем рассматривать создание dll-файлов.

Создание DLL

Загружаем Lazarus и выбираем команду "Файл -> Создать…". Откроется окно, в котором нам нужно будет выбрать "Библиотека":

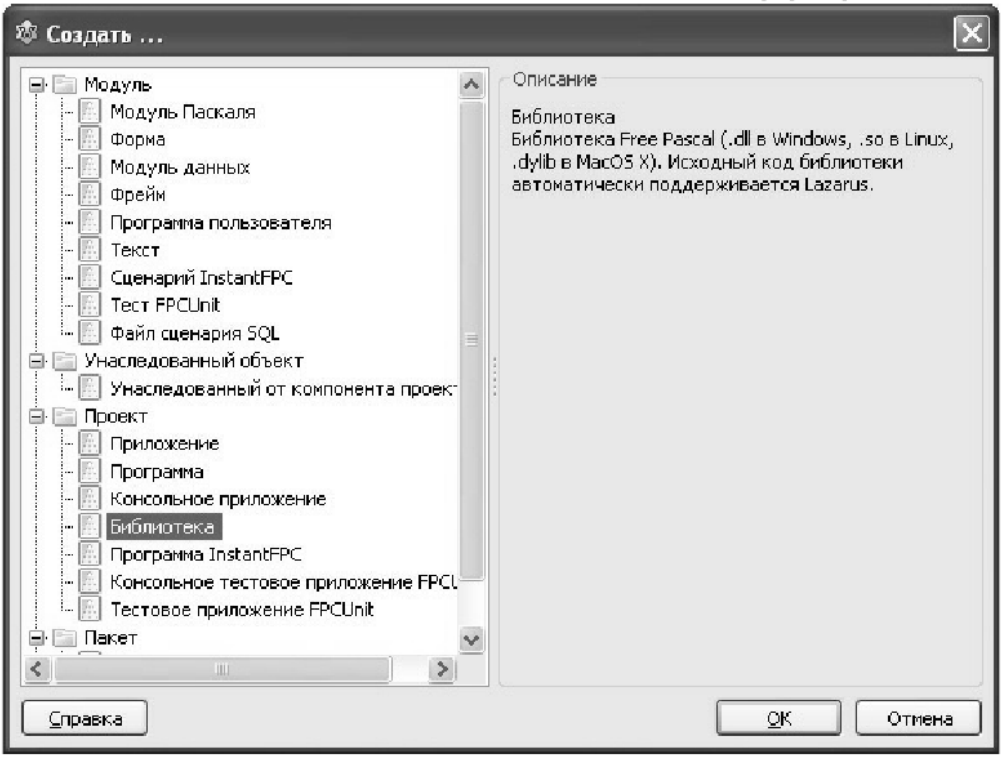


Рис. 26.1. Создание динамической библиотеки

В результате будет создан проект, в котором будет находиться следующий код:

```
library Project1;

{$mode objfpc}{$H+}

uses
  Classes
  { you can add units after this };

begin
end.
```

Нажмите кнопку "Сохранить всё", и сохраните проект под именем MyFirstDLL в папку 26-01. Как видите, первая строка кода

автоматически изменилась на

```
library MyFirstDLL;
```

а в указанной папке появилось три файла: MyFirstDLL.lpi, MyFirstDLL.lpr и MyFirstDLL.lps. Вы можете сразу же попробовать скомпилировать проект командой "Запуск -> Компилировать" или кнопками <Ctrl+F9> и в результате получите четвертый файл MyFirstDLL.dll. Это и есть динамическая библиотека, которая пока ещё ничего не умеет делать.

В разделе `uses`, после модуля `Classes`, через запятую мы можем добавить и другие, необходимые нам модули, однако не забывайте, что это будет увеличивать результирующий размер библиотеки. Поэтому нужно включать только те модули, без которых действительно, не обойтись. Обратите внимание - после последнего модуля точку с запятой ставить не нужно, она стоит после комментария, который сообщает, что вы можете добавлять модули после этого, то есть `Classes`. Комментарий можете оставить, а можете и удалить.

Весь основной код библиотеки должен находиться над завершающими `begin…end`. Для примера научим наш модуль делать несколько полезных вещей:

1. Переводить числа из обычных арабских в римские, и наоборот.
2. Делать простую шифрацию указанной строки.
3. Возвращать количество дней до дня рождения.

Поскольку нам придется иметь дело со строками, то следует иметь в виду вот что: в динамических библиотеках рекомендуют применять строковый тип `PChar`. Это связано с тем, что вашей DLL могут пользоваться программисты, создающие приложения на других платформах. В среде Windows тип `PChar` считается "родным", все WinAPI-функции используют этот тип. Внутри DLL-файла можно использовать любой строковый тип, но для передачи параметров и для получения результатов лучше использовать `PChar`.

Далее привожу полный код нашей библиотеки, после чего разберем некоторые моменты:

```
library MyFirstDLL;

{$mode objfpc}{$H+}

uses
  Classes, DateUtils, SysUtils
  { you can add units after this };

const
  R: array[1..13] of string[2] =
  ('I', 'IV', 'V', 'IX', 'X', 'XL', 'L', 'XC', 'C', 'CD', 'D', 'CM', 'M');
  A: array[1..13] of Integer =
  (1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000);

{шифрация/дешифрация}
function Code(s: PChar; Key: integer): PChar; stdcall;
var
  i: integer;
  ss: string;
begin
  ss:= s;
  for i:= 1 to length(s) do
    ss[i]:= char(Ord(ss[i]) xor Key);
  Result:= PChar(ss);
end;

{дней до очередного дня рождения}
function BeforeBirthday(Birthday:TDateTime): Integer; stdcall;
var
  d,m,y: word; //день, месяц и год
  mydate: TDateTime;
begin
  mydate:= Birthday;
  //разберем дату, изменим год на текущий и соберем обратно:
  DecodeDate(mydate, y, m, d);
  y:= YearOf(Date);
  mydate:= EncodeDate(y, m, d);
  //если ДР сегодня:
  if mydate = Date then Result:= 0
```

```
  //если будет:
  else if mydate > Date then Result:= DaysBetween(Date, mydate)
  //если уже был:
  else begin
    //установим дату ДР на следующий год и снова посчитаем:
    y:= y + 1;
    mydate:= EncodeDate(y, m, d);
    Result:= DaysBetween(Date, mydate);
  end; //else
end;

{Арабские в римские}
function ArToRom(N: integer): PChar; stdcall;
var
  i: integer;
  s: string;
begin
  s := '';
  i := 13;
  while N > 0 do begin
    while A[i] > N do Dec(i);
    s := s + R[i];
    Dec(N, A[i]);
  end;
  Result := PChar(s);
end;

{Римские в арабские}
function RomToAr(s: PChar): Integer; stdcall;
var
  i, p: Integer;
begin
  Result := 0;
  i := 13;
  p := 1;
  while p <= Length(s) do begin
    while Copy(s, p, Length(R[i])) <> R[i] do begin
      Dec(i);
      if i = 0 then Exit;
```

```
    end; //while 2
    Result := Result + A[i];
    p := p + Length(R[i]);
  end; //while 1
end;

exports
  Code name 'Code',
  BeforeBirthday name 'BeforeBirthday',
  ArToRom name 'ArToRom',
  RomToAr name 'RomToAr';

begin
end.
```

Листинг .

Разберем код. Прежде всего, мы включили в раздел `uses` два дополнительных модуля:

```
uses
  Classes, DateUtils, SysUtils
  { you can add units after this };
```

Эти модули были необходимы для работы с датами - для получения текущей даты, для разборки даты на составляющие (год, месяц, день) и на обратную сборку, а также для определения количества дней от одной даты до другой.

Далее, у нас описаны две константы:

```
const
  R: array[1..13] of string[2] =
  ('I', 'IV', 'V', 'IX', 'X', 'XL', 'L', 'XC', 'C', 'CD', 'D', 'CM', 'M');
  A: array[1..13] of Integer =
  (1, 4, 5, 9, 10, 40, 50, 90, 100, 400, 500, 900, 1000);
```

Эти константы представляют собой массивы - первый строковый, на два символа, второй - массив целых чисел. Оба массива нужны для замены типа цифр. Например, римская III соответствует арабской 3, римская X - это арабская 10, и так далее. Тут следует иметь в виду, что отрицательных римских цифр не бывает, и что минимальная цифра - единица. Кроме того, тип `integer` имеет максимальное значение 2 147 483 647. Вряд ли вам потребуется переводить на римские большую цифру - римские обозначения не так популярны и используются, в основном, в литературе, для обозначения номера главы или раздела. Однако, помните, что мы включаем в библиотеку только инструменты конвертирования, а проверку допустимых значений должен делать программист в использующей нашу DLL программе.

Пойдем дальше. Первой у нас описана функция кодирования / декодирования строк:

```
{шифрация/дешифрация}
function Code(s: PChar; Key: integer): PChar; stdcall;
var
  i: integer;
  ss: string;
begin
  ss:= s;
  for i:= 1 to length(s) do
    ss[i]:= char(Ord(ss[i]) xor Key);
  Result:= PChar(ss);
end;
```

Здесь мы прежде всего видим, что функция принимает параметры - строку типа `PChar` и целое число - ключ, и возвращает также строку `PChar`, хотя внутри самой функции используется тип `String`. Еще мы видим, что после объявления имени функции указывается ключевое слово `stdcall`, означающее, что для вызова этой функции будет использовано стандартное соглашение.

Чтобы в программе можно было использовать инструменты из динамических библиотек сторонних разработчиков, были разработаны специальные соглашения по вызову процедур. Эти соглашения

определяют различные правила вызова подпрограмм: как будут передаваться параметры - через стек, через регистры, через динамическую память; кто ответственный за очистку стека - вызывающая или вызываемая программа и т.д. Так, если используется стек, то чтение будет происходить справа налево. То есть, последние загруженные данные будут считаны первыми.

По умолчанию, используется соглашение `register`. Его имеет смысл использовать тогда, когда вы создаете на Lazarus и DLL, и использующую её программу. Такой вызов работает быстрее, однако он почти не применяется, так как обычно программисты предпочитают использовать более гибкие соглашения `stdcall` и `cdecl`.

Вызовы `register` и `pascal` передают параметры слева направо, то есть первый параметр слева вычисляется и передается в первую очередь, а последний параметр справа - вычисляется и передается последним. Вызовы `cdecl` и `stdcall` передают параметры наоборот, справа налево. В таблице ниже указаны способы вызова процедур:

Таблица 26.1. Сведения по соглашениям вызова подпрограмм

Команда вызова	Обработка параметров	Ответственный за очистку стека	Разрешена ли передача параметров через регистры
`Register`	Слева-направо	Подпрограмма	Да
`Pascal`	Слева-направо	Подпрограмма	Нет
`Cdecl`	Справа-налево	Вызывающая программа	Нет
`Stdcall`	Справа-налево	Подпрограмма	Нет

Совсем уж влезать в дебри машинного языка, пожалуй, не стоит, запомните только несколько рекомендаций. Для программ на Windows чаще всего используют соглашение `stdcall`. А если вы создаете DLL, которую затем могут использовать Си-программисты, то указывайте соглашение `cdecl`.

Пойдем дальше. Команда

```
ss:= s;
```

не просто присваивает переменной `ss` переданную в параметре строку, она одновременно делает преобразование типов из `PChar` в `String`. Если вы не забыли, такие преобразования можно делать явно, например:

```
ss:= String(s);
s:= PChar(ss);
```

В нашей функции преобразование из `PChar` в `String` не указано, так как оно происходит по умолчанию, однако обратное преобразование все же было сделать необходимо:

```
Result:= PChar(ss);
```

Внутри подпрограммы устроен цикл

```
for i:= 1 to length(s) do
  ss[i]:= char(Ord(ss[i]) xor Key);
```

Здесь, очередному символу строки ss присваивается результат работы побитовой операции `XOR`. Кстати, шифрование текста обычно делают именно с помощью `XOR`, наш пример только более простой. Что в данном случае делает `XOR`? Оператор `XOR` используется для инвертирования определённых битов числа (еще говорят, что `XOR` - операция исключающего ИЛИ). Для отдельных битов оператор `XOR` работает следующим образом (смотреть слева направо):

```
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
```

То есть, если два аргумента равны, `XOR` возвращает `False` (для

отдельных битов - ноль). А если аргументы различаются, XOR возвращает True (или 1 для битов).

Каждый символ таблицы ASCII имеет собственное числовое обозначение. Так, английская "A" имеет номер 65, английская "B" - 66, и так далее, это всё мы обсуждали в лекции № 5. Компьютер работает только в двоичной системе, для него "A" - это не десятичная 65, а двоичная 0100 0001. Так вот, XOR обрабатывает не всё число целиком, а отдельно каждый его бит. В нашем примере слева от XOR указан очередной символ строки, а справа - число-ключ, от которого зависит изменение бита символа. Учитывая, что в таблице ASCII может быть максимум, 255 символов, слишком большой ключ указывать не стоит, во избежание ошибок. 10 вполне достаточно, хотя можете и поэкспериментировать (у меня после значения ключа 31 русские буквы при шифрации начинали искажаться, так что выбирайте значения от 1 до 30). Таким образом, мы каждый символ строки кодируем в совершенно другой символ. В более сложных системах шифрования ключ может меняться от бита к биту.

Когда все символы строки обработаны, строка преобразуется в PChar, и результат возвращается обратно в программу.

Следующей у нас идет функция подсчета количества дней до следующего дня рождения. День рождения указывается в параметре, который имеет тип TDateTime. Чтобы подсчитать количество дней до очередного дня рождения, нам требуется получить новую дату - день и месяц дня рождения, а год мы должны указать текущий. Для этого мы вначале "разбираем" дату на составляющие:

```
DecodeDate(mydate, y, m, d);
```

Так мы получили отдельно, год, месяц и день. Затем мы изменили год на текущий:

```
y:= YearOf(Date);
```

После чего собрали дату обратно. Но это только полдела. Ведь может быть три варианта:

1. День рождения сегодня. В этом случае мы возвращаем 0 дней до дня рождения.
2. Дня рождения в этом году еще не было. Мы возвращаем количество дней от текущей даты до даты дня рождения, для этого используем функцию `DaysBetween`, которая возвращает разницу в днях между двумя указанными датами. Кстати, при желании вы можете использовать и другие подобные функции: `YearsBetween` - возвращает количество лет между двумя датами, `MonthBetween` - количество месяцев, `WeeksBetween` - количество недель, `HoursBetween` - количество часов, `MinutesBetween` - количество минут, `SecondsBetween` - количество секунд и `MilliSecondsBetween` - количество миллисекунд. Только имейте в виду, что если вам нужно получить разницу в часах, минутах, секундах или миллисекундах, то вместо `Date`, которая возвращает только текущую дату вам нужно будет использовать `Now`, которая возвращает текущие дату и время.
3. День рождения в этом году уже был. В этом случае нам нужна новая дата - дата дня рождения в следующем году. Для этого нам опять нужно "разобрать" дату, прибавить к году единицу и вновь собрать её. После чего возвращаем оставшееся количество дней от текущей даты до новой даты дня рождения.

Функции перевода римских цифр в арабские и обратно, для экономии места предлагаю рассмотреть самостоятельно - у нас ведь лекция про создание и вызов DLL, а не про синтаксис Паскаля. Если вы внимательно изучали предыдущие лекции, то сможете разобраться с синтаксисом. До этого мы не рассматривали только процедуру декремента `Dec`, которая уменьшает значения аргумента. По умолчанию, она уменьшает аргумент на единицу. Или можно указать аргумент и через запятую, значение, на которое нужно уменьшить аргумент:

```
Dec(i);  //уменьшает i на единицу
Dec(N, A[i]);  //уменьшает N на значение A[i]
```

С остальным синтаксисом вы должны быть уже знакомы.

В самом конце библиотеки у нас указан список функций, которые можно

будет вызывать из внешних программ:

```
exports
  Code name 'Code',
  BeforeBirthday name 'BeforeBirthday',
  ArToRom name 'ArToRom',
  RomToAr name 'RomToAr';
```

В библиотеке может быть и больше процедур и функций, в списке `exports` нужно указывать только те, которые будут доступны извне. У этих же функций указывались и соглашения вызовов. В нашем случае, все имеющиеся функции должны быть доступны извне.

Теперь о способе описаний списка. Директива

```
  Code name 'Code',
```

говорит о том, что наша функция `Code` должна вызываться из внешней программы по имени `Code`. Для чего это сделано? Допустим, во внешней программе уже есть функция `Code`, тогда мы получим конфликт имен. В этом случае мы могли бы вызывать подпрограмму под другим именем, например:

```
  Code name 'NewCode',
```

Тогда во внешней программе мы запрашивали бы не функцию `Code`, а функцию `NewCode`.

Кроме того, в списке `exports` можно указывать передачу подпрограмм не по именам, а по индексам, например, так:

```
exports
  Code, BeforeBirthday, ArToRom, RomToAr;
```

Тогда бы подпрограммам автоматически были бы присвоены индексы, от 1 до 4. Но индексы можно присвоить и принудительно, с помощью

директивы `index`, например, так:

```
exports
  Code index 1,
  BeforeBirthday index 2,
  ArToRom index 3,
  RomToAr index 4;
```

Что нам дают индексы? В среде Windows подпрограммы по индексам вызываются чуть быстрее, однако запоминать, под каким индексом находится нужная подпрограмма сложнее. Кроме того, использование индексации имеет смысл только в Windows, так как другие ОС формируют индексы автоматически. В общем, рекомендуется обращаться к подпрограммам по именам, как мы и сделали в нашей библиотеке.

На этом работа с динамической библиотекой закончена. Поскольку кнопка Run на панели инструментов для DLL недоступна, нажмите <Ctrl+F9> или выберите команду "Запуск -> Компилировать". В результате получите готовую библиотеку - файл MyFirstDLL.dll.

Вызов DLL из внешней программы

Библиотеку DLL можно связать с приложением двумя способами:

- статическим связыванием
- динамическим связыванием

Статическое связывание самое простое, оно подразумевает, что DLL будет загружена сразу, как только приложение начнет выполняться. Это самый простой способ использования DLL, обращение к процедурам и функциям динамической библиотеки такое же, как к обычным процедурам и функциям. Но есть и минусы использования статистического связывания.

Во-первых, при загрузке программы должны подгружаться и все используемые DLL, а это увеличивает время загрузки (при современной технике, правда, ненамного).

Во-вторых, чтобы пользователь мог работать с программой, он должен иметь все эти используемые DLL. Обычно конечно, так и есть, но бывают и исключения. Можно было бы раздать незарегистрированным пользователям усеченную версию программы, без некоторых ключевых DLL. Тогда они могли бы пользоваться программой, но не всеми её возможностями, а только основными. Зарегистрированным пользователям можно было бы дополнительно рассылать остальные DLL. При статическом связывании это недоступно.

В-третьих, при статическом связывании все загруженные DLL занимают память всё время, что работает программа, вне зависимости от того, использует ли она эти DLL в настоящее время, или нет.

Динамическое связывание подразумевает, что требуемая DLL будет подгружаться только тогда, когда требуется выполнить какую-то процедуру или функцию из неё. После чего DLL можно выгрузить, освободив память. Динамическое связывание также имеет плюсы и минусы. Плюсы в том, что программу можно использовать, не имея всех DLL. Загрузка программы будет осуществляться быстрее, поскольку DLL будут подгружаться по мере необходимости, а не вместе с программой.

Минусы же в том, что из-за необходимости подгружать и выгружать библиотеку, будет увеличиваться время обращения к её функциям и процедурам. Кроме того, само обращение к ним будет намного сложнее.

Статическое связывание DLL

Разберем вначале более простой способ. Откройте Lazarus с новым проектом, имя формы измените на fMain, в `Caption` напишите "`Статическое связывание DLL`", установите свойства `BorderStyle` в `bsDialog`, а `Position` в `poDesktopCenter`. Нажмите кнопку "Сохранить всё" и сохраните проект под именем Proba в папку 26-02. Модуль формы назовите как обычно, `Main`.

Теперь скопируйте файл MyFirstDLL.dll из папки 26-01 в папку 26-02, чтобы наш проект мог пользоваться этой DLL.

Далее, перейдите в Редактор кода. Чтобы мы могли использовать

функции и процедуры DLL, их нужно объявить так же, как в DLL. Объявления нужно делать сразу после ключевого слова `implementation`, и после директивы процессору `{$R *.lfm}`:

```
implementation

{$R *.lfm}
function Code(s: PChar; Key: integer): PChar; stdcall;
        external 'MyFirstDLL.dll';
function BeforeBirthday(Birthday:TDateTime): Integer; stdcall;
        external 'MyFirstDLL.dll';
function ArToRom(N: Integer): PChar; stdcall;
        external 'MyFirstDLL.dll';
function RomToAr(s: PChar): Integer; stdcall;
        external 'MyFirstDLL.dll';
```

Как видите, объявление требуемых нам функций такое же, каким было в DLL, но также после объявления была добавлена директива `external`:

```
        external 'MyFirstDLL.dll';
```

которая означает, что эти функции программа должна искать в соответствующем файле динамической библиотеки, ведь программа может использовать несколько DLL. И это всё, что нужно для статического связывания. В дальнейшем мы сможем обращаться к этим функциям, как к любым другим.

Займемся компонентами. Установите на форму метку `TLabel`, в `Caption` которой напишите "`Укажите день рождения:`". Далее установите компонент `TDateEdit` с вкладки Misc Палитры компонентов, здесь пользователь будет вводить или выбирать требуемую для проверки дату рождения. Свойство `Name` компонента переименуйте в `DE1`. Правее установите простую кнопку TButton, в `Caption` которой напишите "`До дня рождения`", а `Name` переименуйте в `bBeforeBirthday`.

Ниже установим компоненты для проверки перевода формата цифр из арабских в римские, и наоборот. Вначале установим поясняющую метку TLabel, в Caption которой напишем "Укажите целое число от 1 до 2147483647:". Далее установим строку TEdit, которую переименуем в eNumbers, и очистим свойство Text. Ниже установим две простые кнопки TButton, которые переименуем в bArToRom и bRomToAr, а в Caption этих кнопок напишем соответственно, "Арабское в римское" и "Римское в арабское".

Наконец, займемся компонентами для шифрования текста. Сначала метку TLabel с текстом в Caption "Впишите текст:". Далее строку TEdit, которую переименуем в eCode, и у которой очистим свойство Text. Ниже установим простую кнопку TButton, переименуем её в bCode, и в свойстве Caption напишем "Шифровать/дешифровать". Измените положение и размеры компонентов так, чтобы ваша форма выглядела примерно так:

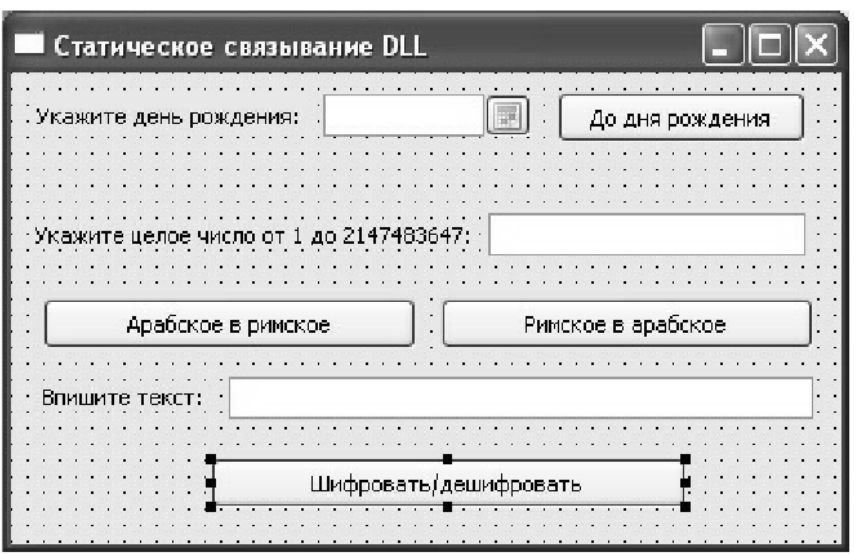


Рис. 26.2. Форма приложения

Нам осталось написать код события OnClick для наших четырех кнопок, и опробовать в действии работу с DLL. Код OnClick для кнопки bBeforeBirthday будет такой:

```
procedure TfMain.bBeforeBirthdayClick(Sender: TObject);
begin
  if DE1.Text = '' then exit;
  ShowMessage('До дня рождения осталось ' +
            IntToStr(BeforeBirthday(DE1.Date)) + ' дней');
end;
```

Просто, не правда ли? Если пользователь не выбрал дня рождения, оставив строку `DE1` пустой, мы выходим из процедуры, ничего не делая. Иначе мы выводим сообщение со сборной из трех частей строкой. Вызов функции из DLL:

```
IntToStr(BeforeBirthday(DE1.Date))
```

вернет нам в виде строки количество оставшихся до дня рождения дней.

Для кнопки bArToRom код события `OnClick` будет еще проще:

```
procedure TfMain.bArToRomClick(Sender: TObject);
begin
  if eNumbers.Text = '' then exit;
  eNumbers.Text:= ArToRom(StrToInt(eNumbers.Text));
end;
```

Здесь мы делаем единственную проверку - не пуста ли строка `eNumbers`. Имейте в виду, что в реальном приложении вам еще потребовалась бы проверка на содержимое текста: число ли это, целое ли, входит ли оно в диапазон от 1 до 2 147 483 647? В нашей пробной программе для экономии места мы предполагаем, что пользователь не будет вводить некорректных данных.

Для кнопки bRomToAr код похожий:

```
procedure TfMain.bRomToArClick(Sender: TObject);
begin
  if eNumbers.Text = '' then exit;
  eNumbers.Text:= IntToStr(RomToAr(PChar(eNumbers.Text)));
```

```
  end;
```

В реальной программе вам также пришлось бы делать проверку на корректность ввода данных римских цифр. Здесь же мы будем считать, что пользователь ввел правильные римские цифры.

Ну и, наконец, для кнопки bCode код будет таким:

```
procedure TfMain.bCodeClick(Sender: TObject);
begin
  if eCode.Text = '' then exit;
  eCode.Text:= Code(PChar(eCode.Text), 10);
end;
```

Выше я советовал вам выбирать размер ключа в диапазоне от 1 до 30, в вызове функции `Code` из DLL я указал размер ключа 10. Вот, собственно, и все. Сохраните проект и запустите его. Если вы все сделали правильно, ваша программа будет работать, как надо:

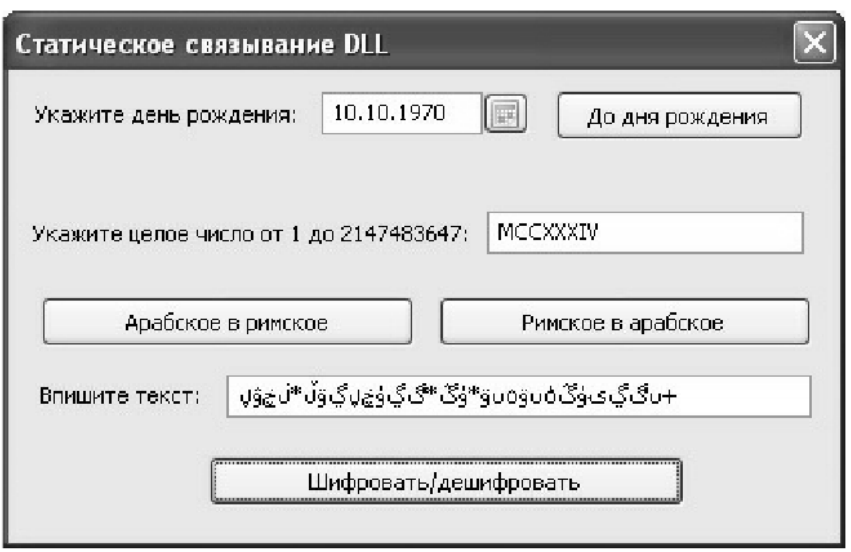


Рис. 26.3. Окно рабочей программы

Динамическое связывание DLL

Динамическое связывание происходит намного сложнее статического, причем динамическое связывание в Lazarus несколько отличается от такого же связывания в Delphi, поэтому примеры Delphi тут работать не будут.

Итак, начнем с того, что процедуру динамического связывания можно разбить на четыре этапа:

1. Загрузка динамической библиотеки функцией `LoadLibrary` и получение её дескриптора - числового идентификатора библиотеки в системе. Такой идентификатор представляет собой целое число. Если загрузка библиотеки была неудачной, то в дескриптор запишется ноль.
2. Получение адреса нужной функции (или функций) с помощью `GetProcAddress`.
3. Непосредственно работа с нужной функцией из DLL.
4. Выгрузка DLL из памяти.

Причем вы можете выполнять все действия по загрузке библиотеки в память и её выгрузке как для вызова каждой процедуры или функции отдельно, так и для вызова группы подпрограмм. Например, пользователь захотел выполнить некую операцию. Для этой операции требуется вызвать, скажем, десять подпрограмм (процедур и функций) из DLL. Будет глупо открывать и закрывать DLL отдельно для каждой подпрограммы, правильней будет открыть DLL, выполнить все эти десять подпрограмм, а затем закрыть DLL.

Но в нашем примере для наглядности мы будем загружать и выгружать DLL отдельно для каждой операции.

Для этого приложения нам потребуются точно такие же компоненты на форме, как и у предыдущего. Если вам не лень, то можно вернуться назад, и подготовить форму таким же образом, как и у проекта Proba. Однако можно сделать проще. Загрузите в Lazarus проект Proba из папки 26-02, для этого вам нужно загрузить файл Proba.lpi или Proba.lpr. Далее выведите на передний план Редактор формы. Щелкните правой кнопкой мыши по свободному месту формы и выберите команду "Выделить всё". При этом окажутся выделенными все компоненты формы. Затем выберите команду главного меню "Правка ->

Копировать".

Теперь начнем новый проект. Выберите команду "Файл -> Создать -> Приложение", и у вас откроется новый проект с пустой формой. Растяните форму по ширине и высоте, чтобы на ней легко уместились все компоненты, позже размеры формы можно будет подкорректировать. Теперь выберите команду главного меню "Правка -> Вставить". При этом все нужные компоненты появятся на форме на том же самом месте. Кроме того, они сохранят свои имена и прочие настройки. Подкорректируйте размер формы. Теперь переименуйте форму в fMain, в `Caption` напишите "`Динамическое связывание DLL`", в `BorderStyle` выберите `bsDialog`, а в `Position` - `poDesktopCenter`. Сохраните проект в папку 26-03 под именем Proba2, модулю формы дайте имя `Main`. Не забудьте скопировать файл MyFirstDLL.dll и в эту папку.

Теперь нам потребуется некоторая подготовка перед тем, как мы начнем динамически связывать нашу DLL. Прежде всего, в раздел `uses` добавьте еще один модуль - `Dynlibs`. Именно в нём описаны необходимые инструменты для динамического подключения библиотек.

Далее, перед глобальным разделом `var` нам нужно описать типы наших подключаемых функций, а в самом разделе `var` добавить переменную-дескриптор библиотеки, а также по переменной на каждую функцию:

```
type
  TCode = function(s: PChar; Key: integer): PChar; stdcall;
  TBeforeBirthday = function(Birthday:TDateTime): Integer; stdcall;
  TArToRom = function(N: integer): PChar; stdcall;
  TRomToAr = function(s: PChar): Integer; stdcall;

var
  fMain: TfMain;
  MyH: THandle = 0; //для дескриптора библиотеки
  Code: TCode;
  BeforeBirthday: TBeforeBirthday;
  ArToRom: TArToRom;
  RomToAr: TRomToAr;
```

```
implementation

{$R *.lfm}
```

Обратите внимание, как мы описываем тип функций:

```
TCode = function(s: PChar; Key: integer): PChar; stdcall;
```

Тип мы назвали `TCode`, вы можете дать другое имя, но для типов и классов традиционно принято начинать имя с большой буквы "T". В этом типе мы указали, что создается функция с такими то параметрами (как в DLL), которая возвращает тип `PChar`, и будет использовать соглашение `stdcall`. При этом имени самой функции мы не указываем. Таким же образом мы создаем еще три типа.

Далее, в глобальном разделе var мы добавляем такую переменную:

```
MyH: THandle = 0; //для дескриптора библиотеки
```

Это - дескриптор. Когда мы загрузим DLL в память, в эту переменную попадет идентификатор нашей библиотеки. Ведь в памяти всё время загружено множество самых разных DLL, используемых системой, и Windows как-то нужно различать, к которой из них вы обращаетесь. По умолчанию сразу же выставляем нулевое значение. Если попытка считать DLL в память будет неудачной, например, эта DLL отсутствует, то дескриптор также будет нулевым.

Далее мы создаем по переменной для каждой вызываемой из DLL функции. Например, для функции `Code` мы создаем переменную

```
Code: TCode;
```

Тип `TCode` мы описали выше, здесь просто используем его же. В принципе, эти переменные необязательно делать глобальными, можно было описать их в разделе `var` той процедуры, откуда будем вызывать. Но в больших проектах одну и ту же DLL-функцию обычно вызывают из

множества различных процедур, поэтому удобней описывать переменные, как глобальные.

Описав дескриптор и переменные для функций, мы сделали всю подготовительную работу.

Переходим ко второму этапу - динамической загрузке DLL, использовании нужной функции и выгрузке DLL из памяти.

Сгенерируйте событие `OnClick` для кнопки bBeforeBirthday. Код события отличается от прошлого проекта, он более сложный:

```
procedure TfMain.bBeforeBirthdayClick(Sender: TObject);
begin
  //выходим, если пусто:
  if DE1.Text = '' then exit;
  //открываем библиотеку и получаем ее дескриптор:
  MyH:= LoadLibrary('MyFirstDLL.dll');
  //выходим, если ошибка:
  if MyH = 0 then begin
    ShowMessage('Ошибка открытия библиотеки MyFirstDLL.');
    exit;
  end;
  //получаем адрес нужной функции:
  BeforeBirthday:= TBeforeBirthday(GetProcAddress(MyH, 'BeforeBirthday'));
  //если функция прочиталась, вычисляем, иначе выходим:
  if @BeforeBirthday <> nil then  ShowMessage('До дня рождения осталось '
     IntToStr(BeforeBirthday(DE1.Date)) + ' дней')
  else ShowMessage('Нужная функция отсутствует в библиотеке');
  //выгружаем библиотеку из памяти:
  FreeLibrary(MyH);
end;
```

Давайте подробно разбирать весь этот код. Для начала, как и в прошлом проекте, мы сделали проверку - есть ли что-то в строке `DE1`, и если нет, то выходим, так как в дальнейших действиях смысла нет. Если текст есть (пользователь вписал или выбрал дату рождения), то выполняется дальнейший код. Прежде всего, мы загружаем в память библиотеку,

сразу же получая дескриптор на неё:

```
MyH:= LoadLibrary('MyFirstDLL.dll');
```

Функция `LoadLibrary` описана в модуле `Dynlibs`, который мы подключили в разделе `uses`, и пытается открыть указанную в параметре библиотеку. Если имя библиотечного файла указано без адреса, как в нашем случае, то файл библиотеки ищется в той же папке, откуда запущена программа. Если его там нет, то библиотека ищется в текущей папке (она может отличаться от папки с программой), затем в системных каталогах Windows и, наконец, в папках, указанных в системной переменной `Path`. Поэтому DLL лучше устанавливать в папку с программой, или в системную папку Windows, если эту DLL будут использовать несколько приложений. Такой папкой может быть, например,

C:\Windows\system32

Если файл MyFirstDLL.dll был найден, то `LoadLibrary` загрузит его в память, а дескриптор передаст в `MyH`. Если по какой то причине загрузить библиотеку не получилось, то `LoadLibrary` передаст в `MyH` значение 0. Поэтому дальше мы делаем проверку - успешно ли загрузилась библиотека? И если нет, то выводим соответствующее сообщение, и выходим:

```
if MyH = 0 then begin
  ShowMessage('Ошибка открытия библиотеки MyFirstDLL.');
  exit;
end;
```

Далее, мы получаем адрес в памяти, где находится нужная нам процедура:

```
BeforeBirthday:= TBeforeBirthday(GetProcAddress(MyH, 'BeforeBirthday'));
```

Для этого мы указываем имя типа этой функции и используем функцию `GetProcAddress`. Эта функция работает так. В качестве параметров

она получает дескриптор нужной библиотеки и название функции (процедуры), которую нам требуется оттуда вызвать. Затем она возвращает адрес этой функции в памяти, который попадает в нашу переменную `BeforeBirthday`. Кстати, необязательно давать этим переменным такие же имена, как и у функций, просто так удобней.

Если по какой то причине адрес функции получить не удалось, например, в памяти находится устаревшая версия DLL, где эта функция отсутствует, то `GetProcAddress` вернет значение `nil`, то есть, ничего. Именно поэтому мы делаем ещё одну проверку:

```
if @BeforeBirthday <> nil then  ShowMessage('До дня рождения осталось '
   IntToStr(BeforeBirthday(DE1.Date)) + ' дней')
else ShowMessage('Нужная функция отсутствует в библиотеке');
```

Поскольку переменная `BeforeBirthday` является, на самом деле, указателем, то в ней находится не функция из DLL с аналогичным именем, а адрес этой функции в памяти. Чтобы посмотреть, указывает ли на что-то этот указатель, мы используем оператор получения адреса `@`, указанный перед именем указателя:

```
if @BeforeBirthday <> nil
```

Если по данному адресу находится значение, не равное `nil`, значит, требуемая функция доступна, и мы выводим сообщение, как в предыдущем проекте:

```
ShowMessage('До дня рождения осталось ' + IntToStr(BeforeBirthday(DE1.
```

Иначе функция недоступна, о чем мы и сообщаем пользователю, не обращаясь при этом к функции:

```
else ShowMessage('Нужная функция отсутствует в библиотеке');
```

В принципе, в нашем небольшом проекте можно было бы обойтись и без всех этих многочисленных проверок, но я хотел показать, каким

образом динамически подключаются DLL в реальных проектах. Это на первый взгляд весь этот код кажется сложным, а когда разберетесь и опробуйте его, он станет достаточно понятным. Под конец мы выгружаем DLL из памяти:

```
  FreeLibrary(MyH);
```

После этого, обращаться к её функциям уже будет нельзя. Код обращения к остальным функциям мы так подробно разбирать не будем, поскольку он аналогичен рассмотренному выше коду. Код события `OnClick` кнопки bArToRom следующий:

```
procedure TfMain.bArToRomClick(Sender: TObject);
begin
  if eNumbers.Text = '' then exit;
  //открываем библиотеку и получаем ее дескриптор:
  MyH:= LoadLibrary('MyFirstDLL.dll');
  //выходим, если ошибка:
  if MyH = 0 then begin
    ShowMessage('Ошибка открытия библиотеки MyFirstDLL.');
    exit;
  end;
  //получаем адрес нужной функции:
  ArToRom:= TArToRom(GetProcAddress(MyH, 'ArToRom'));
  //если функция прочиталась, вычисляем, иначе выходим:
  if @ArToRom <> nil then eNumbers.Text:= ArToRom(StrToInt(eNumbers.Tex
  else ShowMessage('Нужная функция отсутствует в библиотеке');
  //выгружаем библиотеку из памяти:
  FreeLibrary(MyH);
end;
```

Код события `OnClick` кнопки bRomToAr такой:

```
procedure TfMain.bRomToArClick(Sender: TObject);
begin
  if eNumbers.Text = '' then exit;
  //открываем библиотеку и получаем ее дескриптор:
```

```
  MyH:= LoadLibrary('MyFirstDLL.dll');
  //выходим, если ошибка:
  if MyH = 0 then begin
    ShowMessage('Ошибка открытия библиотеки MyFirstDLL.');
    exit;
  end;
  //получаем адрес нужной функции:
  RomToAr:= TRomToAr(GetProcAddress(MyH, 'RomToAr'));
  //если функция прочиталась, вычисляем, иначе выходим:
  if @RomToAr <> nil then
    eNumbers.Text:= IntToStr(RomToAr(PChar(eNumbers.Text)))
  else ShowMessage('Нужная функция отсутствует в библиотеке');
  //выгружаем библиотеку из памяти:
  FreeLibrary(MyH);
end;
```

И, наконец, код `OnClick` кнопки bCode такой:

```
procedure TfMain.bCodeClick(Sender: TObject);
begin
  if eCode.Text = '' then exit;
  //открываем библиотеку и получаем ее дескриптор:
  MyH:= LoadLibrary('MyFirstDLL.dll');
  //выходим, если ошибка:
  if MyH = 0 then begin
    ShowMessage('Ошибка открытия библиотеки MyFirstDLL.');
    exit;
  end;
  //получаем адрес нужной функции:
  Code:= TCode(GetProcAddress(MyH, 'Code'));
  //если функция прочиталась, вычисляем, иначе выходим:
  if @Code <> nil then eCode.Text:= Code(PChar(eCode.Text), 10)
  else ShowMessage('Нужная функция отсутствует в библиотеке');
  //выгружаем библиотеку из памяти:
  FreeLibrary(MyH);
end;
```

Сохраните проект, запустите его на выполнение и убедитесь, что работа динамически связанной DLL ничем не отличается от статически связанной DLL. Замедления вызовов функций если и есть, то практически не заметны. Какой из этих методов выбирать, зависит от конкретной ситуации и предпочтений программиста. Я, к примеру, обычно использую статическую связку с DLL.

Тестирование и отладка

Лекция носит факультативный характер. Здесь мы рассматриваем виды допускаемых в программировании ошибок, способы тестирования и отладки программ, инструменты встроенного отладчика.

Цель лекции

Освоить работу с встроенным отладчиком, изучить категории ошибок, способы их обнаружения и устранения.

Тестирование и отладка программы

Чем больше опыта имеет программист, тем меньше ошибок в коде он совершает. Но, хотите верьте, хотите нет, даже самый опытный программист всё же допускает ошибки. И любая современная среда разработки программ должна иметь собственные инструменты для отладки приложений, а также для своевременного обнаружения и исправления возможных ошибок. Программные ошибки на программистском сленге называют багами (англ. bug - жук), а программы отладки кода - дебаггерами (англ. debugger - отладчик). Lazarus, как современная среда разработки приложений, имеет собственный встроенный отладчик, работу с которым мы разберем на этой лекции.

Ошибки, которые может допустить программист, условно делятся на три группы:

1. Синтаксические
2. Времени выполнения (run-time errors)
3. Алгоритмические

Синтаксические ошибки

Синтаксические ошибки легче всего обнаружить и исправить - их обнаруживает компилятор, не давая скомпилировать и запустить программу. Причем компилятор устанавливает курсор на ошибку, или

после неё, а в окне сообщений выводит соответствующее сообщение, например, такое:

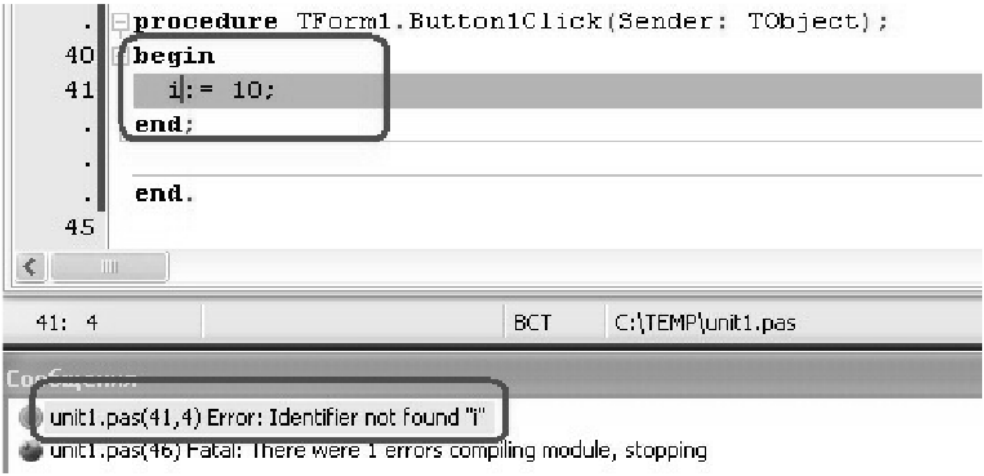


Рис. 27.1. Найденная компилятором синтаксическая ошибка - нет объявления переменной i

Подобные ошибки могут возникнуть при неправильном написании директивы или имени функции (процедуры); при попытке обратиться к переменной или константе, которую не объявляли (рис. 27.1); при попытке вызвать функцию (процедуру, переменную, константу) из модуля, который не был подключен в разделе `uses`; при других аналогичных недосмотрах программиста.

Как уже говорилось, компилятор при нахождении подобной ошибки приостанавливает процесс компиляции, выводит сообщение о найденной ошибке и устанавливает курсор на допущенную ошибку, или после неё. Программисту остается только внести исправления в код программы и выполнить повторную компиляцию.

Ошибки времени выполнения

Ошибки времени выполнения (run-time errors) тоже, как правило, легко устранимы. Они обычно проявляются уже при первых запусках программы, или во время тестирования. Если такую программу запустить из среды Lazarus, то она скомпилируется, но при попытке загрузки, или в момент совершения ошибки, приостановит свою работу,

выведя на экран соответствующее сообщение. Например, такое:

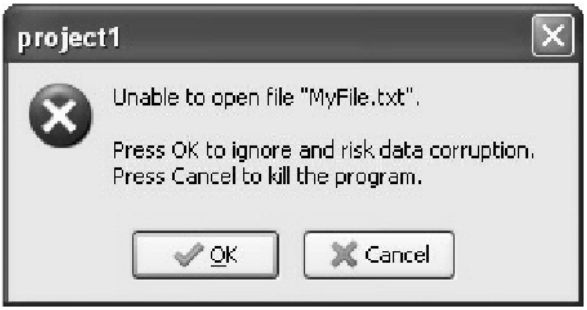


Рис. 27.2. Сообщение Lazarus об ошибке времени выполнения

В данном случае программа при загрузке должна была считать в память отсутствующий текстовый файл MyFile.txt. Поскольку программа вызвала ошибку, она не запустилась, но в среде Lazarus процесс отладки продолжается, о чем свидетельствует сообщение в скобках в заголовке главного меню, после названия проекта. Программисту в подобных случаях нужно сбросить отладчик командой меню "Запуск -> Сбросить отладчик", после чего можно продолжить работу над проектом.

Ошибка времени выполнения может возникнуть не только при загрузке программы, но и во время её работы. Например, если бы попытка чтения несуществующего файла была сделана не при загрузке программы, а при нажатии на кнопку, то программа бы нормально запустилась и работала, пока пользователь не нажмет на эту кнопку.

Если программу запустить из самой Windows, при возникновении этой ошибки появится такое же сообщение. При этом если нажать "OK", программа даже может запуститься, но корректно работать все равно не будет.

Ошибки времени выполнения бывают не только явными, но и неявными, при которых программа продолжает свою работу, не выводя никаких сообщений, а программист даже не догадывается о наличии ошибки. Примером неявной ошибки может служить так называемая утечка памяти. Утечка памяти возникает в случаях, когда программист забывает освободить выделенную под объект память. Например, мы объявляем переменную типа `TStringList`, и работаем с ней:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новая строка');
end;
```

В данном примере программист допустил типичную для начинающих ошибку - не освободил класс `TStringList`. Это не приведет к сбою или аварийному завершению программы, но в итоге можно бесполезно израсходовать очень много памяти. Конечно, эта память будет освобождена после выгрузки программы (за этим следит операционная система), но утечка памяти во время выполнения программы тоже может привести к неприятным последствиям, потребляя все больше и больше ресурсов и излишне нагружая процессор. В подобных случаях после работы с объектом программисту нужно не забывать освобождать память:

```
begin
  MySL:= TStringList.Create;
  MySL.Add('Новая строка');
  ...; //работа с объектом
  MySL.Free; //освободили объект
end;
```

Однако ошибки времени выполнения могут случиться и во время работы с объектом. Если есть такой риск, программист должен не забывать про возможность обработки исключительных ситуаций. В данном случае вышеприведенный код правильней будет оформить таким образом:

```
begin
  try
    MySL:= TStringList.Create;
    MySL.Add('Новая строка');
    ...; //работа с объектом
  finally
    MySL.Free; //освободили объект, даже если была ошибка
  end;
```

```
end;
```

Итак, во избежание ошибок времени выполнения программист должен не забывать делать проверку на правильность ввода пользователем допустимых значений, заключать опасный код в блоки `try…finally…end` или `try…except…end`, делать проверку на существование открываемого файла функцией `FileExists` и вообще соблюдать предусмотрительность во всех слабых местах программы. Не полагайтесь на пользователя, ведь недаром говорят, что если в программе можно допустить ошибку, пользователь эту возможность непременно найдет.

Алгоритмические ошибки

Если вы не допустили ни синтаксических ошибок, ни ошибок времени выполнения, программа скомпилировалась, запустилась и работает нормально, то это еще не означает, что в программе нет ошибок. Убедиться в этом можно только в процессе её тестирования.

Тестирование - процесс проверки работоспособности программы путем ввода в неё различных, даже намеренно ошибочных данных, и последующей контрольной проверке выводимого результата.

Если программа работает правильно с одними наборами исходных данных, и неправильно с другими, то это свидетельствует о наличии алгоритмической ошибки. Алгоритмические ошибки иногда называют логическими, обычно они связаны с неверной реализацией алгоритма программы: вместо "+" ошибочно поставили "-", вместо "/" - "*", вместо деления значения на 0,01 разделили на 0,001 и т.п. Такие ошибки обычно не обнаруживаются во время компиляции, программа нормально запускается, работает, а при анализе выводимого результата выясняется, что он неверный. При этом компилятор не укажет программисту на ошибку - чтобы найти и устранить её, приходится анализировать код, пошагово "прокручивать" его выполнение, следя за результатом. Такой процесс называется отладкой.

Отладка - процесс поиска и устранения ошибок, чаще алгоритмических. Хотя отладчик позволяет справиться и с ошибками времени

выполнения, которые не обнаруживаются явно.

Работа с отладчиком

Давайте от теории перейдем к практике. Загрузите Lazarus с новым проектом, установите на форму простую кнопку и сохраните проект в папку 27-01. Имена проекта, формы, модуля и кнопки изменять не нужно, оставьте имена, данные по умолчанию.

Далее, сгенерируйте событие `OnClick` для кнопки, в котором напишите следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  i: Integer;
  st: TStringList;
begin
  //создаем список строк:
  st:= TStringList.Create;
  try
    //генерируем список:
    for i:= -3 to 3 do begin
      st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
    end;
    //выводим список на экран:
    ShowMessage(st.Text);
  finally
    //st будет освобождена даже в случае run-time ошибки:
    st.Free;
  end;
end;
```

Что мы тут делаем? Целочисленную переменную `i` используем в качестве счетчика для цикла `for`. Цикл производим от -3 до 3, то есть, 7 раз, включая ноль. В теле цикла мы делим 10 на значение `i`, результат оформляем в виде строки и добавляем к списку строк `st`. Выше говорилось, что подобные действия нужно заключать в блок обработки

исключительных ситуаций `try-finally-end`, что мы и сделали.

Если вы внимательно изучали курс, то невооруженным глазом видите, что при четвертом проходе цикла произойдет ошибка времени выполнения - деление 10 на ноль. Такой очевидный пример больше всего подходит для знакомства с встроенным отладчиком, так как вы уже знаете, где будет ошибка, и сможете проанализировать работу отладчика. Поэтому притворимся, что не подозреваем об ошибке.

Итак, программу мы написали, сохранили, пора её компилировать. Нажмите кнопку "Запустить" на Панели управления (или <F9>). Программа нормально скомпилировалась и запустилась. Нажмем кнопку Button1. И тут же получаем ошибку:

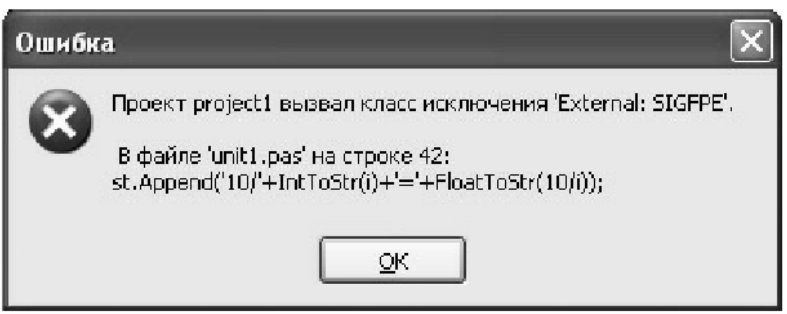


Рис. 27.3. Сообщение Lazarus об ошибке

Судя по сообщению, ошибка произошла во время выполнения кода 42-й строки. Ладно, нажмем "OK" и командой "Запуск -> Сбросить отладчик" прекратим выполнение программы. Вернемся к коду и проанализируем 42-ю строку (если вы добавляли пустые строки, то у вас будет другой номер):

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

Ну что, ничего криминального тут нет, почему же произошла ошибка? Код верный и должен был нормально выполняться… Когда вы заходите в подобный тупик, помочь вам может здравый смысл и встроенный отладчик. Здравый смысл говорит, что ошибка произошла где-то в теле цикла `for`. А чтобы воспользоваться отладчиком, нужно приостановить выполнение программы на этом цикле, чтобы потом

построчно его продолжить. Для остановки работы программы служат так называемые точки останова (англ. breakpoints).

Точки останова - это строки, перед выполнением которых отладчик приостанавливает выполнение программы, и ждет ваших дальнейших действий.

Вы можете установить одну такую точку или несколько, в различных частях кода. Поскольку ошибка возникает в 42-й строке, разумней будет приостановить выполнение на предыдущей, 41-й строке. Переведите курсор на эту строку, на любое её место.

Установить точку останова можно разными способами:

- Командой главного меню "Запуск -> Добавить точку останова -> Точка останова в исходном коде". В открывшемся окне "Параметры точки останова" нажать "OK".
- Щелкнуть по строке правой кнопкой, и в всплывающем меню выбрать "Отладка -> Переключить точку останова".
- Нажать "горячую клавишу" <F5>.
- Щелкнуть по нужной строке в левой части Редактора кода, где указаны номера строк.

Последние два способа наиболее удобны, но выбирать вам. В любом случае, строка с установленной точкой останова окрасится красным цветом:

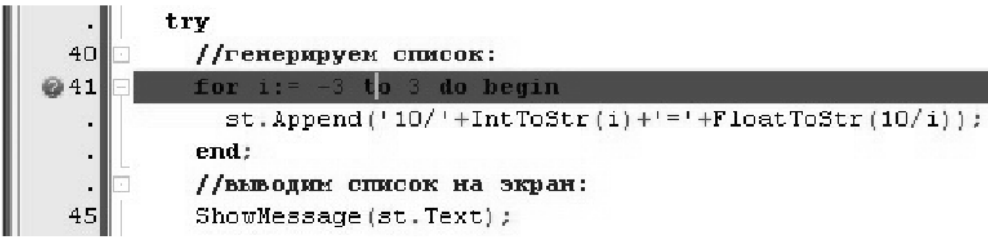


Рис. 27.4. Строка с точкой останова

Снять точку останова удобней также последними двумя способами. Точка останова у нас есть, снова нажимаем кнопку "Запустить". Программа начинает свою работу, нажимаем кнопку "Button1".

Теперь программа не вывела ошибку, а приостановила свою работу и вывела на передний план Редактор кодов с выделенной серым цветом строкой, которая в данный момент готовится к выполнению:

```
  .   try
 40     //генерируем список:
 41     for i:= -3 to 3 do begin
  .       st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
  .     end;
```

Рис. 27.5. Строка, которая будет выполнена далее

Тут очень важно понимать, что программа была остановлена ДО выполнения этой строки, а не ПОСЛЕ неё. То есть, в настоящий момент переменной `i` еще не присвоено значение -3. Далее, мы можем выполнять с отладчиком различные действия, которые собраны в разделе главного меню "Запуск". Обычно требуется пошаговое выполнение программы. Для этого можно использовать команду "Запуск -> Шаг в обход" (или <F8>), "Запуск -> Шаг с входом" (или <F7>) или "Запуск -> Шаг с выходом" (или <Shift+F8>). "Шаг в обход" означает, что если в коде будет встречен вызов какой-нибудь функции или процедуры, отладчик выполнит их и остановится на следующей после вызова строке. При выборе "Шаг со входом", отладчик также пошагово будет выполнять и вызываемые функции-процедуры. "Шаг с выходом" подразумевает, что если в строке нет вызовов функций, то остановки происходят, как при "Шаг в обход". Если в строке есть выражение, то остановка происходит вначале перед строкой, затем перед вычислением каждой функции, чтобы мы имели возможность просмотреть значения параметров, передаваемых в функцию.

У нас вызовов функций нет, поэтому мы можем воспользоваться как <F7>, так и <F8> (чаще всего используют <F8> - Шаг в обход).

Итак, нажмем <F8>, и отладчик выполнит строку с точкой останова, и выделит серым следующую строку. Снова нажмем <F8>, и снова будет выделена эта строка - был выполнен шаг цикла. Нажав несколько раз <F8>, мы добьемся появления на экране всё той же ошибки, которая заблокирует дальнейшее выполнение программы. Становится понятно, что цикл нормально выполняется несколько проходов, после чего всё же возникает ошибка. Включаем логику: внутри цикла у нас изменяется только переменная i, значит, ошибка как-то связана с ней. А как узнать,

как именно?

Здесь нам на помощь приходит еще один полезный инструмент отладчика - наблюдение за значениями переменных. Сбросьте программу командой "Запуск -> Сбросить отладчик". Теперь снова нажмите кнопку "Запустить", а потом снова кнопку "Button1". Отладчик снова приостановил выполнение программы на строчке с циклом, однако не спешите нажимать <F8>. Для начала, добавим наблюдение над переменной i. Делается это командой "Запуск -> Добавить наблюдение", которая была недоступна, пока программа не начала выполняться. В строке "Выражение" укажите переменную i, и нажмите "OK":

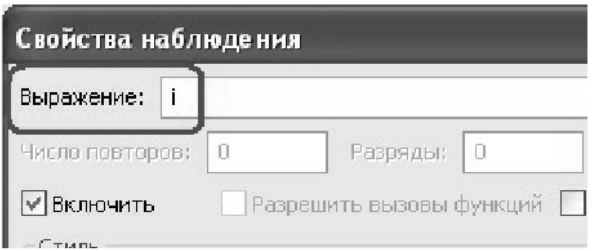


Рис. 27.6. Установка наблюдения за переменной

Теперь отладчик наблюдает за значениями переменной `i`, но нам от этого не легче - мы-то не видим этих значений! Чтобы их увидеть, нужно вывести на экран окно Списка наблюдений. Делается это командой "Вид -> Окна отладки -> Окно наблюдений" или "горячими клавишами" <Ctrl+Alt+W>.

Расположите окно ниже Редактор кода, на место Окна сообщений. В этом окне вы сможете видеть выражение - нашу переменную `i`, и её текущее значение. Чтобы показать работу с отладчиком более наглядно, давайте добавим еще одно выражение, за которым будем наблюдать. Выберите "Запуск -> Добавить наблюдение" и в строке "Выражение" укажите не просто переменную, а выражение

```
10/i
```

которое у нас должно вычисляться внутри цикла. В окне Списка

наблюдения вы увидите и переменную `i`, и выражение, а также их значения:

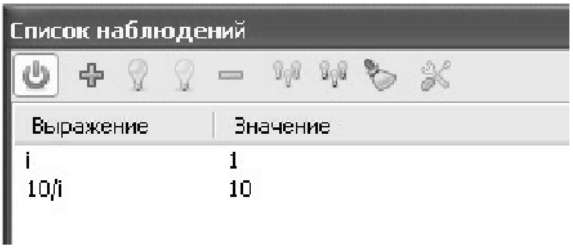


Рис. 27.7. Окно Списка наблюдений

Поскольку переменной `i` еще не было присвоено значения -3, то в колонке значений вы, скорее всего, увидите 1, которым по умолчанию была проинициализирована наша переменная. Соответственное значение будет и у выражения. Теперь мы готовы двигаться дальше. Нажимаем <F8>. В Списке наблюдений сразу же изменилась картина - `i` теперь равно -3, а выражение -3,3333…

Нажимаем <F8> ещё раз. Снова значения изменились, теперь `i = -2`, а выражение = -5. Мы понимаем, что цикл работает, и два его шага были сделаны. Нажимаем <F8> еще два раза. Сейчас переменная содержит ноль, а значение выражения указывает "`inf`". Однако строка с вычислением еще не была выполнена, не забываем об этом. Снова нажимаем <F8>, и снова получаем ошибку. А в значениях переменной и выражения видим слово "`evaluating`", что переводится, как "оценка". Теперь мы наглядно видим, что в строке

```
st.Append('10/'+IntToStr(i)+'='+FloatToStr(10/i));
```

возникает ошибка, когда переменная `i` равна нулю. И тут уже несложно догадаться, почему эта ошибка возникает - потому что происходит попытка деления 10 на 0.

Это можно проверить, пропустив выполнение вычисления, когда `i=0`. Закройте окно с ошибкой, сбросьте отладчик. Снова нажмите кнопку "Запустить", и кнопку "Button1". Снова выведите окно Списка наблюдения. Нажимайте <F8>, пока `i` не станет 0, а выражение - `inf`.

Теперь, в Окне наблюдений щелкните правой кнопкой мыши по строке с переменной `i`, и в всплывающем меню выберите команду "Вычислить/Изменить". В открывшемся окне вы увидите строку "Выражение", где будет указана переменная `i`. В поле "Результат" будет указано значение 0. А в строке "Новое значение" нам нужно указать значение, которое мы желаем принудительно присвоить переменной. Тут укажем 1 и нажмем <Enter> или кнопку "Изменить". В поле "Результат" значение должно смениться на 1. Снова нажмем <F8>, значение `i` изменится на 2, ошибки не будет. Нажав <F8> еще несколько раз, мы доберемся до конца программы и увидим сообщение, которое она и должна была вывести по нашему замыслу:

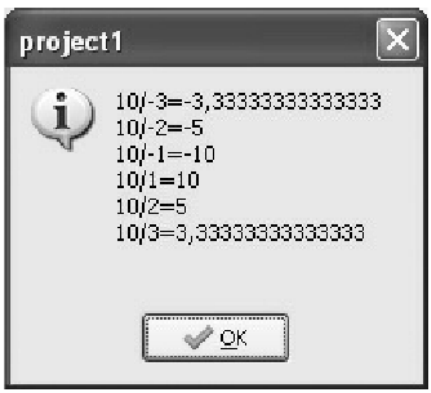


Рис. 27.8. Результирующее сообщение программы

Как видите, вычисление, где `i` равна нулю, было пропущено.

Встроенный отладчик имеет и другие инструменты, с которыми вы сами сможете со временем освоиться, экспериментируя с ними.

Создание справочной системы

Лекция носит факультативный характер. Мы рассматриваем все этапы разработки справочной системы, осваиваем программы NVU и MS HTML Help Workshop, разрабатываем справочную систему, подключаем его к проекту Lazarus.

Цель лекции

На лекции осваиваются приемы разработки справочной системы и подключения этой системы к проекту Lazarus.

Создание справочной системы

Любой серьезный проект должен быть снабжен подробной справочной системой, позволяющей пользователю разобраться с тем, как работает ваша программа. Ведь если бы не "хэлпы", мы с вами вряд ли смогли самостоятельно освоить какую-нибудь сложную программу. И в этой лекции мы научимся создавать справочные системы и подключать их к проекту. Напомню, что хоть Lazarus и является кросс-платформенной средой разработки программ, мы осваиваем работу с ним под управлением ОС Windows. Если вы пользуетесь Lazarus в среде Linux, Unix или другой не Windows ОС, то рекомендации из этой лекции вам вряд ли подойдут. В этом случае, вам придется поискать данный материал самостоятельно.

Итак, справка может быть двух форматов: *.hlp и *.chm. Оба формата разработаны корпорацией Microsoft. Формат HLP является устаревшим, хотя и встречается до сих пор, но в настоящий момент наиболее распространенным форматом справки в среде Windows является CHM, его и рассмотрим.

CHM (Microsoft Compressed HTML Help, Microsoft Compiled HTML Help, CHM) - формат файлов контекстной справки, разработанной Microsoft в 1997 году в качестве замены формата WinHelp - HLP. Файл CHM является сжатым набором HTML-страниц, своего рода, электронной книгой, и может содержать весь набор Web-страниц: текст, таблицы, рисунки, ссылки, медиа-файлы, и прочее, что делает такую справочную

систему мощным инструментом. Все входящие в CHM-справку файлы сжаты алгоритмом LZX, что делает справку более компактной.

В Интернете я нашел упоминание, что в состав Lazarus входит инструмент для создания CHM-файлов. И в самом деле, я нашел этот инструмент по адресу (папки установки Lazarus у меня создавались по умолчанию):

C:\lazarus\tools\chmmaker

По этому адресу я нашел проект, который по идее, нужно скомпилировать, и получить готовый инструмент. Впрочем, достаточно взглянуть на окно проекта, чтобы понять, насколько данный инструмент примитивен. Говорить серьезно о создании с его помощью справочной системы не приходится. А значит, нам нужно найти другой инструмент.

Этот инструмент должен быть:

- бесплатным;
- достаточно мощным для создания справок любой сложности.

И тут на ум сразу приходят две программы: HTM2CHM и Microsoft HTML Help Workshop. Первая написана русским программистом Ярославом Кирилловым, обладает русским и достаточно дружелюбным интерфейсом. Функционально она проигрывает Microsoft HTML Help Workshop. Кроме того, компилируемые программой CHM-файлы получаются несколько большего размера. Однако если требуется простая небольшая справка, то лучшего инструмента вам не найти. Мы не будем рассматривать программу HTM2CHM, поскольку она достаточно проста, вам нетрудно будет освоить её самостоятельно. Рассмотрим более сложный инструмент - Microsoft HTML Help Workshop. Как видно из названия, программа разработана корпорацией Microsoft, является бесплатным, свободно распространяемым продуктом. Скачать её можно с сайта производителя, я нашел её по адресу: ссылка: http://www.microsoft.com/en-us/download/details.aspx?id=21138

Или же вы можете скачать этот инструмент с моего сайта, который задумывался для поддержки этого курса: скачать: http://old.intuit.ru/department/se/plazarus/28/htmlhelp.rar

Это HTML Help Workshop версии 4.74.8702 - программа для создания справочной системы.

HTML Help Workshop (hhw.exe) - программа, обладающая достаточно большими возможностями, хотя и непростая в эксплуатации, и не имеет русского интерфейса. Программа позволяет создавать все необходимые файлы справки, из которых затем компилируется единый проект.

Давайте вообще, разберемся, что представляет собой CHM-справка. По сути, это электронная интерактивная книга, позволяющая использовать содержание, индексы (тэги), полнотекстовый контекстный поиск, избранные статьи справки. И всё это создается на основе отдельных HTML-файлов. И вот теперь мы подходим к другому вопросу: что такое HTML? HTML - Hyper Text Markup Language - Язык разметки гипертекста, основа, на которой строятся все Web-сайты, те самые веб-странички, которые мы загружаем в свои браузеры. Получается, чтобы создавать справочную систему, требуется изучить ещё какой то HTML? В общем, да, если вы будете использовать только встроенные средства HTML Help Workshop. Веб-страничку можно написать в любом текстовом редакторе, хоть в Блокноте, если вы знаете разметку HTML. Однако конечно, есть и более простые пути.

Существует множество визуальных редакторов, которые позволяют сохранять содержимое в виде HTML-файлов. В первую очередь приходит на ум офисная программа MS Word. Да, она также позволяет создавать HTML-файлы, однако не советую её использовать: внутри таких файлов очень много лишних тегов - мусора. Кроме того, ссылки на изображения в HTML-файлах должны быть представлены тегом `img`, например:

```
<img src="logo.png">
```

а MS Word вставляет изображения в коллекцию `Shapes`.

Другим вариантом в Интернете нередко советуют программу из того же офиса MS FrontPage. Что ж, в первых версиях эта программа действительно, была хороша: достаточно простая, позволяла создавать веб-странички визуально, не требовала от пользователя знаний HTML. Однако со временем, FrontPage (как и другая продукция Microsoft) всё

более усложнялась, в неё добавлялись всё новые, неизвестно кому нужные функции, и теперь MS FrontPage - неповоротливый монстр, для изучения которого потребуется достаточно увесистый учебник.

Есть и третий, более удачный вариант. На просторах Интернета существует много визуальных HTML-редакторов, не требующих знаний HTML. Есть и платные программы, и бесплатные. Есть довольно сложные системы, вроде Macromedia Dreamweaver, позволяющие строить целые сайты, есть программы попроще.

Для создания HTML-файлов я весьма рекомендую простой и бесплатный визуальный HTML-редактор с русским интерфейсом NVU. Этот редактор вы можете скачать с сайта разработчика: ссылка: http://nvu.mozilla-russia.org/

или с моего сайта: ссылка: http://lazarusprog.tk/files/nvu.rar

Это NVU 1.0 - Визуальный HTML редактор (WYSIWYG-редактор, от англ. What You See Is What You Get, "что видишь, то и получишь"). Этот редактор не требует знаний HTML, достаточно прост в освоении и имеет подробную справку на русском языке:

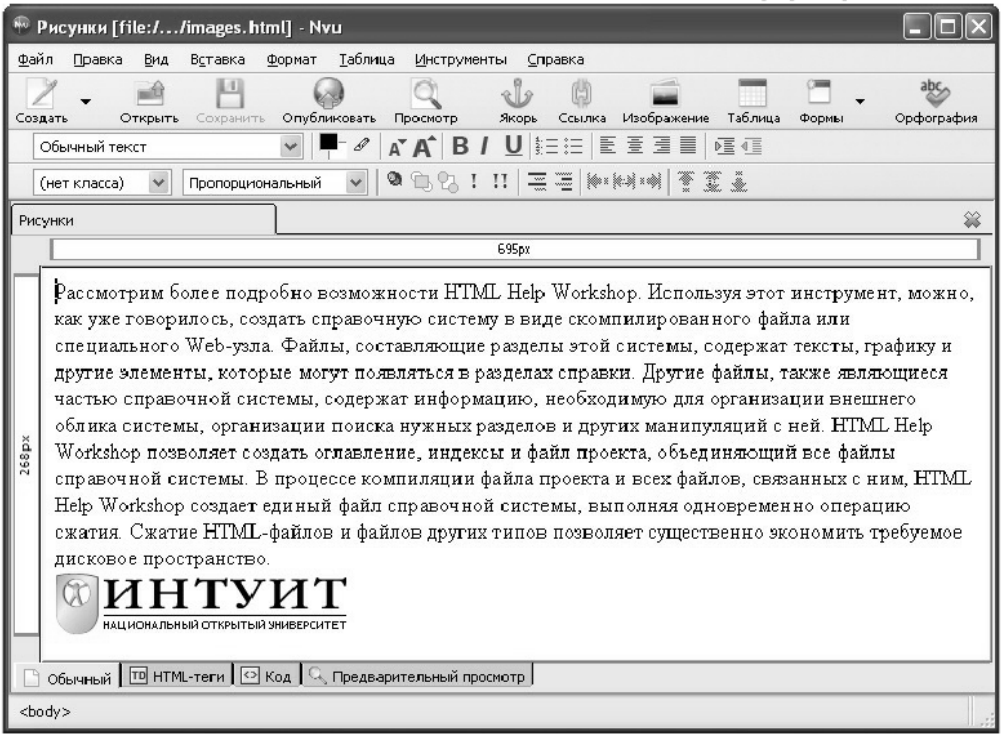


Рис. 28.1. HTML-редактор Nvu

Рассматривать создание веб-страниц в этом редакторе мы не будем, там всё достаточно просто и прозрачно, а лекция и без того получится большой. Замечу только, что имена HTML-файлам лучше давать латинскими символами и не использовать пробелы, то есть, вместо "Команды меню.html" файл будет лучше назвать "CommandsMenu.html". Дело в том, что если вы будете размещать в каких то страницах руководства ссылки на другие страницы справочника, то HTML Help Workshop может дать неверные результаты с именами файлов на русском языке. Пробелы в имени файла тоже могут сослужить плохую службу.

Теперь вернемся к нашей справке. Надеюсь, вы сохранили проект Блокнота-шифровальщика из 18-й лекции? Для этого проекта мы и будем создавать справочную систему. Прежде всего, нам нужно разработать структуру будущей справки. Содержимое справки представляет собой древовидную структуру, в которой могут быть книги и страницы. Книга в HTML Help Workshop - это раздел, который имеет

подразделы, такие книги могут быть вложенными. Страница - это конечный раздел, который не имеет подразделов. Наша программа не настолько сложная, поэтому и структура будет достаточно простой:

- Назначение программы
- Команды меню
  - Файл
  - Правка
  - Формат
  - Кодирование
  - Справка
- Обратная связь

Здесь мы имеем две страницы: "Назначение программы" и "Обратная связь", а также книгу "Команды меню", которая в свою очередь, содержит ещё 5 страниц: "Файл", "Правка", "Формат", "Кодирование" и "Справка".

Таким образом, нам придется сделать 8 HTML-файлов, отдельно на книгу и на каждую из страниц. Я не буду приводить полные листинги этих файлов - содержимое справки будет зависеть от вашей фантазии, но некоторые показательные теги я приведу. Вот, например, как я указывал ссылки на существующие страницы руководства:

```
<body>
В программе CodeBook имеются следующие пункты меню:<br>
<ul>
  <li><a href="File.html">Файл</a></li>
  <li><a href="Pravka.html">Правка</a></li>
  <li><a href="Format.html">Формат</a></li>
  <li><a href="Coder.html">Кодирование</a></li>
  <li><a href="Spravka.html">Справка</a></li>
</ul>
</body>
```

Рис. 28.2. Демонстрация ссылок на другие страницы руководства

Код я привёл в виде изображения, чтобы теги примера не конфликтовали с тегами самой лекции. Причем текст HTML-файла

открыт в простом Блокноте, чтобы увидеть теги в NVU, нужно открыть вкладку "Код" в нижней части рабочего окна.

А так у меня вставляется изображение:

```
<br>
<img style="width: 540px; height: 307px;" alt="Вид редактора с зашифрованным текстом"
title="Зашифрованный текст" src="MyText.JPG"><br>
<span style="font-style: italic;"><br>
```

Рис. 28.3. Демонстрация ссылок на изображение

В результате должна получиться примерно такая страничка со вставленным изображением:

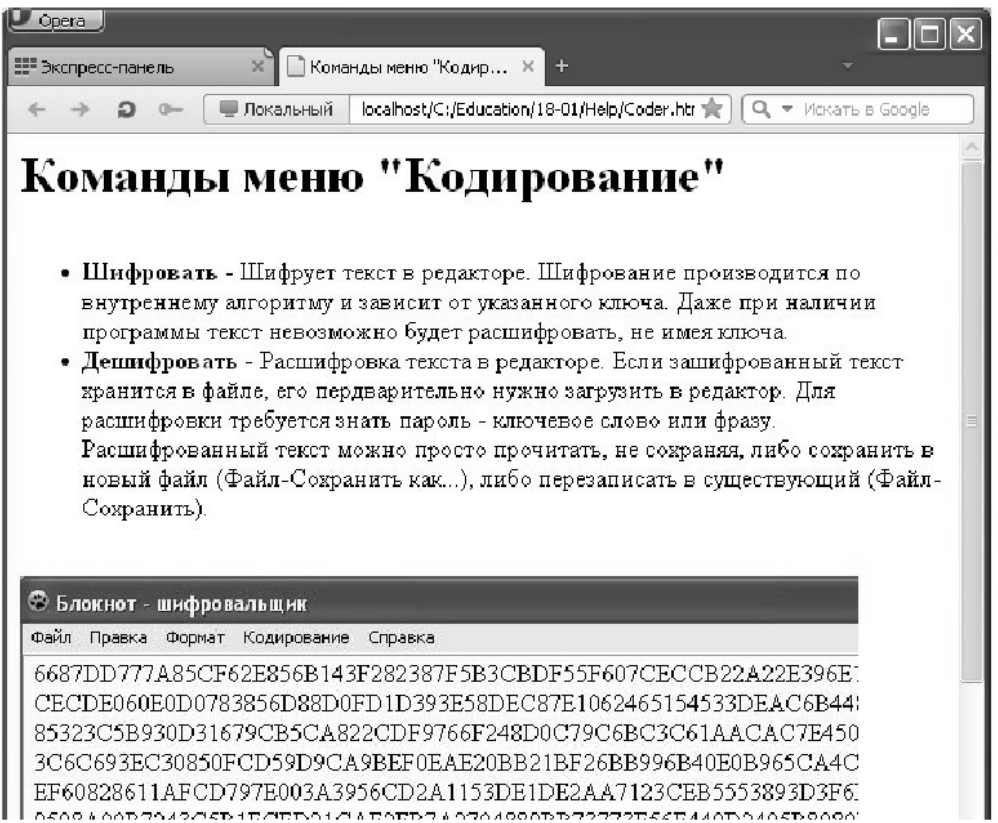


Рис. 28.4. Внешний вид странички с изображением

Надеюсь, смысл вы уловили, и правильно сделали все 8 html-файлов (у меня есть ещё 9-й файл - изображение MyText.jpg).

Наш Блокнот-шифровальщик находится в папке 18-01, если вы следовали моим рекомендациям. Создадим в этой папке еще одну папку Help, где будем собирать нашу справочную систему. У меня получился такой адрес:

C:\Education\18-01\Help

Я поместил туда 8 html-файлов:

- Coder.html
- CommandsMenu.html
- File.html
- Format.html
- Naznachenie.html
- ObratSvyaz.html
- Pravka.html
- Spravka.html

и картинку MyText.JPG, которую я вставлял в страничке Coder.html (см. рис. 28.4).

Теперь все приготовления сделаны, можно загружать программу HTML Help Workshop и заняться, наконец, созданием CHM-файла. Загрузив программу, выберите команду меню "File -> New", или щелкните кнопку "New" на Панели управления. Перед вами появится следующее окошко:

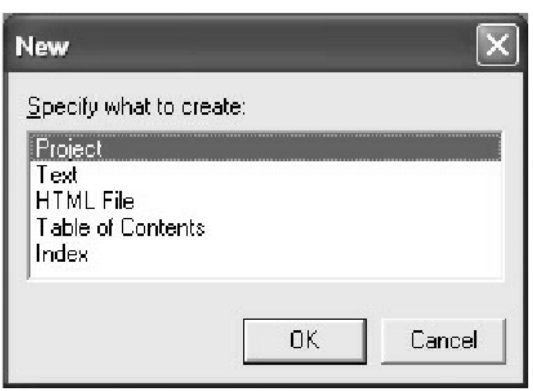


Рис. 28.5. Окно выбора объекта в HTML Help Workshop

Итак, здесь вы можете выбрать для создания следующие объекты:

- Project - Проект в целом. Это нам выбирать еще рано, сначала нужно будет сделать Таблицу Содержания.
- Text - Простой текстовый файл.
- HTML File - Файл HTML, веб-страничка. Однако, как уже говорилось, возможности этого редактора весьма скромны, и чтобы им пользоваться, требуются знания гипертекстовой разметки.
- Table of Contents - Таблица Содержания, файл формата HHC (Hyper Help Contents). Здесь откроется редактор, в котором нужно будет ввести содержание контекстной справки, точно такое же, как разработанная нами ранее древовидная структура. И нужно будет связать каждую книгу и страницу со своим HTML-файлом.
- Index - Создание индексного файла, или файла указателей. В этом файле собираются ключевые слова, фразы, по которым затем строится индекс. Сейчас в Интернете такие ключевые слова нередко называют тэгами. Файл имеет формат HHK (Hyper Help Keyword).

Создание Таблицы содержания

Как говорилось выше, вначале нужно будет сделать Таблицу содержания, поэтому выбирайте объект Table of Contents.

В окне HTML Help Workshop будет открыт объект TOC (Table Of Contents):

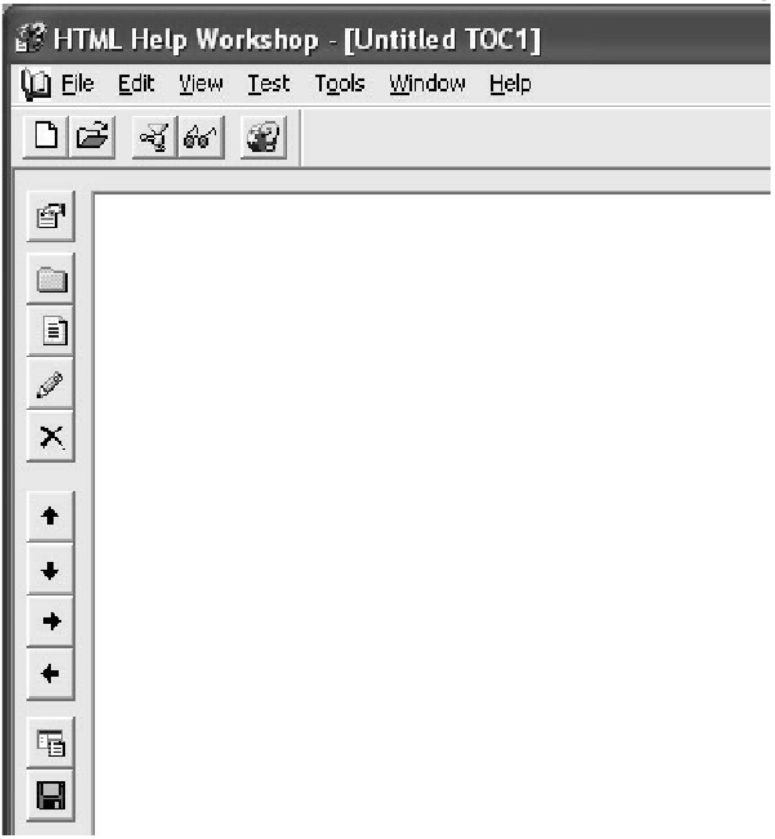


Рис. 28.6. Таблица содержания в HTML Help Workshop

Значимыми для нас являются некоторые кнопки на левой боковой панели инструментов. Самая верхняя кнопка - Свойства таблицы содержания. Две следующих кнопки позволяют добавить в таблицу книгу или страницу, это то, что нам сейчас понадобится.

Нажмите кнопку "Insert a page" (названия кнопок появляются, если некоторое время подержать над ними указатель мыши) для добавления страницы. В поле "Entry title" требуется указать то название раздела, которое будет отображаться в таблице. У нас это будет "Назначение программы" (без кавычек, конечно). Затем нужно нажать кнопку "Add/Edit", чтобы указать соответствующий разделу HTML-файл. Откроется еще одно окно "Path or URL", в котором требуется ввести имя HTML-файла или его Интернет-адрес. Можно воспользоваться кнопкой "Browse" для выбора файла, но тогда файл будет указан вместе с его относительным адресом, например, так:

..\..\Education\18-01\Help\Naznachenie.html

У нас с вами все файлы проекта справочника будут находиться в одной папке, поэтому относительный адрес тут не нужен. Удалите его, оставив в строке "File or URL" только имя соответствующего разделу HTML-файла, у меня это:

Naznachenie.html

Нажмите кнопку "OK", и тогда окно добавления раздела Таблицы содержания будет выглядеть так:

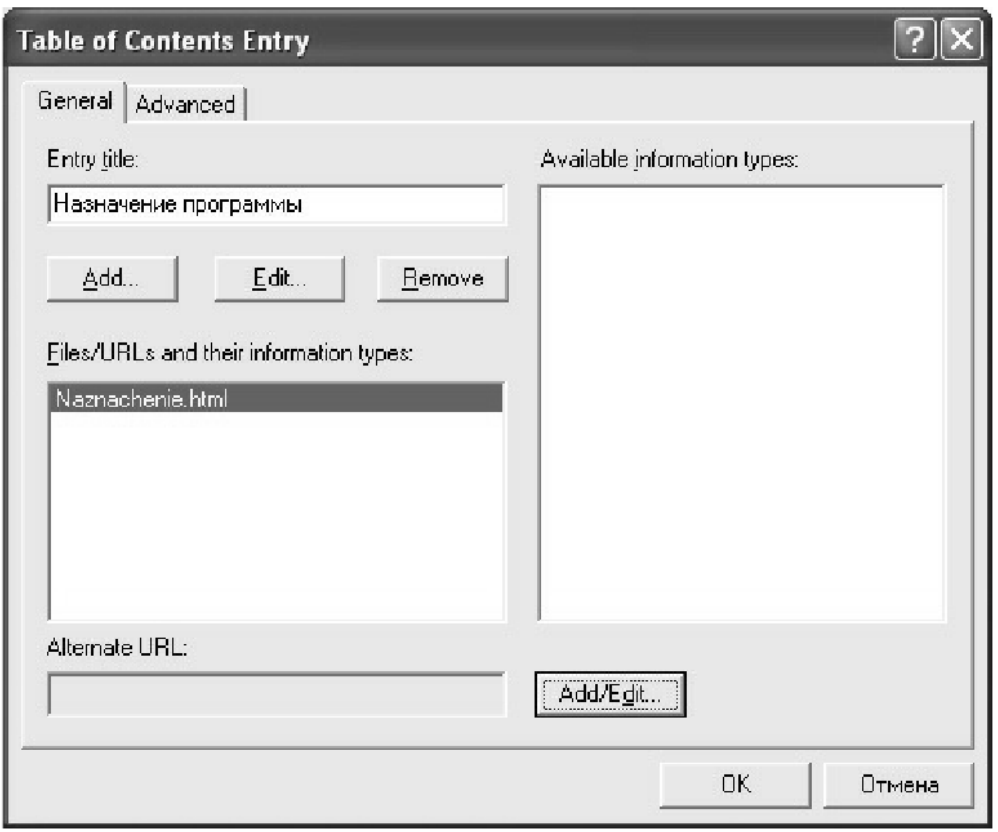


Рис. 28.7. Добавление раздела в Таблицу содержания

Нажмите "OK", и раздел будет добавлен в Таблицу содержания.

Теперь нажмите кнопку "Insert a heading", потому что в этот раз нам

нужно добавить не страницу, а книгу "Команды меню". На запрос уверены ли вы, что желаете добавить книгу в таблицу содержания, ответьте утвердительно. Процесс добавления книги абсолютно такой же, как и страницы. Укажите название раздела "Команды меню" и укажите HTML-файл, соответствующий этому разделу. Не забудьте удалить относительный адрес, если вы воспользовались кнопкой "Browse" для выбора файла.

Когда вы укажите всё, что нужно, и подтвердите все запросы, книга будет добавлена в Таблицу содержания, но окажется ВЫШЕ предыдущей страницы! А в нашей древовидной структуре она должна быть во второй строчке. Тут на помощь приходят четыре кнопки со стрелками. Первые две позволяют перемещать объект выше и ниже, а вторые - влево и вправо. Таким образом, мы можем не только установить раздел на нужную строчку Таблицы, но и передвинуть его на нужную позицию древовидной структуры. Ведь в третьей строчке у нас должна быть страница "Файл", которая будет вложенной в книгу "Команды меню", а стало быть, должна быть сдвинута вправо.

Проделав эту операцию нужное количество раз, добавьте все остальные страницы в Таблицу содержания:

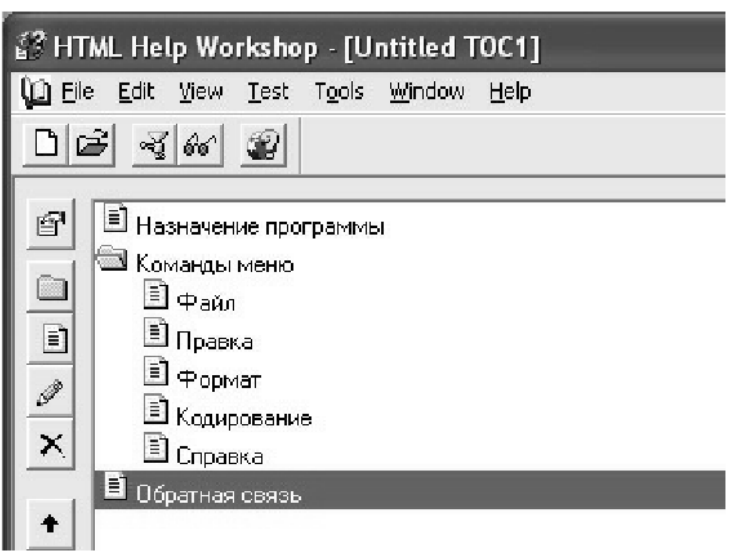


Рис. 28.8. Разделы Таблицы содержания

Не забывайте убирать относительный адрес HTML-страниц.

Когда закончите работу, нажмите нижнюю кнопку "Save file" на левой панели инструментов для сохранения Таблицы содержания. Имя таблицы можете оставить без изменения, "Table of Contents.hhc", но убедитесь, что сохраняете его в ту же папку, где уже находятся HTML-файлы справки. Впрочем, эта самая папка уже должна быть указана в окне сохранения Таблицы, если вы добавляли HTML-страницы кнопкой "Browse".

Создание файла проекта

Командой главного меню "File -> Close all" закройте созданную ранее Таблицу разделов. Затем выберите команду "File -> New" или нажмите кнопку "New" на Панели инструментов, чтобы начать создание нового объекта. В этот раз мы создаем Проект, поэтому в окне выбора объекта укажите Project.

Появится окно мастера создания проекта, и вначале вам будет предложено конвертировать существующий проект устаревшего формата WinHelp. Мы с вами создаём проект "с нуля", поэтому флажок "Convert WinHelp project" не устанавливаем, а сразу нажимаем "Далее".

На следующем шаге вам будет предложено ввести имя файла проекта или выбрать существующий файл. Тут проще всего нажать кнопку "Browse", убедиться, что открыта папка "Help", в которой у нас хранятся все файлы проекта справки, в поле "Имя файла" указать имя файла проекта. Тут есть одно замечание: результирующая справка будет иметь такое же имя, как и файл проекта, а имена справочных файлов обычно совпадают с именами программы. Файл программы у нас называется CodeBook.exe, значит, вписываем имя CodeBook и нажимаем "Открыть". Вы вернетесь в окно с именем файла, это имя будет указано вместе с адресом, а имя файла проекта автоматически получит расширение HHP (Hyper Help Project). У меня получилось так:

C:\Education\18-01\Help\CodeBook.hhp

Нажимаем "Далее", и откроется следующее окно мастера. Тут нам будет предложено указать, какими файлами проекта мы уже располагаем. У нас с вами есть HTML-файлы с разделами и файл Таблицы содержания, устанавливаем соответствующие флажки:

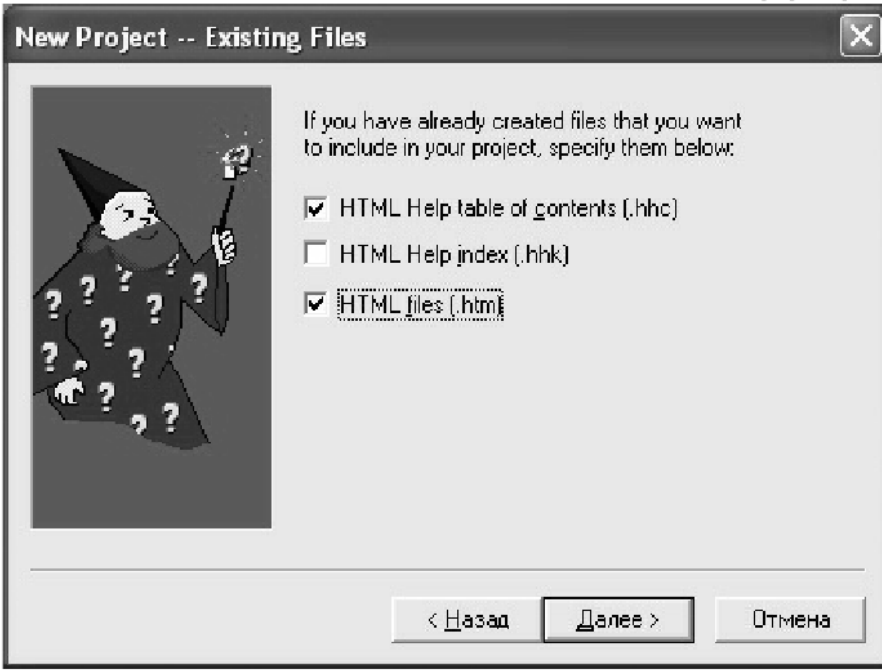


Рис. 28.9. Окно мастера создания проекта - указание существующих объектов

Нажимаем "Далее". На следующем шаге будет предложено указать файл Таблицы содержания. Нажмите "Browse" и выберите файл таблицы (Table of Contents.hhc). Нажимаем "Открыть", адрес и имя файла Таблицы попадет в поле мастера. Нажимаем "Далее".

На следующем шаге нам предлагается ввести все HTML-файлы проекта. Можно добавлять их по очереди, а лучше все файлы разом. Нажмите кнопку "Add", откроется окно открытия файлов. По умолчанию, должна быть открыта папка "Help". В окне вы увидите 8 HTML-файлов. Выделите первый из них, затем, удерживая кнопку <Shift>, щелкните по последнему. Все 8 файлов окажутся выделенными. Нажмите "Открыть", и все эти файлы попадут в окно мастера:

Рис. 28.10. Окно мастера создания проекта - указание существующих HTML-файлов

Нажимаем кнопку "Далее", затем "Готово". Всё, проект создан.

Создание индексов

Наша справка почти готова, мы уже можем скомпилировать результирующий файл справки, однако не мешает добавить в неё некоторые дополнительные функции. Индекса то у нас еще нет!

Индекс - это файл с расширением HHK (Hyper Help Keyword), содержащий ключевые слова и фразы, и ссылки на страницы справки, к которым они относятся.

Сейчас в Интернете много сайтов - библиотек, блогов, новостных сайтов и т.п. - где вы можете найти информацию по так называемым тегам. Вы же видели поле, куда вводят искомые слова и фразы, и кнопку "Найти" рядом? Те ключевые слова, которые вы туда вводите, в сфере Интернета называются тегами, а в проекте справки - индексами.

Одна страничка справки может иметь множество ключевых слов. С другой стороны, одно ключевое слово может ссылаться на множество страничек справки. Грамотная организация индексов сильно облегчит конечному пользователю работу со справкой, тут всё зависит от вашей фантазии и знаний психологии пользователя.

В окне HTML Help Workshop открыт файл проекта, в левой части окна вы видите три вкладки: "Project", "Contents" и "Index". Чтобы начать создание индексов, нужно перейти на вкладку "Index". Поскольку индексный файл в проекте еще не указан, то будет выведен запрос - создаем ли мы новый индексный файл, или подключаем существующий? Мы создаем новый файл, поэтому оставьте выделенной радиокнопку "Create a new index file" и нажмите "OK". Вам будет предложено ввести имя индексного файла, причем по умолчанию будет предложено имя Index.hhk. Не вижу особого смысла менять это имя. Убедитесь только, что в поле "Папка" по-прежнему открыта наша папка "Help", индексный файл нам нужно сохранять тоже в неё. Нажмите "Сохранить", и индексный файл будет создан.

Пока еще он пустой - индексов нет. Займемся их созданием. Вообще, следует продумать заранее, какие ключевые слова к какому разделу справки привязать. У меня получилось так:

Таблица 28.1. Примерный список ключевых слов и фраз для индексного файла

Файл и название раздела	Ключевые слова
Coder.html Команды меню "Кодирование"	кодирование, декодирование, шифровать, дешифровать, расшифровать, текст
CommandsMenu.html Команды меню	пункты меню, файл, правка, формат, кодирование, справка
File.html Команды меню	файл, работа с файлом, создать, открыть, сохранить, статистика, выход

"Файл"	сохранить, статистика, выход
Format.html Команды меню "Формат"	формат, шрифт, цвет, перенос по словам, автоматический перенос текста
Naznachenie.html Назначение программы	CodeBook, блокнот-шифровальщик, блокнот, пароль, ключевое слово, версия
ObratSvyaz.html Обратная связь	обратная связь, сайт, e-mail, электронный ящик, автор
Pravka.html Команды меню "Правка"	правка, отменить, вырезать, копировать, вставить, удалить, выделить всё, буфер обмена

У вас имена файлов и названия разделов могут отличаться. Кроме того, возможно, ваша фантазия подскажет, как расширить список ключевых слов.

Со списком разобрались, приступаем к созданию индексов. В окне HTML Help Workshop должна быть открыта вкладка "Index". Обратите внимание на кнопки левой боковой панели инструментов, они почти такие же, как у Таблицы содержания. Нас интересует вторая кнопка "Insert a keyword", на ней изображен ключ.

Нажмите на неё. Откроется окно добавления индекса. Ключевое слово или фразу нужно вписать в поле "Keyword", затем нажать кнопку "Add". Будет открыто окно, в котором нужно выбрать раздел, на который будет ссылаться данное ключевое слово. Если это слово встречается в разных разделах, то придется создавать несколько индексов: с одним ключевым словом, но разными ссылками на разделы. Для больших справочных

Итак, ключевое слово указали, раздел справки в поле "HTML titles" выбрали - поля "File or URL" и "Title" должны заполниться автоматически. Нажмите "OK", и новый индекс появится в окне добавления индекса:

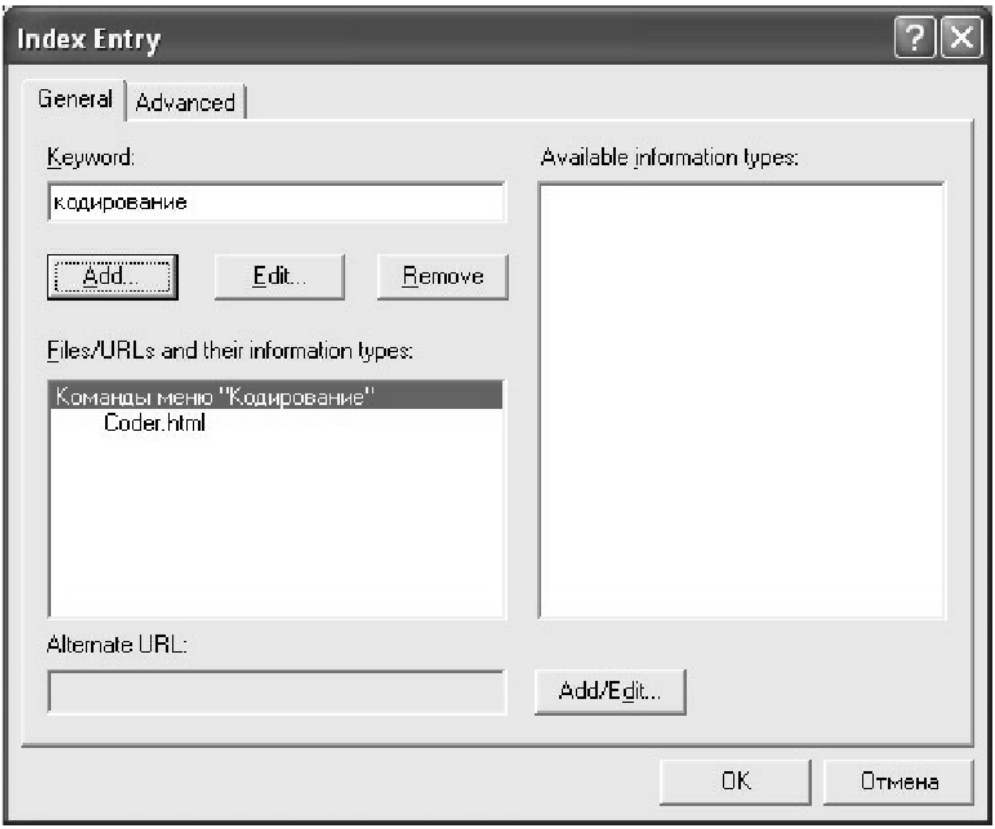


Рис. 28.11. Окно добавления ключевого слова

Ещё раз нажмите "OK", и на вкладке "Index" появится ключевое слово "кодирование". Ничего сложного, кроме того, что подобную операцию придется повторять многократно, для каждой пары "Ключевое слово - Ссылка на раздел". Так что запаситесь терпением, и введите все ваши ключевые слова и фразы.

Каждый раз вам будет выводиться запрос - следует ли создавать новый индекс на верху списка - отвечайте утвердительно, мы этот список отсортируем потом. Если вы случайно ошиблись, можете нажать третью сверху кнопку с изображением карандаша для редактирования

индекса, или четвертую - для удаления, только этот ошибочный индекс должен быть выделен.

Не знаю, как у вас, а у меня получилось 46 ключевых слов и фраз! Не мешало бы рассортировать их в алфавитном порядке. Для этого служит третья снизу кнопка "Sort keywords alphabetically" с буквами A и Z и стрелкой вниз - нажмите её, и список будет отсортирован. Если вам доведется работать с большой справкой, индексы можно создавать не за один подход, а добавлять их время от времени. Но потом всё же не забывайте их снова сортировать, иначе индексы будут выглядеть неряшливо.

Всё, работу с индексами закончили, нажмите нижнюю кнопку "Save file" для сохранения индексного файла.

Создание полнотекстового поиска

Индексы могут существенно облегчить пользователю поиск нужной справки, однако всё же полнотекстовый поиск для этого используют чаще. Полнотекстовый поиск - это когда пользователь вписывает искомое слово/фразу, а справочная система ищет это слово/фразу по всей справке. Индексы при этом не используются, то есть, найти можно любое слово, не только проиндексированное.

За удобство использования полнотекстового поиска приходится платить некоторым замедлением поиска, однако на современных ПК это замедление почти незаметно.

Включить полнотекстовый поиск очень просто. В окне HTML Help Workshop откройте вкладку "Project". Самая верхняя кнопка левой боковой панели инструментов - "Change project options" (Изменить параметры проекта). Нажмите её, перейдите на вкладу "Compiler" и включите флажок "Compile full-text search information":

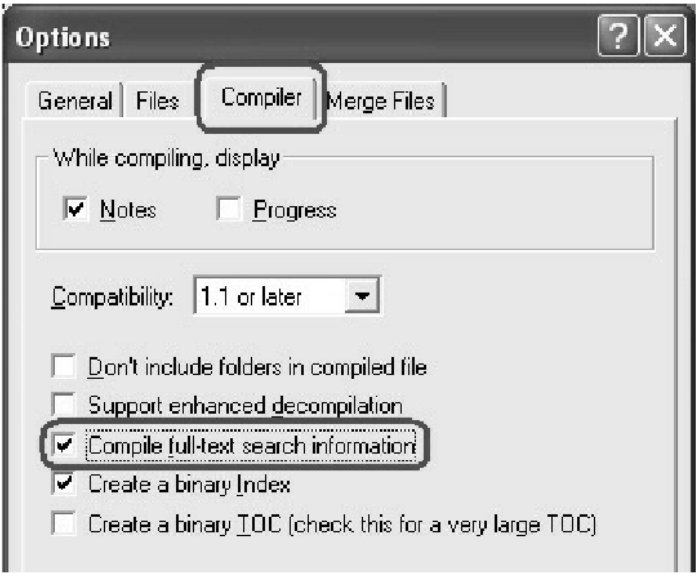


Рис. 28.12. Окно изменения параметров проекта

Кроме того, стоит прямо сейчас указать, какая страница справки будет открываться по умолчанию. Для этого перейдите на вкладку "General", и в поле "Default file" выберите нужный файл. Я, например, выбрал Naznachenie.html. Не забудьте нажать кнопку "OK", чтобы подтвердить изменения.

Создание вкладки "Избранное"

Вкладка "Избранное" в справочной системе может послужить достойным завершающим штрихом. В эту вкладку пользователь будет иметь возможность добавлять те разделы справки, к которым чаще всего обращается. Создается эта вкладка тоже достаточно просто.

На вкладке "Project" HTML Help Workshop, на левой боковой панели инструментов, третья кнопка сверху - кнопка "Add/Modify window definitions" (Добавить/Изменить определение окна). Справка ведь выводится в окне, мы своё окно пока не создавали, пользовались тем, что создается по умолчанию, а потому к настройкам окна доступа не имеем. Вот нам и нужно создать своё окно справки. Нажмите на эту кнопку. Вам будет предложено ввести имя нового окна. Имя может быть любое, например, Favorites.

Как только вы введете имя и нажмете "OK", откроется окно Window Types с настройками нашего нового окна. Для добавления вкладки "Избранное" нам нужно перейти на вкладку "Navigation Pane" и включить флажок "Favorites tab":

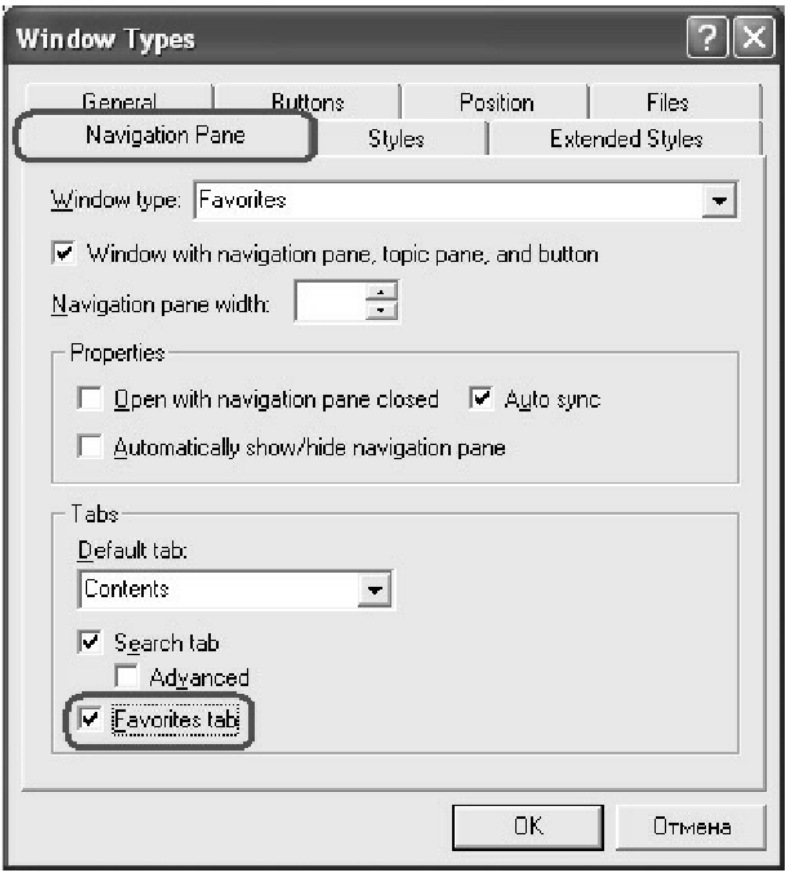


Рис. 28.13. Включение вкладки Избранное

Кроме того, перейдите на вкладку "Files" и в поле "Default" также укажите файл справки, который будет открываться по умолчанию.

Нажмите "OK", и на этом наша работа с проектом завершена. Не забудьте сохранить проект командой меню "File -> Save project" или кнопкой с изображением дискетки.

Компиляция справки

Наш проект закончен, осталось скомпилировать саму справку - файл CodeBook.chm. Делается это очень просто: выберите команду главного меню "File -> Compile…". Вам будет предложено выбрать файл проекта, для которого компилируется справка, по умолчанию будет открыт наш проект

C:\Education\18-01\Help\CodeBook.hhp

Это нам и нужно. Рекомендую включить флажок "Save all files before compiling" (Сохранить все файлы перед компиляцией). После чего можете нажать кнопку "Compile".

Если вы всё сделали правильно, в правой части окна выйдет отчет о проделанной компиляции:

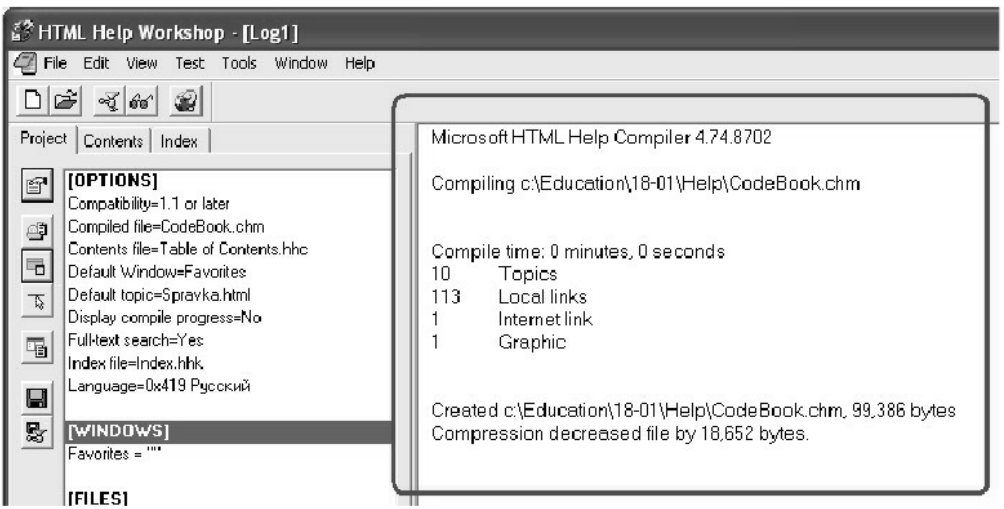


Рис. 28.14. Отчет о компиляции

Как видите, ошибок в отчете нет, указано количество тем, локальных и Интернет-ссылок, изображений. Правда, не всегда эти количества соответствуют действительности, но тут уж ничего не поделать.

Также указано, что был создан файл CodeBook.chm размером 99,386 байт. На этом работа с проектом завершена, можете закрыть HTML Help Workshop.

Затем можете в Проводником или файловым менеджером открыть

папку Help с проектом справки, и запустить файл CodeBook.chm прямо оттуда:

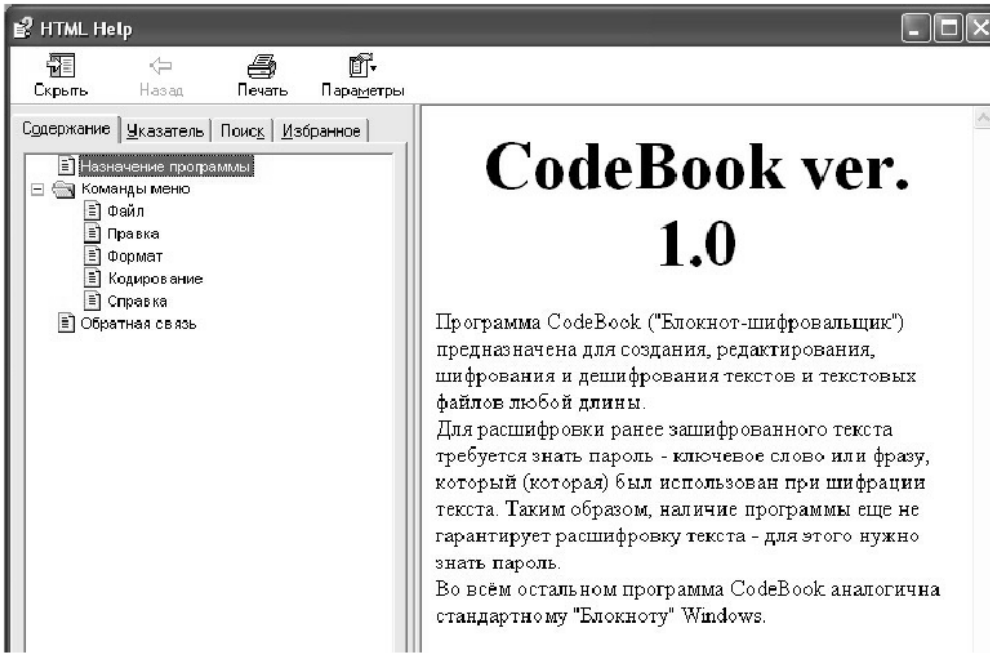


Рис. 28.15. Работающая справочная система

Как видите, справка открывается, на ней присутствуют все 4 вкладки: "Содержание", "Указатель", "Поиск" и "Избранное". Можете поэкспериментировать с этими вкладками.

Есть пару замечаний. Первое: для показа этой справки система задействует инструменты программы MS Internet Explorer, но на каждой Windows эта программа установлена, так что беспокоиться тут нечего. Второе: справка самостоятельно запоминает последнее положение и размеры, которые установил пользователь. При последующем запуске справки эти положение и размеры будут восстановлены, что сильно облегчает нам работу.

Включение справки в проект Lazarus

Надо сказать, механизм подключения справочной системы к проекту в Lazarus ещё сыроват, и информации по этому вопросу немного. Одни

советуют использовать несколько десятков строк кода, другие говорят, что это не помогает… В официальной документации (на английском языке) Lazarus советуют использовать компоненты `THTMLHelpDatabase` и `THTMLBrowserHelpViewer` с вкладки System. В качестве примера предлагают проект

C:\lazarus\examples\helphtml

Этот пример, почему то, выводит справку в браузере, установленном в системе по умолчанию. Затем мне всё же удалось запустить справку, однако справка открылась в окне просмотра Lazarus, а не системы. Это окно, как оказалось, с кириллицей "не дружит" - вместо русских букв выходили прямоугольнички.

В конце концов, мне удалось найти достаточно простой способ подключить к проекту справочную систему, о нем я вам сейчас и расскажу.

Прежде всего, скопируйте файл справки CodeBook.chm в папку с проектом программы. Можно было бы оставить его и в папке Help, особой разницы, откуда запускать справочник нет. Однако когда все файлы программы находятся в одной папке, то эту программу проще подготовить к распространению.

Далее, откройте проект программы в Lazarus - запустите файл CodeBook.lpi из папки с проектом, у меня это C:\Education\18-01. Откроется Lazarus с проектом программы. В Редакторе кода убедитесь, что открыта вкладка "Main" - вкладка главного окна. Перейдите на редактор форм. Откройте редактор главного меню (дважды щелкните по компоненту `MainMenu1`).

Нам нужно добавить еще один элемент в раздел меню "Справка". Выделите этот раздел, выделите его элемент "О программе", щелкните правой кнопкой мыши и выберите команду "Вставить новый пункт (перед)". Появится новый пункт меню. Выделите его, свойство `Name` переименуйте в `HelpHelp`, в свойстве `Caption` напишите `Справка`, а в свойстве `ShortCut` выберите горячую клавишу <F1>.

Сгенерируйте для этого пункта меню событие `OnClick`, код которого будет довольно простой:

```
procedure TfMain.HelpHelpClick(Sender: TObject);
begin
  //вызываем справку:
  WinExec('hh.exe CodeBook.chm',SW_SHOW);
end;
```

Процедура `WinExec` вызывает внешнее системное приложение hh.exe (системный просмотрщик справок) с загруженным файлом справки CodeBook.chm. Однако WinExec сразу же работать не будет - эта процедура прописана в модуле Windows, который нужно добавить в раздел `uses`. Добавьте, сохраните проект, перекомпилируйте его и запустите. Нажмите <F1> или выберите команду меню "Справка -> Справка" - наша справка должна загрузиться.

Если вы захотите распространять эту программу, вам для этого будет достаточно двух файлов: CodeBook.exe (сама программа) и CodeBook.chm (справка).

Создание инсталлятора

Лекция носит факультативный характер. Здесь мы рассмотрим создание инсталляционного файла для нашей программы, изучим работу с системой Inno Setup.

Цель лекции

На лекции осваиваются приемы разработки инсталляционного файла нашей программы.

Зачем нужны инсталляторы

Наш курс подходит к концу. Мы написали множество проектов, научились делать справку и подключать её к программе, но хотелось бы ещё какие-то из проектов опубликовать в Сети! А что для этого требуется? Установочный файл. Кстати, если вы будете писать программы на заказ, то создание инсталлятора почти наверняка будет обязательным условием проекта.

Казалось бы, можно обойтись каким-нибудь популярным архиватором вроде WinRAR, сделать на нем самораспаковывающийся SFX-архив, прописать, какой файл программы куда должен быть распакован. Однако так вы получите лишь примитивный инсталлятор, в котором пользователю не предоставляется никаких возможностей выбора, а это плохо - даже если пользователь при установке программ обычно использует все настройки "по умолчанию", нажимая только кнопки "Далее" или "Next", ему нравится осознавать, что от него что-то зависит, что он имеет возможность выбора.

Предположим, ваш проект достаточно сложен, содержит десятки файлов. Вы желаете, чтобы пользователь имел выбор способа установки вашей программы: куда устанавливать, нужно ли создавать ярлыки на рабочем столе и в панели быстрого запуска? Устанавливать ли полную версию, выборочную или минимальную? Регистрировать ли программу в реестре или создать INI-файл? Кроме того, пользователь может когда-нибудь захотеть эту программу удалить со своего ПК, а значит, нужно предусмотреть и обратный процесс - деинсталляцию.

Все эти операции сделать в обычном архиваторе будет невозможно, или, по крайней мере, чрезвычайно сложно. Зато эти же операции легко создаются в специальных программах-инсталляторах, которые входят в группу утилит программиста. О них и будем говорить на этой лекции.

Обзор инсталляторов

Некоторые среды разработки программ имеют в своем составе инсталляторы. Так, в состав Delphi входит бесплатный инсталлятор Install Shield Express, который, кстати, можно скачать с сайта производителя. Считается, что если в состав IDE входит инсталлятор, то для создания инсталляционного файла (или CD) лучше пользоваться им. Install Shield Express, например, имеет очень удобную функцию включения в инсталляционный пакет "движка" баз данных BDE, который, впрочем, уже устарел, и теперь практически не используется. Ничего другого выдающегося у Install Shield Express не наблюдается, поэтому даже Delphi-программисты предпочитают более "продвинутые" инсталляторы.

Lazarus вообще не имеет в своем составе подобной утилиты, однако существует множество инсталляторов сторонних разработчиков - как платных, так и бесплатных, с помощью которых Lazarus-программист может создать инсталляционный файл или даже загрузочный CD/DVD.

Какие ещё инсталляторы существуют? Их множество: Setup Factory, Visual Studio Installer, Nullsoft Install System, Smart Install Maker и много-много других.

Особо хотелось бы выделить один из самых популярных бесплатных инсталляторов, обладающих широкими возможностями - Inno Setup. Это система создания инсталляторов для Windows-программ. Inno Setup появился в 1997 году и является не просто бесплатным, но и открытым проектом (Open Source), превосходит многие коммерческие установщики по функциональности и стабильности. Мы будем рассматривать создание инсталляторов на основе Inno Setup.

Inno Setup можно скачать с сайта производителя, официальная страничка: ссылка: http://www.jrsoftware.org/isinfo.php

Или же, вы можете скачать эту программу с моего сайта: скачать: http://old.intuit.ru/department/se/plazarus/29/isetup-5.5.4.rar

Следует заметить, что при написании курса рассматривалась версия Inno Setup 5.5.4, а они достаточно часто обновляются. Так что на сайте производителя, возможно, вы найдете более свежую версию.

Файл лицензии

Обычно при установке какой либо программы, пользователю вначале предлагается прочитать и принять лицензионное соглашение (лицензию) программы. Пользователь читает (или чаще, не читает) лицензию, устанавливает радиокнопку с надписью "Я принимаю условия" (или что-то аналогичное), после чего становится доступной кнопка "Далее". Наверное, наш инсталлятор должен также выводить какую то лицензию? Где её взять, или как её сделать, не будучи юристом?

Тут совет простой: найдите схожую с вашей программу, и используйте текст её лицензии, переделав лишь название программы, копирайта (того, кому принадлежит авторское право), номер версии, и т.п. То есть, если вы собираетесь опубликовать в Интернете вашу программу, как бесплатную (freeware), то и лицензию копируйте у другой бесплатной программы. Поскольку все лицензии имеют примерно одинаковые предлагаемые условия и используют одинаковую терминологию, то ничего страшного в данной хитрости нет. Создайте обычный текстовый файл с текстом лицензии, например, в стандартном Блокноте. Мы рассмотрим создание инсталлятора для нашего Блокнота-шифровальщика из лабораторной работы лекции № 18, поэтому и файл с лицензионным соглашением должен быть об этой программе. Назовите файл License.txt (так обычно называют файлы с лицензией), а текст этого файла может быть примерно таким:

"CodeBook" (Блокнот-шифровальщик)

Copyright (c) 2014, В.Ю. Ачкасов

Версия 1.0 (freeware)

До установки и/или использования этого программного обеспечения, внимательно ознакомьтесь с условиями этого лицензионного соглашения и ограниченной гарантии.

ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ

АВТОРСКОЕ ПРАВО

Программное обеспечение "CodeBook", включая документацию и другие дополнительные материалы, защищено законами и международными соглашениями об авторских правах, а также другими законами и договорами, регулирующими отношения авторского права. Данное программное обеспечение лицензируется, а не продается.

Всеми авторскими правами на программу "CodeBook" владеет только её автор - Вячеслав Юрьевич Ачкасов. Все права, не предоставленные здесь явно, сохраняются за автором. Автор оставляет за собой право отменить действие данной лицензии в любой из следующих версий программы.

ИСПОЛЬЗОВАНИЕ И РАСПРОСТРАНЕНИЕ:

Текущая версия "CodeBook", документация и сопутствующие файлы могут свободно использоваться, а также копироваться, передаваться и распространяться любыми способами при соблюдении следующих условий:

- Файлы должны распространяться в неизменном исходном виде. Это должен быть исходный архив, сформированный автором. Не допускается взимание платы за распространение "CodeBook" без письменного разрешения автора.
- "CodeBook" не может быть продан или перепродан, включен в состав другого программного обеспечения, использоваться в службе поддержки коммерческого предприятия, или использоваться для иного способа получения прибыли, без письменного разрешения автора.
- При цитировании, перепечатке или другом использовании любых фрагментов документации или файлов, входящих в "CodeBook", ссылка

на автора и источник цитирования обязательна.

- Вы не можете использовать, копировать, эмулировать, создавать новые версии, сдавать в наем или аренду, продавать, изменять, передавать программу или любые из ее составляющих, иначе, чем определено настоящим лицензионным соглашением.

- Вы не можете декомпилировать, дизассемблировать, изучать код программы другими способами, кроме случаев, когда такая деятельность явно разрешается законом, несмотря на это ограничение.

- Любое такое нелегальное использование означает автоматическое и немедленное прекращение действия настоящего соглашения и может преследоваться по закону.

ГАРАНТИИ

"CodeBook" распространяется по принципу "КАК ЕСТЬ". Любые последствия, финансовые и другие потери, могущие наступить вследствие использования "CodeBook" самой по себе и/или совместно с другим программным обеспечением не могут являться поводом для предъявления каких-либо претензий к автору и официально уполномоченным распространителям.

Установка и/или использование "CodeBook" означает, что вы понимаете положения настоящего лицензионного соглашения и согласны с ними. Если почему-либо вы не согласны с этим лицензионным соглашением, вы не имеете права устанавливать или использовать данное программное обеспечение и вам необходимо удалить файлы дистрибутива "CodeBook" с ваших устройств хранения информации и прекратить их использование.

Copyright (c) 2014, В.Ю. Ачкасов

Разумеется, вы можете изменить этот текст - добавить или удалить какие либо разделы, изменить фамилию правообладателя на свою, добавить собственные данные для обратной связи (e-mail, ваш сайт). Но в целом, текст файла с лицензионным соглашением должен выглядеть примерно так.

Создание инсталлятора в Inno Setup

Мы рассмотрим самый простой случай создания инсталлятора. Буквально, в несколько кликов мыши. Однако возможности Inno Setup куда шире. При необходимости, вы освоите их самостоятельно с помощью достаточно подробной справки (правда, на английском языке, хотя в Интернете можно найти и русифицированные варианты).

Итак, для создания инсталлятора нам прежде всего потребуется создать папку с файлами программы, пусть это будет

C:\CodeBook

В эту папку нам нужно будет скопировать три файла:

1. CodeBook.exe - Исполняемый файл программы
2. CodeBook.chm - Файл справочной системы
3. License.txt - Файл с лицензионным соглашением

Конечно, количество и состав файлов зависит от проекта - некоторые проекты могут иметь десятки, а то и сотни файлов. Нам же для данного проекта достаточно будет этих трех файлов.

Загружаем Inno Setup. Выходит следующее окно приветствия:

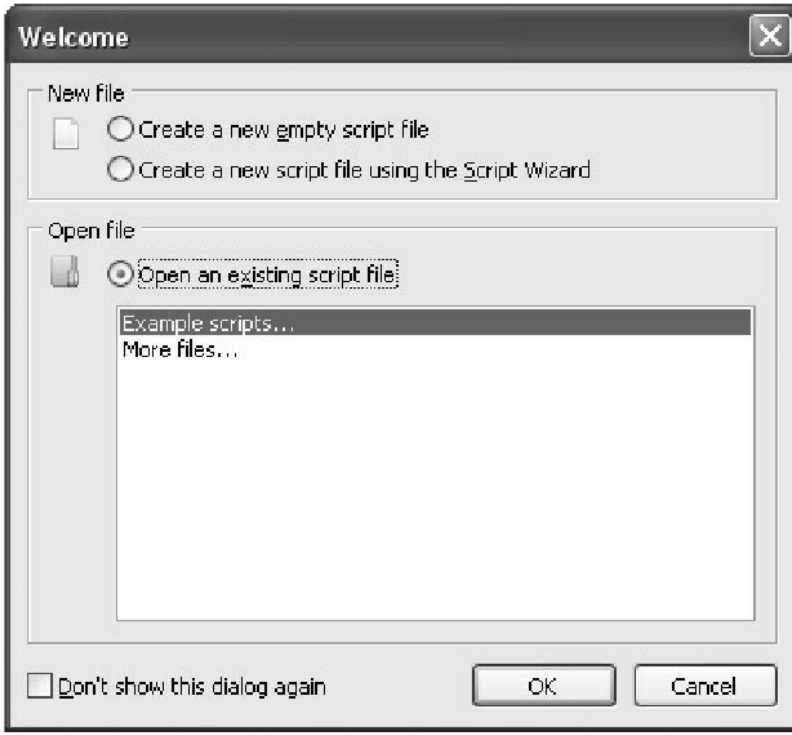


Рис. 29.1. Начальное окно Inno Setup

Здесь мы можем выбрать следующие пункты:

В секции "New file":

- Create a new empty script file - Создание нового пустого скриптового файла. Дело в том, что каждый проект создания инсталлятора в Inno Setup - это набор скриптов, внутренних команд, которые описывают сценарий создания инсталлятора и используемые настройки. Если вы изучите эти скрипты, то сможете создавать инсталляторы любой сложности. Но на данном этапе нам это не нужно.
- Create a new script file using the Script Wizard - Создание нового скриптового файла с использованием Мастера Сценариев. Это то, что нам сейчас нужно, включите эту радиокнопку.

Есть еще раздел "Open file" с пунктом Open an existing script file - Открыть и использовать существующий файл сценариев. Этот пункт может вам

пригодиться, если вы захотите изменить существующий сценарий.

Итак, включаем Create a new script file using the Script Wizard и нажимаем "OK". На следующем шаге будет выведено приветствие Мастера сценариев, тут просто нажимаем кнопку "Next". Флажок Create a new empty script file включать не нужно.

Далее выйдет окно с основной информацией о приложении:

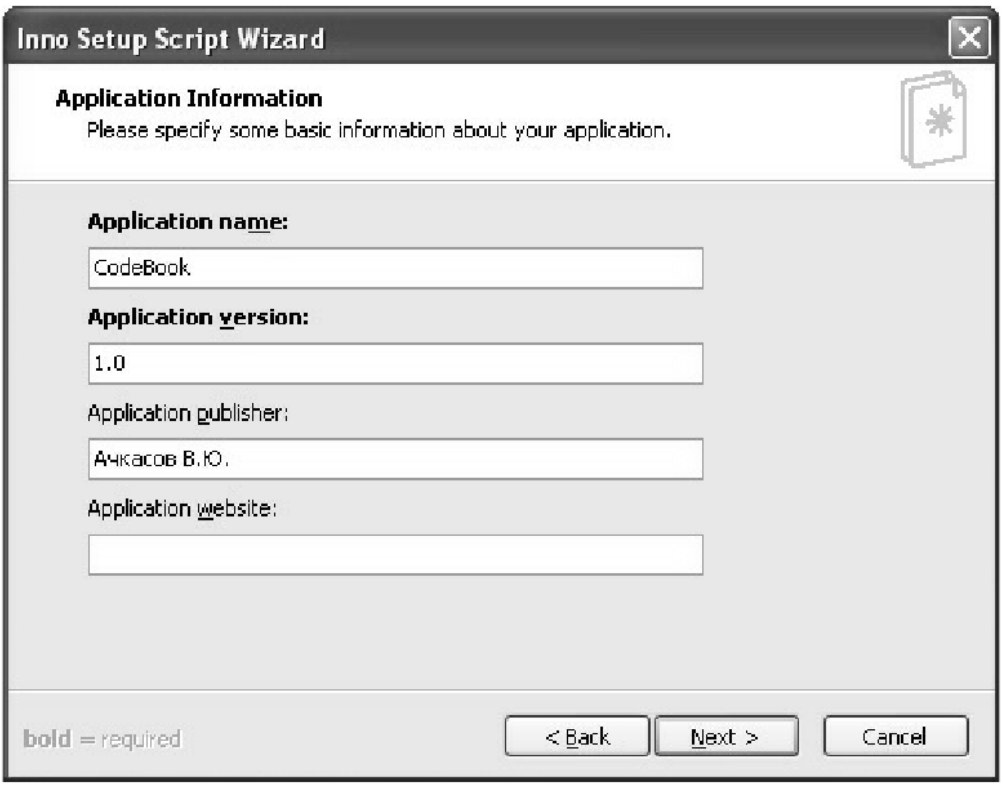


Рис. 29.2. Окно основных настроек

Здесь нам нужно указать следующие настройки:

-
- Application name - Название приложения, в нашем случае CodeBook.
- Application version - Версия приложения, в нашем случае 1.0.
- Application publisher - Кому принадлежит авторское право, тут

можете указать свою Фамилию, Имя и Отчество.

- Application website - Официальный сайт программы. Если такого сайта нет, оставляем поле пустым.

Нажимаем "Next". Далее будет выведено окно папки проекта:

Рис. 29.3. Окно папки проекта

В поле "Application destination base folder" мы должны указать, куда по умолчанию будет устанавливаться наша программа. По умолчанию, это папка C:\Program Files. Так и оставим: если пользователь укажет другой путь, то программа установится туда, иначе - в C:\Program Files.

В поле "Application folder name" требуется указать папку, куда будет скопирована программа. В нашем случае получается, что полным адресом установки программы будет C:\Program Files\CodeBook.

Флажок "Allow user to change the application folder" во включенном

положении позволит пользователю изменить папку установки программы, предложенную по умолчанию. Правила хорошего тона этого требуют, так что оставляем флажок включенным.

Флажок "The application doesn't need a folder" (Это приложение не нуждается в папке) означает, что программе не требуется собственная папка, значит, она будет установлена по адресу C:\Windows. Это иногда делают для системных утилит, но наша программа в папке нуждается, так что флажок должен быть выключен.

Нажимаем "Next". Далее будет выведено окно, в котором будет предложено указать главный файл программы, а также добавить остальные файлы (и папки), которые должны быть установлены:

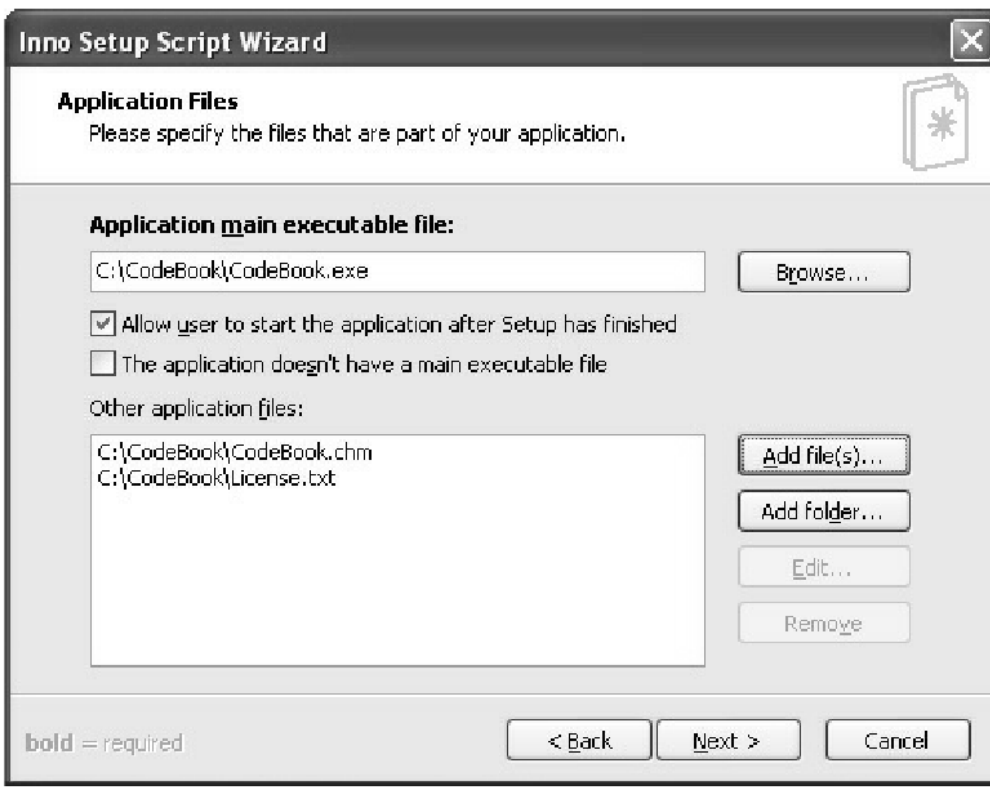


Рис. 29.4. Указание файлов (и папок) проекта

В поле "Application main executable file" нужно указать главный файл программы. Нажмите кнопку "Browse" и найдите этот файл, как на

рисунке (все файлы создаваемого инсталлятора мы поместили в папку C:\CodeBook, если вы помните).

Кнопка "Add file(s)" позволяет добавить остальные необходимые файлы. Нам нужно добавить файл справки (обязательно), и файл лицензии (если желаем, чтобы инсталлятор его тоже скопировал при установке), как на рисунке.

Кнопка "Add folder" позволяет добавлять вложенные папки в папку программы. У нас таких папок нет, так что этого мы не делаем.

Флажок "Allow user to start the application after Setup has finished" (Позволить пользователю запуск приложения после установки) оставляем включенным, в этом случае, после установки выйдет запрос о запуске приложения.

А вот флажок "The application doesn't have a main executable file" (У приложения нет главного исполняемого файла) нам включать не нужно.

Нажимаем "Next". Далее будет выведено окно папки стартового меню:

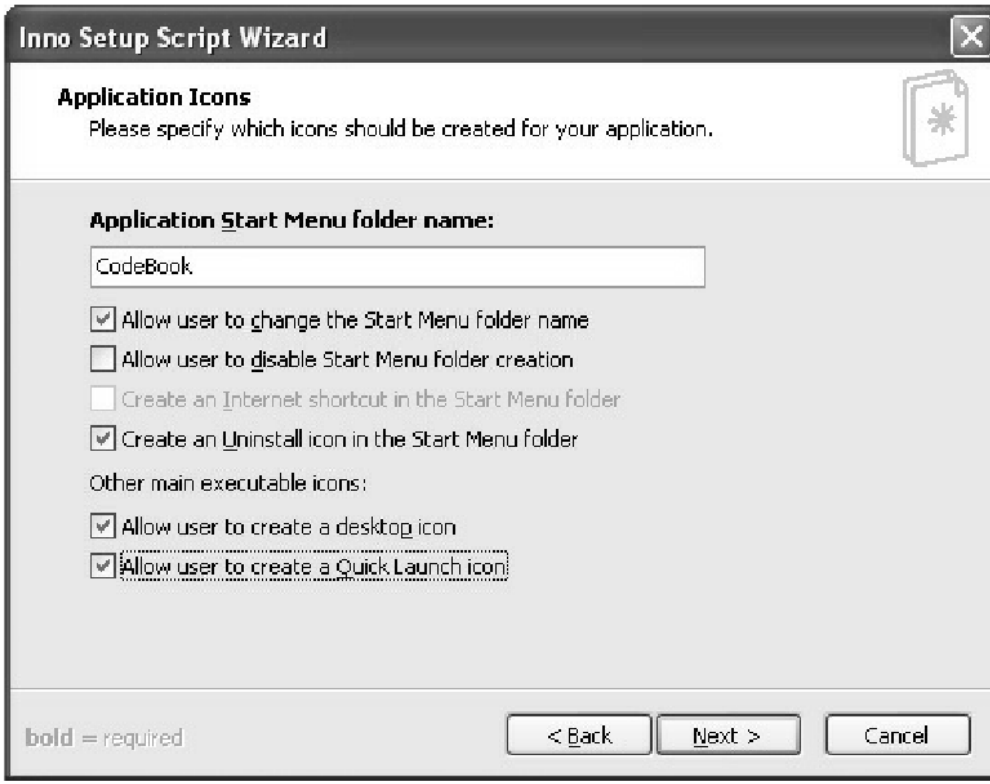


Рис. 29.5. Окно папки стартового меню программы

Здесь, в поле "Application Start Menu folder name" (Имя папки стартового меню программы) требуется ввести имя папки стартового меню, в которой будут располагаться ярлыки программы. Обычно название этой папки совпадает с названием программы, но при желании, можете его изменить.

Флажок "Allow user to change the Start Menu folder name" (Позволить пользователю изменить имя папки Стартового Меню) лучше оставить включенным, так требуют правила хорошего тона.

Флажок "Allow user to disable Start Menu folder creation" (Позволить пользователю отключить создание папки Стартового Меню) включать не нужно - мы ведь хотим, чтобы папка CodeBook присутствовала в Стартовом Меню!

Флажок "Create an Internet shortcut in the Start Menu folder" при включенном

состоянии добавляет в папку Стартового меню ссылку на официальный сайт программы, если таковой существует. Если вы помните, мы не указывали сайт программы, поэтому данный флажок не доступен. Однако если официальный сайт программы существует, и мы его указали, то данный флажок нужно будет включить, это приведет к тому, что в папку Стартового меню с программой будет добавлены ссылка на сайт программы.

Флажок "Create an Uninstall icon in the Start Menu Folder" при включенном состоянии создаст команду деинсталляции программы. Правила хорошего тона просто требуют, чтобы данный флажок был ВКЛЮЧЕН!

Последние два флажка позволяют пользователю выбрать, будут ли создаваться ярлыки программы на Рабочем столе, и на панели быстрого запуска. Оба флажка желательно включить, так пользователю будет дано больше свободы выбора.

Нажимаем "Next". Далее будет предложено указать файл(ы) с различной информацией:

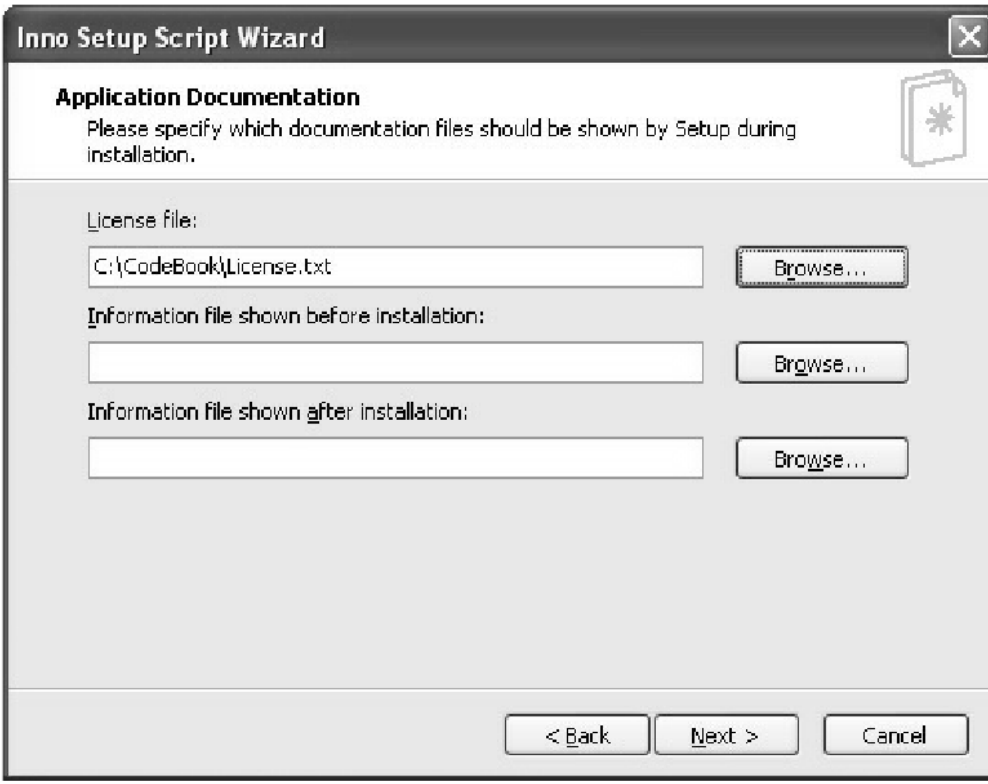


Рис. 29.6. Окно выбора файлов с дополнительной информацией

В поле "License file" предлагается указать файл с лицензией, она будет выведена в окне инсталлятора в процессе установки программы. Нажмите "Browse" и выберите наш файл лицензии, как на рисунке.

В поле "Information file shown before installation" предлагается указать текстовый файл с информацией, которая будет выведена в окне инсталлятора ПЕРЕД установкой. Если у вас есть советы об установке, которые желательно вывести пользователю, можно оформить их в виде текстового файла и указать этот файл в данном поле. У нас такой информации нет (да обычно её вообще не указывают), оставляем поле пустым.

В поле "Information file shown after installation" предлагается указать текстовый файл с информацией, которая будет выведена в окне инсталлятора ПОСЛЕ установки. Такой информации у нас также нет.

Нажимаем "Next". Нам будет предложено выбрать языки установки:

Рис. 29.7. Окно выбора языков установки

Список возможных языков впечатляет. Учитывая, что у пользователя может быть английская версия Windows, оставляем включенным английский язык, и, конечно же, включаем русский. Имейте в виду, что это язык инсталлятора программы, а не язык интерфейса самой программы! Первое окно инсталлятора в любом случае выйдет на языке операционной системы. Затем пользователь может выбрать другой язык, и следующие окна инсталлятора будет уже на нем.

Снова нажимаем "Next". В следующем окне, в поле "Custom compiler output folder" нам потребуется указать папку, в которую будет помещен результат компиляции проекта - собственно, сам инсталляционный файл. Укажем папку

C:\OutCodeBook (папка уже должна существовать):

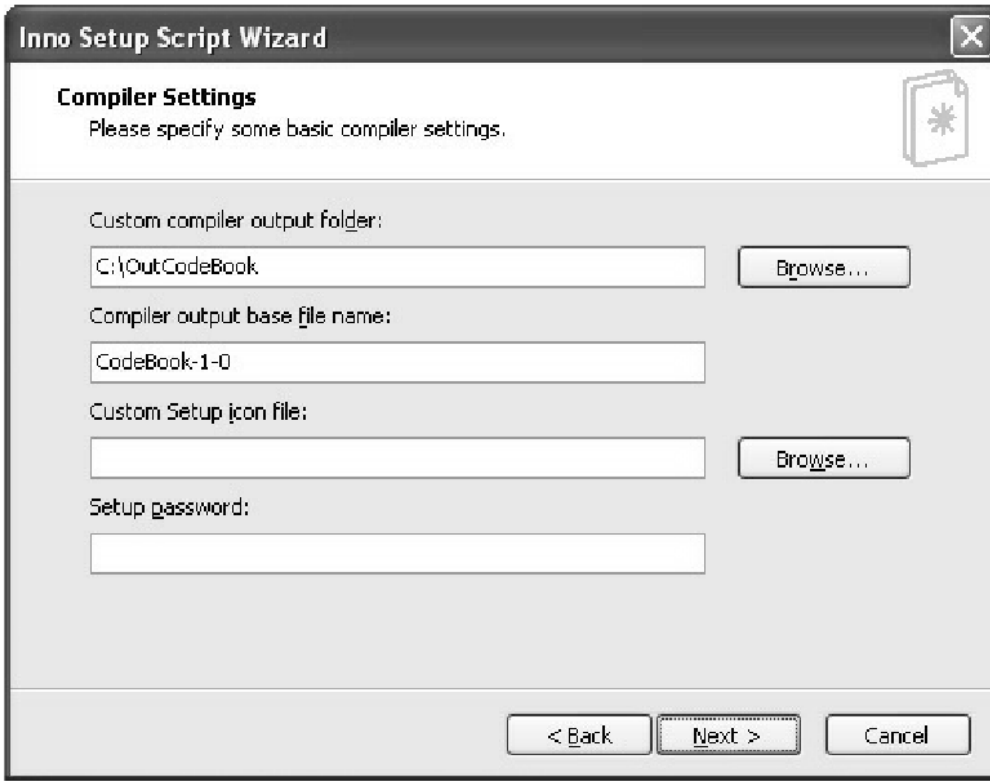


Рис. 29.8. Окно конечных параметров установки

В поле "Compiler output base file name" нужно будет ввести имя инсталляционного файла нашей программы. Можно оставить его по умолчанию - setup, можно изменить на install, но сейчас всё чаще инсталляционный файл содержит название и версию программы (как это принято в Unix-подобных ОС). У меня это имя CodeBook-1-0. Расширение *.exe будет добавлено автоматически, тут его указывать не нужно.

Если у вас есть графический ICO-файл с "иконкой", которую вы желаете применить к вашему приложению, то можете выбрать этот файл в поле "Custom Setup icon file". Если вы это сделаете, то новая "иконка" программы будет отображаться и в папке Стартового Меню, и на Рабочем столе, и в панели Быстрого запуска. У нас такого файла нет, мы его не указываем, значит, будет использована "иконка" по умолчанию - стандартная иконка Lazarus-программ - изображение синего отпечатка лапы гепарда.

И, наконец, в поле "Setup password" можно указать пароль запуска установки. Кто не знает пароль, не сможет установить программу. Эту возможность можно использовать для создания платных программ - кто заплатил за программу, тому высылаете пароль, чтобы он смог установить её. Однако не стоит возлагать на эту возможность больших надежд. Скорее всего, первый же оплативший программу пользователь выложит этот пароль в Интернете, и других оплат вы не дождетесь. Кроме того, концепция shareware подразумевает, что вы даете пользователю пробный период, например, 15 дней, после которого он либо должен заплатить за программу, либо удалить её с ПК. Не зная пароля, пользователь не сможет и опробовать программу. Так что не указываем никакого пароля (тем более, что наша программа имеет лицензию freeware, то есть, бесплатная), и нажимаем "Next".

В следующем окне запрашивается, нужно ли включать в проект #define-директивы. Это своего рода, константные имена команд. Оставляем флажок "Yes, use #define compiler directives" включенным, и нажимаем "Next" в последний раз.

Выйдет завершающее окно мастера. Нажмем кнопку "Finish", и проект инсталлятора будет создан. Однако это еще только проект, а не сам инсталлятор. После работы мастера выйдет запрос:

Рис. 29.9. Завершающий запрос: нужно ли компилировать новый скрипт сейчас?

Нажмите "Да". Далее выйдет запрос, нужно ли сохранять скрипт проекта перед компиляцией? Скрипт проекта - это обычный текстовый файл с расширением ISS, в котором хранится вся та информация, которую мы вводили в мастере. Если есть вероятность, что вы будете этот проект компилировать еще раз, имеет смысл нажать "Да" и сохранить скрипт.

Далее будет предложено ввести имя скрипта, назовем его, как и программу - CodeBook. По умолчанию, будет открыта папка с остальными файлами программы. После сохранения скрипта произойдет компиляция проекта, окно Inno Setup будет выглядеть так:

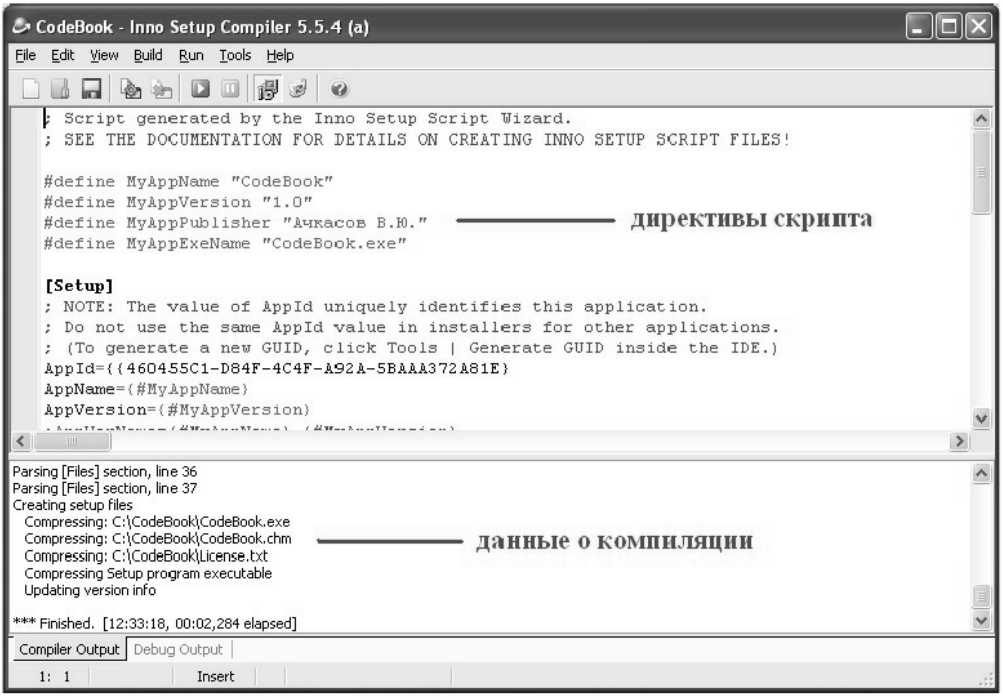


Рис. 29.10. Окно Inno Setup после компиляции

В верхней части расположены директивы скрипта - сценарий создания инсталлятора. В нижней - информация о компиляции. Теперь, если вы заглянете в выходную папку (C:\OutCodeBook), которую указывали ранее, то обнаружите там файл инсталлятора CodeBook-1-0.exe. Причем, если три файла проекта занимают 1,9 Мб, то файл инсталляции будет занимать всего то менее 900 Кб. Сжатие получилось более чем вдвое!

Теперь можете запустить инсталлятор и протестировать его работу: установите программу, посмотрите, что находится в папке C:\Program Files\CodeBook (если не изменяли папку установки), посмотрите, создалась ли папка CodeBook в Стартовом меню, какие там команды (должны быть две команды: "CodeBook" и "Деинсталлировать

CodeBook"). Удалите программу, попробуйте установить её на другом языке.

Примечание: программа для создания инсталляторов Inno Setup настолько популярна, что для неё сторонними разработчиками создаются различные расширения. Так, могу посоветовать расширение Inno Script Studio, которое является надстройкой над Inno Setup. Inno Script Studio имеет более понятный интерфейс (причем есть возможность установки русского интерфейса), упрощенную последующую работу с директивами скрипта.

Наш проект CodeBook окончательно завершен, теперь можете дарить его друзьям и одноклассникам, размещать в Интернете, применять полученные знания в других проектах. Надеюсь, данный курс был вам полезен. Удачи в программировании!

Содержание

Титульная страница	2
Выходные данные	3
Лекция 1. Введение в Lazarus	4
Лекция 2. Анатомия проекта	24
Лекция 3. Работа с компонентами	34
Лекция 4. Основы кода	57
Лекция 5. Символы и строки	72
Лекция 6. Стандартные строковые функции и сообщения	98
Лекция 7. Логические типы, конструкции и компоненты	114
Лекция 8. Числа	134
Лекция 9. Подпрограммы	147
Лекция 10. Циклы и переключатель case	167
Лекция 11. Экранная заставка	183
Лекция 12. Дата и время	193
Лекция 13. Массивы простые, многомерные и динамические	210
Лекция 14. Коллекции (массивы) строк и компоненты для них	227
Лекция 15. Диалоги	250
Лекция 16. Организация меню и панелей инструментов	266
Лекция 17. Модули	283
Лекция 18. Блокнот - шифратор	300
Лекция 19. Деревья	322
Лекция 20. Многооконные приложения. SDI- и MDI-интерфейсы	339
Лекция 21. Консольные приложения и параметры	

программы
Лекция 22. Тип TStringList. Работа с папками 364
Лекция 23. Файлы 374
Лекция 24. Записи и вариант. Сетка строк TStringGrid 395
Лекция 25. Телефонный справочник 415
Лекция 26. DLL 434
Лекция 27. Тестирование и отладка 464
Лекция 28. Создание справочной системы 476
Лекция 29. Создание инсталлятора 501