Содержание

Введение

Эта книга посвящена популярному в нашей стране и перспективному во всём мире языку программирования Delphi. Она направлена на всех программистов, от начинающего, до профессионала. В любом случае, я советую всем прочитать её полностью. Как показывает практика, большинство людей научились программированию по книгам, но ни одна из книг, которые я видел, не объясняет принципиальных основ работы Windows и компьютера в целом. Без понимания этих вещей не возможно написать эффективный код.

Я решил восполнить этот пробел. Я постараюсь написать так, чтобы, прочитав мой труд, любой человек смог стать настоящим программистом. Несмотря на это, я не гарантирую, что именно ты сможешь стать профессионалом. Как показала практика, из всех обучающихся программированию, только 30% становятся настоящими программистами и только к ним относится понятие профессионал. Я обучил достаточно много людей и у меня этот показатель свыше 70%. Оставшиеся 30% смогли научиться писать программы, смогли понять основы, но почему-то не обладают способностью самостоятельно мыслить. У них постоянно возникают вопросы, ответы на которые можно получить, затратив всего лишь небольшое усилие. Надо просто немного подумать. Но у них это не получается или не хотят. Может это лень, а может просто человеку не интересно самостоятельно мыслить.

Я могу научить многому, но без стремления самому додумывать то, что я не досказал, ты не сможешь самостоятельно писать программы. В течение всей книги я буду рассказывать различные методы программирования, напишу некоторые шаблоны, покажу некоторые приёмы и хитрости, но описать абсолютно всё я не смогу. Программирование – это такая область, в которой нужно постоянное обучение. Нельзя прочитав только одну книгу останавливаться на достигнутом. Нужно постоянно совершенствоваться и обучатся.

Прежде чем приступить к чтению самой книги, я хочу сделать несколько замечаний. Первое, я буду очень часто использовать в тексте выражение «Язык программирования Delphi». Многие утверждают, что Delphi – это среда разработки, которая использует язык программирования Pascal (Паскаль). В принципе, я не утверждаю, что это ошибка. Но всё же в Delphi от старого Паскаля осталось очень мало, поэтому я считаю, что это не просто среда разработки, это уже целый собственный язык. Это лично моё мнение и ты можешь с ним соглашаться или нет.

Теперь о содержимом книги. Я постарался описать всё так, чтобы было понятно даже человеку, который только недавно познакомился с компьютером. Возможно, продвинутым пользователям большую часть начала книги будет скучно читать, но это только в начале. Практически с самого начинала, я начну описывать специфичные вещи, среди которых можно будет найти для себя полезное, даже опытному программисту. Поверь мне, это действительно так и моё утверждение основано на уже сложившейся практике. Это связано с тем, что все книги упускают из виду некоторые очень важные тонкости, которые желательно знать для понимания принципа работы программ. Без этого понимания тяжело двигаться дальше и любые новые технологии будут казаться тяжёлыми и сложными.

Прежде чем ты приступишь к чтению книги, я тебе дам один совет. Книгу желательно читать полностью, от начала и до конца, потому что материал излагается постепенно и некоторые вещи могут быть непонятны если что-то пропустить. Как только ты почувствуешь, что набрал достаточно знаний и способен самостоятельно писать хотя бы простейшие программы, можешь сделать единственный скачок на главу «Дополнительная информация». В ней тебе необходимо прочитать, как отлаживать

приложения, потому что при самостоятельном написании программ всегда появляются ошибки/опечатки. Эта глава рассказывает, как находить такие ошибки. В ней же ты прочитаешь некоторые приёмы по работе с редактором кода, которые тебе могут пригодиться в будущем при программировании собственных приложении, да и при работе с моими примерами.

После прочтения этой главы нужно вернуться на ту главу, на которой ты остановился до этого и продолжить чтения книги без каких-либо скачков. Иначе какой-то важный момент может быть упущен и нагнать потом будет очень тяжело, потому что ты можешь не заметить, что что-то упустил.

Структура книги

В этом месте принято описывать содержание глав книги. Я поступлю так же, чтобы ты мог иметь представление, что я буду описывать в моей книге. К тому же, это поможет тебе легко найти потом интересующую тебя главу, или наоборот, узнать какие главы ты уже хорошо знаешь, и читать не стоит.

Глава 1. «Основные принципы работы компьютера». Эта глава посвящена принципам работы компьютера. В ней я постараюсь рассказать всё, что необходимо знать о том, как компьютер производит расчёты и выполняет различные команды. Эту главу я ещё не видел ни в одной из книг, которые можно купить в магазине (признаюсь, что я не много видел книг). А это же основы, без которых невозможно понимание самого принципа программирования. Конечно же, можно обойтись и без неё, потому что и обезьяну можно научить кидать гранату. Но только с помощью этих знаний, можно понять, что и зачем ты пишешь в своей программе.

В принципе, эту главу можно и опустить, потому что научится программированию можно и без этого. Но только с пониманием работы железа можно стать настоящим программистом. Поверь мне, эти знания необходимы.

Глава 2. «Машинная математика». В этой главе я постараюсь объяснить тебе основы машинной математики. Это основа программирования. Ты познакомишься с логикой выполнения программ, и сам научишься строить логику будущей программы.

Мы познакомимся с гениальным изобретением всех времён – «блок-схемы». Они очень хорошо помогают начинающим программистам в понимании работы логики компьютера. Конечно же, в будущем ты сможешь научиться писать программы без использования блок-схем, но на начальном этапе это очень удобный инструмент для построения логики будущей программы.

Глава 3. «Оболочка Delphi». В этой главе я опишу процесс установки Delphi 6, расскажу о входящих в поставку утилитах. После этого мы запустим оболочку Delphi 6 и рассмотрим, из чего она состоит. В этой главе будут в основном начальные сведения о Delphi и её могут пропустить те, кто уже знаком с Delphi. Хотя в конце главы я буду говорить о настройках оболочки, поэтому желательно всё же прочесть всем. Возможно, ты что-то и не знал.

Глава 4. «Визуальная модель Delphi». В этой главе я рассказываю про визуальную модель Delphi. На чём построена вся теория программирования в этой оболочке. А так же мы затронем теорию объектно-ориентированного программирования, без понимания которого невозможно движения дальше.

Глава 5. «Основы языка программирования в Delphi». В этой главе мы познакомимся с типами данных Delphi и напишем нашу первую программу. Хотя она будет простой, и в ней не будет содержаться ни строчки кода, я разберу её по косточкам. Мы познакомимся со всеми её внутренностями и узнаем, из чего состоит скелет любой программы на Delphi.

Глава 6. «Работа с компонентами». В этой главе я расскажу все основы работы с компонентами. Я опишу основные свойства, которые можно встретить у большинства объектов. А так же мы познакомимся с событийной моделью Windows, и основными событиями главной формы.

Глава 7. «Палитра компонентов Standard». В этой главе мы познакомимся с закладной *Standard* палитры компонентов. Я распишу все компоненты, для чего они предназначены, и как их использовать. Мы здесь напишем громадное количество примеров с использованием этих компонентов и закрепим всё описанное на практике.

Глава 8. «Учимся программировать». В этой главе я постарался подробно рассказать про циклы, логические операции, работу со строками и многое другое. Это последняя глава, в которой я рассказываю про самые основы программирования. Те, кто уже имеет опыт программирования в Delphi могут пропустить эту главу.

Глава 9. «Создание рабочих приложений». Здесь будет рассказана основа многооконных приложений. Сейчас уже трудно себе представить программу, состоящую только из одного главного окна. Большинство приложений состоит хотя бы из нескольких окон, а некоторые даже из сотен. Здесь же будет описано, как создавать главное меню программы.

Глава 10. «Основные приёмы кодинга». На первый взгляд тут находиться сборная солянка. Тут и работа с массивами, файлами, реестром, преобразование данных, структуры и указатели. Всё это собрано под одной крышей потому что я постарался построить книгу так, чтобы ты мог изучать всё последовательно, по мере надобности. Меня просто бесит литература, в которой сначала описывают разные функции и бесполезные примеры (которые в жизни не пригодятся), и только к концу книги находишь что-то действительно полезное. Я даю полезную информацию намного раньше, чтобы изучение программирования не показалось тебе рутинным и скучным.

Глава 11. «Обзор дополнительных компонентов Delphi». После того, как я рассказал об основных приёмах программирования, можно уже рассматривать остальные компоненты Delphi и сразу же писать к ним достаточно полезные в будущем примеры. Если бы я сделал это раньше, то ничего интересного в качестве примеров привести просто не смог бы.

Глава 12. «Графические возможности Delphi». Здесь будет рассказано всё, что касается графики. Я покажу, как можно рисовать встроенными средствами в Delphi различные фигуры и как работать с изображениями разного формата.

Глава 13. «Печать Delphi». Эта глава будет полностью посвящена печати и только печати. Я покажу как выводить на принтер текст и графику, как учитывать разрешение принтера и многое другое.

Глава 14. «Delphi и базы данных». Все знают, что на Delphi очень легко писать базы данных, потому что в него встроены сильнейшие для этого средства. В этой главе тебе предстоит в этом убедиться. Я покажу как работать с локальными базами MS Access и дам множество полезных примеров.

Глава 15. «Создание отчётности». Здесь я покажу, как можно экспортировать данные из твоих таблиц в Excel и подготавливать к печати документы любой сложности.

Глава 16. «Работа с DBF, Paradox и XML базами данных». В этой главе будет рассказано, как работать с другими таблицами, отличными от Access. Здесь будет описана технология доступа к данным через BDE и dbExpress.

Глава 17. «Потоки». Windows – многозадачная система и позволяет писать многопоточные приложения, в которых операции выполняются параллельно. В этой главе ты познакомишься с много поточностью и напишешь примеры.

Глава 18. «Динамические библиотеки». В этой главе будет рассказано всё необходимое, что касается динамических библиотек. Ты увидишь, как создавать библиотеки с математическими процедурами и функциями, как хранить окна в библиотеках и увидишь реальные примеры их использования.

Глава 19. «Разработка собственных компонентов». В этой главе пойдёт рассказ о том, как создавать свои VCL компоненты, как устанавливать чужие разработки в Delphi и как работать с пакетами компонентов.

Глава 20. «Мультимедиа». Эта глава полностью посвящена принципам программирования звука и видео. Я покажу в ней, как создавать приложения для работы со звуком с использованием встроенных в Delphi компонентов и без них.

Глава 21. «Графика OpenGL». Есть две достаточно перспективные разработки для профессиональной работы с компьютерной графикой – OpenGL и DirectX. Я решил достаточно подробно описать в этой книге только OpenGL, а по DirectX ты сможешь найти немного начальной информации на компакт диске в директории «*Документация*».

Глава 22. «OLE, COM, ActiveX». В этой главе будут описаны основные принципы технологий OLE, COM и ActiveX. Все эти термины взаимосвязаны и должны описываться вместе. Я не очень люблю эти технологии, но описать обязан, потому что иногда мне приходится работать с ними и возможно, что и ты когда-нибудь столкнёшься с этой технологией.

Глава 23. «Буфер обмена». Кнопки «*Копировать*» и «*Вставить*» есть практически в любом полноценном приложении. Я думаю, что ты тоже захочешь вставить такую возможность в свои программы. В этой главе я дам максимум полезной теоретической и практической информации, чтобы ты смог сделать свои программы более привлекательными, добавив возможность переноса данных между приложениями.

Глава 24. «Дополнительная информация». Эта глава единственная, которую ты можешь прочитать вне очереди. Как только ты почувствуешь, что твоих знаний достаточно для написания собственных небольших приложений, то ты можешь перескочить на эту главу. Здесь будут описаны некоторые приёмы работы с оболочкой Delphi, которые смогут тебе помочь при разработке собственных приложений, а так же принципы тестирования и отладки твоих программ.

Глава 25. «Сплошная практика». Напоследок я опишу несколько интересных программ, чтобы ты мог увидеть некоторые приёмы программирование, которые могут пригодиться тебе в будущем. Эту главу можно рассматривать как дополнительный материал ко всему сказанному выше.

Глава 1. Основные принципы работы компьютера

1.1 Основы работы персонального компьютера.

Прежде чем программировать компьютер, мы должны понять, как он работает. Как говорил какой-то полководец: «Нужно хорошо изучить своего врага!!!». Возможно, это говорил и не полководец, но это не важно :). Кодинг – это постоянная борьба с машиной. Нужно заставлять её делать то, что тебе нужно. Поэтому любой программист просто обязан знать его внутренности.

Компьютер состоит из следующих основных компонентов: процессор, память, видеокарта, винчестер (жёсткий диск) и различные разъёмы для подключения дополнительных устройств. Все эти компоненты связаны между собой с помощью шлейфов и шин.

Вся информация в компьютере хранится на винчестере. Когда ты запускаешь какую-нибудь программу, то она сначала загружается в память и только потом процессор начинает выполнять содержащиеся в ней инструкции. Чем больше программа, тем дольше она загружается.

Результат работы программы выводится на экран через видеокарту. На любой видеокарте есть чип памяти, в котором отображается всё содержимое экрана. Когда тебе нужно вывести что-то на экран, ты просто копируешь эти данные в видеопамять, и видеокарта автоматически выводит его содержимое на монитор.

Это всё, что необходимо знать о работе компьютера. Пока я описать только общие черты, а в следующих разделах я опишу необходимые вещи более подробно. В основном нас будет интересовать работа процессора, поэтому ему я уделю особое внимание. Остальное пока достаточно знать в общих чертах.

1.2 Двоичная система работы процессора.

Компьютеры изобрели достаточно давно. В те времена электроникой даже и не пахло. Первые компьютеры были ламповыми и занимали очень много места. Для того, чтобы управлять такой махиной нужно было очень много обслуживающего персонала.

Уже тогда был заложен принцип работы компьютера, который действует до сих пор. А именно, данные передаются с помощью какого-то сигнала (для нас не имеет значения какого, потому что мы не электронщики) методом «есть сигнал или нет» или по-другому «включён или выключен». Так появился «бит» bit. Бит это единица информации, которая может принимать значение или 0, или 1, т.е. «включён или выключен». Восемь бит объединяются в байт, т.е. один байт равен 8 битам. Почему именно 8? Да потому что первые компьютеры были восьми разрядными и могли работать одновременно только с 8-ю битами, например, 010000111.

Немного позже ты узнаешь, что в один байт можно записать любое число до 255. Но это очень мало, поэтому чаще используют более крупные градации:

1. Два байта = слово.
2. Два слова = двойное слово.

Итак, компьютер стал работать в двоичной системе исчисления. Но как же тогда записать число 135, если у нас единица информации может быть только или 0 или 1. Просто в двоичной системе. Давай разберёмся, как это работает.

Для начала вспомним, как работает наша десятичная система исчисления, к которой мы привыкли. Для этого рассмотрим число 519578246. Я специально выбрал такое число, чтобы оно состояло из восьми разрядом. Теперь запишем его, как на рисунке ниже:

Номер разряда	8	7	6	5	4	3	2	1	0
	5	1	9	5	7	8	2	4	6

Как видишь, я пронумеровал разряды, начиная с нуля до восьми, и справа налево. Теперь представь себе, что это не целое число, а просто набор разрядов. 5, 1, 9, 5, 7, 8, 2, 4 и 6. Как из этих разрядов получить целое число? Наверно некоторые скажут, что надо просто записать их подряд. А если я спрошу, почему? Вот тут появляется математика. Нужно каждый разряд умножить на 10 (степень исчисления) возведённую в степень номера разряда. Непонятно? Попробую оформить в виде формулы:

$$Разряд\_0 * (10^{Номеррязряда}) + Разряд\_1 * (10^{Номеррязряда}) + ...$$

Давай посчитаем по этой формуле, начиная с нулевого разряда. Получается, что 6 нужно умножить на 10 в нулевой степени $6*10^0=6$. Потом прибавить 4*10 в 1 степени $4*10^1= 40$ (итого уже 46). Потом 2*10 во второй степени $2*10^2=200$ (итого 246). Потом 8*10 в 3 степени $8*10^3= 8000$ (итого 8246) и так далее. В итоге получится число 519578246.

А теперь рассмотрим двоичную систему. Здесь каждый разряд может быть или 0 или 1 (2 состояния). Кстати, в десятичной системе у нас каждый разряд мог быть от 0 до 9, то есть десять состояний. Давай рассмотрим следующий байт - 010000111. Запиши его на листке бумаги так, как показано на рисунке ниже.

Номер разряда	8	7	6	5	4	3	2	1	0
Биты	0	1	0	0	0	0	1	1	1

Здесь действует та же самая формула, только нужно возводить в степень не 10, а двойку. Опять же произведём расчёт, начиная с нулевого разряда, т.е. справа налево. Получается, что первую 1 мы должны умножить на 2 в нулевой степени ($1*2^0=1$). Следующую единицу нужно умножить на 2^1 получается 2 (итого 2+1=3) и т.д. Вот как это будет выглядеть полностью:

$$(1*2^0)+(1*2^1)+(1*2^2)+(0*2^3)+(0*2^4)+(0*2^5)+(0*2^6)+(1*2^7)+(0*2^8)=135.$$

Вот так, оказывается, выглядит в двоичной системе число 135. Давай теперь научимся пересчитывать числа из десятичной системы в двоичную систему. Для этого нужно число 135 разделить на 2. Получается 67 и остаток 1 (запомним 1). Теперь 67 делим на 2, получается 33 и остаток 1 (теперь две единицы, т.е. 11). Теперь 33 делим на 2, получаем 16 и остаток 1 (теперь три единицы, 111). Теперь 16 делим на 2, получаем 8 и остаток 0 (всего 0111). Теперь 8/2=4 и остаток 0 (00111). 4/2=2 и остаток 0 (000111). Теперь 2/2=1 и остаток 0 (итого 0000111). 1 на два не делится, значит, просто дописываем её 10000111. Получилось первоначальное число.

Вот так происходит преобразование чисел. В двоичную систему исчисления. Таким же образом можно перевести число в любую систему (двоичная, восьмеричная,

шестнадцатеричная и т.д). Для более полного закрепления материала я решил привести таблицу, в которой показаны соответствия десятичных чисел двоичным. Попробуй сам перевести пару чисел туда и обратно.

Десятичное	Двоичное
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Таблица 1. Таблица соответствия десятичных и двоичных чисел

В компьютере принято вести расчёт в двоичной или шестнадцатеричной системе. Вторая вошла в обиход, когда компьютеры стали 16-и разрядными.

Шестнадцатеричная система выглядит немного по-другому. Каждый разряд уже содержит не 2 состояния (как в двоичной) или десять (как в десятичной), а шестнадцать. Поэтому один разряд может принимать значения от 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Буква «A» соответствует 10, «B» соответствует 11 и т. д. Например, число 1A в шестнадцатеричной, равно 26 в десятичной. Почему? Да по всё то же формуле. Только здесь нужно возводить 16 в степень номера разряда. «A» - это десять, нужно умножить на $16^0 = 10$. 1 – первый разряд нужно умножить на $16^1 = 16$. 10+16=26.

Десятичное	Двоичное	Шестнадцатеричное
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Таблица 2. Таблица соответствия десятичных, двоичных и шестнадцатеричных чисел

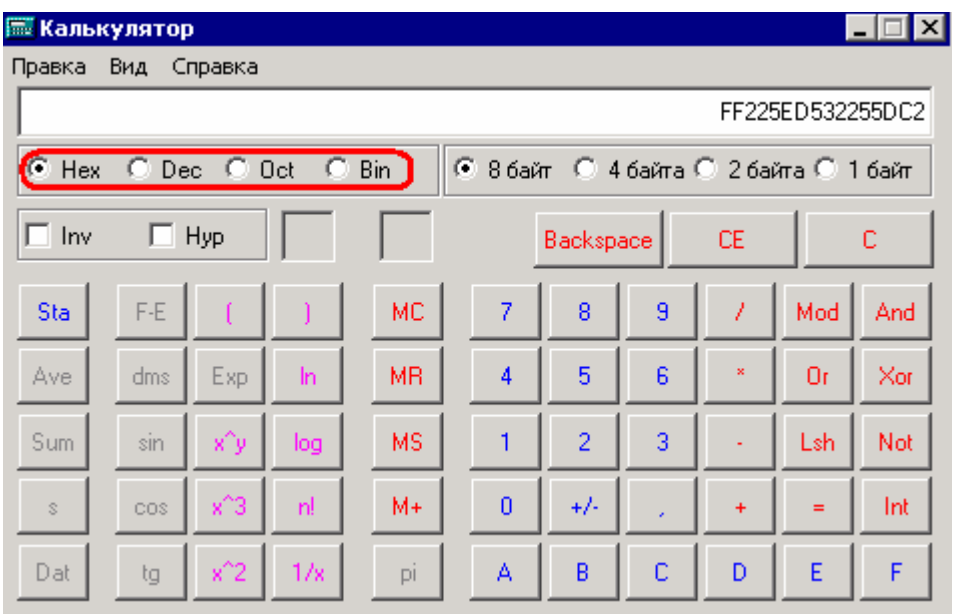
На протяжении всей книги мы будем иногда встречаться с шестнадцатеричной системой исчисления (без этого никуда не денешься), поэтому, когда нужно будет показать, что число шестнадцатеричное, я буду ставить перед ним знак решётки #, например, #13. В других языках, например Assembler или C++ принято ставить в конце числа букву h, например, 13h. Но эта книга о Delphi, поэтому я буду писать так, как принято в этой среде разработки, чтобы потом не возникало никаких проблем.

Но это всё целые числа. С числами с плавающей точкой совершенно другая история. Если заранее предусмотрено, что число может быть отрицательным, то его длинна сокращается ровно на один бит. Если неотрицательное целое число может быть 8-ми битным, то число со знаком будет 7-и битным. Первый бит будет означать знак. Если первый бит равен 1, то число отрицательное, иначе положительное.

В дробных числах один байт может быть отведён для целой части и один для дробной. Никогда не смешивают целую и дробную часть в одно целое. За счёт этого, дробные числа всегда будут занимать больше памяти, и операции с ними проходят намного дольше.

На первый взгляд перевод чисел очень сложный, но вручную им пользоваться не обязательно. Человек уже давно придумал для себя хорошего помощника - калькулятор. С его помощью без проблем можно перевести число в любую систему.

Запусти встроенный в Windows калькулятор (Пуск -> Программы -> Стандартные -> Калькулятор).Теперь выбери из меню «Вид» пункт «Инженерный». На рисунке ниже показано окно, которое ты должен увидеть:



Внешний вид калькулятора

Для перевода числа в другую систему, просто набери его и потом выбери нужную систему исчисления. На рисунке я обвёл красным цветом кнопки переключения системы исчисления:

- Hex – шестнадцатеричная.
- Dec – десятичная.
- Oct – восьмеричная.
- Bin – двоичная.

Возникает вопрос – зачем я тогда так долго рассказывал о преобразованиях, когда так легко воспользоваться калькулятором? Ответ прост – НАДО. Поверь мне. Если ты будешь понимать, как происходит преобразование, то тебе потом легче будет работать с этими числами.

Никогда не полагайся только на технику. Всегда полезно знать, как и зачем она что-то делает. Если ты разберёшься с шестнадцатеричным представлением данных, то сможешь простые преобразования делать в уме. Ну а если ты ещё и собираешься стать хакером, то тебе просто необходимо научится хорошо оперировать разными системами исчисления.

1.3 Машинный язык.

Данные на диске также хранятся в двоичном виде. Даже текстовые файлы на диске выглядят в виде нулей и единиц. Точно так же выглядит и любая программа, только её называют машинным кодом. Давай с ним познакомимся немного поближе.

Любая программа представляет собой последовательность команд. Эти команды называются *процессорными инструкциями*. По этим инструкциям процессор определяет, что и как ему нужно делать. Когда ты запускаешь программу, компьютер загружает её машинный код в память и начинает выполнять. Наша задача, как программистов написать эти инструкции, чтобы компьютер понял, что мы от него хотим.

Реальная программа, которую выполняет компьютер, представляет собой последовательность единиц и нулей. Такую последовательность называют машинным языком. Но человек не способен эффективно думать единицами и нулями. Для нас легче воспринимается осмысленный текст, а не сумасшедшие числа в двоичной системе измерения, с которой мы не привыкли работать. Например, команда складывания двух регистров выглядит так: #03C3. Нам это мало о чём говорит, и запомнить такую команду очень тяжело. На много проще написать «сложить число1+ число2».

Первое время программисты писали в машинных кодах, пока кому-то не пришла в голову идея: «Почему бы не писать текст программы на понятном языке, а потом заставлять компьютер переводить этот текст в машинный код?». Идея действительно заслуживала внимания. Так появился первый компилятор – программа, которая переводила текст программ в машинный код.

Вот тут, я думаю надо сделать паузу и рассказать тебе небольшую историю языков программирования. Она достаточно интересна и поучительна. Ну а потом мы продолжим изучения принципов работы компьютера и познакомимся с содержимым процессора и его работой.

1.4 История языков программирования.

Как мы уже выяснили, компьютер - примитивное существо, которое мыслит нулями и единицами, из которых складываются числа. Так что все, что может делать процессор, так это оперировать этими числами. Так и программы - это тоже числа, которые воспринимаются процессором как команды к выполнению каких-то действий.

Мы также выяснили, что первые программисты писали программы в машинных кодах. Тогда еще не было компиляторов и приходилось все писать числами. Ты даже представить себе не можешь, какой это адский труд. Постоянно держать в памяти таблицу машинных кодов - это тебе не таблица умножения. Например, тебе понятно число 8BC3. Нет? А это простая команда копирования между двумя ячейками регистров. Это просто пример, потому что тогда регистры были другие и процессоры были на много проще.

Со временем компьютер стал умнеть. Он все так же оперировал числами, но делал это намного быстрее. Но программист - это человек, а не железка и ему очень тяжело создавать логику в числах. Намного легче работать с привычными словами. Например, все ту же команду удобней записать словами типа "скопировать ebx в eax". Но что делать, если компьютер не понимает слов, а только числа? Выход есть - написать такую программу, которая будет превращать текст в машинные коды. Пусть компьютер сам создает байт-код. Такую программу назвали компилятором. А язык, на котором писался текст программы, назвали языком программирования.

```
MainUnit.pas.131: InitViewGrid;
00527F7C 8BC3            mov eax,ebx
00527F7E E84D010000      call TSborDataForm.InitViewGrid
MainUnit.pas.132: InitProgramm;
00527F83 8BC3            mov eax,ebx
00527F85 E8B6030000      call TSborDataForm.InitProgramm
MainUnit.pas.133: if lDeviceNum>=0 then
00527F8A 83BB1004000000  cmp dword ptr [ebx+$00000410],$00
00527F91 7C26            jl +$26
MainUnit.pas.135: ScanPortThr := TScanPortThread.Create(true);
00527F93 B101            mov cl,$01
00527F95 B201            mov dl,$01
00527F97 A1D4A45200      mov eax,[$0052a4d4]
00527F9C E8EB94EFFF      call TThread.Create
00527FA1 8BF0            mov esi,eax
00527FA3 89B314040000    mov [ebx+$00000414],esi
MainUnit.pas.136: ScanPortThr.lDeviceNum:=lDeviceNum;
00527FA9 8B8310040000    mov eax,[ebx+$00000410]
00527FAF 89464C          mov [esi+$4c],eax
MainUnit.pas.137: ScanPortThr.Resume;
00527FB2 8BC6            mov eax,esi
00527FB4 E81F98EFFF      call TThread.Resume
MainUnit.pas.141: try
00527FB9 33C0            xor eax,eax
00527FBB 55              push ebp
00527FBC 680B805200      push $0052800b
00527FC1 64FF30          push dword ptr fs:[eax]
```

Программа в машинных кодах и Assembler

И вот был написан первый компилятор. Эту программу назвали Assembler, что переводится, как "сборщик". Писать на нем практически так же, как и в машинных кодах, только теперь уже использовались не числа, а понятные человеку слова. Например, все та же команда копирования регистров теперь выглядела так: "mov eax, ebx". То есть цифры заменились на понятные слова.

Вроде все прекрасно и удобно, но почему-то среди программистов возникли споры и разногласия. Кто-то воспринял новый метод с удовольствием. А кто-то говорил, что машинные коды лучше. Любители языка Assembler хвалили компилятор за то, что

программировать стало проще и быстрее, а противники утверждали, что программа, написанная в кодах, работает быстрей. Говорят, что эти споры доходили до драк и иногда лучшие друзья становились врагами. А в принципе, и те и другие были правы. На языке Assembler действительно программу писать легче и быстрей, а в машинных кодах программа работала быстрее.

Тогда никто не мог себе представить, чем же все может закончиться. Но время показало свое. С помощью Assembler программы писались быстрее, а это один из основных факторов успеха любой программы на рынке. Люди начинают пользоваться тем продуктом, который выходит на рынок первым. Даже если более поздний вариант лучше, человека трудно переубедить перейти на другую версию. Здесь играет большую роль фактор привычки. К тому же, к тому моменту, когда программист напишет свою первую версию в машинных кодах, программист на языке Assembler выпустит уже пару новых версий своего шедевра.

Вот так и получилось, что те, кто программировал на языке Assembler превратились в убегающих вперед, а те, кто программировал в машинных кодах превратился в вечно догоняющих. В конце концов, первые убежали на столько, что вторые не смогли догнать, и вынуждены были или перейти на Assembler или отойти от программирования на совсем.

Вот тут начался бум. Языки программирования стали появляться один за другим. Так появились C, ADA, FoxPro, Fortran, Basic, Pascal и другие. Некоторые из них были предназначены только для детей, а некоторые и для профессиональных программистов. И тут споры перенеслись в другую плоскость - какой язык лучше. И этот спор длится уже около 30 лет и конца ему не видно. Некоторые говорили, что это Pascal, другие утверждали что C, ну а кое-кто утверждал что это Visual Basic. Этот спор разделился на две части:

1. Какой язык самый лучший?
2. Что лучше - язык высокого уровня или низкого?

Первый спор не может закончиться до сих пор. Каждый пытается доказать, что его язык программирования самый могучий, удобный и создаёт самый быстрый код. Мне кажется, что этот спор не закончится никогда. В принципе, меня это устраивает, потому что это своеобразная конкуренция. Благодаря ей происходит развитие и мы летим вперед.

Так все же, какой язык лучше? На этот вопрос я дам ответ, но только немного позже.

Наиболее интересным был спор: "Что лучше - язык высокого уровня или низкого?". Язык низкого уровня это тот, который наиболее приближен к командам процессора, то есть Assembler. К языкам высокого уровня относят C, Pascal, Basic и др. Этот спор проходил в той же манере, как и спор между любителями Assembler и любителями программирования в машинных кодах. Только теперь приверженцы Assembler утверждали, что их код самый быстрый, а любители языков высокого уровня утверждали, что они напишут программу быстрей, чем самый лучший программист на языке Assembler.

Спор продолжался достаточно долгое время. И опять победила скорость разработки и удобство языка программирования. Любителям Assembler пришлось отступить, потому что теперь они превратились в «догоняющих», и не смогли угнаться за языками высокого уровня.

Конечно же, нельзя сказать, что машинные коды и Assembler на совсем ушли из нашей жизни. Они используются до сих пор, но в очень ограниченном количестве. Язык Assembler используется только в качестве вставок для языков высокого уровня, а машинные коды используются для написания того, чего нельзя сделать компилятором (да и для написания самого компилятора они нужны). Ушедшие технологии живут, и будут жить, но рядовой программист очень редко встречается с ними.

Следующей ступенью стало объектно-ориентированное программирование. Язык C превратился в C++, Pascal превратился в Object Pascal и так держать. И снова борьба. И снова скорость разработки против быстроты кода. Опять споры, драки и оскорбления.

Война длилась несколько лет. Сколько времени было потрачено в спорах, сколько волос было вырвано на голове в процессе доказательств крутизны именно его кода. А результат - победила скорость и удобство разработки, т.е. объектно-ориентированное программирование (ООП).

Последней крупной революцией происходящей в программировании я считаю переход на визуальное программирование. Этот переход происходит прямо на наших глазах. Визуальность дает нам еще более удобные средства разработки для более быстрого написания кода, но проигрывает ООП по быстроте работы. Вот многие начинающие и стоят на перекрестке, какой язык выбрать.

Лидеров в визуальных языках является Borland, а приверженцем ООП остается Microsoft. Конечно же, Билл Гейтс пытается встроить в свои языки визуальность, но она примитивна по сравнению с такими гигантами, как Delphi, Kylix или C++ Builder. Это связано с изначальной дырой MFC, которая не может работать визуально. Нужна глобальная переработка кода, которую почему-то не хотят делать. Вот народ и стоит на двух атомных бомбах и ожидает взрыва одной из них. Как ты думаешь, какая бомба рванет? Что победит - скорость разработки или скорость кода? Я не буду отвечать на этот вопрос. История говорит сама за себя, а мы подождем подтверждение этому.

Я считаю, что прогресс не будет стоять на месте и переход на новые технологии программирования рано или поздно состоится. Поэтому я уже перешел на Delphi. Если ты хочешь успеть за прогрессом, то ты тоже обязан вступить в партию любителей Borland. Выбирай любой из его компиляторов, и ты не ошибешься. Для тебя есть все, что угодно Delphi, JBuilder, Kylix или C++ Builder. Как видишь у Бормана есть визуальные варианты всех языков, и они действительно лучшие.

Я уже сказал, что самая лучшая технология - визуальность. Твоя среда разработки просто обязана быть визуальной, потому что за этим наше будущее. Если ты хочешь получить визуальность + мощь разработки, то твоя среда от Borland. С помощью языков этой фирмы можно сделать абсолютно все. Так что с этим мы покончили. Вердикт окончательный и обжалованию не подлежит.

Мне осталось только ответить на вопрос: "Какой язык программирования лучше?". Я уже несколько лет пытаюсь ответить на этот вопрос, но окончательного решения вынести не могу. Даже у того же Visual C++ от Microsoft есть свои плюсы. Как это не странно, но положительные стороны есть у всех. Вопрос остается только за тем, что ты будешь писать? Я могу дать примерно такую градацию:

1. Если ты будешь писать базы данных, программы общего значения или утилиты, то твой язык Delphi или C++ Builder.

2. Если это игры, то желательно Visual C++ или Watcome C плюс знание Assembler. Но это не значит, что нельзя использовать Delphi или C++ Builder. В этих средах ты потеряешь не намного больше в скорости работы, поэтому на большинстве игр можно не обращать внимания на эту потерю.

3. Если это будут драйверы и работа с железом, то тут критичен размер файла, а значит твой язык чистый C или Assembler.

И все же большую массу программ занимают утилиты и базы данных. А тут визуальность необходима, если ты хочешь оказаться впереди. Визуальные языки будут жить и за ними будущее. И на протяжении всей этой книги я буду тебе рассказывать про самый лучший (это на мой взгляд, и он может отличаться от других) - Delphi.

1.5 Исполнение машинных инструкций.

Прежде чем переходить дальше, я должен познакомить тебя с несколькими понятиями:

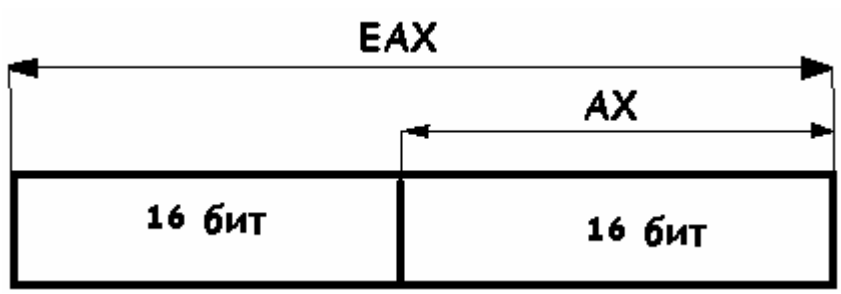
Сегмент – это просто область памяти. Раньше, когда операционные системы (ОС) были 16 битными, процессор не мог работать с памятью размером более 64 килобайт (это максимум, что можно записать в два байта). Поэтому память делилась на сегменты по размеру и по назначению. На данный момент мы используем 32-ю ОС, которая может адресовать до 4 Гбайт оперативной памяти. Поэтому можно сказать, что память стала сплошной. Но деление по назначению всё-таки осталось. Существуют следующие сегменты памяти:

- ***Сегмент кода*** – в эту область памяти загружается машинный код, который будет потом выполнятся процессором.
- ***Сегмент данных*** – это область памяти для хранения данных.
- ***Сегмент стека*** – область памяти для хранения временных (локальных) данных и адресов возврата из процедур.

Каждой запущенной программе отводится свой сегмент кода, данных и стека. Поэтому данные одной программы не могут пересекаться с данными или кодом другой программы, если конечно же не произошёл сбой.

Регистр – ячейка памяти в процессоре. Размер ячеек зависит от его разрядности. В 32-х разрядных процессорах ячейки 32-битные. Мы будем говорить о 32-х разрядных процессорах, а значит и о 32-х битных регистрах. Таких ячеек там несколько и каждая из них предназначена для определённых целей. Один регистр состоит из двух ячеек, значит в 32-битном процессоре регистр равен 2*32=64 бит.

Когда компьютер был ещё 16 битным, все регистры были тоже 16-и битными. С появлением 32-й платформы размер ячейки регистра тоже увеличился, до 32 бит. Но для совместимости со старыми программами они как бы делятся на две части (те же две ячейки). Первая – это тот старый регистр, а вторая – дополнительные 32 бит. На словах не совсем понятно, поэтому давай посмотрим на рисунок.



На этом рисунке показан регистр EAX. Полная его длинна – 32 бита, но младшая половина – это регистр AX (16 битный вариант регистра). То есть, если мы попросим процессор показать нам содержимое регистра AX, то мы увидим половину регистра EAX. Иногда это очень даже удобно, особенно когда тебе надо прочитать только половину числа из регистра.

Теперь реальный пример. Допустим, в регистре EAX находится шестнадцатеричное число #21CD52B, тогда в регистре AX будет находиться последние 16 бит, а именно D52B.

Сейчас я начну описание регистров, и если его имя начинается с буквы «E», то это значит, что он 32-битный и для него существует и 16-и битный вариант без буквы «E».

Сегментные регистры - CS, DS, SS и ES. (есть ещё, но нас пока интересуют только эти):

- ***Регистр CS*** - регистр сегмента кода, в нём хранится начальный адрес сегмента кода.
- ***Регистр DS*** - регистр сегмента данных, в нём хранится начальный адрес сегмента данных. В этом сегменте располагаются глобальные переменные.
- ***Регистр SS*** - регистр сегмента стека, в нём хранится начальный адрес сегмента стека. Здесь располагаются локальные переменные.
- ***Регистр ES***. Для использования дополнительного сегментного регистра.

Разницу между глобальными и локальными переменными мы рассмотрим чуть позже, когда будем изучать сам язык программирование. Сейчас я ещё скажу только то, что в сегменте сохраняются и переменные, которые передаются в процедуры и адрес возврата из процедуры (куда нужно вернуться по окончанию выполнения кода процедуры).

Регистры общего назначения EAX, EBX, ECX и EDX, все эти регистры 32-х битные. В 16-разрядных процессорах, они были 16-разрядными и назывались AX, BX, CX и DX. Они могут использоваться в программе по собственному усмотрению, но в некоторых случаях им отведена определённая роль. В регистр EAX в основном записываются результаты арифметических вычислений, а регистр ECX используется в качестве счётчика. Регистр EAX используется для хранения результатов вычисления.

Очень часто, прежде чем выполнить какую-то команду, процессор загружает необходимые данные в регистры и только после этого выполняет необходимую инструкцию. Но возможны варианты, когда вычисления идут напрямую с памятью.

Регистровые указатели ESP и EBP. ESP - это указатель стека, который обеспечивает его использование в памяти. EBP – обеспечивает доступ к данным и указателям на данные переданные через стек.

Индексные регистры ESI и EDI. Эти регистры используются при сложении и вычитании, а так же для расширенной адресации.

Если ты собираешься писать простенькие утилиты или базы данных, то эти знания ты не будешь использовать. Но если ты хочешь пойти дальше, то желательно не просто знание, но и понимание процесса работы процессора. Поэтому я сейчас попробую на пальцах (а точнее сказать на примере) объяснить процесс его работы.

Итак, сейчас я опишу процесс выполнения программы более подробно. В любом случае, это тебе пригодится.

При старте программы, исполняемый код загружается в сегмент кода. Регистр CS сразу устанавливается в значение, указывающее на начало этого сегмента. Данные программы загружаются в сегмент данных (это константы и любые другие дополнительные данные). На этом же этапе происходит подготовка (инициализация) к работе сегмента стека. Если программе переданы какие-нибудь значения, то они автоматически заносятся в стек.

Сегменты кода содержит код только одной программы. В один сегмент не может быть загружен код двух абсолютно разных программ. Точно так же и сегмент данных, и сегмент стека. При каждом старте новой программы,

операционная система отводит её свои собственные сегменты кода, данных и стека.

После этих подготовительных действий, ОС готова к выполнению кода. Напомню, что регистр CS указывает на начало сегмента кода. Есть ещё один регистр, о котором я ещё не сказал – EIP. Этот регистр указывает на текущую выполняемую команду в сегменте кода.

Процессор последовательно выполняет все команды, находящиеся в сегменте кода. Иногда необходимо перескочить не на следующую точку программы, а совершенно в другое место. В этом случае происходит такой переход, и команды начинают последовательно выполнятся, уже начиная с новой точки.

Я уже сказал, что любые операции, вычисления могут производиться с регистрами и с памятью. Например, к значению регистра EAX прибавить значение EBX – это сложение регистров. Можно складывать и значения находящиеся в оперативной памяти. Но надо помнить, что вычисления с регистрами происходит намного быстрее, потому что регистр – это та же оперативная память, только находящаяся в процессоре. Если ты собираешься произвести с одним и тем же числом две операции, то намного эффективнее будет располагать его в регистре.

Глава 2. Машинная математика.

До перестройки, в нашей стране практически не обучали программистов. Большинство программистов были выходцами с кафедр математики, на которых очень часто были какие-то предметы с уклоном в сторону информатики. На этих курсах учили писать блок-схемы – это схема, описывающая логику программы.

Я с блок-схемами познакомился на первом курсе института. Первое впечатление – полное фуфло. Но со временем я понял их достоинства. Возможно, что тебе покажется это слишком просто, но всё же желательно прочитать эту главу полностью. Здесь я расскажу теорию всего процесса программирования. В дальнейшем нам останется только познакомится с практикой, и мы на коне ☺.

2.1 Основы машинной математики.

На любом языке программирования можно выполнять математические операции любой сложности. Delphi не исключение. Но пока что мы не будем рассматривать все возможности, а остановимся только на основных. Вот основные математические операции языка Delphi:

Математическая операция	Описание
*	Умножить
/	Разделить
Sqr	Квадрат
Sqrt	Квадратный корень
+	Сложение
-	Вычитание
:=	Присвоить значение.

Математические операции

В таблице перечислены основные математические операции языка программирования Delphi. Они выполняются в том же порядке, в котором перечислены. Например, в формуле 2+2*2 результатом будет 6, потому что сначала выполняется операция умножения, а потом сложения. Если ты хочешь сначала выполнить сложение, а потом вычитание, то как и в простой математике должен использовать скобки: (2+2)*2=8. В этом случае результат уже будет совершенно другим.

Для изучения компьютерной математики ты должен знать следующие понятия:

Переменная – это память, в которую можно записывать различные значения. Чаще всего этой памяти присваивается в соответствие имя. Например, я завожу переменную с именем F. Ей я могу присваивать значения, например 5. Для этого мне нужно записать F:=5. Знак двоеточие + равно означает операцию «присвоить».

Значения переменных можно копировать из одной в другую. Допустим, что у меня есть ещё одна переменная G. Я могу присвоить ей значение переменной F с помощью простого присваивания G:=F. После этого в переменной G у меня тоже будет значение 5.

Переменной можно присваивать результаты каких-то вычислений, например: F:=10/2. Это достаточно простой пример. А вот уже целое выражение с использованием переменных:

```
F:=5;
G:=10;
F:=G/2;
```

Имя переменной может состоять как из одной буквы, так и из нескольких букв. Например, имя переменной может быть: Str, или MyPeremen. Единственное ограничение – имя должно состоять из английских букв и не должно использовать зарезервированные слова (о зарезервированных словах немного позже). Ты так же можешь в имени использовать числа (желательно в конце), например Str1, Str2, Str3 и так далее.

Мой тебе совет, назначай переменным осмысленные имена. Когда ты начнёшь писать большие программы, тяжело будет разобраться, что означает переменная i или b. Желательно давать более осмысленные имена. В течении всей книги я постараюсь тебя к этому приучить.

Тип переменной – тип значения, которое можно записать в переменную (в память). Очень часто используют термин *«Тип данных»*, потому что это действительно тип данных хранящихся в переменной. Он показывает, какого типа информация находится в конкретной переменной. В Delphi принято обязательно указывать типы переменных, чтобы сразу можно было увидеть какую информацию можно туда записать.

Существует несколько основных типов переменных:

Название типа	Описание	Дополнительная информация
Integer	Целое число	Переменная этого типа может принимать в качестве значения любые целые числа, как положительные, так и отрицательные.
Real	Вещественное число	Переменная этого типа может принимать в качестве значения целые и дробные числа со знаком и без.
String	Строка	Переменная этого типа может принимать в качестве значения любые символы и наборы символов.
Boolean	Булево значение	Может принимать значение true или false (истина или ложь). Этот тип очень часто используется для организации логики.

Основные типы данных в Delphi

Это только основные типы. Реально их намного больше. Когда мы перейдём к программированию, я познакомлю тебя с большим количеством типов данных.

Строки – любые символы и наборы символов. В языке Delphi они выделяются одинарными кавычками, например, ‘Привет’. Строки так же можно присваивать переменным, как и любое другое значение.

Str – строковая переменная.
Str:=’Привет!!!’

На этом основные сведения о машинной математике подошли к концу. Пора применить наши знания на практике.

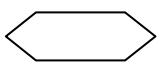
2.2 Блок-схемы.

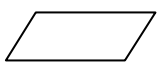
Давай сразу зададим какой-нибудь простой пример, на котором попробуем расписать логику его решения. Допустим, нам надо получить произведение двух чисел. В человеческой логике мы должны выполнить следующие операции:

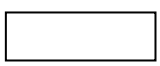
1. *Старт.*
2. *Ввести число.*
3. *Ввести число 2.*
4. *Умножить число 1 на число 2.*
5. *Вывести результат.*

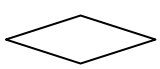
Простейшая и подробная логика, которой оперирует человек. Но машина немного сложнее и в её логике нужно рассуждать немного по-другому. Для отображения машинной логики удобней перечисления шагов не удобно, поэтому давай знакомится с блок схемами на этом примере.

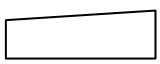
Блок схемы принято чертить различными квадратами, овалами и прямоугольниками. Я особо не буду придерживаться стандартов, потому что это не особо имеет значения, но некоторых особенностей буду придерживаться. Вот основные типы блоков используемых мной:

Начало работы – 

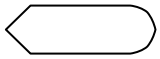
Данные - 

Процесс - 

Логика - 

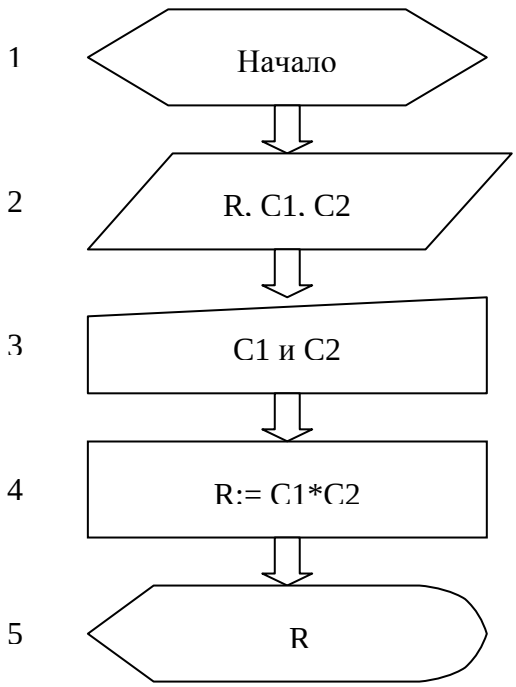
Ввод данных - 

Запись/чтение с диска - 

Вывод на экран - 

Есть и другие, более извращённые блоки, но я ими не буду пользоваться. Большинство блоков я буду оформлять, как просто прямоугольник, потому что от формы блока суть особо не изменится. Самое главное (на мой взгляд) выделить отдельным видом блока начало блок-схемы и логику. Всё остальное можно оформить однообразно, наглядность от этого пострадает, но не сильно.

Итак, наша первая блок-схема, умножения 2-х чисел будет выглядеть так:



На первый взгляд всё слишком сложно. Но это только первый взгляд. Реально здесь ничего сложного нет, просто очень громоздко и простейшая операция перемножения превращается в несколько операций. Но всё же надо объяснить происходящее поподробнее, чтобы мы могли продвинуться дальше и разобраться с более сложными примерами.

Первый блок – это начало. Если ты соберёшься строить блок-схемы, то обязательно указывай его, чтобы сразу можно было увидеть, начало логики.

Второй блок – перечисляет переменные, которые нам нужны для вычислений. Я использую три переменные R, C1 и C2. В переменную R будет помещён результат вычисления. Переменные C1 и C2 используются для хранения введённых данных. Пока я не указываю тип данных хранящихся в переменных, но подразумеваю, что это будут или целые числа, или вещественные.

Третий блок – здесь показывается, что надо ввести значения переменных C1 и C2.

Четвёртый блок – здесь показывается, на необходимость произвести умножения C1 на C2 и результат записать в переменную R.

Пятый блок – вывод результата на экран.

Это простая блок-схема, в которой нет ничего особенного, и ты даже не можешь увидеть всех её преимуществ. Следующие примеры будут уже использовать логику, поэтому блок-схемы будут более сложными, и ты сможешь ощутить всю прелесть машинной математики.

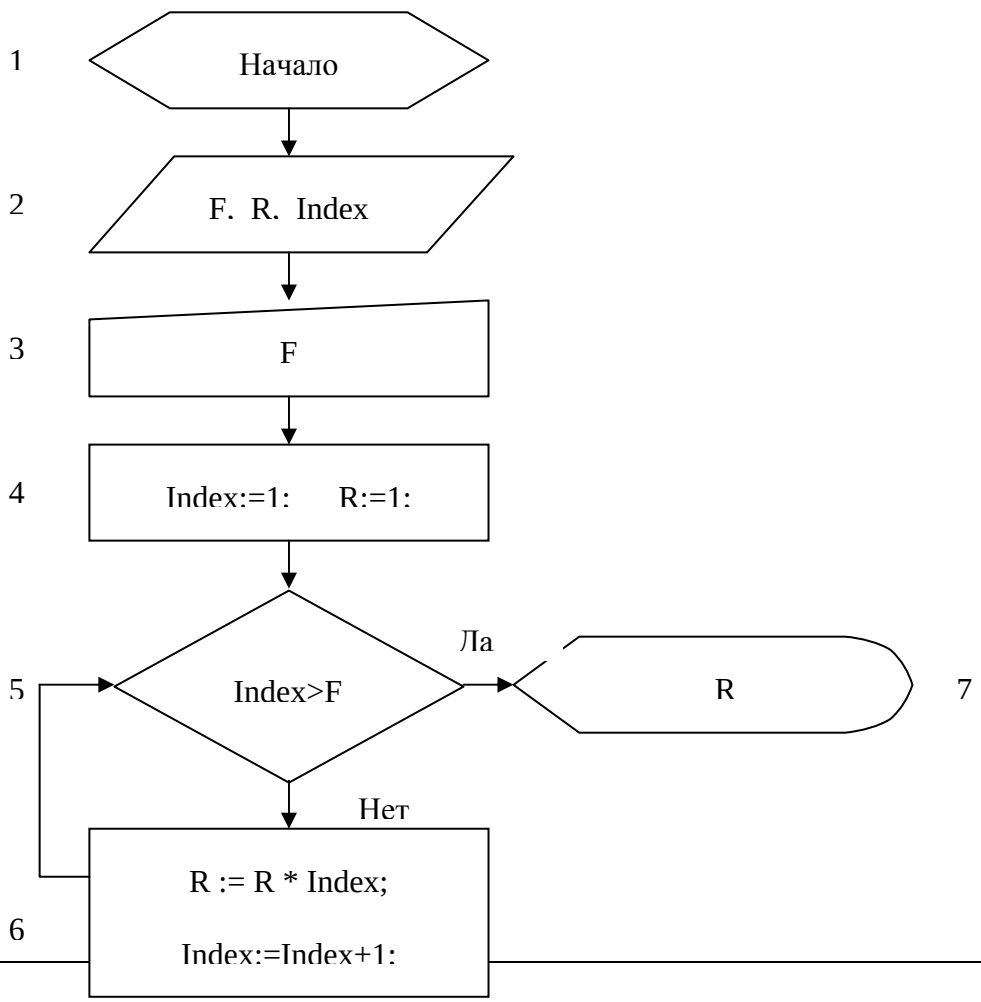
2.3 Машинная логика и циклы.

В любом языке программирования есть куча операций сравнения, которые позволяют реализовать машинную логику. Что понимается под логическими операциями? Это операции сравнения на «равенство», «больше» или «меньше».

Математическая операция	Описание
=	Равно
>	Больше
<	Меньше

Давай рассмотрим простейшую логику компьютера на примере факториала. Факториал – это произведение от 1 до какого-то числа. Например, 5 факториал равен 1*2*3*4*5=120. Факториал обозначается как знак восклицания «!».

Давай напишем алгоритм в виде блок-схемы:



Представим, что мы не знаем число, факториал которого мы должны вычислить. Это число будет вводить пользователь. Поэтому в общем случае формула будет выгладить, как число F факториал (F!). Так что нам надо перемножить все числа от 1*2*3*4*…*F. Всё это можно сделать последовательно. Умножить 1*1. Затем проверить, не превысили ли мы число F, если нет, то результат прошлого вычисления * на 2. Снова проверить. Если не превысили F, то снова умножить результат прошлого вычисления на 3. И так далее.

Теперь я постараюсь объяснить каждый блок, чтобы ты понял, как эта блок-схема работает. Это очень важно. Работа блок-схемы очень похожа на то, как мыслит компьютер. Именно так ты должен будешь размышлять, когда начнёшь писать первые программы.

Итак, давай рассмотрим каждый блок в отдельности:

1. Это опять начало.
2. Я объявляю три переменных: F, R и Index. F – здесь я буду хранить число, факториал которого надо вычислить. R – это для результата. Index - здесь я буду держать счётчик вычислений.
3. В этом блоке происходит ввод числа, факториал которого надо вычислить. Если пользователь ввёл 5, то мне нужно вычислить 5!.
4. Задаю начальные значения переменным. На этом этапе я устанавливаю счётчик и результат равным 1. Почему именно 1? Да потому что мне нужно перемножить все числа начиная от 1 до числа факториала.
5. В этом блоке я проверяю – счётчик (index) больше числа, которое ввёл пользователь? Если «нет», то надо перейти на блок 6. Допустим, пользователь ввёл число 5. Наш счётчик пока равен 1. 1 меньше 5, значит, мы должны перейти на блок 6.
6. Здесь я вычисляю результат R:=R*Index. На этом этапе у меня Index=1 и R=1. Значит, R:=1*1. Результат будет 1, значит, в R опять будет единица. Далее, я увеличиваю index на 1 (Index:=Index+1), после чего Index становится равным 2 (Index:=1+1). И снова возвращаюсь на блок 5. В нём опять происходит проверка Index>F. Index сейчас равен 2, а это меньше пяти. Значит, снова идём на блок 6. Здесь опять расчёт R:=R*Index (R:=1*2), после чего R становится равным 2. Опять увеличиваем счётчик (Index:=Index+1), и Index становится равным 3. Снова на блок 5.

… И так далее.

Я не стал дальше расписывать. Попробуй сам пройтись по этой блок-схеме, рассуждая так же, как и я. Очень важно понять, как это работает.

В этой блок схеме мы задействовали очень интересную и очень удобную вещь, как циклы. Цикл – повторяющееся выполнение какого-то блока. В данном случае мы несколько раз выполняем шестой блок. Если бы мы знали заранее число факториала, то мы могли бы упростить нашу блок-схему, написав формулу типа R:=1*2*3*4*5. Но мы не знаем числа, которое введёт пользователь. Поэтому нам приходится делать цикл на каждом этапе которого, вычисляем промежуточный результат.

Попробуй сам написать блок-схему арифметической прогрессии. Её формула достаточно проста и ты без проблем сможешь модифицировать блок-схему. Разница только в том, что арифметической прогрессии рассчитывается сумма чисел от 1 до определённого числа F.

Нарисуй эту блок-схему на бумаге и попробуй мысленно пройтись по ней, выполняя каждый блок, как бы это делала машина. Даже если ты думаешь, что блок схемы слишком

просты, то попробуй написать что-нибудь более сложное. Ты просто обязан научиться мыслить как компьютер. Только так ты сможешь объяснить ему то, что тебе нужно, и он тебя сможет понять.

2.4 Программирование машинной логики.

Подошло время превратить нашу логику, описанную в блок-схеме в настоящую программу. Пока эта программа будет существовать только на бумаге, но со временем мы её сможем превратить в настоящий исполняемый модуль.

Сначала напишем нашу программу на русском языке:

Начало программы.

Переменные:
F, R, Index – это целые числа;

Начало кода
F:=5;
R:=1;
Index:=1;

От 1 до 5 выполнять
Начало цикла
*R:=R*INDEX;*
INDEX:=INDEX+1;
Конец цикла

Вывести на экран переменную R.
Конец кода

Если не обращать внимания на то, что всё написано русским языком, то можно считать, что мы уже написали первую программу. В принципе, программа на любой языке программирования выглядит приблизительно так, как мы это описали. В данном случае, наша программа состоит из следующих блоков:

1. Начало программы
2. Описание переменных.
3. Начало кода (заметь, что описание переменных, это не код программы).
4. Заполнение переменных начальными значениями.
5. Запуск цикла от 1 до 5.
6. Выполнение в цикле расчёта.
7. Вывод результата.

В принципе, ничего нового здесь нет. Я просто описал блок-схему словами, похожими на программный язык. В дальнейшем, когда ты сам начнёшь писать свой собственный код, ты должен будешь действовать так же. Сначала построить алгоритм программы в виде блок-схемы или хотя бы мысленно представить алгоритм работы будущей программы и только потом перенести всё это в компьютер.

Не пугайся, не всё так сложно. Я строил такие блок-схемы только на начальном этапе. Сейчас я уже пишу программы без использования всякой дополнительной логики. Ты тоже сможешь так же, если поймёшь, как мыслит машина и сможешь думать её логикой.

Это то же самое, что разговаривать с англичанином. Мало знать английских слов, нужно ещё и мыслить как он. Я сразу вспоминаю случай, который случился со мной в Испании:

Как-то раз, ко мне подошла девушка и, узнав, что я из России стала просить у меня деревянную ложку в качестве сувенира. У меня не было ложек, я просто не брал их с собой. Поэтому мне пришлось сказать ей No, т. е. «Нет». Я это воспринимал, как у меня НЕТ ложки. Она меня спрашивала, почему НЕТ, воспринимая мой ответ, как будто я отказываюсь дать ей эту чёртову ложку. На английском «NO» - это отрицание, а по нашему «Нет» - это не только отрицание, но и признак отсутствия (например, у меня нет ложки). Если бы я тогда мыслил, как англичанин, то я бы смог ответить: «I don’t have». Но этому я научился намного позже, тогда я ещё очень слабо владел английским языком.

Точно так же и машинный язык. В большинстве случаев он схож с человеческим, потому что его создавали люди. Но иногда разница чувствительна, потому что машина не всегда может мыслить, как человек. Поэтому ты должен научится объяснять свои мысли понятным для машины языком.

Я думаю, что мы готовы перейти к изучению программирования. Возможно, в будущем я улучшу эту главу, если увижу, что не все мои читатели смогли понять, что я пытаюсь сказать. Да и в любом случае, сколько не старайся, основная часть знаний прейдёт только на практике. Только практика сможет закрепить описанные мною здесь знания. И всё же, чем больше ты усвоил из сказанного ранее, тем легче будет дальше.

Глава 3. Начальные сведения о Delphi.

Delphi – визуальная среда разработки программ. Она прячет от нас все сложности программирования, превращая его в увлекательный процесс. При создании обычных приложений, тебе не придётся задумываться о регистрах, стеке и многом другом. Но это не значит, что я потратил время зря, рассказывая тебе внутренности компьютеры и принципы работы программы. Если ты захочешь стать профессиональным программистом, то нам пригодится всё.

В этой главе я расскажу:

1. Как установить Delphi 6.
2. Что входит в комплект поставки.
3. Расскажу, как пользоваться оболочкой Delphi 6.

Те, кто уже знаком с Delphi, возможно уже знают большую часть из того, что я буду говорить. Но возможно, я всё же скажу что-то новое. Ну а если нет, то нет. Я пишу библию по Delphi и хочу описать всё. Возможно не сразу и придётся в будущем расширять разделы, но в итоге должна получится книга, которая будет рассказывать обо всём.

3.1. Установка Delphi 6.

Сейчас я постараюсь подробно описать ход установки Delphi 6. Он достаточно прост и у большинства не вызовет никаких проблем, но есть люди, которые только решили заняться программированием или только начали осваивать компьютер. Мы не должны забывать про этих людей, тем более, что все мы когда-то были начинающими.

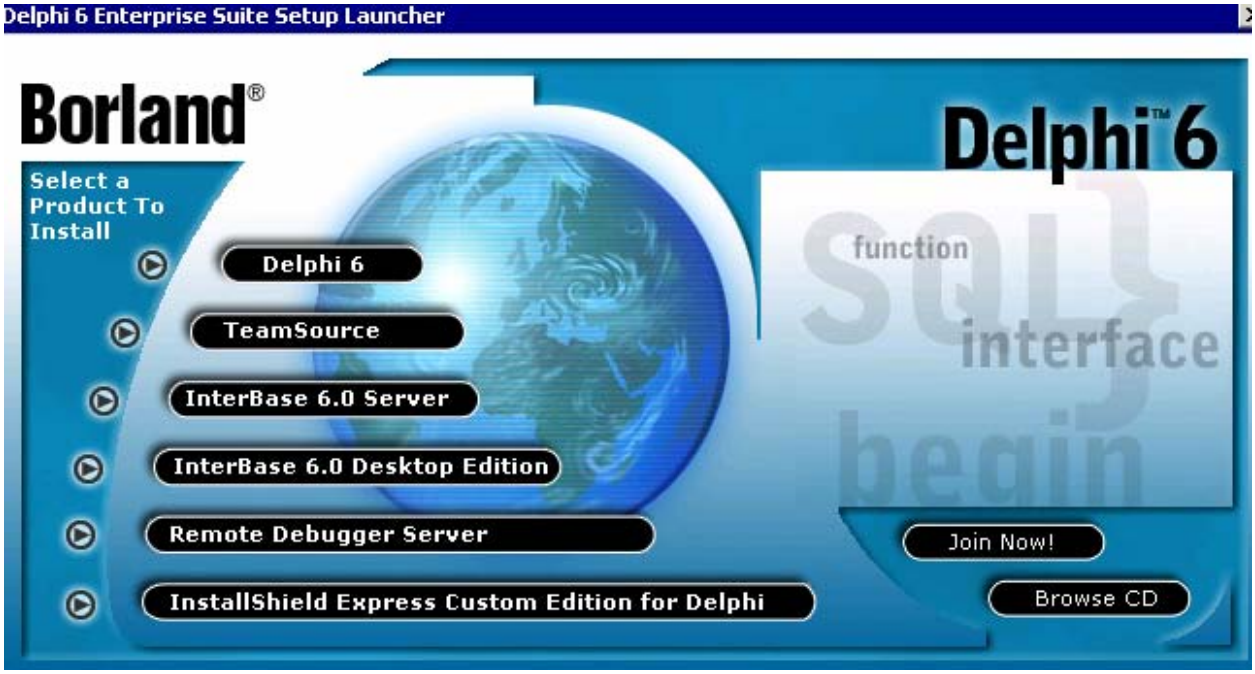


Рис 3.1 Автозапуск программы установки.

Вставь компакт диск с Delphi 6 в CD-ROM. Если ты не отключил автозагрузку, то перед тобой откроется окно автозапуска. Выбери в этом окне пункт Delphi 6. Если автозапуск отключён, но открой содержимое диска и в корне запусти программу install.exe или из директории Install программу setup.exe.

После этого запустится программа установки Delphi 6. Давай последовательно рассмотрим каждый шаг процесса установки. Я буду последовательно расписывать каждое окно. Ты должен выполнять все действия которые я буду описывать и после этого нажимать кнопку “Next” («Далее»), чтобы перейти на следующий шаг.

Некоторых из описываемых мною шагов у тебя может и не быть. Это связано с тем, что я буду описывать процесс установки Enterprise Edition версии Delphi. В Standard или Professional версии некоторые компоненты отсутствуют, поэтому при её установке некоторые окна просто не нужны.

---

1. Окно приветствия в программу установки Borland Delphi 6, Enterprise Edition.

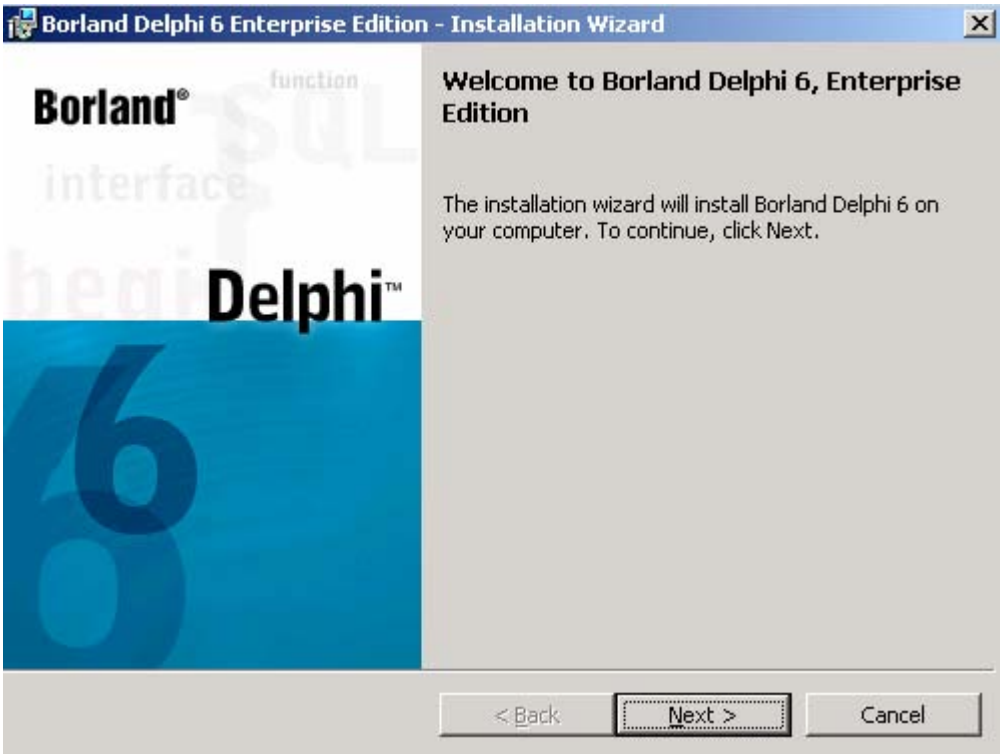


Рис 3.2 Окно приветствия программы установки Delphi 6.

2. Окно ввода серийного номера. Если у тебя не ворованная версия (бывают и такие) с пиратского диска, то вместе с диском должен идти серийный номер (Serial Number) и код авторизации (Authorization Key).

3. Следующее окно содержит лицензию на использование Delphi 6. Все лицензии приблизительно одинаковые, поэтому я её даже не читал. Если хочешь (и знаешь английский язык), то можешь прочитать. Если влом, то просто выдели пункт: «I accept the terms in the license agreement» (я принимаю условия лицензионного соглашения) и нажми «next».

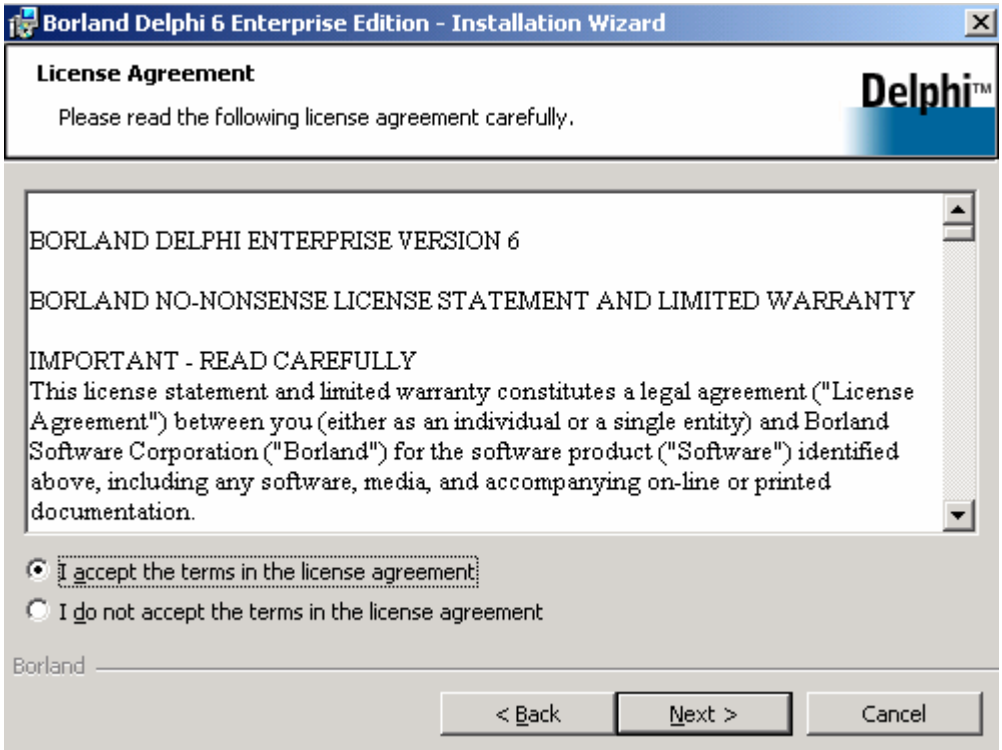


Рис 3.3 Окно лицензионного соглашения.

4. Окно, в котором описаны некоторые замечания по установке Delphi 6. Опять же, ничего особенного здесь не описано. Если ты обладаешь хорошими знаниями английского, то можешь прочитать. Если нет, то и мучится переводить не стоит. У меня с английским не фонтан, хотя и знаю, и всё же я ни разу не читал, что тут написано.

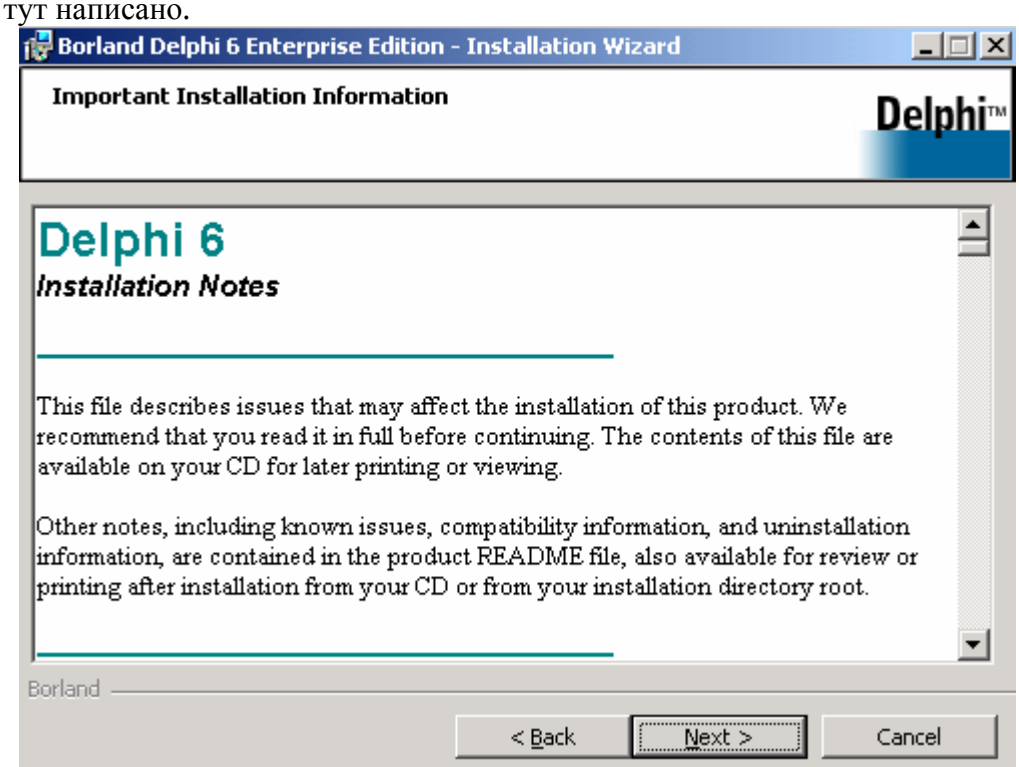


Рис 3.4 Окно замечаний по процессу установки Delphi 6.

5. Следующее окно – выбор типа установки. Тебе, как и везде, доступны три типа:

- **Typical** – типичный. Устанавливается то, что посчитали нужным разработчики.
- **Compact** – компактный. Устанавливается необходимый минимум.
- **Custom** – выборочный. Ты сам можешь настроить, что нужно устанавливать на твой компьютер.

Я предпочитаю настраивать необходимое самостоятельно, поэтому всегда выбираю «Custom» и тебе советую.

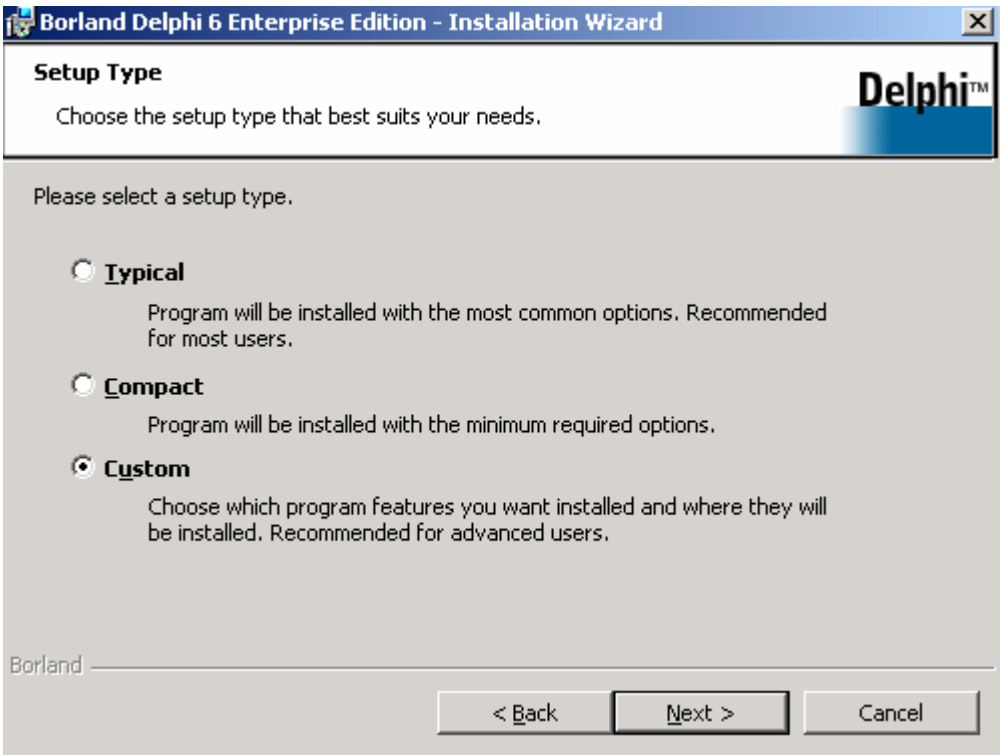


Рис *3.5 Окно выбора типа установки.*

После нажатия кнопки «Next» перед тобой может появиться окно со следующим содержанием:

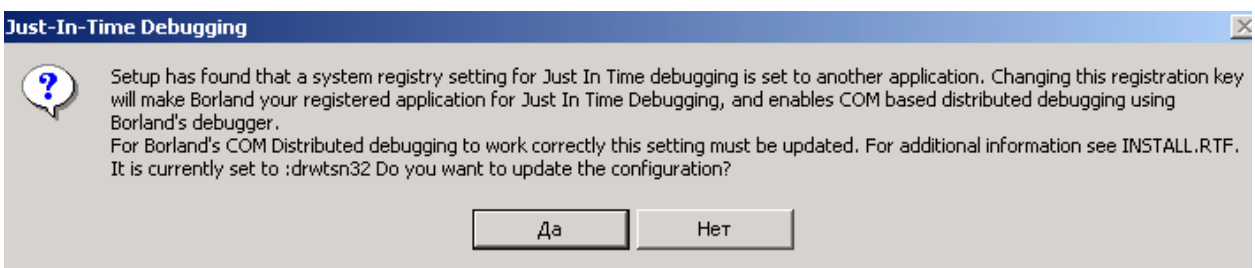


Рис 3.6 Установка системного отладчика.

Здесь написано, что в системе нет, или установлен другой отладчик запущенных прог. Ты наверно не раз уже видел окна типа: «Программа выполнила не допустимую операцию и будет закрыта». В этом окне бывает три кнопки: «Закрыть», «Отладка» и «Помощь». Если ты нажмёшь «Отладка», то должен будет загрузиться отладчик, в

котором можно будет просмотреть, из-за чего произошла ошибка. Правда, тут без знаний языка Assembler не обойтись.

Так вот. Это предупреждение Delphi гласит, что он хочет стать отладчиком в системе. После этого, если ты нажмёшь «отладка» в окне сообщения об ошибке, загрузится Delphi и позволит тебе отладить сбойную программу и найти ошибку. Можешь нажать «Да», если хочешь использовать для отладки Delphi или «Нет», чтобы Delphi отстал от тебя.

6. В этом окне ты можешь выбрать компоненты, которые нужно устанавливать. Для выбора, устанавливать компонент или нет, нужно щёлкнуть по кнопке напротив него и в меню один из пунктов:

- **Install on Local Hard Drive** – инсталлировать на локальный жёсткий диск.
- **Install Entire Feature on Local Hard Drive** - инсталлировать все возможности на локальный жёсткий диск. В принципе, этот пункт не отличается от предыдущего. Просто если ты выберешь его у заголовка ветки, то будет выбрана вся ветка.
- **Do not Install** – не инсталлировать.

По умолчанию выбрано всё, как инсталлировать на жёсткий диск.

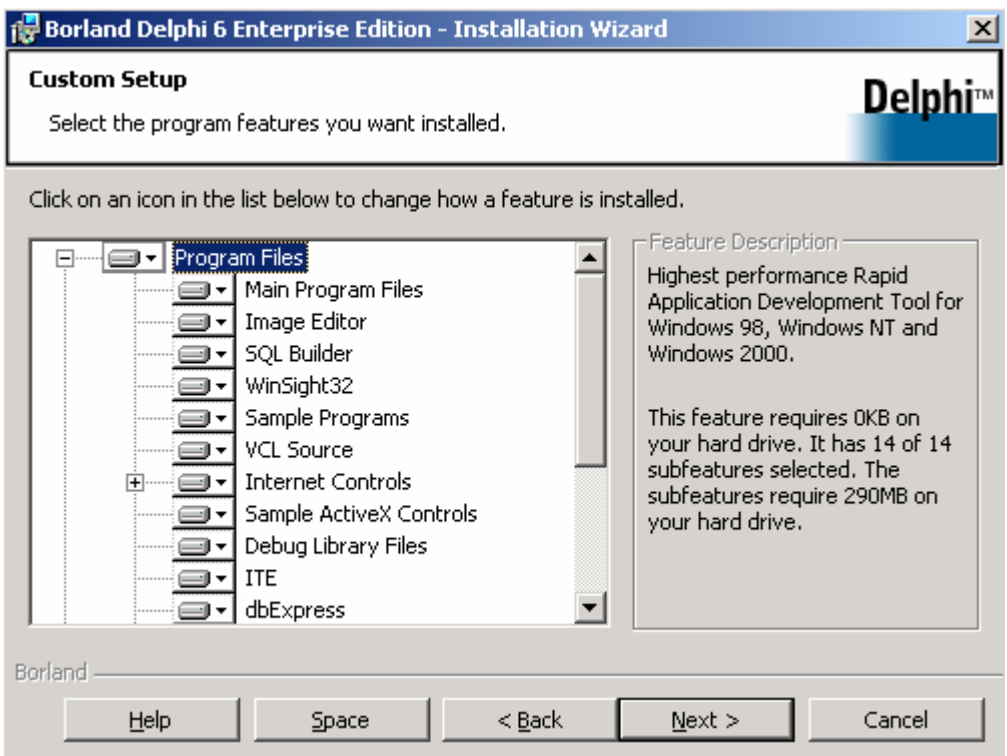


Рис 3.7 Окно выбора компонентов Delphi.

Давай рассмотрим все компоненты, которые тубу доступны. Я оформил их в виде дерева, так же, как они представлены в окне, чтобы тебе легче было разбираться. Те компоненты, которые устанавливать необязательно я буду указывать отдельно.

- **Program Files** – Это заголовок ветки, в которой находятся необходимые для Delphi компоненты.
  - **Main Program Files** –основные файлы Delphi, в том числе и сама визуальная оболочка, в которой ты будешь писать программы.

    - **Image Editor** – программа (графический редактор) для редактирования иконок, курсоров и ресурсов.
    - **SQL Builder** – программа для визуального создания SQL запросов (это запросы к базам данных). *Можешь её не устанавливать, толку от него мало*.
    - **WinSight32** – программа для наблюдения за системой. *Можешь её не устанавливать*.
    - **Sample Programs** – примеры программ написанных на Delphi.
    - **VCL Source** – исходники библиотеки VCL. Полезно только для опытных пользователей. Начинающий не сможет разобраться в исходных кодах VCL.
    - **Internet Controls** – компоненты для работы с Internet.
    - **Sample ActiveX Controls** – простые компоненты ActiveX.
    - **Debug Library Files** – файлы библиотеки отладки программ.
    - **ITE** – нужна для поддержки многоязыковой поддержки.
    - **DbExpress** – компоненты для доступа к базам данных.
    - **Microsoft Office Controls** – компоненты доступа к Microsoft Office.
    - **Help Files** – файлы помощи Delphi.
    - **Indy** – низкоуровневые компоненты для работы с Internet.
- **Shared Files** – разделяемые файлы.
    - **Microsoft SDK Help Files** – файлы помощи Microsoft. Здесь в основном помощь по WinAPI и другим технологиям Microsoft.
    - **Image Files** – Набор картинок, иконок и курсоров.
    - **Sample Data Files** – примеры баз данных. Эти базы используются в примерах поставляемых с Delphi. Я эти базы использовать не буду, поэтому нам они не нужны. Если ты будешь самостоятельно разбираться с примерами из Delphi, то возможно они тебе понадобятся.
    - **Debugger** – отладчик приложения.
- **BDE** – библиотека доступа к базам данных. Она старая и кривая, поэтому я её использовать не рекомендую. Вместо этого мы будем работать с ADO и DbExpress. Хотя в книги я дам описание этой технологи, потому что для доступа к старым базам данных (Paradox, DBF) она работает нормально.
- **Database Desktop** – программа для создания баз данных. Этой программой хорошо создаются только старые базы данных (Paradox, DBF).

7 ● Если ты выбрал установку BDE, то следующим окном у тебя будет выбор баз данных, поддержку которых ты хочешь увидеть. Не смотря на то, что в этом списке есть все современные базы данных, я не советую тебе организовывать доступ к ним через BDE. Она работает нестабильно и сама фирма Borland согласна с этим.

Начиная с Delphi 6, фирма Borland включила в поставку очень хорошую и удобную библиотеку dbExpress. Именно её желательно использовать для доступа к базам данных. Ещё, можно использовать ADO. Я постараюсь в этой книге дать описание всем этим технологиям.

В любом случае, если ты решил установить на всякий случай BDE, то выбери драйверы, которыми ты будешь пользоваться. В списке в основном показаны редкие вещи, которые врят ли у тебя установлены, поэтому, от того что ты выберешь погода не измениться. Единственное, что надо правильно указать – какую версию DAO ты будешь использовать. Тебе доступны: DAO 3.0 (используется в MS Office 95), и DAO 3.5 (используется в MS Office 97). Обязательно укажи правильную версию, установленного у

тебя MS Office. Если у тебя установлен Office 2000, то укажи “DAO 3.5 (MS Access 97) Driver”.

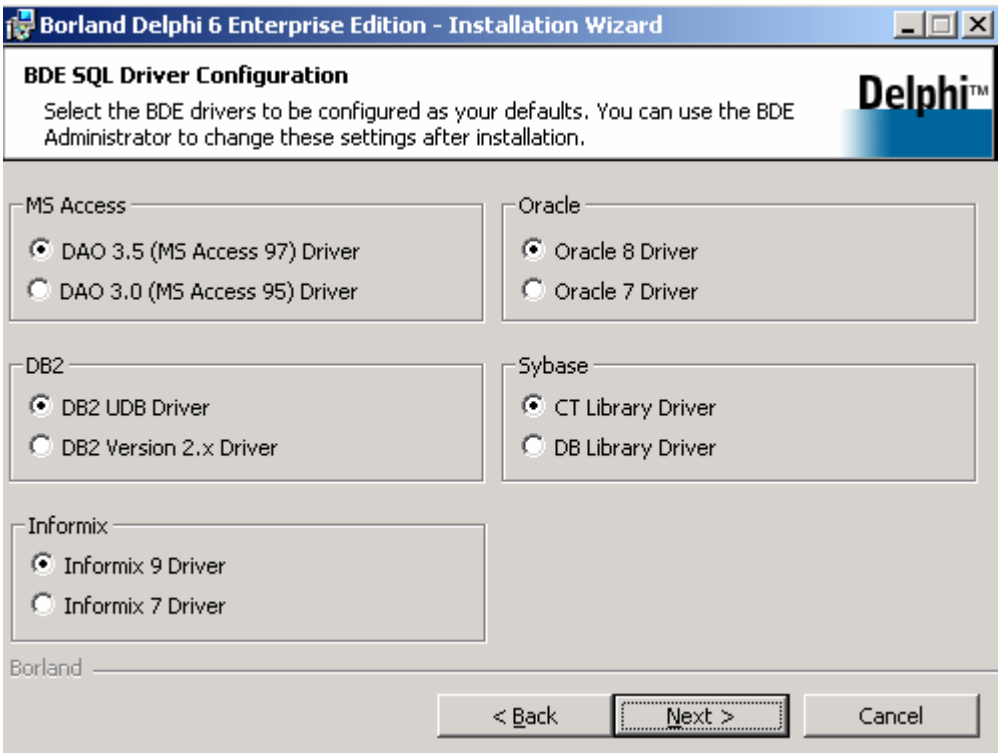


Рис *3.8 Окно настроек драйверов баз данных*

8. Следующее окно предлагает тебе установить CORBA – это технология распределённых вычислений. Я наверно не буду её рассматривать в этой книге, поэтому можешь её не ставить и выбрать пункт “No VisiBroker/CORBA Support”. Если ты решил установить себе CORBA, то выбери необходимую тебе версию.

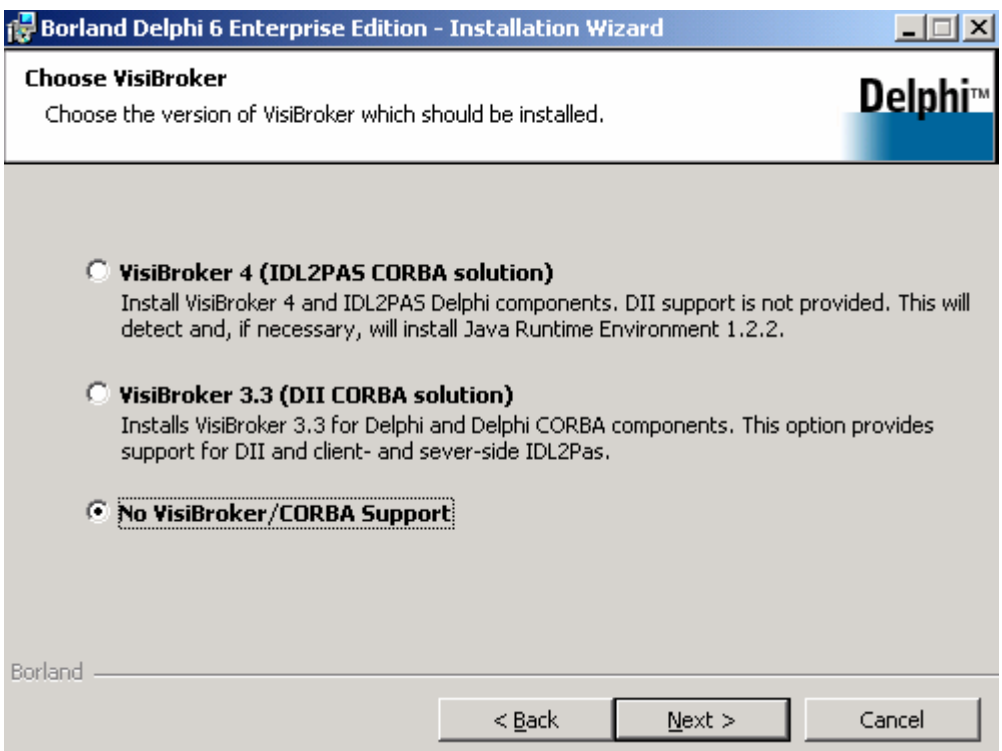


Рис 3.9 Окно выбора версии CORBA.

9. Выбор версии компонентов Microsoft Office. Если ты выбрал установку «Microsoft Office Controls», то здесь ты должен указать версию компонентов.

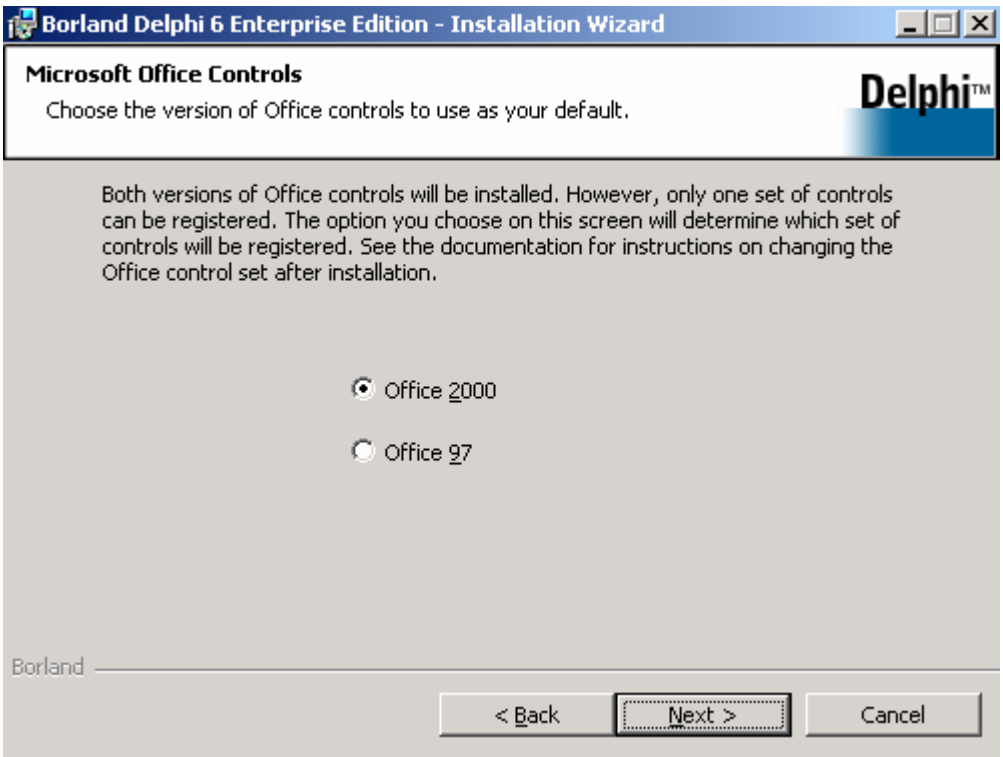


Рис *3.10 Выбор версии компонентов Microsoft Office.*

10. Следующее окно – дополнительная лицензия. Можешь почитать, что тут написано, но ничего нового не узнаешь. Просто выдели пункт «I accept the terms in the license agreement» (я принимаю условия лицензионного соглашения) под лицензионным соглашением и нажми «next». Я даже не буду показывать это окно, потому что там просто ничего особенного нет.

11. В этом окне ты можешь выбрать пути, куда будут установлены различные компоненты Delphi. Если у тебя достаточно места на системном диске, то желательно всё установить туда. Хотя и на других дисках проблем не замечалось.

Здесь у тебя четыре пути:

- **Program Files** – путь куда будут установлены программные файлы.
- **Shared Files** – путь куда будут установлены разделяемые файлы.
- **BDE and SQL Links** – путь куда будет установлен BDE.
- **Database Desktop** - путь куда будет установлен Database Desktop.

Как видишь, здесь разделение такое же, как и при выборе необходимых при установке компонентов. Если ты при выборе компонентов не выбрал установку BDE, то и соответствующего пути здесь не будет.

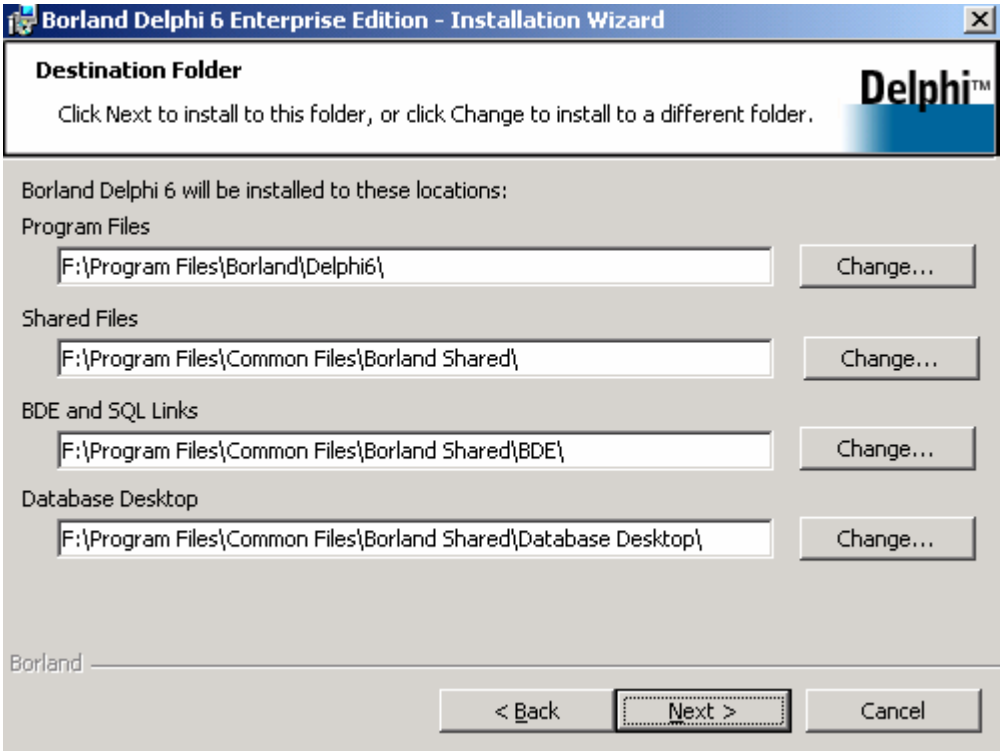


Рис 3.11 Окно выбора пути установки.

12. В этом окне тебя спросят: «Сохранять результат установке на жестком диске?». Фирма Borland рекомендует делать это, но если у тебя проблемы с дисковым пространством, то можешь не устанавливать.

13. Уведомление о том, что сейчас начнётся установка Delphi 6 на твой жёсткий диск. Просто нажми “Next”. После этого начнётся копирования выбранных файлов на твой компьютер. Подожди немного, установка не такая уж и долгая, всё зависит от скорости твоего привода CD-ROM и компьютера.

3.2 Замечание по установке в Windows 2000.

В Windows 2000 иногда возникают проблемы с установкой. Особенно это проявлялось в Delphi 5. При попытке запустить setup.exe программа просто не запускалась, не выдавая никаких сообщений. Это связано с неправильной обработкой пути временных директорий.

Для исправления ошибки нужно щёлкнуть правой кнопкой по «Мой компьютер» и выбрать в появившемся меню пункт «Свойства». Перед тобой откроется окно «свойства системы». В этом окне перейди на закладку «Дополнительно и нажми на кнопку «Переменные среды». Перед тобой откроется окно, как на рисунке 3.11. Здесь находятся настройки всех путей в системе. На первый взгляд всё нормально, но если дважды щёлкнуть по какому-нибудь пути, то появится окно его редактирования в котором путь выглядит совершенно по другому «%USERPROFILE%\Local Settings\Temp». Именно вот это: «%USERPROFILE%» не может понять инсталлятор в Delphi. Измени на нормальный путь все пути Tmp и Temp. Только обязательно указывай на существующую директорию иначе некоторые программы не смогут запускаться.

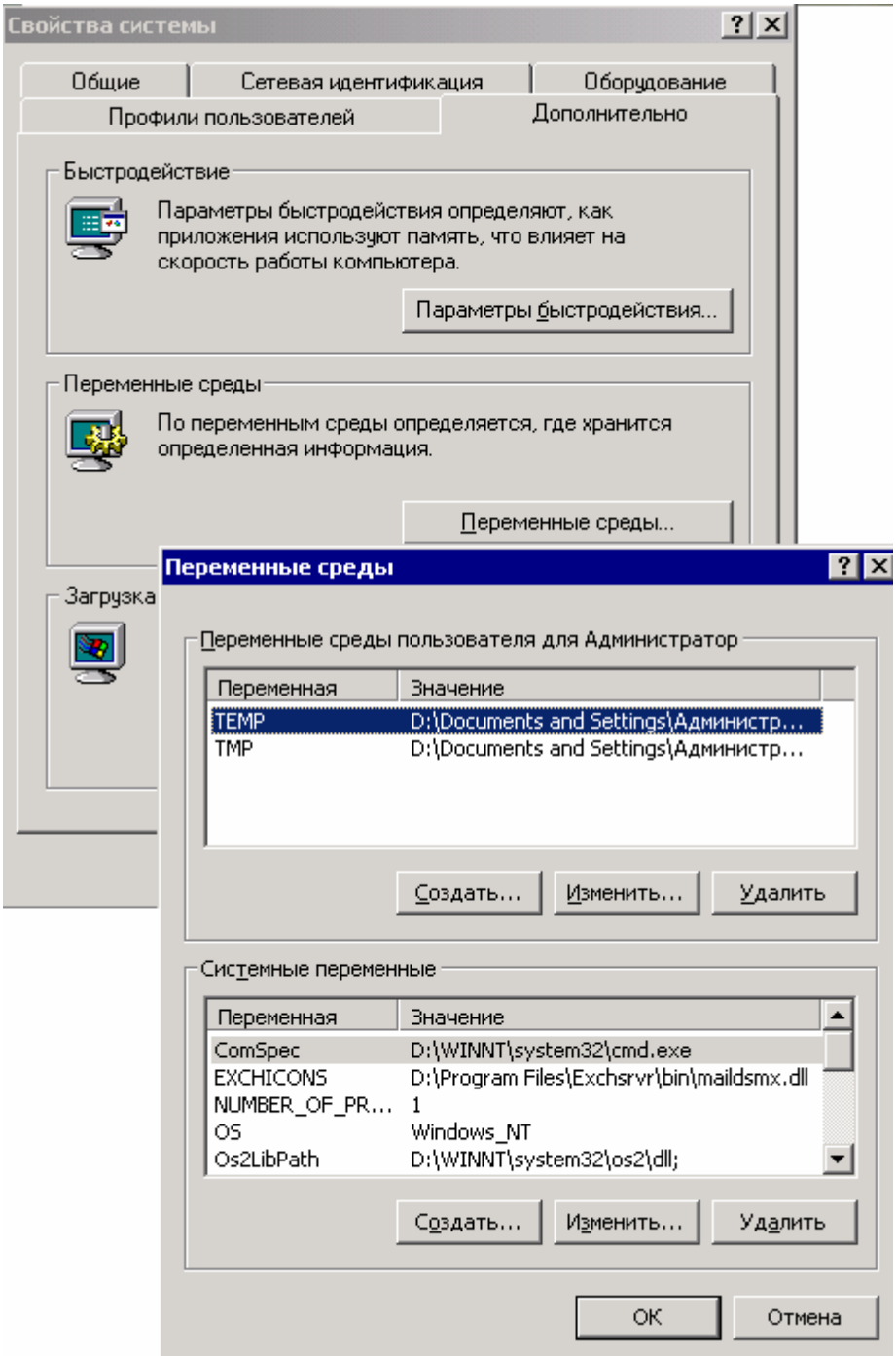


Рис 3.12 Переменные среды.

Эта проблема проявляется не только в Delphi, а в большинстве программ собранных инсталлятором InstallShield старой версии.

В Delphi 6 эта проблема пока не замечалась. Пока что он устанавливался на все машины без проблем.

3.3 Оболочка Delphi 6.

Пора в первый раз запустить Delphi 6. Для этого выбери из меню Пуск -> Программы -> Borland Delphi 6 -> Delphi 6.

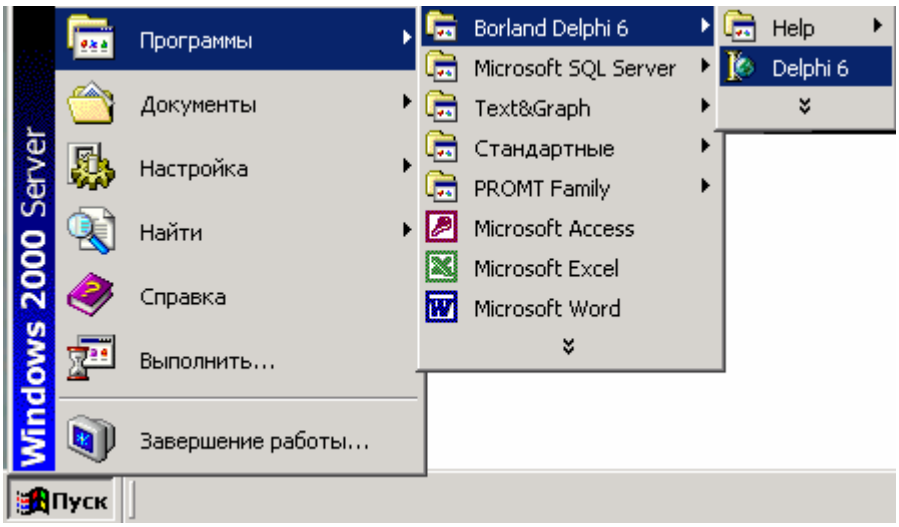


Рис 3.13 Запуск Borland Delphi 6

После запуска перед тобой откроется окно, похожее на рис 3.14.

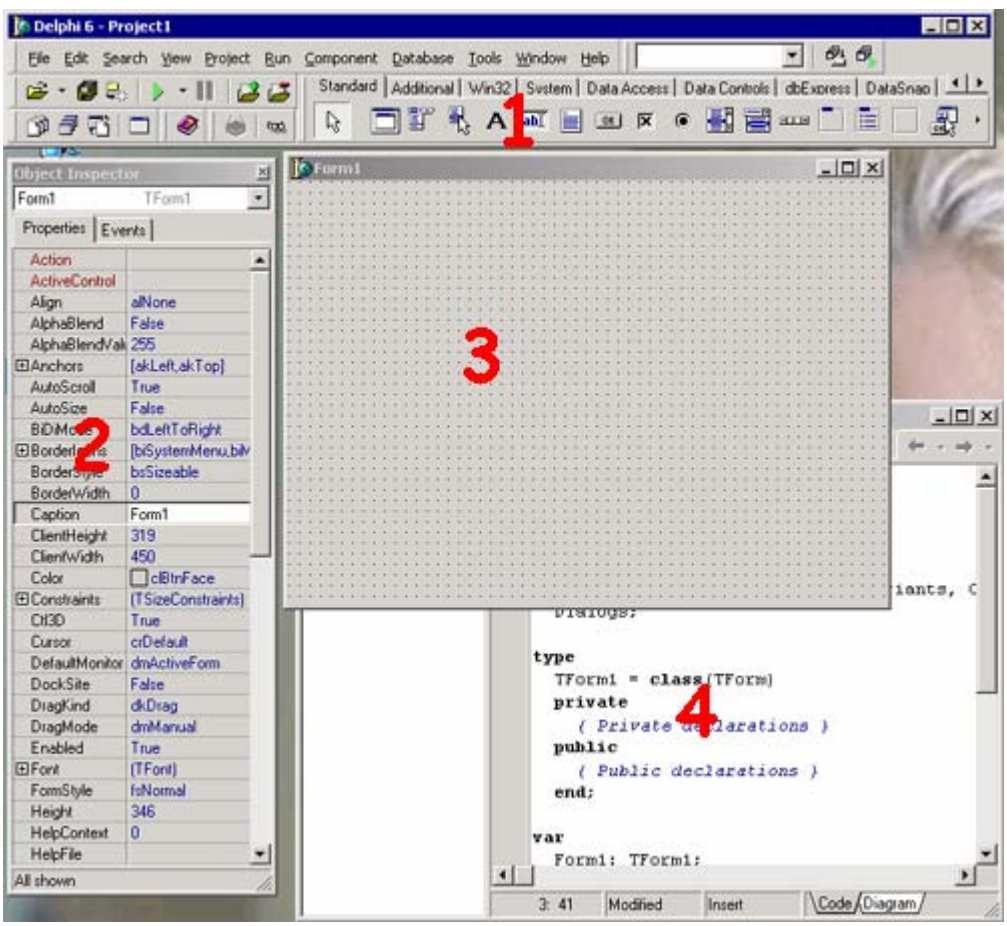


Рис. 3.14 Главное окно Delphi 6.

Перед тобой четыре основных окна, с которыми ты будешь постоянно работать:

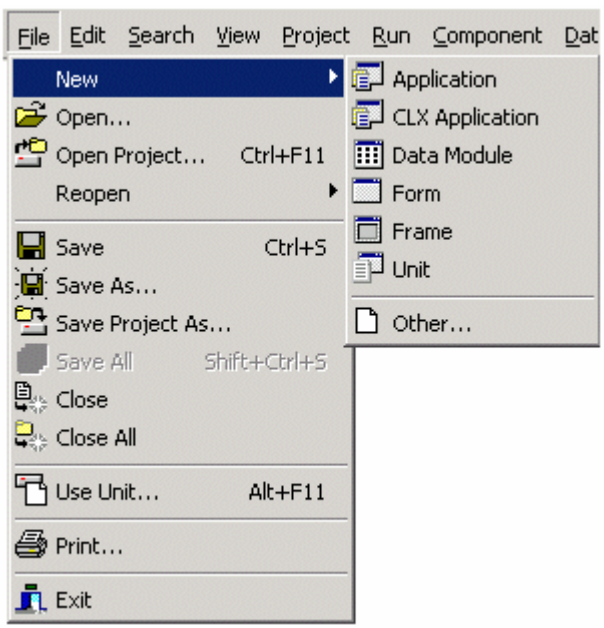
1) Главное окно программы. На нём находится основное меню, панели инструментов и палитра компонентов.
2) Объектный инспектор. Он предназначен для управления объектами и состоит из двух закладок:
   - Properties – на этой закладке будут перечислены свойства выделенного объекта. Имя и тип выделенного объекта отображается в выпадающем списке, вверху окна.
   - Events – события. Здесь можно создавать и изменять реакцию объекта на различные события.
3) Форма. Это уже готовая визуальная форма будущей программы.
4) Редактор кода. В этом окне ты будешь писать программу на языке Delphi.

Есть ещё одно окно *Object TreeView*, но о нём я расскажу позже.

3.4 Главное меню.

Давай рассмотрим основное меню Delphi 6. Оно достаточно сильно отличается от предыдущих версий за счёт некоторых удобных усовершенствований. Давай рассмотрим основные пункты, которые нам понадобятся в ближайшее время.

В меню «File» ты можешь увидеть следующие пункты:



- **New** – создание нового проекта, формы или шаблона. Если ты наведёшь на этот пункт, то перед тобой раскроется подменю, в котором можно увидеть:
  - *Application* – создать новое приложение.
  - *CLX Application* – создать новое CLX приложение.
  - *Data Module* – создать Data Module – модуль данных.
  - *Form* – создать новую форму.
  - *Frame* – создать новый кадр.
  - *Unit* – создать новый модуль.
- **Open** – открыть существующий файл поддерживаемый Delphi 6
- **Open Project** – открыть существующий проект.
- **Reopen** – повторно открыть.
- **Save** – сохранить текущий модуль.

- **Save As** – сохранить текущий модуль под новым именем.
- **Save Project As** – сохранить проект под новым именем.
- **Save All** – сохранить всё.
- **Close** – закрыть текущий модуль.
- **Close All** – закрыть всё.
- **Use Unit** – использовать модуль.
- **Print** – печатать модуль.
- **Exit** – выход.

Я несколько раз употреблял здесь слово *модуль*, поэтому я думаю, что надо бы пояснить, что это такое. ***Модуль*** – Файл, содержащий код программы или часть кода. Чаще всего это простой текстовый файл. Сейчас под модулем стали понимать и файлы, содержащие визуальную часть программы. Хотя код и визуальная часть хранятся в разных файлах, они неразделимы.

Меню «Edit» содержит все основные команды редактирования текста, плюс специфичные команды работы с визуальными объектами. Если ты работал с векторной графикой, то для тебя не составит проблем разобраться с ними. Если возникнут какие-то проблемы, то ты успеешь ещё всё нагнать на практике.

Я сейчас не буду рассматривать все команды, потому что всё равно они не запомнятся. Ты сможешь их запомнить, только если будешь всё зубрить. Но это глупо. Когда мы перейдём к практике и начнём писать программы, всё само отложится в твоей памяти. Поэтому я только назову некоторые команды редактирования текста:

- **Undo** – отменить последнюю операцию ввода текста.
- **Redo** – повторить последнюю операцию ввода текста.
- **Cut** – вырезать выделенный текст в буфер обмена.
- **Copy** – копировать выделенный текст в буфер обмена.
- **Paste** – вставить в текущую позицию курсора содержимое буфера обмена.
- **Delete** – удалить выделенный фрагмент.
- **Select All** – выделить всё.

Для выделения текста, в редакторе, надо установить курсор на начало выделяемого фрагмента, нажать клавишу Shift и стрелками на клавиатуре перевести курсор в позицию конца выделяемого фрагмента.

Остальные меню я не буду рассматривать, потому что это только забьёт твою голову лишней информацией и ничего не отложится. То что я рассказал, для начала будет достаточно, а остальное мы рассмотрим по мере надобности.

3.5 Настройка Delphi 6

В Delphi достаточно много настроек и все их знать не обязательно. Мы познакомимся только с теми, которые действительно могут на что-то повлиять и что-то улучшить.

Для изменения основных настроек, нужно выбрать из меню Tools пункт Environment Options (настройки окружения). Перед тобой откроется окно, как на рисунке 3.15.

На первой странице Preferences (Предпочтения) ты можешь указать следующие опции:

- **Autosave options** – опции авто-сохранения. Здесь есть два пункта:
  - ***Editor Files*** – редактируемых файлов (модулей). Если ты поставишь галочку напротив этого пункта, то модули будут сохраняться автоматически.

- ***Project desktop*** – рабочей среды. Если ты поставишь галочку напротив этого пункта, то состояние всех окон будет сохраняться автоматически.

- **Compiling and Running** – настройки процесса компилирования и запуска готовой программы. Здесь доступны следующие параметры:
  - **Show compiler progress** – во время компиляции показывать окно состояния. В этом окне отображается информация о процессе компиляции и результате. Это окно очень удобно, особенно для начинающих. Я сам им очень часто пользуюсь, когда пишу маленькие программы. Единственный недостаток – компиляция проходит немного дольше. При маленьких проектах это незаметно, но с большими программами задержка может быть ощутимой.
  - **Warn on package rebuild** – предупреждать, когда необходима перекомпиляция пакета. Лично я не разу не встречался с ситуацией, когда это сообщение помогло бы мне. Так что я это отключил.
  - **Minimize on run** – минимизировать оболочку, когда запущена программа. Действует, когда ты запускаешь из Delphi. Если ты запустишь скомпилированную программу из проводника, то Delphi не будет минимизирован.
  - **Hide designers on run** – прятать окна объектного инспектора и визуальной формы при запуске программы. По умолчанию этот параметр выставлен, но я советую тебе его отключить.

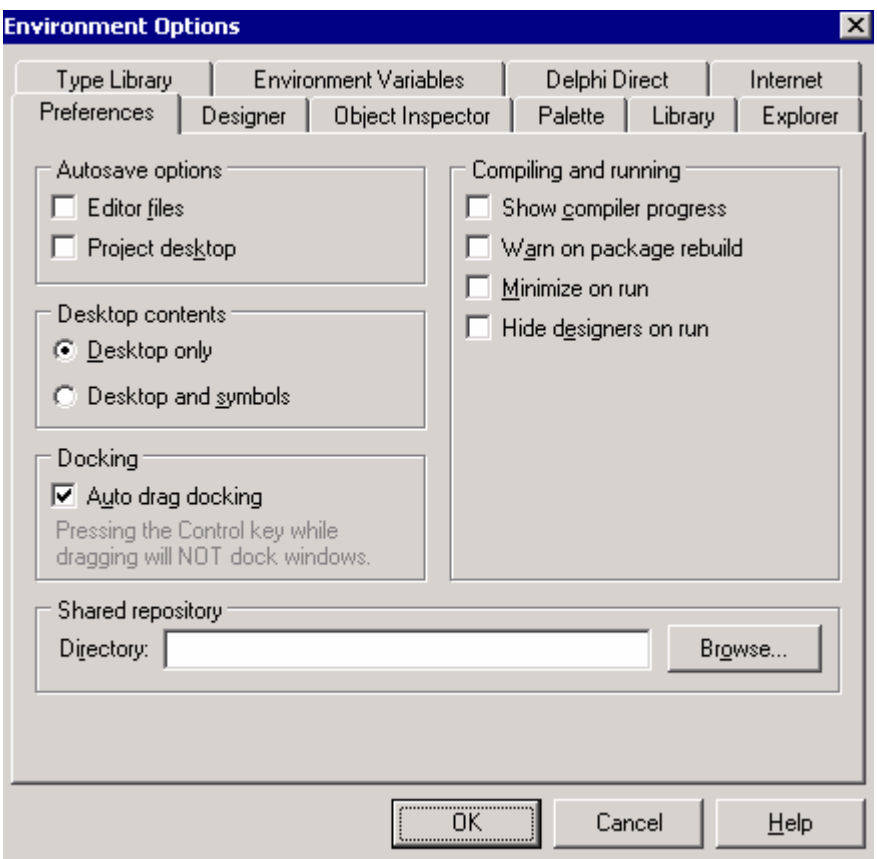


Рис. 3.15. Настройки окружения

Остальные параметры не так интересны. Единственное, на чём я хочу ещё остановиться – окно процесса компиляции.

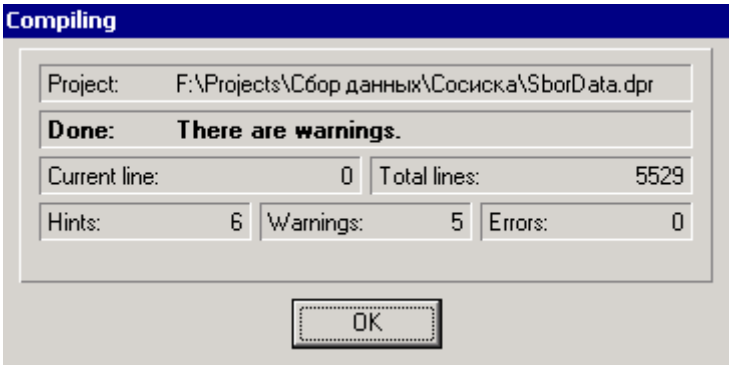


Рис 3.16. Окно, отображающее процесс компиляции.

Как я уже говорил, окно действительно удобно и его желательно включать. Как только ты попросишь Delphi скомпилировать твою программу, перед тобой появится окно, как на рисунке 3.16. В этом окне удобно отображается состояние компиляции. Тебя здесь должно интересовать три значения:

- **Hints** – сообщения. Это простые сообщения, которые указывают на места, где можно улучшить код. Например, ты объявил переменную, но не пользовался ей. В этом случае появится соответствующее сообщение. Это, конечно же, не ошибка и программа всё же будет скомпилирована. Но зато благодаря этим сообщениям ты можешь увидеть, где ты объявил лишнее или возможно просто забыл.
- **Warning** – предупреждения. На них уже нужно обращать более пристальное внимание. Например, ты объявил переменную, затем попытался её использовать, не присвоив начальное значение. В этом случае появится предупреждение. Это опять же не ошибка и программа будет скомпилирована, но Delphi предупреждает тебя о возможной ошибке. Такие предупреждения нужно проверять, потому что ты действительно мог забыть что-то сделать и это может привести к фатальной ошибке.
- **Errors** – это уже самые настоящие ошибки. Они указывают на те места, где ты допустил грубую ошибку, из-за чего программа не может быть скомпилирована.

Даже если ты откажешься от показа окна состояния компиляции, ты всё равно увидишь все сообщения, ошибки и предупреждения. С этим окном просто удобнее.

Теперь перейдём на закладку Designer окна настроек окружения (рис 3.17).

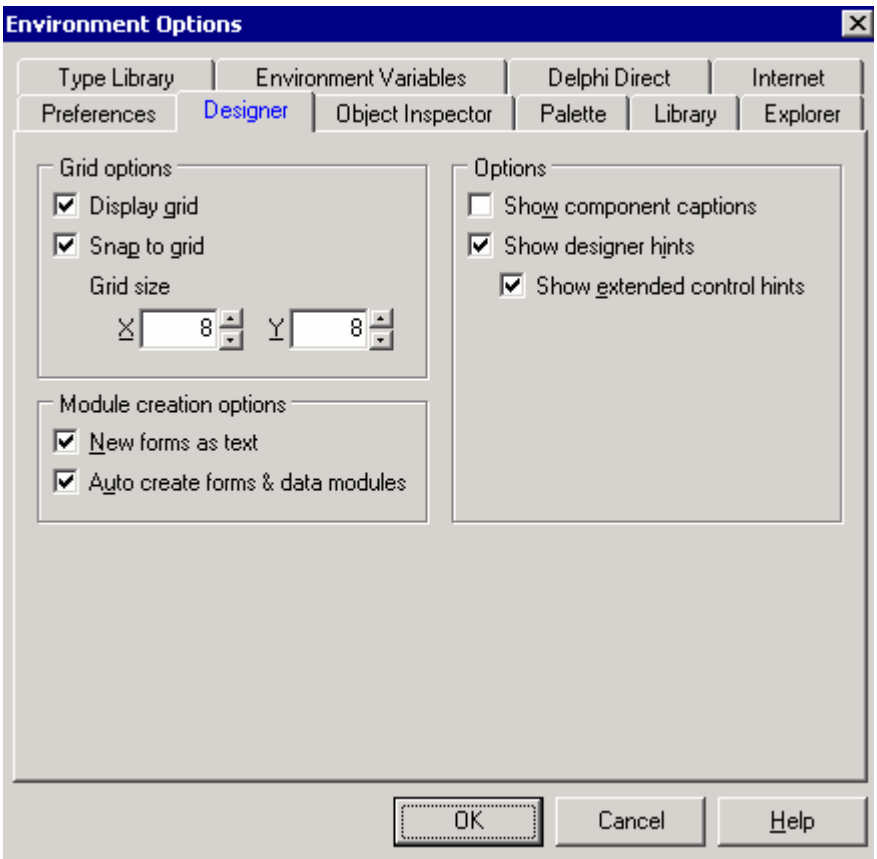


Рис 3.17 Закладка Designer окна настроек окружения.

Здесь очень интересными являются параметры:

- **Display grid** – показать сетку.
- **Snap to grid** – перемещать объекты по сетке.

Я советую тебе постоянно использовать сетку. Это позволит тебе улучшить внешний вид твоей программы. По умолчанию сета выглядит как 8x8 пикселей. Если ты захочешь изменить это значение, то советую тебе установить значения кратные 2.

Всё, больше в этом окне я ничего не буду расписывать, потому что настройки по умолчанию очень удобны, а их расписывание занимает лишнее время и место. Так что нажимай «ОК» и мы двинемся дальше.

Глава 4. Визуальная модель Delphi.

Я уже не раз говорил, что Delphi это визуальная среда разработки программ. Это значит, что большую часть оформления внешнего вида ты будешь делать с использованием мышки, расставляя необходимые объекты на дизайнере форм.

Прежде чем начать расставлять эти объекты, и знакомится с ними поближе, я постараюсь тебе рассказать немного теории об объектах и компонентной модели Delphi. Это основа, которую должен знать и понимать любой программист. Я уже говорил, что состряпать простую программу можно научить даже обезьяну, но для самостоятельного написания настоящих программ необходимо понимание всех основ. Иначе ты сможешь только повторять описанные мною шаблоны и не сможешь их кардинально изменить или улучшить.

В этой главе я постараюсь дать все необходимые базовые знания, которые в последствии мы применим на практике уже в следующей главе это книги.

В этой главе мы пока ещё будем использовать абстрактный язык программирования. Но, начиная уже со следующей главы, мы уже начнём знакомиться с Delphi.

4.1 Процедурное программирование.

Если ты читаешь книгу полностью, то наверно уже прочёл про историю языков программирования. С развитием языков программирования развивались и технологии, используемые при написании кода.

Первые программы писались сплошным текстом. Эта была простая последовательность команд, записанная в столбец. Всё это выглядело приблизительно так:

Команда 1
Команда 2
Команда 3
…
…
Команда N.

Таким образом, можно сделать очень мало. Единственное, что было доступно программиста для создания логики – условные переходы. У*словные переходы* – переход на какую-то команду при определённых условиях. Например:

Если выполнено условие, то перейти на команду 1 иначе на команду 3.
Команда 1
Команда 2
Команда 3

…
Команда N.

Это единственная логика, с помощью которой можно было выполнять определённые действия, зависящие от каких-то ситуаций. Но программы оставались плоскими и неудобными.

Следующим шагом стал процедурный подход. При этом подходе какой-то код программы мог объединяться в отдельные блоки. После этого, такой блок команд можно вызывать из любой части программы. Например:

```
Начало процедуры 1
  Команда 1
  Команда 2
Конец процедуры 1

Начало программы
  Команда 1
  Команда 2
  Если выполнено условие, то выполнить код процедуры 1.
  Команда 3
Конец программы.
```

Таким образом, появилась возможность использовать один и тот же код в одной программе неоднократно. Код программ стал более удобным и легче для восприятия.

В процедуры можно передавать различные значения, заставляя их что-то считать:

```
Начало процедуры 1 (Переменная 1: строка)
  Команда 1
  Команда 2
Конец процедуры 1

Начало программы
  Команда 1
  Команда 2
  Если выполнено условие, то выполнить код процедуры 1.
  Команда 3
Конец программы.
```

В этом примере я после начала процедуры, в скобках указал тип передаваемой переменной. Таким образом, в процедуру можно засунуть код какой-нибудь математики, а потом только передавать ей разные значения.

Сразу хочу заметить, что использование процедуры часто называют «Вызов процедуры». Это действительно так процедура как бы вызывается.

Давай рассмотрим пример процедуры приближённый к реальности:

```
Начало процедуры 1 (Переменная 1: Целое число)
  Посчитать факториал числа находящегося в Переменная 1.
  Вывести результат на экран.
Конец процедуры 1

Начало программы
  Процедура 1 (10)
  Процедура 1 (5)
  Процедура 1 (8)
Конец программы.
```

В этом примере я условно написал процедуру, которая вычисляет факториал переданного ей значения и потом выводит результат на экран. В самой программе я просто использую эту процедуру **Процедура 1 (10)**, передавая ей разные значения.

Вот так я получил универсальную процедуру, которая считает факториал и выводит результат на экран. Как это всё работает? Давай рассмотрим алгоритм:

1. Начало программы.
2. Вызов процедуры **Процедура 1 (10)**, и передача ей значения 10.
3. Процедура вычисляет факториал числа 10 и выводит результат на экран.
4. Выход из процедуры и продолжение выполнения программы.
5. Вызов процедуры **Процедура 1 (5)**, и передача ей значения 5.
6. Процедура вычисляет факториал числа 5 и выводит результат на экран.

И так далее. Как видишь, код программы очень удобный.

Но процедуры – это мелочи жизни. Более удобным стало использование функций. *Функция* – это та же процедура, только она умеет возвращать значение, то есть результат своего выполнения.

```
Начало Функции 1: Тип возвращаемого значения – целое число
  Команда 1
  Команда 2
Конец Функции 1

Начало программы
  Команда 1
  Команда 2
  Если выполнено условие, то выполнить код процедуры 1.
  Команда 3
Конец программы.
```

Заметь, что после двоеточия идёт описание типа возвращаемого значения.

Теперь рассмотрим ту же ситуацию с факториалом. Допустим, нам надо рассчитать факториал для последующего использования, но не для вывода на экран. Такая задача легко решается с помощью функций:

```
Начало Функции 1 (Переменная 1: Целое число): Целое число
  Посчитать факториал числа находящегося в Переменная 1.
  Вернуть результат расчёта.
Конец Функции 1

Начало программы
```

Переменная 1:=Функция 1 (10)
Переменная 2:=Функция 1 (5)
Переменная 3:= Переменная 1+Переменная 2
Вывести на экран Переменную 3
Конец программы.

В этом примере я в «переменную 1» записывается результат расчёта факториала 10! (**Переменная1:=Функция 1 (10)**). В «переменную 2» записывается результат расчёта факториала 5!. Потом я складываю значения обеих переменных и записываю результат в переменную 3. И последнее – вывод на экран переменной 3.

4.2 Объектно-ориентированное программирование.

Следующим шагом в развитии технологий программирования было появление объектно-ориентированного программирования. Здесь уже код перестал быть «плоским». Теперь уже программист оперирует не просто процедурами и функциями, а целыми объектами.

Объект – совокупность свойств и методов и событий. Что означает «совокупность»? Это значит, что объект состоит из свойств методов и событий.

Свойства – это простые переменные, которые влияют на состояние объекта. Например, ширина, высота – это свойства объекта.

Методы – это те же процедуры и функции, т.е. это то, что объект умеет делать (вычислять). Например, объект может иметь процедуру для вывода какого-то текста на экран. Эта процедура и есть метод объекта.

События – это те же процедуры и функции, которые вызываются при наступлении определённого события. Например, если изменилось какое-то свойство объекта, может быть сгенерировано соответствующее событие и вызвана процедура для обработки реакции на это событие.

Простой объект выглядит так:

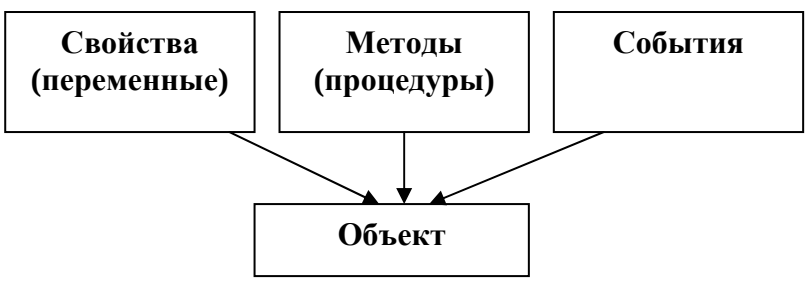


Рис 4.2.1 Графическое представление объекта.

Давай рассмотрим простой объект – кнопка. Такой объект должен обладать следующим минимальным набором:

Свойства:

1. Левая позиция (X).
2. Верхняя позиция (Y).
3. Ширина.
4. Высота.
5. Заголовок.

Методы:

1. Создать кнопку.
2. Уничтожить кнопку.
3. Нарисовать кнопку.

События:
1. Кнопка нажата.
2. Заголовок кнопки изменён.

Объект работает как единое целое свойств, методов и событий. Например, ты изменил заголовок кнопки. Объект генерирует событие ***«Заголовок кнопки изменён»***. По этому событию вызывается метод ***«Нарисовать кнопку»***. Этот метод рисует кнопку в позиции, указанной в свойствах объекта и выводит на кнопке текст, указанный в свойстве ***«Заголовок»***.

У каждого объекта обязательно присутствуют два метода: ***«создать объект»*** и ***«уничтожить объект»***. Во время создания объекта происходит выделение памяти для хранения необходимых свойств, и заполняются значения по умолчанию. Во время уничтожения объекта происходит освобождение выделенной памяти.

Метод для создания объекта называется *конструктором* (constructor). Метод для уничтожения объекта называется деструктором (destructor).


*Процесс создания объекта называется **инициализацией**.*

Теперь рассмотрим процесс применения нашей кнопки. Он выглядит следующим образом:
1. Создание кнопки с помощью вызова метода ***«Создать кнопку»***.
2. Изменение необходимых свойств.

Всё. Наша кнопка готова к работе.


***Объект** – это сложный тип. Это значит, что ты можешь объявлять переменные типа «объект» (как мы объявляли переменные типа число или строка) и обращаться к объекту через эту переменную.*

На языке программирования это будет выглядеть немного сложнее:
1. Объявить переменную типа Кнопка.
2. В эту переменную проинициализировать объект.
3. Можно использовать объект.

Вот тут нужно собрать всё, что я уже говорил об объектах в кучу и дать небольшие пояснения, чтобы мы могли двинуться дальше.

Итак, мы можем объявлять переменные типа «объект». Давай объявим переменную *Объект1* типа *Кнопка*. Теперь мы можем создать кнопку, для чего есть конструктор (метод для создания объекта), который выделяет новую память для объекта. Процесс инициализации объекта кнопки выглядит так: переменной *Объект1* нужно присвоить результат работы конструктора объекта *Кнопка*. Конструктор выделит необходимую

объекту память и присвоит свойствам значения по умолчанию. Результат этого действия будет присвоен переменной *Объект1*.

После всех этих действий, мы можем получить доступ к созданному объекту через переменную *Объект1*.

Давай напишем небольшую программу на русском языке, которая опишет весь процесс создания объекта:

Начало программы.

Переменные:
Объект1 – Кнопка;

Начало кода
Объект1:= Кнопка.Создать объект
Объект1.Заголовок:='Привет'
Объект1.Уничтожить объект.
Конец кода

Доступ к свойствам и методам объектов осуществляется как *ИмяПеременнойТипаОбъект.Свойсво* или *ИмяПеременнойТипаОбъект.Метод* (имя объекта – точка – свойство или метод). Таким образом мы изменили в вышеуказанном примере свойство заголовок (*Объект1.Заголовок*) присвоив ему значение *'Привет'*. Точно так же мы получили доступ к конструктору (метод «Создать объект») и деструктору (метод «Уничтожить объект»).

Создание объекта – обязательно. Если ты попробуешь использовать следующий код, то у тебя произойдёт ошибка:

Начало программы.

Переменные:
Объект1 – Кнопка;

Начало кода
Объект1.Заголовок:='Привет'
Конец кода

Здесь я просто пытаюсь изменить заголовок без создания и уничтожения объекта. Это ОШИБКА!!! Переменная *Объект1* ничего не хранит. Её просто необходимо сначала проинициализировать.

Без инициализации можно использовать только простые переменные, такие как число или строка. Тут Delphi выделяет под них память автоматически, потому что размер этих переменных - фиксированный и Delphi уже на этапе компиляции знает, сколько памяти нужно отвести под переменную.

Сложные переменные типа объектов обязательно должны инициализироваться. Это связано ещё и с размещением данных. Простые переменные хранятся в стеке (сегмент стека), а сложные переменные типа объектов хранятся в памяти компьютера. Как я уже говорил, при старте

программы, сегмент стека инициализируется автоматически. Поэтому переменные могут спокойно размещаться в уже подготовленной памяти сегмента стека. Когда ты создаёшь объект, он создаётся в нераспределённой памяти компьютера. Так как память ещё нераспределена, её нужно сначала подготовить. После того как объект уже не нужен, эту память нужно освободить, вызвав деструктор объекта. Простые переменные освобождать не надо, потому что стек уничтожается автоматически.

Объекты – очень удобная вещь. Он работает как шаблон, на основе которого создаются переменные типа объектов. Например:

Начало программы.

Переменные:
Объект1 – Кнопка;
Объект2 – Кнопка;

Начало кода
Объект1:= Кнопка.Создать объект
Объект2:= Кнопка.Создать объект

Объект1.Заголовок:='Привет'
Объект2.Заголовок:='Пока'

Объект1.Уничтожить объект.
Объект2.Уничтожить объект.
Конец кода

Инициализация объекта очень часто ещё называется созданием экземпляра объекта. И это тоже правильно. Когда ты вызываешь конструктор, в памяти создаётся новый экземпляр объекта. Можно сказать, что наши переменные «Объект1» и «Объект2» указывают на собственные экземпляры объекта «Кнопка», т.е. мы получаем две независимые копии объекта (кнопки) в памяти. Любую из них можно независимо менять и она не будет влиять на другую переменную.

В этом примере я объявляю две переменных типа *Кнопка*. Потом инициализирую их и меняю заголовок на нужный. Таким образом, я получил из одного объекта две кнопки с разными заголовками. Обе кнопки работают автономно и не мешают друг другу, потому что им выделена разная память. Таким образом, мы просто создаём новые объекты на основе шаблона объекта *Кнопка*, и потом используем их раздельно, меняя разные свойства шаблона и используя его методы.

Для уничтожения объекта, всегда есть метод Free. Так что если объект тебе больше не нужен и ты хочешь его уничтожить, то просто вызови этот метод:

Объект1.Free.

Я, конечно же, забегаю вперёд, объясняя тебе метод **Free**. Но всё же тебе не помешало бы уже познакомиться с ним. Пора приводить реальные строчки кода, и потихонечку вникать в смысл программирования.

4.3 Компонентная модель.

Компоненты - это более совершенные объекты. Грубо говоря, компоненты – это объекты, с которыми можно работать визуально.

Когда создавалась технология объектно-ориентированного программирования (ООП), о визуальности ещё никто не думал, и она существовала только в мечтах программистов. А когда фирма Borland создавала свою первую визуальную оболочку для Windows, пришлось немного доработать концепцию ООП, чтобы с объектами можно было работать визуально.

До появления шестой версии, в Delphi существовала только одна компонентная модель – VCL (Visual Component Library – визуальная библиотека компонентов). В шестой версии появилась новая библиотека CLX (Borland Component Library for Cross Platform – кросс платформенная библиотека компонентов).

VCL – библиотека компонентов разработанная только под Windows. Она очень хорошая и универсальная, но работает только под Windows.

В 2000 году фирма Borland решила создать визуальную среду разработки для Linux. В основу этой среды разработки легла Delphi и VCL. Но просто создать новую среду разработки было слишком просто и не эффективно. Было принято решение сделать новую библиотеку компонентов, с помощью которой можно было бы писать код как под Windows, так и под Linux. Это значит, что код, написанный в Delphi под Windows должен без проблем компилироваться под Linux без дополнительных изменений.

Так в 2001 году появилась новая среда разработки Kylix, которая смогла компилировать исходные тексты, написанные на Delphi для работы в операционной системе Linux. В качестве компонентной модели использовалась новая библиотека CLX. В принципе, это та же самая VCL с небольшими доработками. Даже имена объектов остались те же.

4.4 Наследственность.

Объекты обладают достаточно большим количеством преимуществ. Наследственность – одно из них. В некоторых книгах это свойство объектов начинают описывать только к середине книги, но это громадная ошибка. Наследственность объектов – это основа ООП. Даже в самой простой программе мы встречаемся с наследственностью, поэтому нам просто необходимо разобраться с этим уже сейчас.

Одно из величайших достижений в ООП - наследование. Рассмотрим пример. Вы написали объект - "гараж". Теперь вам нужно написать объект "дом". Гараж – это, грубо говоря, однокомнатный дом. Оба эти здания обладают одинаковыми свойствами - стены, пол, потолок и т.д. Поэтому желательно взять за основу гараж и доделать его до дома. Для этого ты создаёшь новый объект "Дом" и пишешь, что он происходит от "Гаража". Твой новый объект сразу примет все свойства гаража. Это значит, что у тебя уже будут стены, пол и потолок, и остается добавить только интерьер. Теперь у тебя будет уже два объекта: гараж и дом. Можно ещё создать будку для собаки. Для этого снова создаём объект "Будка" который происходит от "Гаража" (можешь произвести от "дома", чтобы в будке у

собаки был интерьер :)). Нужно только уменьшить размер гаража и он превратится в будку. В итоге у тебя получается древовидная иерархия наследования свойств. На рис. 4.4.1 я показал примерную иерархию наших объектов.

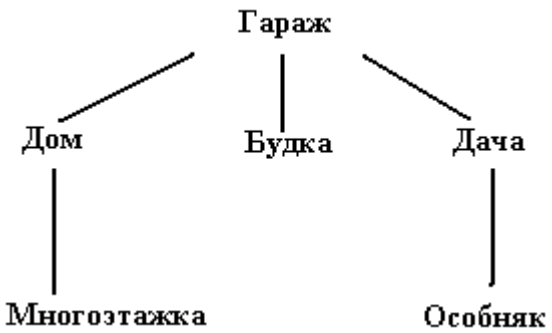


Рис 4.4.1 Иерархия гаража.

Тут тебе ещё нужно запомнить два понятия: *предок* и *потомок*. *Предок* – объект, от которого происходит какой-нибудь другой объект. *Потомок* – объект, который происходит из другого. Например, *гараж* – это предок для *дома*, *будки* и *дачи*. *Дом*, *будка* и *дача* – потомки от *гаража*. Один объект может быть и потомком и предком одновременно. Например, объект *дом* является потомком от *гаража* и является предком для *многоэтажки*.

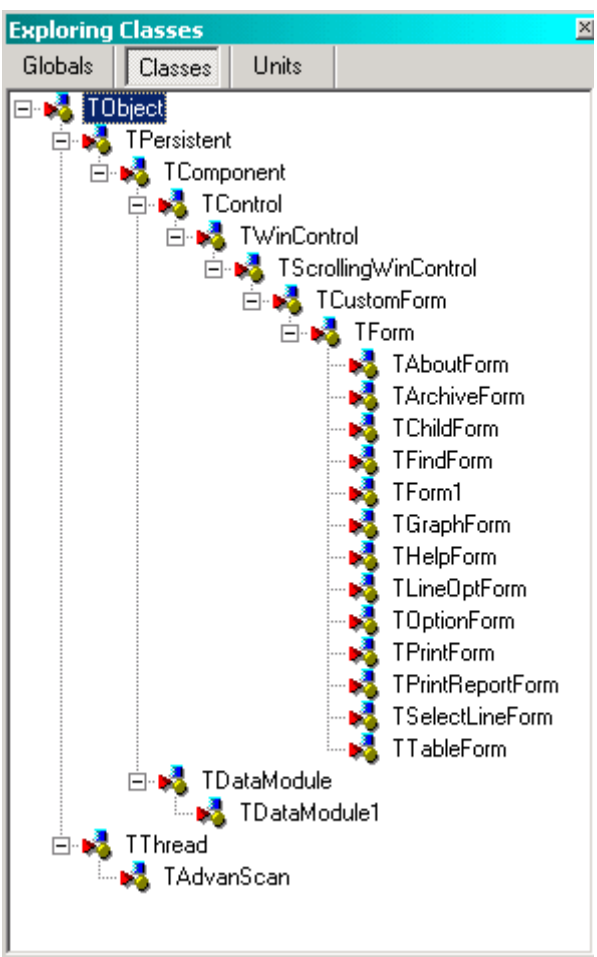


Рис 4.4.2 Иерархия ООП в Delphi.

Точно такой же метод наследования принят и в объектно-ориентированных языках. На рисунке 4.4.2. Я показал небольшую иерархию, взятую из Delphi. Как видно из рисунка, все объекты происходят от TObject. Это базовый объект, который реализует основные свойства и методы. От него происходит TPersistent и TThread. Это только на моём рисунке. В реальности, от TObject происходит намного больше объектов, и все они знают о его свойствах и методах и наследуют их себе.

Есть ещё один прикол, который очень удобен в ООП - полиморфизм. Что это за утка? Представим, что у гаража дверь открывается вверх, а у дома должны открываться в сторону. Дом происходит от гаража, поэтому у него дверь будет открываться тоже вверх. Как же тогда быть? Ты просто должен переписать у дома процедуру, отвечающую за открытие двери. Тогда твой дом получит все свойства гаража, но за открывание двери подставит свою процедуру. Что-то подобное и есть полиморфизм, когда объекты разных иерархий по-разному реагируют на одно и тоже событие.

Для того чтобы можно было изменить процедуру, отвечающую за открывание двери, она должна быть объявлена у гаража, как «виртуальная» (virtual). Виртуальная процедура говорит о том, что в порождённом объекте она может быть заменена.

И это ещё не всё. В гараже у нас стены голые, а в доме мы хотим повесить на них картины. Для реализации этого в ООП есть оболденная штука, как вызов метода предка. Рассмотрим пример:

```
Процедура отвечающая за создание стен у гаража.
  Начало
    Создать стены
  Конец

Процедура отвечающая за создание стен у дома.
  Начало
    Вызвать объект предка.
    Повесить на стены картины.
  Конец
```

В процедуре отвечающей за создание стен у гаража мы создаём стены. У дома тоже есть такая процедура, т.е. мы её переопределяем. На первый взгляд процедура гаража должна пропасть и у дома придётся снова создавать стены, но это не так. Нужно просто вызвать объект предка (тогда создадутся стены), а потом спокойно вешать на стены картины.

Глава 5. Основы языка программирования Delphi.

Мы уже достаточно нахватались теории, и пора приступить к изучению самого языка программирования Delphi. До этой главы мы писали на каком-то абстрактном языке программирования, но сейчас пришла пора познакомится с Delphi. Теперь мы будем писать только на нём, и превратим наши теоретические знания в практику.

В этой главе ты познакомишься с основами программирования на Delphi. Мы научимся пользоваться компонентами, их свойствами и методами. А так же увидим на практике объектную модель Delphi.

В течение всей книги я буду использовать два термина: объектная модель и компонентная модель. Под обоими терминами я буду понимать одно и тоже. Как я уже сказал, компоненты отличаются от объектов только возможностью работы с ними визуально. А в остальном оба термина идентичны.

В этой главе мы создадим нашу первую программу, и внимательно рассмотрим, из чего она состоит. Я даже не буду повторять, что эта часть книги так же описывает основы программирования и закладывает фундамент наших знаний. Её понимание так же важно, как и понимание предыдущих частей книги.

5.1 «Hello World» или из чего состоит проект.

В большинстве книг по программированию (особенно C++), описание начинают с программы “Hello world”. Это самая простая программа, которая выводит на экран окно, в заголовке которого написано эти два заветных слова “Hello world”. В своё время появилось даже несколько анекдотов по этому поводу.

Авторы книг по Delphi упускают этот пример, считая его слишком простым. Они сразу начинают описание компонентов и работу с ними. Это ошибка. Действительно, написать программу подобную “Hello world” очень просто с точки зрения программирования. Для этого не надо писать ни одной строчки. Зато на таком примере очень удобно рассказывать про принцип программирования на Delphi и структуру проекта.

Итак, давай напишем эту программу и разберём, что делает Delphi по полочкам. Запусти оболочку Delphi. Перед тобой откроется окно разработки, которое мы разобрали в третьей главе. Оболочка Delphi загружается с уже созданным пустым проектом. Окно дизайнера с заголовком Form1 как раз сигнализирует об этом.

Давай закроем этот проект и создадим новый. Выбери из меню *File* пункт *Close All*. Перед тобой останется только главное окно и «Объектный инспектор», в котором ничего недоступно.

Теперь открой «Менеджер проектов», для этого из меню *View* выбери пункт *Project Manager*. Перед тобой откроется окно, как на рисунке 5.1.1. Как видишь, окно абсолютно пустое, в нём только горит надпись <No Project Group>.

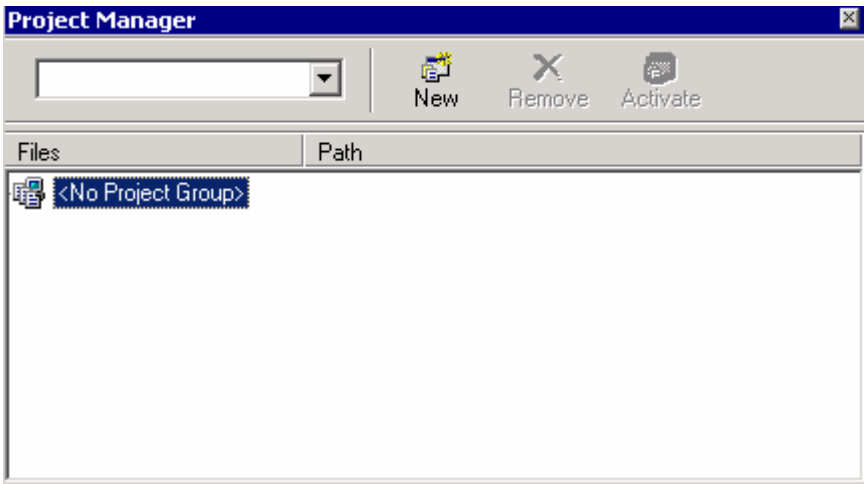


Рис. 5.1.1 Менеджер проектов.

Вот теперь создадим новый проект и посмотрим, что произойдёт. Это можно сделать тремя способами:

1. Выбрать из меню *File* пункт *New* и затем *Application.*
2. Выбрать из меню *File* пункт *New* и затем *Other*. Перед тобой откроется окно, как на рисунке 5.1.2. В этом окне нужно выбрать пункт Application и нажать OK.
3. В окне менеджера проекта нажать кнопку *New* и сразу откроется окно, как на рисунке 5.1.2. Дальше понятно.

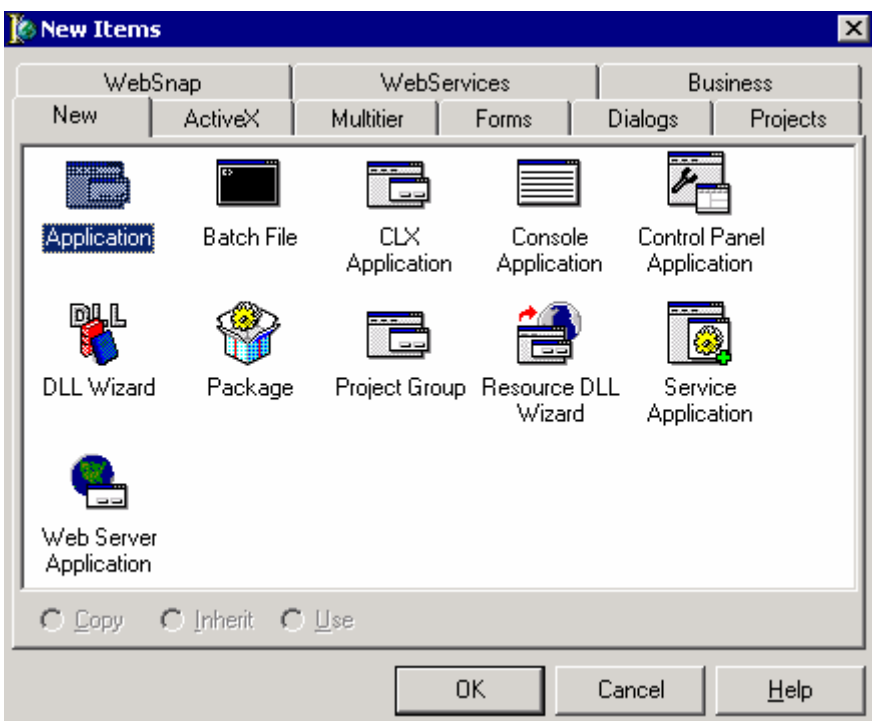


Рис. 5.1.2 Создание нового проекта.

Если действительно ты хочешь научиться программированию и использованию оболочки Delphi, то читай эту книгу и одновременно пробуй всё, что я говорю. Только так это отложится в памяти.

Попробуй закрыть проект (выбери из меню *File* пункт *Close All*), потом создать его одним способом. Потом снова закрыть и создать другим способом. Результат будет один и тот же. Но лучше использовать один из первых двух способов. Третий немного отличается и его отличие я покажу немного ниже.

Давай теперь посмотрим на менеджер проектов. Он изменился и достаточно сильно (посмотри на рисунок 5.1.3).

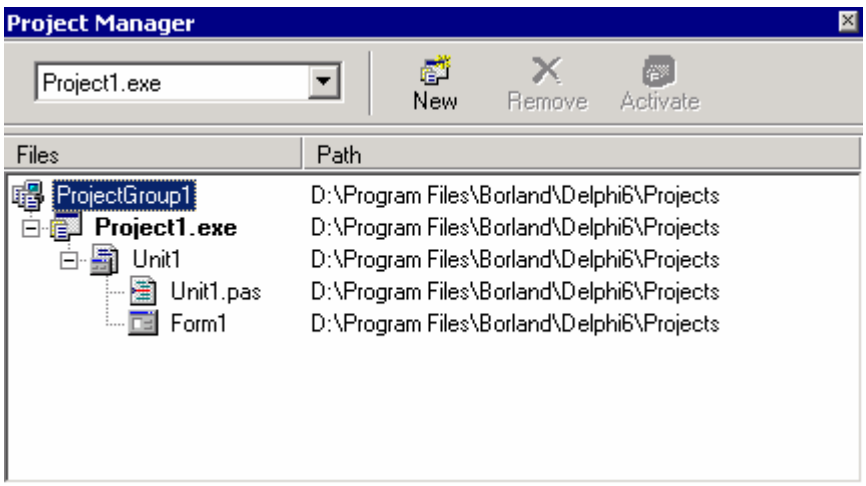


Рис. 5.1.3 Менеджер проекта при созданном проекте.

В менеджере проектов появилось целое дерево. Давай рассмотрим каждый пункт этого дерева:

- ProjectGroup1 (заголовок дерева) – имя группы проектов. В одной группе проектов может быть несколько приложений. В нашем случае мы создали одно новое приложение, поэтому в группе будет только оно. Если ты нажмёшь кнопку *New* в окне менеджера проектов и создашь новое приложение, то оно будет добавлено в существующую группу проектов.
  - Project1.exe – имя проекта (приложения). Когда ты создаёшь новое приложение, Delphi даёт ему имя Project плюс порядковый номер.
    - Unit1 – модуль. Проект состоит из модулей. Каждое окно программы – это отдельный модуль. Модуль состоит из двух файлов:
      - Unit1.pas – С расширением *.pas* показываются файлы, содержащие исходный код модуля. Имя файла такое же, как и у модуля.
      - Form1 – это визуальная форма. Она сохраняется в файле с таким же именем, как у модуля, но с расширением *.dfm*.

Давай сразу сохраним наше новое приложение. Для этого выбери из меню *File* пункт *Save All*. Сначала Delphi запросит ввести имя модуля (рисунок 5.1.4). По умолчанию, уже указано текущее имя Unit1.pas. Давай введём «*MainModule*». Заметь, что нельзя вводить имена с пробелами или на русском языке. Если ты попытаешься ввести что-то подобное, то произойдёт ошибка. Не забудь выбрать директорию, куда хочешь сохранить модуль и проекты. Желательно, чтобы всё хранилось в одной директории. Нажми кнопку «Сохранить».

Теперь Delphi запросит у тебя имя будущего проекта. Давай введём «*HelloWorld*». Заметь, что нельзя вводить имена с пробелами или на русском языке. Нажми кнопку

«Сохранить». Проект сохранится под именем «*HelloWorld.dpr*». Когда ты захочешь снова открыть пример, то тебе нужно открыть именно этот файл. Не надо открывать файлы с расширением *.pas*, потому что это всего лишь составляющая часть проекта и поэтому ничего тебе не даст. Нужно открывать файлы с расширением *.dpr*.

Старайся выбирать имена, наиболее отображающие содержимое модуля, чтобы потом легче было разобраться с файлами в больших проектах. К тому же желательно помнить, что имя проекта задаёт имя будущего запускного файла. Если ты оставишь имя как Project1, то и запускной файл будет называться Project1.exe.

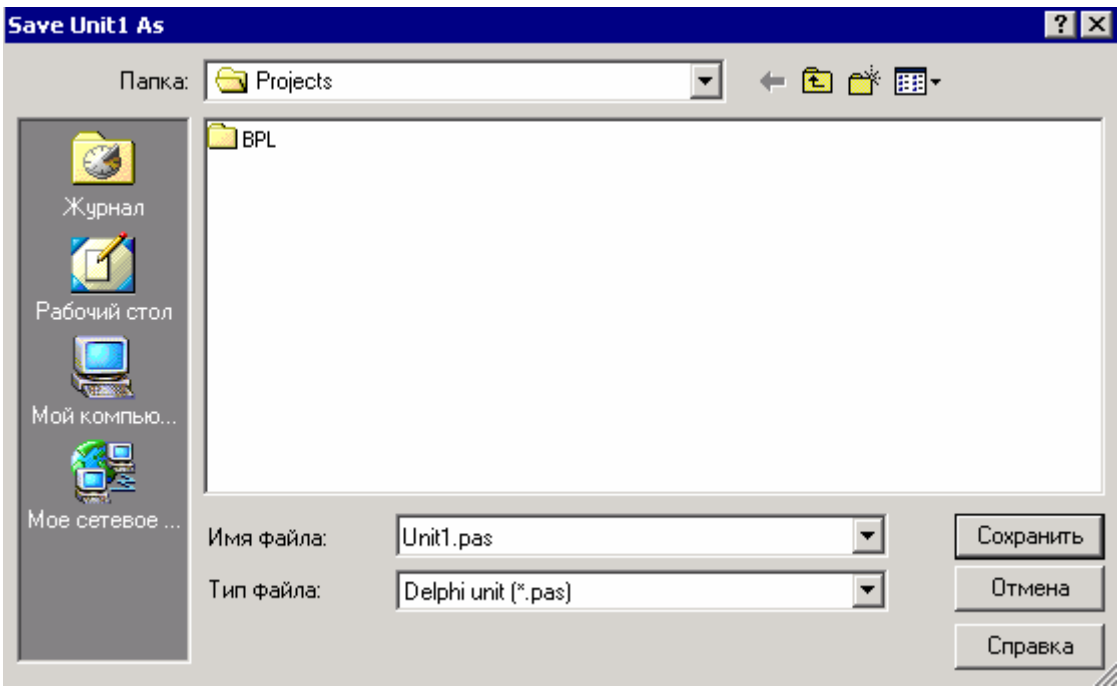


Рис. 5.1.4 Запрос ввода имени модуля.

Давай теперь посмотрим, как изменился наш менеджер проектов. Как видишь, имя проекта изменилось на *HelloWorld*, а имя модуля на *MainModule*.

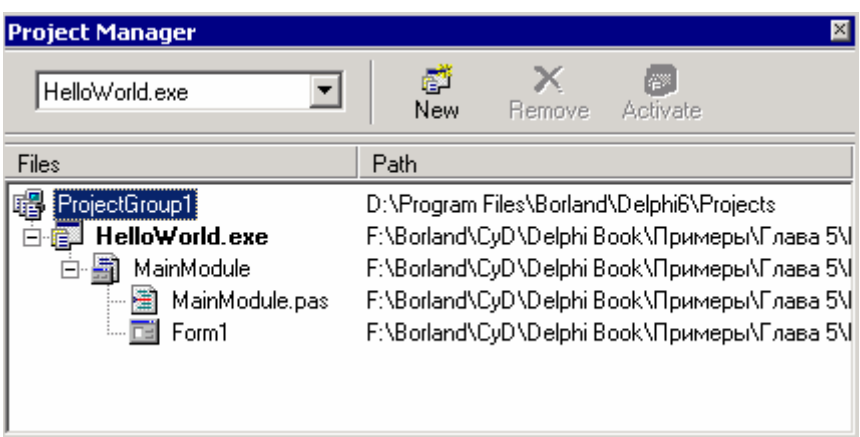


Рис. 5.1.5 Менеджер проекта.

Теперь перейди в директорию, куда ты сохранил проект и посмотри, какие файлы там присутствуют. На моём компакт диске это директория \Примеры\Глава 5\Hello World. Давай посмотрим на содержимое этих файлов:

- HelloWorld.cfg – файлы с расширением .cfg содержат конфигурацию проекта.
- HelloWorld.dof - файлы с расширением .dof содержат опции проекта.
- HelloWorld.dpr – файлы с расширением .dpr это сам проект. В этом файле находится описание используемых в проекте модулей и описание инициализации программы. Этот файл можно использовать и для написания кода. В будущем, мы научимся писать код и в этом модуле.
- HelloWorld.res - файлы с расширением .res содержат ресурсы проекта, такие как иконки, курсоры и др.
- MainModule.pas - файлы с расширением .pas содержат исходный код модулей.
- MainModule.dfm - файлы с расширением .dfm содержат визуальную информацию о форме.
- MainModule.ddp – файлы с расширением .ddp - вспомогательный файл модуля
- MainModule.dcu - файлы с расширением .dcu – откомпилированный модуль. Когда компилируется программа, все модули проекта собираются в один и получается запускной файл. У тебя пока что может не быть этого файла, потому что ты ещё не производил компиляции.

Немного позже мы узнаем и внутренности некоторых из этих файлов. Пока просто можешь просмотреть их.

А теперь давай вернёмся к нашей программе Hallo World, которую мы должны написать. Но сначала, давай посмотрим, что у нас уже есть. Для этого нужно скомпилировать наш проект (который пока что пустой и содержит только то, что создал Delphi), так что выбери из меню *Project* пункт *Compile* HelloWorld. Если ты выбирал в настройках Delphi показывать окно состояния компиляции, то ты увидишь вот такое окно:

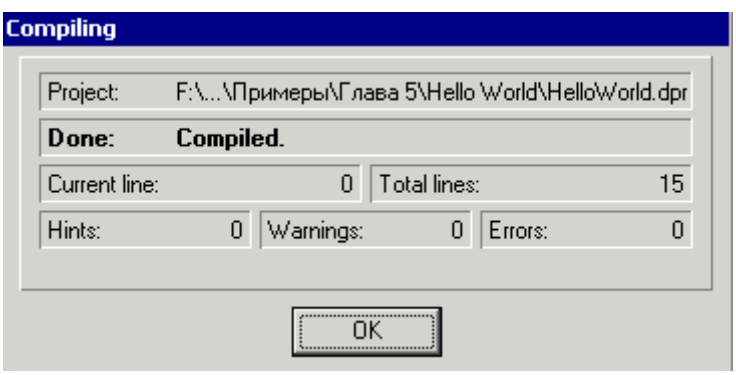


Рис. 5.1.6 Результат компиляции.

Как видишь, нет никаких сообщений, предупреждений или ошибок. Ещё бы, ведь мы ещё ничего не делали :). Программа скомпилирована. Просто нажми «ОК» чтобы закрыть это окно.

Теперь перейди в директорию, где ты сохранил проект. Там появится запускной файл HelloWorld.exe. Запусти его и ты увидишь окно, как на рисунке 5.1.7.

Как видишь перед нами пустое окно, т.е. программа почти готова, хотя мы ещё вообще ничего не делали. Нам нужно только изменить заголовок окна на *«Hello World»* и всё.

На языке C++ такая простая программа описывалась бы с такими подробностями в течении 20-30 страниц. И это не шутка.

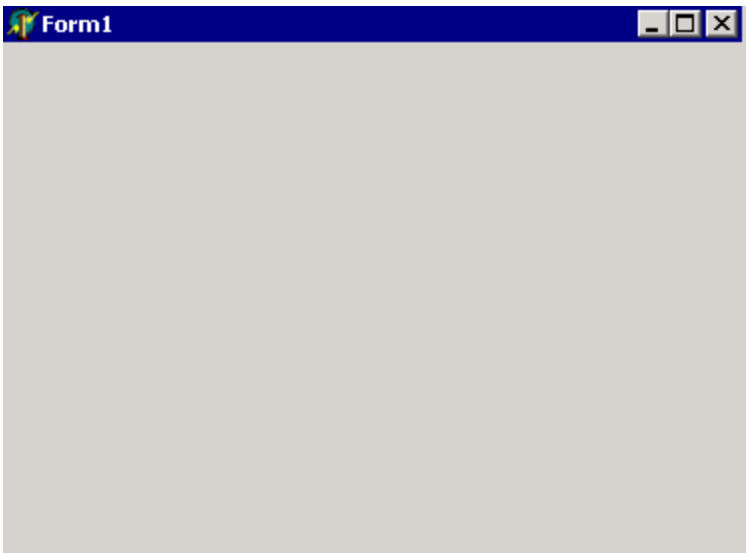


Рис. 5.1.7 Программа в действии.

Итак, нам осталось изменить заголовок. Для этого вспоминаем ООП. В Delphi всё объекты, значит и окно программы тоже объекты. Заголовок окна – это скорей всего свойство окна. Для того, чтобы изменить это свойство, нужно перейти в объектный инспектор, найти так свойство Caption (Заголовок) и ввести в него *Hello World* (рисунок 5.1.8). Ввёл, нажал клавишу Enter и программа готова.

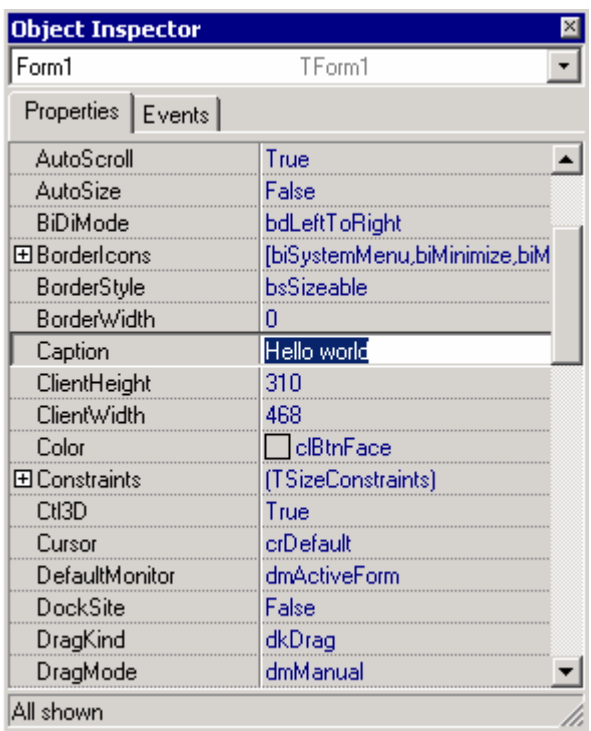


Рис. 5.1.8 Объектный инспектор.

Теперь давай запустим нашу программу. Для этого можно снова скомпилировать её и запустить файл. Но на этот раз мы пойдём по-другому. Выбери из меню *Run* пункт *Run*

(или нажми клавишу F9). Delphi сразу откомпилирует и запустит готовую программу. Результат можно увидеть на рисунке 5.1.9

Как видишь, программирование не настолько страшно, как его расписывают.

Рис. 5.1.9 Hello World в действии.

Заведи себе в привычку перед каждой компиляцией сохранять весь проект (File-> Save All). Обидно будет, если пол дня тяжёлой работы пропадёт даром от какой-нибудь внештатной ситуации. А они бывают, если ты понаставишь в Delphi компонентов сторонних разработчиков с корявыми руками. Я уже много раз встречал коряво написанные компоненты, которые вышибали Delphi в даун. Поэтому лучше лишний раз сохранится.

В принципе, на этом можно было бы остановиться и рассмотреть код модуля главной формы, но я хочу перед этим показать тебе свойства проекта и как ими управлять. Выбери из меню *Project* пункт *Options* и ты увидишь окно, как на рисунке 5.1.10. Окно разбито на множество закладок и с некоторыми из них мы познакомимся сейчас, а остальные оставим на будущее. Слишком много информации ни к чему хорошему не приведёт, тем более, что нам пока большинство настроек ненужно.

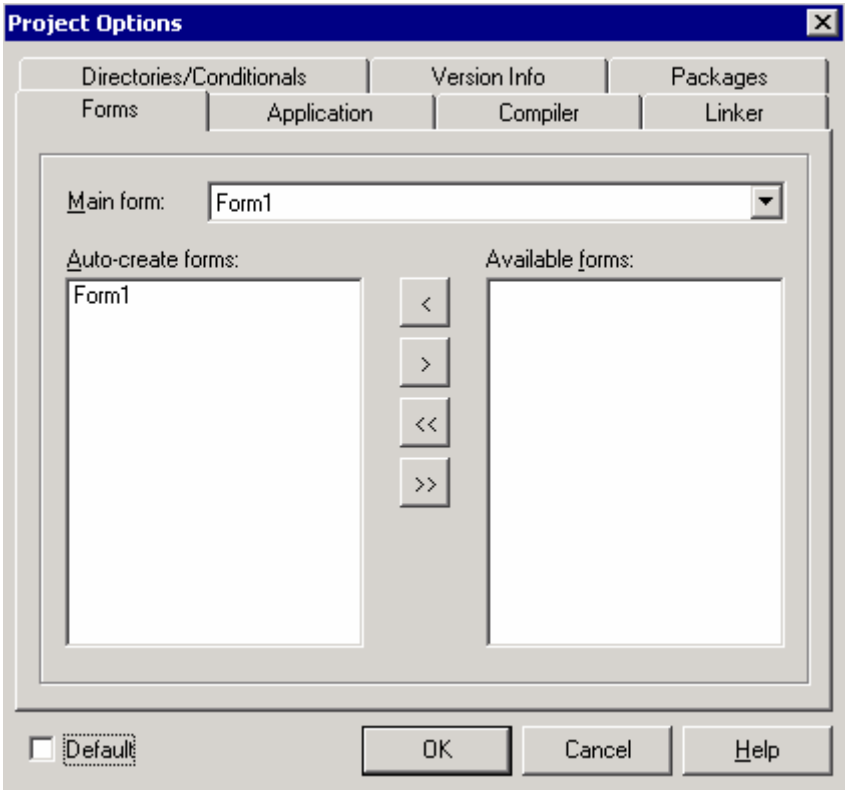


Рис. 5.1.10 Свойства проекта

На первой закладке *Forms* ты можешь настраивать формы проекта. Вверху ты видишь выпадающий список *Main form*. Здесь можно выбрать форму, которая будет являться главной для приложения. У нас только одна форма *Form1* и она будет главной.

Чуть ниже расположены два списка. Слева находиться список *Auto-create forms* – автоматически создаваемые формы. При запуске программы, все описанные здесь формы будут инициализироваться автоматически. Справа находиться список *Available forms* – доступные формы. В этом списке будут находиться формы, которые не будут создаваться автоматически. Такие формы мы обязаны инициализировать самостоятельно. Между списками расположены кнопки, с помощью которых ты можешь перемещать формы между списками.

Теперь перейди на закладку *Applications*. Здесь ты можешь настраивать следующие поля:

Title – заголовок, который будет отображаться в панели задач.

Help file – имя файла помощи.

Icon – иконка приложения. По умолчанию используется иконка Delphi, но ты можешь её сменить, нажав на кнопку *Load Icon*.

Target file extension – расширение результирующего файла. Если здесь ничего не указано, то запускной файл будет иметь расширение *exe*. Ты можешь указать любое другое значение, но файл от этого не измениться. Это будет тот же самый запускной файл, только ему присвоиться другое расширение.

Следующая закладка – *Version Info*. Если поставить галочку в *Include version information in project*, то в запускной файл будет встроена информация о версии программы. На этой же закладке ты можешь указать номер версии, сборки и выпуска (рисунок 5.1.11).

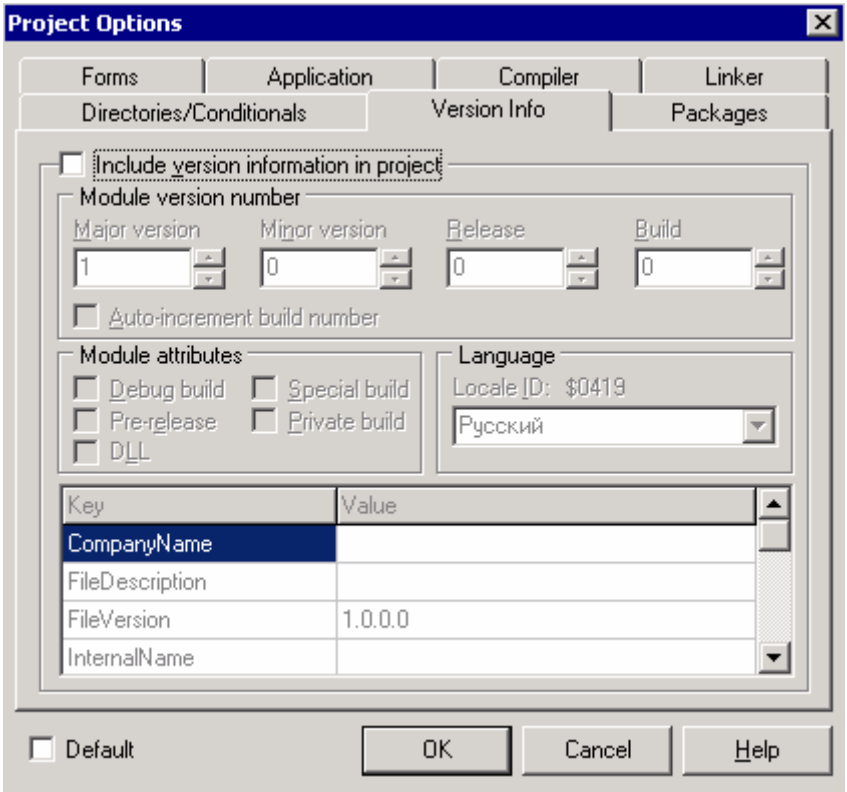


Рис. 5.1.11 Свойства проекта

На компакт диске, в директории ***\Примеры\Глава 5\Hello World*** *ты можешь увидеть пример этой программы.*

5.2 Язык программирования Delphi.

Язык программирования Delphi достаточно прост в обучении, но очень эффективен и достаточно мощный. Самое первое, с чем тебе надо познакомится – это комментарии.

Комментарии - это любой тест который абсолютно не влияет на код программы. Он никогда не компилируется и не вставляется в .exe файл, а используется только для пояснений кода. Комментарии могут оформляться двумя способами:

1. Всё, что идёт после двойного слеша воспринимается комментарий. Так можно оформить только одну строку.
2. Всё, что заключено в фигурные кавычки { и } тоже комментарий. Так можно заключить в комментарий сколько угодно строк.

Рассмотрим пример:

```
//Это комментарий.
Это уже не комментарий

{Это снова комментарий
И это тоже}
```

Я буду постоянно использовать комментарии, чтобы пояснять код программ, которые будут описываться в книге.

Вот теперь ты готов к изучению языка программирования Delphi. Создай новый проект или открой программу HalloWorld, разницы нет. Теперь перейди в редактор кода, который показан на рисунке 5.2.1.

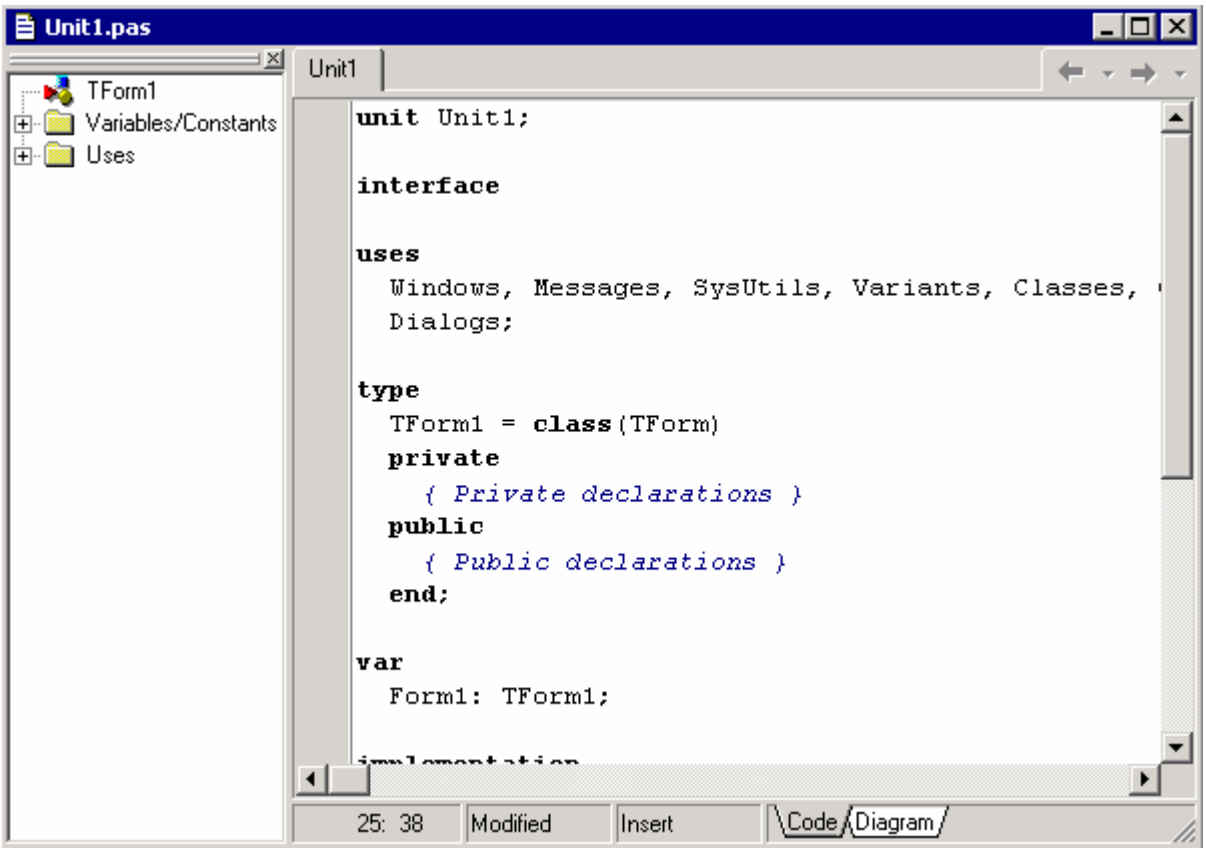


Рис. 5.2.1 Hello World в действии.

Здесь для тебя уже написана заготовка будущей программы. Точнее сказать только для этой формы. Если ты захочешь, чтобы в твоей программе было два окна, то тебе придётся создать ещё одну форму, а значит, появится ещё один подобный модуль.

Давай досконально рассмотрим, что тут написано. Я специально выделил комментарии другим шрифтом, и синим цветом, чтобы они выделялись от кода программы.

```
unit Unit1; //Имя модуля

interface

uses //После этого слова идёт перечисления подключённых модулей.
  Windows, Messages, SysUtils, Variants, Classes,
  Graphics, Controls, Forms,  Dialogs;

Type //После этого идёт объявление типов
  TForm1 = class(TForm) //Начало описания нового объекта TForm1

//Здесь описываются компоненты и события

  private //После этого слова можно описывать закрытые данные объекта
    { Private declarations }//Подсказка, которую сгенерировал Delphi

    {Здесь можно описывать переменные и методы, доступные только для объекта TForm1}
```

```
  public //После этого слова можно описывать открытые данные объекта
    { Public declarations }//Подсказка, которую сгенерировал Delphi

    {Здесь можно описывать переменные и методы доступные из любого другого модуля}
  end;

var //Объявление глобальных переменных
  Form1: TForm1; //Это описана переменная Form1 типа объекта TForm1

implementation

{$R *.dfm} //Подключение .dfm файла (файл с данными о визуальных объектах)

end. // end с точкой - конец модуля
```

Возможно, не всё ещё понятно, но сейчас я попробуй объяснить всё более подробно. Но даже после этого некоторые вещи останутся непонятными, да и не всё отложится в твоей памяти, потому что будет слишком много информации. Когда мы начнём писать программы, то всё встанет на свои места.

Практически все строчки заканчиваются знаком ";" (точка с запятой). Этот знак показывает конец оператора. Он ставится только после операторов и никогда не используется после ключевых слов типа uses, type, begin, implementation, private, public и т.д. В последствии ты познакомишься со всеми ключевыми словами, большинство из них выделяется жирным шрифтом. Сразу видно исключение – end, после которого точка с запятой ставится.

В самом начале у нас стоит имя модуля. Оно может быть любым, но таким же, как и имя файла без расширения, так что изменять вручную его не желательно. Если уж сильно хочется изменить, то просто сохрани модуль с новым именем (выбери *File->Save As*)

Желательно давать модулям понятные имена, чтобы ты мог по имени определить, что находится внутри.

Далее идёт подключение глобальных модулей. Все процедуры, функции, константы описаны в каком-нибудь модуле, и прежде чем эти процедуры использовать, нужно подключить этот модуль. Ты можешь знать о существовании какой-нибудь функции. Но чтобы об этом узнал компилятор, ты должен указать модуль, где описана эта функция понятным для компилятора языком. Например, тебе надо превратить число в строку. Для этого в Delphi уже есть функция IntToStr. Она описана в модуле SysUtils. Если ты хочешь воспользоваться этой функцией, то тебе нужно только подключить этот модуль к своему модулю и использовать уже готовую функцию. В этом случае тебе не надо писать собственный вариант преобразования чисел в строку. Только подключи и используй.

Самое сложное - разобраться с объявлениями типов. Весь код, который ты пишешь, должен относится к какому-нибудь типу. Их мы описываем после ключевого слова type. Строка **TForm1 = class(TForm)** говорит о том, что мы создаём новый объект *TForm1*, который будет происходить от объекта *TForm*. А это значит, что TForm1 будет обладать всеми возможностями *TForm* , и плюс то, что мы захотим.

TForm - это стандартный объект-форма, который входит в библиотеку VCL и CLX. Все окна в Delphi относятся к этому объекту.

Вспоминаешь, что я говорил об ООП. Вот оно наследование. Чтобы объявить какой-то объект, ты должен в разделе **type** написать:

```
Имя объекта = class
  //Свойства, методы и события объекта
end;
```

Кстати, в старом Паскале использовалось понятие «объект». В Delphi принято называть объекты *классами*, как это делается в C++. Разницы в этих понятиях я не вижу, хотя некоторые пытаются вложить в эти понятия разный смысл, но я этого делать не буду. Понятия *класс* и *объект* в моей книге будут означать одно и то же.

Если ты хочешь, чтобы твой объект наследовал свойства другого объекта, то ты должен указать после ключевого слова *class* имя объекта, который будет являться родителем для твоего объекта.

```
Имя объекта = class(Имя предка)
  //Свойства, методы и события объекта
end;
```

Вот так и получается, что запись **TForm1 = class(TForm)** создаёт новый объект **TForm1**, который является потомком объекта **TForm**.

Вернёмся к нашему коду. После объявления объекта идут объявления его свойств, методов и событий, которые заканчиваются ключевым словом **end;**. Объявления делятся на три части: *private*, *protected* и *public*. Хотя у тебя по умолчанию Delphi не создаёт раздел *protected*, ты можешь написать его сам.

До начала описания разделов идёт описание компонентов входящих в состав объекта и событий объекта. Тут ты не можешь ничего писать вручную, это создаётся самим Delphi. Зато внутри разделов ты можешь описывать любые переменные, процедуры и функции.

```
Имя объекта = class(Имя предка)

  //Здесь описываются компоненты и события

  private //После этого слова можно описывать закрытые данные объекта

    {Здесь можно описывать переменные и методы, доступные только для объекта TForm1}

    Переменная1: Integer;
    Procedure Proc1;

  public //После этого слова можно описывать открытые данные объекта

    {Здесь можно описывать переменные и методы доступные из любого другого модуля}
    Переменная2: Integer;
    Переменная3: String;
    Procedure Proc2;
    Procedure Proc3;
end;
```

При описании действует одно правило: внутри раздела, сначала описываются переменные, а потом процедуры и функции. Нельзя описывать сначала процедуры, а потом переменные или делать это вразноброс. Рассмотрим раздел *private*.

```
private
  Переменная1: Integer;
  Procedure Proc1;// Это процедура, после неё не может быть переменных
  Переменная2: String;//Это ошибка. Уже объявлена одна процедура.
```

С типами разобрались, теперь перейдём к описанию глобальных переменных. Оно начинается после ключевого слова **var** и идёт всегда после объявления типов.

Глобальные переменные – переменные, которые хранятся в стеке, создаются при запуске программы и уничтожаются при выходе из программы. Это значит, что они доступны всегда и везде, пока запущена твоя программа.

```
var //Объявление глобальных переменных
  Form1: TForm1; //Это описана переменная Form1 типа объекта TForm1
```

Ключевое слово **implementation** мы пока трогать не будем, а оставим на будущее, когда наши знания о Delphi улучшаться.

Последнее, что нам осталось рассмотреть в этой главе – ключ **{$R *.dfm}**. В фигурных скобках могут быть не только комментарии, но и ключи для компилятора. Они отличаются тем, что выглядят как **{$Буква Параметр}**. *Буква* – указывает на тип ключа. В нашем случае, мы используем букву R. Этот ключ указывает на то, что надо подключить **.dfm** файл (файл с данными о визуальных объектах).

Ключ R – это единственное, что нам пока надо знать. Со временем мы изучим ключи более подробно.

Любой код в Delphi заключается между **BEGIN** и **END. BEGIN** – означает начало кода, а **END –** конец. Например, когда ты пишешь процедуру, то сначала нужно написать её имя (как это делать мы поговорим позже), а потом заключить её код между **BEGIN** и **END**. Мы ежу говорили об этом, когда писали на абстрактном языке программирования, но я решил повториться, чтобы ты не забывал, что это относится к Delphi.

5.3 Типы данных в Delphi.

Как я уже говорил, на языке программирования всё должно иметь свой тип. Мы уже знаем, что существует четыре основных типа данных: целые числа, вещественные числа (дробные), строки и булевы.

Все переменные должны относится к какому-то типу. Почему? Я уже отвечал на этот вопрос и чтобы вспомнить это, нужно вспомнить, что такое переменная. *Переменная* – область памяти, способная хранить информацию. Чтобы знать, что хранится в этой памяти, мы должны указать, к какому типу относится переменная. Если переменная относится к типу целых чисел, то в памяти, отведённой под переменную, хранится целое число.

Рассмотрим, какие бывают типы переменных.

5.3.1 Целочисленные типы данных.

В переменных целого типа, информация представлена в виде чисел, не имеющих дробной части. Они используются для математических вычислений и любых других операциях, где нужна работа с числами.

Существует несколько видов целых типов данных. Они в основном отличаются размером отводимой памяти для хранения данных:

Тип	Диапазон возможных значений	Размер памяти для хранения данных	Примечание
Integer	–2147483648..2147483647	4 байта (32-бита)	Знаковое
Cardinal	0..4294967295	4 байта (32-бита)	Без знака
Shortint	–128..127	1 байт (8 бит)	Знаковое
Smallint	–32768..32767	2 байта (16 бит)	Знаковое
Longint	–2147483648..2147483647	4 байта (32-бита)	Знаковое
Int64	$-2^{63}..2^{63}-1$	8 байт (64 бита)	Знаковое
Byte	0..255	1 байт (8 бит)	Без знака
Word	0..65535	2 байта (16 бит)	Без знака
Longword	0..4294967295	4 байта (32-бита)	Без знака

В этой таблице я перечислил все типы целых чисел. В примечании указано, какого типа могут быть числа – со знаком или без (т.е. только положительные). В зависимости от объёма памяти отводимого под хранение данных, зависит максимальное число, которое можно записать в эту переменную.

Целочисленным переменным ты можешь присваивать как десятичные числа, так и шестнадцатеричные. Для этого перед шестнадцатеричным числом нужно поставить знак доллара - $. Сразу же смотрим на пример:

```
var
 i:Integer;
begin
 i:=11;
 I:=$a;
end;
```

В этом примере я сначала присваиваю в переменную I число 10, а потом $A – шестнадцатеричное A, что тоже равно десяти. Так что первая и вторая строка делают одно и то же и результат будет одинаковый.

5.3.2. Вещественные типы данных.

Вещественные или дробные типы данных предназначены для хранения чисел с плавающей точкой. Некоторые считают, что лучше использовать именно такие типы данных, вместо целочисленных. Это заблуждение. Операции с плавающей точкой отнимают у процессора больше времени и требуют больше памяти. Поэтому используй переменные этого типа только там, где это действительно необходимо.

Тип	Диапазон возможных значений	Максимально количество цифр в числе	Размер в байтах
Real48	2.9×10^{-39} .. 1.7×10^{38}	11–12	6
Real	5.0×10^{-324} .. 1.7×10^{308}	15-16	8
Single	1.5×10^{-45} .. 3.4×10^{38}	7-8	4
Double	5.0×10^{-324} .. 1.7×10^{308}	15-16	8
Extended	3.6×10^{-4951} .. 1.1×10^{4932}	19-20	10
Comp	$-2^{63}+1$ .. 2^{63}	19-20	8
Currency	–922337203685477.5808.. 922337203685477.5807	19-20	8

Заметь, что все эти типы знаковые и могут содержать не только положительные, но и отрицательные значения.

Очень важно помнить, что вещественные числа не равны целым. Например, вещественное число 3.0 не будет равно целому числу 3. Для того, чтобы сравнить оба этих числа, нужно округлить вещественное число.

5.3.3 Символьные типы данных.

Символьные данные могут хранить текст, например, для вывода на экран или в окно диалога. Символьные данные – это простая цепочка из чисел. Каждое число – это порядковый номер символа в таблице символов. Например, если представить наш алфавит в виде таблицы символов (только представить), то число 0 будет означать букву *А*, число 1 будет означать букву *Б*, и так далее. Это значит, что слово кот в числовом виде будет выглядеть так:

---

10 14 18

Здесь 10 – это буква *К*, 14 – это *О*, 18 – это *Т*. Именно в виде таких последовательностей чисел и выглядят строки в компьютерной памяти.

Самые первые таблицы символов были 7 – битными (ASCII). А так как в 7 битов можно засунуть максимум число 127, то и количество символов в таблице равнялось 127. Хотя данные хранились в 7 битах, под каждый символ всё же отводились все 8, т.е. один байт. Это связано с тем, что память в компьютере разбита по ячейкам в 8 бит, а не в 7. Поэтому один бит оставался свободным.

Но тут возникает проблема, помимо букв в таблице должны содержатся ещё и цифры от 0 до 9, и служебные символы типа знака равно, больше, меньше и так далее. Таким образом, получилось, что в такой таблице не хватило места для букв из языков большинства национальностей.

В таблице ниже, ты можешь увидеть таблицу ANSI, которая используется в Windows:

Таблица ANSII

Симво Номе Симво Номе Симво Номе Симво Номе Симво Номе

л	р	л	р	л	р	л	р	л	р
	1	8	56	o	111	¦	166	Э	221
	2	9	57	p	112	§	167	Ю	222
	3 :		58	q	113	Ё	168	Я	223
	4 ;		59	r	114	©	169	а	224
	5 <		60	s	115	Є	170	б	225
	6 =		61	t	116	«	171	в	226
	7 >		62	u	117	¬	172	г	227
	8 ?		63	v	118		173	д	228
	9 @		64	w	119	®	174	е	229
	10 A		65	x	120	Ї	175	ж	230
	11 B		66	y	121	°	176	з	231
	12 C		67	z	122	±	177	и	232
	13 D		68	{	123	І	178	й	233
	14 E		69	|	124	і	179	к	234
	15 F		70	}	125	ґ	180	л	235
	16 G		71	~	126	µ	181	м	236
	17 H		72		127	¶	182	н	237
	18 I		73	Ђ	128	·	183	о	238
	19 J		74	Ѓ	129	ё	184	п	239
	20 K		75	,	130	№	185	р	240
	21 L		76	ѓ	131	є	186	с	241
	22 M		77	„	132	»	187	т	242
	23 N		78	…	133	ј	188	у	243
	24 O		79	†	134	S	189	ф	244
	25 P		80	‡	135	s	190	х	245
	26 Q		81	€	136	ї	191	ц	246
	27 R		82	‰	137	А	192	ч	247
	28 S		83	Љ	138	Б	193	ш	248
	29 T		84	‹	139	В	194	щ	249
-	30 U		85	Њ	140	Г	195	ъ	250
	31 V		86	Ќ	141	Д	196	ы	251
	32 W		87	Ћ	142	Е	197	ь	252
!	33 X		88	Џ	143	Ж	198	э	253
"	34 Y		89	ђ	144	З	199	ю	254
#	35 Z		90	‘	145	И	200	я	255
$	36 [		91	’	146	Й	201		
%	37 \		92	“	147	К	202		
&	38]		93	”	148	Л	203		
	39 ^		94	•	149	М	204		
(	40 _		95	–	150	Н	205		
)	41 `		96	—	151	О	206		
*	42 a		97		152	П	207		
+	43 b		98	™	153	Р	208		
,	44 c		99	љ	154	С	209		
-	45 d		##	›	155	Т	210		
.	46 e		##	њ	156	У	211		
/	47 f		##	ќ	157	Ф	212		
0	48 g		##	ћ	158	Х	213		
1	49 h		##	џ	159	Ц	214		
2	50 i		##		160	Ч	215		
3	51 j		##	Ў	161	Ш	216		
4	52 K		##	ў	162	Щ	217		
5	53 L		##	J	163	Ъ	218		
6	54 M		##	¤	164	Ы	219		
7	55 N		##	Ґ	165	Ь	220		

Если посмотреть на эту таблицу, то можно увидеть, что вместо английской буквы *А* в памяти будет стоять число 65, а вместо русской буквы *П* мы увидим 207. Получается, что слово *Привет* в памяти машины будет выглядеть так:

207 208 200 194 210

Нулевой символ в таблице использовать не стали и зарезервировали как полный ноль. Чуть ниже ты увидишь, как программисты нашли достойное применение этому символу. Первые символы в таблице (те, где в поле символ пусто) – это служебные символы, такие как символы клавиши ESC, TAB, Enter и др. Как ты понимаешь, у этих символов нет графического отображения, но у них должны быть номера.

В Delphi используется 8-битовая расширенная таблица символов, где задействованы все 8 битов (ANSI – таблица). Эта таблица берётся из самой операционной системы - Windows. Так что количество символов и их расположение зависит от ОС.

Для того, чтобы удовлетворить все национальности, уже давно ввели поддержку UNICODE (16-битная таблица). В ней первые 8 бит совпадают с таблицей ANSI, а остальные являются специфичными. Начиная с Windows 2000, эта кодировка начинает использоваться всё шире и шире.

В Delphi присутствуют следующие основные типы строк:

Тип	Максимальная длинна строки	Память отводимая для хранения строки	Примечание
ShortString	255 символов	От 2 до 256	
AnsiString	2^{31}	От 4 байтов до 2 Гбайт	8 битовые
WideString	2^{30}	От 4 байтов до 2 Гбайт	UNICODE

Строки в Delphi заключаются в одинарные кавычки. Например, ты можешь объявить переменную Str типа строки и присвоить ей значение ‘Hello World’ вот так:

```
VAR
  Str:AnsiString;
BEGIN
  Str:='Hello World'; //Присваиваем в Str значение 'Hello World'.
END;
```

Так как строки – это массив символов, то ты можешь получить доступ к отдельному символу. Для этого нужно после имени переменной указать в квадратных скобках номер символа, который тебе нужен, только не забывай, что буквы в строках нумеруются, начиная от 1. Большинство массивов в языках программирования нумеруются с нуля (строки – это те же массивы, только символов). Но тут получается исключение, которое надо запомнить.

Нулевой символ в строке указывает на длину строки. Его нельзя изменять прямым доступом, поэтому лучше вообще не обращаться напрямую к нулевому символу. Если что-то надо, то для этого есть специальные функции, которые мы рассмотрим немного позже.

А теперь посмотрим на примере, как можно работать с отдельными символами в строке:

```
VAR
  Str:AnsiString;
BEGIN
  Str:='Hello World'; //Присваиваем в Str значение 'Hello World'.
  Str[1]:='T';// В первый символ присваиваю значение 'T'.
END;
```

После присваивания в первый символ переменной Str буквы 'T', эта строка будет хранить строку *'Tello World'*.

Так как строка – это набор символов, а символ – это число указывающее на конкретный символ в таблице, мы можем создавать строки из шестнадцатеричных чисел. Например, если мы хотим присвоить в переменную Str строку *'Hello World'* плюс символы конца строки и перевода каретки (символа перехода на новую строку), то нужно воспользоваться шестнадцатеричными числами, потому что на клавиатуре нет этих символов и мы не можем их набрать. Конец строки в таблице символов находится на позиции #13, а перевода каретки #10. Таким образом код превращается в такой:

```
VAR
  Str:AnsiString;
BEGIN
  Str:='Hello World'#13#10; //Присваиваем Str значение 'Hello World'+#13#10.
  Str:=#100#123#89; //Присваиваю в Str строку из символов
                    //в шестнадцатеричном представлении.
END;
```

Очень часто в моих примерах можно встретить тип Char - это просто один символ. Мы редко будем использовать его в чистом виде, в основном он будет присутствовать в наших программах в виде массива символов.

На протяжении всей книги я буду чаще всего использовать тип String, потому что он очень удобен в использовании и практически ничем не отличается от описанных выше. Объявляется этот тип следующим образом:

```
var
 s:String;
 s1:String[200];
```

В этом примере я объявил две строковые переменные *s* и *s1*. Первая объявлена как простая строка *String*, а у второй, после типа в квадратных скобках стоит число 200. Что это за число? Это размер строки.

Когда переменная такого типа храниться в памяти, то её символы нумеруются с единицы. Если ты хочешь прочитать 2-й символ, то нужно обращаться к нему как *s[2]*. Это надо помнить, потому что уже говорил, что в большинстве случаев нумерация идёт с нуля. В данном случае, это можно назвать исключением.

Почему же эта строка нумеруется с единицы? Может нулевой символ просто не используется? Если взглянуть на строку данного типа в памяти машины, то можно

увидеть, что в нулевом символе храниться длинна строки. Получается, что если прочитать значение нулевого символа *s[0]*, то мы получим строку!!! Возможно так, но прямое обращение к нулевому символу не желательно, особенно не стоит его изменять. Если хочешь узнать длину строки, то используй функцию *Length*, а чтобы установить длину используй *SetLength*. Хотя с процедурами и функциями я буду знакомить тебя немного позже, здесь я покажу тебе маленький абстрактный пример:

```
var
 s:String;
begin
 s:='Привет!!!';
 Длинна:=Length(s);
 SetLength(s, 50);
end;
```

В этом примере, в первой строчке кода я присваиваю строковой переменной *s* первый пришедший в мою голову текст. Во второй строке я показываю, как можно узнать длину строки. В последней строчке, я вызываю процедуру *SetLength*, чтобы установить новую длину строки. Здесь, цифра 50 показывает значение новой длины строки.

5.3.4. Булевы типы.

С помощью переменных этого типа очень удобно строить логику. Переменная булева типа может принимать только одно из двух значений – TRUE или FALSE. Тебе это ничего не напоминает? Совсем недавно я рассказывал тебе про биты, которые имею два состояния 1 или 0, включён или выключен. Переменные булева типа занимают только один бит и принимают только эти два значения (0 или 1), но для удобства в программировании используют понятие TRUE (истина) или FALSE (ложь).

Я постараюсь использовать слово «Булев» пореже. Лучше и понятнее (на мой взгляд) называть их *логическими переменными*.

Для объявления логических переменных используется слово **Boolean**. Давай рассмотрим пример работы с такими типами:

```
VAR
  b:Boolean; // Объявляю логическую переменную b
  Str:AnsiString; // Объявляю строковую переменную Str
BEGIN
  b:= true;
  if b=true then
    Str:='Истина'
  else
    Str:='Ложь'
END;
```

В этом примере я объявил две переменные: b (логическая) и Str (строковая). Потом я присваиваю переменной b значение TRUE. Дальше действительно нужны хорошие пояснения, потому что идёт логическая конструкция if .. then, требующая пояснений.

Мы уже с тобой рисовали блок-схемы и в них использовали логику типа «если выполняется какое-то условие, то выполнить какое-то действие». Конструкция if … then

действует так же. Слово if переводится как «если». Слово then переводится как «то». Получается, что конструкция «если условие выполнено то …» выглядит на языке программирования как «if условие выполнено then …».

Частным случаем является конструкция «if … then … else». Слово «else» переводиться как «иначе». То есть если условие выполнено, то выполнится то, что написано после then, иначе выполнится то, что написано после else.

Я уже говорил, что все операторы в Delphi заканчиваются точкой с запятой. Это нужно чтобы отделять команды друг от друга, вед одна команда может быть записана на две строки или две команды в одной. Так вот, после оператора идущего перед ***else*** *никогда не ставится точка с запятой. Так, в примере выше не стоит точка с запятой после* **Str:='Истина'**, *потому что потом идёт* ***else***.

В примере я проверяю, если переменная b равна true, то переменной Str присвоить значение **'Истина'**, иначе присвоить значение **'Ложь'**.

В Delphi можно сравнивать булевы типы в упрощённом виде. Например, предыдущий код можно написать так:

```
VAR
  b:Boolean; // Объявляю логическую переменную b
  Str:AnsiString; // Объявляю строковую переменную Str
BEGIN
  b:= true;
  if b then
    Str:='Истина'
  else
    Str:='Ложь'
END;
```

В этом примере я просто написал «*if b then*». Если не указано, с чем мы сравниваем, то проверка происходит на правильное значение. Это значит, что переменная будет проверяться на истину (равна ли она true), т.е. этот код идентичен предыдущему.

5.4.5. Массивы.

И последнее, с чем мы сегодня познакомимся, это будут *массивы*. Конечно же, я не стал расписывать сейчас все возможные типы, потому что сейчас тебе ещё рано о них знать. Мы познакомимся с остальными типами по мере надобности. Ну а теперь о массивах.

Массив – это просто последовательность переменных одного типа. Например, массив целых чисел будет выглядеть так 15 23 36 41. Для того, чтобы объявить переменную типа массив нужно в разделе VAR написать так:

Имя переменной **: array [***диапазон значений***] of** ***Тип переменных в массиве***

Диапазон значений оформляется виде *Начальное значение .. Конечное значение*. Между начальным и конечным значением ставится две точки. Рассмотрим пример объявление массива из 100 целых чисел:

```
VAR
  b:array [0..99] of Integer;
BEGIN
 b[0]:=1;
 b[1]:=2;
END;
```

В этом же примере я показал, как осуществить доступ к элементам массива. Как видишь, это делается так же, как мы получали доступ к отдельным буквам в строках. Я же говорил, что строки – это то же массивы, поэтому доступ к их элементам одинаковый.

5.4.6. Странный PChar.

Из основных типов, которые нам понадобятся в будущем мне осталось рассказать только про тип *PChar*. Этот тип широко используется в WinAPI функциях (функции ОС Windows) и когда мы будем обращаться к ним напрямую, то для передачи строк придется использовать именно этот тип, потому что старые WinAPI функции не могут работать с типом *String*.

Переменная типа *PChar* – это указатель на начало строки, т.е. переменная указывает на первый символ строки в памяти машины. Когда программе надо обратиться к этой переменной, то она идёт по этому адресу и начинает читать строку. В отличии от типа *String*, здесь символы нумеруются с нуля. Но как же тогда программа узнает длину строки? Переменная – это только указатель на начало, символы нумеруются с нулевого, и где же тогда храниться длина строки? Попробуй догадаться. Если ничего не приходит в голову, то я открываю секрет – нигде. Действительно, у строк типа PChar нигде не указывается длина строки.

Для того, чтобы понять, как программа узнаёт длину строки, нужно вспомнить, как хранятся символы строк в памяти машины. Как ты помнишь, каждый символ – это число и у нас есть одно число, которое не используется – ноль. Так вот, когда ты читаешь строку *PChar*, то программа читает все коды символов по указанному адресу, пока не встретиться этот нулевой код. Именно нулевой код является признаком конца строки.

Тип *PChar* нельзя использовать напрямую, потому что это указать на память. По этому указателю должны быть выделена какая-то область памяти. Это значит, что следующий пример будет недействителен:

```
var
 s:PChar
begin
 s:='Привет';
end;
```

В этом примере я объявил строку *s* типа *PChar* и пытаюсь присвоить ей текст. Такая операция невозможна, потому что *s* – указатель и ни на что не указывает. Мы просто его

объявили, но не выделили ему память. Про выделение памяти мы поговорим позже и здесь эту тем затрагивать слишком рано, но один способ объявления такой переменной мы можем рассмотреть:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 s:array[0..200] of char;
 s1:PChar;
begin
 s1:=s;
end;
```

В этом примере я объявил переменную *s*, как массив из 200 элементов типа *char*. Тип *char* – это просто одиночный символ. Получается, что *s* – массив из 200 символов или проще говоря та же самая строка. Я так же объявил переменную *s1* типа *PChar*.

Между *begin* и *end* у меня только одна строчка кода. В ней я присваиваю переменной *s1* значение переменной *s*. Теперь *s1* указывает на область памяти, в которой находиться массив из 200 символов.

Этот способ мы будем использовать редко, потому что чаще всего мы будем работать с типом *String* и когда надо преобразовывать его к типу *PChar*. Но об этом подробнее в разделе о преобразовании типов.

5.4 Процедуры и функции в Delphi

Мы уже познакомились с процедурным программированием на теории. Сейчас нам предстоит узнать, как это выглядит в Delphi.

Процедуры и функции - это некий участок кода, который выделен в отдельный блок. Простая процедура выглядит так:

```
procedure Exampl;
var
 i:Integer;//Объявление локальной переменной
begin
 i:=10;//Присваиваю переменной значение
end;
```

Любая процедура начинается с ключевого слова **procedure.** После этого слова идёт её имя. В нашем примере она называется Exampl.

Внутри процедуры, после слова var идёт объявление локальных переменных. Я объявляю одну переменную i типа integer (целое число). Мы уже знакомы с глобальными переменными, которые появляются при старте программы и уничтожаются после выхода. А это локальная переменная и её можно использовать только внутри этой процедуры. После выхода из неё, переменная автоматически уничтожается.

Код процедуры начинается после слова **begin** и заканчивается после слова **end**. Внутри блока begin ... end процедуры я присваиваю переменной i значение 10. В принципе, ничего не происходит, потому что после присваивания числа процедура заканчивается, и переменная i уничтожается.

Для вызова процедуры нужно написать только Exampl.

Если процедура относится к объекту (то есть является его методом), то нужно написать в объявлении имя объекта, а после точки имя процедуры. Вот пример процедуры относящейся к объекту формы Form1:

```
procedure TForm1.Examp2;
begin
 Exampl; //Вызываем процедуру Exampl написанную ранее.
end;
```

В этой процедуре Examp2 я вызываю написанную ранее Exampl. Все имена процедуры должны начинаться с латинских букв и могут заканчиваться цифрами. Нельзя использовать русские буквы и имена процедур не могут начинаться с цифр.

Если процедура не относится к объекту, то есть ещё одно правило: она должна быть описана до использования. Например:

```
procedure Examp2;
begin
 Exampl; //Произойдёт ошибка, потому что процедура Exampl; описана ниже
end;

procedure Exampl;
var
 i:Integer;
begin
 i:=10;
end;

procedure Examp4;
begin
 Exampl; //Здесь ошибки не будет, потому что Exampl; описан выше.
end;
```

Если процедура относится к объекту, то не имеет значения, где она написана и где её вызываете. Потому что объекты имеют область описания (которую мы уже рассматривали), и она доступна компилятору:

```
type
  TForm1 = class(TForm)
  private:
    procedure Exampl;
    procedure Examp2;
  public:
    procedure Examp3;
    procedure Examp4;
  end;
```

По этому описанию компилятор узнаёт о существовании процедур, поэтому ты можешь их реализовывать в любом порядке, ошибок не будет. Мы уже знакомы с такими описаниями, оно находится в начале любого модуля.

Теперь разберёмся с функциями. Это те же процедуры, только умеют возвращать значения. Простейшая функция выглядит так:

```
function Exampl:Integer;
var
 i:Integer;//Объявление локальной переменной
begin
 i:=10;//Присваиваю переменной значение
 Result:=i; // Возвращаю значение i
end;
```

Я объявил функцию, которая будет возвращать значение типа integer (целое число) function Exampl: Integer. Для возврата значения, его нужно присвоить к переменной Result, как я это делаю в примере выше.

Для вызова функции можно делать так:

```
procedure TForm1.Examp2;
var
 x:Integer;
begin
 x:=Exampl; //Вызываем процедуру Exampl написанную ранее.
end;
```

В этом примере я присваиваю переменной x значение, возвращаемое функцией Exampl.

Все остальные правила объявления функций такие же, как и у процедур. Теперь посмотрим, как можно передавать значения внутрь процедур и функций:

```
function Exampl(index:Integer):Integer;
begin
 Result:=index*2; // Возвращаю переданное значение index
                  // умноженное на 2
end;
```

После имени функции, в скобках указывается тип переменной, который можно передать внутрь функции или процедуры. В моём случае это переменная index типа Integer. После скобок указывается двоеточие и тип возвращаемого значения. Я буду возвращать значение типа Integer.

Что же будет возвращать наша функция? Результат её выполнения можно записывать в **Result** или присваивать самому имени функции. В примере выше я присваиваю результат выполнения index*2 в переменную **Result**. Эта переменная нигде не описана, но она зарезервирована, как переменная, возвращающая значения из функции.

Как я уже сказал, результат можно присваивать и имени функции, вот как это будет выглядеть на примере:

```
function Exampl(index:Integer):Integer;
```

```
begin
 Exampl:=index*2; // Возвращаю переданное значение index
                  // умноженное на 2
end;
```

Оба предыдущих примера выполняют одно и тоже, только записаны немного по-разному.

Вызов моей функции будет такой:

```
procedure TForm1.Examp2;
var
 x:Integer;
begin
 x:=Exampl(20); //Вызываем процедуру Exampl написанную ранее.
end;
```

Я предаю в функцию Exampl значение 20, а она мне вернёт 20 умноженное на 2, то есть 40. Это я показал пример с функцией, но точно так же можно поступать и с процедурами, передавая им значения.

Помни, что процедуры и функции практически одно и то же. Разница только в том, что функции умеют возвращать значения. С этим мы уже знакомы на теории, но теперь увидели это практически на практике, а точнее сказать на реальных примерах.

Попробуй сейчас внимательно посмотреть на следующий пример и найти в нём ошибку:

```
procedure TForm1.Examp2;
var
 x:Integer;
begin
 Result:=x*20;
end;
```

В этом примере я переменной **Result** присваиваю значение вычисления **x*20**. Вроде вы всё правильно, но это же процедура, а она не может возвращать значение. Значит, компилятор может выдать ошибку на то, что переменная **Result** не определена.

И последнее, что я хочу тебе показать – это досрочный выход из процедур/функций. Когда процедура выполняет заложенные внутри неё операторы и встречает оператор **exit**, то она производит моментальный выход из процедуры.

```
procedure TForm1.Examp2;
var
 x:Integer;
begin
 x:=20;
 exit;
 x:=10; // Этот код никогда не будет выполнен.
end;
```

В этом примере я присваиваю в переменную **x** значение 20. Потом программа встречает оператор **exit** и производит мгновенный выход, поэтому строчка с присваиванием переменной **x** значения 10 никогда не будет выполнена.

Этот пример не совсем удачный, поэтому давай рассмотрим функцию, которая будет возвращать результат деления двух переданных значений.

```
function Exampl(index1, index2:Integer):Real;
begin
 Exampl:=index1/ index2;
end;

procedure Examp2;
var
 x: Real;
begin
 x:=Exampl(20, 10); //Вызываем процедуру Exampl написанную ранее.
end;
```

В этом примере я передаю функции два значения **index1** и **index2**. Обе переменные целого типа. В качестве результата я возвращаю результат деления **index1** на **index2**. В процедуре **Examp2** показан пример вызова функции.

Теперь допустим, что в **index2** нам передали 0. В этом случае будет произведена попытка деления на 0, что вызовет ошибку. Вполне логичным было бы сделать проверку, если в **index2** содержится 0, то выйти из функции.

```
function Exampl(index1, index2:Integer):Real;
begin
 Если index2 равен 0, то exit.
 Exampl:=index1/ index2;
end;

procedure Examp2;
var
 x:Real;
begin
 x:=Exampl(20, 0); //Вызываем процедуру Exampl написанную ранее.
 x:=x*2; // Здесь произойдёт ошибка
end;
```

Здесь я написал логику простыми словами, потому что мы будем изучать её в следующей главе, а сейчас нам достаточно только понимать смысл.

В этом примере я произвожу проверку **index2** на ноль и в случае равенства выхожу из функции.

Если **index2** действительно будет равна нулю, то в этом случае функция вернёт неопределённое значение, потому что результат вычисляется после операторы выхода. Если потом попробовать воспользоваться возвращённым неопределённым значением, то произойдёт ошибка. Я специально показал тебе строку с попыткой умножения переменной **x** на 2. Неопределённое значение нельзя использовать. В переменную **x** мы пока ещё не заносили никакого значения, поэтому не используй её.

Чтобы избавиться от таких проблем, можно при входе в функцию сразу задавать значение по умолчанию.

```
function Exampl(index1, index2:Integer):Real;
begin
 Result:=1;
 Если index2 равен 0, то exit.
 Result:=index1/ index2;
end;
```

В этом примере, я в первой же строке присваиваю переменной **Result** значение 1. Теперь у меня результат определён с самого начала. После этого я проверяю второй параметр на ноль и если равенство верно, то произойдёт выход. Теперь после выхода у меня нет неопределённых значений, потому что **Result** уже содержит значение 1.

Если второй параметр не равен нулю, то выполниться деление первого параметра функции на второй и результат запишется в **Result**.

Этот пример наглядно показывает, что **Result** работает как простая переменная, хотя она нигде не описана. Она всегда существует в функциях и имеет тип возвращаемого функцией значения. Если кто-то подумал, что переменной Result надо присваивать значение только в самом конце или по ней происходит выход из функции, то это не так. Result можно изменять где угодно и можно даже использовать как простую переменную для своих нужд.

5.5 Рекурсивный вызов процедур

Ты наверно уже слышал про такое понятие, как *рекурсивный вызов*. Если не слышал, то сейчас узнаешь. *Рекурсивный вызов* – это когда процедура вызывает сама себя. Допустим, что внутри процедуры тебе нужно выполнить абсолютно тот же код, только с другими параметрами. Не писать же из-за этого новую процедуру.

Вспомним нашу классическую задачу – расчёт факториала. Мы уже научились её решать с помощью цикла, а теперь я покажу, как рассчитать факториал с помощью рекурсивного вызова процедур. Хотя этот пример неэффективен и легче (да и быстрее) сделать то же самое с помощью простого цикла, но всё же я покажу этот пример в познавательных целях.

Для расчёта нам понадобиться функция, назовём её *MulNumber*. Ей будет передаваться одно число, а возвращаться будет результат умножения переданного числа на число меньшее на единицу.

```
function TForm1.MulNumber(index: Integer): Integer;
begin
 Result:=Index*Index-1;
end;
```

Если мы будем пользоваться такой функцией, то нам понадобиться вызывать её для каждого числа факториала. Это совсем уже никому ненужно, поэтому давай добавим сюда рекурсию:

```
function TForm1.MulNumber(index: Integer): Integer;
begin
 Result:=Index*MulNumber(index-1);
end;
```

Теперь переменная index умножается на результат вызова функции *MulNumber* с параметром на единицу меньшим чем *Index*. Получается, что прежде чем перемножить Index и *MulNumber* сначала выполниться процедура *MulNumber* и потом уже произойдёт умножение. Но при расчёте *MulNumber* с новым значением опять будет вызвана эта же функция, но с ещё более маленьким значением. В принципе, нас это устраивает, потому что нужно произвести перемножение всех чисел от начального значения Index и до 1. Но как программа узнает о том, что нам нужно остановиться на этой единице? Да никак. Она будет продолжать уменьшать *index* и снова вызывать саму себя с новым параметром. Так переменная *Index* уменьшиться до отрицательного значения и быстро уйдёт в бесконечность. Вот такая ситуация плачевна и приводит к ошибке программы, потому что рекурсию невозможно прервать.

Мы сами должны написать код, который будет прерывать рекурсию:

```
function TForm1.MulNumber(index: Integer): Integer;
begin
 if Index=1 then
  begin
   Result:=1;
   exit;
  end;
 Result:=Index*MulNumber(index-1);
end;
```

Здесь вначале происходит проверка, если *Index* равно 1, то дальше уже рассчитывать не надо и нужно выходить из процедуры. В качестве результата я возвращаю 1, потому что его перемножение на другое число при расчёте факториала не повлияет на результат. Таким образом мы сделали прерывание на 1 и после этого рекурсия заканчивается. При *Index* равной единице не произойдёт очередного вызова процедуры *MulNumber*.

Теперь, чтобы рассчитать пример мы должны всего лишь из любой точки вызвать эту процедуру и указать в качестве параметра число, факториал которого нам надо рассчитать.

Теперь я покажу более полезный алгоритм с использованием рекурсии – поиск файла на диске. Это будет именно алгоритм, потому что показывать сейчас код будет слишком сложным занятием.

```
Function FindFile(Имя файла, Директория);
begin
 Получить список содержимого директории;
 Проверить содержимое директории;
 Если среди файлов нет искомого,
 то вызвать функцию FindFile для вложенных директорий,
 чтобы повторить поиск там.
end;
```

Это чисто абстрактный алгоритм и в реальности код будет работать немного по другому. Но главная его цель – показать принцип рекурсии и более полезный пример её использования.

5.6 Встроенные процедуры

Есть ещё один очень интересный способ использования процедур. Допустим, что у тебя есть какой-то часто повторяющийся код, который может вызываться только из одной процедуры. Если ты опишешь этот код в виде отдельной, независимой процедуры, то его можно будет вызывать где угодно. Но зачем это нужно? Если ты уверен, что код будет вызываться только из одной процедуры, то имеет смысл написать его внутри именно этой процедуры.

Получается, что нам нужно объявить процедуру, внутри другой процедуры. Как это сделать? Да очень просто.

```
procedure Examp1(Sender: TObject);

function Suma(i,j:Integer):Integer;
begin
 Result:=i+j;
end;

var
 i:Integer;
begin
 i:=Suma(10,20);
end;
```

Здесь я объявил функцию *Suma* внутри процедуры *Examp1*, об этом говорит то, что функция описана после объявления процедуры *Examp1* и до начала её раздела **var**. Такую функцию можно без проблем вызывать, но только из кода процедуры *Examp1*. Если попытаться её вызвать из другого места, то произойдёт ошибка:

```
procedure Examp1(Sender: TObject);

function Suma(i,j:Integer):Integer;
begin
 Result:=i+j;
end;

var
 i:Integer;
begin
 i:=Suma(10,20);
end;

procedure Examp2(Sender: TObject);
var
 i:Integer;
begin
```

```
 i:=Suma(10,20); //Здесь будет ошибка.
end;
```

Здесь я добавил новую процедуру *Examp2* и пытаюсь из неё вызвать функцию *Suma*, но то не возможно, потому что здесь эта функция недоступна. Её можно вызывать только из *Examp1*, где она и описана.

Глава 6. Работа с компонентами.

Мы уже создали первую и самую простую программу, и разобрались, из чего она состоит. Помимо этого, я напихал в тебя теории по самые «не хочу», так что пора увидеть всё сказанное на практике.

И вот только теперь мы начнём знакомиться с компонентами и писать примеры с их использованием. Вот это как раз то, с чего начинается большинство книг по Delphi. Но я не буду их повторять, а постараюсь описать как можно подробнее и сказать то, что недосказано в других книгах.

Изучение компонентов будет постепенное. Сначала мы попробуем их в действии и познакомимся с их основными возможностями, а потом залезем внутрь и разберём все косточки. Но это потом. Сейчас мы смотрим только на одёжку.

В Delphi для тебя уже собрана большая коллекция компонентов, позволяющих визуально создать любой интерфейс программы. В этой главе мы познакомимся с тремя основными закладками палитры компонентов: standard, additional и Win32. Но прежде чем это сделать, я расскажу про основную форму.

6.1 Основная форма и её свойства.

Основная форма – это окно будущей программы. На нём ты можешь располагать визуальные компоненты в любом виде и порядке.

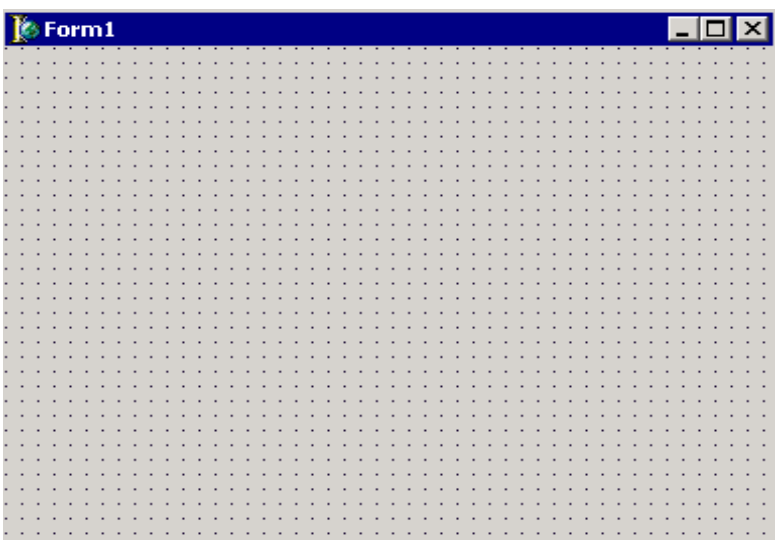


Рисунок 6.1 Основная форма

Как ты уже знаешь, если выделить какой-то компонент, то в объектном инспекторе появятся его свойства и события. Сейчас я рассмотрю основные свойства и события формы. Большинство из них присутствуют и у компонентов. Поэтому в будущем я уже не

буду повторяться: Когда я буду описывать событие, которое может быть не только у формы, но и у компонента, то я так и буду писать «форма/компонент».

Я буду показывать свойства и их назначения, а ты создай новый проект в Delphi и пробуй играться с этими свойствами. Так ты лучше сможешь понять, на что они влияют.

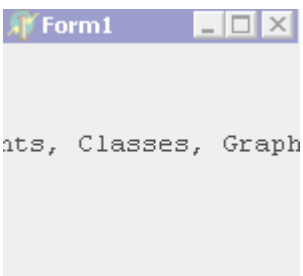
ActiveControl - Указывает на компонент, который должен быть активным по умолчанию.

Align - Выравнивание компонента. Любой компонент может быть выровнен по одной из сторон родительского компонента. Этому свойству можно присвоить следующие значения:

alNone – нет выравнивания. Как нарисовал, так и будет.
alBottom – выравнивание по нижнему краю.
alLeft - выравнивание по левому краю.
AlRight - выравнивание по правому краю.
AlTop - выравнивание по верхнему краю.

Компоненты выравниваются относительно формы, а форма выравнивается относительно окна.

AlphaBlend - Тип свойства – логический. Свойство формы. Означает, имеет ли форма прозрачность. Если это свойство равно true, то окно будет прозрачным. На рисунке справа показан пример полупрозрачного окна. Степень прозрачности задаётся через свойство AlphaBlendValue.

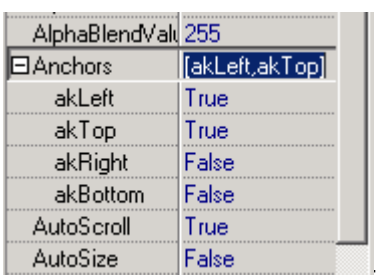


ВНИМАНИЕ!!! Прозрачность работает не на всех системах.

AlphaBlendValue - Тип свойства – целое число. Степень прозрачности формы. Здесь можно задавать числовое значение степени прозрачности от 0 до 255. Если поставишь 0, то форма будет абсолютно прозрачной. 255 означает полную непрозрачность. Чтобы сделать форму полупрозрачной, нужно выставить какое-нибудь промежуточное значение (можно 127).

*На компакт диске, в директории **\Примеры\Глава 6\Прозрачное окно** ты можешь увидеть пример программы. Могу дать гарантию, что он работает в Windows 2000. Но в Win9x пример может не работать.*

Anchors - Это свойство есть и у формы и у компонентов. Оно показывает, как происходит закрепление к родительскому объекту. Это свойство раскрывающееся. Если ты щелкнешь по квадрату слева от имени свойства, то раскроется список из четырёх дополнительных свойств:



akLeft – прикреплять левый край (по умолчанию true).
akTop – прикреплять верхний край (по умолчанию true).
akRight – прикреплять правый край (по умолчанию flase).
akBottom – прикреплять нижний край (по умолчанию false).

По умолчанию прикрепление происходит по левому и верхнему краю.

AutoScroll - Тип свойства – логический. Будет ли форма автоматически производить скроллинг, или нет.

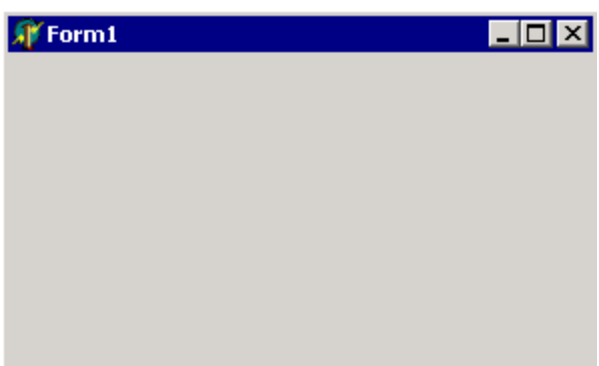
AutoSize - Тип свойства – логический. Должны ли компоненты на форме автоматически корректировать размеры.

BorderIcons - Свойство определяющее, какие кнопки должны присутствовать у окна. Это свойство раскрывающееся. Если ты щелкнешь по квадрату слева от имени свойства, то раскроется список из четырёх свойств:

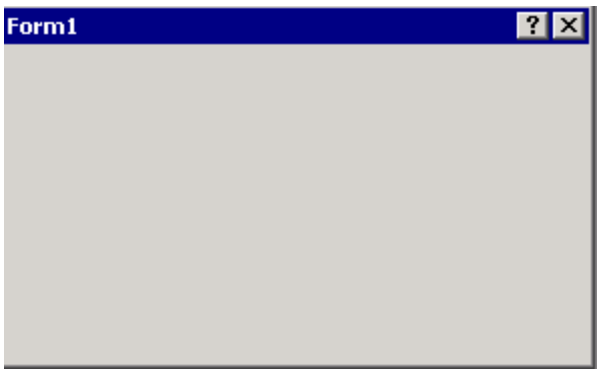
biSystemMenu – показать меню (иконка слева в строке заголовка окна) и другие кнопки заголовка окна.
biMinimize – кнопка минимизации окна.
biMaximize – кнопка максимизации окна.
biHelp – кнопка помощи.

BorderStyle – Свойство формы. Отвечает за вид оборки окна. Это свойство может принимать следующие значения.

bsSizeable – установлено по умолчанию. Стандартное окно, с нормальной оборкой, которое может изменять свои размеры. Смотри скрин окна:

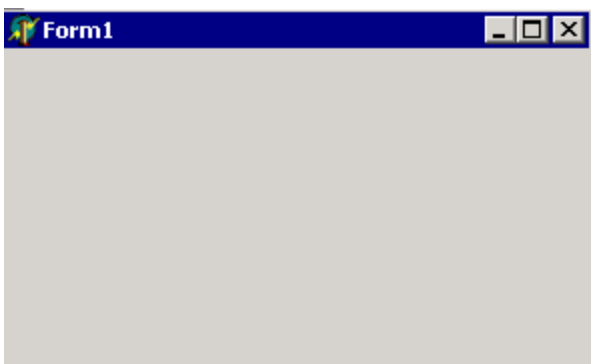


bsDialog – окно выглядит в виде диалога. Смотри скрин окна:

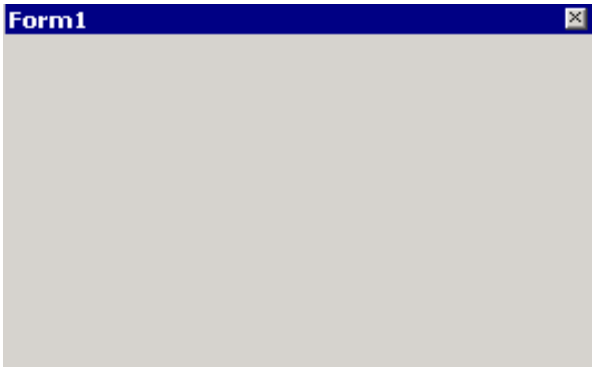


bsNone – окно вообще без оборки. Смотри скрин окна:

bsSingle – На первый взгляд это простое окно, а если попробовать изменить его размеры, то можно получить облом. Это окно с фиксированным размером и изменять его мышкой нельзя. Изменить размер можно только кнопкой Maximize. Смотри скрин окна:



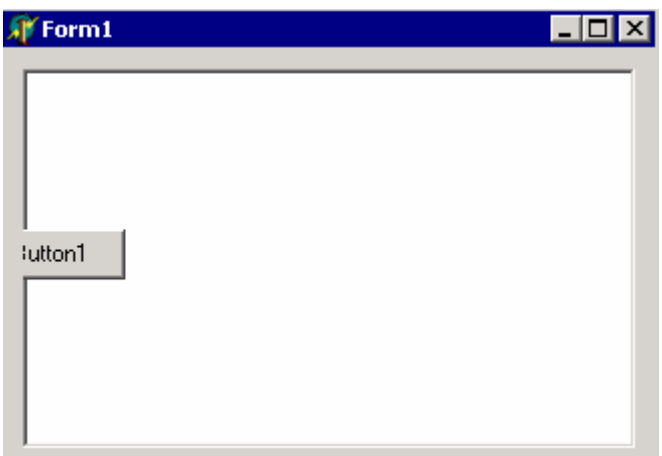
bsSizeToolWin – окно с тонкой оборкой. Особенно это заметно в заголовке окна. Смотри скрин окна:



bsToolWindow – я не буду приводить скрин этого окна, потому что он ничем не отличается от предыдущего. Единственная разница – у этого окна нельзя изменять размеры окна.

Все примеры окон можно найти на компакт диске в директории ***\Примеры\Глава 6\Окна****. Но я тебе советую попробовать самому создать приложение и поиграть с его свойствами.*

BorderWidth – ширина оборки окна. Пока что все окна, которые мы рассматривали, имели ширину оборки равную нулю. На скрине ниже показано окно с оборкой равной 10. В центре окна растянута на всю площадь поле для ввода текста и несмотря на это, в окне по краям видны широкие оборки. Для больше го эффекта я поместил на форму ещё и кнопку, которая исчезает при пересечения оборки.

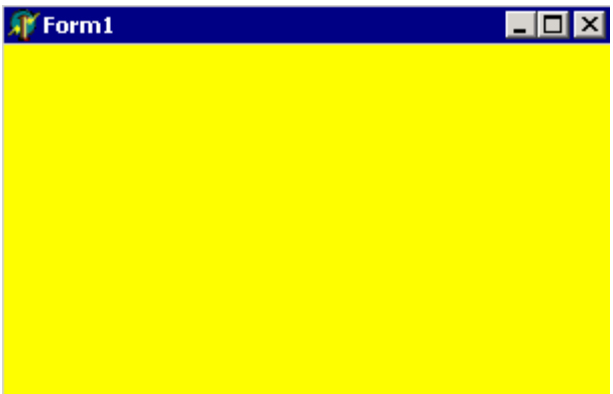


Caption – это строковое свойство, которое отвечает за заголовок окна. Мы уже использовали его, когда писали программу «Hello World».

ClientHeight – это свойство в виде целого числа показывает высоту клиентской области окна. Это высота без учёта ширины оборки и системного меню, только рабочая область.

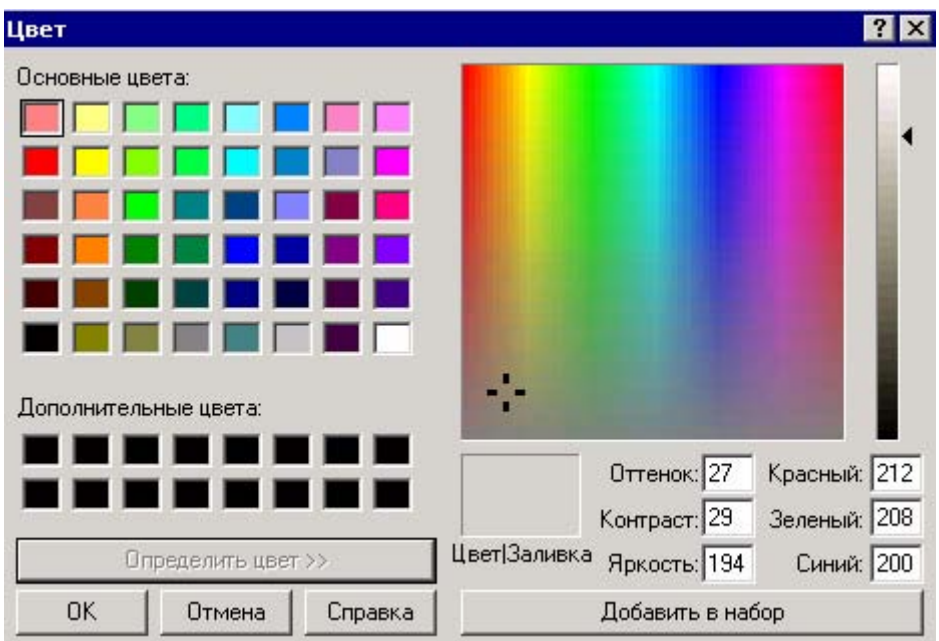
ClientWidth - это свойство в виде целого числа показывает ширину клиентской области окна. Это ширина без учёта ширины оборки и системного меню, только рабочая область.

Color – цвет клиентской области окна.



На компакт диске, в директории ***\Примеры\Глава 6\Цвет окна*** *ты можешь увидеть пример программы.*

В списке выбора есть все системные цвета, которые ты можешь выбрать. Но если ты хочешь использовать какой-то специфичный цвет, то можешь дважды щёлкнуть по этому параметру, и перед тобой откроется стандартное окно выбора цвета:



Constraints – в этом свойстве содержатся максимальные значения размеров окна.

MaxHeight – максимальная высота окна.
MaxWidth – максимальная ширина окна.
MinHeight – минимальная высота окна.
MinWidth – минимальная ширина окна.

Если ты установишь эти значения, то окно нельзя будет растянуть больше максимального размера и уменьшить меньше минимального.

*На компакт диске, в директории **\Примеры\Глава 6\Размер окна** ты можешь увидеть пример программы в которой главное окно нельзя увеличить более чем 400x400 и меньше 200x200.*

Ctl3D – Тип свойства – логический. Оно указывает - показывать окно/компонент в псевдо 3D плоскости или нет. Этот параметр остался ещё от Windows 3.1, когда он действительно имел смысл. Сейчас даже если ты отключишь 3D, окно сильно не изменится. Поэтому про это свойство можно забыть.

Cursor – это свойство отвечает за курсор, который будет отображаться при наведении мышкой на форму/компонент. Тебе доступны следующие курсоры:

Имя курсора	Вид	Имя курсора	Вид
crNone	Нет	CrArrow	
crCross		crIBeam	
crSizeNESW		crSizeNS	
crSizeNWSE		crSizeWE	
crUpArrow		crHourGlass	
crDrag		crNoDrop	
crHSplit		crVSplit	
crMultiDrag		crSQLWait	
crNo		crAppStart	
crHelp		crHandPoint	
crSize		crSizeAll	

DockSite - Тип свойства – логический. Указывает, можно ли на форму/компонент бросать другие компоненты с помощью Drag&Drop. Это свойство создаёт эффект, который ты мог наблюдать в MS Office, когда панели инструментов можно отрывать от формы и прикреплять обратно. Вот это свойство как раз и разрешает прикреплять компоненты.

*На компакт диске, в директории **\Примеры\Глава 6\Dock** ты можешь увидеть пример программы в которой можно отрывать панель от формы и прикреплять обратно.*

DragKind – вид перетаскивания объекта при Drag&Drop. Здесь тебе доступны два варианта:

dkDrag – стандартный Drag&Drop при котором объект остаётся на месте.

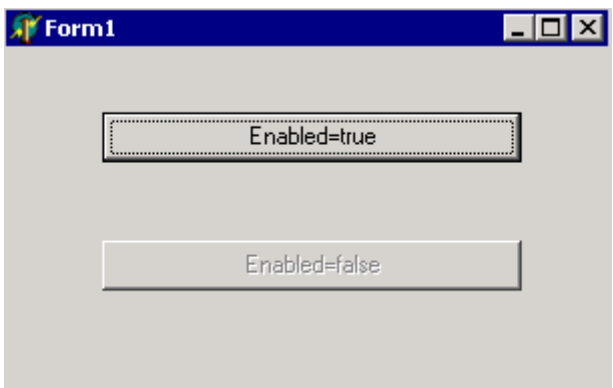
dkDock – перетаскивать сам объект. Этот параметр следует выбрать, если нужно чтобы компонент мог прикрепляться к другим компонентам или форме. В примере к предыдущему свойству панель имеет именно такое свойство, чтобы она могла прикрепиться к форме.

DragMode – режим Drag&Drop. Здесь тебе доступны два варианта:

dmManual – ручной режим. При таком режиме ты сам должен запускать перетаскивание объекта.

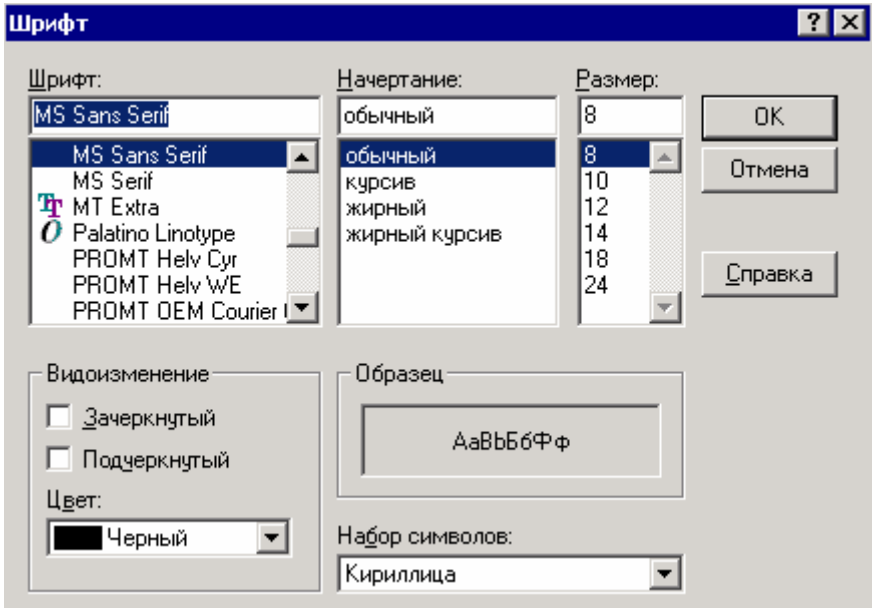
dmAutomatic – режим Draq&Drop будет включатся автоматически, если пользователь начал тащить мышкой компонент. При этом не нужно писать дополнительный код, как при ручном режиме.

Enabled – Тип свойства – логический. Доступность компонента. Если это свойство равно true, то пользователь может работать с этим компонентом. Иначе компонент недоступен и окрашен серым цветом.



На компакт диске, в директории ***\Примеры\Глава 6\Enable*** *ты можешь увидеть пример программы в которой есть две кнопки на форме. Одна из них доступна, а другая нет.*

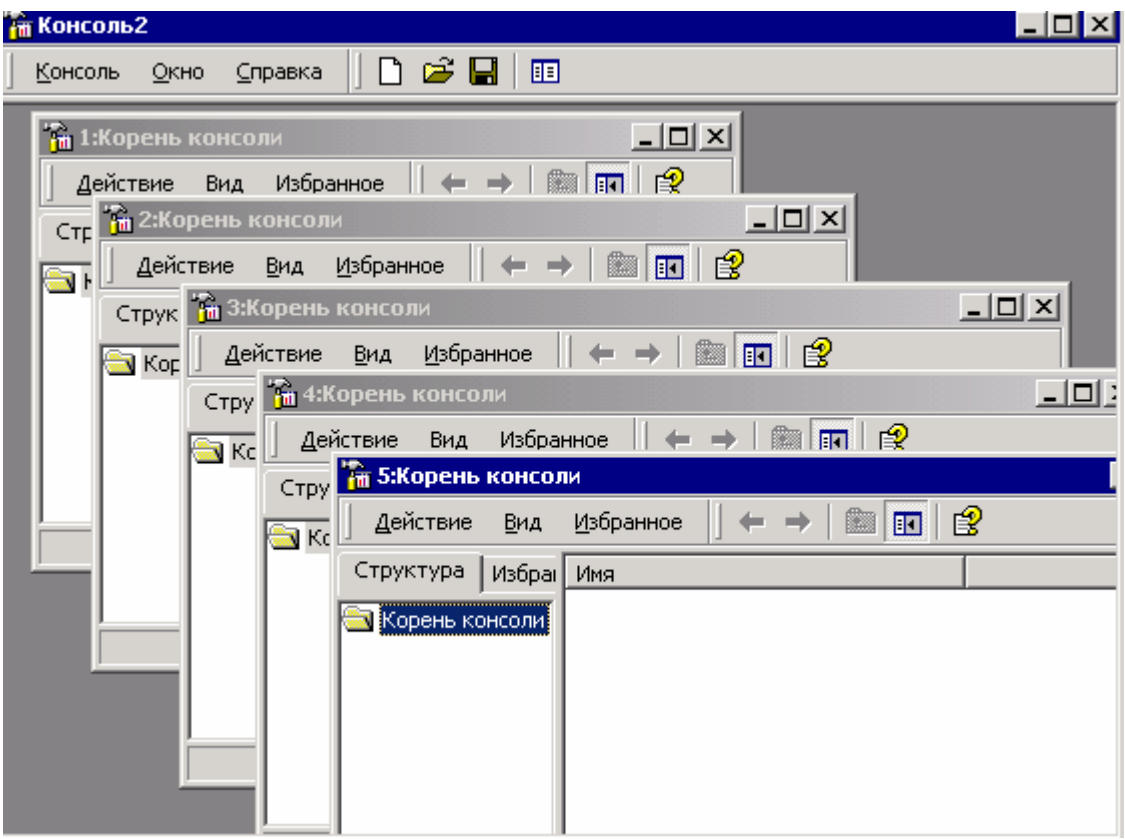
Font – шрифт используемый при выводе текста на форме. Если ты дважды щёлкнешь по этой строке, то перед тобой появится стандартное окно Windows выбора шрифта:



FormStyle - Стиль формы. Здесь тебе доступны для выбора следующие варианты

fsNormal – нормальное окно.

fsMDIForm – Окно является родительским для MDI окон. Если ты помнишь старый Office, то должен помнить, что там внутри основного окна можно было перемещать другие окна. Это окна относятся к классу MDI – мультидокументные окна. Хотя Microsoft не рекомендует использовать MDI окна и вроде как сама отказалась от их использования, а в Windows 2000 консоль MMC выполнена именно так:



fsMDIChild – окно является дочерним MDI окном. *fsMDIForm* - создаёт главное окно, а *fsMDIChild создаёт дочернее, то есть то окно, которое будет внутри главного.*
ВНИМАНИЕ!!! – главное окно не может быть такого типа.

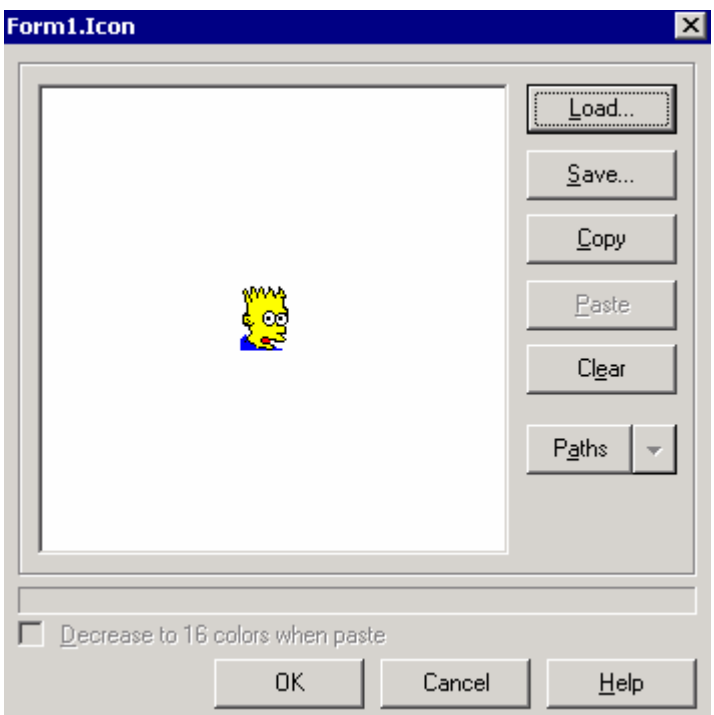
fsStayOnTop – Окно с этим параметром будет находиться всегда поверх остальных.

Height – Тип свойства – целое число. высота окна.

Hint – текст подсказки, который будет появляться в строке состояния при наведении мышкой на форму/компонент.

HorzScrollBar – параметры горизонтальной полосы прокрутки. Этот параметр я пока не буду рассматривать, потому что если его раскрыть, то там будет столько настроек, что это тема отдельного разговора.

Icon – иконка отображающаяся в заголовке окна. Если дважды щёлкнуть по этому свойству, то появится окно загрузки иконки.



В этом окне есть следующие кнопки:

Load – загрузить иконку из файла.
Save – сохранить иконку в файл.
Copy – копировать иконку в буфер обмена.
Paste – Вставить иконку из буфер обмена.
Clear – очистить текущую иконку.

Left – Тип свойства – целое число. левая позиция окна.

Menu – меню, которое используется в главном окне. Этот параметр стоит рассмотреть отдельно.

Name – имя формы/компонента. Помни, что какое имя ты здесь введёшь, так и будет называться объект, отвечающий за эту форму/компонент (только в начале добавится буква T).

Мы с тобой уже рассмотрели исходник, который сгенерировал Delphi для пустого проекта. Там объект назывался TForm1. Обрати внимание, что здесь написано просто From1. Если ты попробуешь изменить свойство *Name*, то и имя объекта измениться.

Старайся давать формам/компонентам понятные имена. Так легче будет понять, для чего они предназначены. Я думаю, что удобней будет работать с компонентом, если у него имя будет ExitButton или NewFileButton, а не просто Button1 и Button2.

ParentFont – Тип свойства – логический. Если это свойство равно true, то для вывода текста будет использоваться тот же шрифт, что и у родительского объекта. Иначе используется тот, что укажешь ты.

Position – Позиция окна при старте приложения. Здесь тебе доступны следующие варианты:

poDefault – Windows сам будет решать, где расположить окно и какие будут его размеры.

poDefaultPosOnly - Windows сам будет решать только где расположить окно, а размеры его будут такими, какими установишь ты в свойствах.

poDefaultSizeOnly - Windows будет решать только какими будут размеры окна, а позиция будет такая, какую ты укажешь в свойствах.

poDesigned – И размер, и позиция будут такими, какими ты укажешь в свойствах.

poDesktopCenter – окно будет располагаться по центру рабочего стола.

poMainFormCenter – окно будет располагаться по центру основной формы.

poOwnerFormCenter – окно будет располагаться по окна владельца. То есть того окна, которое вызвало это.

poScreenCenter - окно будет располагаться по центру экрана.

ShowHint – Тип свойства – логический. Оно показывает - нужно ли показывать подсказки.

Tag – это свойство имеет тип – целое число. Оно ни на что не влияет и ты можешь его использовать в своих целях.

Top – Тип свойства – целое число. верхняя позиция окна.

TransparentColor – Тип свойства – логический. Является ли форма или компонент прозрачным. В отличии от AlphaBlend, эта прозрачность работает всегда. Зато нельзя сделать полупрозрачные формы и компонеты.

TransparentColorValue - прозрачный цвет.

VertScrollBar – Вертикальная полоса прокрутки. Она имеет те же параметры, что и горизонтальная и мы рассмотрим её отдельно.

Visible - Тип свойства – логический. Если оно равно true, то форма/компонент видимые. Иначе форма/компонент невидим.

Width - Тип свойства – целое число. Ширина окна.

WindowState – состояние окна после запуска. Тебе доступны следующие параметры:

wsNormal – окно показывается в нормальном состоянии.

wsMaximized - окно показывается максимизированным.

wsMinimized - окно показывается минимизированным.

На этом обзор свойств формы считаю законченным.

6.2 Событийная модель Windows.

Вся работа операционной системы (ОС) Windows основана на понятии *события*. Что это значит? Давай попробуем разобраться.

Внутри ядра Windows создаётся очередь событий. Когда какое-нибудь приложение или устройство, изменило своё состояние и хочет сообщить об этом операционной системе, то оно помещает в эту очередь соответствующее сообщение. ОС Windows обрабатывает его и если необходимо, то реагирует на изменения.

Давай рассмотрим реальный пример события и реакции на него. Допустим, что мы передвинули курсор мыши. Она генерирует событие и помещает его в очередь сообщений. Когда Windows доходит до обработки этого сообщения, то он получает новые координаты курсора мышки. Так как положение курсора изменилось, ОС должна перерисовать его в новой позиции на экране. После этого, Windows переходит к обработке следующего сообщения.

Если в очереди нет сообщений, то Windows переходит в состояние ожидания. Но такое бывает очень редко. Даже когда ты не работаешь за компьютером, и он простаивает, в фоне работает очень много процессов, которые отнимают процессорное время и генерируют свои события.

Когда ты нажимаешь на кнопку, также генерируется событие, что кнопка нажата. Любое действие, которое несёт в себе какие либо изменения может генерировать системное событие. Это очень эффективная и удобная модель, благодаря которой и реализуется многозадачность Windows.

Давай взглянём на простую очередь (она не является очередью ОС Windows, это просто пример):

Событие	Идентификатор приложения	Дополнительно
Нажата клавиша	261	A
Перерисовать экран	385	(12, 46, 336, 267)
Перемещена мышь	261	(356,451)

Первая колонка показывает тип события. Вторая колонка показывает идентификатор приложения, которое сгенерировало событие. В третьей показываются дополнительные параметры. Так, например, при нажатии клавиши на клавиатуре, в качестве дополнительного параметра идёт буква, которую нажали. Конечно же, в реальной ситуации будет не буква, но у нас же это просто пример.

ОС Windows берёт первую строку из очереди и обрабатывает её. Потом берёт вторую строку. Она уже относится к другому приложению. Третья строка опять относится к первому приложению. Таким образом, ОС последовательно обрабатывает события разных приложений, что даёт многозадачность.

Конечно же, многозадачность построена не только на сообщениях и здесь много дополнительных факторов. Но очереди играют достаточно большую роль.

В Delphi все компоненты так же работают через события. Ты будешь постоянно создавать обработчики событий для разных ситуаций. Например, можно создать обработчик события для нажатия клавиши на клавиатуре и производить в нём какие-то действия. Например, по нажатию определённой клавиши, можно выводить окно (действие как у горячих клавиш).

Обработчик события – это простая процедура или функция, которая вызывается по наступлению какого-то события.

6.3 События главной формы.

Здесь я дам описание большинству событий, которые может отлавливать главная форма приложения. Конечно же, тебе доступно намного больше и ты можешь создать ещё свои обработчики, но я опишу наиболее часто используемые события. События можно увидеть на закладке Events объектного инспектора.

Событие	Описание

OnActivate	Когда приложение стало активным
OnCanResize	Это событие генерируется перед тем, как изменить размер окна. Здесь ты можешь запретить какие-либо изменения или производить какие-то подготовительные действия.
OnClick	Генерируется, когда пользователь щёлкнул по форме.
OnClose	Генерируется, когда окно закрывается.
OnCloseQuery	Генерируется до закрытия окна. В этом обработчике происходит запрос на закрытие, поэтому из этого обработчика можно вывести окно, которое будет запрашивать подтверждение на закрытие. Ты такие подтверждения видишь в каждом втором приложении типа «Вы уверены, что хотите закрыть окно?».
OnCreate	Генерируется, когда окно создаётся.
OnDblClick	Генерируется, когда пользователь дважды щёлкнул по окну.
OnDeactivate	Генерируется, когда окно деактивируется.
OnDestroy	Когда окно уничтожается.
OnHide	Генерируется, когда окно исчезает из виду. Событие генерируется даже тогда, когда память, выделенная для окна, не уничтожается.
OnKeyDown	Генерируется, когда нажата клавиша на клавиатуре.
OnKeyPress	Генерируется, когда нажата и отпущена клавиша на клавиатуре.
OnKeyUp	Генерируется, когда отпущена клавиша на клавиатуре.
OnMouseDown	Генерируется, когда нажата кнопка мыши.
OnMouseMove	Генерируется, когда двигается мышка.
OnMouseUp	Генерируется, когда отпускается кнопка мыши.
OnMouseWheel	Генерируется колёсиком мыши.
OnMouseWheelDown	Генерируется, когда колёсико мыши прокручено вниз.
OnMouseWheelUp	Генерируется, когда колёсико мыши прокручено вверх.
OnPaint	Генерируется, когда надо перерисовать окно.
OnResize	Генерируется, когда надо изменить размеры окна.
OnShortCut	Когда нажата горячая клавиша.
OnShow	Когда показывается окно, но до фактической прорисовки. В этот момент окно уже создано и готово к отображению, но ещё не прорисовалось на экране.

Это основные события, которые может генерировать форма. Когда я буду рассматривать компоненты, то я не буду снова расписывать те события, которые уже перечислены здесь, потому что разницы в них нет. Поэтому я буду затрагивать только те события, которые специфичны для конкретного компонента.

6.4 Палитра компонентов.

В этой части мы будем знакомиться с основными компонентами Delphi, которые расположены на палитре компонентов.

Палитра компонентов состоит из нескольких закладок:

1. ***Standard***. Все эти компоненты являются аналогами Windows компонентов. Хотя то, что находится в Windows – это не компоненты, это кал :). А вот Borland превратил этот кал в конфетку и сделал его доступным нам.
2. ***Additional*** – дополнительные компоненты.
3. ***Win32*** – компоненты, которые есть только в семействе Win32 операционных систем. В это семейство входят Windows 9x, Windows ME, Windows 2000, Windows NT, Windows XP. Наверно легче было сказать, что не входит, потому что это только Windows 3.1.
4. ***System*** – системные компоненты, с помощью которых облегчается доступ к системе.
5. ***Database Access*** – компоненты доступа к базам данных.
6. ***Data Controls*** – компоненты для работы с базами данных.
7. ***dbExpress*** – ещё компоненты доступа к базам данных, которые пришли на смену BDE.
8. ***BDE*** – старые компоненты доступа к базам данных.
9. ***ADO*** – это тоже компоненты для доступа к базам данных, только по технологии Active Data Object (ADO).
10. ***InterBase –*** компоненты доступа к базе данных InterBase.
11. ***WebServices*** – компоненты доступа к сети Интернет.
12. ***InternetExpress*** – компоненты доступа к сети Internet.
13. ***FastNet*** – Сетевые компоненты. Мы врят ли будем их использовать.
14. ***QReport*** – компоненты для создания отчётности.
15. ***Dialogs*** – компоненты облегчающие доступ к стандартным диалогам.
16. ***Win3.11*** – компоненты доступа к компонентам Win 3.1.
17. ***Samples*** – различные примеры. Некоторые из этих компонентов доступны в исходных кодах и поставляются вместе с Delphi.

На этом пока хватит. Давай переходить к следующей главе, где мы познакомимся с палитрой компонентов Standard и входящими в неё компонентами. В процессе этого мы будем учиться работать не только с компонентами, но и с самим языком программирования Delphi.

Глава 7. Палитра компонентов Standard.

В этой главе моей книги я буду рассказывать о компонентах, которые находятся на закладке Standard палитры компонентов. Одновременно с этим мы будем писать программы с использованием этих компонентов, и изучать язык программирования Delphi.

Здесь я не буду просто перечислять компоненты и их свойства. Мы будем писать вполне работающие приложение. Функциональность их пока будет очень слабая, но всё же программы будут вполне рабочими. Я постараюсь писать примеры как можно интереснее, но это уже больше зависит от самих компонентов.

Несмотря на то, что мне ещё надо рассказать немного про основы языка программирования Delphi, я решил сначала рассказать про компоненты, чтобы мы хоть немного попрактиковались в программировании. Ну а потом я расскажу уже последнюю часть теории самого языка Delphi. После этого нам останется только изучать компоненты и различные технологии.

Я ещё раз прошу тебя, повторяй все действия за мной самостоятельно. Я понимаю, что на диске есть все исходники моих примеров, но это мои исходники. Ты сможешь чему-то научиться, только если сам попробуешь. Поэтому я стараюсь дать максимальное количество практики.

Рис 7.1 Палитра компонентов, закладка «Standard».

Итак, переходим к рассмотрению компонента кнопка. Хотя он находится в середине закладки, я решил начать с него, потому что он проще и при рассмотрении других компонентов мы будем постоянно использовать эту кнопку.

7.1 Кнопка (TButton).

Кнопка в Delphi происходит от объекта *TButton*. Когда ты устанавливаешь на форму новую кнопку, то ей даётся имя по умолчанию *Button1*.

Давай напишем маленькую программу с использованием кнопки. Для этого создай новое приложение. Теперь щёлкни по изображению кнопки  на палитре компонентов. После этого щёлкни по форме в любом месте. На форме сразу же появится кнопка с заголовком *Button1*.

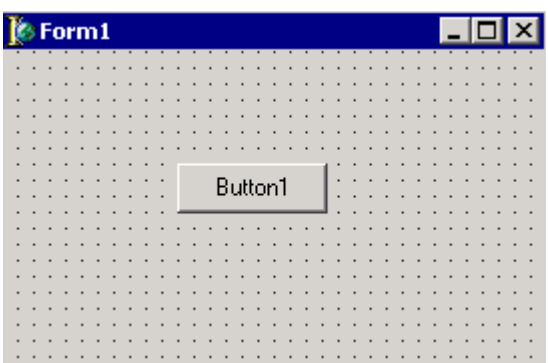


Рис 7.1.1 Форма с кнопкой.

Есть ещё один способ установить кнопку на форму – дважды щёлкнуть по изображению кнопки. Но в этом случае, кнопка окажется в центре формы, а не там, где мы хотим.

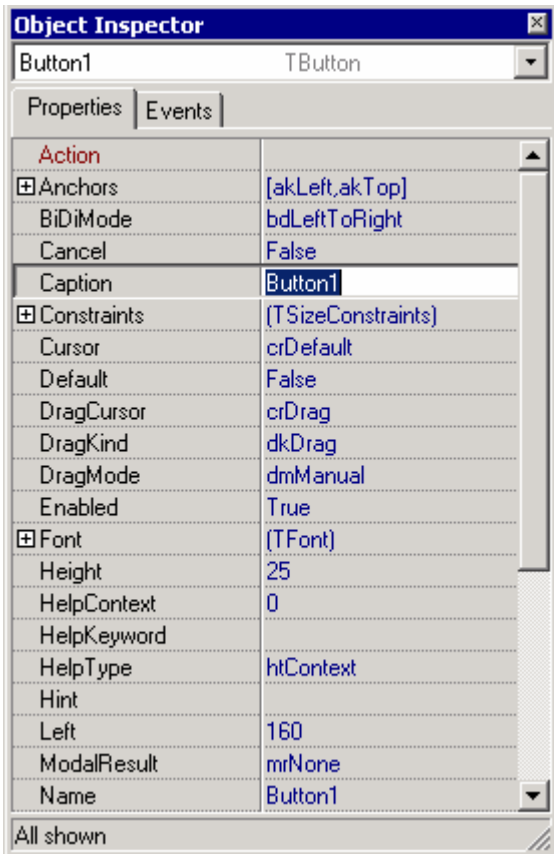


Рис 7.1.2 Объектный инспектор со свойствами кнопки.

Выдели кнопку и перейди в объектный инспектор. Сейчас здесь показаны свойства кнопки. Оглянись. Как видишь, большинство свойств нам уже знакомо, по свойствам формы, поэтому я не буду их расписывать. Есть только одно новое свойство – ModalResult, но мы с ним познакомимся позже.

Давай изменим заголовок кнопки. За заголовок формы у нас отвечало свойство *Caption*. Здесь то же самое. Найди свойство *Caption* и измени содержащийся там текст на «Нажми меня».

Давай сразу изменим свойство *Name* у кнопки. Я говорил, что любым компонентам и формам лучше давать понятные имена, чтобы не было бардака. Так давай с самого начала будем привыкать к нормальной жизни. Найди свойство *Name* и измени его на ***MyFirstButton***. Пускай имя кнопки пока не отражает никакого смысла, ведь она ещё ничего не делает.

Давай также изменим имя формы. Для этого сними выделение с кнопки (щёлкни в любом месте формы). Вверху окна, должна загореться надпись *Form1 TForm1*, как на рис 7.1.3.

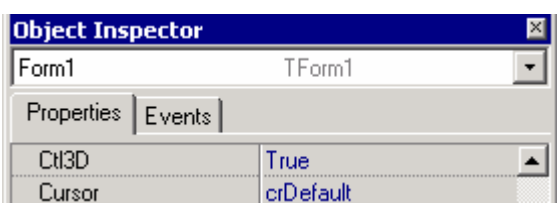


Рис 7.1.3 Объектный инспектор со свойствами кнопки.

Теперь найди здесь свойство *Name* (оно должно быть равно *Form1*) и измени его значение на *MainForm* (это переводится как - главная форма).

Попробуй запустить программу (нажми F9). Ты можешь спокойно нажимать на кнопку, только ничего не происходит.

Давай усложним наш пример и поймаем событие, когда нажимается кнопка. Для этого перейди на закладку Events. Когда мы рассматривали события формы, я говорил, что за клик мышкой отвечает событие OnClick. Для кнопки есть такое же событие. Найди его и щёлкни по нему дважды. Delphi должен создать в редакторе кода процедуру - обработчик события OnClick. По умолчанию ей даётся имя в виде имени компонента (нашей кнопки) плюс имя события без приставки *On*. В нашем случае получается, что имя процедуры обработчика получается *MyFirstButtonClick*.

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

end;
```

В объектном инспекторе, напротив строки OnClick тоже должно появится имя процедуры обработчика (смотри рисунок 7.1.4).

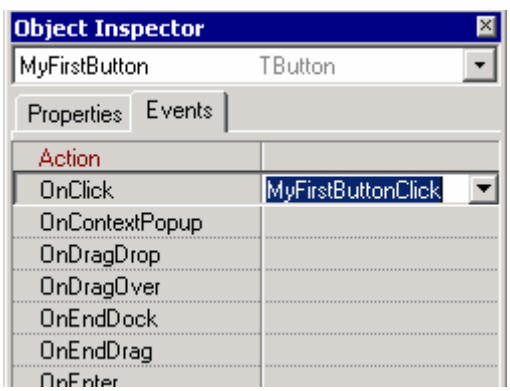


Рис 7.1.4 Закладка Events

Давай вернёмся в редактор кода и посмотрим, что там создал для нас Delphi? Это процедура *MyFirstButtonClick*. Ей передаётся один параметр *Sender* объектного типа *TObject*. При начале выполнения процедуры, в переменной *Sender* будет находиться указатель на объект, который вызвал этот обработчик. Это очень важно, потому что одна процедура обработчик может обрабатывать нажатия сразу нескольких кнопок. Или вообще, компоненты разного типа. По содержимому этой переменной можно узнать, какой именно компонент сгенерировал событие. Дальше мы будем использовать эту возможность, когда это понадобится.

Давай напишем внутри процедуры (между *begin* и *end*) команду *Close*. Эта команда закрывает окно. Теперь наша процедура должна выглядеть так:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  Close;
end;
```

Попробуй запустить программу и нажать на кнопку. Программа закроется.

 *На компакт диске, в директории* ***\Примеры\Глава 7\Button*** *ты можешь увидеть пример программы.*

Откуда я взял, что команда **Close** закроет нашу программу? А оттуда, что это метод формы. Мы могли просто написать так:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  Form1.Close;
end;
```

Это то же самое. Разницы абсолютно никакой нет. Так почему же я написал просто *Close*. Да потому что процедура относится к объекту *Form1* и внутри неё я имею право использовать его свойства и метода без указания владельца. По умолчанию будет браться *Form1*. Но если я захочу из этой процедуры закрыть другую форму, например *Form2*, то мне придётся указать, что я хочу именно *Form2*.

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
  Form2.Close;
end;
```

Если я просто напишу *Close*, то закроется *Form1*, а не *Form2*.

7.2 Играем со свойствами кнопки (логические операции).

Этот пример я уже описывал в журнале «Хакер». Он достаточно простои и удобный для обучения. Сейчас я снова напишу программу, у которой будет только одна кнопка. При наведении на неё мышкой, кнопка будет убегать.

Воспользуемся предыдущим примером и улучшим его. Для начала выдели форму и измени свойство *AutoScroll* на *False*, чтобы на форме не появлялись автоматически полосы прокрутки.

Теперь создай для кнопки обработчик события *OnMouseMove*. Для этого выдели кнопку и перейди в объектном инспекторе на закладку Events. Здесь ты уже создавал обработчик *OnClick*, теперь щёлкни дважды напротив строки *OnMouseMove*, чтобы создать соответствующий обработчик.

Если ты всё сделал правильно, то Delphi должен создать следующую процедуру для обработки сообщения *OnMouseMove*.

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

end;
```

Напиши здесь следующее:

```
procedure TForm1.MyFirstButtonMouseMove(Sender: TObject;
  Shift: TShiftState; X, Y: Integer);
var
 index:integer;
begin
 index:=random(4);
 case index of
  0: MyFirstButton.Left:=MyFirstButton.Left+MyFirstButton.Width;
  1: MyFirstButton.Left:=MyFirstButton.Left-MyFirstButton.Width;
  2: MyFirstButton.Top:=MyFirstButton.Top+MyFirstButton.Height;
  3: MyFirstButton.Top:=MyFirstButton.Top-MyFirstButton.Height;
 end;

 if MyFirstButton.Left<0 then
   MyFirstButton.Left:=0;

 if (MyFirstButton.Left+MyFirstButton.Width)>Form1.Width then
    MyFirstButton.Left:=Form1.Width-MyFirstButton.Width;

 if MyFirstButton.Top<0 then
     MyFirstButton.Top:=0;

 if (MyFirstButton.Top+MyFirstButton.Height)>Form1.Height then
    MyFirstButton.Top:=Form1.Height-MyFirstButton.Height;
end;
```

Пока просто перепиши содержимое этого листинга. Скоро мы подробно рассмотрим, что тут написано. Запусти программу и попробуй нажать на кнопку. Как только ты попытаешь навести на неё мышкой, кнопка будет убегать от тебя.

На компакт диске, в директории ***\Примеры\Глава 7\Button1*** *ты можешь увидеть пример этой программы.*

Обязательно сначала посмотри, как работает пример. Когда наиграешься, возвращайся к чтению книги, и мы рассмотрим исходник.

В разделе **Var** я объявил одну переменную *index* типа целое число. В первой строчке я присваиваю этой переменной случайное число с помощью функции *random*:

```
index:=random(4);
```

Функция *random* возвращает случайное число. В качестве единственного параметра ей нужно передать число, которое будет означать максимально возможное случайное число. Я передаю цифру 4. Это значит, что функция вернёт мне число от нуля до четырёх (0<=X<4). Само число 4 в диапазон возможных значений не входит, все случайные числа будут меньше него.

После этого я проверяю, какое число мне сгенерировала функция *random* с помощью конструкции:

```
case Переменной of
```

```
 Значение1: Действие1;
 Значение2: Действие2;
 …
 …
end;
```

Конструкция ***Case*** сравнивает переменную с перечисленными между ключевыми словами ***of*** и ***end*** значениями и если одно из них совпадает, то выполняет соответствующее действие. Например, допустим, что наша переменная равна числу «Значение2». В этом случае будет выполнено «Действие2». При этом «Действие1» и другие выполнятся не будут.

Если тебе нужно, чтобы при равенстве значений выполнялось несколько действий, то необходимо заключить их в логические кавычки ***Begin … end***. Например:

```
case Переменной of
 Значение1:
   Begin
    Действие1_1;
    Действие1_2;
    …
   End;

 Значение2:
   Begin
    Действие2_1;
    Действие2_2;
    …
   End;
 …
 …
end;
```

Теперь вернёмся к нашему примеру. В нём используется следующий ***Case***:

```
case index of
  0: MyFirstButton.Left:=MyFirstButton.Left+MyFirstButton.Width;
  1: MyFirstButton.Left:=MyFirstButton.Left-MyFirstButton.Width;
  2: MyFirstButton.Top:=MyFirstButton.Top+MyFirstButton.Height;
  3: MyFirstButton.Top:=MyFirstButton.Top-MyFirstButton.Height;
 end;
```

Если переменная *Index* равна 0, то выполнится следующее действие:

```
MyFirstButton.Left:=MyFirstButton.Left+MyFirstButton.Width
```

Что это? Здесь я присваиваю свойству *Left* (левая позиция) кнопки *MyFirstButton* значение левая позиция этой же кнопки плюс её ширина. Это значит, если ты попытался навести мышкой на кнопку, и функция *Random* сгенерировала 0, то левое значение кнопки будет увеличено на ширину кнопки. А это значит, что кнопка сдвинется от тебя вправо. Если ты уже запускал пример, то ты уже понял меня.

Если значение переменной *Index* равно 1, то я наоборот, уменьшаю левую позицию кнопки на её ширину. А это значит, что кнопка убежит влево. Если значение переменной *Index* равно 2, то я увеличиваю верхнюю позицию кнопки на её высоту. А это значит, что кнопка убежит вниз. Надеюсь, что смысл понятен. Давай двигаться дальше.

После конструкции ***case .. of .. end*** идёт проверка: «Не убежала ли кнопка за пределы окна. Сначала я проверяю левую позицию кнопки:

```
if MyFirstButton.Left<0 then
  MyFirstButton.Left:=0;
```

Здесь идёт проверка, если левая позиция кнопки (**MyFirstButton.Left**) меньше нуля, то установить её в ноль.

В следующей строке я проверяю, если левая позиция кнопки плюс её ширина больше ширины окна, то левой позиции присвоить значение «ширина окна» минус «ширина кнопки»:

```
if (MyFirstButton.Left+MyFirstButton.Width)>Form1.Width then
   MyFirstButton.Left:=Form1.Width-MyFirstButton.Width;
```

Точно так же я проверяю и верхнюю позицию, чтобы она не вылезла за пределы окна.

В этой проверке я использую конструкцию ***if .. then***. Я уже говорил о ней в теории, а теперь поговорим на практике. Я вообще буду иногда повторяться, потому что в кодинге это полезно.

Итак, конструкция ***if .. then*** выглядит так:

```
If Условие then
 Выполнить действие;
```

Если условие выполнено, то будет выполнено одно следующее действие. Если ты хочешь выполнить два действия, то должен заключить их в логические скобки ***Begin … end***. Например:

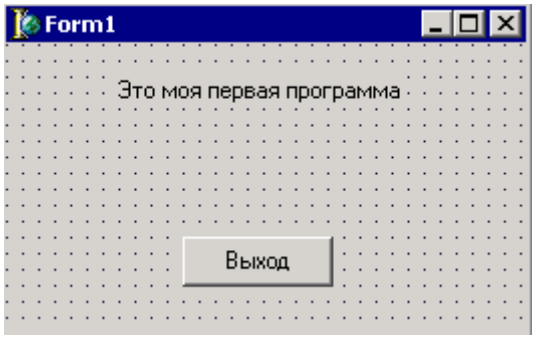
```
If Условие then
  Begin
    Действие1;
    Действие2;
    Действие3;
    ...
  End;
```

В этом случае все действия между ***Begin*** и ***end*** будут выполнены, если условие верно. Таким образом, ***Begin … end*** группирует последовательность действий в одно.

7.3 Надписи (TLabel).

Этим компонентом мы будем пользоваться практически в каждом примере, выводя надписи для других компонентов.

Создай новое приложение. Брось на форму один компонент TLabel  и измени у него свойство ***Caption*** на «Это моя первая программа». У тебя должно получится нечто подобное.



*На компакт диске, в директории **\Примеры\Глава 7\Label** ты можешь увидеть пример этой программы.*

Но это слишком просто. Давай я тебе покажу, как сделать с помощью этого компонента очень красивый визуальный эффект:

Щёлкни дважды по свойству *Font* компонента Label1. Перед тобой появится окно свойств шрифта. Сделай шрифт побольше и измени цвет на белый.

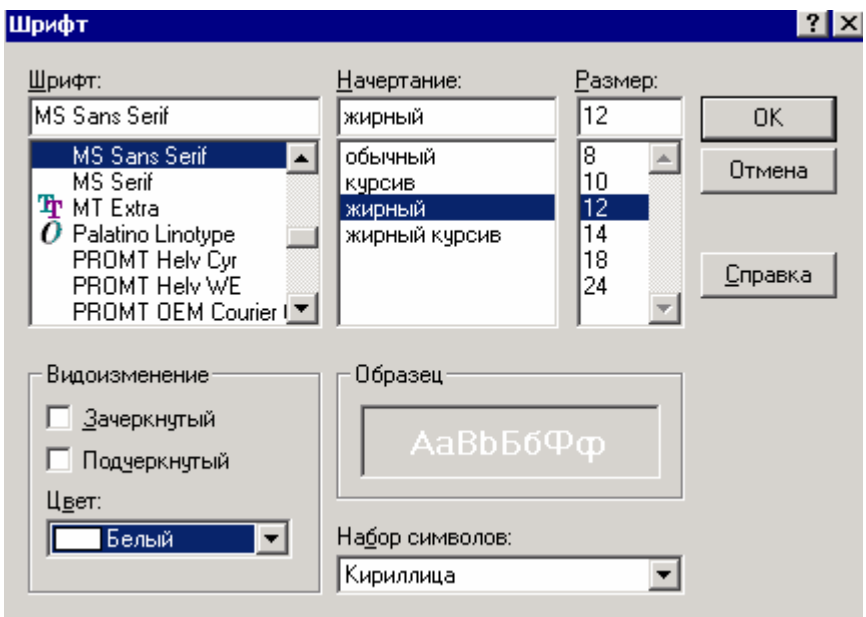


Рис 7.3.1 Изменение свойств шрифта

Теперь запомни значения свойств *Left* и *Top* компонента. У меня это 16 и 16. Теперь скопируй этот компонент в буфер обмена (меню Edit->Copy). Щёлкни по форме, чтобы снять выделение с компонента *Label1*. Теперь вставь копию компонента (меню Edit->Paste). Delphi создаст новый компонент *Label2*, который будет копией первого. Выдели новый компонент и измени свойства Left и Top на 18 и 18, чтобы второй компонент был как бы немного сдвинут поверх первого. Щёлкни дважды по свойству *Font* и измени цвет на синий. И наконец измени свойство *Transparent* (прозрачный) на *true*.

Если ты всё сделал правильно, то у тебя должно получится нечто похожее на рисунок 7.3.2

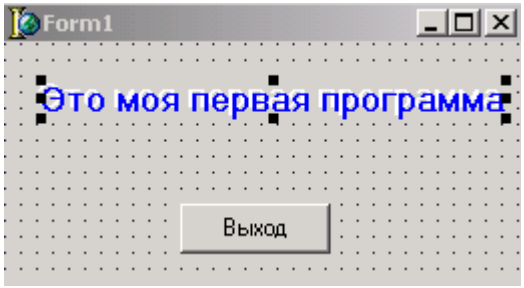


Рис 7..3.2 Результат

Получается как бы надпись с тенью. Симпотно? Я думаю, что да.

7.4 Строки ввода (TEdit).

С помощью строк ввода мы постоянно будем вводить различную информацию в наши программы. Давай попробуем написать несколько программ с использованием этого компонента, чтобы понять все тонкости работы с ним.

Создай новый проект. Брось на него два компонента *TEdit*  с палитры компонентов *Standard* и одну кнопку. У тебя должно получится нечто подобное:

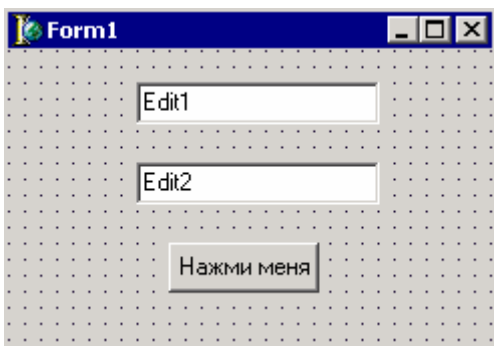


Рис 7.4.1 Форма

Давай очистим у обоих компонентов *TEdit* свойство *Text*. Это свойство отвечает за содержимое строки ввода. Мы просто очистим, чтобы после старта программы обе строки ввода были пустыми.

Теперь создай для кнопки обработчик события *OnClick*. Мы это уже делали, поэтому у тебя не должно быть с этим проблем. Кстати, если дважды щёлкнуть по кнопке, то Delphi автоматически создаст этот обработчик события. Ну а если он уже создан, то просто перенесёт тебя в то место, где написан код этого обработчика.

Итак, создай обработчик и напиши в нём следующее:

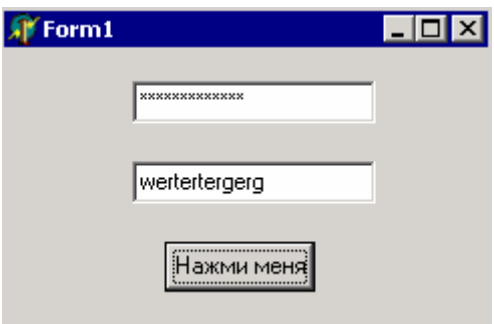
```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
 Edit2.Text:=Edit1.Text;
end;
```

По нажатию кнопки я копирую содержимое свойства *Text* компонента *Edit2* в свойство *Text* компонента *Edit1*.

Попробуй запустить программу. Теперь введи в первую строку ввода какой-нибудь текст, а потом нажми кнопку. Во второй строке ввода появится тот же текст.

На компакт диске, в директории ***\Примеры\Глава 7\TEdit*** *ты можешь увидеть пример этой программы.*

Давай немного улучшим пример. Теперь измени у первой строки ввода свойство *PasswordChar* на звёздочку «*». Теперь запусти программу и попробуй ввести в эту строку текст. Вместо текста будут появляться звёздочки, как при вводе пароля в какой-нибудь программе:



Именно таким образом делаются строки ввода паролей. Попробуй нажать на кнопку и во вторую строку ввода перенесётся текст, который ты вводил.

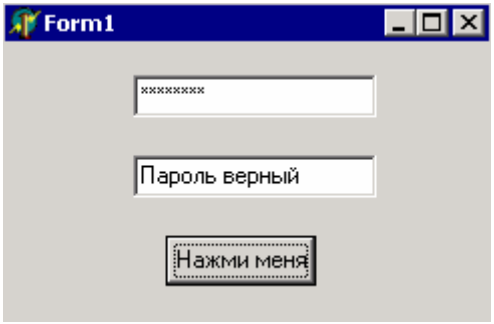
На компакт диске, в директории ***\Примеры\Глава 7\PasswordChar*** *ты можешь увидеть пример этой программы.*

Давай теперь сделаем проверку на ввод пароля. Найди в процедуру обработчик события *OnClick* для кнопки и напиши там следующее:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
 if Edit1.Text='password' then
  Edit2.Text:='Пароль верный'
 else
  Edit2.Text:='Неверно';
end;
```

Попробуй запустить программу. Если ты введёшь в первую строку ввода слово *password* и нажмёшь кнопку, то во второй строке появится надпись «*Пароль верный*», иначе будет надпись «Пароль неверный».

На компакт диске, в директории ***\Примеры\Глава 7\PasswordChar1*** *ты можешь увидеть пример этой программы.*

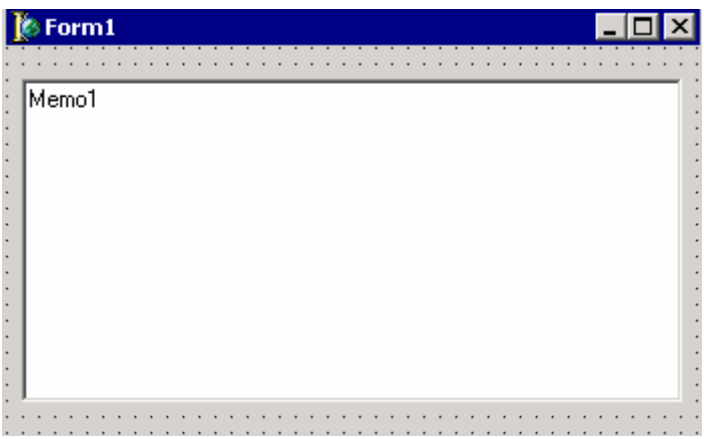


Этих знаний достаточно для написания практически любой программы со строкой ввода. Мы будем регулярно сталкиваться с этим компонентом, поэтому ты ещё успеешь познакомиться с ним поближе.

7.5 Многострочное поле ввода (TMemo).

Теперь мы познакомимся с многострочными компонентами ввода инфы. Для этого на закладке *Standard* есть компонент *TMemo*.

Создай новый проект. Установи на форму компонент *TMemo*  .

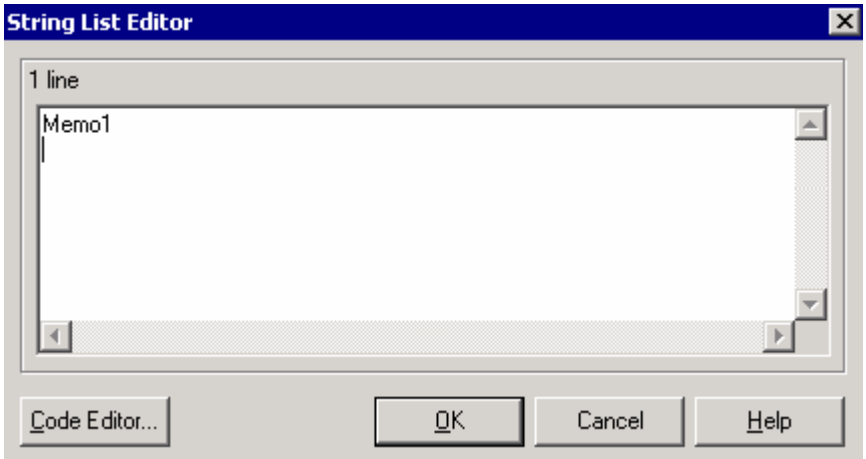


7.5.1 Форма с компонентом TMemo

По умолчанию, в нём уже присутствует одна строка текста равная имени компонента. За содержимое текста отвечает свойство *Lines*. Это свойство целый объект типа *TStrings*, и имеет свои свойства и методы. Немного позже мы познакомимся с некоторыми из них.

Давай сначала просто очистим содержимое компонента *Memo1*. Для этого дважды щёлкни по свойству *Lines*. Перед тобой откроется окно редактора строк (смотри рисунок 7.5.2). Это окно содержит простенький текстовый редактор, в котором можно набрать многострочный текст. Мы не будем этого делать, а просто удалим всё содержимое. Как только сделаешь это, нажми кнопку «ОК».

Теперь давай напишем небольшую программу, которая будет выполнять простые функции текстового редактора. Сложные вещи не получаться, потому что *TMemo* – это простой компонент. Для создания более сложных текстовых редакторов есть другой компонент.



7.5.2 Редактор строк.

Итак, добавь на форму пока только одну кнопку. Измени её свойство *Caption* на «*Очистить*» и имя на *ClearButton*. Кстати, давай изменим имя и компонента *Memo1* на *MainMemo*. Создай для кнопки обработчик события *OnClick*. В нём напиши следующее:

```
procedure TForm1.ClearButtonClick(Sender: TObject);
begin
 MainMemo.Lines.Clear;
end;
```

Здесь я вызываю метод *Clear* объекта *Lines*, который в свою очередь принадлежит объекту *MainMemo*. Немного запутано, но со временем ты поймёшь, что это очень даже удобно. У *MainMemo* есть свойство *Lines*, значит, чтобы получить к нему доступ мы должны написать *MainMemo.Lines*. У объекта *Lines* есть метод *Clear*, который очищает содержимое линий. Вот так и получается эта конструкция.

Попробуй запустить программу и написать внутри компонента *Memo* какой-нибудь текст. Потом нажми кнопку очистить, чтобы уничтожить всё, что ты ввёл.

На компакт диске, в директории ***\Примеры\Глава 7\Memo*** *ты можешь увидеть пример этой программы.*

Давай теперь усложним наш пример, добавив возможность сохранения введённого текста и загрузки его обратно.

Создай обработчик события *OnShow* для формы и напиши там следующее:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 MainMemo.Lines.LoadFromFile('memo.txt');
end;
```

Здесь я вызываю метод *LoadFromFile* объекта *Lines*. Ему нужно передать только один параметр – имя файла, откуда будет происходить загрузка данных. Есть только один недостаток – если ты сейчас попытаешься запустить программу, то во время загрузки произойдёт ошибка, потому что файла *memo.txt* нет. Тут есть два выхода:

1. Заранее создать этот файл.

2. При старте проверять, есть ли файл, и только если он существует производить загрузку текста.

С первым способом всё понятно. А вот для реализации второго способа нужно воспользоваться функцией *FileExists*, Ей нужно передать имя файла, которое надо проверить, и если такой файл существует, то она вернёт true. Так что давай изменим обработчик события *OnShow* следующим образом:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 if FileExists('memo.txt') then
  MainMemo.Lines.LoadFromFile('memo.txt');
end;
```

Теперь создадим обработчик события *OnClose*. В нём напишем процедуру сохранения содержимого *MemoMemo*.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 MainMemo.Lines.SaveToFile('memo.txt');
end;
```

Здесь используется метод *SaveToFile*, который работает так же, как и LoadFromFile, только этот сохраняет данные.

*На компакт диске, в директории **\Примеры\Глава 7\Memo1** ты можешь увидеть пример этой программы. Не запускай пример с привода CD-ROM, потому что при выходе из программы происходит попытка сохранить имеющиеся данные. А так как на CD-ROM записать невозможно, произойдёт ошибка.*

Теперь нам осталось только научится программно добавлять, удалять и изменять строки в компоненте *TMemo* и можно считать, что мы досконально изучили этот компонент.

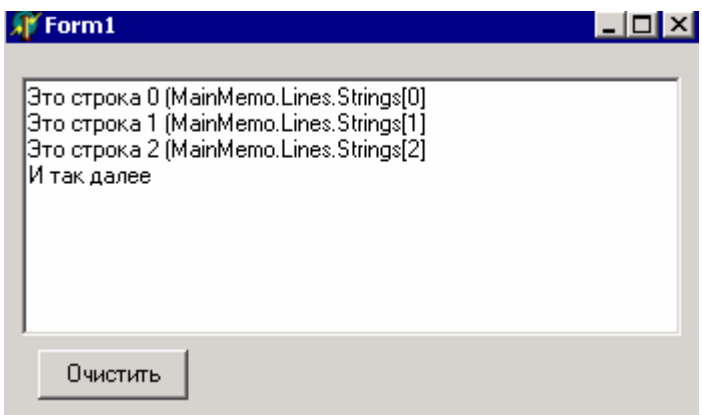


Рис 7.5.3. Хранение строк в TMemo

В компоненте *TMemo* строки хранятся как простая последовательность строк. Я набрал в прошлом примере текст, и после закрытия программы посмотрел, как он выглядит в файле. Вот содержимое memo.txt:

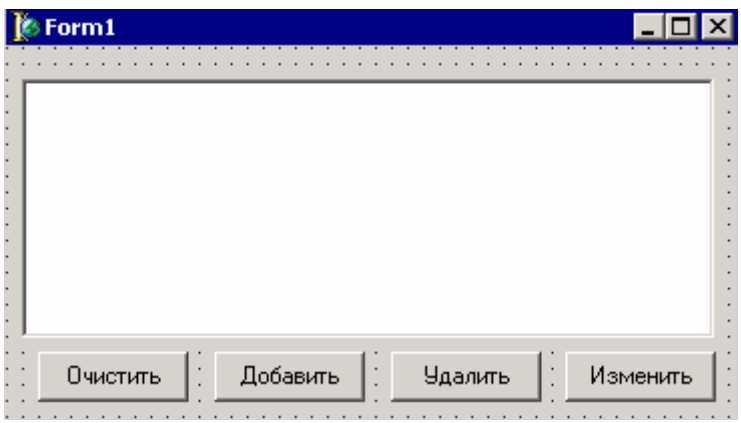
Как видишь, в файле четыре строчки (вторая пустая). Точно так же строки расположены и в памяти.

Для доступа к каждой строке можно воспользоваться свойством Strings объекта *Lines*. На рисунке 7.5.3 наглядно показано, как получить доступ к строкам. Чтобы получить нулевую строку, нужно написать *MainMemo.Lines.Strings[0]*, для первой строки *MainMemo.Lines.Strings[1]*, и так далее.

Давай напишем пример, который будет получать доступ к строкам, чтобы увидеть всё это на практике. Добавь к предыдущему примеру три кнопки:

1. Добавить. Я дал имя этой кнопке - *AddButton*.
2. Удалить. Я дал имя этой кнопке – *DelButton*.
3. Изменить. Я дал имя этой кнопке – *ChangeButton*.

Можешь их расположить следующим образом:



Теперь создадим обработчик события *OnClick* для кнопки «Добавить». Здесь мы будем программно добавлять новую строку в *MainMemo*. Для этого у объекта *Lines* есть метод *Add*, у которого есть только один параметр – текст новой строки:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 MainMemo.Lines.Add(' Новая строка');
end;
```

Теперь создадим обработчик события *OnClick* для кнопки удалить. По этому событию мы должны удалить строку, в которой находится сейчас курсор. Для этого напиши в обработчике следующий текст:

```
procedure TForm1.DelButtonClick(Sender: TObject);
begin
 if MainMemo.Lines.Count<>0 then
  MainMemo.Lines.Delete(MainMemo.CaretPos.Y);
end;
```

В первой строке я здесь проверяю, сколько строк в компоненте *MainMemo*. Для этого есть свойство *Count* объекта *Lines*. Если оно не равно нулю (*MainMemo.Lines.Count<>0)*, значит, строки есть, и мы можем удалить текущую строку. Знак <> означает «не равно».

Для удаления я использую метод *Delete* объекта *Lines*. В качестве единственного параметра нужно передать номер строки для удаления. Но как узнать, какая строка сейчас является текущей? Для этого у *MainMemo* есть свойство *CaretPos,* которое указывает на текущую позицию курсора.

CaretPos – это переменная типа *TPoint*. Этот тип переменной – запись. С таким типом мы подробно познакомимся немного позже. Единственное, что сейчас необходимо знать, так это то, что этот тип похож на объект, только у него нет методов, а только свойства. У *TPoint* есть два свойства «X» и «Y». X – указывает на текущую колонку, а Y указывает на текущую строку. Таким образом, я могу узнать текущую строку с помощью *MainMemo.CaretPos.Y*.

Итак *MainMemo.Lines.Delete* удаляет указанную строку. Я указываю текущую строчку с помощью *MainMemo.CaretPos.Y*. Я думаю, что задача выполнена.

Нам осталось только написать обработчик события *OnClick* для кнопки «Изменить». В нём напиши следующее:

```
procedure TForm1.ChangeButtonClick(Sender: TObject);
begin
 MainMemo.Lines.Strings[MainMemo.CaretPos.Y]:='Horrific';
 MainMemo.Lines.Strings[0]:='Текст изменён';
end;
```

В первой строке я изменяю текущую строку на *'Horrific'*. Второй строкой кода я изменяю первую строчку *ManMemo* на *'Текст изменён'*.

Здесь тебе уже всё должно быть знакомым. Так что можно считать, что мы изучили основные возможности компонента *TMemo*.

*На компакт диске, в директории **\Примеры\Глава 7\Memo2** ты можешь увидеть пример этой программы. Не запускай пример с привода CD-ROM, потому что при выходе из программы происходит попытка сохранить имеющиеся данные. А так как на CD-ROM записать невозможно, произойдёт ошибка.*

7.6 Объект TStrings.

В предыдущей части я познакомил тебя на практике с объектом *TStrings*. Свойство *Lines* компонента *TMemo* имеет такой тип. Это очень сильный объект, с которым мы будем очень часто встречаться на протяжении всей книги, поэтому я решил здесь остановиться и рассказать о нём подробнее.

Объект *TStrings* это набор строк. Везде, где информация поделена на строки этот объект является мощнейшим средством для хранения и работы со строками. Представь себе простой текстовый файл. Как хорошо, когда с ним можно работать именно разбив содержимое на строки, а не со всей сплошной информацией. Когда я буду рассказывать о работе с файлами, то этот объект тоже будет присутствовать. Я думаю, что ты ещё не раз будешь возвращаться к этой главе, чтобы вспомнить, для чего предназначено то, или иное свойство или метод.

Здесь я дам только описание свойств и методов объекта и начнём мы конечно же со свойств.

Свойства объекта TStrings:

Count – это свойство, которое ты можешь только читать. Здесь храниться количество строк, содержащихся в объекте.

Strings – здесь храниться сам набор строк. К любой строке ты можешь получить доступ, написав такую конструкцию:

```
Переменная:=Имя Объекта.Strings[Номер строки];
Имя Объекта.Strings[Номер строки]:= Переменная;
```

Первая строка кода запишет в переменную содержимое указанной строки. Вторая строка наоборот запишет содержимое переменной в указанную строку. Запомни, что строки в этом объекте нумеруются с нуля.

Text – в этом свойстве хранятся все строки в виде одной целой строки.

Методы объекта TStrings:

Add(Строка) – этот метод добавляет строку, указанную в качестве параметра в конец набора строк объекта. Этот метод возвращает номер, под которым добавлена эта строка.

Append(Строка) – этот метод тоже добавляет строку, указанную в качестве параметра в конец набора строк объекта. Метод ничего не возвращает.

AddStrings(Набор строк типа TStrings) – этот метод добавляет все строки из другого объекта типа *TStrings*.

Assign – этот метод присваивает вместо своего набора строк, указанный в качестве параметра новый набор.

Clear – удалить все строки из объекта.

Delete(номер строки) – удалить строку под указанным номером.

Equals(Набор строк типа TStrings) – сравнить собственный набор строк с указанным в качестве параметра. Если наборы равны, то метод вернёт *true* иначе вернёт *false*.

Exchange(Номер1, Номер2) – поменять местами строки указанных номеров.

Get(номер строки) – метод возвращает строку указанного номера.

IndexOf(Строка) – найти указанную в качестве параметра строку. Если такая строка существует в наборе, то метод вернёт её индекс, иначе –1.

Insert(Номер, Строка) – вставить в набор новую строку под указанным номером.

LoadFromFile(Имя файла) – загрузить набор строк из указанного текстового файла.

SaveToFile(Имя файла) – сохранить набор строк в указанный текстовый файл.

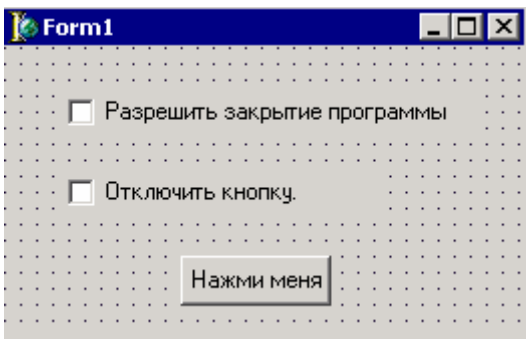
Move(Номер1, Номер2) – переместить строку под номером 1, но место номера 2.

Пока этих свойств и методов больше чем достаточно. В этой книге будут рассматриваться в основном эти методы. Я не описывал всё, и те возможности, которыми ты не будешь пользоваться (знаю по своему опыту) я описывать не стал.

7.7 CheckBox.

Теперь мы переходим к рассмотрению компонента *CheckBox*. Как всегда, давай создадим маленькую программу, которая будет использовать этот компонент.

Создай новое приложение, брось на него одну кнопку и два компонента *TCheckBox*  . Первому компоненту дай заголовок (Caption) равным «*Разрешить закрытие программы*» и имя (Name) равным «AllowCloseCheckBox». Второму компоненту дай заголовок (Caption) равным «*Отключить кнопку*» и имя (Name) равным «EnableButtonCheckBox». Кнопке я дал имя «*MyFirstButton*».



Создай обработчик события *OnClick* для компонента *EnableButtonCheckBox* (это второй *CheckBox*). В нём напиши следующее:

```
procedure TForm1.EnableButtonCheckBoxClick(Sender: TObject);
begin
 MyFirstButton.Enabled:=not EnableButtonCheckBox.Checked;
end;
```

Здесь я присваиваю свойству Enabled нашей кнопки значение **not** *EnableButtonCheckBox.Checked*. Что это значит? Свойство *Checked* компонента *EnableButtonCheckBox* показывает: стоит ли галочка на это *CheckBox*-е. Если да, то свойство *Checked* равно *True*, иначе *False*. Оператор *not* меняет это состояние на противоположное. Это значит, что если свойство *Checked* было равно *True*, то в *MyFirstButton.Enabled* будет присвоено противоположное (*False*).

Можешь попробовать запустить пример, и посмотреть что происходит. Когда ты ставишь галочку напротив «*Отключить кнопку*», свойство *Checked* этого компонента меняется на *True*. Срабатывает событие *OnClick* и в свойство *Enabled* кнопки присваивается значение свойства *Checked* компонента *CheckBox*, изменённое на противоположное, т.е. *False*. А когда свойство *Enabled* кнопки равно *False* она становится недоступной.

Чтобы окончательно разобраться с работой примера понажимай на *EnableButtonCheckBox* при запущенной программе. Потом попробуй убрать из исходного кода оператор ***not*** и снова запусти программу.

Теперь давай создадим обработчик *OnClick* для кнопки. В нём напиши следующее:

```
procedure TForm1.MyFirstButtonClick(Sender: TObject);
begin
 if AllowCloseCheckBox.Checked then
  Close;
end;
```

Здесь я проверяю, если свойство *Checked* компонента *AllowCloseCheckBox* (первый *CheckBox* на форме) равно *True*, то закрыть программу (выполнить метод *Close*). Иначе ничего не произойдёт.

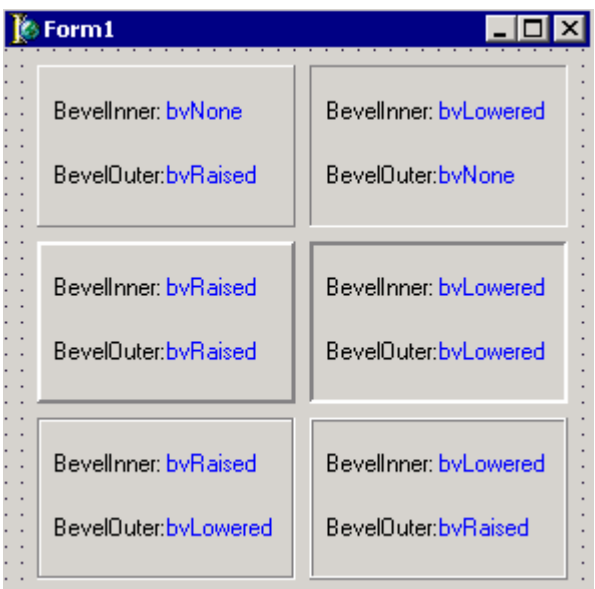
В принципе, это всё, что нужно знать и как можно использовать компонент *TConboBox*.Он достаточно простой, но в большинстве программ незаменим.

7.8 Панели (TPanel).

Вообще-то, если идти по порядку, то сейчас должен был быть *TRadioBox*. Но я решил немного перескочить сразу на *TPanel*. Потому что я начну её использовать уже в следующих примерах.

TPanel это компонент в виде панели. Он ведёт себя, так же как и форма. Ты можешь на нём располагать компоненты, и если ты передвинешь панель, то все компоненты установленные на ней тоже передвинутся.

Панель может выглядеть по разному. За внешний вид отвечают два свойства: *BevelInner* и *BevelOuter*. Я написал маленькую программу, которая ничего не делает, зато она показывает, как может выглядеть панель с различными вариантами установленных параметров. На панели синим шрифтом написаны установленные значения *BevelInner* и *BevelOuter*:



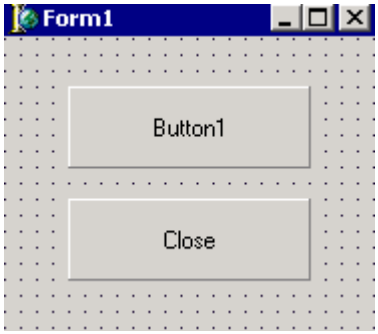
*На компакт диске, в директории **\Примеры\Глава 7\Panel** ты можешь увидеть пример этой программы.*

Давай напишем пример, в котором будем программно менять внешний вид панели. Для этого создай новое приложение и установи на форму два компонента *TPanel* с палитры компонентов *Standard* .

В этом примере я не буду менять имена панелей и оставлю их по умолчанию *Panel1* и *Panel2*. Не знаю почему, но я так захотел.

Единственное, что мы поменяем – это свойства *Caption* обеих панелей. У первой я написал «*Button1*», а у второй – «*Close*».

На рисунке ниже ты можешь увидеть форму будущей программы:



Теперь создадим обработчик события *OnMouseDown* для первой панели.

```
procedure TForm1.Panel1MouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 Panel1.BevelOuter:=bvLowered;
end;
```

Одна строчка кода меняет вид панели. Создадим ещё обработчик события *OnMouseUp* для первой панели. По этому событию мы меняем вид панели на исходный:

```
procedure TForm1.Panel1MouseUp(Sender: TObject;
  Button: TMouseButton;  Shift: TShiftState; X, Y: Integer);
begin
 Panel1.BevelOuter:=bvRaised;
end;
```

Попробуй запустить программу и нажать на первую панель. Когда ты нажмёшь мышь, панель изменит вид на вогнутый. При отпускании мыши панель возвращает вид на исходный. Таким образом, панель начинает работать как кнопка.

Раз так, давай создадим экзотичную кнопку. Для второй панели изменим свойства:

- ***BevelOuter*** на *bvRaised*
- ***BevelInner*** на *bvLowered*.

Теперь создадим обработчик события *OnMouseDown* для второй панели.

```
procedure TForm1.Panel2MouseDown(Sender: TObject;
   Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 Panel2.BevelOuter:=bvLowered;
 Panel2.BevelInner:=bvRaised;
end;
```

Здесь меняется вид панели на вогнутый. Теперь создадим обработчик события *OnMouseUp* для второй панели. По этому событию мы меняем вид панели на исходный:

```
procedure TForm1.Panel2MouseUp(Sender: TObject;
```

```
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
 Panel2.BevelOuter:=bvRaised;
 Panel2.BevelInner:=bvLowered;
 Close;
end;
```

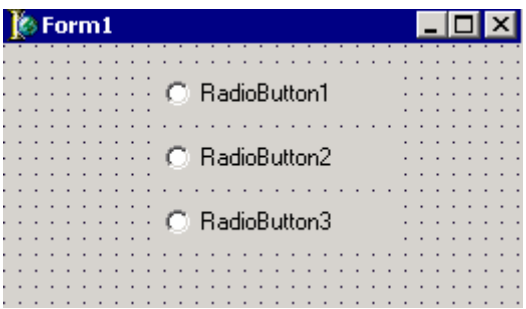
---

*На компакт диске, в директории **\Примеры\Глава 7\Panel1** ты можешь увидеть пример этой программы.*

7.9 Кнопки выбора TRadioButton.

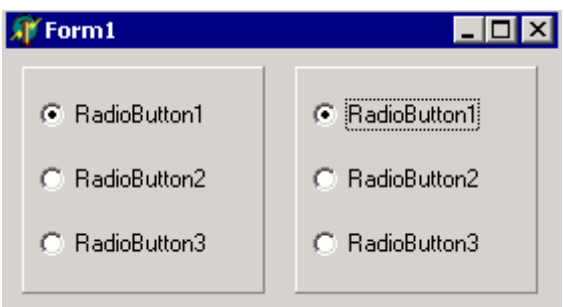
Эти кнопки очень похожи на *TCheckBox* даже по методу работы. У них так же есть свойство *Checked*, которое отображает её состояние. Если *RadioButton* выделен, то это свойство равно *True*, иначе равно *False*. Единственная разница – если у тебя на форме есть несколько таких компонентов, то одновременно может быть выделен только один.

Давай посмотрим это на практике. Брось на форму несколько компонентов *RadioButton* . Теперь запусти программу и попробуй пощёлкать.



*На компакт диске, в директории **\Примеры\Глава 7\RadioButton** ты можешь увидеть пример этой программы.*

Как видишь, ты не можешь выделить сразу два компонента *RadioButton*. А как же тогда сделать возможность двойного выбора на форме? Для этого компоненты *RadioButton* можно убрать на панели:



*На компакт диске, в директории **\Примеры\Глава 7\RadioButton1** ты можешь увидеть пример этой программы.*

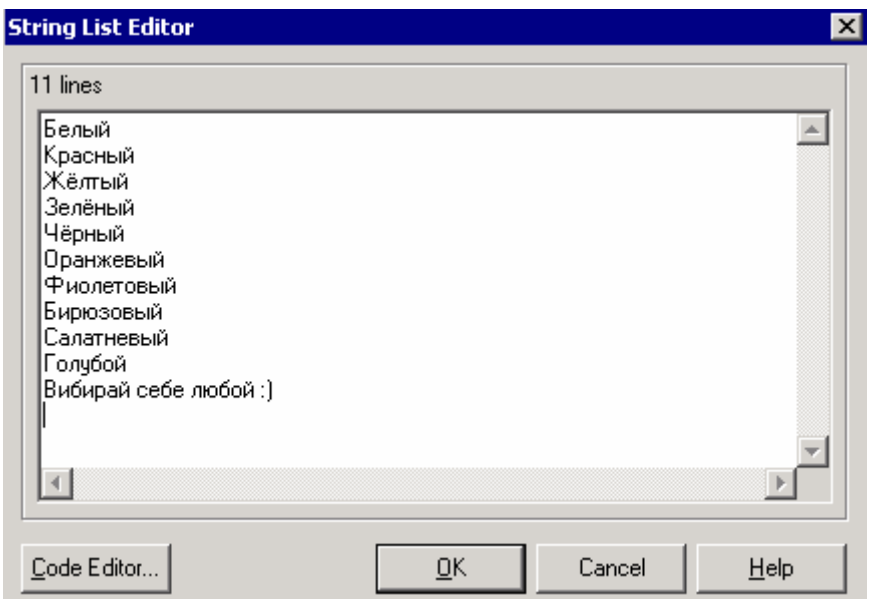
Больше ничего нового я не могу сказать. Как я уже сказал, работа с этим компонентам происходит так же, как и с *CheckBox*.

7.10 Списки выбора (TListBox).

Списки выбора хранят в себе какие-то списки (например, параметров) и дают пользователям возможность выбирать один или несколько параметров.

В работе списки достаточно просты. Чтобы получить доступ к строкам списка нужно воспользоваться свойством *Items* объекта *TListBox*. Это свойство имеет тип *TStrings*. Это ничего тебе не напоминает? Такой же тип у свойства *Lines* объекта *TMemo*. Значит работа со строками списка нам уже известна и не вызовет затруднений, потому что всё, что мы говорили про методы и свойства *Lines* объекта *TMemo* так же относится и к свойству *Items* объекта *TListBox*.

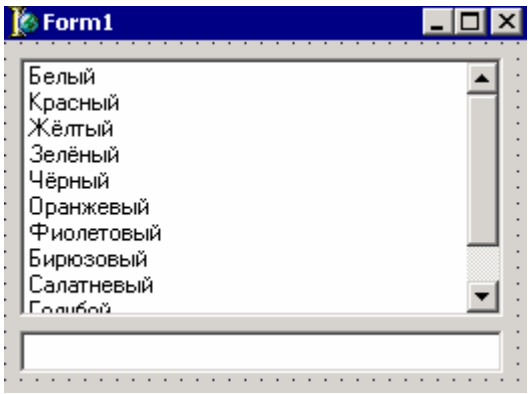
Давай напишем маленькое приложение с использованием этого компонента. Создай новый проект. Брось на форму один компонент *TListBox*  и один компонент TEdit. Теперь дважды щёлкни по свойству Items компонента *ListBox1*. Перед тобой откроется редактор строк:



Я набрал здесь названия всех основных цветов:

Белый
Красный
Жёлтый
Зелёный
Чёрный
Оранжевый
Фиолетовый
Бирюзовый
Салатневый
Голубой
Вибирай себе любой :)

После этого жми «ОК», чтобы сохранить введённые данные. У тебя должна получиться такая форма:



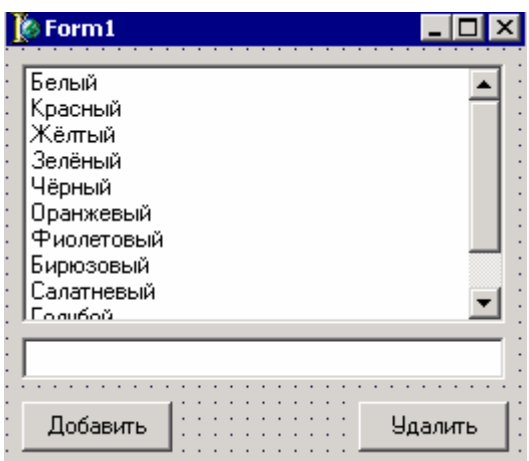
Давай теперь создадим обработчик события *OnClick* для списка выбора. В нём напишем следующее:

```
procedure TForm1.ListBox1Click(Sender: TObject);
begin
 Edit1.Text:=ListBox1.Items.Strings[ListBox1.ItemIndex];
end;
```

Свойство *ItemIndex* объекта *ListBox1* указывает на выделенную строку списка выбора. С помощью *ListBox1.Items.Strings* мы можем получить доступ ко всем строкам списка. В результате получается, что я присваиваю в *Edit1* текст выделенной строки в списке выбора.

На компакт диске, в директории ***\Примеры\Глава 7\ListBox*** *ты можешь увидеть пример этой программы.*

Всё очень похоже на работу с *TMemo*. Для большей ясности, давай добавим к нашему приложению ещё кнопку «Добавить» и «Удалить» строку.



Для кнопки добавить, в обработчике события *OnClick* напишем следующий код:

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

```
 ListBox1.Items.Add('Новая строка')
end;
```

Для кнопки удалить напишем такой текст:

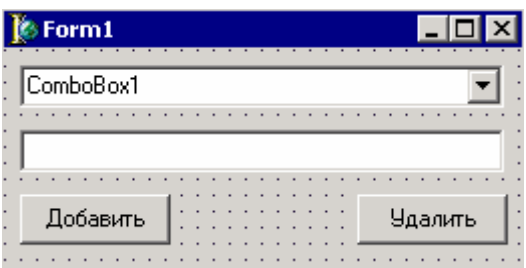
```
procedure TForm1.DelButtonClick(Sender: TObject);
begin
 ListBox1.Items.Delete(ListBox1.ItemIndex);
end;
```

Я думаю, что комментарии излишни. Полная аналогия с ***TMemo***.

7.11 Выпадающие списки (TComboBox).

Выпадающие списки по своей работе, свойствам и методам похожи на списки выбора. Я бы сказал, что это полная копия.

Давай создадим приложение похожее на предыдущее, только вместо *ListBox* будет *ComboBox*.



Теперь создадим обработчик события *OnChenge* для выпадающего списка *ComboBox1*. Это событие происходит, когда пользователь выбрал какой-нибудь элемент списка. По нему мы напишем следующий текст:

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
 Edit1.Text:=ComboBox1.Items.Strings[ComboBox1.ItemIndex];
end;
```

Как видишь, это вообще полная копия кода из предыдущего примера, только используется другое имя компонента.

Теперь напишем код для кнопки «Добавить»:

```
procedure TForm1.AddButtonClick(Sender: TObject);
begin
 ComboBox1.Items.Add('Новая строка')
end;
```

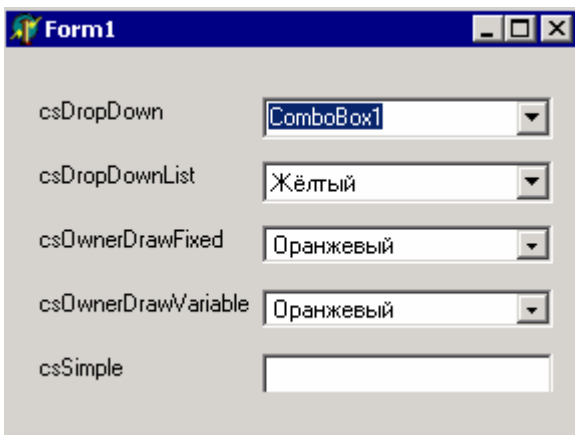
Для кнопки удалить код будет таким:

```
procedure TForm1.DelButtonClick(Sender: TObject);
begin
 ComboBox1.Items.Delete(ComboBox1.ItemIndex);
end;
```

На компакт диске, в директории ***\Примеры\Глава 7\ComboBox*** *ты можешь увидеть пример этой программы.*

Как видишь, наблюдается полная аналогия со списком выбора и *TMemo*. Содержимое списков можно сохранять с помощью *ComboBox1.Items.SaveToFile('Имя файла')*, а загружать с помощью *ComboBox1.Items.LoadFromFile('Имя файла')*.

Существует несколько типов выпадающих списков. За тип списка отвечает свойство *Style*. Я написал маленькую программу, которую ты можешь найти на диске в ***\Примеры\Глава 7\ComboBox1***, которая показывает все типы списков в действии.



Этот пример не может показать все разницы стилей, поэтому я дам ещё небольшое описание:

CsDropDown – основной стиль. При нём ты можешь не только выбирать значения из списка, но и вводить в строку свои.

CsDropDownList – при этом стиле можно только выбирать из списка.

CsOwnerDrawFixed – при этом стиле ты можешь рисовать элементы сам. Высота элементов фиксированная.

CsOwnerDrawVariable - при этом стиле ты можешь рисовать элементы сам. Отличается от предыдущего тем, что высота элементов не фиксированная.

CsSimple – только строка ввода.

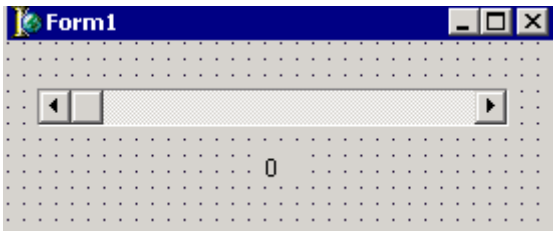
Немного позже в этой книге я покажу тебе как рисовать внутри списков выбора и выпадающих списков.

7.12 Полосы прокрутки (TScrollBar).

Полосы прокрутки очень часто используются для прокручивания какого-то действия. Например, когда ты слушаешь музыку, ты можешь прокрутить её в любое место с помощью простой полосы прокрутки. Или если информация не помещается в окно, её так же прокручивают с помощью таких полосок, но в

большинстве случаев это делается автоматически, и тут ты вмешиваться будешь очень редко.

Давай посмотрим на полосу прокрутки в действии. Создай новое приложение. Брось на форму один компонент *TLabel* и одну полосу прокрутки *TScrolBar* . У тебя должна получится форма следующего вида:



У компонента *Label1* измени свойство *Caption* на «0». Теперь создай обработчик события *OnChange* для полосы прокрутки и напиши там следующее:

```
procedure TForm1.ScrollBar1Change(Sender: TObject);
begin
 Label1.Caption:=IntToStr(ScrollBar1.Position);
end;
```

В этом коде я присваиваю свойству *Caption* компонента *Label1* значение текущей позиции ползунка полосы прокрутки. Текущее значение ползунка можно получить с помощью свойства *Position* объекта *ScrollBar1*. Только тут есть одно «НО». Это свойство имеет тип целое число, а свойство *Caption* компонента *Label1* – это строка. Поэтому нам надо превратить целое число в строку. Для этого есть функция IntToStr. Ей нужно передать число, а она нам вернёт строку. Поэтому если вызвать эту функцию с параметром текущей позиции ползунка (*IntToStr(ScrollBar1.Position)*), результат её работы можно присвоить свойству *Caption* компонента *Label1*.

Попробуй запустить программу и подвигать ползунок. Значение позиции будет отображаться в компоненте *Label1*.

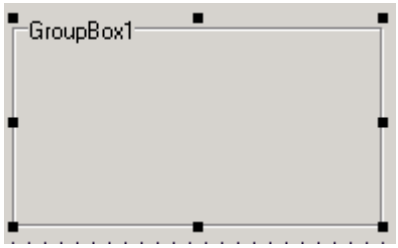
На компакт диске, в директории ***\Примеры\Глава 7\ScrollBar*** *ты можешь увидеть пример этой программы.*

В этом примере мы написали пример горизонтальной полосы прокрутки. Чтобы сделать её вертикальной, нужно свойство *Kind* поменять на *sbVertical*. И ещё, значение ползунка изменяется от 0 до 100. Чтобы изменить эти значения есть свойства *Min* (по умолчанию равно нулю) и *Max* (по умолчанию равно 100).

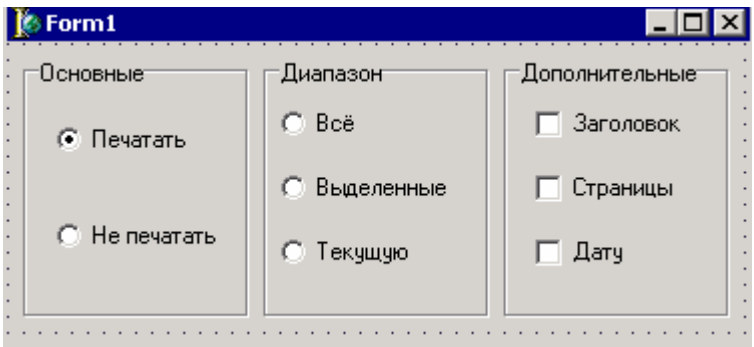
Больше ничего особенного в работе полос прокрутки нет. Поэтому давай двигаться дальше.

7.13 Группировка объектов (GroupBox).

Компонент *GropBox* очень удобно использовать для группировки каких-то компонентов. На вид это простая панель только с заголовком наверху:



За текст отображаемый в заголовке отвечает свойство *Caption*. Больше ничего особенного эта панель делать не умеет. Я написал маленькую программу, чтобы показать, как можно использовать компонент *TGroupBox*.



На компакт диске, в директории ***\Примеры\Глава 7\GroupBox*** *ты можешь увидеть пример этой программы.*

Панель *TGroupBox* в основном используют для группировки компонентов *TRadioButton*.

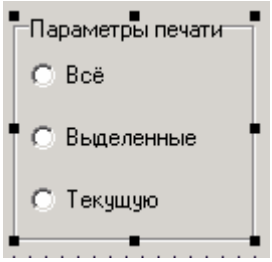
7.14 Группа компонентов RadioButton (TRadioGroup).

Если установить этот компонент на форму, то на первый взгляд он выглядит, как простой *TGroupBox*.

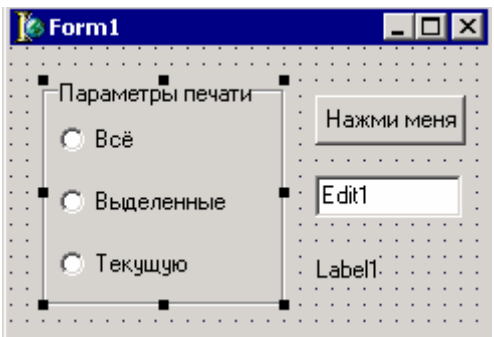
Создай новый проект и прось на него один компонент *TRadioBox*. Но это только на первый взгляд. Щёлкни по свойству *Items* и перед тобой появится уже знакомый редактор строк. Введи туда три строчки:

- Всё
- Выделенные
- Текущую

Нажми «ОК» и твой ты увидишь *GropBox* содержащий в себе три компонента *TRadioBox*. Зачем же нужен такой гибрид? Потерпи немного, пока мы не напишем пример до конца. Тогда ты сможешь оценить все прелести этого гибрида и возможно полюбишь его.



Брось на форму ещё одну кнопку, одну строку ввода и один *TLabel*. Расположи их так, как показано на скрине ниже:



Теперь создадим обработчик события *OnClick* для компонента *RadioGroup1*. В нём напишем следующее:

```
Label1.Caption:=IntToStr(RadioGroup1.ItemIndex);
```

Свойство *ItemIndex* компонента *RadioGroup1* показывает, какой компонент сейчас выделен. Компоненты пронумерованы в таком же порядке, как записаны их имена в списке. Это свойство имеет тип целое число, поэтому его приходится превращать в строку с помощью *IntToStr*.

По нажатию кнопки напишем следующий текст:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Edit1.Text:=IntToStr(RadioGroup1.ItemIndex);
end;
```

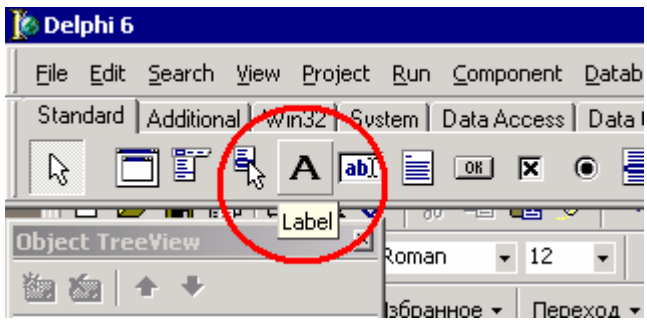
По нажатию кнопки я делаю то же самое, только номер выделенного компонента помещаю в *Edit1*.

А теперь посмотрим на преимущества этого компонента. Представим, что у нас просто стоит три компонента ***TRadioButton***. Чтобы узнать, какой из них сейчас выделен, нужно проверить свойство *Checked* всех этих компонентов. А при использовании группы ***TRadioGroup*** ничего этого не надо. Нужно только проверить свойство *ItemIndex*, компонента ***TRadioGroup*** и никаких проблем.

7.15 Ответы на вопросы.

Здесь я буду отвечать на вопросы, которые могут возникнуть у тебя при чтении этой главы.

Вопрос: Почему, когда я навожу мышкой на компонент, например ***TLabel*** выскакивает подсказка, в которой написано *Label*, а не ***TLabel***? Куда девается буква «Т»?



Ответ: действительно, в подсказках всегда отсутствует буква «Т». Просто компонент называется *Label*, а объект этого компонента называется *TLabel*. Так принято, что имена всех объектов всегда начинаются с буквы «Т». Это не значит, что так обязательно. Это значит, что так желательно. Просто взглянул на имя и видишь, что это имя объекта. А в подсказках показывают имя компонента, к которому нет такого соглашения, поэтому там нет никаких букв вначале.

Глава 8. Учимся программировать.

В этой главе я буду описывать основные принципы программирования. Мы познакомимся с циклами на практике, увидим различные приёмы кодинга, и всё это будет снабжено громадным количеством полезных примеров.

Я всегда говорил, что все знания приходят с практикой. Ты сможешь по настоящему что-то освоить только когда попробуешь что-нибудь сам. Поэтому я ещё раз прошу тебя повторять все описываемые мной действия самостоятельно. Все исходники, которые идут на диске, я привёл только для того случая, когда у тебя что-то не получается. Ты можешь посмотреть их содержимое, но после этого ты просто обязан повторить все действия самостоятельно. Иначе тебе никакая книга не поможет.

Итак, я приступаю к последней главе, в которой я буду описывать чисто теорию. После этого мы будем знакомиться уже только с разными приёмами и технологиями кодинга. И это последняя глава, в которой я закладываю основы кодинга.

8.1 Циклы (for..to..do).

Циклы – это основа любого кодинга. Мы будем использовать их практически в каждой программе. Мы уже познакомились с ними на практике. Напомню тебе, как выглядит логика цикла:

```
От 1 до 5 выполнять
  Начало цикла
   R:=R*INDEX;
   INDEX:=INDEX+1;
  Конец цикла
```

Это логика расчёта факториала. Давай эту логику перенесём на язык программирования Delphi.

Цикл в Delphi оформляется как:

```
for переменная:=начальное значение to конечное значение do
 Действие;
```

После слова for нужно присвоить какой-нибудь переменной начальное значение. Эта переменная будет использоваться в качестве счётчика. После каждого выполнения действия этот счётчик будет увеличиваться на единицу, пока переменная не превысит конечного значения.

В общем виде цикл выглядит так:

```
for..to..do
 Действие1
```

Рассмотрим пример:

```
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
  Sum:=0;
  for index:=0 to 5 do
    Sum:=Sum+index;
end;
```

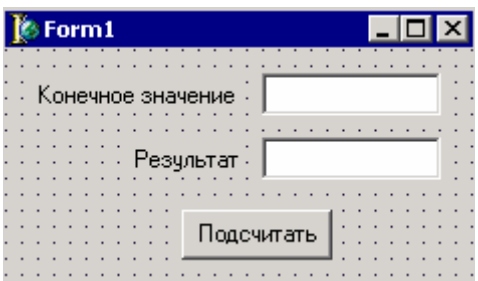
В этом примере я объявляю две переменные index и sum типа целое число. Сначала я присваиваю переменной Sum значение 0. После этого запускаю цикл в котором переменная index будет изменяться от 0 до 5.

Теперь посмотрим поэтапно, что здесь происходит:

1. На первом этапе переменная index равна нулю. Sum тоже равна нулю, значит выполнится операция Sum:=0+0. Результат Sum=0;
2. На втором этапе index увеличена на 1 значит, выполнится действие Sum:=0+1. Результат Sum=1.
3. Здесь index увеличена на 1 и уже равна 2, а Sum=1. Значит, выполнится действие Sum:=1+2. Результат Sum=3.
4. Здесь index увеличена на 1 и уже равна 3, а Sum=3. Значит, выполнится действие Sum:=3+3. Результат Sum=6.
5. Здесь index увеличена на 1 и уже равна 4, а Sum=6. Значит, выполнится действие Sum:=4+6. Результат Sum=10.
6. Здесь index увеличена на 1 и уже равна 5, а Sum=10. Значит, выполнится действие Sum:=5+10. Результат Sum=15.

Заметь, что мы не увеличиваем переменную index, она увеличивается автоматически.

Давай перенесём этот код в программу, чтобы мы могли убедиться в этом на реальном примере. Создай новое приложение. Брось на форму два компонента TLabel, два компонента TEdit и одну кнопку.



Компонент *Edit1* я переименовал в *EndEdit*, а *Edit2* я переименовал в *ResultEdit*. Теперь по нажатию кнопки я написал следующий код:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
 Sum:=0;

EndCount:=StrToInt(EndEdit.Text);

 for index:=0 to EndCount do
  Sum:=Sum+index;

 ResultEdit.Text:=IntToStr(Sum);
end;
```

В принципе, текст тот же самый. Единственная разница – я запускаю цикл, начиная от 0 до числа введённого в компонент EndEdit. EndEdit содержит текст, а мне нужно превратить его в число, поэтому я использую функцию StrToInt для преобразования строки в число. Эта функция работает так же, как и IntToStr, которая наоборот преобразовывала число в строку.

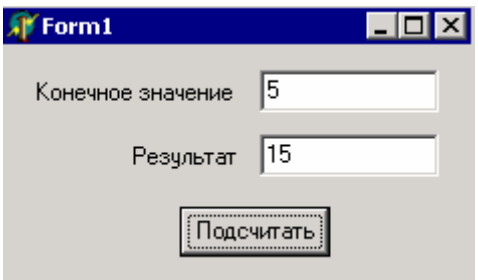
Результат преобразования я сохраняю в переменной EndCount:

```
EndCount:=StrToInt(EndEdit.Text);
```

После этого я уже запускаю цикл, в котором переменная index будет изменятся от 0 до значения EndCount (в котором находится число введённое в EndEdit).

```
for index:=0 to EndCount do
```

Запусти программу и введи в строку «*Конечное значение*» число 5. После этого нажми на кнопку и в строке результата должно появится число 15.



На компакт диске, в директории ***\Примеры\Глава 8\for..to..do*** *ты можешь увидеть пример этой программы.*

Хочу ещё отметить, что после цикла *for* будет выполняться только одно действие. Например, если ты захочешь выполнить два действия подряд, то ты должен заключить их в скобки **begin** и **end**:

```
for index:=0 to EndCount do
  begin
    Sum:=Sum+Index;
    Sum:=Sum+1;
  end;
```

В этом примере, на каждом шаге цикла я увеличиваю Sum ещё на единицу. Если ты попробуешь написать так:

```
for index:=0 to EndCount do
    Sum:=Sum+Index;
    Sum:=Sum+1;
```

то выполнятся в цикле будет только строчка Sum:=Sum+index. Вторая строка Sum:=Sum+1; выполнится только по окончанию расчёта цикла.

Если ты что-то не понял, вернись к главе, где я описывал блок-схемы. Там был описан пример, который работает как цикл *for..to..do*. Попробуй нарисовать блок схему для этого примера сам и мысленно пройти её шаги.

8.2 Циклы (while).

Ещё одной разновидностью цикла является цикл while. Это слово переводиться как «пока». Это значит, что цикл будет выполняться, пока выполняется условие.

В общем виде он выглядит следующим образом:

```
while условие do
 Действие
```

Сразу рассмотрим пример:

```
var
 index:integer;
begin
 index:=0;

 while index<10 do
  index:=index+1;
end;
```

В этом примере я объявляю переменную index. В первой строке кода я присваиваю ей 0. После этого я запускаю цикл. В условии я написал index<10, это значит, что будет выполняться следующее действие (index:=index+1) пока переменная index меньше 10.

Здесь так же выполняется только одно действие. Если ты захочешь выполнить в цикле сразу два действия, то ты должен заключить их в скобки *begin* и *end*.

Давай напишем предыдущий пример с использованием цикла while. Измени код по нажатию кнопки на следующий:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
 Sum:=0;
 index:=0;

 EndCount:=StrToInt(EndEdit.Text);

 while index<EndCount do
  begin
   Sum:=Sum+index;
   index:=index+1;
  end;

 ResultEdit.Text:=IntToStr(Sum);
end;
```

Тут мне надо обнулять не только переменную Sum, но и index, чтобы начальное значение было равно нулю, и цикл шёл от этого нуля до введённого значения. Обрати так же внимание на то, что здесь мне нужно самостоятельно увеличивать переменную index

(index:=index+1), для этого я эту строчку добавил в цикл. Она вместе с расчётом суммы объединены с помощью *begin* и *end*.

Попробуй запустить программу и ввести число 5. Результатом расчёта будет 10. Если ты помнишь предыдущий пример, то там результат был 15. В чём проблема? Почему разные результаты? В прошлом примере мы выполняли цикл от 0 до 5 включительно. Здесь будет выполняться цикл от 0 и до того момента, пока выполняется условие index<5. Когда index=5 условие не выполнится, и расчёт с цифрой 5 не будет производиться.

Для решения этой проблемы можно поменять условие цикла на index<=5 (переменная index меньше или равна EndCount). В этом случае расчёт с цифрой 5 будет считаться. Или можно вводить цифру 5.

 *На компакт диске, в директории* ***\Примеры\Глава 8\while*** *ты можешь увидеть пример этой программы.*

8.3 Циклы (Repeat).

Теперь давай разберём ещё один тип циклов repeat..until. Этот тип цикла похож на while. Даже смысл цикла похож. Он означает, что выполнять действия, пока не наступит определённое событие. Только тут есть пару отличий:

1. В цикле while действия выполнялись, пока условие верно. В цикле repeat действия будут выполняться, пока условие не верно и прекращается, когда оно станет верным.
2. В цикле while верность условия проверяется перед началом действий. Это значит, что если условие заведомо неверно, то действия цикла не будут выполнены. В цикле repeat сначала выполняется действие, а потом происходит проверка. Это значит, что если условие заведомо не верно, действие всё равно будет выполнено один раз.

В общем виде цикл repeat выглядит так:

```
repeat
  действия
until Условие;
```

Заметь, что в этом случае, действий может быть несколько. Тут уже не надо объединять несколько действий в *begin..end*, потому что repeat..until уже действует как объединение нескольких действий.

Давай рассмотрим прошлый пример с использованием этого типа цикла:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  index:Integer;
  sum:Integer;
  EndCount:Integer;
begin
 Sum:=0;
 index:=0;

 EndCount:=StrToInt(EndEdit.Text);
```

```
 repeat
  Sum:=Sum+index;
  index:=index+1;
 until index>EndCount;

 ResultEdit.Text:=IntToStr(Sum);
end;
```

В этом примере действия будут выполняться в цикле, пока переменная *index* не станет больше числа указанного в EndCount.

*На компакт диске, в директории **\Примеры\Глава 8\repeat** ты можешь увидеть пример этой программы.*

8.4 Управление циклами.

Работой цикла можно ещё и управлять. В Delphi есть два оператора управления циклами – **break** и **continue**. Начнём мы рассмотрение с оператора **break**.

Допустим, что тебе надо разделить число 10 на числа начиная от –3 до 3 и вывести результат в TListBox.

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r:Integer;
begin
 for i:=-3 to 3 do
  begin
   r:=round(10/i);
   ListBox1.Items.Add('10/'+IntToStr(i)+'='+IntToStr(r));
  end;
end;
```

В этом примере я запускаю цикл, в котором переменная **i** будет изменяться от –3 до 3. На каждом шаге я делю 10 на значение **i** и сохраняю результат в переменную **r**.

При делении используется функция round, которая округляет переданное ей значение. В качестве значения мы передаём результат деления 10 на переменную **i round(10/i).** Так что в переменную **r** будет записан округлённый до целого результат деления. Этого пока достаточно знать о функции **round**, потому что с ней мы познакомимся поближе немного позже.

После этого я добавляю результат в ListBox1, преобразовывая переменную **r** в строку.

Давай посмотрим, что произойдёт, когда переменная **i** будет равна 0. В этом случае число 10 будет делиться на 0, а значит, произойдёт ошибка, потому что на 0 делить нельзя. Как же тогда выйти из этой ситуации? Можно на каждом этапе цикла проверять значение **i**, и если оно равно 0, то не выполнять никаких действий. Вот два возможных решения:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r:Integer;
```

```
begin
 for i:=-3 to 3 do
  begin // Это начало для оператора for
   if i<>0 then
    begin // Это начало для оператора if
     r:=round(10/i);
     ListBox1.Items.Add('10/'+IntToStr(i)+'='+IntToStr(r));
    end;// Этот end для оператора if
  end; // Этот end для оператора for
end;
```

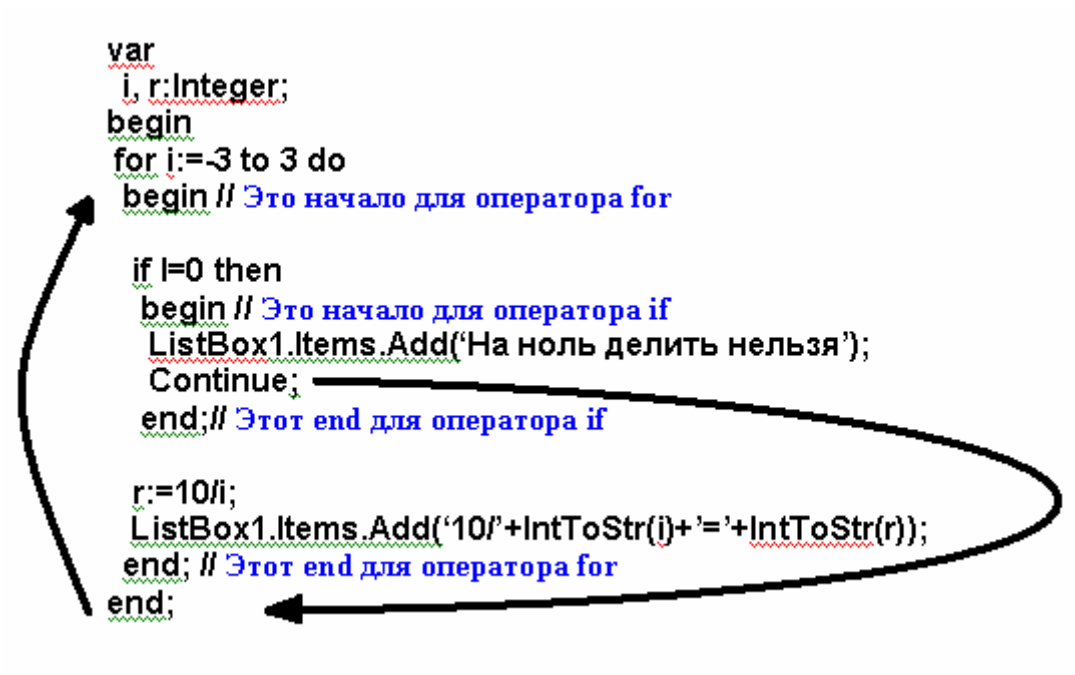
В этом примере я на каждом этапе проверяю значение **i**, и если оно не равно 0, то только в этом случае произвожу вычисления.

Это очень простое решение для маленьких и простых программ. Но если твой цикл большой и выполняет много действий, то такое решение будет как минимум неудобно и может потеряться читабельность кода. В худшем случае, вообще может ничего не получиться. Вот тут на помощь приходит оператор **continue**:

```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r:Integer;
begin
 for i:=-3 to 3 do
  begin // Это начало для оператора for

   if I=0 then
    begin // Это начало для оператора if
     ListBox1.Items.Add('На ноль делить нельзя');
     Continue;
    end;// Этот end для оператора if

   r:=round(10/i);
   ListBox1.Items.Add('10/'+IntToStr(i)+'='+IntToStr(r));
  end; // Этот end для оператора for
end;
```

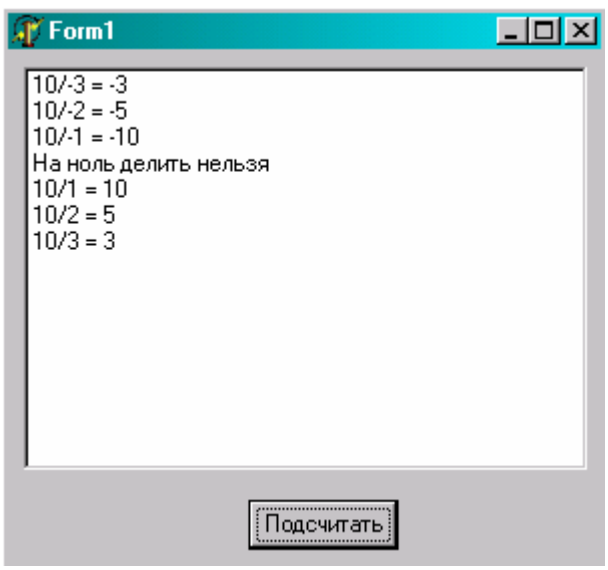
В этом примере я так же проверяю на каждом этапе значение переменной **i**. Если оно равно нулю, то я вывожу сообщение в *ListBox1* о том, что на ноль делить нельзя и выполняю оператор **continue**. Как только программа встречает такой оператор, то она сразу прерывает дальнейшее выполнение цикла и переходит на следующий шаг. Это то же самое, что выполнить команду: «Остановиться дальнейшее выполнение программы, увеличить значение переменной **i** и начать выполнение цикла со следующим значением».

Посмотри на следующий рисунок, где я показал стрелками процесс выполнения оператора **continue**:



Как только программа встречает оператор **continue**, она перескакивает на конец цикла, где увеличивается значение счётчика (в данном случае переменная **i**) и продолжается выполнение уже со следующего шага.

На следующем экране показана форма с результатом работы нашего примера:



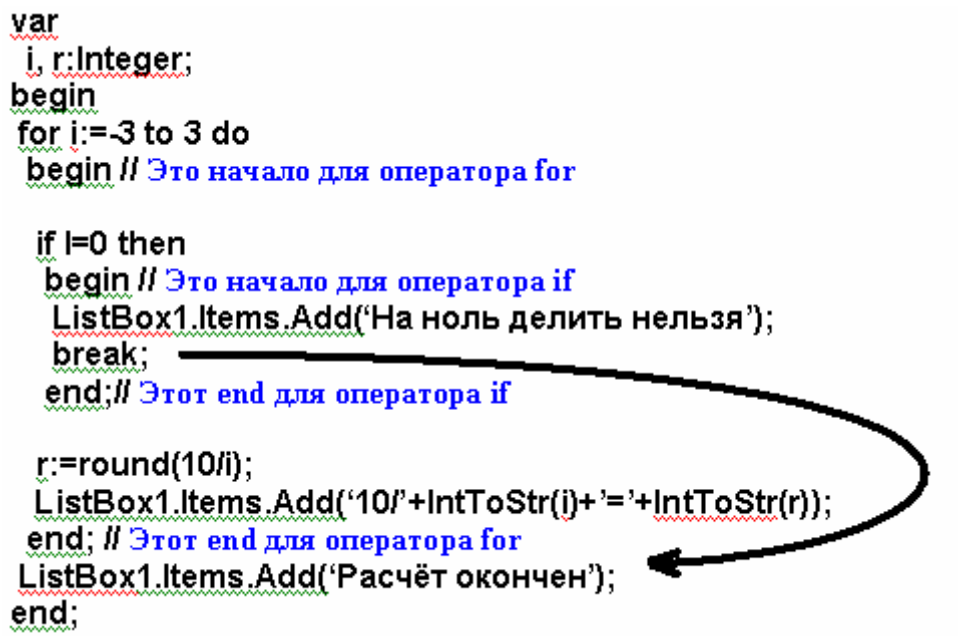
На компакт диске, в директории ***\Примеры\Глава 8\continue*** *ты можешь увидеть пример этой программы.*

Теперь давай разберёмся с оператором **break**. Как только программа встречает такой оператор, цикл прерывается и выполнение передаётся следующему оператору после цикла. Давай возьмём наш предыдущий пример и заменим в нём **continue** на **break**:

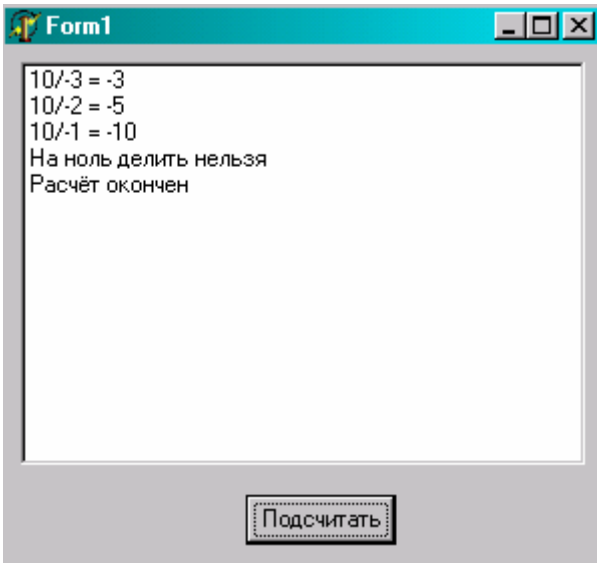
```
procedure TForm1.CalculateButtonClick(Sender: TObject);
var
  i, r:Integer;
begin
```

```
for i:=-3 to 3 do
 begin // Это начало для оператора for

  if I=0 then
   begin // Это начало для оператора if
    ListBox1.Items.Add('На ноль делить нельзя');
    break;
   end;// Этот end для оператора if

  r:=round(10/i);
  ListBox1.Items.Add('10/'+IntToStr(i)+'='+IntToStr(r));
 end; // Этот end для оператора for
ListBox1.Items.Add('Расчёт окончен');
end;
```

В этом случае, если **i** будет равно нулю, то выведется сообщение о том, что на ноль делить нельзя и цикл прервётся. После этого, управление передастся следующему оператору после цикла:



На следующем рисунке показан экран программы нашего кода:



Как видишь, после встречи в нулём, в цикле больше ничего не выполняется.

На компакт диске, в директории ***\Примеры\Глава 8\break*** *ты можешь увидеть пример этой программы.*

8.5 Логические операторы

С логическими операторами я тебя уже знакомил, и мы с ними уже достаточно много работали. Но здесь я дам более полную информацию по логическим операциям. Точнее сказать, по одной – **IF**.

Как ты уже знаешь, она выполняет проверку – « Если какое-то условие выполнено, то выполнить следующее за условием действие. Если нужно выполнить несколько действий, то их нужно объединить с помощью *begin…end*.

В общем виде логика выглядит так:

```
If Условие выполнено then
 Действие1;
```

Если нужно выполнить два действия, то нужно написать так:

```
If Условие выполнено then
 begin
   Действие1;
   Действие2;
 end;
```

А если надо проверить сразу два условия? В этом случае можно поступить двумя способами:

```
If Условие1 выполнено then
```

```
If Условие2 выполнено then
  Действие1;
```

Если условие один верно, то выполнится следующее за логикой действие, а это вторая проверка. Если вторая проверка условия верна, то выполнится действие. Если хотя бы одно из условий не выполнится, то цепочка прерывается, и действие не будет выполнено.

Есть ещё один способ:

```
If (Условие1 выполнено) and (Условие2 выполнено) then
  Действие1;
```

В этом примере я объединил две проверки в одну. Если **Условие1** и **Условие2** верны, то выполнится действие.

А если тебе нужно выполнить действие, если хотя бы одно из условий верно? Не обязательно, чтобы оба сразу, а хотя бы одно. В этом случае можно для объединения использовать не **and**, а **or**. Это будет выглядеть так:

```
If (Условие1 выполнено) or (Условие2 выполнено) then
  Действие1;
```

Если ты объединяешь два условия в один **if**, то их обязательно нужно оградить скобками. Если ты их не поставишь, то будет ошибка. Вот пример неправильного оформления:

```
If Условие1 выполнено or Условие2 выполнено then
  Действие1;
```

В качестве условий можно применять следующие операторы:

Оператор	Описание	Пример использования
<	Меньше	Index < 10
>	Больше	Index > 10
= =	Равно	Index = = 10
< =	Меньше либо равно	Index < = 10
> =	Больше либо равно	Index > = 10

Существует одно исключение, при котором оператор может отсутствовать. Если ты проверяешь булеву переменную, то оператор можно опустить. Например:

```
var
 b:Boolean;

begin
 b:=true;
```

```
if b then
 Выполнить действие;

end;
```

В этом примере, происходит проверка булевой переменной **b**. Но с чем её сравнивают, не указано. Как ты знаешь, булевы переменные могут принимать одно из двух значений: *true* или *false* (истина или ложь). Так вот, в таком случае происходит проверка на истину. Если булева переменная равна *true*, то действие будет выполнено.

До сих пор мы рассматривали сокращённый вид логики **if**. В полном виде она выглядит так:

```
If Условие выполнено then
 Действие1
else
Действие2
```

Я много раз говорил, что каждый оператор должен заканчиваться точкой с запятой. Запомни, что после любого оператора перед ***else*** *точка с запятой не ставится.*

Давай рассмотрим пару примеров:

```
var
 i:integer;

begin
 if i>0 then
  i:=i+10 //Заметь, что точки с запятой нет.
else
  i:=i+20

  if i<0 then
   begin
     i:=10;
     i:=i-2;
   end //Заметь, что точки с запятой нет.
else
   begin
     i:=20;
     i:=i-2;
   end;
```

Вот теперь я рассказал всё необходимое о логических операциях и как с ними работать. Так что можно двигаться дальше и постигать что-то новое.

 *На компакт диске, в директории* ***\Примеры\Глава 8\if*** *ты можешь увидеть пример программы, в которой используются различные типы логических операций.*

При использовании логических операций – результат сравнения всегда должен быть логическим. Нельзя написать так:

```
if i:=0 then
```

Здесь в операторе используется присваивание, а у него нет результата, а значит, произойдёт ошибка. Чтобы не делать таких ошибок, просто запомни, что при использовании оператора **if** обязательно должны использоваться только логические операции равенства, больше, меньше и так далее. Единственный случай, когда можно опускать операторы сравнения – когда проверяется логическая переменная.

```
var
 perem:Boolean;
begin
 perem:=true;
 if perem then
  perem:=false;
end;
```

В этом примере нет никаких операторов типа равенства, больше или меньше, потому что переменная **param** – логическая и проверяется она. Если она равна true, то следующее действие выполниться, если нет, то пропуститься.

8.6 Работа со строками.

В этой части расскажу, как можно работать со строками. Мы поговорим об основных функциях работы со строками и применим их в действии. Для этого я напишу несколько полезных примеров, и мы разберём их по косточкам.
Для начала, давай рассмотрим основные функции для работы со строками.

Length

Эта функция возвращает длину строки. У неё есть только один параметр – строка, длину которой надо вернуть. Функция Length выглядит так:

function Length(S): Integer;

Пример использования функции:

```
var
  Str:string;
```

```
 Index:Integer;
begin
 Str:='Привет с большого будуна';
 index:= Length(Str);
end;
```

В этом примере я объявил две переменные Str (строка) и index (целое число). В первой строчке кода я присваиваю в переменную Str строку «*Привет с большого будуна*». После этого я присваиваю переменной index длину строки Str. Результат, в переменной index будет число 24 – длина строки (если, конечно, я правильно подсчитал☺).

Copy

Эта функция возвращает указанный отрывок строки. Например, тебе нужно получить из строки «*Привет с большого будуна*» символы начиная с 7-го по 10-й. Это легко сделать с помощью функции ***copy***. У неё есть три параметра:

1. Строка, из которой нужно получить отрывок текста.
2. Начальный символ.
3. Количество нужных символов.

function Copy(S; Index, Count: Integer): string;

Рассмотрим пример:

```
var
  Str1:string;
  Str2:Integer;
begin
 Str1:='Привет с большого будуна';
 Str2:= copy(Str1, 7, 10);
end;
```

Здесь я объявил две строковых переменных: **Str1** и **Str2**. В первой строчке кода я присваиваю в переменную Str1 строку «*Привет с большого будуна*». В следующей строке, я копирую в переменную Str2 десять символов из Str1, начиная с седьмого символа. Результатом будет в **Str2** строка: «*с большого*».

Delete

Эта функция удаляет кусок текста из указанной строки. У неё есть три параметра:

1. Строка, из которой нужно удалить отрывок текста.
2. Начальный символ, начиная с которого будут удаляться символы.
3. Количество символов для удаления.

В общем виде функция выглядит так:

procedure Delete(var S: string; Index, Count:Integer);

Рассмотрим пример:

```
var
 Str1:string;
begin
 Str1:='Привет с большого будуна';
 Delete(Str1, 7, 10);
end;
```

В этом примере я удаляю из строки ***Str1*** символы, начиная с 7-го по 10-й. В результате в переменной ***Str1*** останется только строка «*Привет будуна*».

Pos

Эта функция ищет указанные символы в строке. Если эти символы найдены, то она вернёт порядковый номер, начиная с которого найдена нужная строка. У функции два параметра:

1. Строка, которую надо искать.
2. Строка, в которой надо искать.

Если подстрока не найдена, то функция вернёт ноль.

function Pos(Substr: string; S: string): Integer;

Рассмотрим пример:

```
var
 Str1:string;
 index: integer;
begin
 Str1:='Привет с большого будуна';
 index:=Pos('большого', Str1);
end;
```

В этом примере я запускаю поиск строки «*большого*» в строке *Str1*. В данном случае, строка «*большого*» есть в строке переменной *Str1*, и начинается с 9-го символа. Результат, в переменной *index* будет число 9.

Insert

Эта процедура вставляет одну строку в другую, начиная с указанного символа. У неё есть три параметра:

1. Строка, которую надо вставить.
2. Строка, в которую надо вставить.
3. Позиция, куда надо вставить.

В общем виде функция выглядит так:

procedure Insert(Source: string; var S: string; Index: Integer)

Рассмотрим пример:

```
var
  Str1:string;
  index: integer;
 begin
  Str1:='Привет с будуна';
  Insert('большого', Str1, 9);
end;
```

Здесь я вставляю в строку *Str1* текст «*большого*» начиная с девятого символа. Результатом будет строка «*Привет с большого будуна*».

На этом и закончим обзор функция для работы со строками и двинемся дальше по ступенькам к знаниям ☺.

8.7 Исключительные ситуации.

Пора нам уже познакомится с исключительными ситуациями. Для чего они нужны? Допустим, что у тебя есть участок кода, где может произойти ошибка. Как сделать так, чтобы программа не вылетала при её возникновении? Очень просто. Надо заключить этот код в блок проверки исключений и тогда твоя программа выдержит даже цунами :).

И так. Простейший блок исключений выгляди как:

```
TRY
  //Здесь ты пишешь код, в котором может произойти ошибка
 EXCEPT
  //Если ошибка произошла, то выполнится этот код
 END;
```

Рассмотрим простейший пример:

```
 TRY
  x:=y/0;
 EXCEPT
   //Здесь можно вывести сообщение об ошибке.
 END;
x:=0;
```

Между TRY и EXCEPT я вставил маленькое действие - деление на ноль. Компьютер не умеет делать такие вещи, поэтому произойдёт ошибка и выполнится код между EXCEPT и END. После обработки ошибки процедура заканчивает выполнение и все остальные операторы не будут выполнены, как, например, в нашем случае - **x:=0**;

Если бы мы поменяли 0 на любое другое число, то ошибки бы не было, и код между EXCEPT и END никогда не выполнился бы.

Давай посмотрим ещё пример:

```
var
 b:Tbitmap
begin
 b:= TBitmap.Create;

 TRY
  b.Canvas.Rectangle(1,1,100,100);

 EXCEPT
   //Здесь можно вывести сообщение об ошибке.
  b.free;
  exit;
 END;
 b.free;
end;
```

В этом примере мы создаём объект b типа Tbitmap (картинка). Потом начинаем блок TRY. В этом блоке мы пытаемся начать рисование. Если во время рисования произошла ошибка, то можно сообщить об этом пользователю и освобождается память b.free. После этого происходит выход из процедуры. Если ошибок не было, то просто освобождается память b.free.

Теперь давай разберёмся с ещё одним типом исключительных ситуаций - **TRY ... FINALLY**:

```
TRY
 //Здесь ты пишешь код, в котором может произойти ошибка
FINALLY
 //Этот код выполнится в любом случае
END;
```

Между *TRY* и *FINALLY* ты пишешь свой сомнительный код, в котором может произойти ошибка. А между *FINALLY* и *END* ты пишешь код, который должен выполнится в не зависимости от результата кода. В этом случае мы не можем информировать пользователя об ошибки, потому что в разделе *FINALLY* мы не знаем, произошла ошибка или нет. Зато вот такой пример будет уместен:

```
var
 b:TBitmap
begin
 b:= TBitmap.Create;
 TRY
  b.Canvas.Rectangle(1,1,100,100);
 FINALLY
  b.free;
 END;
end;
```

Подобный пример, мы уже рассматривали. В этом случае мы создаем объект b и пытаемся нарисовать. В разделе FINALLY мы удаляем созданный объект b. Теперь мы уверены, что b всегда будет удалён корректно, и мы освободим память. Потому что код

между *FINALLY* и *END*, будет выполняться всегда, вне зависимости от произошедших ошибок.

Если бы все программы в Windows были написаны корректно и сомнительные участки кода заключались бы в блоки исключительных ситуаций, пользователь забыл бы, что такое синий экран смерти. Если ты собираешься писать коммерческое программное обеспечение, то ошибки в нём непростительны. Никто не будет покупать дырявые программы, которые будут вылетать через каждые пять минут. Это я тебе говорю из своего опыта.

Будь внимателен. Ошибки могут появиться даже в самых неожиданных местах. Пользователь очень часто может тыкать не туда, куда надо и сбивать твоё творение с пути праведного ☺. Конечно же, можно говорить, что пользователь безрукий, но этот безрукий тебе платит за то, чтобы программа работала. Поэтому от его безмозглых действий нужно предохранятся. Исключительные ситуации тут один из лучших способов.

Глава 9. Создание рабочих приложений.

В этой главе я уже начну рассказывать о том, как создавать самые настоящие приложения. До этого момента я писал какие-то примеры, но они были примитивны и отображали только самое необходимое.

Начнём мы рассмотрение главы с создания меню. Это единственное, что опустил при рассмотрении палитры компонентов «**Standard**». Сейчас уже настало время восполнить этот пробел.

После этого мы научимся создавать панели кнопок и научимся с ними работать. Я буду описывать примеры, и показывать, как их создавать с точки зрения эргономики и дизайна. Хотя художник из меня никакой, но хоть какую-то красоту в наших программах мы наведём.

Эта глава будет насыщена интересными примерами и их описаниями. Я не люблю особо грузить сухой теорией и стараюсь все свои слова подкреплять практикой. Получается как бы я «отвечаю за базар» ☺.

9.1 Создание главного меню программы.

Эту часть я буду описывать чисто на практике. Теорию будем поглощать по мере продвижения нашего примера. Давай создадим маленькую программу, которая будет содержать какое-то меню. Какое именно не особо важно, лишь бы научится с ним работать.

Создай новое приложение. Брось на форму один компонент ***MainMenu***  . Теперь посмотрим какие есть свойства у этого компонента:

- *AutoHotkeys* – будут ли создаваться автоматически клавиши быстрого вызова. Если ты выберешь *maAutomatic*, то Delphi будет автоматически создавать клавиши. При *maManual* придётся это делать вручную.
- *AutoMegre* – автоматическое слияние с дочерними окнами.
- *Images* – сюда можно подключать списки картинок, которые смогут отображаться на пунктах меню.
- *Items* – здесь описываются пункты меню.

Давай сразу подключим список картинок. Брось на форму компонент ***ImageList*** с закладки **Win32**  . Теперь дважды щёлкни по нему и перед тобой откроется окно:

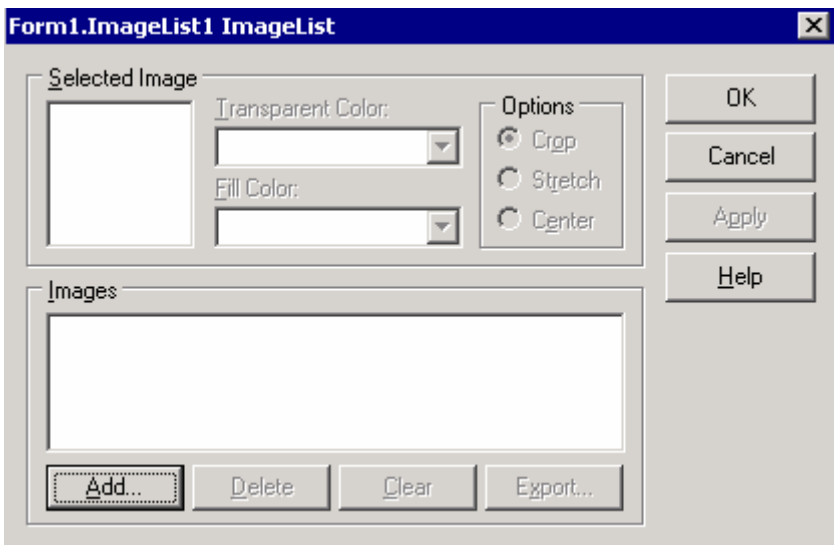


Рис 9.1 Добавление картинок

Здесь нажми кнопку *Add* чтобы добавить картинку. Откроется стандартное окно открытия файла. Открой какую-нибудь картинку, и она добавится в список. Желательно, чтобы она была размером 16x16. Именно такие габариты используются по умолчанию.

 *На компакт диске, в директории* ***\Images*** *ты можешь найти большое количество картинок для кнопок. А в директории* ***\Примеры\Глава 8\Menu*** *находятся исходники примера и картинки, которые я использовал в нём.*

Можешь таким образом добавить несколько картинок. Какие добавил я, смотри на рисунке:

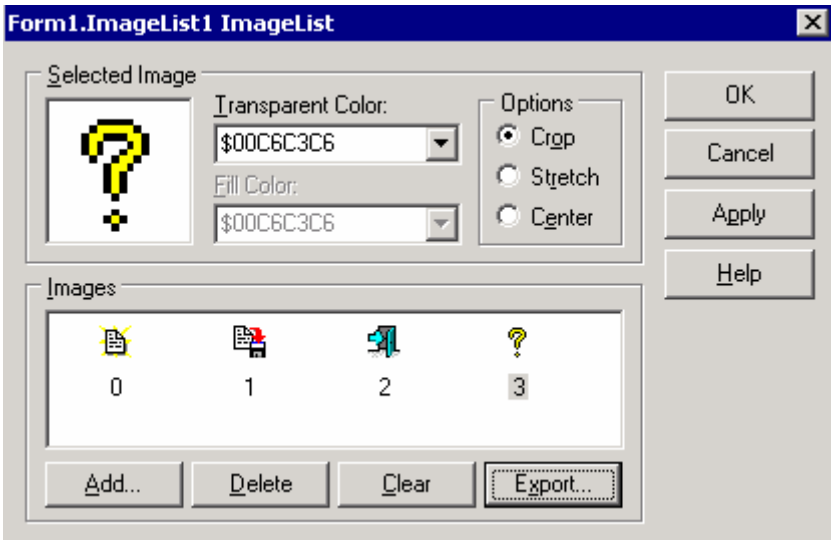


Рис 9.2 Заполненный список картинок.

Теперь подключим наш список картинок к менюшке. Выдели компонент *MainMenu1* и у свойства *Images*, в выпадающем списке выбери пункт *ImageList1*.

Теперь создадим само меню. Для этого дважды щёлкни по свойству Items и перед тобой откроется редактор меню:

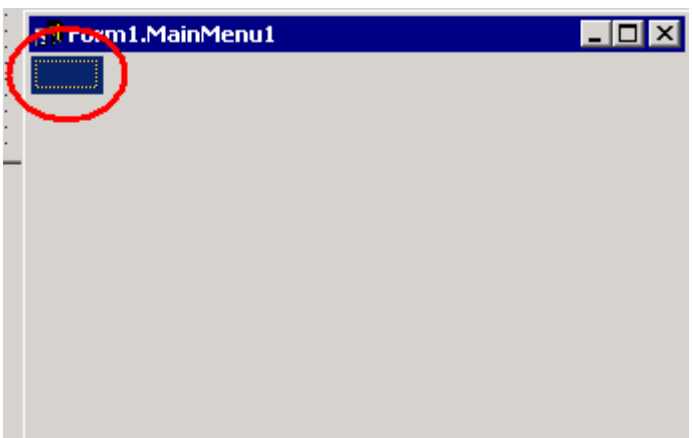


Рис 9.3 Редактор меню.

Этот же редактор можно вызвать, если дважды щёлкнуть по компоненту *MainMenu1*.

Красным кругом на рисунке я выделил уже созданный пункт. Перейди в объектный инспектор и набери в свойстве *Caption* слово «Файл». Как только ты нажмёшь кнопку Enter, будет создано меню «Файл»:

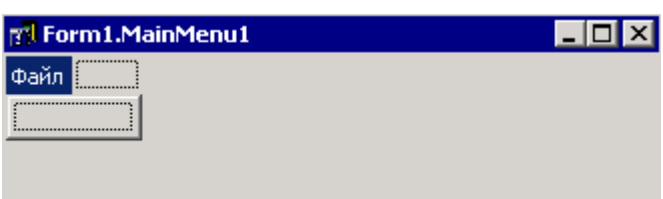
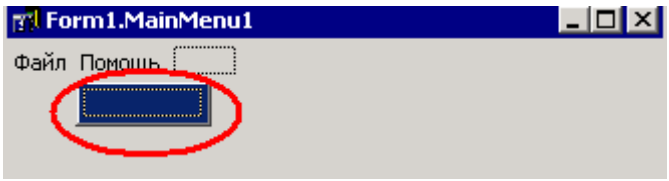


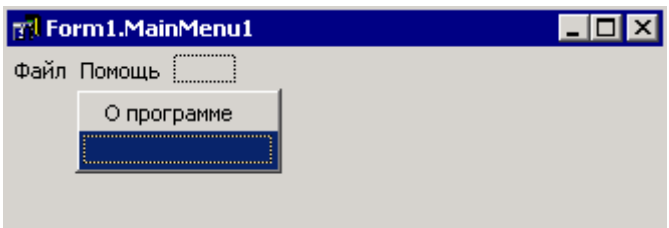
Рис 9.3 Меню «Файл».

Давай создадим ещё и меню «Помощь». Щёлкни справа от созданного меню (в рамочке обведённой пунктиром) и снова перейди в объектный инспектор. Там введи в свойстве *Caption* слово «Помощь».

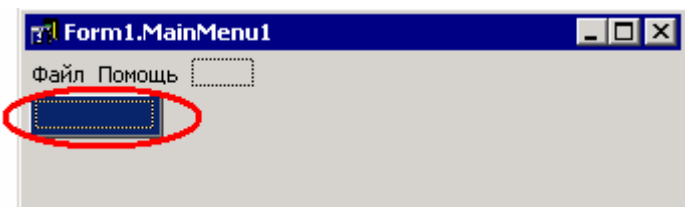
Теперь создадим подпункт для меню «Помощь». Щёлкни в рамке чуть ниже меню «Помощь», как показано на рисунке:



Здесь, в свойстве *Caption* мы введём слово «О программе». У тебя должно получится что-то похожее на этот скрин:

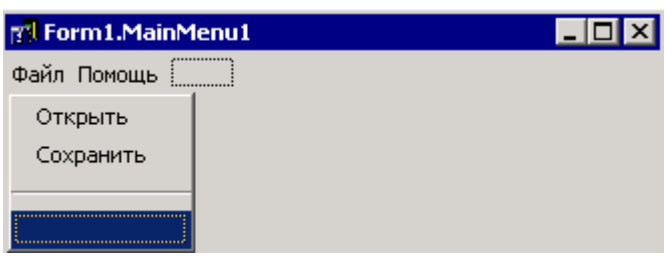


Теперь, таким же образом заполним меню «Файл». Выдели его. Теперь щёлкни в рамочке чуть ниже.



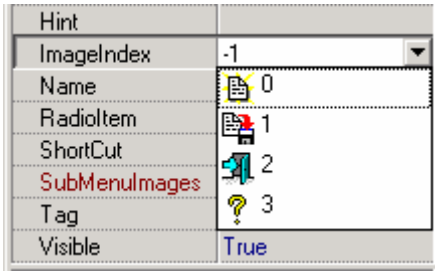
Здесь мы напишем в свойстве *Caption* слово «Открыть». Когда ты нажмёшь Enter или перейдёшь на другой пункт меню в редакторе, создаться пункт «Открыть» и тут же немного ниже создаётся пустой пункт. Щёлкни по нему и введи в свойстве *Caption* слово «Сохранить».

Теперь снова щёлкни на новом пункте меню и у него в свойстве *Caption* просто тире «-». Это заставит Delphi создать сепаратор:



И, наконец, создадим последний пункт – «Выход». Теперь назначим каждому пункту меню картинки. Я опишу, как это делать на примере пункта «Открыть», а остальные ты сделаешь сам.

Выдели пункт «Открыть». Теперь в объектном инспекторе щёлкни по выпадающему списку свойства «ImageIndex». Перед тобой откроется список всех картинок, которые ты подключил:



Выбери тот, что тебе хочется, и картинка уже подключена. В редакторе меню картинка не будет видна, зато в редакторе форм ты сразу можешь увидеть её.

*На компакт диске, в директории **\Примеры\Глава 9\menu** ты можешь увидеть пример этой программы.*

Теперь создадим обработчик события нажатия по пункту меню. Для этого выбери в дизайнере меню пункт «*Выход*» и щёлкни по нему дважды или перейди на закладку Events и дважды щёлкни по событию *OnClick*. Эти действия заставят Delphi создать обработчик события по нажатию меню. В этом обработчике напишем следующее:

```
procedure TForm1.N7Click(Sender: TObject);
begin
 Close;
end;
```

Здесь мы используем метод формы ***Close***. Если я не ошибаюсь, то я уже говорил о нём. Если нет, то напомню, что этот метод закрывает форму. Если мы закрываем главную форму, то закроется всё приложение.

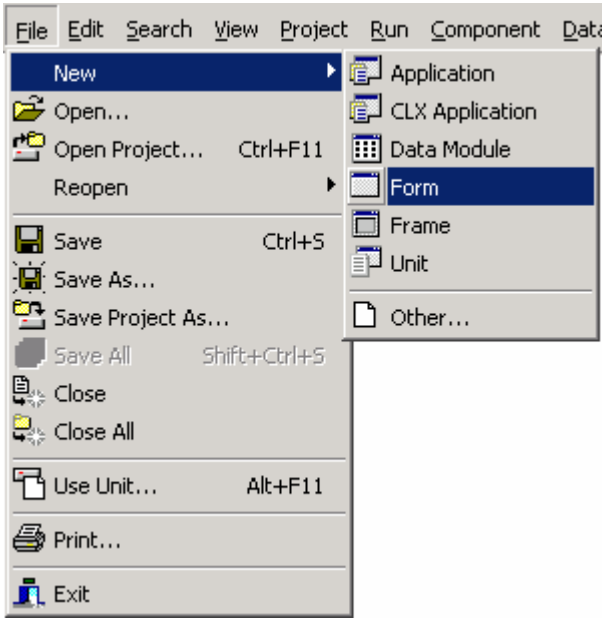
*На компакт диске, в директории **\Примеры\Глава 9\menu1** ты можешь увидеть пример этой программы.*

9.2 Создание дочерних окон.

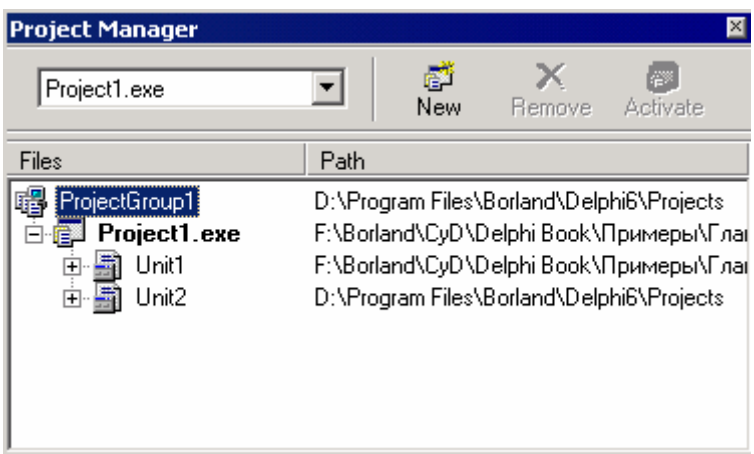
До сих пор мы создавали простые приложения, состоящие из одного только главного окна. В этой части главы я расскажу, как создать приложение, состоящее из одного главного окна, которое может вызывать дочернее окно.

Для примера я буду использовать исходник из предыдущей части главы, где мы создали меню. Открой тот проект.

Для начала создадим новую форму. Для этого, из меню File выбери пункт New, а затем выбери пункт Form, как показано на рисунке ниже. Это касается только Delphi 6. В более старых версиях нужно просто выбрать File -> New Form (я постараюсь всегда показывать отличия старых версий от Delphi 6).



Delphi должен создать новую чистую форму. Открой менеджер проектов (View -> Project Manager). Посмотри на его содержимое и убедись, что в твоём проекте Project1.exe теперь есть две формы: *Unit1* и *Unit2*:



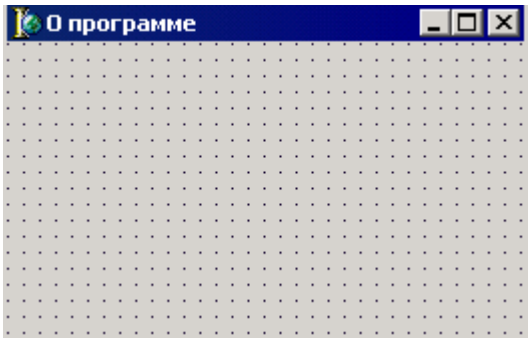
Двойной щелчок по любой из форм в менеджере проектов вызовет форму в дизайнер для редактирования. В принципе, новая форма уже открыта, и нам не надо дважды щёлкнуть. Хотя можешь попробовать дважды кликнуть по *Unit1*, и Delphi моментально откроет эту форму для редактирования.

Давай сразу сохраним новую форму. Для этого при выделенной новой форме нажми Ctrl+S. Перед тобой появится окно для ввода имени формы. Я тебе говорил, что имена нужно задавать осмысленные, хотя сам это правило пока что постоянно нарушал. Теперь я этого делать не буду. Это окно у нас будет показывать информацию о программе, поэтому я назвал его AboutUnit.pas. Модуль главной формы я переименовал в MainUnit.pas.

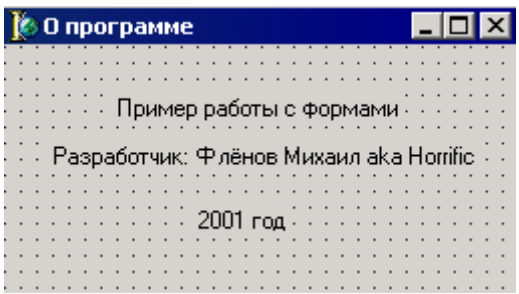
Нельзя просто так переименовывать имена модулей. Для этого желательно использовать меню ***File->Save As****.*

Сразу измени и имя формы с *Form2* на *AboutForm*.

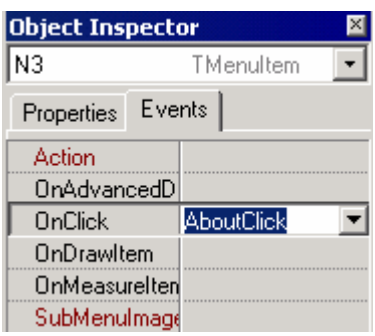
Сохранили. Теперь изменим заголовок формы на *«О программ»*.



Можешь ещё приукрасить как-нибудь эту форму. Лично я брошу несколько компонентов TLabel, чтобы сделать надписи. Но это уже не важно. Для нас главное научится работать с этими формами.



Теперь нужно показать это окно. Давай создадим обработчик события *OnClick* для пункта меню «*О программе*» у нашей главной формы. Когда ты создашь обработчик, то Delphi даст процедуре тупое название типа *N4Click*. Число у тебя может отличаться. Как видишь, имя процедуры ничего нам не говорит. Но мы же договорились, что всё будем называть понятными именами, поэтому переименуй её в объектном инспекторе в AboutClick.

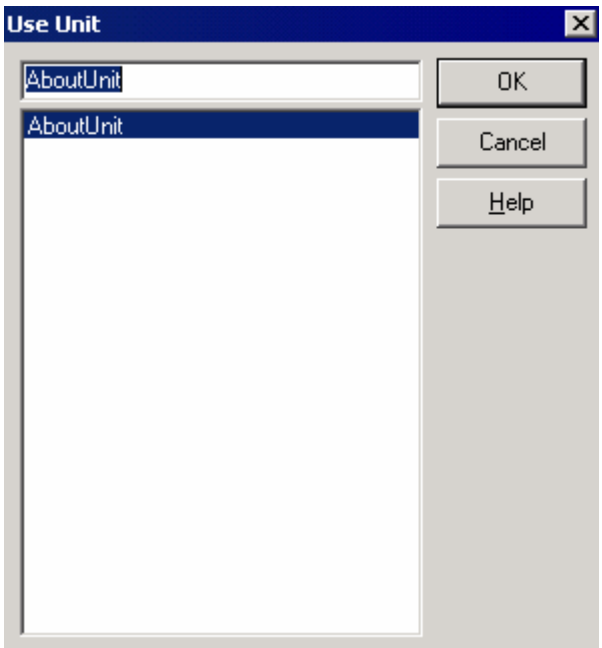


Теперь в получившемся обработчике напишем следующее:

```
procedure TForm1.AboutClick(Sender: TObject);
begin
 AboutForm.ShowModal;
end;
```

В этом коде я вызываю метод *ShowModal* окна *AboutForm*. Этот метод показывает форму в режиме Modal. В этом режиме окно получает полное управление, и пока оно не закроется, главная форма не будет работать.

Если ты попробуешь сейчас откомпилировать код, то получишь ошибку. В Delphi 5 это будет просто ошибка, что *AboutForm* не найдена. Это потому, что эта форма описана в нашем модуле *AboutUnit*, а мы используем её в *MainUnit*. Чтобы *MainUnit* смог увидеть форму, описанную в *AboutUnit*, нужно её подключить. Для этого перейди в модуль *MainUnit* и из меню *File* выбери пункт *Use Unit*. Перед тобой откроется окно:



В этом окне нужно выбрать модуль, который ты хочешь подключить и нажать кнопку «*OK*». Что после этого изменится? Посмотрим на следующий отрывок кода:

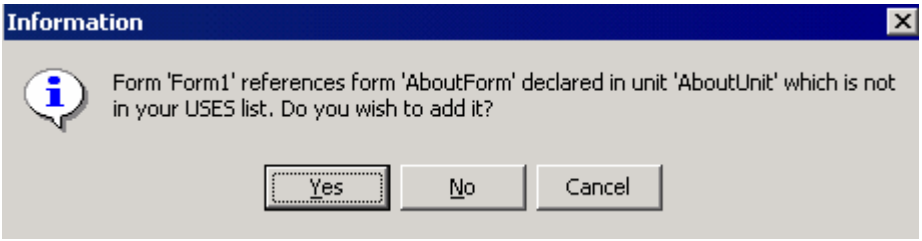
```
var
  Form1: TForm1;

implementation

uses AboutUnit;

{$R *.dfm}
```

Как видишь, здесь появилась новая строчка uses. Точно такая же есть в начале модуля, но там мы подключали стандартные заголовочные файлы. Здесь же мы подключаем модули написанные нами. В принципе, мы могли подключить модуль AboutUnit и в самом начале, но это делать не желательно.

В принципе, эту строку **uses** мы можем написать и вручную на этом месте и не выполнять никаких описанных мною выше действий. Так что выбирай, какой способ тебе удобнее – прописывать вручную или делать это автоматически.

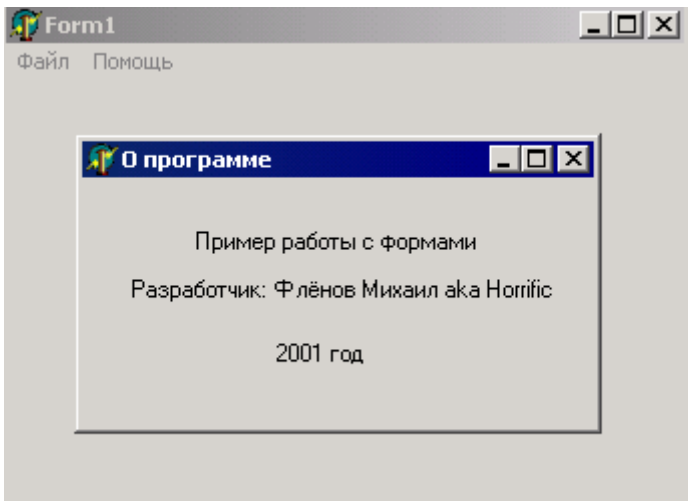
Теперь к форме *MainUnit* мы подключили наш модуль *AboutUnit* и можем смело использовать её содержимое.

Обладатели Delphi6 находятся в более удобном положении. Если ты забыл подключить модуль и откомпилировал код, то помимо ошибок ты увидишь следующее окно:



Здесь написано, что ты из главного модуля ссылаешься на форму *AboutForm*, которая объявлена в модуле *AboutUnit*. Тебе также предлагается подключить этот модуль. Если ты нажмёшь «Yes», то Delphi моментально сделает все действия для подключения.

Вот теперь можно компилировать код. Запусти программу и попробуй выбрать пункт меню «*О программе*». Если ты всё сделал правильно, то ты увидишь вторую созданную нами форму.



На компакт диске, в директории ***\Примеры\Глава 9\Forms*** *ты можешь увидеть пример этой программы.*

9.3 Модальные и не модальные окна.

В предыдущем примере мы создали главное окно, которое вызывает дочернее в виде модального окна. Что значит модальное? Это значит, что управление полностью передаётся ему. Как только программа натыкается на код *AboutForm.ShowModal*, работа главной формы останавливается, и управление полностью передаётся дочерней форме. Пока модальное окно не закроется, главная форма работать не будет!!!

Рассмотрим простой пример:

```
procedure TForm1.AboutClick(Sender: TObject);
var
 Index:Integer;
begin
 AboutForm.ShowModal;

 Index:=10;
end;
```

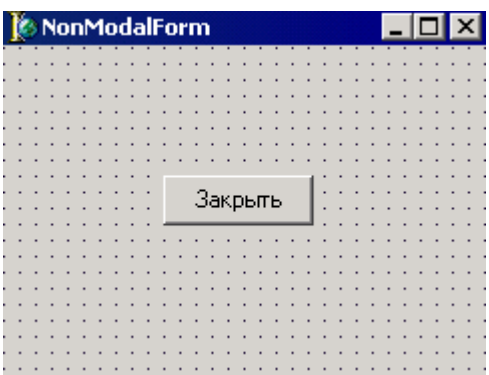
В этом примере я показываю модальное окно и после этого присваиваю переменной **Index** значение 10. Так вот, переменная **index** получит значение 10, но только после того, как модальное окно *AboutForm* закроется.

Для того чтобы создать не модальное окно, нужно вызвать метод ***Show***. В этом случае главная форма создаст дочернее, показав его на экране, и смело продолжит выполняться дальше. Это позволит тебе работать с обеими формами одновременно, переключаться между ними и код обеих форм будет выполнятся как бы параллельно. Рассмотрим пример:

```
procedure TForm1.AboutClick(Sender: TObject);
var
 Index:Integer;
begin
 AboutForm.Show;
 Index:=10;
end;
```

В этом случае я создаю немодальное окно, и выполнение кода не останавливается на точке *AboutForm.Show*, в ожидании закрытия окна, а спокойно продолжается дальше. То есть будет показано дочернее окно и моментально переменной index будет присвоено значение 10.

Давай создадим ещё одну форму. Сразу переименуем её свойство *Name* в *NonModalForm*. Можешь что-нибудь написать на ней, а я брошу только одну кнопку, с помощью которой можно будет закрыть это окно:



Сохрани новую форму под именем *NonModalUnit.pas*.

Теперь вернёмся в главную форму и допишем в раздел **uses** имя модуля *NonModalUnit*:

```
uses AboutUnit, NonModalUnit;
```

Если не хочешь подключать эту форму вручную, то выбери из меню *File* пункт *Use Unit*, и выбери там имя модуля для подключения.

Всё, модуль подключён, теперь можно его использовать. Создадим обработчик события для пункта меню «*Сохранить*» и напишем в нём следующее:

```
procedure TForm1.SaveClick(Sender: TObject);
begin
 NonModalForm.Show;
end;
```

Здесь я показываю форму *NonModalForm* как немодальное окно. Это значит, что если ты запустишь программу и выберешь из меню пункт «*Сохранить*», то увидишь окно новой формы и сможешь спокойно переключатся между главной формой и NonModalForm без каких-либо проблем.

*На компакт диске, в директории **\Примеры\Глава 9\Forms1** ты можешь увидеть пример этой программы.*

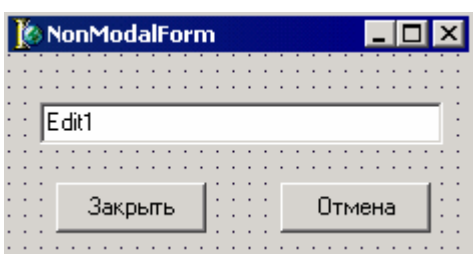
В последствии мы ещё очень много будем использовать оба типа окон (в основном модальные) и ты познакомишься с их работой более подробно.

9.4 Обмен данными между формами.

Зачем нужны все эти формы, если нельзя передавать параметры между ними и взаимно использовать процедуры. Когда мы вызываем какой-то диалог, мы хотим, чтобы пользователь ввёл в него какие-то данные. После этого мы должны получить введённые данные в главном окне и как-то обработать. Помимо этого, на диалоге очень часто располагаются кнопки «*Да*» и «*Нет*». Мы просто обязаны знать, какую кнопку нажал пользователь и в зависимости от этого обрабатывать ввод или нет.

В этой части главы мы закрепим всё то, о чём говорили в предыдущих частях про формы и научимся с ними работать. Пока мы знаем только, как их создавать и выводить на экран, а работа с ними осталась за кадром. Пора это исправить.

В прошлой части мы создали немодальное окно для пункта меню «Сохранить». Я немного изменил то окна, добавив на него строку ввода:



Теперь посмотрим на очень интересное свойство кнопки «Закрыть» - ModalResult. В этом свойстве мы можем задавать значение, возвращаемое при закрытии окна. Давай выберем здесь «*mrOk*». Теперь если мы покажем окно как модальное и потом закроем его кнопкой «Закрыть», то функция ShowModal вернёт нам значение *mrOk*.

Я ещё добавил на форму кнопку «Отмена», у которой свойство *ModalResult* я установил в *mrCancel*. Кстати, теперь ты должен очистить обработчики событий *OnClick*, для кнопок. Когда ты указал в свойстве *ModalResult* возвращаемое значение, кнопка уже автоматически умеет закрывать окно и не нужно создавать для неё обработчик *OnClick* и в нём писать метод *Close*.

В связи с этим, предлагаю изменить обработчик события OnClick для пункта меню «Сохранить»:

```
procedure TForm1.SaveClick(Sender: TObject);
begin
 if NonModalForm.ShowModal=mrOK then
  Application.MessageBox(PChar(NonModalForm.Edit1.Text),
       'Ты ввёл:', MB_OKCANCEL)
end;
```

Теперь построчно рассмотрим код. В первой строке я вызываю модальное окно и сразу проверяю возвращаемое значение. Если оно равно *mrOK* то выполняю следующее действие (if NonModalForm.ShowModal=mrOK then).

Вторая строка показывает стандартное окно диалога. Я это делаю с помощью метода MessageBox объекта Application. У этого метода три параметра:

1) Строка, которая будет показана внутри окна.
2) Строка заголовка окна.
3) Кнопки, которые будут на окне.
   - MB_OK – кнопка «ОК».
   - MB_OKCANCEL – кнопки «ОК» и «Отмена».
   - MB_RETRYCANCEL – кнопки «Повторить» и «Отмена».
   - MB_YESNO – кнопки «Да» и «Нет».
   - MB_YESNOCANCEL – кнопки «Да», «Нет» и «Отмена».

В качестве текста сообщения в окне я вывожу текст, введённый в строку ввода нашего модального окна (NonModalForm.Edit1.Text).

Теперь если пользователь нажмёт кнопку «Закрыть» в модальном окне, то появится окно с введённым текстом. Иначе ничего не произойдёт.

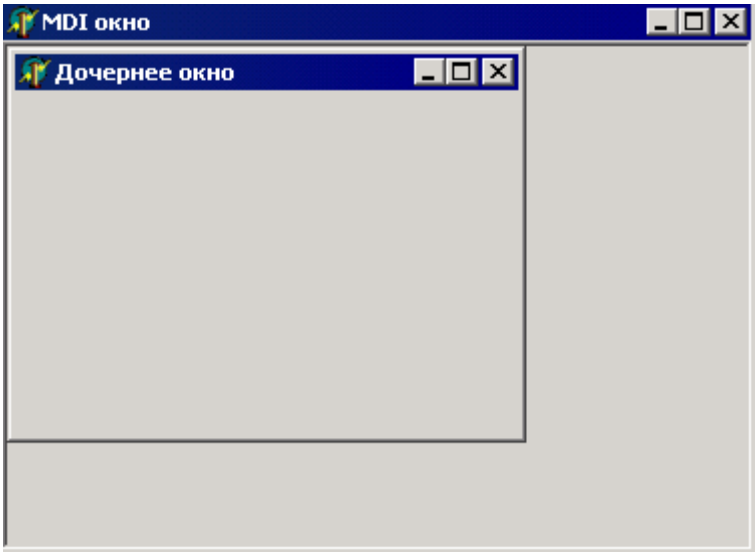
Вот и всё. В этом примере мы смогли показать окно, получить результат его выполнения и вывели введённый текст.

 *На компакт диске, в директории* ***\Примеры\Глава 9\Forms2*** *ты можешь увидеть пример этой программы.*

9.5 Многодокументные MDI окна.

Что такое многодокументные MDI окна? Это когда главное окно содержит внутри себя несколько подчинённых окон. Дочерние окна чем то похожи на немодальные. Они так же не тормозят главное окно и работают независимо, только их область видимости ограничивается главным окном. Они находятся как бы внутри главного окна.

Хотя Microsoft уже не рекомендует использовать эту технологию, и убрала её из MS Word, сама её продолжает использовать. В Windows 2000 ярким примером MDI окон является консоль MMC.



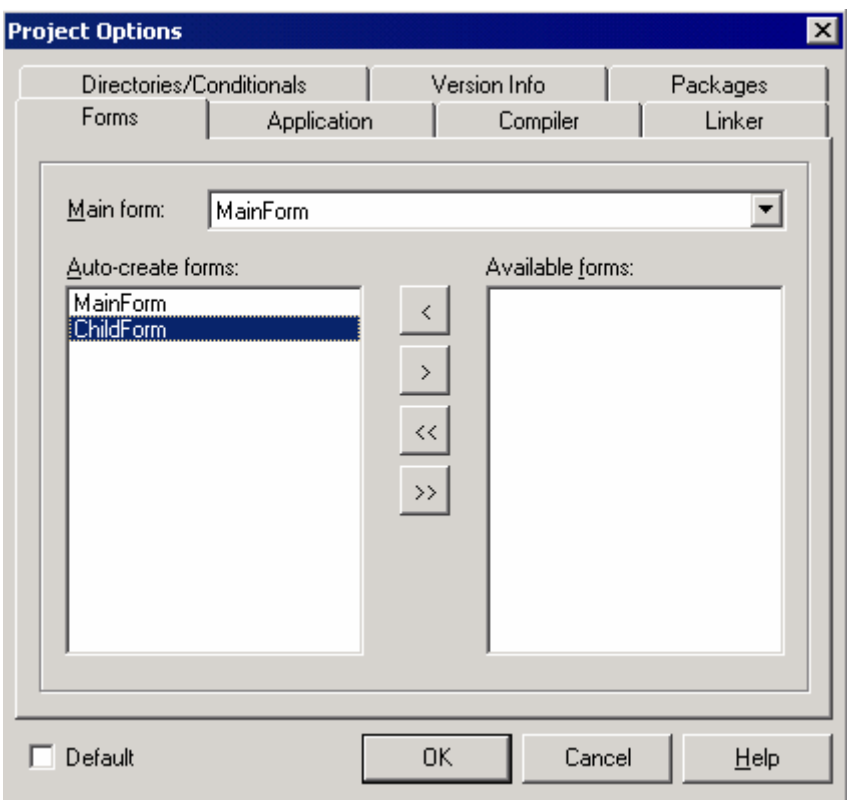
Пример MDI окна

Давай создадим простейшую MDI программу. Создай новое приложение. Сохрани главное окно под именем ***MainModule***, а проект под именем ***mdi***. Теперь измени свойство ***FormStyle*** у формы на *fsMDIForm*.

Теперь создай ещё одно окно (дай ему имя ChildForm) и измени у него свойство ***FormStyle*** у формы на *fsMDIChild*.

Вот и всё. Никакого геморроя, а MDI программа уже готова. Можешь запустить и посмотреть, как она работает.

*На компакт диске, в директории **\Примеры\Глава 9\MDI** ты можешь увидеть пример этой программы.*

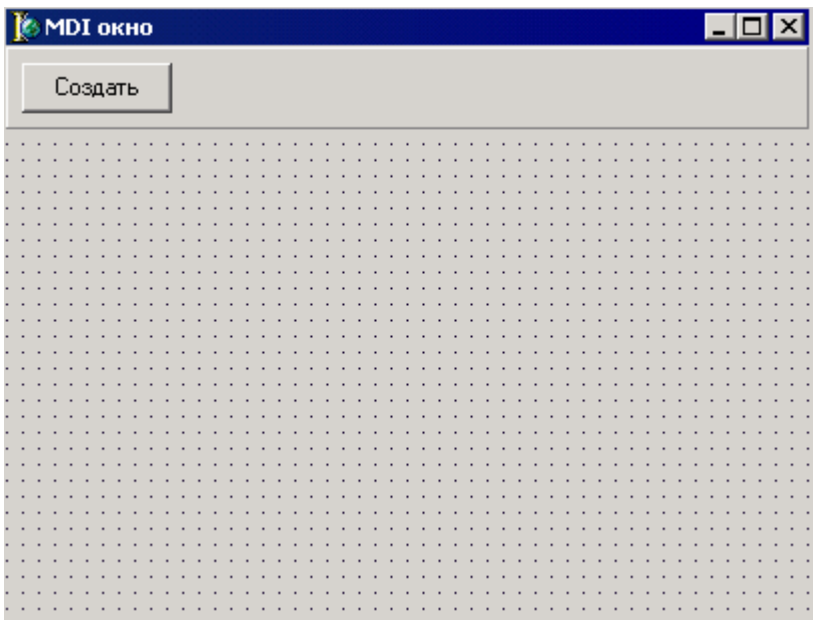


В нашем случае программа запускается и сразу создаётся дочернее окно. Как убрать его и создавать в рантайме (во время выполнения программы)? Очень просто выбери пункт меню «Options» из меню «Project» и ты увидишь следующее окно, показанное на рисунке выше.

В левой части окна перечислены те формы, которые будут создаваться автоматически (*Auto-create forms*). Выдели тут ChildForm (наше дочернее окно) и перемести его в список *Available forms*, нажав кнопку  .

Теперь наша дочерняя форма не будет создаваться автоматически, и это придётся делать вручную. Ну, ничего, это не такая уж и проблема, как-нибудь справимся и победим эту проблему.

Брось на форму панельку и растяни её по верхнему краю окна (свойство Align надо установить в *alTop*). Теперь на панель бросим кнопку и дадим ей заголовок «Создать».



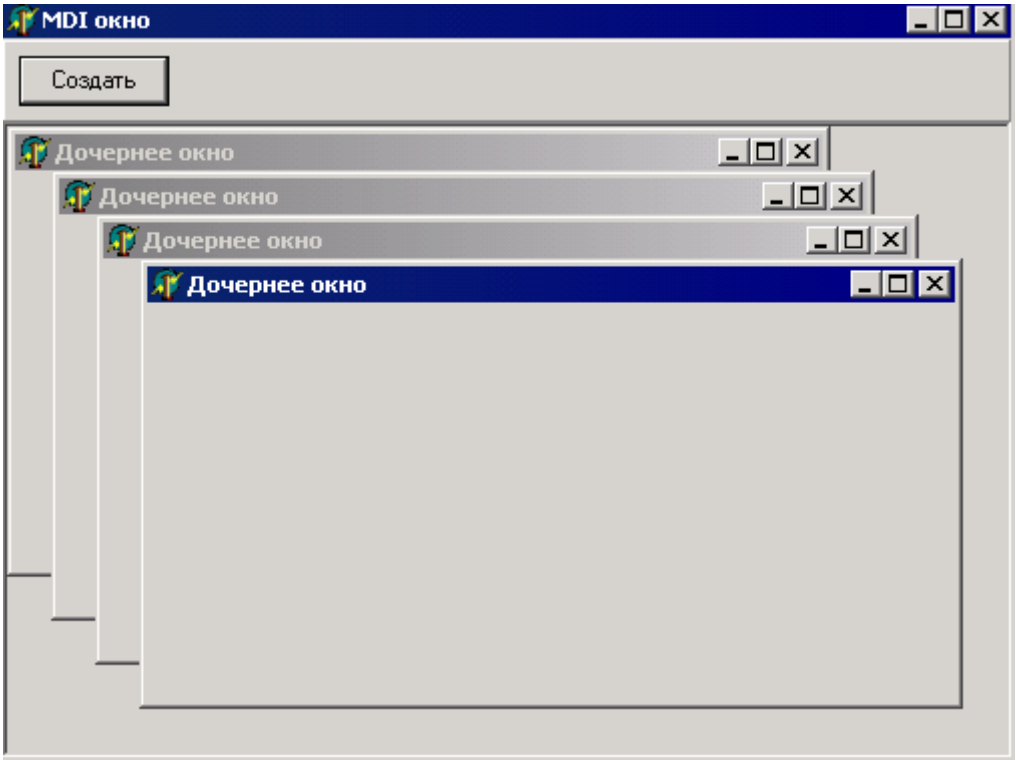
По нажатию этой кнопки мы будем вручную создавать окно

```
procedure TMainForm.CreateButtonClick(Sender: TObject);
begin
 ChildForm:= TChildForm.Create(Owner);
end;
```

Здесь я переменной ChildForm присваиваю указатель на новое созданное окно TChildForm.Create. Переменная ChildForm объявлена в модуле дочернего окна в разделе var:

```
var
  ChildForm: TChildForm;
```

Теперь запусти программу и попробуй несколько раз нажать на кнопку «Создать». У тебя должно создаться сразу несколько дочерних окон:



Но если ты попытаешься закрыть любое из них, оно просто свернётся. Чтобы окно закрывалось, нужно создать обработчик события ***OnClose*** для дочерней формы и в нём написать

```
procedure TChildForm.FormClose(Sender: TObject; var
                Action: TCloseAction);
begin
 Action:=caFree;
end;
```

Вот теперь наше приложение готово. В процессе книги мы ещё будем писать подобные приложения, поэтому успеем познакомиться с подобными приложениями.

На компакт диске, в директории ***\Примеры\Глава 9\MDI*** *ты можешь увидеть пример этой программы.*

Напоследок дам несколько полезных свойств и методов формы, пример которых уже реализован в примере выше.

ActiveMDIChild – указывает на активное дочернее окно. Например, тебе надо изменить заголовок активной дочерней формы. Как тебе узнать, какая из них активная, когда их несколько? Очень просто. Создай в главной форме кнопку и по её нажатию напиши:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
 ActiveMDIChild.Caption:='Активное дочеренее окно';
end;
```

В этом коде я изменяю свойство *Caption* активной формы. Если нет активной дочерней формы (бывает, когда дочерних форм вообще нет), то свойство *ActiveMDIChild* равно **nil**.

MDIChildCount –целое число указывающее на количество дочерних окон.

MDIChildren – через это свойство можно получить доступ к любому дочернему окну. Например, второе окно можно получить с помощью *MDIChildren[2]*.

Давай попробуем изменить заголовки всех дочерних окон. Для этого запустим цикл от 0 до MDIChildCount и изменим все заголовки:

```
for i:=0 to MDIChildCount-1 do
 MDIChildren[i].Caption:=’Новый заголовок’;
```

Есть ещё несколько интересных методов:

ArrangeIcons – выстроить иконки всех дочерних окон.
Cascade – выстроить каскадом все дочерние окна.
Next – Следующее дочернее окно.
Previous – Предыдущее дочернее окно
Tile – тоже выстроить дочерние окна.

9.6 Инициализация окон.

Вот теперь мы написали уже достаточно много примеров и готовы узнать, как инициализируются окна. В этой части я покажу: из чего состоит сердце нашей программы, где инициализируются окна и как управлять этим процессом. До этого момента я не хотел этого показывать, чтобы не забивать тебе голову, но теперь это необходимо, для продолжения разговора.

Создай новый проект. Сохрани модуль главной формы под именем *MainUnit.pas*, а проект под именем *SplashProject.dpr*. Теперь выбери из меню *Project* пункт *View Source*, чтобы увидеть исходный код проекта:

```
program SplashProject;

uses
  Forms,
  MainUnit in 'MainUnit.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Всё это ничто иное, как содержимое файла *SplashProject.dpr*. Первой строкой стоит имя программы **program SplashProject**. В этой строке ничего менять нельзя. После этого идёт уже знакомый раздел **uses** в котором можно подключать необходимые модули. У нас подключен модуль *Forms* (позволяет работать с формами) и *MainUnit* (модуль главного окна).

Между **begin** и **end** выполняется три строчки:

1. *Application.Initialize* – запускает инициализацию приложения. Убирать не рекомендуется.

2. *Application.CreateForm (TForm1, Form1)* – метод *CreateForm* инициализирует форму. У него два параметра – имя объекта и имя переменной, которая в последствии будет указывать на созданный объект. В нашем случае это имя формы *TForm1* и имя переменной *Form1*. Переменная *Form1* автоматически описывается в модуле объекта *TForm1* (в нашем случае это модуль *MainUnit.pas*) в разделе **var**:

```
var
  Form1: TForm1;
```

3. *Application.Run* – после инициализации всех форм можно запускать выполнение программы с помощью метода *Run* объекта *Application*.

Здесь везде используется объект *Application*. Этот объект всегда существует в твоих программах в единственном экземпляре и создаётся здесь с помощью строки *Application.Initialize*. С этим объектом мы будет знакомиться постепенно на протяжении всей книги, а сейчас достаточно этих знаний.

Теперь создай новую форму *File->New->Form* и сохрани её под именем *SplashUnit.pas*. Снова посмотри на исходник проекта, он должен быть уже таким:

```
program SplashProject;

uses
  Forms,
  MainUnit in 'MainUnit.pas' {Form1},
  SplashUnit in 'SplashUnit.pas' {Form2};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

В раздел **uses** добавилось объявление нового модуля, а между **begin** и **end** появился код создания новой формы.

Теперь войди в свойства проекта (из меню *Project* нужно выбрать пункт *Option*). На закладке *Forms*, в списке *Auto-create forms* у нас описано две формы. Выдели *Form2* и перенеси её в список *Available forms*. Закрой окно свойств кнопкой *OK* и посмотри на исходник проекта. Как видишь, строка инициализации второй формы исчезла. Это потому

что мы перенесли её из списка автоматически создаваемых форм в список доступных форм. Теперь, чтобы использовать *Form2*, мы её должны сначала создать.

Чтобы дальше можно было удобнее работать, переименуй главную форму *Form1* в *MainForm*, а вторую форму *Form2* в *SplashForm*. Теперь брось на главную форму кнопку и по её нажатию напиши следующий код:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
 Application.CreateForm(TSplashForm, SplashForm);
 SplashForm.ShowModal;
 SplashForm.Free;
end;
```

Здесь, в первой строке кода я инициализирую форму SplashForm. Во второй, созданное окно выводиться на экран. И в последней строке я уничтожаю окно.

Но есть ещё один способ создания окон, который мы уже использовали и я предпочитаю именно его:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
 SplashForm:=TSplashForm.Create(Owner);
 SplashForm.ShowModal;
 SplashForm.Free;
end;
```

Здесь я присваиваю в переменную *SplashForm* результат вызова метода *Create* объекта *TSplashForm*. Этому методу нужно передать только один параметр – владельца окна. Если владельца нет, то можно передавать **nil**, а в нашем случае я передаю *Owner* – свойство, в котором храниться указатель на текущее окно. Если главным окном должно быть не текущее окошко, то нужно указать имя объекта – *Form1.Owner*.

Давай сделаем так, чтобы наше окно *SplashForm*, появлялось на время загрузки программы. Подобные окна ты видишь при старте Word, Excel и других приложений. Для этого зайди в исходник проекта и подкорректируй его до следующего вида:

```
begin
  SplashForm:=TSplashForm.Create(nil);
  SplashForm.Show;
  SplashForm.Repaint;
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Sleep(1000);
  SplashForm.Hide;
  SplashForm.Free;
  Application.Run;
end;
```

Давай рассмотрим этот код построчно:

1. Я создаю окно *SplashForm*. У этого окна нет владельца, поэтому в качестве параметра методу *Create* я указываю значение **nil**.

2. Отображаю окно на экране не модально.

3. Перерисовать окно с помощью вызова метода *Repaint*.

4. Инициализация приложения.

5. Создаётся форма *TMainForm*.

6. Делаю задержку, чтобы окно *SplashForm* могло хоть немного повисеть на экране. Для этого я использую процедуру *Sleep*, а в качестве параметра я указываю время задержки в миллисекундах. Одна секунда равна 1000 миллисекунд. Для использования этой функции в раздел **uses** нужно добавить модуль *Windows*.

7. Прячу форму *SplashForm*, вызовом метода *Hide*.

8. Уничтожаю окно.

9. Запускаю приложение.

Запусти программу, и ты сначала увидишь окно *SplashForm* (я на него поместил текст *TLabel* с надписью «*Идёт загрузка*»), а потом уже появиться главное окно.

Когда создаются окна (вызывается CreateForm), то программа выполняет обработчики события *OnCreate* всех создаваемых форм. Если у тебя приложение слишком большое и происходят долгие операции в этих обработчиках, то я тебе советую показывать подобное *SplashForm* окно. В этом случае, первым делом создаётся именно оно и отображается на экране. Пользователь видит, что идёт загрузка и спокойно ждёт. Если ты не будешь информировать о ходе загрузки, и на экране ничего появляться не будет, то пользователь может подумать, что программа зависла. Поэтому лучше лишний раз создать такое окно, чем потом пользователи будут ругаться на твою программу.

 *На компакт диске, в директории* ***\Примеры\Глава 9\Splash*** *ты можешь увидеть пример этой программы.*

Глава 10. Основные приёмы кодинга.

Мы уже изучили достаточно теории, и готовы приступить к реальным примерам кодинга. В этой главе мы будем знакомиться с некоторыми приёмами кодинга и писать простые утилиты и программы.

Здесь мы познакомимся с работой с реестром и потоками. Обе эти технологии просто необходимы любому программисту. Мы напишем кучу примеров максимально приближенных к боевым.

Я постараюсь дать побольше примеров различных приёмов кодинга как это делалось в предыдущих главах. Надеюсь, что это тебе в будущем пригодится.

Эта глава будет более практическая и теории здесь будет уже намного меньше. Да и вообще, чем дальше мы движемся, тем меньше теории и больше практики.

Большинство материала будет посвящено сохранению и чтению данных с разных источников информации (например из файлов или реестра). Остальной материал будет описывать то, что может понадобится при работе с источниками инфы.

10.1 Работа с массивами (динамические массивы).

Это одна из необходимых тем в кодинге. Мы будем достаточно часто использовать массивы при программировании, поэтому тебе необходимо прочитать эту часть полностью и понять всё, что я буду говорить.

Я, наверно очень часто говорю, что эта глава очень важна. Это действительно так, ведь я пытаюсь рассказать в своей книге самое необходимое и самое важное. Поэтому каждая её часть является необходимой. Если ты хочешь создавать удобный в понимании код, то придётся изучать различные приёмы и технологии. Я же даю необходимые основы.

Что такое массив? Это просто набор каких-то данных следующих друг за другом. Массив в Delphi обозначается как **array**. Чтобы объявить переменную типа массива нужно описать её в разделе **var** следующим образом:

```
var
  r: array [длина массива] of тип данных;
```

Как определяется, длина массива? Очень просто, это даже похоже на геометрическое определение. Например, тебе нужен массив из 12 значений. Длина такого массива может быть **[0..11]** или **[1..12]**. В квадратных скобках ты должен поставить начальное значение массива и конечное, а между ними две точки.

Тип данных может быть любой из уже пройдённых нами. Например, тебе надо объявить массив их 12 строк, это можно сделать следующим образом:

```
var
 r: array [0..11] of String;
```

В этом примере я объявил переменную r типа массив из 12 строк.

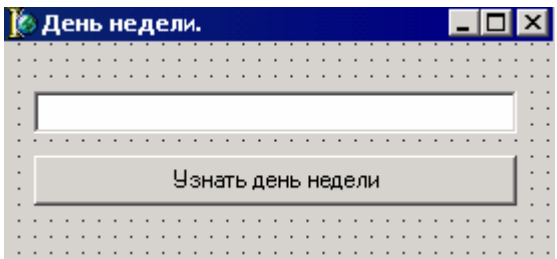
Чтобы получить доступ к какому-то элементу, нужно написать имя переменной массива и после этого, в квадратных скобках написать номер элемента, к которому нужно получить доступ. Например, давай прочитаем 5-й элемент и запишем 7-й элемент нашего массива:

```
var
 r: array [0..11] of String;
 Str:String;
begin
 Str:=r[5];

 r[7]:='Привет';
end;
```

В этом примере я в первой строке кода присваиваю переменной *Str* значение пятого элемента массива (*Str:=r[5];)*. В следующей строке я седьмому элементу присваиваю строку «*Привет*» (*r[7]:='Привет';)*.

Давай напишем какой-нибудь пример для работы с массивами. Допустим, нам надо узнать какой сегодня день недели. Я думаю, это будет полезный примерчик.

Создай новое приложение. Брось на форму один компонент *TEdit* (дадим ему имя *DayOfWeekEdit*) и одну кнопку (дадим ей имя *GetDayButton* и напишем в заголовке *«Узнать день недели*»). У меня получилась вот такая форма:



По нажатию кнопки мы будем узнавать, какой сегодня день недели и записывать результат в строку *TEdit*.

```
procedure TForm1.GetDayButtonClick(Sender: TObject);
var
 day:Integer;
 week: array[1..7] of string;
begin
 week[1] := 'Воскресенье';
 week[2] := 'Понедельник';
 week[3] := 'Вторник';
 week[4] := 'Среда';
 week[5] := 'Четверг';
 week[6] := 'Пятница';
 week[7] := 'Суббота';

 day:=DayOfWeek(Date);
 DayOfWeekEdit.Text:=week[day];
end;
```

Здесь я объявил массив *week* из семи элементов. После этого, я последовательно всем элементам массива присваиваю названия дней недель.

После этого, я узнаю, какой сегодня день недели. Для этого существует функция *DayOfWeek*. Ей нужно передать только один параметр – дату день недели которой нужно узнать. Я передаю результат работы функции *Date*, которая возвращает текущую дату. Получается, что *DayOfWeek* вернёт мне день недели текущей даты.

Вроде всё нормально, *DayOfWeek* возвращает не строку, в которой написано словами какой сегодня день, а число. Если функция возвращает 0, то это воскресенье, если 1 – понедельник, 2- вторник, 3 – среда и так далее. Как видишь, отсчёт идёт с воскресенья (по европейски). Точно так же я заполнял и массив: 1 – это воскресенье, 2 – понедельник и так далее.

После этого, нам надо превратить число в строку. Это делается очень просто. Нам надо получить только соответствующий элемент массива и всё. Если функция вернула нам 2, то это должен быть понедельник. В массиве под вторым номером тоже идёт

«Вторник», поэтому нам просто нужно получить строку находящуюся под вторым номером в массиве. Вот именно это и происходит в последней строке *week[day]*.

На компакт диске, в директории ***\Примеры\Глава 10\Arrays*** *ты можешь увидеть пример этой программы.*

Но это мы только закрепили на практике уже пройденный материал. Давай пойдём дальше и познакомимся с динамическими массивами.

Когда ты хочешь создать динамический массив, то не надо указывать его длину. Ты просто указываешь переменную, и её тип:

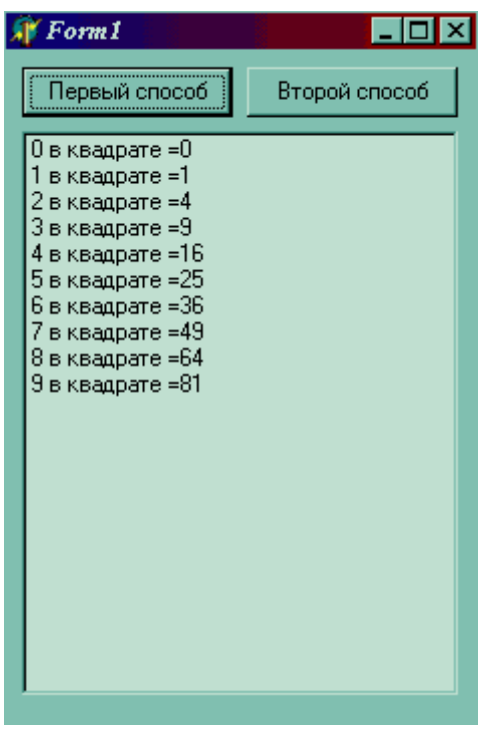
```
r: array of integer;
```

В этом примере я объявил переменную *r* типа массив целых чисел без указания размера (как мы указывали это в квадратных скобках [0..10]).

Чтобы указать размер массива можно воспользоваться функцией *SetLength*. У неё два параметра:

1. Переменная типа динамического массива.
2. Длина массива.

Давай посмотрим всё это на практике. Создай новый проект в Delphi, брось на форму две кнопки и один компонент *TListBox*:



Для первой кнопки мы напишем следующий текст:

```
var
 r:array of integer;
 i:Integer;
begin
 ListBox1.Items.Clear;
```

```
SetLength(r,10);

for i:=0 to High(r)-1 do
 begin
  r[i]:=i*i;
  ListBox1.Items.Add(IntToStr(i)+' в квадрате ='+IntToStr(r[i]));
 end;
```

В области объявлений *VAR* я объявил две переменные. Первая это r которая является массивом чисел типа *Integer*. Вторая *i* это переменная, которую я буду использовать в качестве счётчика. Переходим к самой процедуре:

Первая строка очищает все строки у *ListBox1*. Для этого вызывается процедура *ListBox1.Items.Clear*. Мы это уже проходили, но я напомню. У *ListBox1* есть свойство *Items*, где хранятся все строки. У *Items* есть метод *Clear*, который удаляет все находящиеся в нём строки.

Во второй строке вызывается процедура *SetLength*, которая выделила память для массива r (первый параметр), размером в 10 элементов (второй параметр). Обращение к элементом будет происходить как *r[номер_элемента]*. Элементы будут нумероваться от 0 до 9. Вообще, в программировании всё нумеруется с нуля.

Далее идёт цикл. Функция *High(r)* возвращает количество элементов в массиве r. В итоге получается, что цикл будет выполняться от i:=0 (от нуля), до количества элементов в массиве r минус 1 (до 9). Внутри массива выполняется две строки:

```
r[i]:=i*i .//Здесь i-му элементу массива присваивается i*i.
```

ListBox1.Items.Add(IntToStr(i)+' в квадрате ='+IntToStr(r[i])); Эта строка добавляет новый элемент в *ListBox1*. Функция *IntToStr* переводит число в строку.

С первой процедурой мы разобрались, теперь перейдём ко второй:

```
type
 TDynArr=array of integer;
var
 r:TDynArr;
 i:Integer;
begin
 ListBox1.Items.Clear;
 SetLength(r,10);

 for i:=0 to High(r)-1 do
  r[i]:=i*i;

 SetLength(r,20);
 for i:=10 to High(r)-1 do
  r[i]:=i*i;

 for i:=0 to High(r) do
  ListBox1.Items.Add(IntToStr(i)+' в квадрате ='+IntToStr(r[i]));
```

Эта процедура выполняет похожие действия, но с небольшими особенностями. В начале я объявляю новый тип: *TDynArr=array of integer*. После этого конструкция *r:TDynArr;* будет означать, что r относится к типу *TDynArr*, а тот относится к *array of integer;* . Это тоже самое, что мы писали в первой процедуре r:array of *integer*; , только такая конструкция удобней, если ты захочешь объявить несколько динамических массивов. Тебе не приходится сто раз писать громоздкую строку *r:array of integer;* , ты объявляешь новый массив как *TDynArr*.

Далее идёт всё та же очистка строк и выделение памяти под массив.

```
for i:=0 to High(r)-1 do
  r[i]:=i*i;
```

Эта конструкция заполняет десять элементов квадратами числа *i*. После этого я снова вызываю функцию *SetLength(r,20);*, в которой говорю, что массив теперь будет состоять из 20-и элементов. Таким способом можно как увеличивать количество элементов, так и уменьшать.

```
for i:=10 to High(r)-1 do
  r[i]:=i*i;
```

Здесь я заполняю квадратами числа i элементы начиная с 10 по последний. И в конце я снова заполняю *ListBox1* значениями элементов массива.

 *На компакт диске, в директории* ***\Примеры\Глава 10\DynArrays*** *ты можешь увидеть пример этой программы.*

10.2 Многомерные массивы.

Мы уже разобрались с массивами, но пока это только одномерные массивы, в которых данные располагаются в виде строки. Для Delphi это не предел и он может работать и с несколькими измерениями массива.

Например, допустим, что тебе надо держать таблицу из данных. Таблица будет состоять из пяти колонок и четырёх строк. В этом случае ты можешь завести четыре массива, в каждом из которых будут храниться по 5 элементов. Но это же не солидно!!! Вот тут на встречу приходят многомерные массивы.

Объявляются такие массивы практически так же как и одномерные, разница только в том, что когда ты в квадратных скобках указываешь длину массива, нужно указывать размеры строк и столбцов данных.

Рассмотрим пример объявления двухмерного массива из четырёх строк и пяти столбцов:

```
var
 t:array[0..3, 0..4] of integer;
```

Как видишь, в квадратных скобках перечислены через запятую размеры строк и столбцов. Заметь, что я объявил массив от 0 до 3 – это будет четыре элемента и от 0 до 4, что будет 5 элементов.

Работа с таким массивом тоже достаточно простая:

```
var
 t:array[0..3, 0..4] of integer;
begin
 t[0][0]:=1;
 t[1][0]:=2;
 t[2][0]:=3;
 t[3][0]:=4;
 t[1][1]:=5;
end;
```

После выполнения этого примера наша таблица будет иметь вид:

1 5 0 0 0
2 0 0 0 0
3 0 0 0 0
4 0 0 0 0

Двухмерность не предел и ты можешь создавать и 3-х мерные массивы. Давай молча посмотрим на следующий пример:

```
var
 t:array[0..3, 0..4, 0..2] of integer;
begin
 t[0][0][0]:=1;
 t[1][0][0]:=2;
 t[2][0][0]:=3;
 t[3][0][0]:=4;
 t[1][1][0]:=5;
end;
```

10.3 Работа с файлами.

В этой части я познакомлю тебя, как можно работать с файлами. Мы научимся открывать их, записывать и читать из них данные, ну и, конечно же, закрывать.

Для работы с файлами многие предпочитают использовать WinAPI. Не пугайся этого слова, потому что работа с WinAPI в Delphi очень даже прозрачна и ты не ощутишь никаких проблем. Я тоже любил так работать, пока не нарвался на одну неприятность. В самых первых окнах для чтения из файла использовалась функция _lread. В Windows 95 появилась ReadFile. А сейчас рекомендуют использовать *ReadFileEx*, которая может работать с файлами большего размера. После каждого изменения функций WinAPI приходится переделывать весь код проги, потому что нет гарантии, что старые функции будут работать в новых версиях Windows.

Вот поэтому я стал использовать специализированный в Delphi объект *TFileStream*. Я и тебе советую делать это, потому что если Microsoft снова введёт какие-то нововведения, то Borland учтёт их в объекте и тебе нужно будет только перекомпилировать свои проги. Никаких изменений в код вносить не надо. Ты один раз изменяешь объект (или это делает Borland) и компилируешь все свои проги с новыми возможностями. И в любом случае, использование объекта намного проще.

Итак, давай взглянём на объект *TFileStream*. Для работы с ним, ты должен объявить какую-нибудь переменную типа *TFileStream*.

```
var
 f: TFileStream;
```

Вот так я объявил переменную *f* типа объекта *TFileStream*. Теперь можно проинициализировать переменную.

Инициализация *– выделение памяти и установка значений по умолчанию.*

За инициализацию в любом объекте отвечает метод *Create*. Нужно просто вызвать его и результат выполнения присвоить переменной. Например, в нашем случае нужно вызвать TFileStream.Create(какие-то параметры) и результат записать в переменную *f*.

```
f := TFileStream.Create(параметры);
```

Давай разберёмся, какие параметры могут быть при инициализации объекта *TFileStream*. У этого метода *Create* может быть три параметра, причём последний можно не указывать:

1. Имя файла (или полный путь к файлу) который надо открыть. Этот параметр – простая строка.
2. Режим открытия. Здесь ты можешь указать один из следующих флагов:
    a. *fmCreate* – создать файл с указанным в первом параметре именем. Если файл уже существует, то он откроется в режиме для записи.
    b. *fmOpenRead* – открыть файл только для чтения. Если файл не существует, то произойдёт ошибка. Запись в файл в этом случае не возможна.
    c. *fmOpenWrite* – открыть файл для запись. При этом, во время записи текущее содержимое уничтожается.
    d. *fmOpenReadWrite* – открыть файл для редактирования (чтения и записи).
3. Права, с которыми будет открыт файл. Тут опять ты можешь указать одно из следующих значений (а можешь вообще ничего не указывать):
    a. *fmShareCompat* – при этих правах, другие приложения тоже имеют права работать с открытым файлом.
    b. *fmShareExclusive* – при этом режиме другие приложения вообще не смогут открыть файл.
    c. *fmShareDenyWrite* – при данном режиме другие приложения не смогут открывать этот файл для записи. Файл может быть открыт только для чтения.
    d. *fmShareDenyRead* – при данном режиме другие приложения не смогут открывать этот файл для чтения. Файл может быть открыт только для записи.

e. *fmShareDenyNone* - вообще не мешать другим приложениям работать с файлом.

С первыми двумя параметрами всё ясно. Но зачем же нужны права доступа к файлам. Допустим, что ты открыл файл для записи. Потом его открыло другое приложение и тоже для записи. Вы оба записали какие-то данные. После этого твоё приложение закрыло файл и сохранило все изменения. Тут же другое приложение перезаписывает твои изменения, даже не подозревая о том, что они прошли. Вот так твоя инфа пропадает из файла.

Если ты пишешь однопользовательскую прогу и к файлу будет иметь доступ только одно приложение, то про права можно забыть и даже не указывать.

После того, как ты поработал с файлом, достаточно вызвать метод *Free*, чтобы закрыть файл.

```
f.Free;
```

Теперь давай познакомимся с методами чтения, записи и путешествия внутри файла. Начнём с путешествия. Когда ты открыл файл, позиция курсора устанавливается в самое начало и любая попытка чтения или записи будет происходить в эту позицию курсора. Если тебе надо прочитать или записать в любую другую позицию, то надо передвинуть курсор. Для этого есть метод *Seek*. У него есть два параметра:

1. Число, указывающее на позицию, в которую надо перейти. Если тебе нужно передвинутся на пять байт, то просто поставь цифру 5.

2. Откуда надо двигаться. Тут возможны три варианта:

- *soFromBeginning* – двигаться на указанные количество байт от начала файла.
- *soFromCurrent* - двигаться на указанные количество байт от текущей позиции в файле к концу файла.
- *soFromEnd* – двигаться от конца файла к началу на указанное количество байт.

Не забывай, что один байт – это один символ. Единственное исключение – файлы в формате Unicode. В них один символ занимает 2 байта. Так что надо учитывать в каком виде хранится информация в файле.

Итак, если тебе надо передвинутся на 10 символов от начала файла, то ты можешь написать следующий код:

```
f.Seek(10, soFromBeginning);
```

Метод *Seek* всегда возвращает смещение курсора от начала файла. Этим можно воспользоваться чтобы узнать где мы сейчас находимся, а можно и узнать общий размер файла. Если переместится в конец файла, то функция вернёт нам количество байт от начала до конца, т.е. полный размер файла.

В следующем примере я устанавливаю позицию в файле на 0 байт от конца файла, т.е. в самый конец. Тем самым я получаю полный размер файла:

```
Размер файла := f.Seek(0, soFromEnd);
```

Для чтения из файла нужно использовать метод Read. . И снова у этого метода два параметра:

1. Переменная, в которую будет записан результат чтения.
2. Количество байт, которые надо прочитать.

Давай взглянём на код чтения из файла, начиная с 15 позиции в файле.

```
var
 f:TFileStream; //Переменная типа объект TFileStream.
 buf: array[0..10] of char; // Буфер, для хранения прочитанных данных
begin
 // В следующей строчке я открываю файл filename.txt для чтения и записи.
 f:= TFileStream.Create('c:\filename.txt', fmOpenReadWrite);

 f.Seek(15, soFromCurrent); // Перемещаюсь на 15 символов вперёд.
 f.Read(buf, 10); // Читаю 10 символов из установленной позиции.
 f.Free; // Уничтожаю объект и соответственно закрываю файл.
end;
```

Заметь, что в методе *Seek* я двигаюсь на 15 символов не от начала, а от текущей позиции, хотя мне нужно от начала. Это потому что после открытия файла текущая позиция и есть начало.

Метод *Read* возвращает количество реально прочитанных байт (символов). Если не произошло никаких проблем, то это число должно быть равно количеству запрошенных для чтения байт. Есть только два случая, когда эти числа отличаются:

1. При чтении был достигнут конец файла и дальнейшее чтение стало не возможным.
2. Ошибка на диске или любая другая физическая проблема.

Осталось только разобраться с методом для записи. Для записи мы будем использовать *Write*. У него так же два параметра.

1. Переменная, содержимое которой нужно записать.
2. Число байт для записи.

Пользоваться этим методом можно точно также как и методом для чтения.

Напоследок одно замечание: после чтения или записи текущая позиция в файле смещается на количество прочитанных байт. То есть текущая позиция становится в конец прочитанного блока данных.

Примера пока не будет, потому что в принципе и так всё ясно. А если есть вопросы, то на практике мы закрепим этот материал буквально через один или два раздела, когда будем работать с структурами.

10.4 Работа с текстовыми файлами

В предыдущей части я показал общий случай работы с файлами. В этой части мы познакомимся с частным случаем – текстовые файлы. В них информация расположена не сплошным одинарным блоком, а в виде строк текста. Было бы удобно воспринимать такие файлы в виде наборов строк.

Если мы попытаемся прочитать текстовый файл методами, описанными в прошлой части, то работать с текстом будет неудобно. Допустим, что у нас есть файл из двух строчек:

Привет!!!
Как жизнь?

Если прочитать его с помощью объекта TFileStream, то мы увидим весь текст в одну строку:

*Привет!!!**<CR><LF>**Как жизнь?*

Здесь *<CR>* - конец строки и *<LF>* - перевод каретки на новую строку. Так что, чтобы найти конец первой строки мы должны просканировать весь текст на наличие признака конца строки и перевода каретки (CR и LF). Это очень неудобно, каждый раз сканировать конец строки. А допустим, что у тебя файл из 100 строк и тебе нужно получить доступ к 75-й строке. Как ты думаешь, долго мы будем искать нужную строку? Нет, точнее сказать, что не долго, а неудобно.

Тут на помощь нам приходит объект TStrings, который является простым контейнером (хранилищем) для строк. Можно ещё пользоваться более продвинутым вариантом этого объекта TStringList. Если посмотреть на иерархию объекта TStringList (рис 1), то мы увидим, что TStringList происходит от TStrings. Это значит, что TStringList наследует себе все возможности объекта TStrings и добавляет в него новые.

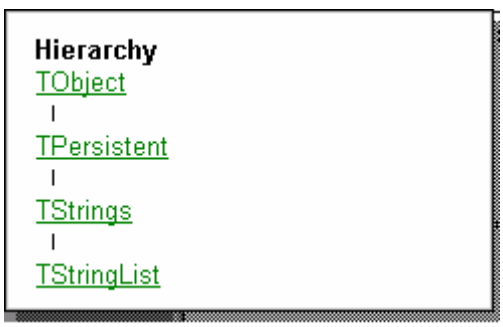


Рис 1Иерархия объекта TStringList

Как всегда для работы с объектом надо объявлять переменную типа объект:

```
var
 f:TStringList; //Переменная типа объект TStringList.
```

Инициализируется эта переменная как всегда методом *Create*. Никаких параметров не надо. Чтобы освободить память объекта и уничтожить его, как всегда применяется метод *Free*. Вот пример простейшего использования объекта:

```
var
 f:TStringList; //Переменная типа объект TStringList.
begin
 f:= TStringList.Create();
 f.Free;
end;
```

В этом примере я только создал новый объект и сразу уничтожил, не используя его.

Давай снова вспомним, что *TStringList* происходит от *TStrings*. Использовать *TStrings* напрямую нельзя, потому что это абстрактный объект. *Абстрактный объект* – объект, который представляет из себя пустой шаблон. Он может даже ничего не уметь делать, а только описывать какой-то вид или шаблон, на основе которого можно выводить полноценные объекты. Вот так *TStringList* добавляет в *TStrings* свои функции так, что он становиться полноценным объектом.

Итак, получается, что мы не можешь объявлять переменные типа *TStrings* и использовать этот объект, потому что это всего лишь шаблон. Это и так и не так. Переменную мы можем объявлять, но использовать сам объект не можем. Зато мы можем объявить переменную типа *TStrings*, но использовать эту переменную как объект *TStringList*, потому что он происходит от первого. Это значит, что следующий пример идентичен предыдущему:

```
var
 f:TStrings; //Переменная типа объект TStringList.
begin
 f:= TStringList.Create();
 f.Free;
end;
```

В этом примере я объявил переменную типа *TStrings*, но при создании проинициализировал её объектом *TStringList*. Это вполне законная запись, потому что объект *TStringList* происходит от *TStrings*. Новая переменная f будет работать, как объект *TStringList*, хотя и объявлена как *TStrings*. Главное – каким объектом переменная проинициализирована.

Такие трюки можно проводить только с родственными объектами, но я постараюсь не использовать их в своей книге, хотя знать ты всё это обязан. А вдруг тебе попадётся исходник трюкача и тогда такая запись может тебя запутать.

Итак, давай познакомимся, как можно работать с помощью *TStringList* с текстовыми файлами. Всё очень просто. У него есть метод *LoadFromFile* для которого нужно указать имя текстового файла. После этого, через свойство *Strings* можно получить доступ к любой строчке, а в свойстве *Count* находиться число указывающее на количество строк в файле.

```
var
 f:TStrings; //Переменная типа объект TStringList.
begin
 f:= TStringList.Create();
 f.LoadFromFile('c:\filename.txt');// Загружаю текстовый файл
 f.Strings[0]; // Здесь находиться первая строчка файла
 f.Strings[1]; // Здесь находиться вторая строчка файла

 f.Free;
end;
```

Давай напишем пример, который будет искать в файле строку *«Привет Вася»*:

```
var
 f:TStrings; //Переменная типа объект TStringList.
 i: Integer; // Счётчик
begin
 f:= TStringList.Create();
 f.LoadFromFile('c:\filename.txt'); // Загружаю текстовый файл

 for i:=0 to f.Count-1 do // Запускаю цикл
  begin // Начало для цикла

   if f.Strings[I] = 'Привет Вася' then //Если i-я строка равна нужной то

   Application.MessageBox('Строка найдена',
       'Поиск закончен', MB_OKCANCEL)

  end; // Конец для цикла

 f.Free;
end;
```

Точно так же очень просто изменять данные в файле. Например, тебе надо изменить 5-ю строку на «*Прощай станция Мир*». Это можно сделать следующим образом:

```
var
 f:TStrings; //Переменная типа объект TStringList.
 i: Integer; // Счётчик
begin
 f:= TStringList.Create();
 f.LoadFromFile('c:\filename.txt'); // Загружаю текстовый файл

 if f.Count>=5 then // Если в файле есть 5 строк то изменить
   f.Strings[5] = 'Прощай станция Мир';

 f.Add('Прощай');// Добавляю новую строку

 f.SaveToFile('c:\filename.txt'); // Сохраняю результат
 f.Free;
end;
```

На всякий случай, прежде чем изменить пятую строку я проверяю, есть ли в файле эти пять строк. Если окажется меньше пяти, то при попытке изменения данных произойдёт ошибка.

В этом же примере я добавляю в конец файла новую строку с помощью вызова метода Add. После этого я сохраняю результат в том же файле с помощью вызова метода *SaveToFile*. Если не вызывать метод сохранения, то все изменения пропадут, потому что данные изменяются в объекте, а не в файле, поэтому объект надо сохранять обратно в файл.

На этом можно закончить рассмотрение объекта, но я хочу ещё показать тебе несколько методов:

1. *Clear* – очистка содержимого объекта.
2. *Insert* – вставить строку. У этого метода два параметра – индекс строки куда нужно вставить и сама строка.

3. *Delete* – удалить строку. Здесь только один параметр – индекс удаляемой строки.

Вот теперь можно считать, что с текстовыми файлами и объектом *TStringList* покончено. Можно двигаться дальше.

10.5 Приведение типов

Сейчас я постараюсь, как можно подробнее остановиться на теме приведения типов. В любой программе может понадобиться преобразование данных из одного типа в другой.

Преобразование типов делиться на два вида: преобразование несовместимых типов и преобразование совместимых типов. В качестве несовместимых типов можно привести пример превращения строки в число. Допустим, что у тебя есть строка «*12345*». Это строка, содержащее число. Но ты не можешь производить с такой строкой математических действий, потому что это строка, хотя и содержащее число. Для начала нужно преобразовать эту строку в число.

Преобразование целых чисел в строку и обратно

Начну я с рассмотрения специальных функций для преобразования несовместимых типов. Самое частое, что может тебе понадобиться – преобразование строк в число и обратно. Допустим, что тебе нужно написать программу, в которой пользователь будет вводить число в компонент *TEdit*. Чтобы получить доступ к содержимому *Edit1* надо написать *Edit1.Text*. Так мы получим текстовое представление числа. Чтобы его преобразовать, необходимо воспользоваться спец функцией. Вот давай и будем знакомиться с подобным примером.

Для преобразования строки в число используется функция *StrToInt*. У неё только один параметр – строка, а на выходе она возвращает число.

```
var
 ch:Integer;
begin
 ch:=StrToInt(Edit1.Text); // Преобразовываю Edit1.Text  в число
end;
```

В этом примере я присвоил в переменную *ch* значение, содержащееся в *Edit1.Text* преобразованное в число. Теперь мы можешь производить математические действия с введённым числом.

Обратное преобразование (превращение числа в строку) можно произвести с помощью функции *IntToStr*.

```
var
 ch:Integer;
begin
 ch:=StrToInt(Edit1.Text); // Преобразовываю Edit1.Text  в число
 ch:=ch+1;
 Edit1.Text:=IntToStr(ch); // Преобразовываю ch  в строку
end;
```

Когда ты преобразовываешь строку в число, ты должен быть уверен в том, что строка содержит число. Если в строке будет хоть один символ не относящейся к цифре, то во время преобразования произойдёт ошибка. Чтобы избавить от ошибок, можно использовать исключительные ситуации, заключая преобразование между *try* и *except*. Но есть ещё один способ – использовать функцию *StrToIntDef* у которой уже два параметра:

1. Строка, которую надо преобразовать
2. Значение по умолчанию, которое будет возвращено, если произошла ошибка.

Итак, наш пример можно подкорректировать следующим образом:

```
var
 ch:Integer;
begin
 ch:=StrToIntDef(Edit1.Text, 0); // Преобразовываю Edit1.Text  в число
end;
```

В этом примере, если произойдёт ошибка во время преобразование, то функция не будет ругаться, а вернёт значение 0.

Преобразование даты в строку и обратно

Теперь познакомимся с преобразованием даты. Для этого есть несколько функций:

1. *DateToStr* – преобразовывает дату в строку. Единственный параметр, который надо указать – переменную типа *TDateTime* и на выходе получим строку.
2. *StrToDate* – преобразование строки в дату. Указываешь строку (например «*11/05/2001*»)и получаешь дату.
3. *FormatDateTime* – форматирование даты и времени. Это очень интересная функция, поэтому на ней я остановлюсь подробнее.

У функции *FormatDateTime* два параметра:

1. Формат строки в которую надо перегнать дату
2. Переменная типа *TdateTime*, которую надо преобразовать.

Самое интересное здесь – это формат строки. Он может содержать следующие символы:

---

d – показать дату не подставляя нули в начале (1, 2, 3 …30, 31).

dd – показать дату подставляя если нужно в начале ноль. В этом случае, если дата меньше 10, то она будет отражаться как 01, 02 … 09.

ddd – показать день недели используя короткий формат (Пн, Вт, Ср…).

dddd – показать день недели с полным названием (Понедельник, Вторник …)

ddddd – показать дату используя короткий формат.

dddddd – показать дату используя полный формат (Например 10 дата /02/2002 будет переведена в «*10 февраля 2002*».

m – показать месяц без добавления нулей (1, 2, …, 11, 12).

mm – показать месяц с добавлением нулей (01, 02, …11, 12).

mmm – показать короткое название месяца.

mmmm – показать полное название месяца (январь, февраль.…).

yy – показать короткий года (98, 99, 00, 01).

yyyy – показать полный год.
h – показать часы не добавляя в начале нулей.
hh – показать часы с добавлением в начале нулей.
n – показать минуты не добавляя в начале нулей.
nn – показать минуты с добавлением в начале нулей.
s – показать секунды не добавляя в начале нулей.
ss – показать секунды с добавлением в начале нулей.
z – показать миллисекунды не добавляя в начале нулей.
zz – показать миллисекунды с добавлением в начале нулей.
am/pm – использовать 12-и часовое представление (до полудня/после полудня).

Это практически полный обзор возможностей, а теперь посмотрим пару примеров:

```
FormatDateTime('dd/mm/yyy', Date()); // Дата будет в виде "24/02/2002"
FormatDateTime('dddddd', Date()); // Дата будет в виде "24 февраля 2002"
FormatDateTime('hh:nn', Time()); // Время будет в виде "10:48"
FormatDateTime('hh:nn - ss', Time()); // Время будет в виде "10:48 - 24"
```

Преобразование вещественных чисел

Теперь перейдём к числам с плавающей точкой. Когда ты строишь математику в своей программе, то можешь столкнуться с вещественными числами. Например, если у тебя есть какая-то формула, в которой используется деление, то результат её выполнения будет всегда дробным, даже если ты уверен в целостности ответа. Например, ты делишь 10 на 2, и должен получить результат 5. Хотя результат целое число, компилятор будет представлять его как дробное.

```
var
 i:Integer;
begin
 i:=10/2;
end;
```

Если ты попытаешься откомпилировать такой код, то увидишь следующую ошибку: «*Incompatible types: 'Integer'and 'Extended'*»

Тут появляется два выхода:

1. Записывать результат в переменную вещественного типа. Но он подходит не всегда, поэтому лучше перейти сразу ко второму методу.
2. Округлять результат

Для округления существует очень удобная функция *round*:

```
var
 i:Integer;
begin
 i:=round(10/2);
end;
```

Если ты решил хранить результат в переменной вещественного типа, то могут возникнуть проблемы с выводом результата. Для этого может понадобиться

преобразование вещественного числа в строку. Для этого есть функция *FloatToStr*, которой надо передать дробное число и получить строку. Точно так же есть и обратное преобразование *StrToFloat*, где ты передаешь строку, а получаешь вещественное число.

Отдельного разговора требует функция *FormatFloat*, которая форматирует вещественное число по твоим нуждам. Тут есть два параметр: строка формата и само число.

Следующая табличка показывает разные варианты формата. В первой колонке показаны возможные форматы указываемые в первом параметре функции *FormatFloat*. В остальных колонках показано, что произойдёт с разными числами при данном формате (табличка взята из файла помощи по Delphi):

Строка указываемая в формате	**Форматируемые числа**			
	1234	-1234	0.5	0
0	1234	-1234	1	0
0.00	1234.00	-1234.00	0.50	0.00
#.##	1234	-1234	.5	
#,##0.00	1,234.00	-1,234.00	0.50	0.00
#,##0.00;(#,##0.00)	1,234.00	(1,234.00)	0.50	0.00
#,##0.00;;Zero	1,234.00	-1,234.00	0.50	Zero
0.000E+00	1.234E+03	-1.234E+03	5.000E-01	0.000E+00
#.###E-0	1.234E3	-1.234E3	5E-1	0E0

Вот пока что и всё, что я хотел сказать тебе про преобразование несовместимых типов.

10.6 Преобразование совместимых типов

Теперь можно познакомиться и с несовместимыми типами. Под совместимыми типами я понимаю типы данных, которые схожи по своим признакам, но хранят данные в разном виде. Например, есть несколько видов строк: строка, оканчивающаяся нулём и строка, первый байт которой указывает на её длину. Вроде и то и другое строки, но если где-то нужен определённый тип строки, то придётся преобразовывать свою строку именно в тот формат.

Преобразование строк

Допустим, что у тебя есть строка типа *String* и ты хочешь её преобразовать в *PChar*. Для такого преобразования нужно написать требуемый тебе тип и в скобках указать свою строковую переменную: *NewStr:=PChar(MyStr);*. В этом примере переменная *MuStr* имеет тип *String*, а мы приводим её в вид *PChar*. Вот так происходит преобразования совместимых типов.

Вот такое преобразование строк мы будем делать очень часто при вызове WinAPI функций, потому что там большинство строк имеет именно *PChar* тип.

Сейчас я не могу придумать какой-нибудь ещё пример преобразования, но уже в этой главе ты воспользуешься такой возможностью в одном из моих примеров.

10.7 Указатели

Пора познакомиться с указателями. Это очень удобная и сильная вещь, которой мы так же будем часто пользоваться. Уже в этой главе мы будем работать с указателями на структуры, которые расположим в динамической памяти.

Но прежде чем что-то объяснять, я хочу показать тебе, зачем нужны указатели. Давай вспомним про процедуры, и я расскажу тебе, как происходит их вызов. Допустим, у тебя есть процедура с именем *MyProc* у которой есть два параметра: число и строка. Как происходит вызов такой процедуры, и как ей передаются эти параметры? Очень просто. Сначала параметры поднимаются в стек (напомню, что стек – это область памяти для хранения твоих временных/локальных переменных). Сначала заносится первый параметр, затем второй и после этого вызывается процедура. Прежде чем процедура начнёт своё выполнение, она достаёт эти параметры из стека в обратном порядке.

Теперь вспомним о наших параметрах. Первый – это число, которое будет занимать два байта. Когда мы подымем его в стек, то оно займёт там свои положенные два байта. Второй параметр – строка. Каждый символ строки – это отдельный байт. Допустим, что наша строка состоит из 10 символов, значит для передачи такой строки в процедуру, в стеке понадобиться 10 байт плюс один байт для указания конца строки или размерности (это зависит от типа строки). Всего для передачи в процедуру нам понадобиться в стеке как минимум 12 байт. Это не так уж и много, поэтому такое можно себе позволить.

А теперь представь, что строка, которую надо передать в процедуру, состоит из 1000 символов. Вот тут нам понадобиться в стеке уже около килобайта. При нынешних размерах памяти на это уже никто не обращает внимания, но народ забывает про то, что такая строка сначала копируется в память стека, а потом достаётся оттудова. Такое копирование большого размера памяти отнимает достаточно много времени и твоя программа тратит лишнее время на бессмысленное копирование большой строки.

Выход из сложившейся ситуации достаточно прост – можно не передавать строку, а только передать указатель на область памяти, где находиться эта строка. Любой указатель занимает всего четыре байта, а это уже существенная экономия. Мы просто передаём два байта указывающих на длинную строку, а процедура достаёт эти два байта и уже знает, где можно взять строку, когда она понадобиться.

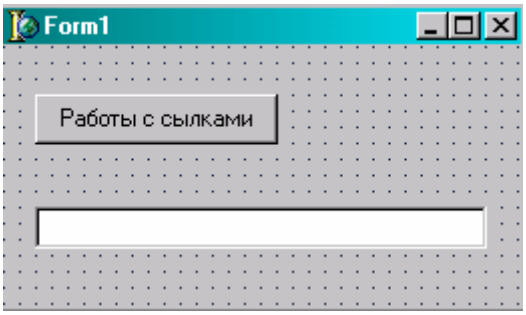
Указатель в Delphi объявляется как *Pointer*. Например, давай объявим переменную *p* типа указатель:

```
var
 p:Pointer
```

Для того, чтобы получить адрес переменной или объекта, необходимо перед его именем поставить знак **@**. Например, у тебя есть строка ***Str*** и чтобы присвоить её адрес в указатель ***p***, надо выполнить следующее: ***p:=@Str;***. Теперь в указателе находиться адрес строки. Если ты будешь напрямую читать указатель, то увидишь адрес, а для того чтобы увидеть содержащиеся по этому адресу данные, надо разыменовывать указатель. Для этого надо написать ***p^***. Итак, мы пришли к следующему:

p:=@Str – получить адрес строки.
p – указатель на строку.
p^ - данные, содержащиеся по адресу указанному в ***p***.

Давай создадим маленькое приложение, которое будет работать с указателями. Для этого создай форму приблизительно следующего содержания:



По нажатию кнопки «Работа со ссылками» напиши следующее:

```
var
 p:Pointer
 Str:String;
begin
 p:=@Str; // Присваиваю указателю ссылку на строку
 Str:='Привет мой друг'; // Изменяю значение строки
 Edit1.Text:=String(p^); // Вывожу текст
end;
```

В этом примере, в первой строке я присваиваю указателю ***p*** ссылку на строку *Str*. После этого я меняю содержимое строки. И в последней строчке я вывожу содержащийся по адресу ***p*** текст. Для этого приходиться явно указывать, что по адресу ***p*** находиться именно строка *String(p^)*. Это необходимо, потому что данные, расположенные по определённому адресу могут иметь совершенно любой тип. Как видишь, жёсткое указание типа похоже на преобразование типов, поэтому никаких проблем с этим не должно возникнуть.

Заметь, что я изменяю строку после присваивания адреса строки в переменную указатель, и изменённые данные всё равно будут отражены в указателе. Это потому что указатель всегда показывает на начало строки, и если мы её изменим, указателю будет всё равно, потому что новые данные будут расположены по тому же адресу, и ***p*** будет указывать на новую строку.

На компакт диске, в директории ***\Примеры\Глава 10\Pointers*** *ты можешь увидеть пример этой программы.*

Мы уже не раз использовали указатели, просто не углублялись в их изучение. Каждая переменная типа объекта – это тоже указатель на объект. Просто его использование выполнено так, чтобы не смущать тебя адресацией и разыменовыванием.

Любой переменной указателю можно присвоить нулевое значение, только это не 0, а **nil**, например p:=nil;. Когда ты присваиваешь нулевое значение, то ты как бы уничтожаешь ссылку. Точно так же, если ты переменной объекту присвоишь нулевое значение, ты его уничтожишь, и объект больше не будет существовать.

А что если у тебя две переменные указывают на один и тот же адрес данных? Неужели при уничтожении одного из них данные уничтожаться и вторая переменная будет указывать на несуществующие данные? Нет, объект будет уничтожен только после того, как все указатели на него будут уничтожены.

10.8 Структуры, записи

Сейчас я хочу тебя познакомить поближе со структурами и записями. Точнее сказать, оба понятия означают одно и тоже. Просто в C++ принято говорить «структура», а в Delphi говорят «запись». Так как я знаю оба языка, я больше привык к понятию «структура», потому что оно больше отражает смысл этого понятия. Тебе нужно только привыкнуть, что структура – это та же запись.

Я уже упоминал о структурах немного раньше. Они похожи на объекты, только не имеют методом и событий, а только свойства.

Для объявления структуры используется следующий вид:

```
Имя структуры = record
 Свойство1: Тип свойства1;
 Свойство2: Тип свойства2;
 ....
end;
```

Давай опишем структуру, в которой будут храниться параметры окна:

```
WindowsSize = record
 Left:Integer;
 Top:Integer;
 Width:Integer;
 Height:Integer;
end;
```

Я назвал новую структуру как *WindowSize* и объявил внутри четыре свойства: Left, Top, Width, Height.

Свойства я объявлял каждое в отдельности, хотя в данном случае все они имеют один тип и их можно просто перечислить через запятую и указать, что все они имеют целый тип:

```
WindowsSize = record
 Left, Top, Width, Height:Integer;
end;
```

Точно так же можно объявлять и переменные в разделе **var**, когда несколько переменных имеют один и тот же тип.

Теперь давай разберёмся, как можно использовать нашу структуру. Для этого надо определить переменную типа структура:

```
var
 ws: WindowSize;
```

Структуры – это простые сгруппированные наборы свойств, которые по умолчанию не требуют выделения памяти. Они создаются локально в стеке. Напоминаю, что стек – область памяти для хранения локальных/временный переменных. Так же следует напомнить чем отличаются локальные переменные от глобальных:

Локальные переменные – объявляются и создаются при входе в процедуру и уничтожаются при выходе. *Глобальные переменные* – создаются при запуске программы и уничтожаются при выходе из неё. Это значит, что глобальные переменные существуют на протяжении всего времени выполнения программы. Локальные переменные существуют, только когда выполняется код процедуры. После первого выполнения процедуры локальные переменные уничтожаются и при следующем вызове создаются снова с нулевым значением. Поэтому, если надо сохранить значение переменной после выхода из процедуры, её следует объявить как глобальную.

Итак, переменная объявлена. Инициализация и уничтожение не требуется, поэтому можно сразу же её использовать. Для доступа к переменным структуры нужно написать имя структуры и через точку указать тот параметр, который тебя интересует. Например, для доступа к параметру *Left* необходимо написать: *ws.Left*.

Давай напишем пример, который будет после закрытия программы сохранять текущие значения позиции окна в структуре, а потом эту структуру будем записывать в файл. Для записи будет использоваться простой бинарный файл, значит, воспользуемся объектом TFileStream.

Итак, создай новое приложение. В разделе **Type** опиши нашу структуру:

```
type
  WindowsSize = record
   Left, Top, Width, Height:Integer;
  end;

  TForm1 = class(TForm)
  private
  public
  end;
```

Теперь создай обработчик события *OnClose* для формы. Здесь мы заполним структуру значениями позиции окна и сохраним в бинарный файл:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
var
  ws:WindowsSize;
  f:TFileStream;
  Str:String;
begin
  ws.Left:=Left; // Заполняем левую позицию
  ws.Top:=Top; // Заполняем правую позицию
  ws.Width:=Width; // Заполняем ширину окна
  ws.Height:=Height; // Заполняем высоту окна

  f:=TFileStream.Create('size.dat', fmCreate); // Создаю файл Size.dat
  f.Write(ws, sizeof(ws)); // Записываю структуру
  f.Free; // Закрываю файл
end;
```

Обрати внимание, что при записи в файл, я должен указывать в качестве параметра буфер памяти для записи. Я указываю свою структуру. В качестве второго параметра нужно указывать размер записываемых данных. Для определения размера структуры я использую функцию *SizeOf*, которая вернёт мне необходимый размер.

Ну а теперь разберёмся с чтением из файла, которое происходит подобным образом:

```
procedure TForm1.FormShow(Sender: TObject);
var
 ws:WindowsSize;
 fs:TFileStream;
begin
 if FileExists('size.dat') then // Если файл Size.dat существует, то
  begin
   fs:=TFileStream.Create('size.dat', fmOpenRead); // Открываю файл Size.dat
   fs.Read(ws, sizeof(ws)); // Читаю содержимое структуры
   fs.Free; // Закрываю файл

// Дальше я устанавливаю сохранённые размеры и позицию окна на родину
   Left:=ws.Left;
   Top:=ws.Top;
   Width:=ws.Width;
   Height:=ws.Height;
  end;
end;
```

Первым делом я вызвал функцию *FileExists* которой указал имя интересующего меня файла. Эта функция проверила на существование файл. Если он существует, то я могу попытаться прочитать иначе это делать бесполезно. При чтении я так же читаю сразу всю структуру.

Как видишь, структуры очень удобны для хранения каких-либо структурированных данных. Конечно же, пример не очень удачный, потому что такие параметры лучше сохранять в реестре, а не в файле, но главное – это сам процесс. Я буду часто использовать структуры там, где это необходимо, потому что они действительно упрощают процесс кодинга.

 *На компакт диске, в директории* ***\Примеры\Глава 10\Records*** *ты можешь увидеть пример этой программы.*

10.9 Храним структуры в динамической памяти

Структуры могут быть не только локальными (храниться в стеке) но и динамическими (располагаться в динамической памяти). Почему память называется динамической? Да потому что стек создаётся автоматически при старте проги, а вот дополнительную память нужно выделять самому. Её можно добавлять и удалять в процессе работы проги, наверно поэтому её называют динамической.

Когда объявляешь структуру, то можешь указать и её динамический тип. Для этого нужно объявить ещё одну переменную и присвоить ей «*^ИмяСтруктуры*». Чаще всего в качестве нового имени используют то же самое имя, только в начале добавляют букву «P» и объявление это делают прямо перед объявлением структуры:

```
type
 PWindowsSize = ^ WindowsSize;
 WindowsSize = record
  Left, Top, Width, Height:Integer;
 end;
```

В этом примере *PWindowsSize* - ссылка на структуру. Теперь, чтобы разместить нашу структуру не в стеке, а в динамической памяти мы должны использовать именно *PwindowsSize*:

```
var
 ws:PWindowSize;
begin
 ws:=New(PWindowSize); // Выделяем память
 ws.Left:=10; // Изменяем одно свойство
 Dispose(ws); // Уничтожаем память
end;
```

В этом примере я объявил переменную *ws* типа *PWindowsSize*. Это значит, что *ws* – это всего лишь указатель и в самом начале он нулевой. Теперь нам надо этому указателю выделить память размером со структуру *PWindowsSize*. Для этого ей надо присвоить результат работы функции *New*. Эта функция выделяет динамическую память под указанный в качестве параметра объект и возвращает указатель на эту память. После этого в указателе *ws* находиться выделенная память, подготовленная для использования в качестве структуры *PWindowsSize*.

Доступ к свойствам остаётся такой же, поэтому нет смысла задерживаться на этом. Но вот в глаза сразу же бросается вызов функции *Dispose*. Так как мы выделили динамическую память, её нужно освободить и для этого служит именно эта функция. Просто передай ей в качестве параметра указатель, и функция корректно обнулит его.

Помни, что если ты объявил переменную типа указатель на структуру (в нашем примере это *PWindowsSize*), то для такого указателя обязательно нужно сначала выделить память и потом освободить его. Если ты объявляешь переменную типа структура (в нашем примере это *WindowsSize*), а не указатель, то такая структура автоматически расположиться в стеке и ничего не надо выделять и освобождать.

10.10 Поиск файлов

В этой главе мы уже узнали о работе с файлами и узнали, что такое структуры и как с ними работать. Сейчас я хочу тебе показать, как можно организовать поиск файлов. В этом примере мы закрепим большинство навыков описанных в этой главе.

Для начала разберёмся с алгоритмом поиска файлов, а потом подробно рассмотрим каждую из необходимых функций:

```
//Запускаю поиск
hFindFile := FindFirst(Маска поиска, Аттрибуты , Информация);
//Проверяю корректность найденного файла
if hFindFile <> INVALID_HANDLE_VALUE then
//Если корректно, то запускаеться цикл repeat - until.
 repeat
```

```
  //Здесь вписаны операторы, которые нужно выполнять.
 until (FindNext(Информация) <> 0);
FindClose(Информация);
```

FindFirst - открывает поиск. В качестве первого параметра выступает маска поиска. Если ты укажешь конкретный файл, то система найдёт его. Но это не серьёзно, лучше искать более серьёзные вещи. Например, ты можешь запустить поиск всех файлов в корне диска C. Для этого первый параметр должен быть 'C:*.*'. Для поиска только файлов EXE, в папке Fold ты должен указать 'C:\Fold*.exe'.

Второй параметр - атрибуты включаемых в поиск файлов. Я использую *faAnyFile*, чтобы искать любые файлы. Тебе доступны

faReadOnly - искать файлы с атрибутом *ReadOnly* (только для чтения).
faHidden - искать скрытые файлы.
faSysFile - искать системные файлы.
faArchive - искать архивные файлы.
faDirectory - искать директории.

Последний параметр - это структура, в которой нам вернётся информация о поиске, а именно имя найденного файла, размер, время создания и т.д. После вызова этой процедуры, я проверяю на корректность найденного файла. Если всё в норме, то запускается цикл **Repeat** - **Until**.

Мы уже рассматривали с тобой циклы, но я всё же решил повториться и напомнить тебе работу используемого мной цикла. Он выполняет операторы, расположенные между **repeat** и **until**, пока условие расположенное после слова **until** является верным. Как только условие нарушается, цикл прерывается.

Хочу предупредить, что функция поиска, может возвращать в качестве найденного имени в структуре *SearchRec* (параметр *Name*) точку или две точки. Если ты посмотришь на директорию, то таких файлов не будет. Откуда берутся эти имена? Имя файла в виде точки указывает на текущую директорию, а имя файла из двух точек указывает на директорию верхнего уровня. Если я встречаю такие имена, то я их просто отбрасываю.

Структура

```
type
TSearchRec = record
  Time: Integer; // Время создания найденного файла
  Size: Integer; // Размер найденного файла
  Attr: Integer; // Атрибуты найденного файла
  Name: TFileName; // Имя найденного файла
  ExcludeAttr: Integer; // Исключаемые атрибуты найденного файла
  FindHandle: THandle; // Указатель необходимый для поиска
  FindData: TWin32FindData; // Структура поиска файла Winndows
end;
```

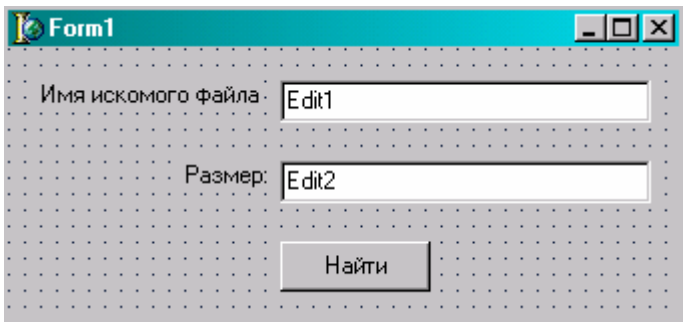
Функция *FindNext* заставляет найти следующий файл, удовлетворяющий параметрам, указанным в функции *FindFirst*. Этой функции нужно передать структуру *SearchRec*, по которой будет определено, на каком месте сейчас остановлен поиск и с этого момента он будет продолжен. Как только будет найден новый файл, функция вернёт в структуре *SearchRec* информацию о новом найдённом файле.

Функция *FindClose* закрывает поиск. В качестве единственного параметра нужно указать всё ту же структуру *SearchRec*.

Давай теперь напишем какой-нибудь реальный пример, который наглядно покажет работу с функциями поиска файлов. Какой бы пример тебе написать?

Давай посмотрим на структуру *TSearchRec*. Как видишь, она умеет возвращать размер найденного файла. Вот и тема для примера – мы напишем код, который будет определять размер указанного файла.

Создай новый проект и брось на форму два компонента TEdit и одну кнопку. Можешь ещё украсить всё это текстом. У тебя должно получиться нечто похожее на этот рисунок:



По нажатию кнопки напиши следующий текст:

```
var
 SearchRec:TSearchRec;
begin
 // Ищем файл
 if FindFirst(Edit1.Text,faAnyFile,SearchRec)=0 then

 // Забираем размер
 Edit2.Text:=IntToStr(SearchRec.Size)+ 'байт';

 //Закрываем поиск
 FindClose(SearchRec);
end;
```

На компакт диске, в директории ***\Примеры\Глава 10\FindFile*** *ты можешь увидеть пример этой программы.*

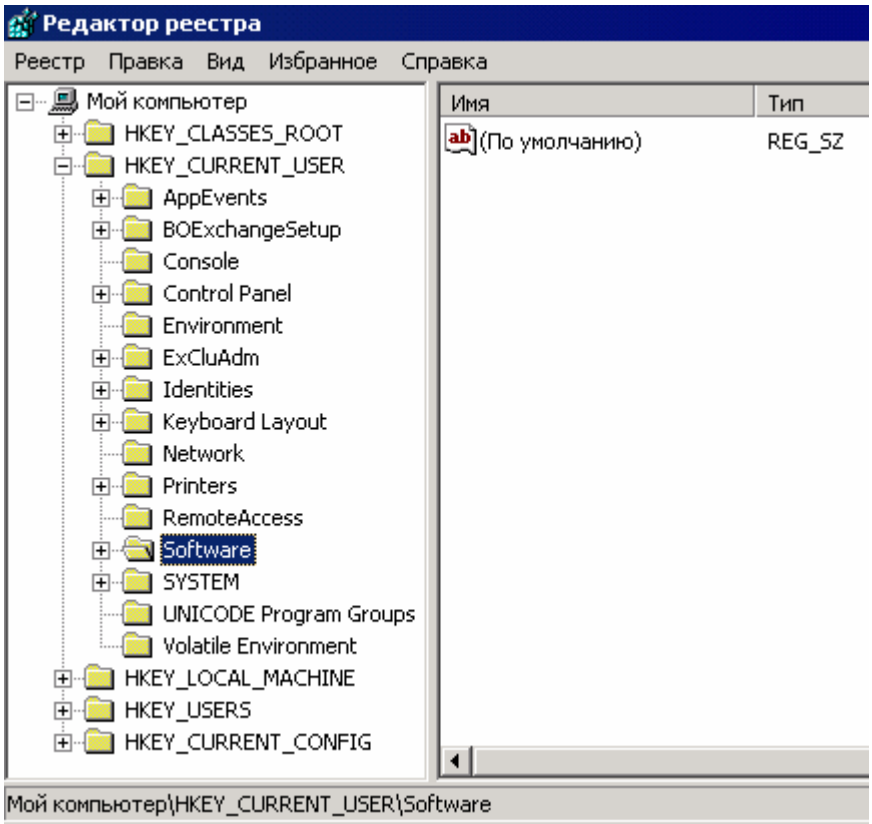
10.11 Работа с системным реестром

В этой главе я хочу тебе рассказать, как можно работать с системным реестром. Лично я люблю для этого использовать объект *TRegIniFile*. Он достаточно прост и удобен, поэтому я и тебе советую работать с ним. Для простого сохранения каких-то параметров программы этого объекта будет достаточно.

Давай разберёмся с этим объектом. Допустим, что у нас есть переменная *RegIni* типа *TRegIniFile*. Чтобы её инициализировать, нужно присвоить переменной результат вызова метода *Create* объекта *TRegIniFile*:

```
RegIni:=TRegIniFile.Create('Software');
```

По умолчанию, при инициализации ты получаешь доступ к разделу HKEY_CURRENT_USER. Методу *Create* нужно передать только один параметр – имя подраздела, который будет сразу открыт в разделе HKEY_CURRENT_USER.



Итак, после выполнения этого кода мы получили доступ к разделу HKEY_CURRENT_USER\Software. А что если ты хочешь открыть ещё подраздел и получить доступ к HKEY_CURRENT_USER\Software\Microsoft. Для открытия подразделов у объекта *TRegIniFile* есть метод *OpenKey*. Вот так можно открыть подраздел «Microsoft»:

```
RegIni.OpenKey('Microsoft', true);
```

У метода *OpenKey* уже два параметра:

1. Имя подраздела, который надо открыть.
2. Надо ли создавать подраздел, если он не существует.

Если в качестве второго параметра передать ***false***, и подраздел не будет существовать, то произойдёт ошибка и ничего не откроется, т.е. ты останешься там же, где и был. Ну а если мы передадим ***true*** и раздел не будет существовать, то программа автоматически создаст его.

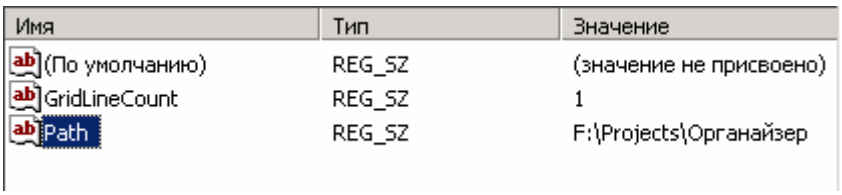
Теперь разберёмся с чтением и записью. Для чтения есть несколько методов:

- *ReadBool* – прочитать булево значение (true или false).
- *ReadInteger* – прочитать целое число.
- *ReadString* – прочитать строку.

Все они очень похожи и имеют одинаковое количество параметров. Единственная разница – в типе третьего параметра. Давай подробно рассмотрим *ReadString*, а остальное уже будем использовать по аналогии с этим методом.

У методов чтения есть три параметра:

1. Имя подраздела, в из которого мы хотим прочитать. Допустим, что мы открыли раздел Microsoft и находимся сейчас в реестре по адресу HKEY_CURRENT_USER\Software\ Microsoft. Если мы захотим прочитать строку из подраздела HKEY_CURRENT_USER\Software\Microsoft\ MySoftware, то в качестве первого параметра ты должен написать *MySoftware*.
2. Имя параметра. Если ты хочешь, чтобы параметр звался как *Path*, то укажи *Path* :).



3. Значение, которое будет использоваться по умолчанию, если такой параметр не существует. Для метода *ReadString* это должна быть строка или переменная типа строка.

Сразу хочу предупредить, что даже если ты будешь читать или записывать в реестр число с помощью методов WriteInteger или ReadInteger, объект TRegIniFile всё равно будет сохранять и читать эти числа как строки. А только после прочтения преобразовывать в число. Так что TRegIniFile реально хранит все данные в реестре только как строки. Если ты хочешь сохранять числа в реестре как самые простые числа, то нужно воспользоваться объектом TRegistry.

Итак, давай взглянём на команду чтения в действии:

```
Str:=RegIni.ReadString('MySoftware', 'Path', 'c:\');
```

В этом примере я читаю из подраздела *MySoftware* параметр *Path*. Если такой параметр не существует и не может вернуть значение, то будет возвращено значение по умолчанию «C:\». Результат чтения я записываю в переменную Str.

Точно так же происходит и запись, только в качестве третьего параметра нам надо указать не значение по умолчанию, а значение, которое надо записать. Для записи используются так же три метода:

- *WtiteBool* – записать булево значение (true или false).
- *WtiteInteger* – записать целое число.
- *WtiteString* – записать строку.

Простейший пример записи выглядит так:

```
RegIni.WriteString('MySoftware', 'Path', 'c:\Windows');
```

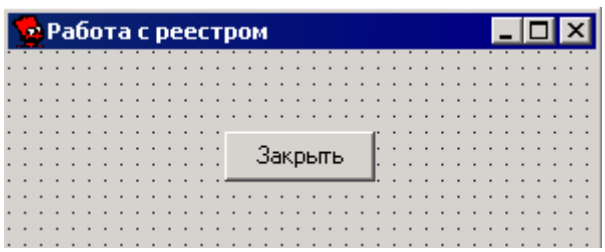
В этом примере я записываю в подраздел *MySoftware* параметр *Path*. Значение, которое будет записано равно третьему параметру - «*C:\Windows*».

После всех операций с реестром, его нужно закрыть с помощью метода Free:

```
RegIni.Free;
```

Для примера давай напишем программку, которая будет сохранять свои параметры при выходе и восстанавливать позицию и размер при запуске.

Я создал простейшую форму с одной только кнопкой «Закрыть».



Теперь я создал обработчик события *OnShow*, в котором я должен восстановить параметры программы, которые были после последнего закрытия проги. Чтобы не загромождать этот обработчик я просто написал вызов метода *LoadProgParam*. Этого метода пока не существует, но мы его скоро напишем.

```
procedure TForm1.FormShow(Sender: TObject);
begin
 LoadProgParam;
end;
```

Теперь я создал обработчик события *OnClose*. Здесь будут сохраняться параметры окна. Я опять не буду ничего загромождать и просто вызову метод *SaveProgParam*.

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 SaveProgParam;
end;
```

Если ты сейчас попытаешься скомпилировать прогу, то получишь три ошибки. Компилятор Delphi проругается на то, что не может найти процедуры *LoadProgParam* и *SaveProgParam*. Давай напишем их. Для этого подымись в начало модуля и найди раздел описания закрытых процедур **private**. Опиши там эти две процедуры без всяких параметров:

```
 private
  { Private declarations }
  procedure LoadProgParam;
  procedure SaveProgParam;
```

Теперь нажми сочетание клавиш Ctrl+Shift+C и Delphi создаст заготовки под эти процедуры:

```
procedure TForm1.LoadProgParam;
begin
end;

procedure TForm1.SaveProgParam;
begin
end;
```

Теперь напишем сами эти процедуры. Начнём с *SaveProgParam*:

```
procedure TForm1.SaveProgParam;
var
 FIniFile: TRegIniFile;

begin
 FIniFile := TRegIniFile.Create('Software'); // Инициализирую реестр

 FIniFile.OpenKey('VR',true); // Открываю раздел
 FIniFile.OpenKey('VR-Online',true); // Открываю ещё один раздел

 if WindowState=wsNormal then
  begin
   FIniFile.WriteInteger('Option', 'Width', Width);
   FIniFile.WriteInteger('Option', 'Heigth', Height);
   FIniFile.WriteInteger('Option', 'Left', Left);
   FIniFile.WriteInteger('Option', 'Top', Top);
  end;

 FIniFile.WriteInteger('Option', 'WinState', Integer(WindowState));

 FIniFile.Free; //Освобождаю реестр
end;
```

После инициализации реестра и подготовки разделов я делаю проверку, в каком состоянии находится окно. Если *WindowState* равно *wsNormal*, то я сохраняю параметры окна. Если нет, то этого делать не надо. Если у тебя стоит разрешение экрана 800x600, по при максимизированном окне значение ширины окна будет 802, а высоты 602. Эти значения больше реального разрешения и если ты установишь их при загрузке, то изменить размеры окна мышкой будет очень трудно.

После этого я сохраняю состояние окна - *WindowState*. Так как оно имеет тип *TWindowState*, то мне приходится приводить этот тип к *Integer* с помощью *Integer(WindowState)*.

Процедура *LoadProgParam* работает таким же образом:

```
procedure TForm1.LoadProgParam;
var
 FIniFile: TRegIniFile;
begin
 FIniFile := TRegIniFile.Create('Software');

 FIniFile.OpenKey('VR',true);
 FIniFile.OpenKey('VR-Online',true);

 Width:=FIniFile.ReadInteger('Option', 'Width', 600);
 Height:=FIniFile.ReadInteger('Option', 'Heigth', 400);
 Left:=FIniFile.ReadInteger('Option', 'Left', 10);
 Top:=FIniFile.ReadInteger('Option', 'Top', 10);

 WindowState:=TWindowState(FIniFile.ReadInteger('Option', 'WinState', 2));

 FIniFile.Free;
end;
```

 *На компакт диске, в директории* ***\Примеры\Глава 10\Register*** *ты можешь увидеть пример этой программы.*

10.12 Потоки

Под потоком я понимаю объект *TStream*, который является базовым объектом для потоков разных типов. В этом объекте реализованы все необходимые свойства и методы, необходимые для чтения и записи данных на различные типы носителей (память, диск, медиа носители). Благодаря этому объекту, доступ к разным типам носителей становиться одинаковым.

В этой главе, когда я описывал работу с файлами, мы уже использовали потоки. Объект *TFileStream* является потомком главного объекта *TStream* и позволяет получить доступ к диску. Точно так же можно получить доступ:

- к памяти через объект *TMemoryStream*.
- к сети через объект *TWinSocketStream*.
- к COM интерфейсу через *TOleStream*.
- к строкам, находящимся в динамической памяти *TStringStream*.

Это не полный список объектов потоков, но даже все эти объекты мы рассматривать не будем. Я покажу тебе только базовый объект *TStream*, а ты потом посмотри на то, как мы работали с *TFileStream* и увидишь, что всё просто. Точно так же можно будет работать с любым другим потоком, без каких либо изменений.

Итак, давай разберёмся со свойствами и методами потока:

Свойства

Position – указывает на текущую позицию курсора в потоке. Начиная с этой позиции будет происходить чтение данных.

Size – размер данных в потоке.

Методы

CopyFrom – метод предназначен для копирования из другого потока. У него два параметра – указатель на поток, из которого надо копировать и число показывающее размер данных подлежащих копированию.

Read – прочитать данные из потока, начиная с текущей позиции курсора. У этого метода два параметра – буфер, в который будет происходить чтение и число показывающее размер данных для копирования.

Seek – переместиться в новую позицию в потоке. У этого метода два параметра:

1. Число, указывающее на позицию, в которую надо перейти. Если тебе нужно передвинутся на пять байт, то просто поставь цифру 5.
2. Откуда надо двигаться. Тут возможны три варианта:
   - *soFromBeginning* – двигаться на указанные количество байт от начала файла.
   - *soFromCurrent* - двигаться на указанные количество байт от текущей позиции в файле к концу файла.
   - *soFromEnd* – двигаться от конца файла к началу на указанное количество байт.

SetSize – установить размер потока. Здесь только один параметр – число, указывающее новый размер потока. Допустим, что тебе надо уменьшить размер файла. В этом случае, с помощью метода *SetSize* потока *TFileStream* ты можешь уменьшить или даже увеличить размер файла.

Write – записать данные в текущую позицию потока. У этого метода два параметра:

1. Переменная, содержимое которой нужно записать.
2. Число байт для записи.

Это основные методы, которые тебе могут понадобиться при работе с потоками. На практике мы ещё встретимся с подобными объектами, и ты ещё раз увидишь, как с ними работать.

Глава 11. Обзор дополнительных компонентов Delphi.

В этой части моей книги я постараюсь дать максимальный обзор дополнительных компонентов Delphi. Мы познакомимся с большинством компонентов, упрощающих создание программ, и я постараюсь описать основные и необходимые при программировании свойства и методы описываемых мною компонентов.

Эта глава будет так же наполнена большим количеством примеров. Так как наши знания о программировании уже очень сильно улучшились, то и программы станут более сложными и более полезными.

В этой главе ты уже должен будешь на реальных примерах почувствовать мощь среды разработки Delphi, и возможно уже будешь готов самостоятельно создавать свои проекты. Но прежде чем ты это начнёшь делать, я советую тебе прочитать в конце книги главу о работе с самой средой разработки. Во многих книгах эта глава идёт в самом начале, а я напишу её в конце, потому что нет смысла рассказывать о том, чему ты ещё не можешь найти применения. Вот когда ты почувствуешь, что имеешь достаточно сил для самостоятельных проектов, вот тогда и изучай работу со средой Delphi и отладку приложений.

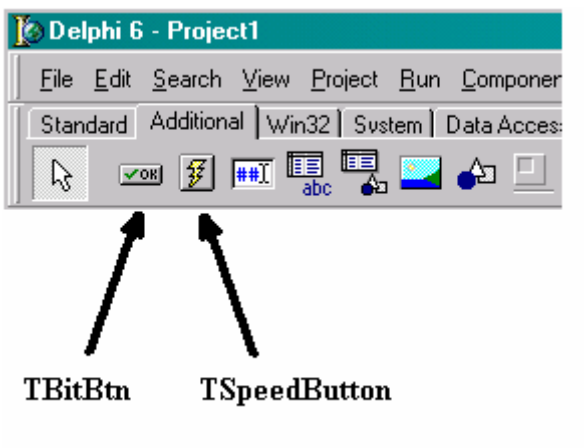
11.1 Дополнительные кнопки Delphi (TSpeedButton и TBitBtn)

Мы уже познакомились с кнопкой ***TButton*** с закладки *Standard*. В Delphi есть ещё два вида кнопок на закладке *Additional* – ***TBitBtn*** и ***TSpeedButton***. Помимо простого текста, они могут содержать и изображения, разница только в том, что ***TBitBtn*** может получать фокус, а ***TSpeedButton*** нет.

Что значит «получение фокуса»? Когда ты щёлкаешь по какому-то элементу управления, то он получает фокус ввода. Например, ты щёлкнул по строке ввода ***TEdit***. После этого в ней появляется курсор для ввода текста, и все события от клавиатуры будут посылаться именно этому компоненту. Точно так же с кнопкой. Если ты щёлкнул по ней, то все нажатия на клавиатуре будут посылаться кнопке. Правда, кнопка не может получать текст, но если ты нажмёшь кнопку *Enter*, когда фокус находиться на кнопке, то это будет равносильно нажатию по кнопке мышкой.

Кнопка ***TSpeedButton*** не может получать фокуса. Это значит, что если ты набирал какой-то текст в строке ввода, а потом щёлкнул по такой кнопке, то обработается соответствующее событие и фокус возвратиться обратно в строку ввода.

Как ты знаешь, фокус выделенного компонента в программах можно менять клавишей *TAB*. Если ты нажмёшь её, то будет выделен следующий по счёту компонент. Так вот клавишей *TAB* невозможно выделить кнопку ***TSpeedButton***.



TBitBtn хорошо подходит там, где нужна кнопка с изображением, а ***TSpeedButton*** для кнопок панели инструментов, потому что такие кнопки никогда не должны получать фокуса ввода. Именно поэтому ***TSpeedButton*** отображена в Delphi на панели инструментов квадратной.

Давай попробуем создать маленькое приложение, в котором будут использоваться оба типа этих кнопок. Запусти Delphi и создай новый проект.

Брось на форму компонент ***TPanel*** с закладки *Standard*. Установи у него свойство *Align* в *alTop*, чтобы панель растянулась по верху формы. Теперь удали текст в свойстве *Caption* и измени высоту (свойство *Height*) на 24.

Брось на панель кнопку ***TSpeedButton*** и установи у неё свойства *Left* (левая позиция) в 0 и *Top* (верхняя позиция) в 1. Ширина (*Width*) кнопки должна быть равна 23, а высота (*Height*) равна 22. Давай ещё изменим имя кнопки на *ExitButton*, потому что я собираюсь сделать выход по нажатию этой кнопке.

Если ты всё сделал правильно, то у тебя должно получится нечто похожее на рисунок 11.1.1.

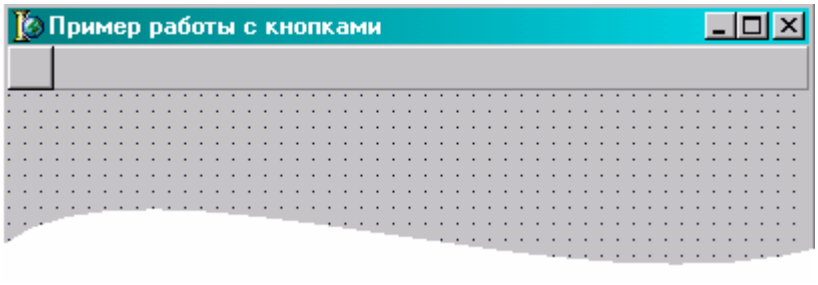


Рис 11.1.1 Форма будущей программы

Теперь дважды щёлкни по свойству *Glyph* и перед тобой должно открыться окно загрузки изображения (рисунок 11.1.2). Нажми на кнопку *Load* и загрузи картинку. С Delphi идёт большая библиотека готовых изображений и расположены они Program Files\Common Files\Borland Shared\Images\Buttons. На диске с книгой ты можешь найти мою подборку дополнительных картинок, которая тебе может пригодиться в последующем.

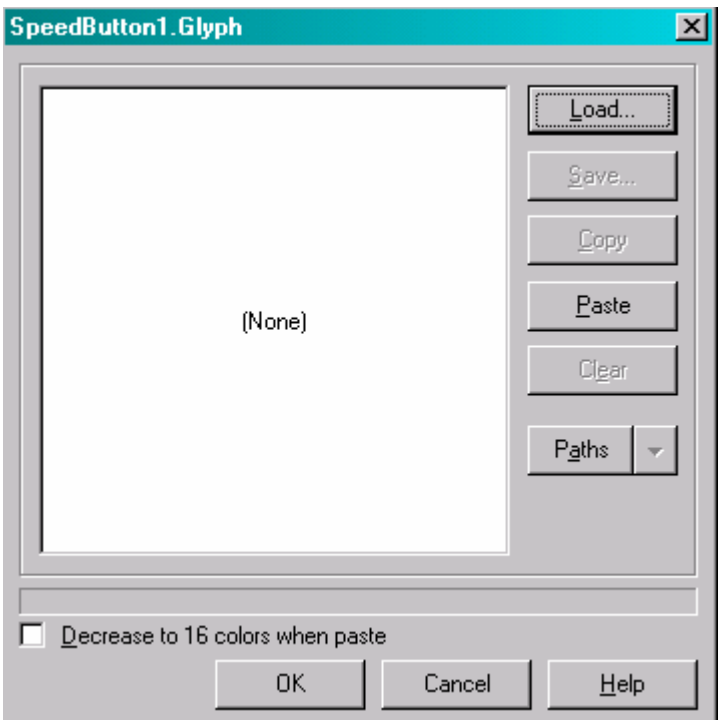


Рис 11.1.2 Окно загрузки изображения

Я решил долго не мучиться и загрузить картинку, которая идёт вместе с Delphi под названием dooropen.bmp. Можешь сделать то же самое. Как только ты выберешь картинку, нажми *OK* чтобы закрыть окно загрузки изображения.

Теперь на кнопке отображается выбранная тобой картинка. Можешь запустить программу, чтобы посмотреть на результат её работы. На рисунке 11.1.3 показан результат нашей программы. Можешь пощёлкать по кнопке, которая пока ещё ничего не делает.

Как видишь, кнопка выглядит немного выпукло, а во всех современных приложениях кнопки плавающие. Это легко исправить, если изменить свойство кнопки *Flat* на *true*.

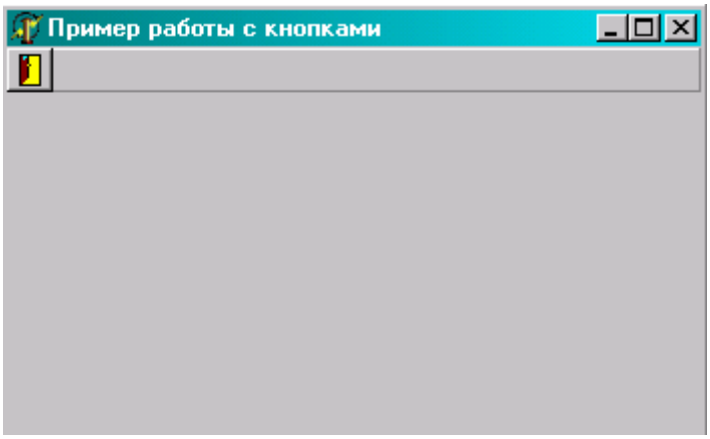


Рис 11.1.3 Первый результат работы.

Теперь создадим для кнопки событие *OnClick*. Для этого можно щёлкнуть дважды по самой кнопке или выделить её и на закладке *Events* объектного инспектора дважды щёлкнуть по свойству *OnClick*. В созданном обработчике события напишем:

```
procedure TForm1.ExitButtonClick(Sender: TObject);
begin
 Close; //Выход из программы
end;
```

Можно запускать программу и проверять её работу в действии.

Теперь брось на форму ещё две кнопки. Я расположил их как на рисунке 11.1.4. В первую из них я загрузил картинку *Bulboff.bmp* (и назвал BulboffButton), а во вторую *Bulbon.bmp* (и назвал BulbonButton).

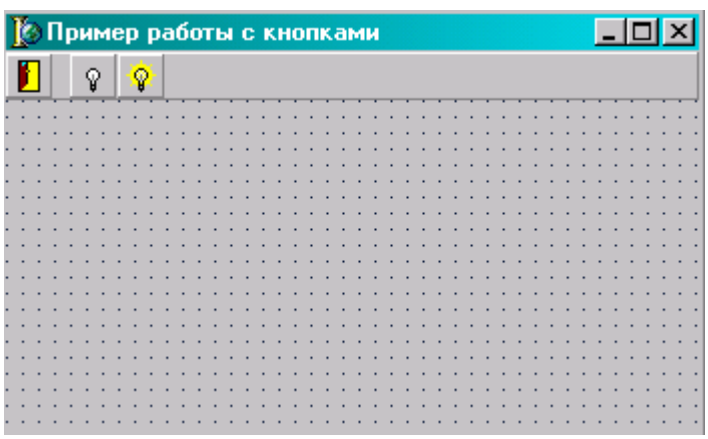


Рис 11.1.4 Улучшенная форма будущей программы.

Теперь установи у обеих кнопок свойство *GroupIndex* равным 1 и у любой из них измени свойство *Down* на *true*. Попробуй теперь запустить программу и понажимать на новые кнопки. Когда нажата одна из них, то другая отжата. Когда ты нажмёшь на другую, то она станет нажатой, а вторая автоматически освободится.

Таким способом часто оформляют такие операции как выравнивание. Например, в том же текстовом редакторе Word выравнивание выполнено именно таким способом.

Возможность кнопки находиться в нажатом и нормальном состоянии появилась после того, как ты сгруппировал две кнопки, присвоив в свойство *GroupIndex* значение 1. Ты можешь создать ещё одну группу кнопок (количество кнопок не ограничено двумя) и присвоить ей значение 2. В этом случае она будет работать независимо от первой. И помни, что свойство *Down* может быть равно *true* только у одной кнопки в группе.

Теперь брось на форму кнопку ***TBitBtn***, назовём её StartBtn. В свойстве *Caption* напиши *Старт* и загрузи сюда любую картинку. Загрузка происходит точно таким же образом, как и при использовании ***TSpeedButton***.

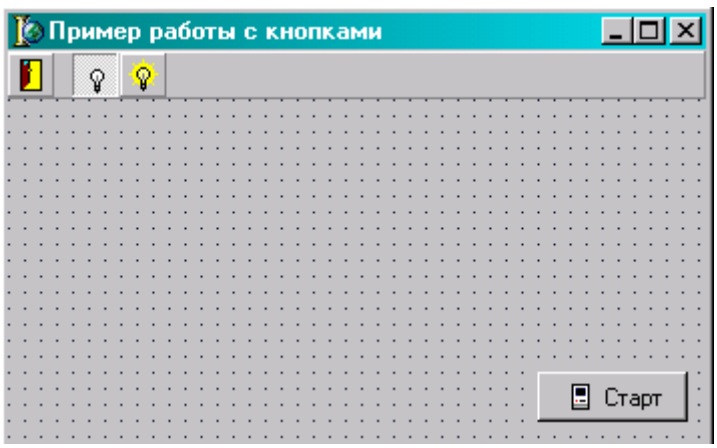


Рис 11.1.5 Улучшенная форма

Кнопки ***TBitBtn*** и ***TSpeedButton*** очень похожи и имеют практически все одинаковые свойства и методы, поэтому больше тут сказать уже практически нечего.

Очень интересным является свойство *Layout*, которое показывает, где должна располагаться картинка, а где текст. На рисунке 11.1.6 показаны разные варианты кнопок, а снизу приписаны установленные значения в свойстве *Layout*.

Рис 11.1.6 Расположения кнопок и текста

Ещё одним интересным свойством является *Kind*. В нём заложен список заранее подготовленных стандартных кнопок. Выбирая любой из них, автоматически изменяется текст и изображение на кнопке. На рисунке 11.1.7 ты можешь увидеть различные кнопки и соответствующие им значения *Kind*. Жаль только, что текст англоязычный, но его изменить очень легко.

Помимо картинки и текста изменяется и свойство ModalResult – результат, который вернёт кнопка для диалогового окна. Ты можешь и сам менять это свойство, для любой кнопки изменяя реакцию кнопки и всей программы.

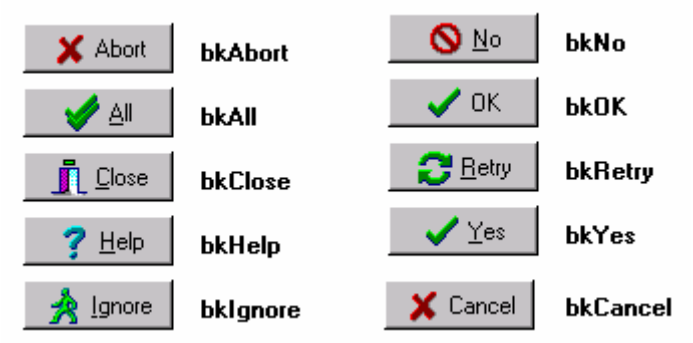


Рис 11.1.7 Различные значения свойства Kind.

Давай напишем пример, который будет запускать дочернее модальное окно по нажатию кнопки *Старт*.

Создай ещё одну форму. Советую сразу открыть *Project Manager*, чтобы легче было переключаться между формами (выбрать в меню *View* пункт *Project Manager*). Теперь брось на форму три кнопки и желательно расположить их, как на рисунке 11.1.8.

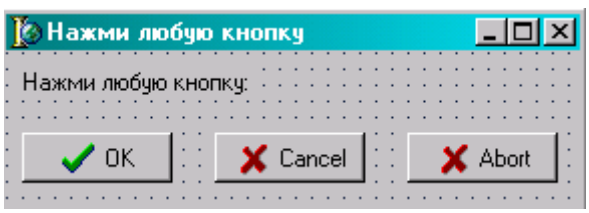


Рис 11.8 Вторая форма нашей программы.

Для первой кнопки установим свойство *Kind* в *bkOK*, для второй в *bcCancel*, а для третей *bkAbort*.

Теперь вернёмся в первую форму (для этого дважды щёлкни в окне *Project Manager* по первой форме) и создадим обработчик события *OnClick* для кнопки *Старт*.

```
procedure TMainForm.StartBtnClick(Sender: TObject);
begin
 Form2.ShowModal; // Показываю вторую форму

 if Form2.ModalResult=mrOk then
  Application.MessageBox('Вы нажали кнопку OK', 'Вы нажали');

 if Form2.ModalResult=mrCancel then
  Application.MessageBox('Вы нажали кнопку Cancel', 'Вы нажали');

 if Form2.ModalResult=mrAbort then
  Application.MessageBox('Вы нажали кнопку Abort', 'Вы нажали');
end;
```

В первой строчке я показываю вторую форму. Я показываю её как модальное окно. После того, как пользователь нажмёт одну из трёх кнопок, наше модальное окно закроется и в свойстве формы *Form2.ModalResult* будет находиться результат, который указан у нажатой кнопки в свойстве *ModalResult*. Вот именно этот результат я и проверяю и в зависимости от этого вывожу на экран необходимое сообщение. Попробуй запустить этот пример и посмотреть его в действии.

*На компакт диске, в директории **\Примеры\Глава 11\BitBtn&SpeedButton** ты можешь увидеть пример этой программы.*

11.2 Самостоятельная подготовка картинок для кнопок

Стандартный размер картинки для кнопки равен 16x16. Можешь создавать изображения и большего размера, но если хочешь, чтобы кнопочки выглядели элегантно, то желательно чтобы они имели именно такой размер.

На сколько мы уже знаем, кнопка может быть в двух состояниях – активная (свойство *Enabled = true*) и неактивная (*Enabled = false*). Когда кнопка неактивна, то её изображение должно отображаться серым цветом. Если ты будешь делать кнопку неактивной, то желательно подготавливать изображение размером 32x16. Такое изображение нужно разделить пополам и слева нарисовать цветную картинку, а справа в оттенках серого, как показано на следующем рисунке:

Когда кнопка активна, Delphi автоматически будет подставлять изображение слева, а когда не активна, то правое изображение.

В принципе, Delphi и сам может автоматически сделать второе изображение для неактивного состояния, поэтому этим правилом можно пренебрегать. Но если во время тестирования программы ты заметил, что в неактивном состоянии изображение исчезает, то нужно подготовить правильную картинку.

11.3 Маскированная строка ввода (TMaskEdit)

Слово «*маскированная*» происходит не от слова прятаться (маскироваться), а от слова «*маска*». Очень часто надо, чтобы пользователь ввёл в программу какие-то данные в определённом формате. Для этого существует компонент TMaskEdit.

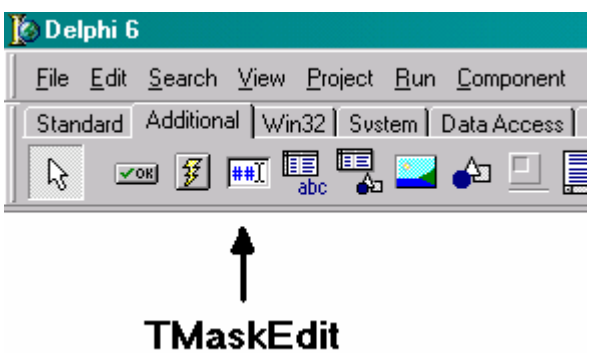


Рис 11.3.1 TMaskEdit.

Давай создадим маленький пример, который проиллюстрирует работу с компонентом ***TMaskEdit***. Создай новое приложение. Брось на него текст (TLabel) в котором напиши «*Введите дату*». Рядом поставь компонент ***TMaskEdit***. Щёлкни по нему

и посмотри на свойства. Как видишь, большинство свойств идентично компоненту ***TEdit*** с палитры инструментов *Standard*.

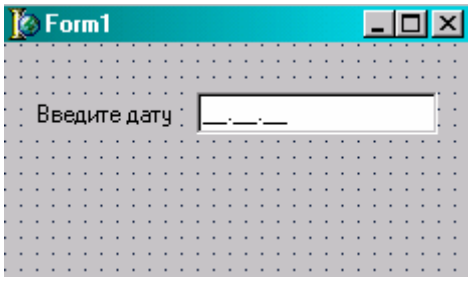


Рис 11.3.2 Форма будущей программы.

Самое интересное здесь свойство – *EditMask*. Щёлкни по нему два раза и перед тобой откроется окно редактора ввода (на рисунке 11.3.3 ты можешь увидеть окно редактора маски).

В строке ввода *Input Mask* ты можешь вводить маску. Справа расположен список примеров. Слева внизу расположена строка *Test Input*, в которой можно протестировать маску.

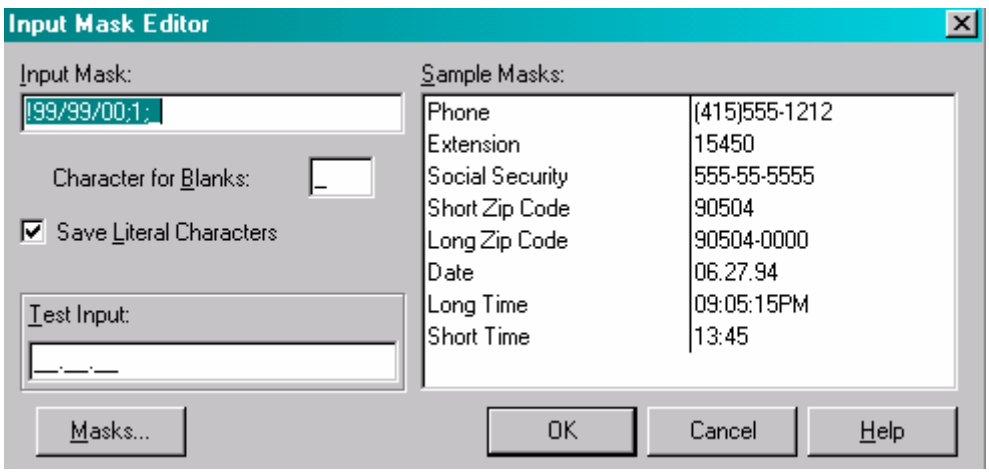


Рис 11.3.3 Редактор маски.

Создавать маску очень просто. Если ты хочешь, чтобы пользователь ввёл четрые числа, потом тире и ещё три числа, то можно в строку *Input Mask* ввести 9999-999. Цифра 9 означает, что на этом месте должна быть любая цифра. Если тебе нужно, чтобы вначале ввода была ещё буква **R**, то укажи маску R9999-999.

На компакт диске, в директории ***\Примеры\Глава 11\MaskEdit*** *ты можешь увидеть пример этой программы.*

11.4 Сеточки (TStringGrid, TDrawGrid)

Очень часто в программах нужны сетки ввода данных. Например, взгляни на электронную таблицу Excel. Её окно построено на основе сетки ввода. Ты можешь себе представить электронную таблицу без такой сетки? Я тоже не могу.

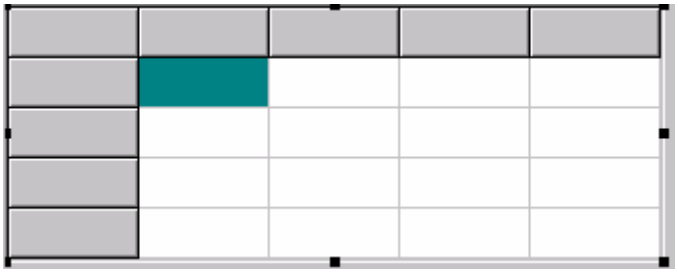


Рис 11.4.1 Самая простейшая сетка.

На рисунке 11.4.1 ты можешь видеть простейшую сетку, которая ещё не умеет ничего делать. В Delphi тебе доступно сразу два вида сеток ***TStringGrid*** и ***TDrawGrid***. Разниза в них незначительная – в ***TStringGrid*** ты можешь вводить данные и они там будут сохраняться и отображаться, а в ***TDrawGrid*** данные могут вводиться, но за отображение должен отвечать только ты. Более понятным языком можно сказать, что ***TStringGrid*** – это сетка строк, а ***TDrawGrid*** – это сетка рисунков.

 - TStringGrid

 - TDrawGrid

Я покажу тебе работу только с ***TStringGrid*** потому что он более распространён и надобность в нём появляется на много чаще.

Создай новый проект и брось на него сетку ***TStringGrid***. Выдели её, и давай рассмотрим её специфичные свойства в объектном инспекторе (те, что нам известны, я рассматривать не буду):

BorderStyle – стиль обрамления. Здесь возможны варианты *bsSingle* или *bsNone*. Мне больше нравиться второе. Но ты можешь попробовать самостоятельно установить оба этих типа и посмотреть, что тебе больше по душе.

ColumnCount – количество колонок в сетке. Оставим, так как есть – 5 штук.

DefaultColWidth – ширина колонок по умолчанию.

DefaultDrawing – рисование по умолчанию. Если здесь установлено *true*, то компонент сам будет отображать введённые данные. Если *False*, то это придётся делать самостоятельно.

DefaultColHeight – Высота строк по умолчанию. Значение установленное здесь достаточно большое, поэтому давай введём 16. Так сетка будет выглядеть более элегантно.

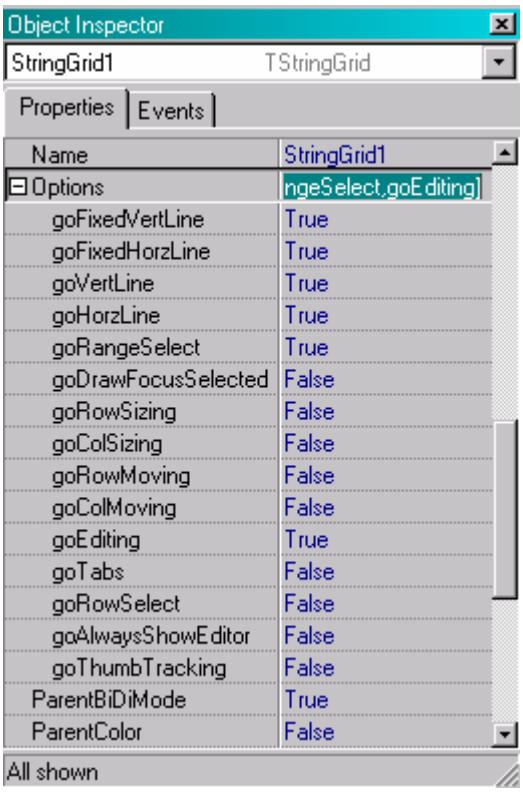
FixedColor – цвет фиксированных колонок и строк. В фиксированные ячейки нельзя вводить текст и они используются в качестве заголовков. На рисунке 11.12 первая колонка и первая строка фиксированы и поэтому отображены цветом элементов управления.

FixedCols – количество фиксированных колонок. Они всегда первые, нельзя создать фиксированную колонку в середине сетки. Это можно сделать только самостоятельно.

FixedRows – количество фиксированных строк. Они всегда первые, нельзя создать фиксированную строку в середине сетки. Это можно сделать только самостоятельно.

GridLineWidth – толщина разделительных линий сеток.

Options – настройки сетки. Если дважды щёлкнуть по этому свойству или один раз по квадратику слева от названия свойства, то раскроется большой список свойств. Давай рассмотрим каждое из свойств:



goFixedVertLine – рисовать вертикальные линии сетки у фиксированных ячеек.

goFixedHorzLine – рисовать горизонтальные линии сетки у фиксированных ячеек.

goVertLine – рисовать вертикальные линии сетки у нефиксированных ячеек.

goHorzLine – рисовать горизонтальные линии сетки у нефиксированных ячеек.

goRangeSelect – позволять выделять несколько ячеек.

goDrawFocuSelected – рисовать фокус выделенной ячейки.

goRowSizing – можно ли изменять размер строк перетягиванием мышкой.

goColSizing - можно ли изменять размер колонок перетягиванием мышкой.

goRowMoving – можно ли перемещать строки. Если *true*, то можно мышкой нажать на фиксированную ячейку строки и перетащить её в новое положение.

goColMoving - можно ли перемещать колонки. Если *true*, то можно мышкой нажать на фиксированную ячейку колонки и перетащить её в новое положение.

goEditing – можно ли вводить с клавиатуры данные в сетку. Для нашего примера установим в True.

goTabs – если здесь установить *True*, то между ячейками можно путешествовать с помощью клавиши *Tabs*.

goRowSelect – если здесь *false*, то выделяется только выделенная ячейка. Если *true*, то вся строка.

goAlwaysShowEditor – если здесь *false*, то когда ты становишься в ячейку, то для её редактирования нужно нажать *Enter* или *F2*. Если *True*, то как только ты выделил ячейку, её сразу можно редактировать.

goThumbTracking – будут ли данные прорисовываться пока пользователь перемещает полосу прокрутки.

RowCount – количество строк. Для первого примера нам хватит и пяти.

ScrollBars – нужно ли показывать полосы прокрутки. Здесь возможны варианты:

ssNone – не показывать.

ssHorizontal – только горизонтальную полосу.

ssVertical – только вертикальную полосу.

SsNone – не показывать.

Итак, давай напишем первый пример. Создай обработчик события OnShow для формы и там напиши:

```
procedure TMainForm.FormShow(Sender: TObject);
begin
 // Заполняем значениями первую колонку
 StringGrid1.Cells[0,1]:='Иванов';
 StringGrid1.Cells[0,2]:='Петров';
 StringGrid1.Cells[0,3]:='Сидоров';
```

```
 StringGrid1.Cells[0,4]:='Смирнов';

// Заполняем значениями первую строку
 StringGrid1.Cells[1,0]:='Год рожд.';
 StringGrid1.Cells[2,0]:='Место рожд.';
 StringGrid1.Cells[3,0]:='Прописка';
 StringGrid1.Cells[4,0]:='Семейное положение';
end;
```

У объекта ***TStringGrid*** есть ещё одно свойство, которое не описано в объектном инспекторе – *Cells*. Это свойство – двухмерный массив из строк, в которых хранятся данные, отображаемые в сетке. Чтобы получить доступ к какой-либо ячейке, нужно записать *StringGrid1.Cells[номер колонки, номер ячейки]*. Обращаю твоё внимание, что нумерация колонок и строк начинается с нуля. Например, если ты хочешь записать во вторую колонку и четвёртую строку текст «*Привет*», то необходимо записать:

```
StringGrid1.Cells[1,3]:='Привет';
```

Таким же способом можно и читать содержимое ячеек:

```
if StringGrid1.Cells[1,3]='Привет' then
 Сделать какие-либо действия;
```

На рисунке 11.4.2 ты можешь увидеть окно, в котором показан наш пример в запущенном виде.

Рис 11.4.2 Самая простейшая сетка.

Давай усилим наш пример и заодно поглубже познакомимся с сеткой. Как видишь, в нашем примере есть поле «*Год рождения*». В это поле должны вводиться даты, а значит, они должны иметь определённый формат. Было бы очень удобно, если бы в это поле можно было бы задать маску ввода, но сетка такое не поддерживает. Но тут можно пойти на одну хитрость – когда пользователь щёлкает по нужному полю, подставлять компонент ***TMaskEdit***, и пускай пользователь вводить информацию в него.

Давай, реализуем сказанное на практике. Для этого брось на форму компонент ***TMaskEdit*** (место расположения не имеет значения), и назови его *DateEdit*. Установи у него маску для ввода даты ***99/99/9999***. Теперь установи у него свойство *Visible* в *false*, чтобы компонент не был виден.

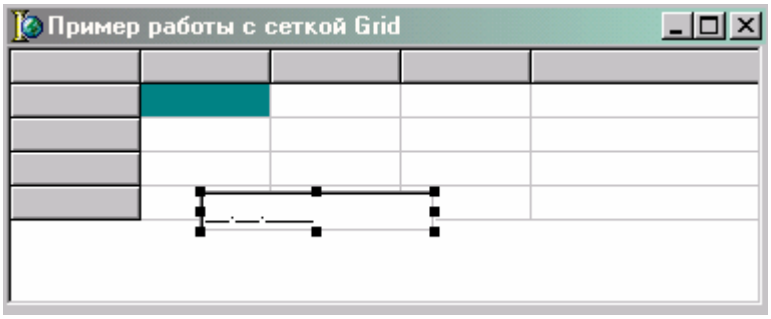


Рис 11.4.3 TMaskEdit на обновлённой форме

Теперь создай обработчик события *OnDrawCell*, в нём мы напишем следующее:

```
procedure TMainForm.StringGrid1DrawCell(Sender: TObject; ACol,
  ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
 DateEdit.Visible := false; // Сделать невидимой компонент DateEdit

 if (gdFocused in State) then // Если текущая ячейка в фокусе то …
  begin
   if ACol=1 then // Если рисуется ячейка первой колонки то …
    begin
     DateEdit.Text:=StringGrid1.Cells[ACol, ARow]; // Записать в DateEdit текст ячейки
     DateEdit.Left := Rect.Left + StringGrid1.Left+2; // Установить левую позицию
     DateEdit.Top := Rect.Top + StringGrid1.top+2; // Установить верхнюю позицию
     DateEdit.Width := Rect.Right - Rect.Left; // Установить ширину
     DateEdit.Height := Rect.Bottom - Rect.Top; // Установить высоту
     DateEdit.Visible := True; // Сделать компонент DateEdit видимым
     exit; // Выход из процедуры
    end;
  end;
end;
```

Этот обработчик вызывается каждый раз, когда надо прорисовать какую-нибудь ячейку. Если прорисовывается вся сетка, то он вызывается для каждой ячейки отдельно.

Для нас Delphi создал процедуру обработчик события **StringGrid1DrawCell** со следующими параметрами:

Sender – здесь передаётся указатель на объект, который сгенерировал событие.

ARow и *ACol* – номер строки и номер столбца (координаты) ячейки, которую надо прорисовать.

Rect – Структура, в которой указаны относительные размеры и положения ячейки. Что я понимаю под словом «*относительные*»? Структура *Rect* выглядит так:

```
type
 TRect = record
   Left, Top, Right, Bottom: Integer;
 end;
```

Как видишь, это структура из четырёх параметров – левой, верхней, правой и нижней позиции. На рисунке 11.4.4 стрелками показана, показан тот размер, который

будет в параметрах Left и Right структуры Rect. Размеры будут указаны в пикселях. В параметре Right будет указано расстояние в пикселях от левого края сетки до правого края ячейки, а в параметре Bottom будет расстояние от верхнего края сетки до нижнего края ячейки.

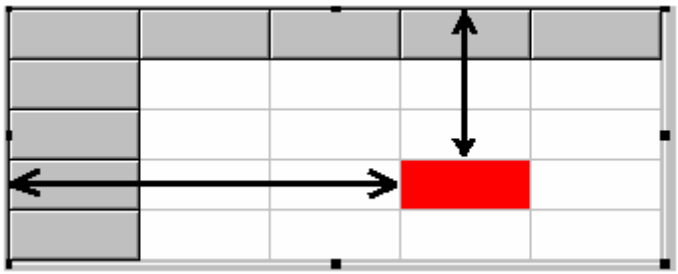


Рис 11.4.4 Левая и верхняя позиция ячейки

Ну и последний параметр, передаваемый нам в процедуру обработчик – *State*. В Нём находиться информация о состоянии ячейки, которую надо прорисовать. Состояния могут быть следующими:

gdSelected – ячейка выделена.

gdFocused – ячейка имеет фокус ввода.

gdFixed - ячейка является фиксированной.

Если в параметре *State* нет ни одного из этих значений, то это простая ячейка.

Параметр *State* объявлен как набор значений. Это значит, что он может принимать любое из указанных значений или их сочетания. Чтобы проверить, установлено ли что-нибудь в *State*, надо написать:

```
if (значение in State) then
  Выполнить действие
```

Именно так я проверяю во второй строчке кода наличие значения *gdFocused*. И только если ячейка, которую надо прорисовать имеет фокус, выполняю следующие действия. Но перед этим я прячу нашу маскированную строку ввода, потому что она могла находиться видимой в другой ячейке, и чтобы не было проблем, лучше её спрятать.

Если рисуемая ячейка в фокусе, то я проверяю, в какой колонке находиться рисуемая ячейка. Если это первая колонка (где мы должны вводить дату), то я должен показать *DateEdit* на месте рисуемой ячейки. Для этого я сначала присваиваю в *DateEdit* текст, который должен находиться в данной ячейке. Затем устанавливаю позицию и размеры компонента *DateEdit*, и только потом показываю его.

Как видишь, способ очень простой и элегантный. Теперь осталось только создать обработчик события *OnChange* для компонента DateEdit. Это событие происходит, когда данные в строке ввода изменились, а это значит, что нам их надо сразу же прописать в редактируемую ячейку сетки иначе они потеряются. Это потому что все данные вводятся в *DateEdit*, а не в сетку, а переносить мы их должны вручную.

```
procedure TMainForm.DateEditChange(Sender: TObject);
begin
 StringGrid1.Cells[StringGrid1.Col, StringGrid1.Row]:=DateEdit.Text;
end;
```

Чтобы ещё больше закрепить материал, я сделал в последней колонке появление компонента ***TCheckBox***, по которому можно менять значение в ячейке между «*Женат*» и «*Холост*». Как это сделано я объяснять не буду, попробуй разобраться сам, потому что код практически идентичный. А я только дам исходник:

Для начала на форму надо бросить компонент *CheckBox* и сделать его невидимым. Потом надо изменить событие *OnDrawCell*:

```
procedure TMainForm.StringGrid1DrawCell(Sender: TObject; ACol,
  ARow: Integer; Rect: TRect; State: TGridDrawState);
begin
 DateEdit.Visible := false;
 CheckBox1.Visible := false;
 if (gdFocused in State) then
  begin
   if ACol=1 then
    begin
     DateEdit.Text:=StringGrid1.Cells[ACol, ARow];
     DateEdit.Left := Rect.Left + StringGrid1.Left+2;
     DateEdit.Top := Rect.Top + StringGrid1.top+2;
     DateEdit.Width := Rect.Right - Rect.Left;
     DateEdit.Height := Rect.Bottom - Rect.Top;
     DateEdit.Visible := True;
     exit;
    end;

   if ACol=4 then
    begin
     CheckBox1.Caption:=StringGrid1.Cells[ACol, ARow];

     if CheckBox1.Caption='Æåíàò' then
      CheckBox1.Checked:=true
     else
      CheckBox1.Checked:=false;

     CheckBox1.Left := Rect.Left + StringGrid1.Left+2;
     CheckBox1.Top := Rect.Top + StringGrid1.top+2;
     CheckBox1.Width := Rect.Right - Rect.Left;
     CheckBox1.Height := Rect.Bottom - Rect.Top;
     CheckBox1.Visible := True;
     exit;
    end;
  end;
end;
```

Ну и конечно же поймать событие *OnClick* компонента *CheckBox1*, чтобы записать изменённое значение обратно в сетку:

```
procedure TMainForm.CheckBox1Click(Sender: TObject);
begin
 if CheckBox1.Checked=true then
  CheckBox1.Caption:='Æåíàò'
 else
  CheckBox1.Caption:='Õîëîñò';
```

```
 StringGrid1.Cells[StringGrid1.Col, StringGrid1.Row]:=CheckBox1.Caption;
end;
```

---

*На компакт диске, в директории **\Примеры\Глава 11\Grid** ты можешь увидеть пример этой программы.*

11.5 Компоненты-украшения (TImage, TShape, TBevel)

Сейчас мы с тобой познакомимся с тремя компонентами, которые чаще всего используются для наведения красоты в приложениях. Но это не значит, что красота их основное назначение, просто на данном этапе нам достаточно и этого. Чуть позже мы создадим что-нибудь более полезное из этих компонентов, но пока….

- TImage

- TShape

- TBevel

Создай новый проект и брось на форму компонент ***TImage***. Теперь щёлкни дважды по свойству *Picture* и перед тобой появится уже знакомое окно загрузки изображения (рисунок 11.5.1).

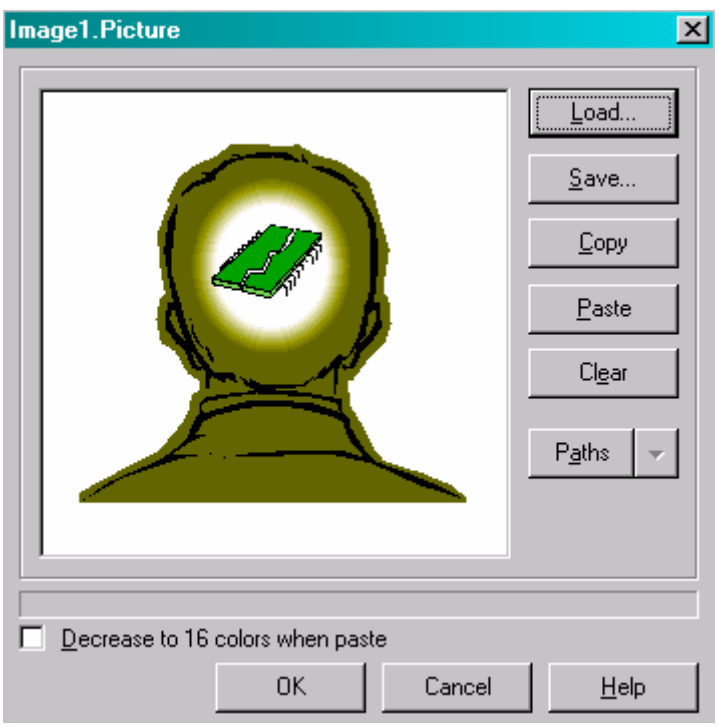


Рис 11.5.1 Окно загрузки изображения

Теперь, если ты хочешь, чтобы твой компонент автоматически принимал размеры загруженной картинки, то установи свойство *AuroSize* в *true*. Если ты хочешь, чтобы

картинка была по центру компонента, то нужно выставить свойство *Center* в *true* (при этом *AuroSize* надо выставить в *false*). Ну а если надо растянуть картинку по всей поверхности компонента, то надо выставить свойство *Stretch* в *true* (при этом *AuroSize* надо выставить в *false*).

Если картинка должна быть прозрачной, то можно выставить свойство *Transparent* в *True*. Хотя такая прозрачность и не очень эффективна при использовании растровых картинок, но на безрыбье и рак будем мясом. Но если ты будешь использовать векторную графику, такую как wmf формат, то прозрачность будет идеальной (как у меня на рисунках).

Остальные свойства ***TImage*** тебе должны быть уже известны, поэтому на них я останавливаться не буду.

Теперь разберёмся с компонентом ***TShape***. Брось один такой экземпляр на форму и посмотри на свойства. Самое интересное здесь – свойство *Shape*, которое отвечает за тип фигуры отображаемой на компоненте. На рисунке 11.5.2 показана форма моей программы, на которой расположено шесть разных видов компонента ***TShape***. Справа от компонента подписано, какое именно значение установлено в свойстве *Shape*.

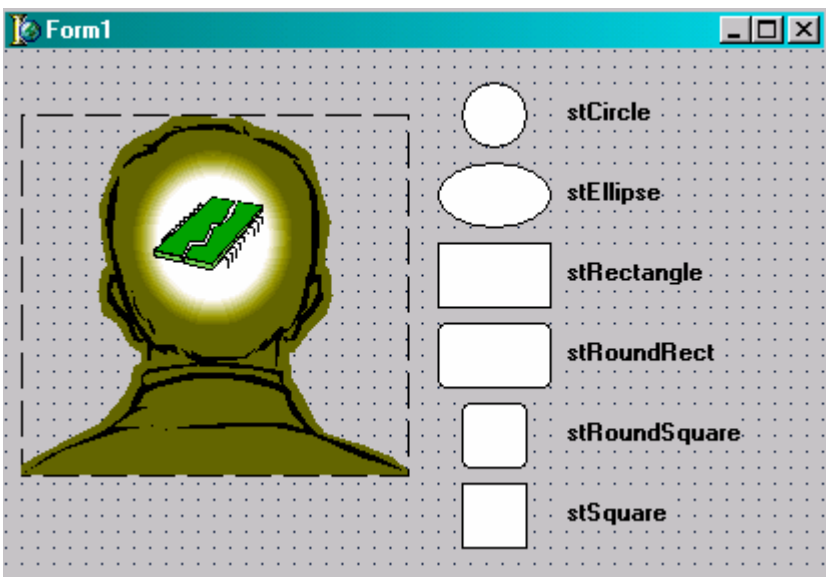


Рис 11.5.2 Окно загрузки изображения

Помимо этого, за отображение отвечают ещё и свойства *Brush* (закраска) и *Pen* (карандаш). Свойство *Brush* отвечает за цвет и стиль закраски нашей фигуры, а свойство *Pen* говорит о стиле и цвете обрамления.

Если дважды щёлкнуть по свойству *Brush*, то появиться список из двух дополнительных свойств:

1. *Color* – цвет заливки;
2. *Style* - способ заливки.

На рисунке 11.5.3 показаны различные типы заливки, которые ты можешь установить и результат их работы. Попробуй сам поиграть с этими значениями, устанавливая различные значения цветов и способов заливки.

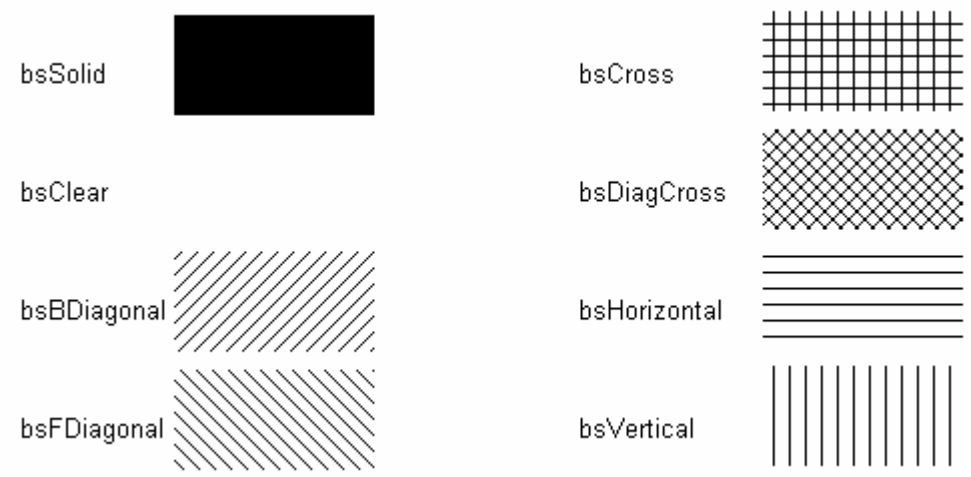


Рис 11.5.3 Различные способы заливки

Когда ты будешь изменять значения, то заметишь, что изменяется только внутренняя окраска компонента, а обрамление будет оставаться в виде тонкой полоски чёрного цвета. За обрамление отвечает свойство *Pen*. Если щёлкнуть по нему два раза, то перед тобой откроется список из четырёх дополнительных свойств:

1. Color - цвет заливки;
2. Mode – режим отображения;
3. Style – стиль линии;
4. Width – толщина линии.

Здесь так же могу посоветовать самому попробовать выставить разные значения, чтобы увидеть результат. Я долго могу расписывать возможности этих свойств, но пока ты сам не увидишь, я не думаю, что что-нибудь будет понятным. На рисунке 11.5.4 ты можешь увидеть различные стили карандаша:

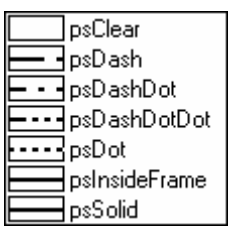


Рис 11.5.4 Стили карандаша

Ну и наконец компонент ***TBevel***, который предназначен для простого обведения чего-либо рамочкой. На первый взгляд этот компонент похож на ***TPanel***, но это только на первый взгляд, потому что на ***TBevel*** нельзя ставить компоненты. Это практически прозрачная рамочка, но только практически. Если ты поставишь её поверх строки ввода, то эта строка будет видна сквозь ***TBevel***, а вот доступа к ней получить будет невозможно.

Самыми интересными свойствами у этого компонента являются *Shape* и *Style*. В разных сочетаниях значений в них можно добиться совершенно невероятных рамочек. На рисунке 11.5.5 я попробовал воспроизвести некоторые возможные варианты, но только некоторые. Узнать о рамке больше ты можешь только если сам попробуешь выставить какие-нибудь значения.

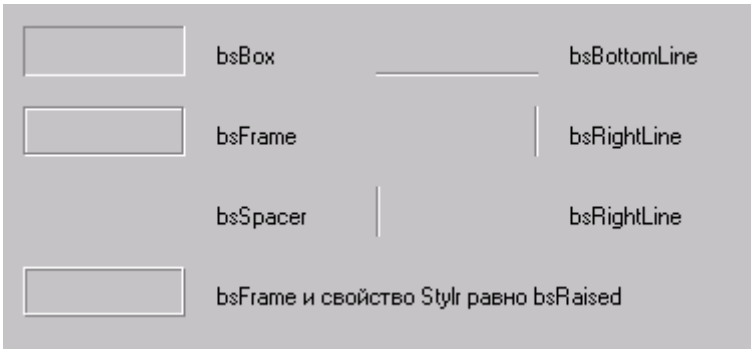


Рис 11.5.5 Разные стили обрамлений

А на рисунке 11.5.6 показано окончательное окно моей программы, в которой я приводил тебе примеры компонентов из этой части моей книги. Как видишь, окошко выглядит очень даже прилично. Я вообще люблю устанавливать на форму компонент TBevel и растягивать его на всю форму (устанавливать свойство *Align* в *alClient*). Это очень сильно украшает окно и абсолютно не влияет на производительность или загруженность памяти.

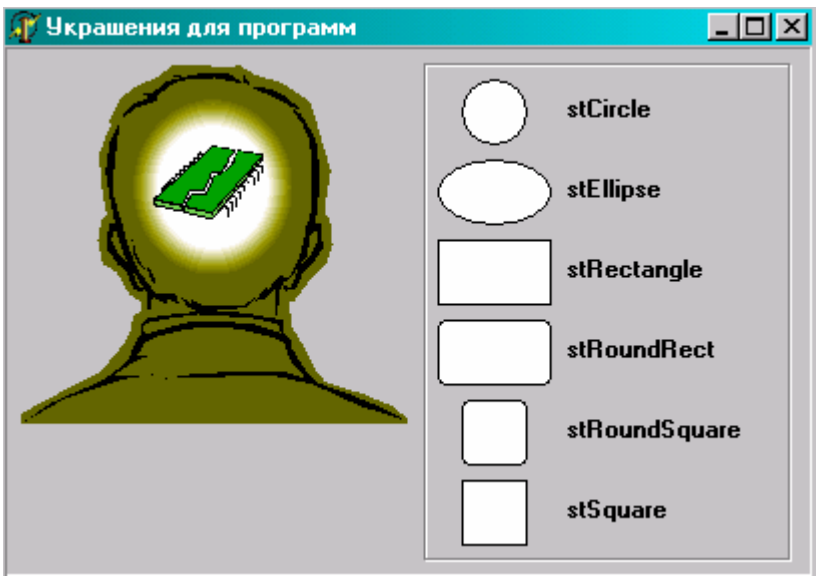


Рис 11.5.6 Окончательное окно

На компакт диске, в директории ***\Примеры\Глава 11\TImage TShape TBevel*** *ты можешь увидеть пример этой программы.*

11.6 Панель с полосами прокрутки (TScrollBox)

Теперь я хочу тебе рассказать про компонент TScrollBox. В заголовке этой главы я назвал его как панель с полосами прокрутки. Это не совсем точный перевод названия компонента, но я решил назвать его именно так, потому что эта название отражает суть выполняемых компонентом действий.

- TScrollBox

Создай новое приложение. Теперь установи компонент ***TScrollBox*** на форму. Теперь брось на компонент *ScrollBox* картинку (***TImage***). Теперь загрузи в *Image1* картинку большого размера, чтобы она не помещалась в пределы экрана, и установи свойство *AutoSize* в *true*. В этот момент компонент *Image1* должен увеличиться до реальных размеров картинки. Если он не будет помещаться в пределы *ScrollBox*, то появятся полосы прокрутки и ты сможешь прокрутить изображение.

Рис 11.6.1 Окно формы

На компакт диске, в директории ***\Примеры\Глава 11\ScrollBox*** *ты можешь увидеть пример этой программы.*

11.7 Маркированный список (TCheckListBox)

TCheckListBox очень похож на простой ***TListBox***, только у каждого элемента списка есть ещё и квадратик для выделения как у ***TCheckBox***. На рисунке 1.7.1 показан пример компонента ***TCheckListBox***.

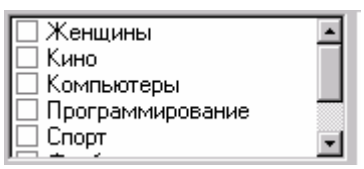


Рис 11.7.1 Компонент TCheckListBox

Давай создадим пример, который будет работать с этим компонентом. Создай новое приложение в Delphi и брось на него компонент ***TCheckListBox***. Теперь дважды щёлкни по свойству *Items* и перед тобой появиться редактор элементов списка (рисунок 1.7.2). Введи там несколько строк на свой выбор.

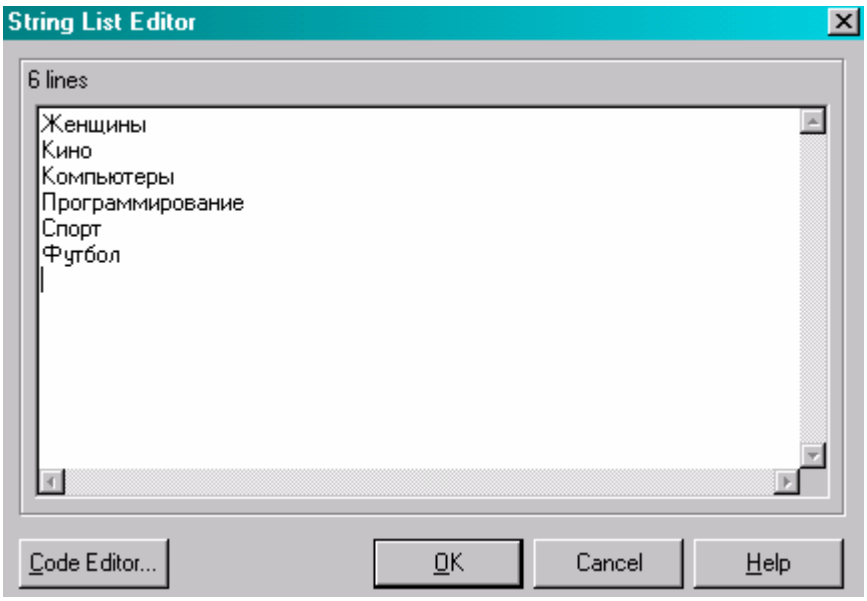


Рис 11.7.2 Редактор строк маркированного списка

У ***TCheckListBox*** есть ещё одно интересное свойство – *columns*, т.е. количество колонок в списке. Если ты укажешь здесь число большее 1, и твой список не будет помещаться в одну колонку, то элементы будут разбиты на указанное количество колонок (см. рисунок 11.7.3).

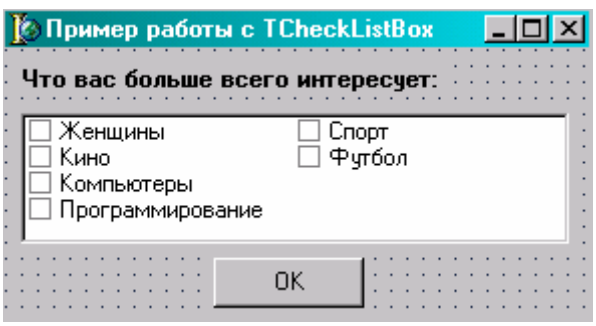


Рис 11.7.3 Список, разбитый на несколько колонок

На рисунке 11.7.3 ты можешь видеть ещё и кнопку *OK*. Добавь её на свою форму. По нажатию этой кнопки мы будем проверять, какие элементы выделил пользователь и сообщать об этом. Создай обработчик события *OnClick* для кнопки и напиши там следующее:

```
procedure TForm1.OKButtonClick(Sender: TObject);
var
 i:Integer;
 Str:String;
begin
 Str:='Вы выбрали ';
 for i:=0 to CheckListBox1.Items.Count-1 do // Запускаю цикл
  if CheckListBox1.Checked[i] then //Если i-й элемент выделен то …
   Str:=Str+CheckListBox1.Items[i]+' '; //Добавить в строку Str текст элемента

 Application.MessageBox(PChar(Str), 'Внимание!!!'); // Вывести на экран строку
end;
```

Теперь можешь запустить приложение и посмотреть на результат работы. На рисунке 11.7.4 показан примерный результат.

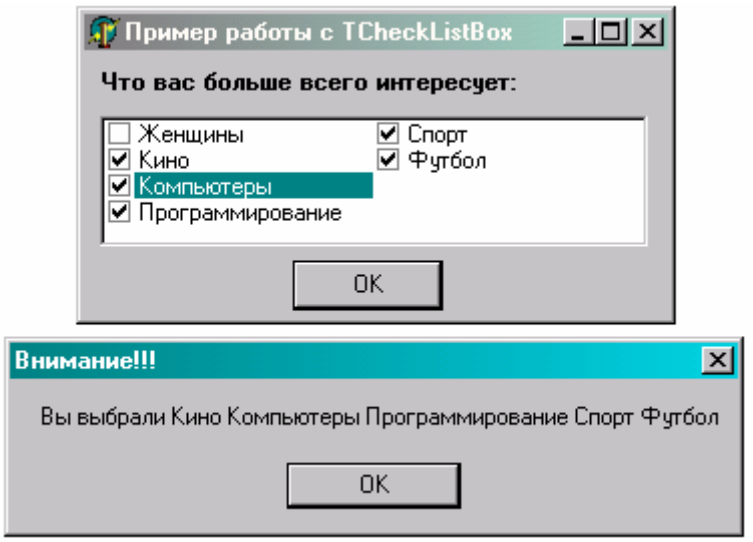


Рис 11.7.4 Результат работы программы

Теперь разберёмся, что же происходит по нажатию заветной кнопки «*OK*». Я объявляю две переменные – целое число i и строку Str. В первой троке кода я присваиваю строке Str текст «Вы выбрали». После этого я запускаю цикл, в котором проверяю все элементы. Если i-й элемент выделен, то добавляю текст строки к переменной Str.

Чтобы узнать, выделена ли какая-то строка, надо проверить свойство Checked компонента CheckListBox1. В квадратных скобках надо указать номер интересующей тебя строки. Например, если ты хочешь проверить нулевую строку, то надо написать:

```
if CheckListBox1.Checked[0] then
…
```

Я перебираю все элементы, поэтому в квадратных скобках указываю параметр **i**. Текст строки можно узнать в свойстве Items компонента CheckListBox1. Чтобы узнать текст нулевой строки надо написать:

```
CheckListBox1.Items[0]
```

*На компакт диске, в директории **\Примеры\Глава 11\CheckListBox** ты можешь увидеть пример этой программы.*

11.8 Полоса разделения (TSplitter)

- TSplitter

Запусти проводник Windows Explorer. Посмотри на его главное окно, которое разбито на две части. Слева ты можешь видеть список дисков и директорий, а справа находятся файлы из выбранной папки. Между двумя половинами окна находиться полоска, которую можно двигать (смотри рисунок 11.8.1), увеличивая или уменьшая одну из половин окна. Вот именно такой эффект легко создать с помощью компонента ***TSplitter***.

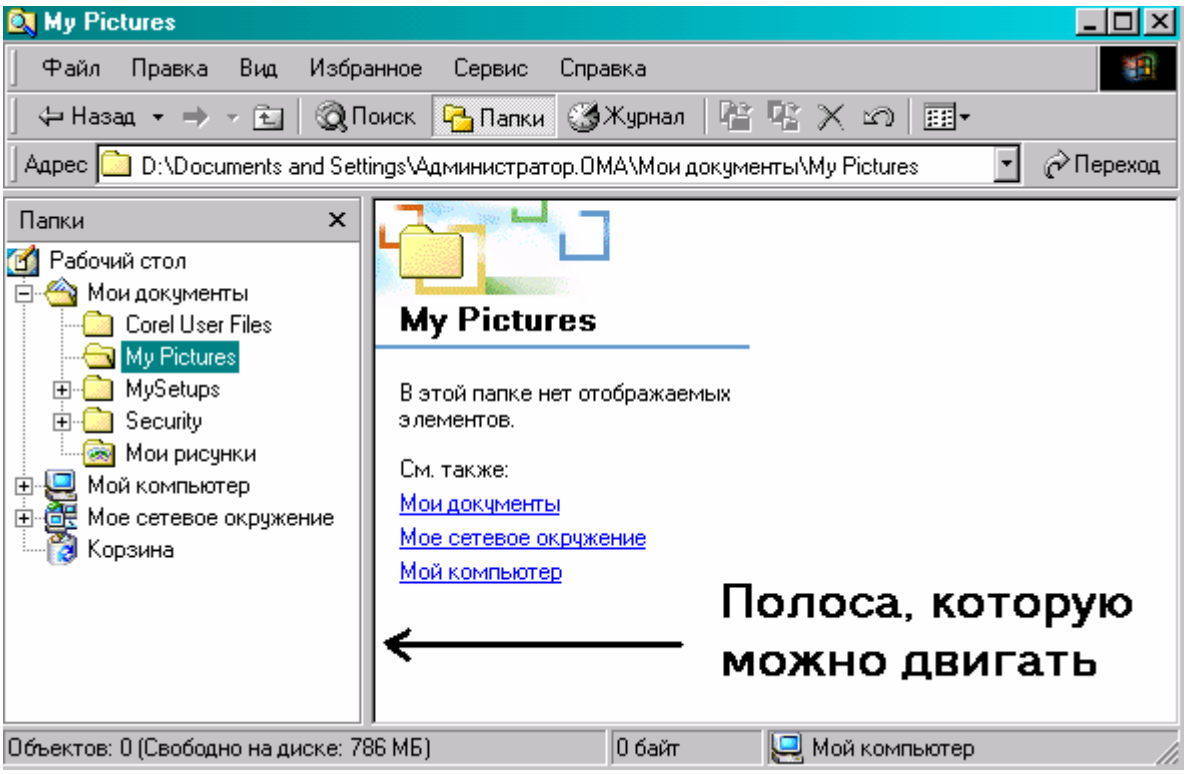


Рис 11.8.1 Проводник Windows

У компонента ***TSplitter*** не так уж и много свойств, поэтому я не буду долго заострять на нём внимания, а просто покажу тебе пример работы с ним.

Создай новое приложение. Теперь брось на форму компонент панели (***TPanel***) и растянем его по верхнему краю формы (установи у него свойство *Align* в *alTop*). В свойстве *Caption* напишем «Верхняя панель». Теперь бросим на форму ***TSplitter*** и у него тоже установим в свойстве *Align* значение *alTop*.

Теперь брось ещё одну панель и установи у неё выравнивание по левому краю. В свойстве *Caption* напиши «Левая панель». Бросим ещё один TSplitter и тоже установим выравнивание по левому краю

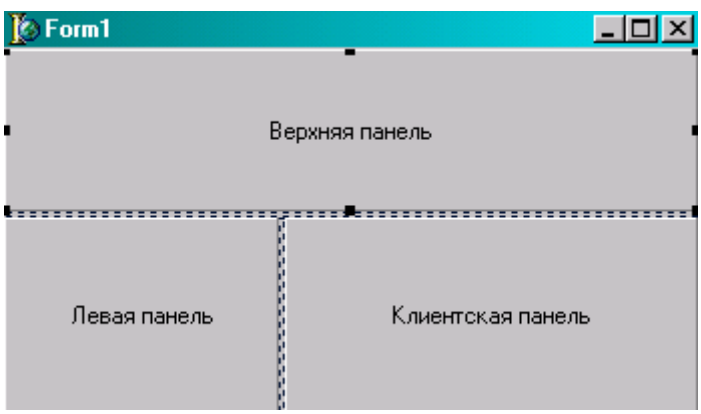


Рис 11.8.2 Форма будущей программы

Ну и наконец последняя панель с выравниванием по всей оставшейся площади формы (свойство *Align* должно быть *alClient*). Ну а в свойстве *Caption* напишем «Клиентская панель».

Если ты всё сделал правильно, то у тебя должно получиться что-то похожее на рисунок 11.8.2 – три панели и между ними разделители ***TSplitter***. Попробуй запустить эту программу и подвигать мышкой разделители. Размеры панелей будут меняться автоматически, что очень удобно для большинства программ.

 *На компакт диске, в директории* ***\Примеры\Глава 11\Splitter*** *ты можешь увидеть пример этой программы.*

11.9 Многострочный текст (TStaticText)

Иногда бывает необходимость создать текст в нескольких строчках. Для этого можно поставить на форму в столбик несколько компонентов ***TLabel***, а можно поступить лучше – использовать компонент ***TStaticText***. Если установить его на форму и отключить свойство *AutoSize*, то компонент не будет автоматически принимать размеры введённого текста, а если введённый текст не вмещается, то он будет разбит на несколько строк.

 **- TstaticText**

На рисунке 11.9.1 ты можешь видеть пример компонента в действии. Я думаю, что не надо писать отдельного примера, потому что в остальном ***TStaticText*** – это полная копия компонента ***TLabel***.

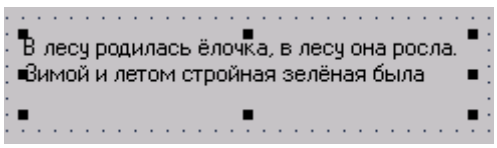


Рис 11.9.1 Компонент TStaticText

11.10 Редактор параметров (TValueListEditor)

В этой части книги мы рассмотрим компонент ***TValueListEditor.*** Это очень удобный компонент для редактирования различных свойств. Например, с помощью этого компонента можно легко создать редактор свойств наподобие того, что используется в объектном инспекторе, где ты изменяешь свойства компонентов.

 **- TValueListEditor**

Создай новый проект и брось на форму один такой компонент. Рассмотрим его основные свойства. В дизайнере тебе доступны несколько интересных вещей:

- *DefaultColimnWidth* – ширина колонок по умолчанию;
- *DefaultColimnHeight* – высота колонок по умолчанию;
- *DisplayOption* – опции отображения компонента. Здесь тебе доступны три подпункта:
  - *doColumnTitles* – нужно ли показывать заголовки колонок;
  - *doAutoColResize* – могут ли колонки автоматически изменять размер;
  - *doKeyColFixed* – являеться ли ключевая колонка фиксированной;

- *TitleCaptions* - имена заголовков. Щёлкни дважды по этому свойству, и ты увидишь простой текстовый редактор, в котором можешь изменять имена заголовков. Я ввёл там только два заголовка - "свойство" и "Установленное значение".
- *FixedColor* - Цвет фиксированной колонки.

• *FixedCols* - Индекс фиксирванной колонки. По умолчанию стоит 0, т.е. фиксированной колонки нет. Измени это значение на 1, чтобы сделать первую колонку фиксированной.

• *KeyOption* – настройки ключевого поля. Их бесполезно менять, если ты установил в свойстве *FixedCols* какое-либо значение. Но здесь доступны следующие подпункты:

- *keyEdit* – название ключа можно редактировать
- *keyAdd* – ключи можно добавлять
- *keyDelete* – ключи можно удалять
- *keyUnique* – ключ должен быть уникальным

• *Strings* - Имена свойств (см рисунок 11.10.1). Этот редактор состоит из двух колонок. В первой колонке ты должен вводить имена ключей (свойств), а в правой их значения по умолчанию. Здесь я уже забил кучу значений:

- Фамилия
- Имя
- Отчество
- Ник
- Год рождения
- Место рождения
- Адрес
- Телефон

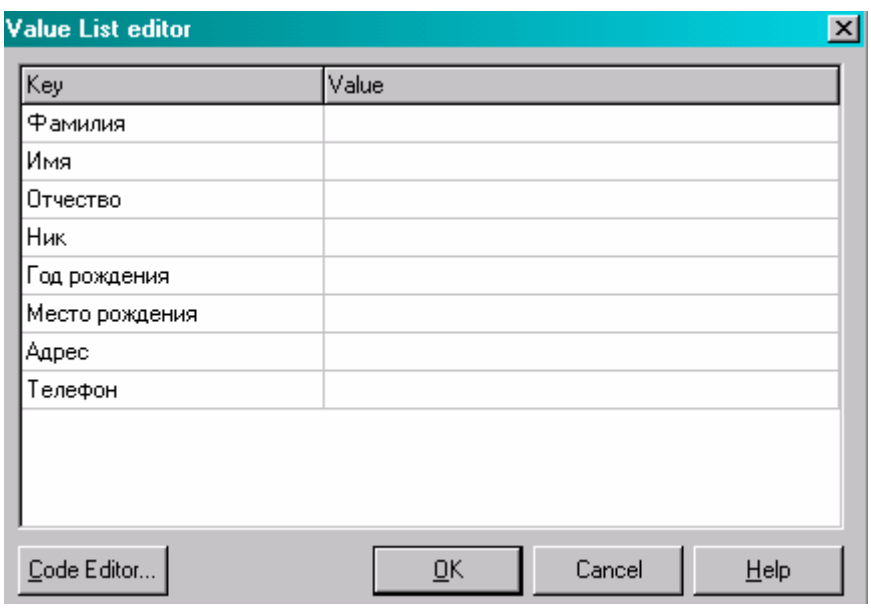


Рис 11.10.1 Редактор строк

Больше в дизайнере пока нет ничего особо заслуживающего нашего внимания. На рисунке 11.10.2 показан рисунок формы, которая должна у тебя получиться.

Списки свойств имеют одну очень удобную особенность – они могут поддерживать свойства с выпадающими списками. Это значит, что нужно указать, какое свойство будет иметь выпадающий список, заполнить его значения и после этого оно будет работать как свойство выравнивания любого компонента в объектном инспекторе.

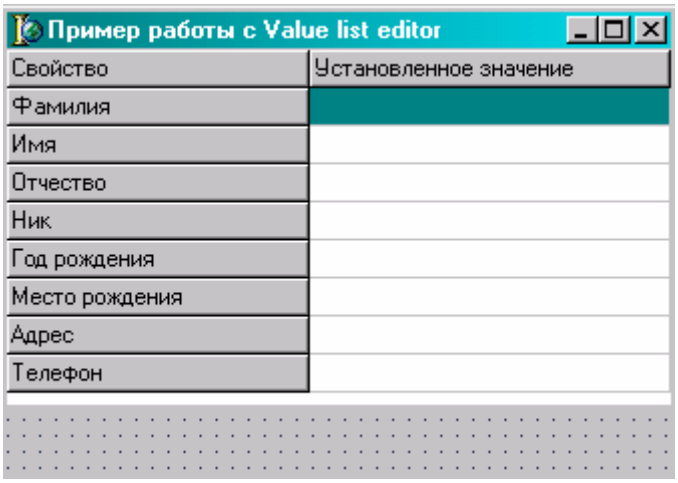


Рис 11.10.2 Форма будущей программы

Теперь давай немного попрограммируем. Создай обработчик события для формы OnShow. В нём мы напишем следующее:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 ValueListEditor1.ItemProps[6].EditStyle:=esPickList;
 ValueListEditor1.ItemProps[6].PickList.Add('Москва');
 ValueListEditor1.ItemProps[6].PickList.Add('Питер');
 ValueListEditor1.ItemProps[6].PickList.Add('Ростов-на-Дону');

 ValueListEditor1.ItemProps[4].EditMask:='99/99/9999';
end;
```

У компонента *ValueListEditor* есть одно свойство, которое ты не видишь в объектном инспекторе – *ItemProps*. В нём хранятся свойства элементов списка. Если ты хочешь изменить свойства 3-го элемента, то надо написать *ValueListEditor1.ItemProps[2]*. Обрати внимание, как и в большинстве компонентов, здесь элементы нумеруются с нуля. Поэтому нужно указывать на 1 меньше необходимого.

Среди свойств элементов есть ещё одно интересное свойство – *EditStyle* – стиль редактирования. Это свойство отвечает за вид элемента. В первой строке кода я изменяю стиль редактирования для 6-й строки (*ValueListEditor1.ItemProps[6].EditStyle*) присваивая ему значение *esPickList*. Это значение заставляет эту строку превратиться в выпадающий список. После этого я заполняю для 6-го элемента элементы, которые будут находиться в выпадающем списке. Для этого я выполняю код *ValueListEditor1.ItemProps[6].PickList.Add(текст элемента)*.

Ещё одно интересное свойство – *EditMask* – маска ввода для элемента. В последней строчке кода моего примера я изменяю маску для 4-го элемента.

*На компакт диске, в директории **\Примеры\Глава 11\ValueListEditor** ты можешь увидеть пример этой программы.*

11.11 Набор закладок (TTabControl)

Иногда на форме надо иметь возможность показывать несколько закладок, на каждой из которых будут располагаться разные компоненты. Такое очень часто можно увидеть в окнах настройки программ, например в MS Word. На рисунке 11.11.1 показано окно «Параметры» MS Word, у которого сверху расположено множество закладок. В зависимости от выбранной в данный момент закладки, в основном окне отображаются относящиеся к ней свойства.

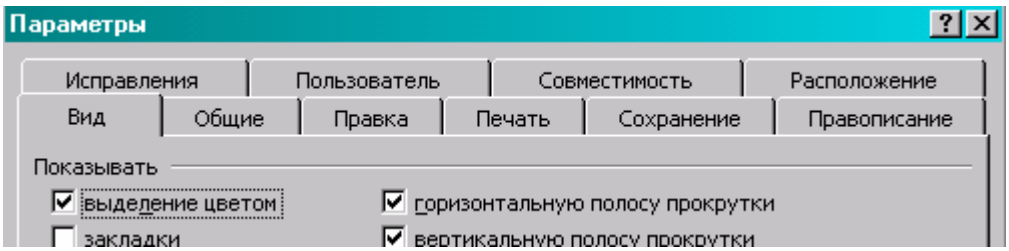


Рис 11.11.1 Окно «Параметры» MS Word

В Delphi, на закладке Win32 палитры компонентов есть два компонента позволяющие создавать подобные закладки. Сейчас мы рассмотрим первый из них - ***TTabControl***.

- TTabControl

Запусти Delphi и создай новое приложение. Сейчас мы как всегда напишем пример и посмотрим на возможности этого компонента. Брось на форму компонент ***TTabControl*** и растяни его на всю форму. Давай сразу же изменим имя компонента (свойство *Name*) на *OptionsTab*.

Теперь пора создать сами закладки. Для этого дважды щёлкни по свойству *Tabs*, и перед тобой появиться уже знакомое окно с простеньким текстовым редактором (Рис 11.11.2). Мы уже много раз работали с таким окном в других компонентах, поэтому оно не должно вызвать страха.

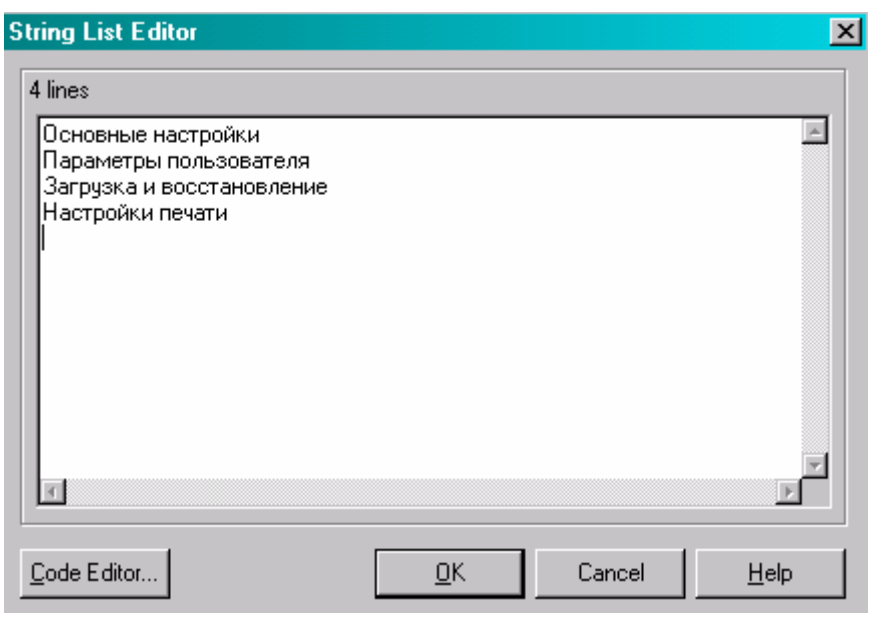


Рис 11.11.2 Окно ввода имён закладок

В этом окне давай введём четыре строки:

- Основные настройки
- Параметры пользователя
- Загрузка и восстановление
- Настройки печати

После нажатия кнопки «*ОК*», на компоненте должны появиться закладки с введёнными названиями (рисунок 11.11.3).

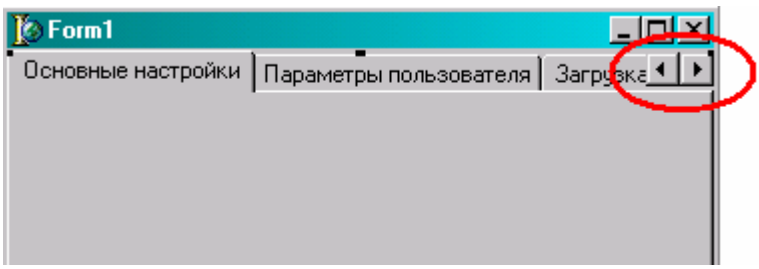


Рис 11.11.3 Компонент TabControl с созданными закладками

Обрати внимание, что мы ввели четыре названия закладки, а у меня на рисунке видно только три. Четвёртая закладка не поместилась, поэтому справа от имён появились две кнопки для скроллинга названий закладок.

Давай установим свойство *MulriLine* в *true*. Результат этого ты можешь увидеть на рисунке 11.11.4. Как видишь, кнопки скроллинга исчезли, зато названия выстроены в две строчки. Что тебе больше подходит – зависит от конкретного предназначения компонента. Иногда нужны многострочные закладки, а иногда они просто мешаются.

Рис 11.11.4 Компонент TabControl с установленным свойством MultiLine

Давай ещё установим свойство *HotTrack* в *true*. Это заставит названия закладок светиться при наведении на них мышью (это можно увидеть, если запустить приложение).

У компонента *TabControl* есть ещё одно интересное свойство – *Style*. Это свойство отвечает за стиль отображения закладок. Здесь можно выбрать из списка одно из следующих значений:

tsTabs – пример таких закладок можно увидеть на рисунках 11.11.3 или 11.11.4

tsButtons – пример таких закладок показан на рисунке 11.11.5

tsFlatButtons – пример таких закладок показан на рисунке 11.11.6

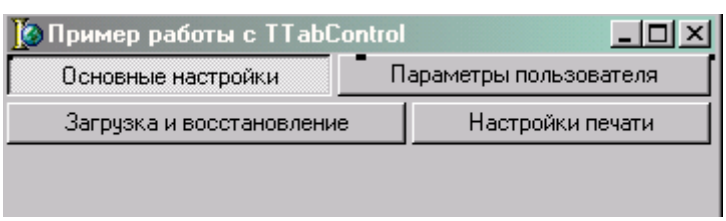


Рис 11.11.5 Закладки в стиле tsButtons

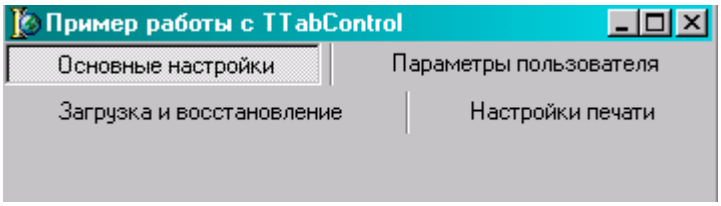


Рис 11.11.6 Закладки в стиле tsFlatButtons

Я не буду менять это свойство и оставлю его по умолчанию.

И перед тем, как мы попробуем создать что-то реальное, я покажу ещё три интересных свойства – *TabHeight*, *TabIndex* и *TabPosition*.

TabHeight – высота кнопок закладок. Если здесь указать 0, то будет использоваться значение по умолчанию.

TabIndex – индекс выделенной сейчас закладки. Номера закладок нумеруются как всегда с нуля, поэтому в нашем случае можно выставлять значения от 0 до 3. Изменяя значение установленное здесь, ты можешь менять выделенную закладку. Ну а когда приложение запущено, по этому свойству можно определять, какую закладку выбрал сейчас пользователь.

TabPosition – позиция закладок. Здесь можно выбрать из списка одно из следующих значений:

tpBottom – закладки должны быть расположены снизу;

tpLeft – закладки должны быть расположены слева;

tpRight – закладки должны быть расположены справа;

tpTop – закладки должны быть расположены сверху;

По умолчанию здесь установлено значение *tpTop*.

Теперь попробуем создать реальное приложение. Попробуй бросить на любую из закладок какой-нибудь компонент и запустить приложение или просто изменить индекс выделенной закладки. Если ты сделаешь это, то заметишь одно неудобство – компонент не привязывается к какой-нибудь закладке. Когда ты выбираешь любую закладку, компонент остаётся видимым. Это значит, что нам самим нужно прятать и показывать компоненты в зависимости от выбранной в данный момент закладки. Сейчас я попробую написать пример, в котором покажу простейший способ избавиться от этого недостатка.

Рис 11.11.7 Форма с четырьмя панелями

Брось на форму 4-е панели и постарайся их расположить рядом, примерно как на рисунке 11.11.7. Это нужно, чтобы ни одна панель случайно не попала поверх другой. Все они должны лежать на компоненте *OptionTab* (это наш ***TTabControl***). Посмотри на окно Object TreeView (рисунок 11.11.8). В нём показана иерархия компонентов, что и на чём лежит.

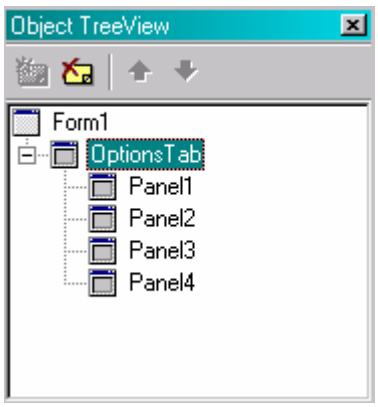


Рис 11.11.8 Иерархия компонентов

Измени у всех панелей свойство *BevelOuter* на *bvLowered*, это сделает панели более приятными на глаз. И ещё, у компонентов *Panel2*, *Panel3* и *Panel4* установи свойство Visible в *true*. Видимой должна остаться только первая панель.

Теперь очисти у всех свойство *Caption* и растяни на всю форму. При растягивании компонентов можно поступить несколькими способами:

1. Выбрать каждый компонент в отдельности в окне Object TreeView (или в выпадающем списке сверху окна объектного инспектора) и устанавливать у него свойство *Align* в *alClient*.
2. Выделить все панели (удерживая клавишу Shift щёлкнуть по всем панелям) и потом у всех сразу выставить свойство *Align* в *alClient*.

Теперь все панели находятся точно друг под другом. Давай бросим на каждую из панелей надпись, что эта такая-то панель. Если у тебя сейчас наверху находиться не 1-я панель, то щёлкни правой кнопкой и выбери из появившегося меню пункт *Control* и подпункт *Send to Back*. Повторяй эти действия, пока наверху не окажется 1-я панель. Брось на эту панель надпись «Здесь можно располагать компоненты для основной настройки».

После этого можешь выделить следующую панель и бросить на неё любые другие компоненты. Желательно таким образом заполнить все панели, бросив на них хотя бы по одному компоненту, чтобы ты смог увидеть, как они будут меняться.

Теперь создай обработчик события *OnChange* для компонента *OptionTab* и в нём напиши:

```
procedure TForm1.OptionsTabChange(Sender: TObject);
begin
 Panel1.Visible:=false;
 Panel2.Visible:=false;
 Panel3.Visible:=false;
 Panel4.Visible:=false;

 case OptionsTab.TabIndex of
  0: Panel1.Visible:=true;
```

```
  1: Panel2.Visible:=true;
  2: Panel3.Visible:=true;
  3: Panel4.Visible:=true;
 end;
end;
```

Этот обработчик вызывается каждый раз, когда пользователь пытается изменить закладку. В самом начале я делаю невидимыми все панели, а потом проверяю, какая именно закладка выделена с помощью оператора case и в зависимости от этого делаю выделенной конкретную панель. Например, если выделена вторая закладка (в *OptionsTab.TabIndex* находиться 1), то видимой станет *Panel2*.

На компакт диске, в директории ***\Примеры\Глава 11\TabControl*** *ты можешь увидеть пример этой программы.*

11.12 Набор страниц (TPageControl)

В прошлой части мы познакомились с компонентом ***TTabControl***. Он достаточно хорош, но работать с ним очень неудобно, потому что постоянно приходиться самому следить, какая сейчас выделена закладка, и в зависимости от этого отображать нужные компоненты. Всех этих недостатков лишён компонент ***TPageControl***, который так же находиться на закладке *Win32* палитры компонентов.

- TPageControl

Компонент ***TPageControl*** обладает практически всеми свойствами ***TTabControl***, плюс несколько дополнительных. Давай посмотрим на него в работе.

Создай новое приложение и брось на форму компонент ***TPageControl***. В этот раз я не стал менять его имя и оставил значение по умолчанию *PageControl1*. Единственное, что я сделал – растянул его на всю форму.

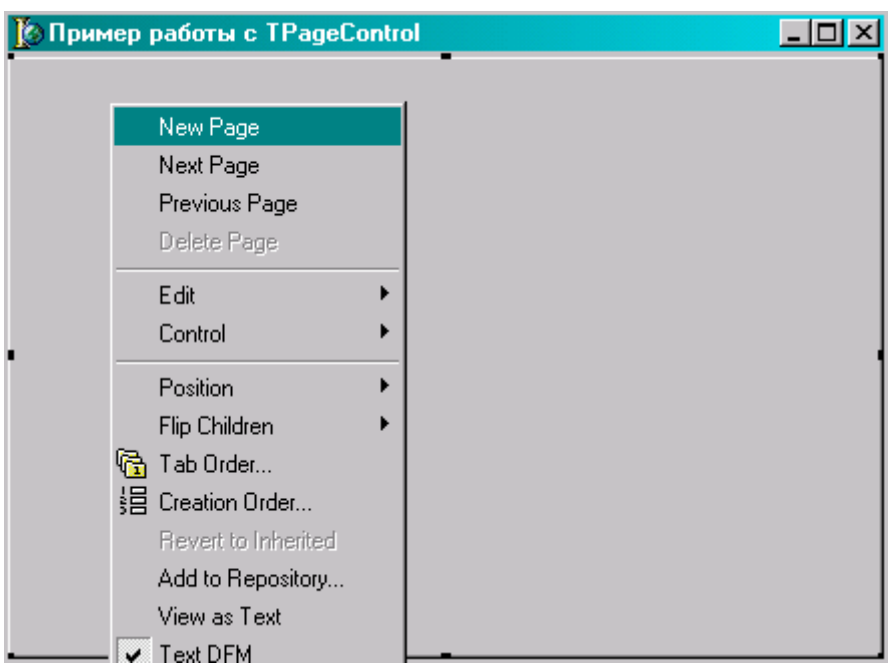


Рис 11.12.1 Меню управления страницами

Щёлкни правой кнопкой мыши по компоненту и перед тобой откроется меню, как на рисунке 11.12.1. Сверху этого меню находиться 4 пункта, с помощью которых можно управлять страницами:

New Page – создать новую страницу (закладку);
Next Page – перейти на следующую страницу (закладку);
Previous Page – перейти на предыдущую страницу (закладку);
Delete Page – удалить выделенную страницу (закладку).

Создай новую страницу. Теперь посмотри в объектный инспектор (рисунок 11.12.2). Обрати внимание, что сверху, в выпадающем списке сейчас показывается выделенный компонент *TabSheet1* типа ***TTabSheet*** – это созданная нами страница. Получается, что когда мы создаём новую страницу, то, как бы создаём отдельный компонент внутри компонента ***TPageControl***. Именно поэтому ***TPageControl*** лишён недостатков компонента ***TTabControl***. Каждая его страница – это отдельный объект внутри целого компонента ***TPageControl***. Если в прошлый раз нам самим приходилось делать что-то подобное с помощью панелей, то тут это делается автоматически.

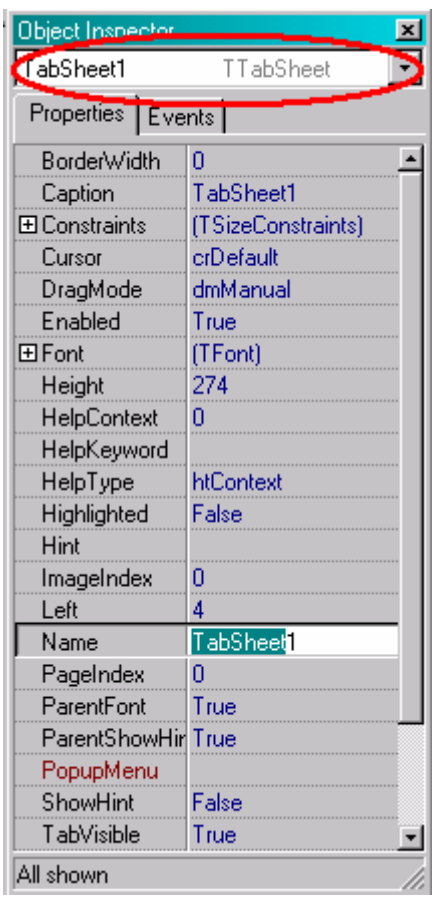


Рис 11.12.2 Свойства страницы

У каждой страницы есть свойство *Caption*, в котором можно написать заголовок страницы. Помимо этого, есть свойство *ImageIndex*, в котором можно выбирать картинку, как мы это делали при создании меню. Для этого нужно бросить на форму компонент ***TImageList*** и загрузить в него картинки. После этого нужно выбрать наш компонент *PageControl* и указать в его свойстве *Images* компонент *ImageList*. После этого в списке

ImageIndex у страниц появятся картинки. Как видишь, весь этот процесс похож на тот, что мы делали при создании меню. Попробуй его проделать самостоятельно.

Давай создадим четыре закладки, как в прошлом примере с именами:

- Основные настройки
- Параметры пользователя
- Загрузка и восстановление
- Настройки печати

На рисунке 11.12.3 показана форма моей будущей программы. Попробуй создать нечто подобное. Практически всё, что необходимо для этого знать, мы уже проходили. Я бросил на каждую закладку несколько компонентов, чтобы можно было видеть, как меняются страницы. Они ничего не делают, а служат просто оформлением для большей наглядности.

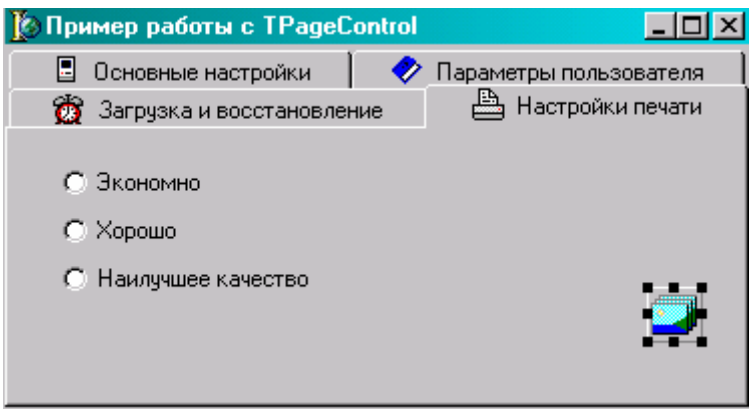


Рис 11.12.3 Форма будущей программы

*На компакт диске, в директории **\Примеры\Глава 11\PageControl** ты можешь увидеть пример этой программы.*

11.13 Набор картинок (TImageList)

Наборы картинок используются для удобного хранения изображений. Мы уже не раз использовали наборы картинок (***TImageList***) и здесь я только объясню некоторые специфичные вещи, которые могут тебе понадобиться. Никаких примеров здесь я писать не буду, а только дам небольшие пояснения к уже не раз сказанному.

- TImageList

В свойствах Height и Width находиться ширина и высота хранящихся картинок. Когда ты добавляешь новую, то она обязательно приводиться к указанным размерам.

Ты можешь программно доставать любую картинку из массива с помощью метода *GetBitmap*. У этого метода есть два параметра:

1. Индекс картинки, которую надо получить.
2. Объект типа TBitmap, куда запишется результирующая картинка.

Например, чтобы получить четвёртую картинку из массива нужно написать так:

```
var
 bitmap:TBitmap;
begin
 ImageList1.GetBitmap(3, bitmap);
end;
```

Обрати внимание, что для выборки 4-й картинки надо указать цифру 3, потому что картинки, как и большинство массивов нумеруются начиная с 0.

Это пока всё, что я хотел сказать. Я не хочу сейчас глубоко углубляться в эту тему, потому что тут надо сначала объяснить работу с графикой, а это тема следующей главы.

11.14 Ползунки (TTrackBar)

Ползунки ***TTrackBar*** чаще всего используются, когда надо дать пользователю выбрать какое-то значения из определённого диапазона. Например, ты наверно не раз пользовался архиваторами, так вот там степень сжатия устанавливается вот таким ползунком. Хочешь ещё пример? Вспомни, как выглядит регуляторы громкости в любой из программ. Чаще всего это опять же будут ползунки.

 **- TTrackBar**

Самый простейший ползунок выглядит, как на рисунке 11.14.1.

Рис 11.14.1 Простейший ползунок

У ползунка есть следующие интересные свойства:

Frequency – этот параметр показывает, как часто надо рисовать риски значений. Допустим, что у тебя ползунок может принимать значения от 0 до 10. Если указать в этом свойстве 2, то будут нарисованы только 5 рисок (рисуется каждая вторая риска), если указать 3, то будет рисоваться каждая третья риска.

Max – максимальное значение ползунка.

Min – минимальное значение ползунка.

Orientation – вид ползунка. В этом свойстве выбор значений производиться с помощью выпадающего списка в котором можно выбрать одно из двух: trHorizontal (ползунок горизонтальный) или trVertical (ползунок вертикальный).

Position – текущая позиция ползунка.

SelStart – в ползунке может быть выделено определённое число значений и это свойство указывает на начало выделения.

SelEnd – конец выделения.

SliderVisible – должен ли быть виден бегунок.

TickMarks – где рисовать риски. Здесь доступны следующие значения:

- *tmBottomRight* – снизу.
- *tmBoth* – снизу и сверху.
- *tmTopLeft* – сверху.

TickStyle – стиль рисок. Здесь доступны следующие значения:

- *tsAuto* – риски рисуются автоматически.
- *tsManual* – рисуется только начальная и конечная риска.
- *tsNone* – риски вообще не рисуются.

Попробуй сам поиграть с этими свойствами, и посмотреть на результат. Я могу долго тебе рассказывать про разные варианты, но пока ты не реально увидишь сам, мои слова тебе не помогут.

Теперь мы готовы написать простенький пример. Для этого я создал три ползунка разной формы и бросил три TLabel с именами Label1, Label2 и Label3 (форма показана на рисунке 11.14.2).

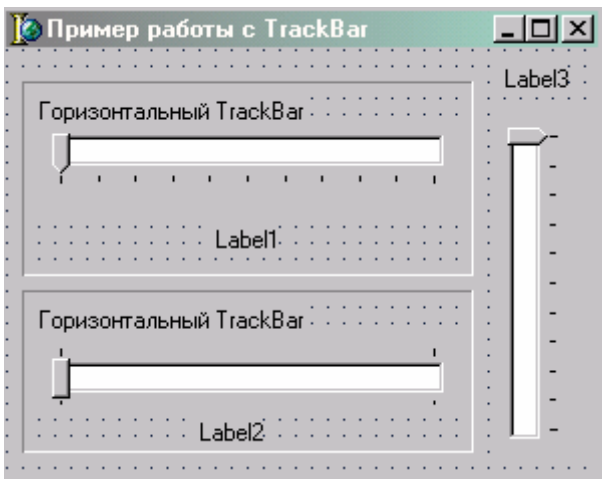


Рис 11.14.2 Форма будущей программы

Теперь я создал обработчик события *OnChange* для первого ползунка и написал там следующий код:

```
procedure TForm1.TrackBar1Change(Sender: TObject);
begin
 Label1.Caption:=IntToStr(TrackBar1.Position);
end;
```

Здесь я преобразовываю текущую позицию ползунка (*TrackBar1.Position*) в строку (потому что позиция имеет тип целого числа) с помощью функции *IntToStr*, и присваиваю результат в Label1. Таким образом, после изменения позиции ползунка я сразу отображаю текущую позицию.

Подобный код написан и для остальных ползунков. Попробуй написать его сам или посмотри его в исходнике.

*На компакт диске, в директории **\Примеры\Глава 11\TrackBar** ты можешь увидеть пример этой программы.*

11.15 Индикация состояния процесса (TProgressBar)

Я очень долго не мог придумать русское название компоненту ***TProgressBar***, потому что ни одно из названий пришедших мне в голову, не могут отразить реального положения. В конце концов, я остановился на понятии «Индикация состояния процесса», потому что именно для этих целей, чаще всего применяется ***TProgressBar***. Вспомни любое окно копирования файлов или любых других данных. Практически в любом таком окне есть бегунок, который показывает, сколько процентов сейчас выполнено.

 **- TProgressBar**

У этого компонента есть три необходимых свойства:
Max – максимальное значение (по умолчанию = 100).
Min – минимальное значение (по умолчанию = 0).
Position – позиция.

Давай рассмотрим пару примеров. Допустим, что тебе нужно вычислить в цикле 100 чисел. В этом случае очень удобно поставить на форму компонент ***TProgressBar*** и отображать в нём текущее вычисляемое значение. Давай рассмотрим общий пример такого случая на реальном примере кода.

Брось на форму одну кнопку и компонент ***TProgressBar***. Теперь по нажатию кнопки напиши следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i:Integer;
begin
 for i:=0 to 20 do
  begin
   //Здесь можно делать какой-то расчёт

   //После расчёта отображаем текущее состояние
   ProgressBar1.Position:=ProgressBar1.Position+5;
   Sleep(100); //Делаю задержку в 100 миллисекунд
  end;
 ProgressBar1.Position:=0;
end;
```

В данном случае я запускаю цикл от 0 до 20. На каждом этапе цикла позиция *ProgressBar* увеличивается на 5 и на двадцатом шаге выполнения цикла будет равна своему максимальному значению – 100. В цикле я также устанавливаю задержку на 100 миллисекунд, чтобы наша полоска не проскочила слишком быстро, и ты хотя бы увидел иллюзию расчёта. В реальных примерах задержка не будет нужна, ведь если расчёт выполняется так быстро, что пользователь даже не увидит движение ползунка, то нет смысла создавать *ProgressBar*.

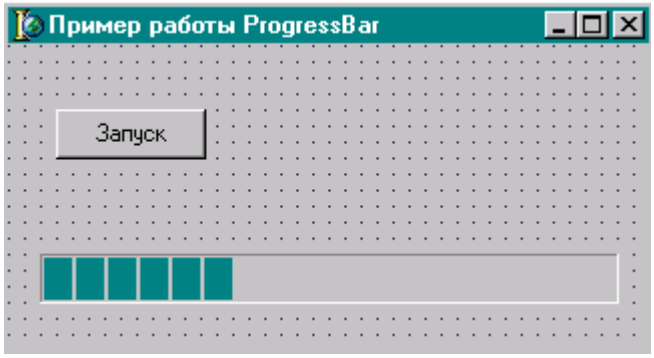


Рис 11.15.1 Форма рабочей программы

Приведённый пример не совсем универсальный, потому что требует, чтобы мы заранее знали приращение 5 единиц на каждом шаге. Тут есть два варианта решения:

1. Изменить свойство Max компонента *ProgressBar* на 20 и на каждом шаге приращивать только единицу. Это очень хороший способ, потому что позиция *ProgressBar* будет изменяться от 0 до 20, и цикл тоже действует в этом диапазоне, так что пример может упроститься до следующего:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i:Integer;
begin
 for i:=0 to 20 do
  begin
   //Здесь можно делать какой-то расчёт

   //После расчёта отображаем текущее состояние
   ProgressBar1.Position:=i;
   Sleep(100); //Делаю задержку в 100 миллисекунд
  end;
 ProgressBar1.Position:=0;
end;
```

В данном случае нам не надо делать приращение для *ProgressBar*, а достаточно только сразу присваивать в свойство позиции значение i, потому что значение позиции и значение i изменяются в одном диапазоне от 0 до 20.

2. Второй способ заключается в расчёте относительного положения состояния ProgressBar. В этом случае позиция ProgressBar должна изменяться от 0 до 100 и нужно воспринимать эти значения как проценты. После этого рассчитывать процент выполнения на каждом шаге цикла. Не пугайся, этот расчёт очень прост и не затруднителен для машины:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i:Integer;
begin
 for i:=0 to 20 do
  begin
   //Здесь можно делать какой-то расчёт

   //После расчёта отображаем текущее состояние
```

```
  ProgressBar1.Position:=round(i/20*100);
  Sleep(100); //Делаю задержку в 100 миллисекунд
 end;
 ProgressBar1.Position:=0;
end;
```

В этом примере позиция рассчитывается по формуле преобразования чисел в проценты, т.е. текущее значение делим на максимальное и умножаем на 100 (в нашем случае 20/i и умножить на 100). Результат этих вычислений нужно округлить, потому что среди операций расчёта было деление, значит общий результат будет в любом случае дробным числом.

В этом случае есть небольшая погрешность из-за операции деления и округления, но в реальных условиях заметить эту погрешность невозможно, потому что она равна менее половины процента. Всё равно при выводе графики всегда бывают погрешности из-за масштабирования.

*На компакт диске, в директории **\Примеры\Глава 11\ProgressBar** ты можешь увидеть пример этой программы.*

11.16 Простейшая анимация (TAnimate)

Очень часто в программах надо создавать какую-нибудь анимацию, чтобы пользователь не сильно скучал, пока проходят какие-нибудь долгие расчёты. Например, когда программа копирует большой файл, то желательно вывести какой-нибудь мультик отображающий копирования. В Delphi создание такой анимации - самое простое дело.

 - **TAnimate**

Этот компонент умеет выводить на экран указанную в свойстве FileName анимацию. Дважды щёлкни по этому свойству и перед тобой откроется окно открытия AVI файла. AVI – это стандартный формат видео файлов в Windows. Только не надо думать, что любой такой файл сможет быть проигран на любой машине только из-за того, что он стандартный.

На самом деле AVI – это очень сложная вещь, потому что в нём могут быть запрятано видео любого типа. Формат AVI – это только оболочка, а содержимое может храниться в любом виде. Например, кадры видео файла могут храниться без сжатия, с простым сжатием RLE или даже в сложном MPEG4. Для воспроизведения файла хранящего данные в нестандартном виде используются специальные программы – кодеки, которые должны быть установлены в системе. Так что если ты хочешь быть уверенным, что файл воспроизведётся на любой машине, то можешь поступать следующими способами:

1. Используй не кодированное хранение данных или стандартное Windows кодирование. В этом случае файлы будут достаточно большими, зато воспроизведутся на любой машине.
2. Использовать кодек, поддерживаемый какой-нибудь версией MediaPlayer, а потом только указать, что для нормальной работы проги нужно иметь установленный MediaPlayer определённой версии или выше.

3. Вместе с программой поставлять и кодек. В этом случае нужно копировать на машину клиента не только свою программу, но и файлы кодека. В этом случае кодеки надо не только скопировать, но и установить в системе, чтобы ОС потом смогла их найти при попытке воспроизведения твоего файла.

Но не всё так страшно. Для стандартных операций уже предусмотрены стандартные видео ролики. Их список можно найти в свойстве CommonAVI. Все эти ролики уже установлены в Windows, и их не надо копировать на другую машину вместе с программой и можно быть уверенным, что они воспроизведутся где угодно. Вот список доступных роликов:

aviCopyFile – ролик копирования файла.
aviCopyFiles – ролик копирования нескольких файлов.
aviDeleteFile – ролик удаления файла.
aviEmptyRecycle – ролик очистки корзины.
aviFindComputer – ролик поиска компьютера.
aviFindFile – ролик поиска файла.
aviFindFolder – ролик поиска директории.
aviRecycleFile –отправка файла в корзину.
aviNone – не использовать стандартных роликов.

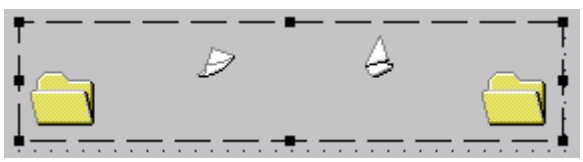


Рисунок 11.16.1 Пример воспроизведения ролика копирования файлов.

Как только ты выбрал AVI файл в свойстве FileName или указал стандартный ролик можно установить свойство Active в true и видео сразу же начнёт воспроизводиться в окне.

11.17 Выпадающий список выбора даты (TDateTimePicker)

На первый взгляд это простейший выпадающий список (TComboBox), но на самом деле, вместо выпадающего списка тут выпадает календарь. Получается очень удобная строка ввода с выпадающим списком в виде календаря.

 **- TDateTimeOicker**

На рисунке 11.17.1 ты можешь увидеть выпадающий список выбора даты в действии.

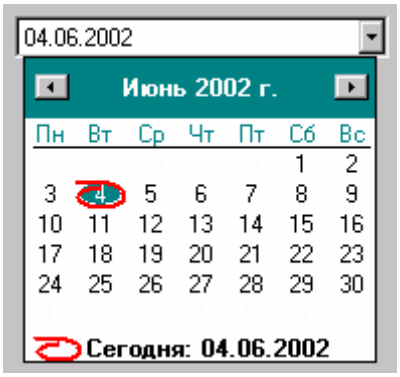


Рисунок 11.17.1 Выпадающий список выбора даты в действии.

У этого компонента большинство свойств схоже с компонентом ***TComboBox***, но есть и свои отличия:

Date – это свойство указывает на выбранную дату.

DateFormat – здесь возможны только два значения: *dfShort* – короткий формат или *dfLong* – длинный формат.

MaxDate – максимальная дата.

MinDate – минимальная дата.

11.18 Календарь (TMonthCalendar)

Предыдущий компонент позволяет выбирать в виде выпадающего списка дату. А что если тебе нужно просто показать календарь? Вот именно для этого и существует ***TMonthCalendar***.

- TMonthCalendar

FirstDayOfWeek – день недели указываемый в качестве первого.

Date – это свойство указывает на выбранную дату.

MaxDate – максимальная дата.

MinDate – минимальная дата.

MultiSelect – есть ли возможность выбирать диапазон чисел месяца.

ShowToday – показывать текущую дату.

ShowTodayCircle – показывать круг текущей даты. По щелчку этого круга календарь перескакивает на текущую дату.

WeekNumbers – показывать номера недель.

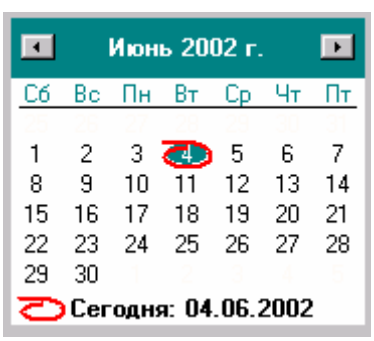


Рисунок 11.18.1 Внешний вид календаря.

На рисунке 11.18.1 ты можешь увидеть внешний вид календаря. Пример я не буду здесь писать, потому что он слишком прост и календарь ещё будет часто использоваться в моих примерах.

11.19 Дерево элементов (TTreeView)

Сейчас нам предстоит познакомиться с достаточно сложным, но мощным компонентом – дерево элементов (***TreeView***). Любая более менее большая программа обязательно использует этот компонент потому что он очень удобен.

 **- TreeView**

Давай сразу напишем пример и познакомимся с деревом на практике. Компонент ***TreeView*** достаточно сложный и с ним нужно знакомиться на практике, чтобы увидеть все прелести работы с ним.

Создай новый проект и брось на него два компонента: *TreeView* и *ImageList*. В список картинок *ImageList* засунь пару любых картинок, потом они пригодятся. А пока добавь ещё три кнопки (*TButton*):

1. Добавить (в свойстве Name укажи AddButton).
2. Добавить элемент (в свойстве Name укажи AddChildButton).
3. Удалить (в свойстве Name укажи DelButton).
4. Изменить заголовок (в свойстве Name укажи EditButton).

В результате у тебя должна быть форма приблизительно такого вида, как показано у меня на рисунке 11.19.1.

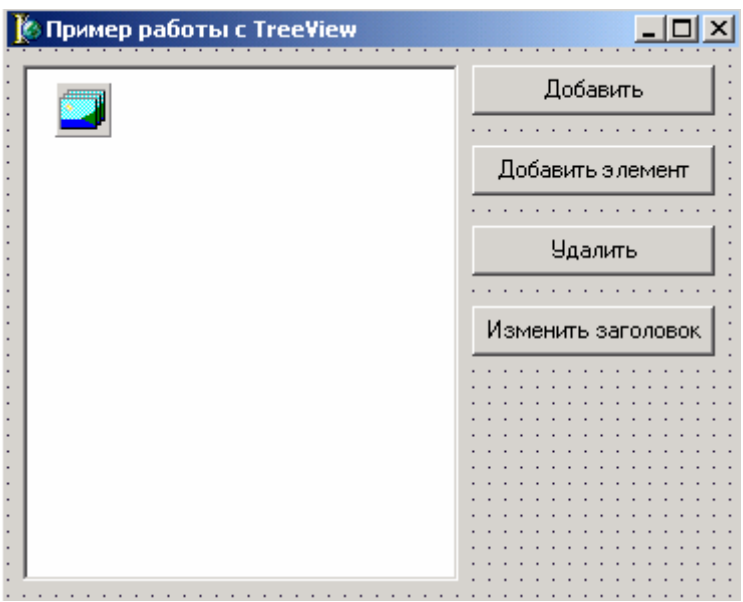


Рисунок 11.19.1. Форма будущей программы

Теперь нужно указать у нашего дерева в свойстве Images установленный набор картинок. После этого можно переходить к кодингу.

По нажатию кнопки «Добавить» мы должны добавлять в дерево новый элемент. Для этого напишем следующий код:

```
procedure TTreeViewForm.AddButtonClick(Sender: TObject);
var
 CaptionStr:String;
```

```
 NewNode:TTreeNode;
begin
 CaptionStr:='';
 if not InputQuery('Ввод имени', 'Введите заголовок элемента',CaptionStr) then exit;

 NewNode:=TreeView1.Items.Add(TreeView1.Selected, CaptionStr);
 if NewNode.Parent<>nil then
  NewNode.ImageIndex:=1;
end;
```

Здесь я объявил две переменные: CaptionStr типа строка String и NewNode типа TTreeNode. Тип TTreeNode – это тип отдельного элемента дерева элементов.

В первой строчке кода я обнуляю строку CaptionStr. Эта строка в будущем будет использоваться для хранения имени будущего элемента дерева.

Вторая строка имеет следующий код:

```
if not InputQuery('Ввод имени', 'Введите заголовок элемента', CaptionStr) then
 exit;
```

Здесь выполняется функция *InputQuery*, которая используется для вывода на экран окна ввода. У этой функции есть три параметра:

1. Заголовок окна ввода.
2. Текст-пояснение, которое подсказывает пользователю, что ему надо вводить.
3. Строковая переменная, в которой мы передаём значение по умолчанию и получаем результат ввода. Если перед вызовом записать в эту переменную какое-нибудь значение, то оно будет использоваться в качестве значения по умолчанию. Но после вызова этой функции этот параметр всегда хранит реально введённое пользователем значение.

На рисунке 11.19.2 ты можешь увидеть это окно ввода.

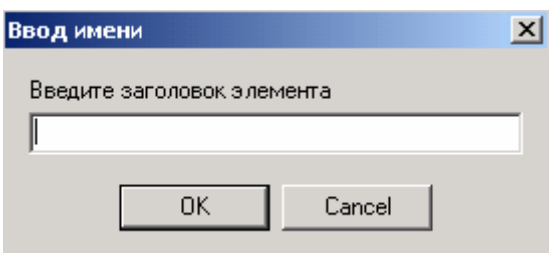


Рисунок 11.19.2. Окно ввода

Если окно было закрыто не кнопкой OK, то происходит выход из процедуры. Об этом говорит наша конструкция:

```
if not InputQuery(…) then
 exit;
```

Следующая строка кода добавляет новый элемент в наше дерево:

```
NewNode:=TreeView1.Items.Add(TreeView1.Selected, CaptionStr);
```

У компонента *TreeView1* есть свойство *Items* в котором хранятся все элементы дерева. Это свойство имеет объектный тип ***TTreeNodes***. Чтобы добавить туда новый

элемент, нужно вызвать метод *Add* объекта *Items*. Получается, что в объекте *TreeView1* есть ещё один объект *Items* в котором хранятся все элементы. Мы уже сталкивались с такими случаями, когда внутри одного объекта был другой объект.

У метода *Add* есть два параметра:

1. Элемент, к которому надо добавить новый. Здесь я передаю выделенный элемент (*TreeView1.Selected*).
2. Заголовок нового элемента.

Результат выполнения этого метода – указатель на новый элемент. Этот результат мы сохраняем в переменной *NewNode*. Теперь мы можем изменять и другие значения этого элемента. Например, как в следующем коде я буду изменять картинку:

```
if NewNode.Parent<>nil then
  NewNode.ImageIndex:=1;
```

Здесь идёт проверка, если свойство *Parent* нашего дерева не равно нулю (т.е. компонент не является верхним в дереве), то изменить значение *ImageIndex* созданного нами элемента на 1 (по умолчанию это значение 0).

По нажатию кнопки «Добавить элемент» пишем следующий код:

```
var
 CaptionStr:String;
 NewNode:TTreeNode;
begin
 CaptionStr:='';
 if not InputQuery('Ввод имени подэлемента',
    'Введите заголовок подэлемента',CaptionStr) then exit;

 NewNode:=TreeView1.Items.AddChild(TreeView1.Selected, CaptionStr);
 if NewNode.Parent<>nil then
  NewNode.ImageIndex:=1;
```

Здесь код практически тот же, что и для кнопки «Добавить». Единственная разница в том, что при добавлении нового элемента мы используем метод *AddChild*. Отличие этого метода от просто *Add* заключается в том, что он добавляет дочерний элемент. Например, если ты выделил в списке какой-то элемент и передал его в качестве первого параметра в *AddChild*, то новый элемент будет как бы подчиняться выделенному. При использовании метода *Add* новый элемент будет находиться на одном уровне дерева с переданным в качестве первого параметра.

Теперь напишем код для кнопки «Добавить»:

```
if TreeView1.Selected<>nil then
  TreeView1.Items.Delete(TreeView1.Selected);
```

Здесь нужно удалить выделенный элемент, поэтому сначала я проверяю, есть ли вообще выделенный элемент в дереве:

```
if TreeView1.Selected<>nil then
```

Если такой элемент есть, то выполниться следующий код:

TreeView1.Items.Delete(TreeView1.Selected);

Здесь мы используем метод *Delete* объекта *Items*, чтобы удалить элемент дерева. В качестве параметра надо передать элемент, который мы хотим удалить (я передаю выделенный *TreeView1.Selected*.

Для кнопки «Изменить заголовок» мы напишем следующий код:

```
procedure TTreeViewForm.EditButtonClick(Sender: TObject);
var
 CaptionStr:String;
begin
 CaptionStr:='';
 if not InputQuery('Ввод имени',
   'Введите заголовок элемента',CaptionStr) then exit;

 TreeView1.Selected.Text:=CaptionStr;
end;
```

Здесь снова я вызываю окно *InputQuery*, чтобы пользователь смог ввести новое имя для выделенного элемента. Теперь, чтобы изменить имя надо изменить свойство *Text* для выделенного элемента: *TreeView1.Selected.Text*.

Давай теперь сделаем возможность сохранения и загрузки данных в наше дерево. Для этого создай обработчик события *OnClose* и напишем в нём следующее:

```
procedure TTreeViewForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
 TreeView1.SaveToFile(ExtractFilePath(Application.ExeName)+'tree.dat');
end;
```

Для сохранения дерева нужно вызвать метод *SaveToFile* и в качестве единственного параметра указать имя файла. В качестве имени файла можно указывать что угодно, но я предпочитаю использовать полный путь, чтобы файл случайно не создался в каком-нибудь другом месте. Для этого я использую следующую конструкцию:

ExtractFilePath(Application.ExeName)+'tree.dat'

Application.ExeName – указывает на имя запущенного файла.

ExtractFilePath – вытаскивает путь к файлу из указанного в качестве параметра пути к файлу. Получается, что вызов *ExtractFilePath(Application.ExeName)* вернёт путь к директории, откуда была запущена прога. Остаётся только к этому пути прибавить имя файла (у меня это *'tree.dat'*) и можно быть уверенным, что файл обязательно будет находиться там же, где и запускной файл.

Теперь нужно загрузить сохранённые данные. Для этого по событию *OnShow* напишем следующее:

```
procedure TTreeViewForm.FormShow(Sender: TObject);
```

```
begin
 if FileExists(ExtractFilePath(Application.ExeName)+'tree.dat') then
  TreeView1.LoadFromFile(ExtractFilePath(Application.ExeName)+'tree.dat');
end;
```

Здесь я сначала проверяю с помощью вызова функции *FileExists* существование файла. Этой функции нужно передать полное имя файла и если такой файл существует, то функция вернёт **true** иначе **false**.

Если файл существует, то можно его загрузить с помощью вызова метода *LoadFromFile*.

Обязательно проверяй файл на существование. Если его нет или кто-то его удалил, а ты попытаешься загрузить данные из несуществующего файла, то произойдёт ошибка.

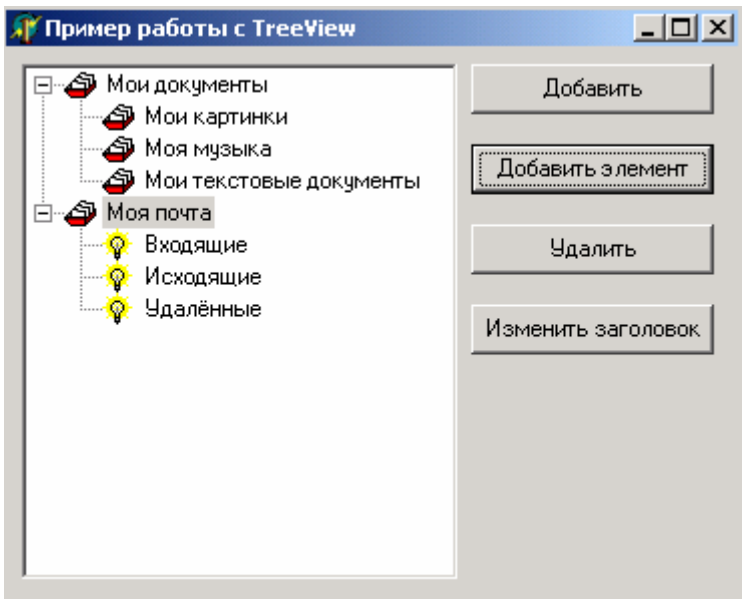


Рисунок 11.19.3. Результат работы программы

На рисунке 11.19.3 показано окно моего примера. Уже на рисунке ты можешь заметить, что дочерние элементы пункта «*Моя почта*» имеют другой рисунок, а дочерние элементы «Мои документы» нет. Почему так получилось? Ведь я же всегда меняю рисунок, если это дочерний элемент? Просто после закрытия программы дерево сохраняется в файле, а после загрузки оно загружается без учёта изображений. Состояние картинок не сохраняется и об этом нам нужно заботиться самостоятельно. Но это уже отдельная история, о которой мы поговорим в другой раз.

*На компакт диске, в директории **\Примеры\Глава 11\TreeView** ты можешь увидеть пример этой программы.*

11.20 Список элементов (TListView)

Следующий компонент тоже достаточно сильно распространён. Попробуй запустить «Проводник» Windows и оглянись. Слева есть дерево каталогов. Как работает такое дерево мы уже разобрались. А вот справа находиться список файлов в выделенной директории. Этот список как раз и храниться в компоненте ***TListView***, с которым мы сейчас и познакомимся.

 **- TListVew**

С работой этого компонента мы так же познакомимся на практике. Для этого мы напишем простейший файловый менеджер и заодно закрепим большинство уже пройденного материала. Но всё это в следующей части этой главы, а сейчас я покажу тебе основные свойства компонента ***TListVew***, чтобы нам легче было двигаться дальше.

У списка элементов очень много свойств отвечающих за обрамление (внешний вид рамки) компонента. Я не буду их все перечислять, потому что разных вариантов очень много, я только советую тебе сейчас остановиться, создать новый проект, бросить на форму один компонент *ListView* и попробуй поиграть со следующими свойствами:

1. *BevelEdges* – здесь ты указываешь, с какой стороны должна быть оборка. По умолчанию со всех сторон стоит true.
2. *BevelInner* – вид внутренней оборки.
3. *BevelOuter* – вид внешней оборки.
4. *BevelKind* – тип оборки.
5. *BorderStyle* – стиль обрамления (плоский или трёхмерный).

Теперь остальные интересные свойства:

1. *Checkboxes* – если здесь **true**, то каждый элемент списка содержит ещё и компонент CheckBox.
2. *ColumnClick* – должны ли заголовки колонок выглядеть как кнопки и принимать сообщения от кнопок мыши.
3. *Columns* – если здесь щёлкнуть дважды мышкой, то появиться маленький редактор колонок списка.
4. *FlatScrollBar* – должны ли полосы прокрутки выглядеть в стиле *Flat* (плавающий).
5. *FullDrag* – полное перетаскивание.
6. *GridLines* – должны ли быть видна сетка, когда компонент выглядит в стиле *vsReport*.
7. *HotTrack* – включение режима *Hot*, когда при наведении на элемент списка могут происходить какие-то действие.
8. *HotTrackStyles* – Это группа свойств в которой описываются действия происходящие при включённом режиме *HotTrack*.
   a. *htHandPoint* – если равно *true*, то при наведении на элемент курсором мыши, курсор принимает вид руки (как в IE при наведении на ссылку).
   b. *htUnderlineCold* – если равно *true*, то надо подчёркивать надписи на элементах даже когда не наведена вышка.
   c. *htUnderlineHot* – если равно *true*, то надо подчёркивать надписи на элементах только когда наведена вышка на элемент.
9. *IconOptions* – группа свойств отвечающих за иконки элементов.
   a. *Arrangement* – расположение иконки сверху или слева.
   b. *AutoArrange* – автоматическое выравнивание.
   c. *WrapText* – переносить надпись под иконкой, когда она не помещается в одну строку.
10. *Items* – это объект, который хранит все элементы списка. Он очень похож на тот, что мы рассматривали в прошлой части при написании примера к дереву элементов.
11. *LargeImage* – здесь указывается компонент TImageList, в котором должны храниться большие иконки для элементов (32x32).
12. *MultiSelect* – есть ли возможность выделять сразу несколько элементов.

13. *RowSelect* – должна ли выделяться вся строка, когда компонент выглядит в стиле *vsReport*.
14. *ShowColumnHeader* – надо ли показывать заголовки, когда компонент выглядит в стиле *vsReport*.
15. *SmallImage* – здесь указывается компонент TImageList, в котором должны храниться маленькие иконки для элементов (16x16).
16. *ViewStyle* - стиль отображения списка. Здесь возможны варианты:
    a. *vsIcon* – больше иконки.
    b. *vsSmallIcon* – маленькие иконки.
    c. *vsList* – список.
    d. *vsReport* – отчёт.

На этом пока остановимся и перейдём к рассмотрению компонента в действии.

11.21 Простейший файловый менеджер.

Здесь я тебе хочу рассказать, как самому написать простейший файловый менеджер. Этим примером мы закрепим наши знания по работе с файлами и научимся работать со списком элементов.

Создай новый проект в Delphi и брось на него следующие компоненты: одну кнопку, одну строку в вода и один список элементов. Всё это расположи примерно так же, как на рисунке 11.21.1.

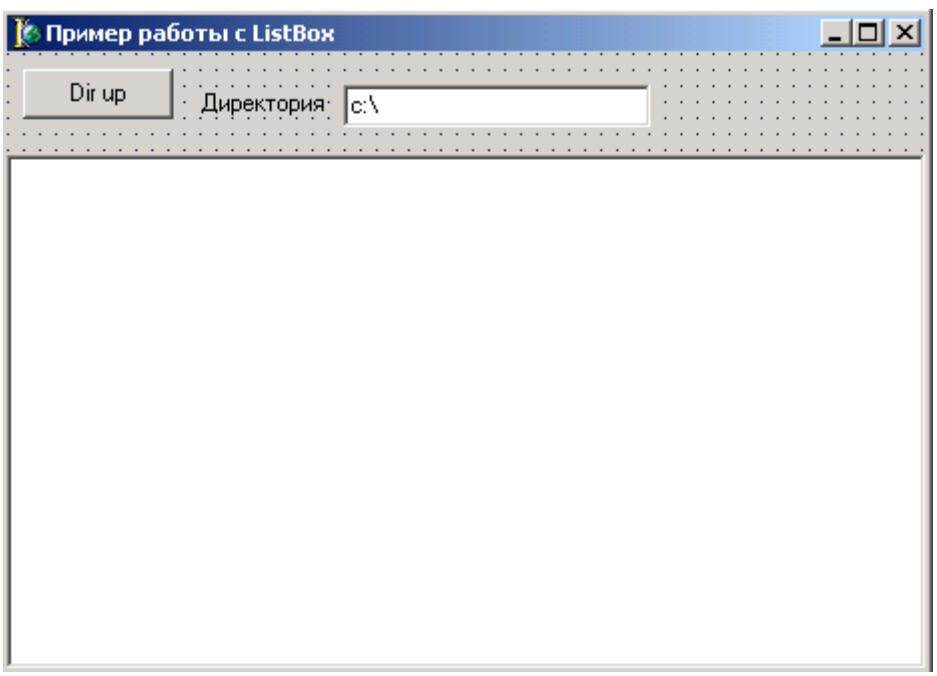


Рис 11.21.1

Для работы примера нужен модуль shellapi , поэтому давай сразу добавим его в раздел uses.

При рассмотрении примеров, обращай внимание на комментарии, они помогут тебе разобраться с происходящим в проге. А я буду расписывать только наиболее интересные моменты.

Эту программу мы начнём писать с процедуры FormCreate. Создай обработчик события OnCreate для формы и напиши в нём следующее:

```
procedure TForm1.FormCreate(Sender: TObject);
var
 SysImageList: uint;
 SFI: TSHFileInfo;
begin
//Создаю списки маленьких и больших иконок.
 ListView1.LargeImages:=TImageList.Create(self);
 ListView1.SmallImages:=TImageList.Create(self);

//Запрашиваю большие иконки
 SysImageList := SHGetFileInfo('', 0, SFI,
 SizeOf(TSHFileInfo), SHGFI_SYSICONINDEX or SHGFI_LARGEICON);
 if SysImageList <> 0 then
  begin
//Присваиваю системные иконки в ListView1
   ListView1.Largeimages.Handle := SysImageList;
   ListView1.Largeimages.ShareImages := TRUE;
  end;
//Запрашиваю маленькие иконки
 SysImageList := SHGetFileInfo('', 0, SFI, SizeOf(TSHFileInfo),
    SHGFI_SYSICONINDEX or SHGFI_SMALLICON);
 if SysImageList <> 0 then
  begin
   ListView1.Smallimages.Handle := SysImageList;
   ListView1.Smallimages.ShareImages := TRUE;
  end;
end;
```

В первых двух строчках я создаю списки маленьких и больших иконок. Свойства *LargeImages* и *SmallImages* имеют тип ***TImageList***, но сразу после создания компонента они равны **nil**. Поэтому я создаю их присваивая результат вызова TImageList.Create(self). После этого они уже проинициализированы и необходимая память выделена.

Тут можно было поступить немного другим способом – поставить на форму два компонента ***TImageList*** и просто указать их в соответствующих свойствах. В этом случае не пришлось бы ничего инициализировать, потому что компоненты стоящие на форме инициализируются автоматически. Но я решил не делать этого, а показать тебе, что объектное свойство можно сразу заставить работать без использования дополнительных компонентов.

После этого я запрашиваю у системы список больших иконок. Для этого я использую функцию *SHGetFileInfo*, которая возвращает информацию о файле, директории или диске. Первый параметр - путь к файлу. Второй - атрибуты. Третий - указатель на ***TSHFILEINFO***. Четвертый - размер ***TSHFILEINFO***. Последний - флаги, указывающие на тип информации запрашиваемый у системы.

Теперь о том, как выглядит функция в моём примере. Первые два параметра пустые, это означает, что нам нужны глобальные данные, а не информация о конкретном файле. Если указать здесь реальные значения файла, то мы получим информацию о нём, а если указать нулевые значения, то мы получим системную информацию.

В качестве флагов я указываю *SHGFI_SYSICONINDEX* и *SHGFI_LARGEICON*. *SHGFI_SYSICONINDEX* означает, что я запрашиваю указатель на системный список иконок (*ImageList*). Второй флаг говорит, что мне нужны большие иконки.

При втором вызове этой функции (чуть ниже в коде) я запрашиваю маленькие иконки (*SHGFI_SMALLICON*). Функция возвращает мне указатель на соответствующий системе *SysImageList*, который я в последствии присваиваю в *ListView1.LargeImages.Handle*. После этого присваивания *ListView1.LargeImages* содержит все системные иконки размера 32x32.

Системный список иконок (ImageList) содержит все иконки, установленные в системе и ассоциированные с разными типами файлов. Эти иконки ты можешь видеть в Windows Explorer у вайлов doc, txt, ini, zip и др.

Теперь создадим обработчик события *OnShow*. В ней я вызываю другую процедуру AddFile, которая считывает все файлы из текущей директории.

```
procedure TForm1.FormShow(Sender: TObject);
begin
 AddFile(Edit1.Text+'*.*',faAnyFile)
end;
```

Процедура AddFile объявлена в разделе **private** нашей формы **Form1** следующим образом:

```
private
  { Private declarations }
  function AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
```

Объяви эту процедуру так же и потом нажми клавиши Ctrl+Shift+C и *Delphi* создаст заготовку для будущей процедуры:

```
function TForm1.AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
begin
end;
```

В эту заготовку напиши следующее:

```
function TForm1.AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
var
 ShInfo: TSHFileInfo;
 attributes: string;
 FileName: string;
 hFindFile: THandle;
 SearchRec: TSearchRec;

 function AttrStr(Attr: integer): string;
  begin
   Result := '';
   if (FILE_ATTRIBUTE_DIRECTORY and Attr) > 0 then Result := Result + '';
   if (FILE_ATTRIBUTE_ARCHIVE and Attr) > 0 then Result := Result + 'A';
   if (FILE_ATTRIBUTE_READONLY and Attr) > 0 then Result := Result + 'R';
   if (FILE_ATTRIBUTE_HIDDEN and Attr) > 0 then Result := Result + 'H';
   if (FILE_ATTRIBUTE_SYSTEM and Attr) > 0 then Result := Result + 'S';
  end;
```

```
begin
 ListView1.Items.BeginUpdate;
 ListView1.Items.Clear;

 Result := False;
 hFindFile := FindFirst(FileMask, FFileAttr, SearchRec);
 if hFindFile <> INVALID_HANDLE_VALUE then
  try
   repeat
    with SearchRec.FindData do
     begin

      if (SearchRec.Name = '.') or (SearchRec.Name = '..') or
      (SearchRec.Name = '') then continue;

      FileName := SlashSep(Edit1.Text, SearchRec.Name);
      SHGetFileInfo(PChar(FileName), 0, ShInfo, SizeOf(ShInfo),
                  SHGFI_TYPENAME or SHGFI_SYSICONINDEX);
      Attributes := AttrStr(dwFileAttributes);
      //Добавляю новый элемент
      with ListView1.Items.Add do
       begin
        //Присваиваю его имя
        Caption := SearchRec.Name;
        //Присваиваю индекс из системного списка изображений
        ImageIndex := ShInfo.iIcon;
        //Присваиваю размер
        SubItems.Add(IntToStr(SearchRec.Size));
        SubItems.Add((ShInfo.szTypeName));
        SubItems.Add(FileTimeToDateTimeStr(ftLastWriteTime));
        SubItems.Add(attributes);
        SubItems.Add(Edit1.Text + cFileName);
        if (FILE_ATTRIBUTE_DIRECTORY and dwFileAttributes) > 0 then
         SubItems.Add('dir')
        else
         SubItems.Add('file');
       end;
      Result := True;
     end;
    until (FindNext(SearchRec) <> 0);
   finally
    FindClose(SearchRec);
   end;
 ListView1.Items.EndUpdate;
end;
```

Неплохая процедурка получилась и надо бы подробно её описать. Я буду делать это по кусочкам, чтобы было легче воспринимать:

```
function TForm1.AddFile(FileMask: string; FFileAttr:DWORD): Boolean;
var
 ...
 ...

function AttrStr(Attr: integer): string;
 begin
  Result := '';
  if (FILE_ATTRIBUTE_DIRECTORY and Attr) > 0 then Result := Result + '';
```

```
  if (FILE_ATTRIBUTE_ARCHIVE and Attr) > 0 then Result := Result + 'A';
  if (FILE_ATTRIBUTE_READONLY and Attr) > 0 then Result := Result + 'R';
  if (FILE_ATTRIBUTE_HIDDEN and Attr) > 0 then Result := Result + 'H';
  if (FILE_ATTRIBUTE_SYSTEM and Attr) > 0 then Result := Result + 'S';
end;
```

После имени процедуры идёт объявление локальных переменных. Это всё понятно и мы не раз уже такое делали. Но после объявления переменных, вместо начала процедуры (**begin**) у меня стоит объявление другой локальной процедуры - **function AttrStr(Attr: integer): string;**. Да, и такое в Delphi тоже возможно. Конечно же эту процедуру можно написать как полноценную, но я решил показать тебе, что такое локальная процедура.

Если одна процедура/функция (внутренняя) объявлена внутри другой (внешней), то внутренняя процедура может быть вызвана только из внешней. Весь остальной код программы не будет знать о существовании где-то внутренней процедуры.

Лично я такие вещи стараюсь не использовать, потому что пока не встречался с ситуацией, когда внутренняя процедура необходима. Я всегда прекрасно обхожусь и без неё. Но всё же рассказать тебе о ней необходимо, потому что ты можешь встретить такую конструкцию в других программах.

После объявления и описания внутренней процедуры идёт начало (**begin**) внешней процедуры. Вот тут уже начинается самое интересное. В самом начале я вызываю два метода компонента ListView1:

```
ListView1.Items.BeginUpdate;
ListView1.Items.Clear;
```

Первый метод *BeginUpdate* говорит о том, что начинается обновление элементов списка. После этого вызова никакие изменения вносимые в элементы не будут отражаться на экране, пока не будет вызван *EndUpdate*.

Когда ты хочешь произвести незначительное изменение, то не надо вызывать эти методы, но когда ты чувствуешь, что здесь элементы списка будут изменяться очень сильно, то лучше все изменения заключить между вызовами *BeginUpdate* и *EndUpdate*. Это связано с тем, что когда ты вносишь хоть какое-то изменение, оно сразу отображается на экране. Логично? Я тоже так думаю. А что если тебе нужно удалить все элементы и потом в цикле добавить в список 1000 новых элементов. В этом случае после удаления и каждого добавления нового элемента будет происходить прорисовка компонента. Вот тут и возникает вопрос: «Зачем после каждого добавления рисовать?». В этом случае намного эффективнее будет добавить все элементы, а только потом их прорисовать все сразу. Вот именно для этого и существуют своеобразные скобки *BeginUpdate* и *EndUpdate*:

```
ListView1.Items.BeginUpdate; // Запрещаем прорисовку

// Делаем необходимые изменения

ListView1.Items.EndUpdate; // Прорисовываем все изменения сразу
```

С этим разобрались, можно ехать дальше. После вызова *BeginUpdate* я очищаю текущий список элементов с помощью вызова *ListView1.Items.Clear*.

Далее идёт цикл поиска файлов, с которым мы уже немного познакомились в 10-й главе моей книги. Здесь я только напомню тебе этот процесс:

FindFirst - открывает поиск. В качестве первого параметра выступает маска поиска. Если ты укажешь конкретный файл, то система найдёт его. Но это не серьёзно, лучше искать более серьёзные вещи. Например, ты можешь запустить поиск всех файлов в корне диска *C*. Для этого первый параметр должен быть *'C:*.*'*. Для поиска только файлов *EXE*, в папке *Fold* ты должен указать *'C:\Fold*.exe'*.

Второй параметр - атрибуты включаемых в поиск файлов. Я использую *faAnyFile*, чтобы искать любые файлы. Тебе доступны

faReadOnly - искать файлы с атрибутом *ReadOnly*.
faHidden - искать скрытые файлы.
faSysFile - искать системные файлы.
faArchive - искать архивные файлы.
faDirectory - искать директории.

Последний параметр - это структура в которой нам вертеться информация о поиске, а именно имя найденного файла, размер, время создания и т.д. После вызова этой процедуры, я проверяю на корректность найденного файла. Если всё в норме, то запускается цикл **Repeat - Until**. Этот цикл выполняет операторы расположенные между **repeat** и **until**, пока условие расположенное после слова *until* является верным. Как только условие нарушается, цикл прерывается. Этот цикл похож на while, но с одним отличием. Если в цикле **while** условие заведомо не верно, то операторы внутри цикла не выполняться. А в **Repeat-Until** выполнятся, потому что сначала происходит выполнение операторов, а лишь затем проверка **Until**. Рассмотрим пример:

```
index:=1;
while index=0 do
 Param:=0;
```

В этом примере оператор Param:=0; не будет выполнен, потому что index=1 и условие заведомо не верно.

```
index:=1;
repeat
 Param:=0;
until index=0;
```

В этом примере Param:=0 выполнится, Потому что сначала выполняется этот оператор, а лишь потом проверка на равенство index нулю.

Хочу предупредить, что функция поиска, может возвращать в качестве найденного имени в структуре SearchRec (параметр Name) точку или две точки. Если ты посмотришь на директорию, то таких файлов не будет. Откуда берутся эти имена? Имя файла в виде точки указывает на текущую директорию, а имя файла из двух точек указывает на директорию верхнего уровня. Если я встречаю такие имена, то я их просто отбрасываю:

```
//Отбрасывание имён с точкой и двумя точками
if (SearchRec.Name = '.') or (SearchRec.Name = '..') or
```

```
 (SearchRec.Name = '') then continue;
```

Далее идёт вызов функции SlashSep:

```
FileName := SlashSep(Edit1.Text, SearchRec.Name);
```

Эта функция и FileTimeToDateTimeStr написаны мной и объявлены в разделе **var** после объявления объекта:

```
var
  Form1: TForm1;
  function SlashSep(Path, FName: string): string;
  function FileTimeToDateTimeStr(FileTime: TFileTime): string;

implementation
```

Я специально объявил их там, чтобы показать тебе, как можно пользоваться функциями не принадлежащими ни одному объекту. Здесь функция *SlashSep* объявлена не внутри объекта, значит она никому не принадлежит.

Вообще-то самостоятельные функции не обязательно где-либо объявлять. Ты можешь без проблем просто реализовать её и нигде не описывать. Но ты должен учитывать, что если ты где-то хочешь использовать эту функцию, то реализация обязательно должна быть раньше. Вот пример правильного использования самостоятельной процедуры/функции:

```
procedure Examp;
begin
end;

procedure Form1.Examp2;
begin
 Examp;
end;
```

В этом примере я создал самостоятельную процедуру Examp и метод объекта Form1 – Examp2. Из метода Examp2 я вызываю самостоятельную процедуру Examp. Этот код правильный, потому что процедура сначала реализовывается, а потом уже используется.

А теперь посмотри на неправильный код:

```
procedure Form1.Examp2;
begin
 Examp;
end;

procedure Examp;
begin
end;
```

В этом примере я пытаюсь вызвать процедуру, которая реализована после вызова и поэтому компилятор выдаст ошибку. Чтобы этого избежать, самостоятельные процедуры можно описывать в разделе var:

```
var
 procedure Examp;

procedure Form1.Examp2;
begin
 Examp;
end;

procedure Examp;
begin
end;
```

А теперь давай посмотрим, как же выглядит функция *SlashSep*:

```
function SlashSep(Path, FName: string): string;
begin
 if Path[Length(Path)] <> '\' then
  Result := Path + '\' + FName
 else Result := Path + FName;
end;
```

Как видишь, здесь я пишу просто «function SlashSep …..» не добавляя имени объекта перед именем функции, потому что эта функция самостоятельная.

Эта функция получает два параметра – путь к файлу и имя файла, которые она должна соединить в одну строку, чтобы получился полный путь к файлу. Но сначала мы должны проверить, заканчивается ли путь (первый полученный параметр – Path) знаком ‘\’. Именно это делается в первой строчке кода:

Path[Length(Path)] <> '\'

Переменная *Path* – это строка типа *String*, а значит мы можем к ней обращаться как к массиву символов. Это значит что чтобы получить доступ к первому символу мы должны написать *Path[1]*. Нам нужно проверить последний символ, поэтому в квадратных скобках я пишу *Length(Path)*. Функция *Length* возвращает длину переданной ей строковой переменной, а это значит, что в квадратных скобках мы указываем длину строки, а это последний символ.

Если бы нам нужен был предпоследний символ, то мы бы написали *Path[Length(Path)-1]*. В этом случае я из длины строки вычитаю единицу и получаю предпоследний символ.

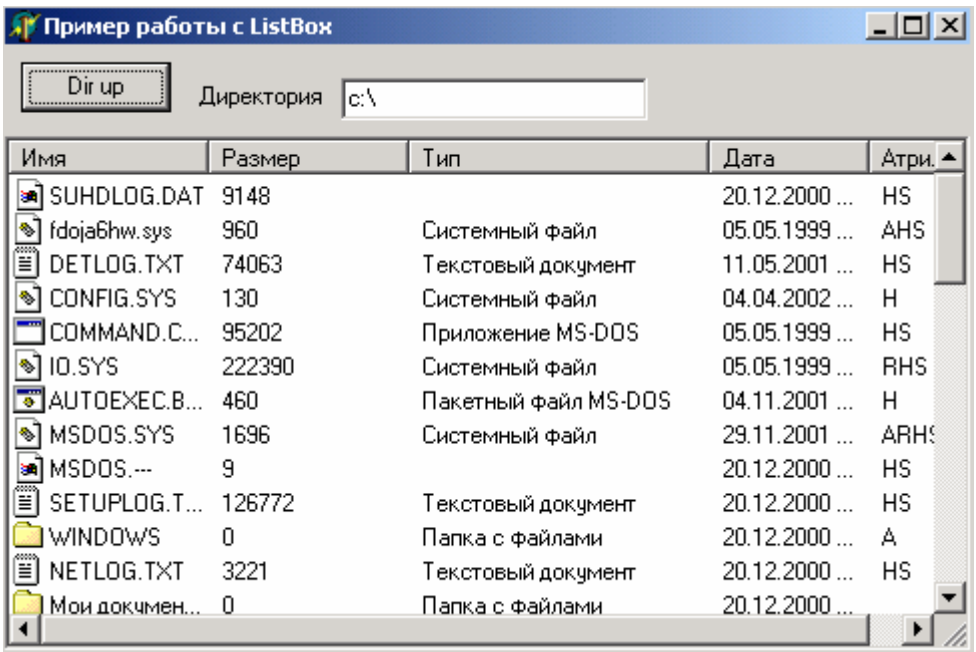
Если последний символ не равен ‘\’, то я добавляю сначала его, а потом имя файла. Если равен, то нужно только добавить имя файла и записать в переменную Result, чтобы функция вернула полный путь к файлу.

С этой функцией покончено и пора вернуться к нашему перечислению файлов. Следующим идёт вызов системной функции SHGetFileInfo. Она возвращает информацию

о файле. Я не буду на ней сейчас останавливаться. В принципе она простая и ты наверно сможешь с ней разобраться, а если нет, то я немного позже вернусь к этому.

Сейчас нас больше интересует работа с компонентом ListView. Следующий код добавляет в список новый элемент: *ListView1.Items.Add*. Я это делаю внутри конструкции with, значит все последующие действия между begin и end будут выполняться с новым элементом. А именно, я изменяю заголовок нового элемента (*Caption := SearchRec.Name*) и картинку (*ImageIndex := ShInfo.iIcon*).

У каждого элемента есть свойство SubItems, которое хранит дополнительную информацию. Когда компонент находиться в режиме отображения иконок, то дополнительная информация не видна. Но если выбрать в свойстве ViewStyle значение vsReport, то компонент будет выглядеть в виде таблицы, где каждый столбец отображает дополнительную информацию:



Чтобы добавить дополнительные колонки к новому элементу надо выполнить SubItems.Add('значение');.

Чтобы колонки отображались, нужно ещё в свойстве Columns указать имена колонок. Если имена колонок не указаны, то ничего отображаться не будет. И не забывай, что при описании колонок первая – это заголовок элементов, а остальные это дополнительные параметры в порядке их добавления с помощью SubItems.Add.

11.22 Улучшенный файловый менеджер (С возможностью запуска файлов)

Давай добавим к нашему файловому менеджеру возможность путешествия по директориям и запуска файлов. Для этого нужно создать обработчик события *OnDblClick* для компонента *ListView* и написать в нём следующее:

```
procedure TForm1.ListView1DblClick(Sender: TObject);
```

```
begin
//Это директория?
 if (ListView1.Selected.SubItems[5] = 'dir') then
  begin
//Если да, то прибавить имя выделенной директории к пути
//и перечитать файлы из неё.
   Edit1.Text:=Edit1.Text+ListView1.Selected.Caption+'\';
   AddFile(Edit1.Text+'*.*',faAnyFile)
  end
 else
//Если нет, то это файл и я его запускаю.
  ShellExecute(Application.MainForm.Handle, nil,
     PChar(Edit1.Text+ListView1.Selected.Caption), '',
     PChar(Edit1.Text), SW_SHOW);
end;
```

В этом коде я в самом начале проверяю по чём мы щёлкнули. Если это директория, то надо перейти в неё, а если файл, то надо его запустить. Для этого я проверяю 5-й дополнительный параметр выделенного элемента: *ListView1.Selected.SubItems[5]='dir'*. Когда я добавлял элементы и дополнительные параметры в *ListView*, то в качестве 5-го указывал для директорий значение 'dir', а для файлов 'file'. Теперь мне надо только проверить этот параметр.

Если выделенная строка – это директория, то я изменяю значение текущей директории в *Edit1.Text* и перечитываю её с помощью вызова AddFile, указав новое значение директории.

Если выделенная строка – это файл, то его надо запустить. Я люблю это делать с помощью вызова функции ShellExecute. У этой функции следующие параметры:

1. Программа, отвечающая за запуск приложения. Тут можно указать nil, но я указал главное окно моей программы (*Application.MainForm.Handle*).
2. Строка, указывающая на операцию, которую надо выполнить. Укажем nil для запуска файла.
3. Строка содержащая полный путь к файлу.
4. Строка параметров передаваемых программе в командной строке.
5. Директория по умолчанию.
6. Команда показа. Здесь я указал SW_SHOW для нормального отображения окна. Можно указать и другие параметры (все ты найдёшь в файле помощи), но чаще всего используются SW_SHOW (нормальный режим), SW_SHOWMAXIMIZED (показать максимизировано) или SW_SHOWMINIMIZED (показать в свёрнутом состоянии).

Функция ShellExecute объявлена в модуле Shellapi, поэтому его необходимо добавить в раздел uses, иначе Delphi не сможет откомпилировать проект.

На компакт диске, в директории ***\Примеры\Глава 11\ListView*** *ты можешь увидеть пример этой программы.*

11.23 Подсказки для чайников (TStatusBar)

Если ОС unix создавалась для профессионалов, то Windows создавалась для пользователей, чтобы им легче было работать. Потом она превратилась в ОС для чайников, ну а сейчас Windows превратили в ОС для полных кретинов, которые с компьютером полностью несовместимы. Так что теперь для успеха любой программы нужно обязательно делать большое количество подсказок, потому что кретины не умеют читать мануалы и файлы помощи. Сейчас уже надо чтобы любой мог сесть за компьютер и сразу начинал работать.

Самым первым способом облегчения жизни бедным юзерам стали строки состояния. Они и сейчас широко используются, потому что просты в использовании и удобны в обращении. Именно с этим компонентом мы сейчас и познакомимся.

 - **TStatusBar**

Поставить компонент на форму, это ещё не значит, что подсказки сразу же сами появятся на панели. Для полноценной работы надо выполнить следующее:

1. У компонента, при наведении на который должна отображаться подсказка, в свойстве *Hint* должен быть внесён текст подсказки.
2. Если ты хочешь, чтобы подсказка выскакивала не только в строке состояния, но и над компонентом, то у него или у родительского окна в свойстве *ShowHint* нужно установить *true*.
3. Мы должны создать обработчик события на подсказки.

Может это звучит сложно, но реально всё просто. Создай новое приложение и брось на него кнопку. Теперь в свойстве *Hint* напиши «*Это кнопка выхода*».

Попробуй запустить приложение и навести на кнопку. Никаких сообщений и подсказок пока не должно быть. Закрой программу и переходи опять в Delphi. Теперь попробуй установить в свойстве *ShowHint* у компонента или у главной формы значение *true*. Если ты установишь только у компонента, то подсказка будет выскакивать только у него. Если у формы, то подсказка будет появляться у всех компонентов на форме, у которых есть текст в свойстве *Hint* и *ParentShowHint* равно *true*.

Можешь запустить приложение и проверить появление подсказки.

Теперь мы добавим к нашему приложению возможность отображения такого же текста в строке состояния. Брось на форму компонент ***TStatusBar***. Теперь перейди в редактор кода и найди раздел **private**. В нём добавь объявление процедуры *ShowHint*:

```
private
  { Private declarations }
  procedure ShowHint(Sender: TObject);
```

Имя процедуры может быть и другим (например *MyShowHint*) но параметр должен быть именно такой.

Теперь нажми Ctrl+Shift+C чтобы Delphi создал заготовку для процедуры:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
end;
```

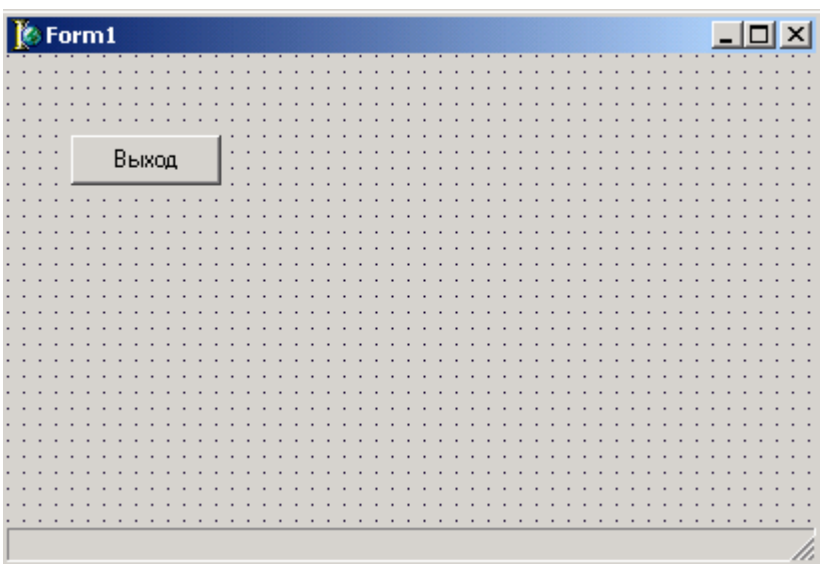
Можешь и сам написать этот текст, но только после текста *{ TForm1 }* или ещё дальше после какой-нибудь процедуры:

```
implementation

{$R *.dfm}

{ TForm1 }

procedure TForm1.ShowHint(Sender: TObject);
begin
end;
```

Внутри процедуры напиши следующее:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
 StatusBar1.SimpleText := Application.Hint;
end;
```

Итак, наша процедура должна будет вызываться каждый раз, когда надо вывести подсказку. Внутри процедуры мы присваиваем в свойство *SimpleText* строки состояния текст находящийся в *Application.Hint*. А в *Application.Hint* всегда находиться подсказка, которую надо сейчас отобразить.



Теперь создай обработчик события *OnShow* для главной формы и в нём напиши:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 Application.OnHint := ShowHint;
end;
```

Здесь мы программно назначаем нашу процедуру *ShowHint* в качестве обработчика события OnHint. Я люблю это делать программно, но можно было поступить и проще:

1. Поставить на форму компонент *TApplicationEvents* с закладки *Additional*.
2. У этого компонента на закладке *Events* создать обработчик события *OnHint* и там сразу же написать «*StatusBar1.SimpleText := Application.Hint*».

*На компакт диске, в директории **\Примеры\Глава 11\Hint** ты можешь увидеть пример этой программы.*

Теперь попробуем создать строку состояния из нескольких панелей. Выдели строку состояния и дважды щёлкни по свойству Panels. Перед тобой должно открыться окно редактора панелей:

В этом окне первая кнопка создаёт новую панель (также можно нажать клавишу Ins), а вторая удаляет выделенную (также можно нажать Del).

Создай новую панель и в её свойстве *Width* (ширина) установи значение 200. Теперь создай ещё одну панель. Всё, можно закрывать окно.

Теперь перейди в процедуру обработчик события OnHint и измени её текст на:

```
procedure TForm1.ShowHint(Sender: TObject);
begin
 StatusBar1.Panels[1].Text := Application.Hint;
end;
```

Здесь я присваиваю текст сообщения (*Application.Hint*) в свойство *Text* первой панели строки состояния.

*На компакт диске, в директории **\Примеры\Глава 11\HintPanels** ты можешь увидеть пример этой программы.*

11.24 Панель инструментов (TToolBar и TControlBar).

Панель инструментов уже давно въелась в нашу жизнь и уже трудно представить себе какой-нибудь хоть более менее значащий проект без неё. Некоторые считают, что меню достаточно, а некоторые наоборот обходятся одной панелью инструментов. Я же считаю, что любое оконное приложение должно иметь и то и другое.

Панель инструментов чаще всего располагается сразу же под меню, но это не обязательно. Иногда удобно расположить его вдоль какой-нибудь другой стороны окна

(левой, правой или нижней). В своей книге я буду располагать его в основном сверху (классический вариант), как это делается в большинстве программ, например MS Word.

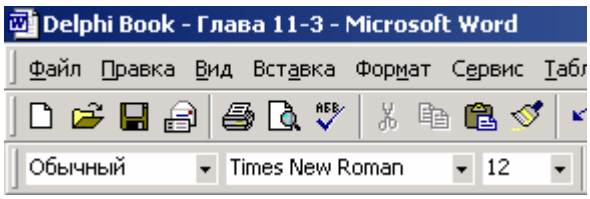


Рис 11.24.1 Панель инструментов MS Word. В ней ты видишь не только кнопки быстрого доступа к командам, но и выпадающие списки ComboBox.

Давай создадим маленькое приложение использующее панель инструментов. Брось на форму компонент *ControlBar*  с закладки **Additional** и измени его свойство *Align* на *alTop*, чтобы растянуть компонент вдоль верхней кромки окна. Сразу же желательно изменить и свойство *AutoSize* на *true*.

Компонент *ControlBar* я не рассматривал, потому что в нём нет ничего особенного, но он хорош тем, что на него удобно располагать панели инструментов. Они автоматически становятся перемещаемыми внутри *ControlBar*. Это значит, что панели можно будет двигать по своему усмотрению. Ну а если свойство *AutoSize* равно *true*, то компонент будет автоматически растягиваться и сужаться, когда ты будешь выстраивать все панели в одну строку или в столбик.

Давай теперь бросим внутрь компонента *ControlBar* одну панель *ToolBar*  с закладки **Win32**. Сразу же изменим одно его свойство – дважды щёлкни по свойству *EdgeBorders* и измени свойство *ebTop* на *false* (рисунок 11.24.2). Это заставит исчезнуть оборочку сверху панели.

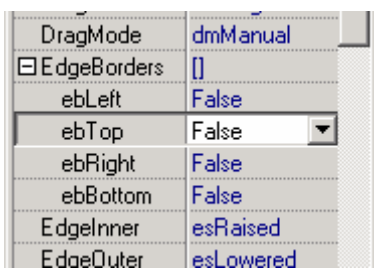


Рис 11.24.2 Убираем ненужную оборочку.

Ещё желательно сразу же изменить и здесь свойство *AutoSize* на *true*, чтобы панель принимала размеры соответствующие кнопкам.

Теперь создадим кнопки на панели. Для этого щёлкни по ней правой кнопкой крысы и выбери из появившегося меню пункт *New Button* (рисунок 11.24.3). Пункт *New Separator* этого же меню создаёт разделитель между кнопками. Если тебе нужно будет удалить кнопку или разделитель, то просто выделаешь его и нажимаешь кнопку Del (на клавиатуре :)).

Таким образом создай две кнопки, потому разделитель и ещё одну кнопку. У тебя должно получиться нечто похожее на рисунок (рисунок 11.24.3).

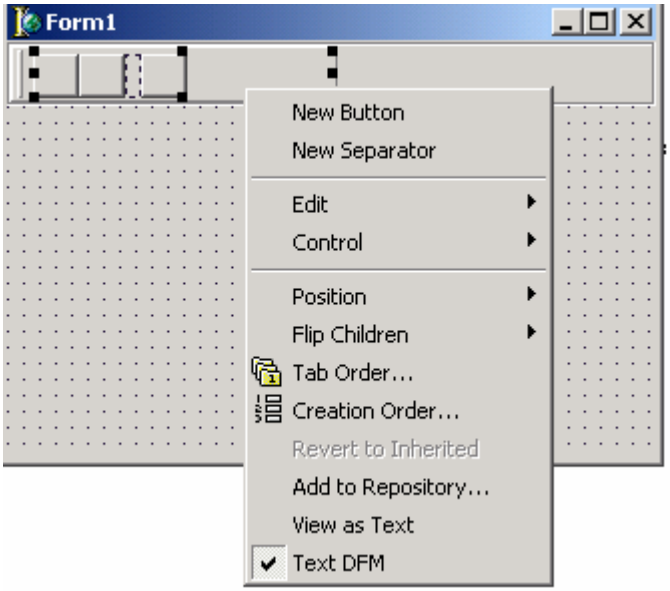


Рис 11.24.3 Создание новой кнопки.

В принципе, простая панель уже создана, но она простая. Теперь необходимо сделать так, чтобы кнопки что-то отображали. Но для начала я советую выделить саму панель и изменить свойство *Flat* на *true*, чтобы кнопки на панели выглядели более изящно (плоско).

Теперь бросим на форму компонент ***TImageList*** и добавим в него три картинки. Их изображение пока не имеет особого значения, поэтому можешь выбирать любые, главное, чтобы размер был 16x16.

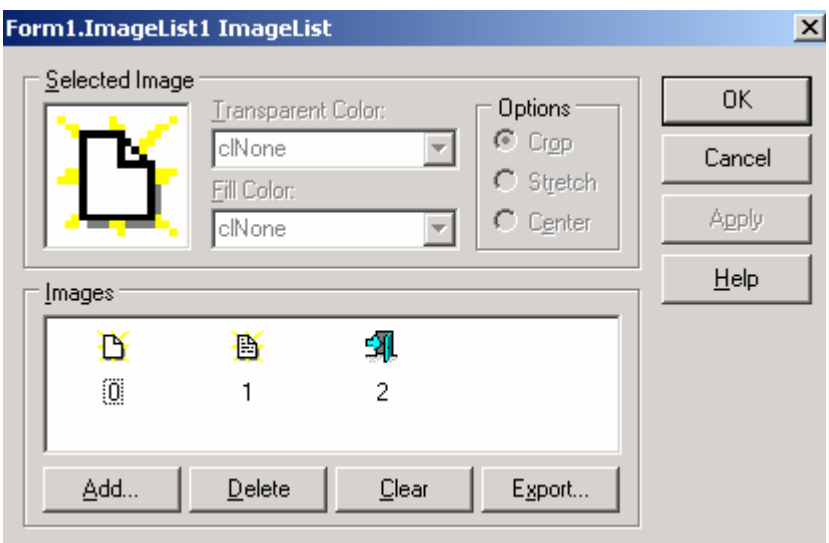


Рис 11.24.4 Набор изображений для панели инструментов

Теперь выдели панель и в свойстве Images укажи созданный набор картинок. На кнопках сразу же отобразятся картинки в той последовательности, в которой ты их добавил. Если ты хочешь изменить картинку на какой-нибудь кнопке, то надо выделить её и изменить свойство *ImageIndex*.

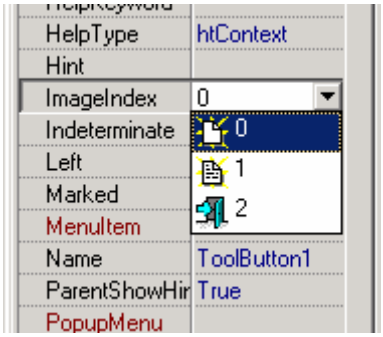


Рис 11.24.5 Изменение картинки для кнопки.

Теперь для каждой кнопки в свойстве *Caption* напиши осмысленный текст (по умолчанию там стоит текст *ToolButton* плюс порядковый номер кнопки). Желательно, чтобы текст соответствовал изображению на картинке. У нас хоть и пример, но всё же он должен быть приближён к боевым условиям.

Давай сделаем так, чтобы панель отображала на кнопках не только картинки, но и указанный в свойствах Caption текст. Для этого установи *true* в свойстве *ShowCaptions* у панели инструментов. Результат ты можешь увидеть на рисунке 11.24.6.

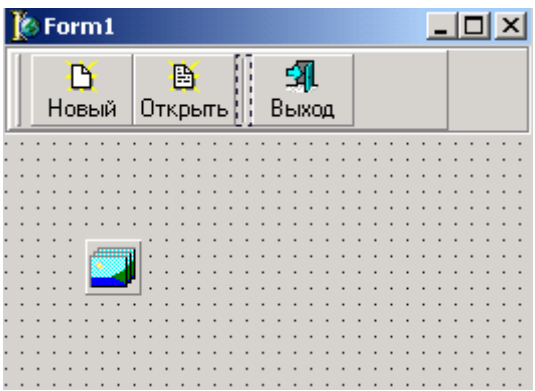


Рис 11.24.6 Панель отображающая картинки и текст.

Как видишь, в данном случае текст отображается под изображением. Если ещё установить свойство List у панели инструментов, то текст будет отображаться справа от картинки (рисунок 11.24.7).

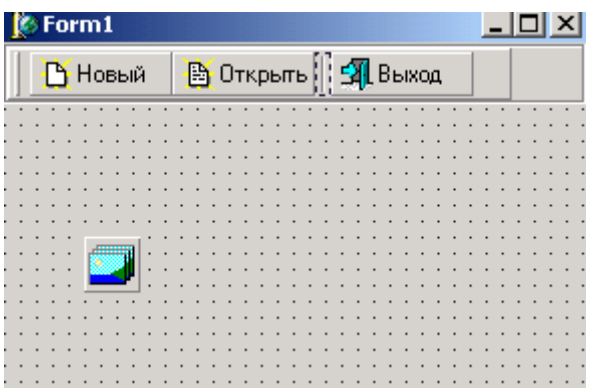


Рис 11.24.7 Панель отображающая текст слева от картинки.

Ну и для закрепления материала, давай создадим обработчик события для какой-нибудь кнопки. У меня последняя кнопка – *Выход*, вот для неё я и создам обработчик. Для этого я щёлкнул по ней дважды и в созданной процедуре обработчике написал ***Close;***.

*На компакт диске, в директории **\Примеры\Глава 11\ToolBar** ты можешь увидеть пример этой программы.*

11.25 Перемещаемые панели и меню в стиле MS (Docking).

Почему-то меня очень часто просят рассказать, как можно добиться такого эффекта как у ToolBar-ов в M$ Office. Для большей ясности - это когда палитру с кнопками можно оторвать от окна и прилепить в другое место или вообще превратить в отдельное окно.

Для того, чтобы TToolBar можно было перемещать, достаточно установить в нём свойство *DragKind* в *dkDock*. Вот и всё. Но главная проблема не в этом. Самое сложное здесь - это сохранить положение TToolBar после выхода из проги и восстановить его при запуске. Для примера я написал маленькую прогу, которую ты должен доделать до полноценной.

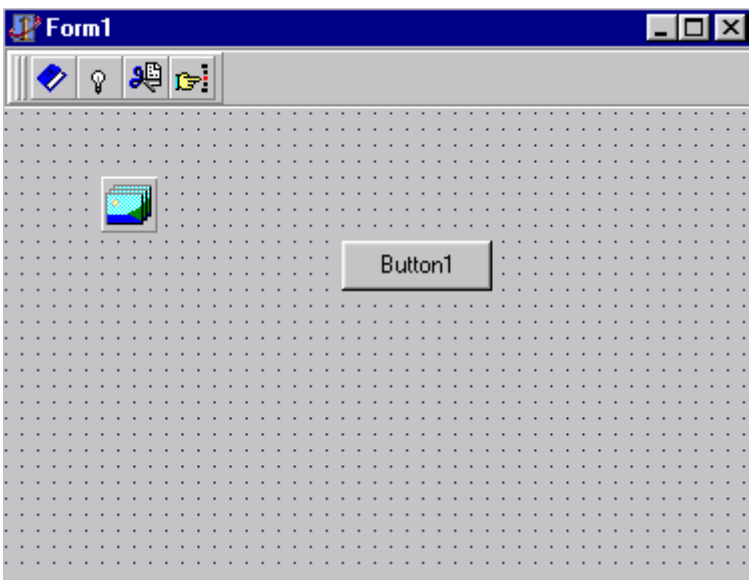


Рис 11.25.1 Форма примера

Для демонстрации мне понадобилась кнопка, по нажатию которой будет выводится положение TToolBar. По нажатию кнопки пишем следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 r:TRect;
begin
if ToolBar1.HostDockSite<>ControlBar1 then
  begin
   GetWindowRect(ToolBar1.Handle, R);
   Application.MessageBox(PChar(IntToStr(r.Left)+'--'+IntToStr(r.Top)),
                    'MM',IDOK);
  end;
end;
```

В первой строке я проверяю, лежит ли ToolBar1 на ControlBar1 с помощью (ToolBar1.HostDockSite<>ControlBar1). Если он лежит, то получить положение ToolBar1 очень просто. Для этого можно узнать всего лишь ToolBar1.Left и ToolBar1.Top.

Если ToolBar1 не лежит на ControlBar1 (ToolBar1 выглядит как отдельное окно), то задача усложняется. Тебе придётся вызывать GetWindowRect, чтобы получить реальное положение ToolBar1 на экране. В качестве первого параметра ты должен передать указатель на ToolBar1, а второй - это переменная типа TRect в которую запишется реальное положение окна. Для удобства я вывожу эти знчения в окне сообщения Application.MessageBox.

Всё это я делаю для наглядности. Теперь ты можешь запустить прогу и переместить ToolBar1 по экрану. Каждый раз, когда ты будешь нажимать кнопку, программа будет выводить окно и показывать тебе реальное положение ToolBar1.

По событию OnShow я написал:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 ToolBar1.ManualDock(nil,nil,alNone);
 ToolBar1.ManualFloat(Bounds(100, 500, ToolBar1.UndockWidth,
                ToolBar1.UndockHeight));
end;
```

ToolBar1.ManualDock заставляет переместится ToolBar1 на новый компонент. В качестве первого параметра указывается указатель на компонент или окно, к которому мы хотим прилепить ToolBar1. Я хочу, чтобы после загрузки ToolBar1 превратился в отдельное окно, поэтому я указываю nil. Второй параметр можешь ставить nil. Он означает компонент внутри компонента указанного в качестве первого параметра, на который мы хотим поместить ToolBar1. Я указал nil. Третий параметр – выравнивание.

С помощью ToolBar1.ManualFloat я просто двигаю ToolBar1 внутри нового компонента. У меня новый компонент nil, т.е. окно, поэтому я двигаю ToolBar1 по окну. Может не совсем понятно? Попробуй запустить пример и поиграть с ним, тогда всё встанет на свои места.

И ещё ToolBar1.UndockWidth и ToolBar1.UndockHeight возвращают размер ToolBar1, когда он выглядит как окно, а не лежит на ControlBar1.

Когда ты будешь использовать это в своей проге для сохранения положения ToolBar1, тебе надо будет написать примерно следующее по событию OnClose:

```
var
 r:TRect;
begin
if ToolBar1.HostDockSite<>ControlBar1 then
  begin
   GetWindowRect(ToolBar1.Handle, R);
   Здесь надо сохранить в реестре R.Left и R.Top.
   А также признак, что ToolBar1 не лежит на ControlBar1
  end
 else
  begin
   Здесь надо сохранить в реестре ToolBar1.Left и ToolBar1.Top.
   А также признак, что ToolBar1 лежит на ControlBar1
  end;
end;
```

На запуск программы ты должен написать примерно следующее:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 Прочитать положение ToolBar1. ControlBar1 mo
  Begin
   ToolBar1.Left:=Сохранённая левая позиция
   ToolBar1.Topt:=Сохранённая верхняя позиция
  End;
Иначе
 begin
  ToolBar1.ManualDock(nil,nil,alNone);
  ToolBar1.ManualFloat(Bounds(Сохранённая левая позиция,
     Сохранённая правая позиция, ToolBar1.UndockWidth,
        ToolBar1.UndockHeight));
 End;
end;
```

Как видишь, подводные булыжники есть. Но всё же ничего сильно сложного нет.

Теперь мы сделаем менюшку в стиле M$. Для этого нужно поставить ещё один ToolBar и установим его свойство *ShowCaption* в true. Создадим на нём две кнопки и назовём их *File* и *Edit* . Теперь установим компонент MainMenu и сделаем его таким как на рисунке 11.25.2. Меню *Not visible* сделаем невидимым (Visible=false), в этом случае всё меню будет подключено к форме но будет не видно. Для чего я это делаю, ведь можно было использовать PopupMenu? А потому что при использовании PopupMenu приходится мучится с клавишами быстрого вызова, а в моём способе они подключаются автоматически вместе с главным меню.

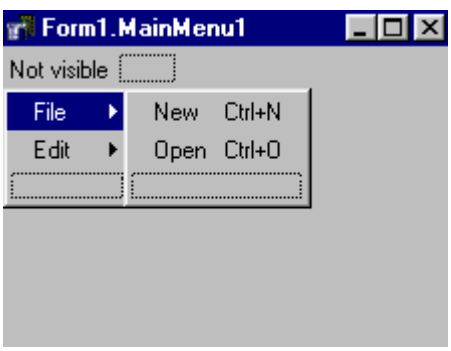


Рис 11.25.2 Меню

Чтобы создать подменю для меню File, нужно щёлкнуть по нём правой кнопкой и выбрать Create Submenu или нажать CTRL+Стрелка в право

Теперь кнопке File в свойстве MenuItem ставим File1 (имя пункта меню), а кнопке Edit ставим Edit1. И на последок обеим кнопкам нужно установить свойство Grouped в true.

Напоследок у каждой кнопки панели инструментов надо установить в свойстве *Grouped – true*.

Но это не единственный способ создания меню (это тот, что я чаще использую). Можно ещё просто создать полноценное главное меню. Потом просто убрать его из свойства главной формы *Menu*, а указать у созданной тобой пустой панели (без всяких кнопок) в таком же свойстве *Menu*.

 *На компакт диске, в директории* ***\Примеры\Глава 11\Dock*** *ты можешь увидеть пример этой программы.*

Глава 12. Графические возможности Delphi.

Вот мы уже постепенно добрались и до 12-й главы моей книги. Здесь я расскажу тебе про то, как Delphi помогает упростить работу с графикой Windows. Эту тему можно было раскрыть даже немного раньше, потому что мы уже немного познакомились с основами и немного рисовали. В этой же главе я постараюсь всё расписать как можно подробнее.

Windows – это графическая оболочка и всё, что ты в ней видишь – это графика. Но для программиста большинство вещей очень сильно упрощены, особенно в Delphi, поэтому мы пока ещё не сильно сталкивались с графическими средствами. Но если ты соберёшься писать какой-нибудь большой проект, то обязательно столкнешься с проблемой рисования при оформлении определённых частей программы.

Хотя я очень мало читал книг на русском языке по Delphi (моя знания идут из англоязычных источников, мануалов и исходников), но в мои руки попадало несколько экземпляров на русском языке. Во всех из них тема графики обсуждается в середине или ближе к концу (в любом случае после описания баз данных). И везде эта тема считалась сложным материалом. С одной стороны материал действительно сложнее, чем базы данных, но не на столько, чтобы пугать читателя. В моей же книге графика будет обсуждаться именно здесь, потому что дальнейшее изучение кодинга (в том числе и баз данных) невозможно без понимания графических возможностей Delphi/Windows.

Так как Windows – это графическая оболочка, то дальнейшее обучение программированию невозможно без прочтения этой главы, так что не пропусти и прочти её полностью, даже если ты думаешь, что графика тебе не нужна.

12.1 Графическая система Windows

В предыдущих главах мы уже встречались с графикой Delphi и немного рисовали. Когда мы делали это, то обращались к объекту *Canvas*. Практически у всех компонентов Delphi есть это объектное свойство. Почему объектное? Да потому что *Canvas* имеет тип объекта ***TCanvas***. То есть в нашем компоненте за рисование отвечает объект ***TCanvas***. Так что если компонент поддерживает рисование, то у него обязано быть такое свойство.

Canvas в переводе с английского означает холст. Получается, что каждый компонент – это холст, на котором нарисовано изображение компонента. Взглянём на кнопку. На самом деле это не кнопка, а холст, на котором нарисовано изображение кнопки и текст. Когда ты щёлкаешь на кнопку, изображение изменяется и приобретает вид нажатой кнопки.

Графика Windows действительно похожа на рисование на холсте. А для такого рисования необходимо две вещи – карандаш (*Pen*) и кисть (*Brush*). Именно такие свойства и присутствуют у объекта *Canvas*. Карандаш используется для рисования линий и контуров, а кисть используется для закраски. У обоих есть свои свойства (цвет, тип и т.д.), но чтобы было понятнее, посмотри на рисунок 12.1.1:

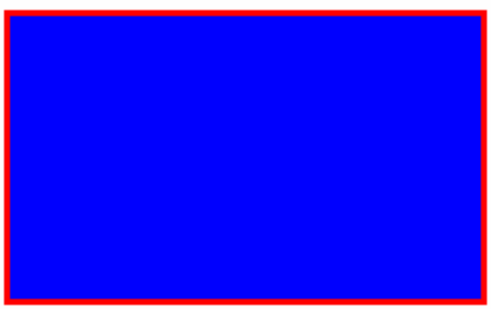


Рисунок 12.1.1 Простой прямоугольник

Это простой прямоугольник. Контур прямоугольника рисуется карандашом (в данном случае красного цвета). Центр прямоугольника закрашивается кистью (у нас синего цвета).

Но не надо думать, что всё приходиться нудно рисовать по пикселям. В Windows для тебя уже заготовлено достаточно инструментов для облегчения процесса работы с графикой. Вообще, большое количество готовых инструментов – это самый большой козырь Windows. Благодаря этому Windows имеет такую популярность у программистов (легче кодить), а значит больше выходит программного обеспечения, а значит, пользователю удобно и есть выбор. Именно это сделало эту ОС популярной.

Графические инструменты Windows объединены под одним названием – GDI (Graphic Device Interface – интерфейс графических устройств). Все функции для работы с графикой находятся одной динамической библиотеке gdi.dll, но подробнее о библиотеках чуть позже.

Ещё одним плюсом GDI является то, что все функции аппаратно независимые. Это значит, что результат вывода графики будет одинаков вне зависимости от графического устройства (видео карты) установленной в компьютере. Ведь каждая карта имеет свои особенности и может работать специфично от других. Но для GDI всё это параллельно. Но тут же выскакивает и минусы GDI:

1. Он не использует ускорения;

2. Слишком медлителен;
3. Поддерживается только двухмерная графика.

Все эти минусы отражаются на том, что GDI не предназначен для создания игр, за то хорошо подходит для офисных приложений. А вот игры хорошо создавать с помощью OpenGL или DirectX, но об этом тоже надо вести отдельный разговор, потому что тема эта слишком большая.

12.2 Первый пример работы с графикой

Давай попробуем написать простейший пример, в котором будет рисоваться простой квадрат. Но для усложнения дела, квадрат будем рисовать на форме и внутри компонента ***TPaintBox***, который очень хорошо подходит для рисования.

Создай новое приложение, и помести на него компонент *PaintBox* с закладки *System*. Постарайся разместить этот компонент на нижней половине окна, как на рисунке 12.2.1.

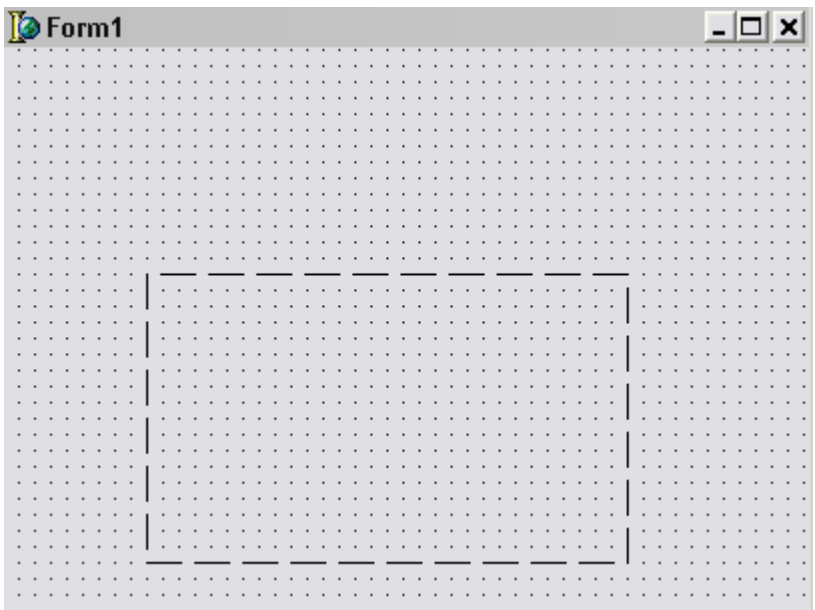


Рис 12.2.1. Форма будущей проги.

Что у формы, что у *PaintBox* есть свойство *Canvas*, значит, на них можно рисовать. Рисование лучше всего производить по событию *OnPaint*, которое так же есть у обоих компонентов. Итак, создадим обработчик события *OnPaint* для формы и напишем тут следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Rectangle(10,10,100,100);
end;
```

Здесь я вызываю метод *Rectangle* объекта *Canvas* нашей главной формы. У этого метода четыре параметра:

1. Левая позиция квадрата;
2. Верхняя позиция квадрата;

3. Правая позиция;
4. Нижняя позиция.

Теперь выдели компонент *PaintBox* и создай такой же обработчик события *OnPaint* для этого компонента. В нём напишите следующее:

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
 PaintBox1.Canvas.Rectangle(10,10,100,100);
end;
```

Здесь я вызываю тот же метод, с таким же параметрами, только для *PaintBox*. Это значит, что этот квадрат будет рисоваться уже внутри компонента *PaintBox*.

Попробуй запустить приложение, и ты увидишь два квадрата (см рисунок 12.2.2). Оба квадрата мы рисуем с помощью метода *Rectangle* с одними и теми же параметрами и по идее, они должны быть нарисованы в одном и том же месте. Но на деле это не так, потому что первый квадрат рисуется на форме, и координаты его отсчитываются относительно формы (10, 0, 100, 100), а второй внутри компонента и координаты отсчитываются относительно этого компонента (10, 0, 100, 100).

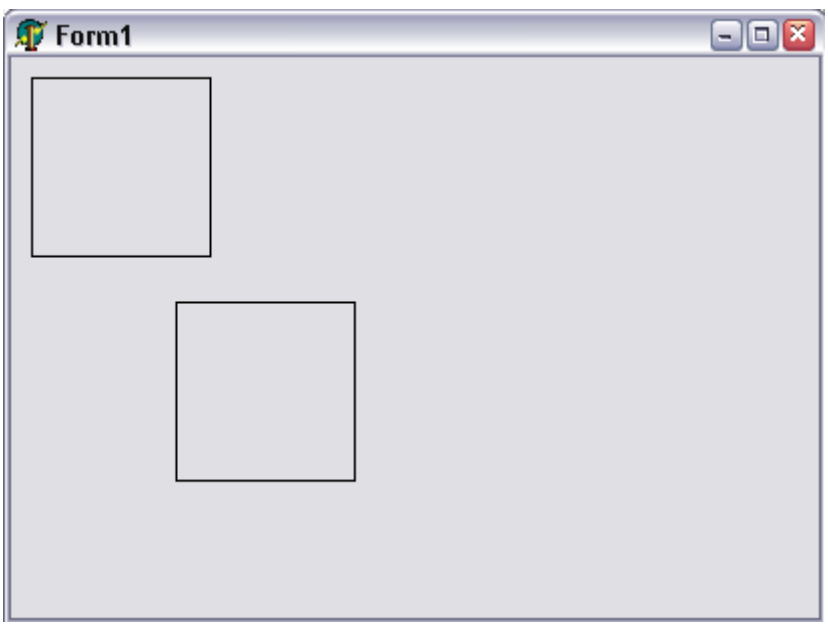


Рис 12.2.1. Форма будущей проги.

*На компакт диске, в директории **\Примеры\Глава 12\Rectangle** ты можешь увидеть пример этой программы.*

12.3 Свойства карандаша

Теперь давай разберёмся с цветом. Как я уже сказал, для рисования используется два понятия цвета – цвет карандаша и цвет кисти. За стиль карандаша (в том числе и цвет) отвечает свойство ***Pen*** объекта ***TCanvas***. За стиль кисти отвечает свойство ***Brush***. И ***Brush*** и ***Pen*** – это тоже объекты, у которых есть свои свойства, о которых мы и поговорим в этой главе.

Для начала разберёмся с объектом ***TPen***. Как я уже сказал, этот объект отвечает за свойства карандаша. У него есть следующие свойства:

Color – цвет карандаша.

Handle – здесь находиться описание карандаша, которое можно использовать при обращении к WinAPI функциям. Вообще-то тебе пора уже запомнить, что у большинства объектов есть свойство *Handle*, которое нужно только для API функций и в повседневных программах мы его использовать не будем.

Mode – режим отображение показывает, как будет рисоваться линия.

Style – стиль карандаша. Существуют следующие стили (графическое отображение стилей линий ты можешь увидеть на рисунке 12.3.1):

- *psSolid* – сплошная линия;
- *psDash* – линия в виде пунктира (состоит из коротких линий);
- *psDot* – линия из точек;
- *psDashDot* – линия с чередующимися чёрточками и точками;
- *psDashDotDot* - линия с чередующимися чёрточками и двумя точками;
- *psClear* – невидимая линия;
- *psInsideFrame* – линия внутри формы. Внешне похожа на сплошную.

Width – ширина карандаша.

Рис 12.3.1 Стили линий

Теперь давай напишем пример, в котором увидим на практике свойства карандаша в действии. Создай новое приложение в Delphi. Создай обработчик события OnPaint и напиши в нём следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 //Рисуем сплошную линию (psSolid)
 Canvas.Pen.Style:=psSolid;
 Canvas.MoveTo(10,20);
 Canvas.LineTo(200,20);

//Рисуем psDash линию
 Canvas.Pen.Style:=psDash;
 Canvas.MoveTo(10,40);
 Canvas.LineTo(200,40);

//Рисуем psDash линию
 Canvas.Pen.Style:=psDot;
 Canvas.MoveTo(10,60);
```

```
 Canvas.LineTo(200,60);

//Рисуем psDashDot линию
 Canvas.Pen.Style:=psDashDot;
 Canvas.MoveTo(10,80);
 Canvas.LineTo(200,80);

//Рисуем psDashDotDot линию
 Canvas.Pen.Style:=psDashDotDot;
 Canvas.MoveTo(10,100);
 Canvas.LineTo(200,100);

//Рисуем psClear линию
 Canvas.Pen.Style:=psClear;
 Canvas.MoveTo(10,120);
 Canvas.LineTo(200,120);

//Рисуем psInsideFrame линию
 Canvas.Pen.Style:=psInsideFrame;
 Canvas.MoveTo(10,140);
 Canvas.LineTo(200,140);
end;
```

Результат работы программы ты можешь увидеть на рисунке 12.3.2.

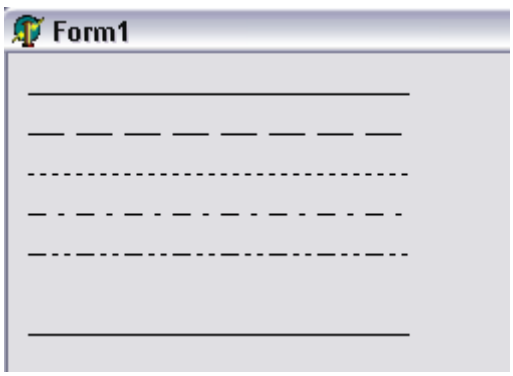


Рисунок 12.3.2 Результат работы программы.

В данном примере, по событию *OnPaint* (когда надо перерисовать форму) я поочерёдно рисую линии разного стиля. Для этого я сначала выбираю нужный стиль (например, *Canvas.Pen.Style:=psSolid* – выбирает стиль сплошной линии).

Потом я перемещаю карандаш в точку начала линии - *Canvas.MoveTo(X, Y)*. Метод *MoveTo* перемещает карандаш в позицию указанную в качестве параметров X, Y. При перемещении не происходит никакого рисования на холсте (*Canvas*). X и Y – это не сантиметры и не миллиметры, а количество пикселей (количество экранных точек).

Отсчёт координаты ***X*** идёт слева на право. Это значит, что левая сторона окна равна нулевой позиции ***X***, а правая сторона окна – максимальное значение. Но это не значит, что ***X*** не может быть отрицательным или больше максимума. Ты без проблем можешь указывать любые значения, только нужно учитывать, что часть линии может уйти за пределы окна.

Отсчёт координаты ***Y*** идёт сверху вниз. Это значит, что верхнее обрамление окна является нулевой точкой ***Y***. При этом заголовок окна (с названием формы и системными кнопками) не входит в пространство окна.

Теперь я должен нарисовать линию с помощью метода *LineTo(X, Y)*. В качестве параметров передаются коечные координаты линии. Отрезок будет нарисован, начиная от текущей позиции карандаша, куда мы перешли с помощью метода *MoveTo* и до координат, указанных при вызове метода *LineTo*.

После прорисовки первой линии, я выбираю следующий стиль и перемещаюсь в позицию на 20 пикселей ниже уже нарисованной линии и рисую следующую линию.

Теперь добавим в нашу программу возможность смены цвета карандаша. Для этого бросим на форму кнопку с надписью «*Изменить цвет*» и компонент *ColorDialog*  с закладки *Dialogs*. Компонент *ColorDialog* предназначен для отображения стандартного диалога выбора цвета. На форме он будет выглядеть в качестве простого квадратика с пиктограммой и при запуске не будет виден. Обновлённую форму ты можешь увидеть на рисунке 12.3.3.

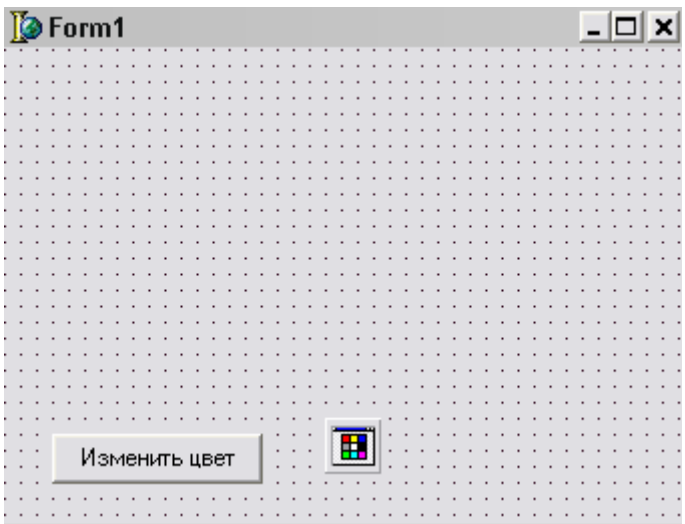


Рисунок 12.3.3 Обновлённая форма будущей программы

По событию *OnClick* для кнопки пишем:

```
if ColorDialog1.Execute then
  Canvas.Pen.Color:=ColorDialog1.Color;
 FormPaint(nil);
```

В первой строке я отображаю окно выбора цвета (*ColorDialog1.Execute*). Если пользователь выбрал цвет, а не нажал «Отмена», то окно возвращает значение *true*, поэтому я проверяю, если результат показа окна равен *true*, то изменить цвет:

```
if ColorDialog1.Execute then
  Изменить цвет холста
```

Напоминаю, что по умолчанию конструкция **if** проверяет указанный код на равенство *true* если не указано обратное. Поэтому эту же конструкцию можно было бы записать так:

```
if ColorDialog1.Execute=true then
   Изменить цвет холста
```

Результат выбранного цвета записывается в свойство *Color* компонента *ColorDialog1*. Именно его мы и присваиваем цвету карандаша *Canvas.Pen.Color*. После этого нужно только перерисовать рисунок. Для этого я явно вызываю процедуру обработчик события *OnPaint* формы. У нас обработчик называется *FormPaint*, именно его я и вызываю.

Можешь запустить программу и проверить результат работы смены цветов линий.

Теперь добавим возможность выбора толщины линии. Для этого бросим компонент *UpDown*  с закладки *Win32*. По событию OnClick по этому компоненту напишем следующий код:

```
procedure TForm1.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
 Canvas.Pen.Width:=UpDown1.Position;
 Repaint;
end;
```

Напоминаю, что компонент *UpDown* состоит из двух кнопок – верхняя увеличивает внутренний счётчик, а нижняя уменьшает. Текущее значение счётчика можно прочитать в свойстве *Position*. Именно это значение я и присваиваю в свойство ширины карандаша *Canvas.Pen.Width*.

После этого я вызываю метод главной формы *Repaint*. Этот метод генерирует событие о том, что надо перерисовать содержимое окна. Это значит, что будет автоматически вызван обработчик события *OnPaint*. Результат – тот же, что и просто вызов напрямую обработчика, как мы это делали после смены цвета, но такой способ считается более правильным. Я в своих программах пользуюсь обоими способами, и отдать предпочтение одному из них не могу. Если говорить о том, какой способ более правильный, то оба они работают без проблем, просто второй способ более эстетичный и красивый, хотя и требует дополнительных затрат на генерацию сообщения о необходимости перерисовать окно.

 *На компакт диске, в директории* ***\Примеры\Глава 12\Pen*** *ты можешь увидеть пример этой программы.*

12.4 Свойства кисти

За параметры кисти отвечает свойство *Brush* объекта *TCanvas*. Как я уже говорил, кисть используется для закраски замкнутых пространств. Она тоже имеет объектный тип как и карандаш, а значит обладает своими свойствами и методами.

У объекта кисти ***TBrush*** есть несколько свойств влияющих на параметры кисти:

Bitmap – картинка, которая будет использоваться в качестве фона закраски. Картинка должны быть формата 8x8 пикселей. Если будет больше, то задействованы будут только пиксели верхнего левого квадрата 8x8.

Мы пока не трогали картинки, поэтому я это свойство рассматривать пока не буду. Единственное, что я сделаю – приведу небольшой кусок кода, а потом ты сможешь вернуться к нему и разобраться самостоятельно:

```
var
  Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create; //Создаётся картинка
  try
    Bitmap.LoadFromFile('MyBitmap.bmp'); //Загружается картинка
    Form1.Canvas.Brush.Bitmap := Bitmap; //Присваивается в качестве фона
    Form1.Canvas.Rectangle(0,0,100,100); // Рисуется квадрат
  finally
    Form1.Canvas.Brush.Bitmap := nil; // Обнуляется фон
    Bitmap.Free; // Уничтожается картинка
  end;
end;
```

Color – так же как и у карандаша, у кисти тоже может быть свой цвет.

Handle – такой же указатель, как и у карандаша, но на кисть.

Style – стиль фона. Здесь могут быть следующие значения: bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross. На рисунке 12.4.1 ты можешь увидеть графическое отображение каждого из стилей.

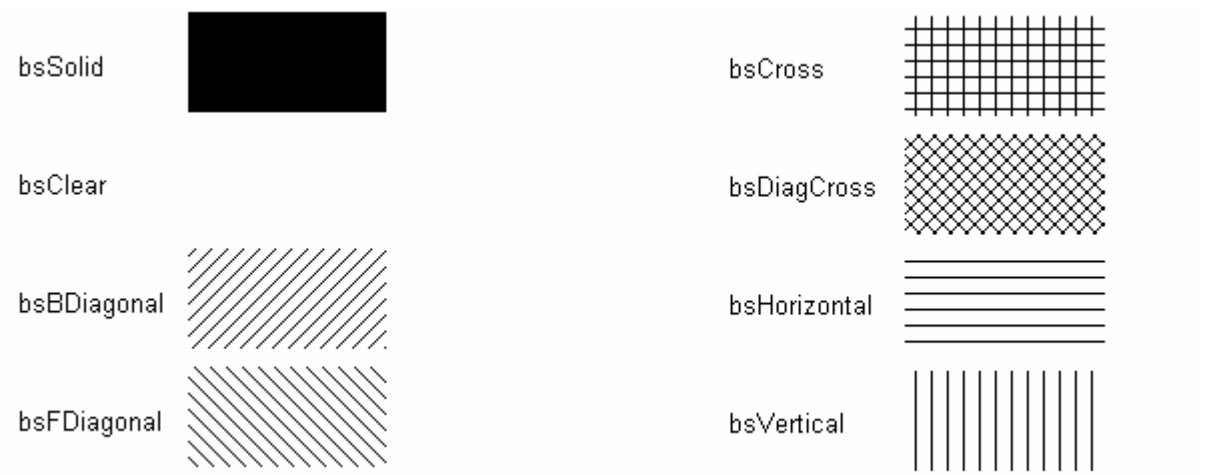


Рисунок 12.4.1 Стили фона

Теперь перейдём к практической части работы с кистью и напишем небольшой пример. Создай новый проект и давай приступим к кодингу.

Как и в прошлом примере, давай создадим обработчик события OnPaint для формы, чтобы по этому событию производить рисование. В обработчике напишем следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Brush.Style:=bsSolid;
```

```
Canvas.Rectangle(10,10,50,50);

Canvas.Brush.Style:=bsBDiagonal;
Canvas.Rectangle(10,110,50,150);

Canvas.Brush.Style:=bsFDiagonal;
Canvas.Rectangle(10,160,50,200);

Canvas.Brush.Style:=bsCross;
Canvas.Rectangle(110,10,150,50);

Canvas.Brush.Style:=bsDiagCross;
Canvas.Rectangle(110,60,150,100);

Canvas.Brush.Style:=bsHorizontal;
Canvas.Rectangle(110,110,150,150);

Canvas.Brush.Style:=bsVertical;
Canvas.Rectangle(110,160,150,200);

Canvas.Brush.Style:=bsClear;
Canvas.Rectangle(10,60,50,100);
end;
```

Здесь код разбит на блоки по две строчки. В первой строчке я задаю стиль кисти, а во второй рисую прямоугольник с помощью метода Rectangle(x, y, r, b), где:

x – левая сторона прямоугольника;
y – верхняя сторона прямоугольника;
r – правая сторона прямоугольника;
b – нижняя сторона прямоугольника.
Чтобы было более ясно, взгляни на рисунок 12.4.2.

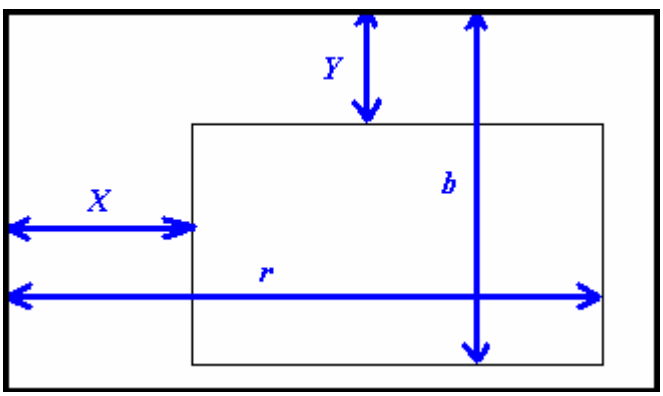


Рисунок 12.4.2 Размеры прямоугольника

Таким образом, я рисую восемь прямоугольников с разными стилями кисти. Если ты запустишь сейчас этот пример, то не заметишь никакой разницы, все прямоугольники будут одинаковыми. Это потому что сейчас цвет кисти имеет такой же цвет, что и форма, поэтому всё сливается. Чтобы увидеть разницу надо изменить цвет фона кисти.

Давай бросим на форму кнопку «Изменить цвет» и компонент *ColorDialog* и по нажатию на неё напишем:

```
if ColorDialog1.Execute then
  Canvas.Brush.Color:=ColorDialog1.Color;
```

```
FormPaint(nil);
```

Здесь я запускаю окно изменения цвета, и если цвет выбран, то присваиваю его кисти *Canvas.Brush.Color:=ColorDialog1.Color*.

Вот теперь можешь запускать программу и смотреть результат. Щёлкни по кнопке «Изменить цвет» и выбери что-нибудь из тёмных, например, синий. Результат смотри на рисунке 12.4.3.

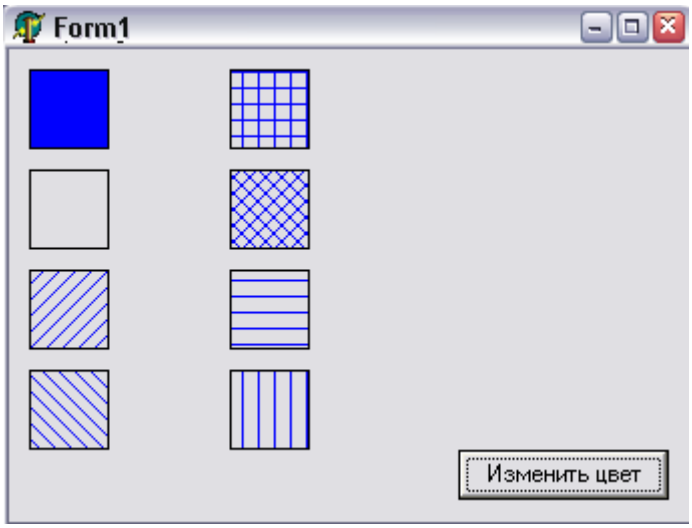


Рисунок 12.4.3 Результат работы программы

Теперь пару замечаний:

1. Заметь, что прямоугольник с невидимой кистью (*Style=bsClear*) я рисую последним, хотя на форме он расположен во второй строке первой колонки. Это связано с тем, что после рисования с невидимой кистью цвет теряется. Попробуй поставить рисование с невидимой кистью где-нибудь раньше, и ты увидишь, что до прямоугольника с прозрачной кистью фон будет того цвета, что ты выбрал, а после будет белого цвета (рисунок 12.4.4).
2. Если программу свернуть и потом развернуть, то цвет кисти опять же будет белого цвета. Это связано с тем, что когда мы нарисовали последний квадрат (который был с прозрачным фоном) цвет кисти всё же изменился на белый. Мы этого не увидели, потому что последний квадрат был с прозрачным фоном. При следующей прорисовке цвет уже изначально белый. Чтобы избавиться от этого эффекта, после каждого рисования с фигур прозрачной кистью надо устанавливать цвет. Для большей надёжности желательно устанавливать цвет непосредственно перед рисованием.

Так как у нас после рисования с прозрачным фоном больше не будет вывод ан экран, то тут уже нет смысла устанавливать цвет. А вот перед началом рисования это делать всё же желательно:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 //Обязательно устанавливаю цвет кисти
 Canvas.Brush.Color:=ColorDialog1.Color;

 //Рисую первый квадрат
```

```
Canvas.Brush.Style:=bsSolid;
Canvas.Rectangle(10,10,50,50);
```

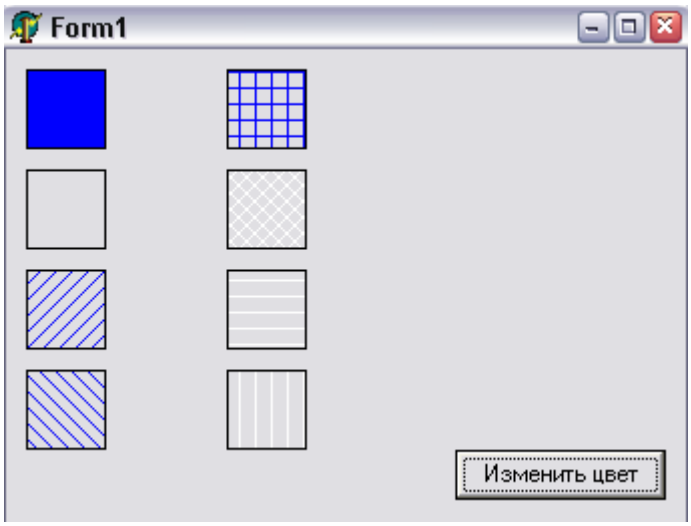


Рисунок 12.4.4 Последние три прямоугольника имеют белый цвет кисти, потому что прям перед этим, я нарисовал прямоугольник с прозрачным фоном.

*На компакт диске, в директории **\Примеры\Глава 12\Brush** ты можешь увидеть пример этой программы.*

12.5 Работа с текстом в графическом режиме

Конечно же, название этой части достаточно расплывчато и не точно, потому что Windows сам по себе графический и вся работа сама по себе уже графическая. Но всё же иногда мы работаем с текстом, воспринимая его как текст, а иногда мы прямо выводим его в виде графики.

Для вывода текста на экран у объекта ***TCanvas*** есть метод *TextOut*. У этого метода три параметра:

1. **X** позиция текста;
2. **Y** позиция текста;
3. Непосредственно строка текста, которую надо вывести.

Создай новое приложение и по событию *OnPaint* напиши:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.TextOut(100,100, 'Привет всем!!!');
end;
```

Здесь я просто вывожу на экран текст в координатах (100, 100).

За стиль шрифта отвечает свойство *Font* объекта ***TCanvas***. Это свойство тоже имеет объектный тип (***TFont***), у которого очень много свойств. Среди них, конечно же, есть и свойство *Color*, так что давай бросим на форму кнопку и *ColorDialog*, чтобы можно было менять цвет текста.

По событию OnClick для кнопки пишем следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if ColorDialog1.Execute then
  FormPaint(nil);
end;
```

Здесь я показываю окно выбора цвета и если цвет выбран, то просто перерисовываю окно. А где же изменение цвета шрифта? Я же уже сказал, когда мы работали с кистью, что цвет нужно менять непосредственно перед рисованием, и здесь это делать нет смысла. Поэтому корректируем событие *OnPaint*:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Font.Color:=ColorDialog1.Color;
 Canvas.TextOut(100,100, 'Привет всем!!!');
end;
```

Теперь бросим на форму компонент *FontDialog* с закладки *Dialogs*. Это почти такой же компонент, как мы использовали для смены цвета, только здесь будет появляться стандартное окно смены шрифта. Добавь на форму кнопку с надписью «Изменить шрифт» и по её нажатию напиши следующее:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 if FontDialog1.Execute then
  Canvas.Font:=FontDialog1.Font;
 FormPaint(nil);
end;
```

В первой строчке я показываю окно смены шрифта так же, как это делалось для смены цвета. Если пользователь выбрал шрифт и нажал «ОК», то я устанавливаю его свойству *Font* объекта *Canvas*. После этого я заставляю форму обновить своё содержимое. При новой прорисовке содержимого формы, текст уже выводиться с новым шрифтом.

Единственный недостаток этого примера – если сначала выбрать большой шрифт, а затем маленький, то старое содержимое текста, написанного большим шрифтом, не уничтожается, а новый рисуется поверх старого (см рисунок 12.5.1).

Самый простейший способ избавиться от такого нежелательного эффекта – после смены шрифта вместо прямого вызова функции *FormPaint(nil)* использовать вызов метода формы *Repaint* или *Invalidate*. О первом из них я уже говорил, и мы его уже использовали. Второй метод *Invalidate* имеет практически тот же смысл, только заставляет прорисоваться полностью всё окно, а не только его содержимое.

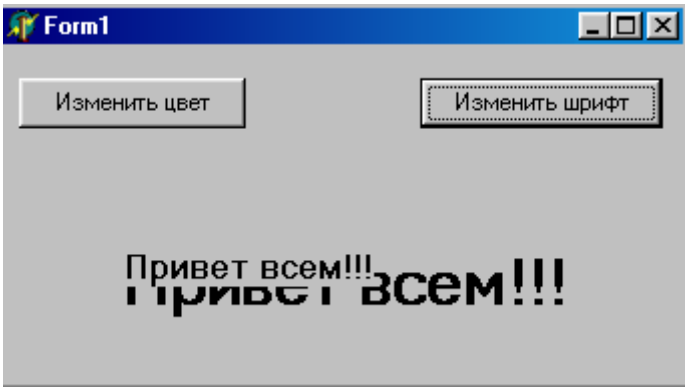


Рисунок 12.5.1 Ненужный эффект наложения старого текста на новый. Маленький новый текст рисуется поверх старого большого.

Итак, мы научились менять все параметры текста с помощью стандартного диалогового окна. Как менять отдельные свойства – это отдельная тема, ведь свойство *Font* имеет объектный тип ***TFont***, с массой свойств – имя шрифта, стиль, цвет, размер. Но всё это отдельная тема и мы её пока опустим.

На компакт диске, в директории ***\Примеры\Глава 12\Text*** *ты можешь увидеть пример этой программы.*

12.6 Вывод текста под углом

Сейчас я хочу показать тебе один трюк – как можно вывести текст под углом. Как ты мог заметить, пока мы увидели только функцию, которая умеет выводить тест горизонтально и никаких больше намёков на возможность развернуть текст и написать его например вертикально.

Создай новый проект. Теперь создай обработчик события *OnCreate* для главной формы. В этой процедуре напиши следующее:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 index:=0;
end;
```

Index - это у нас будет счётчик, но его ещё надо объявить, поэтому иди в объявления private и напиши следующее:

```
private
 { Private-Deklarationen }
 index:Integer;
 cl:Boolean;
```

Теперь мы познакомимся с типами данных Delphi. Мы объявили с тобой две переменные: *index* и *cl* . Первая из них - это целое, знаковое число. Вторая - это буленовское, и может принимать значения ***true*** или ***false*** .

Основные приготовление закончены, и мы можем переходить непосредственно к программированию. Создай обработчик события для главной формы *OnMouseDown*.

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
Var
 A: Integer; // Объявление переменной A - целое число.
begin
   A := random(3600);
   CanvasSetAngle(Canvas, A / 10);
   Canvas.TextOut(X, Y, FormatFloat('##0.0', A/10)+'°');
end;
```

Давай рассмотрим текст процедуры. В первой строчке мы используем функцию *random* , она возвращает случайное значение, но не больше чем число, указанное в скобках. В нашем случае - это 3600.

Вторую строчку я опущу, а рассмотрим сразу третью. *Canvas.TextOut* - выводит текст на форме и мы с ней уже работали.

В качестве текста я опять использовал процедуру: *FormatFloat*. Эта процедура переводит число с запятой (вещественное, или так сказать дробное) с учётом формата.

```
function FormatFloat(
 const Format: string; // Строка формата
Value: Extended // Число
): string;
```

В качестве формата я указал **##0.0**, что приводит указанное число к этому виду, т.е. отрезает все числа после запятой, оставляя только одно (об этом говорит один ноль после запятой в строке формата). Перед запятой может быть любое количество чисел, потому что стоит два знака решётки. В качестве числа я указываю переменную **A** делённую на 10.

Теперь возвращаемся ко второй строке. *CanvasSetAngle* - этой процедуры ещё нет, мы её должны написать. Я сейчас приведу весь текст программы, а потом мы рассмотрим эту процедуру отдельно.

```
unit Textrot1;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls;
type
  TForm1 = class(TForm)
    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Timer1Timer(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private-Deklarationen }
    index:Integer;
    cl:Boolean;
  public
```

```
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure CanvasSetAngle(C: TCanvas; A: Single);
var
  LogRec: TLOGFONT; // Объявляем переменную логического шрифта
begin
 GetObject(C.Font.Handle,SizeOf(LogRec),Addr(LogRec));
 LogRec.lfEscapement := Trunc(A*10);
 LogRec.lfOrientation := Trunc((A+10) * 100);
 C.Font.Handle := CreateFontIndirect(LogRec);
end;

procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
Var A: Integer;
begin
   A := Random(3600);
   CanvasSetAngle(Canvas, A / 10);
   Canvas.TextOut(x, Y, FormatFloat('##0.0', A/10)+'°');
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
 index:=0;
 Canvas.Brush.Style:=bsClear;
end;
end.
```

В качестве параметров для функции *CanvasSetAngle* мы передаём процедуре *Canvas* и угол разворота текста. Угол разворота - имеет значение *Single* - что означает вещественное (дробное). До этого имя процедур вместо нас писал *Delphi*. Эту процедуру тебе придётся вписывать своими руками, потому что она самостоятельная и не принадлежит никакому объекту

Теперь перейдём к содержимому процедуры. Рассмотрим по частям первую строчку.

```
GetObject //Это функция возвращает информацию о графическом объекте
C.Font.Handle // Объект на который нужно получить значение.
SizeOf(LogRec)// Передаем размер возвращаемого значения
Addr(LogRec)// Передаём адрес возвращаемого значения
```

С помощью этой функции, мы получаем информацию о шрифте, используемом нами для рисования. Вторая и третья строчки этой процедуры изменяют значения полученной информации. Четвёртая функция записывает изменённую информацию.

Запусти получившееся приложение, и пощёлкай по форме мышкой. В месте щелчка должен появиться текст под случайным углом. Когда наиграешься, закрой программу и я продолжу.

Теперь поставь на форме Timer , который находиться в закладке System . Создай для таймера обработчик события OnTimer и напиши в нём следующее:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
 CanvasSetAngle(Canvas, index);
 Canvas.TextOut(100, 100, 'CyD Soft');
 index:=index+45;
 if index>=360 then
  begin
   index:=0;
   if cl then
    Canvas.Font.Color:=clBlack
   else
    Canvas.Font.Color:=clRed;
   cl:=not cl;
  end;
end;
```

Эта процедура будет вызываться каждый раз, когда пройдёт интервал времени указанный в свойстве *Interval* компонента *Timer*. По умолчанию там указано 1000 (число в миллисекундах, что равно 1 секунде), значит, процедура будет вызываться через каждую секунду.

Внутри процедуры я выставляю угол, на который надо вывести текст (*CanvasSetAngle*), потом вывожу текст (с помощью *Canvas.TextOut*).

После этого я прибавляю переменной index значение 45. В этой переменной храниться значение угла, под которым надо вывести текст и через каждую секунду эжто значение увеличивается на 45 градусов.

Далее идёт проверка - если index больше или равен 360, значит мы прошли полный круг, а если так, то выполняется следующий код:

```
   index:=0;
   if cl then
    Canvas.Font.Color:=clBlack
   else
    Canvas.Font.Color:=clRed;
   cl:=not cl;
```

Здесь в первой строке я обнуляю переменную index, чтобы начать вывод текста с 0 градусов (мы же уже прошли полный круг). Дальше я проверяю значение переменной *cl* если она равна *true*, то значению цвета текста будет присвоено *clBlack*, что равно чёрному цвету. Иначе цвет смениться на красный *clRed*.

После этого я изменяю значение переменной *cl* на противоположное, о чём говорит конструкция ***cl:=not cl***. Здесь я присваиваю переменной противоположное (*not*) значение её самой. Это значит, что если *cl* равнялась *true*,то после этого кода будет равняться *false*. Так что после прохождения текстом очередного круга, цвет будет меняться с чёрного, на красный и обратно.

 *На компакт диске, в директории* ***\Примеры\Глава 12\TextAngle*** *ты можешь увидеть пример этой программы.*

Попробуй запустить пример. Пускай он немного поработает, чтобы форма заполнилась текстом. Теперь попробуй перекрыть окно программы другим окном или свернуть его. Потом снова восстанови окно. Что ты видишь? Всё, что было нарисовано - исчезло. Это потому что Windows не сохраняет содержимое окна. Мы сами должны его восстанавливать.

Когда окно свернулось и восстановилось, то генерируется событие *OnPaint*, по которому нужно перерисовать содержимое окна. Поэтому все функции рисования стараются располагать именно в обработчике этого события. Так мы будем рисовать и реагировать на события когда надо перерисовать содержимое экрана одной и той же функцией.

В нашем примере использование события *OnPaint* неудобно. В таких случаях на форму ставят компонент *TImage* и рисуют в нём. Этот компонент, в отличии от формы сохраняет своё содержимое. Когда ты рисуешь на холсте компонента *TImage*, то ты рисуешь в специально отведённой памяти. А вот когда компонент *TImage* нуждается в прорисовке, то эта область памяти копируется на сам компонент. Таким образом не надо самостоятельно ни за чем следить. Подробнее с компонентом *TImage* мы познакомимся немного позже.

12.7 Работа с цветом

Мы уже научились менять цвет и даже в предыдущей части узнали, что константы clBlack равна чёрному цвету, а clRed красному. Но есть ещё много констант, которые определяют стандартные цвета для более удобного использования. Вот именно с ними нам предстоит сейчас познакомиться и узнать, как храниться цвет в памяти машины.

Цвет храниться в виде типа *TColor*. Хотя в названии типа в начале стоит буква **T**, этот тип не объектный, а просто число из 4-х байт, хотя реально нас будут интересовать только последние три.

Ты наверно должен знать, что в компьютерной графике цвет представляется тремя составляющими красного, зелёного и голубого (RGB). В разных пропорциях, из этих трёх цветов можно получить любой другой. Например, если взять красного и зелёного по максимуму, а синего вообще не брать, то получиться жёлтый цвет.

Каждый из цветов представляется в виде одного байта, так что для хранения трёх цветов достаточно 3 байтов. Но зачем же тогда для ***TColor*** выделено 4-е байта? Да потому что в компьютере регистры чётные и могут хранить только 1, или 2, или 4 байта. Так что у переменной цвета один байт избыточен (первый) и чаще всего равен нулю. В играх и графических пакетах этому байту нашли применение – он часто указывает прозрачность, но в офисных приложениях его просто игнорируют.

Как ты уже знаешь, один байт может принимать значения от 0 до 255 (в десятичной форме) или от 0 до FF (в шестнадцатеричной). Так что в шестнадцатеричной форме цвет будет выглядеть как $00FFFFFF. Только тут сразу надо отметить, что первые два нуля – это лишний байт, потом идут FF для голубого цвета, потом FF для зелёного и последние FF для красного. Получается, что в памяти цвет храниться как BGR (в обратном порядке). Абсолютно красный цвет будет равен $000000FF, абсолютно зелёный = $0000FF00, а голубой - $00FF0000.

Давай попробуем научиться работать с цветом на практике, заодно и познакомимся с необходимыми функциями. Создай новое приложение и брось на него компоненты так, как показано у меня на рисунке 12.7.1.

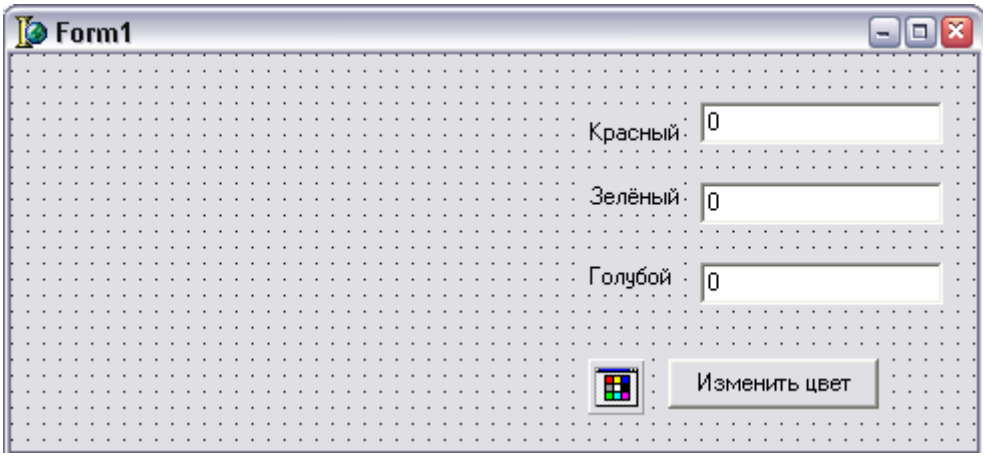


Рисунок 12.7.1 Форма будущей программы

Итак, у нас на форме три компонента ***TEdit***. Для красного я компонент назвал *RedEdit*, для зелёного *GreenEdit*, ну и для синего *BlueEdit*. Так же на форме есть кнопка для смены цвета (её имя не имеет значения) и *ColorDialog*, для смены цвета.

Если ты сам создаёшь пример, а не пользуешься готовым с диска, то постарайся всё разместить так, как у меня на рисунке (ближе к правому краю), потому что слева мы будем рисовать квадрат.

По нажатию кнопки пишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 c:LongInt;
begin
 if not ColorDialog1.Execute then
  exit;

 C:=ColorToRGB(ColorDialog1.Color);
 RedEdit.Text:=IntToStr(GetRValue(C));
 GreenEdit.Text:=IntToStr(GetGValue(C));
 BlueEdit.Text:=IntToStr(GetBValue(C));

 Repaint;
end;
```

В разделе **var** объявлена одна переменная целого типа *Longint*. Это целое число размером в 4-е байта и будет использоваться для хранения значения цвета.

В первой строчке я показываю окно смены цвета *ColorDialog1.Execute*. Если пользователь не выбрал цвет (об этом говорит конструкция **if not**), то мы прерываем выполнение процедуры с помощью выхода из неё - *exit*.

Дальше я преобразовываю выбранный цвет *ColorDialog1.Color* из типа ***TColor*** в простое число с помощью функции *ColorToRGB*. Этой функции надо передать цвет в виде ***Tcolor*** (мы передаём *ColorDialog1.Color*) и она вернёт целое 4-х байтное число, которое я записываю в переменную **C**.

В следующей строке я присваиваю строке ввода *RedEdit* значение красной составляющей цвета. Для этого я сначала использую функцию *GetRValue*. Ей я передаю значение цвета в виде целого числа (переменная **C**). Результат – однобайтное число, которое показывает значение красной составляющей. Этот результат – число и прежде

чем его присваивать в строку ввода, его надо преобразовать в строку. Для этого я превращаю его в текст с помощью знакомой нам функции *IntToStr*.

То же самое я проделываю и с зелёным цветом в следующей строке кода. Только для получения зелёной составляющей я использую функцию *GetGValue*.

Для получения синей составляющей, я использую функцию *GetBValue*. Таким образом, после выполнения таких действий, я разбил 4 байта цвета из переменной **C**, на отдельные байты по цветам и разнёс их в соответствующие строки ввода.

После этого я заставляю окно прорисоваться *Repaint*.

По событию *OnPaint* напишем следующий код:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Brush.Color:=RGB(StrToIntDef(RedEdit.Text, 0),
          StrToIntDef(GreenEdit.Text, 0), StrToIntDef(BlueEdit.Text,0));
 Canvas.Rectangle(10,10, 250, 150);
end;
```

Здесь мне надо проделать обратные действия – превратить три составляющих цвета из строк ввода в одно целое значение цвета. Для этого используется функция *RGB(R, G, B)*. У этой функции три параметра и все они целые числа:

R – значение красного цвета;

G – значение зелёного цвета;

B – значение синего цвета.

В качестве параметров я передаю значения указанные в соответствующих строках ввода, предварительно преобразовывая их из строк в числа. Чтобы было яснее, в приведённом выше коде я раскрасил код соответствующими цветами.

Результат преобразования цвета я записываю в цвет кисти. После этого я рисую прямоугольник, у которого цвет фона будет тот, что мы выбрали.

И последнее что мы сделаем – создадим обработчик события *OnChange* для всех строк ввода. Выдели сначала строку ввода для красного цвета, потом удерживая Shift щёлкни по остальным. У тебя должны быть выделены все строки ввода серыми рамками. Теперь перейди в объектном инспекторе на закладку *Events* и дважды щёлкни по *OnChange*, чтобы создать обработчик. В нём напиши:

```
procedure TForm1.RedEditChange(Sender: TObject);
begin
 Repaint;
end;
```

Попробуй запустить этот пример. Теперь выбери какой-нибудь цвет, и ты увидишь составляющие этого цвета. Можешь даже напрямую изменять значения в строках ввода, и результат моментально будет отражаться на цвете прямоугольника. Только помни, ни одна из составляющих не может быть больше 255!!!

На рисунке 12.7.2 ты можешь увидеть пример работы моей программы. Потренируйся с ним и возможно ты поймёшь то, что не понял из моих слов. Словами можно объяснить многое, а практика показывает ещё больше.

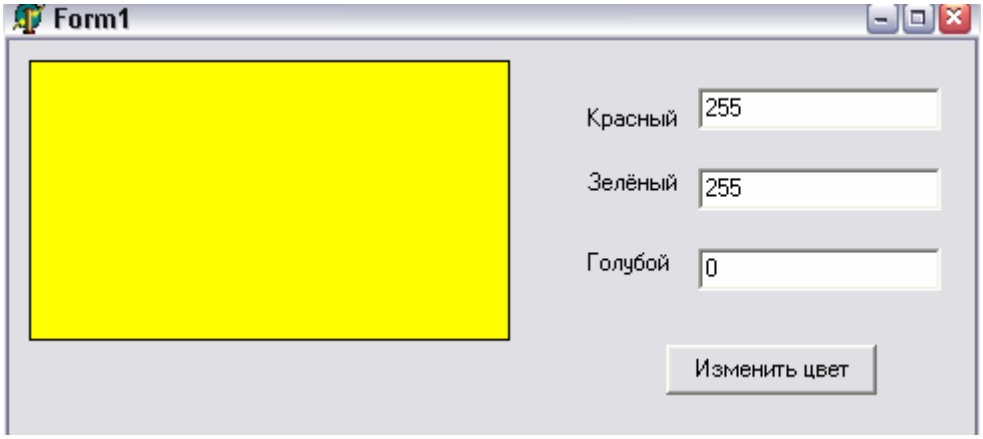


Рисунок 12.7.2 Результат работы программы

На компакт диске, в директории ***\Примеры\Глава 12\Color*** *ты можешь увидеть пример этой программы.*

Ну а теперь познакомимся с константами, которые уже заготовлены для основных цветов. Ты можешь их реально использовать в своих программах и присваивать, как в примере из главы 12.6. Я не буду перечислять все константы, потому что ты можешь их сам в любой момент найти, если щёлкнешь в объектном инспекторе по свойству *Color* любого компонента. Всё, что ты там увидишь в списке, это и есть константы, которые можно использовать. Я сам всегда пользуюсь этим методом, потому что сразу вижу константу и цвет.

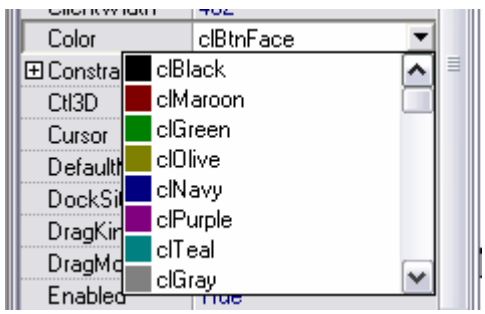


Рисунок 12.7.3 Цветовые константы

12.8 Методы объекта TCanvas

Последнее, что нам надо узнать в этой части книги – познакомиться с методами рисования. Пока что мы использовали только линии, прямоугольники и текст, но на этом возможности объекта ***TCanvas*** не заканчиваются.

Pixels

Первое, чем мы познакомимся, не будет методом – это свойство *Pixels*. Это свойство – двухмерный массив, указывающий на битовую матрицу изображения. Что это значит? Проще всего показать. Допустим, что тебе нужно поставить точку чёрного цвета в координатах (10,10). Для этого ты пишешь следующий текст:

```
Canvas.Pixels[10,10]:=Black;
```

С помощью этого же свойства можно узнать цвет в какой-нибудь точке. Например:

```
var
 c:TColor;
begin
 c:=Canvas.Pixels[10,10];
 if c=clBlack then
   //Точка с координатами (10, 10) чёрного цвета
End;
```

TextWidth и TextHeight

Давай сразу познакомимся ещё с двумя методами для работы с текстом – *TextWidth* и *TextHeight*. Обоим методам нужно передать какой-нибудь текст и первый из них вернёт его ширину, а второй – высоту в пикселях. Эти метода очень удобны, когда тебе нужно выводить форматированный текст.

Допустим, что тебе надо вывести две строки текста. Ты можешь вывести одну из них, а потом чуть ниже вывести другую. А вот теперь самое интересное – на сколько ниже? Ведь если взять слишком много, то будет большой промежуток, а если слишком мало, то одна строка наедет на другую или попросту её перекроет. Эти методы позволяют точно узнать длину или высоту текста, в зависимости от используемого шрифта.

Используй их каждый раз, когда это нужно и не надейся на собственные расчёты, потому что шрифт на разных машинах может выглядеть по-разному.

Arc

Следующий метод объекта ***TCanvas*** – это метод *Arc*, который предназначен для рисования дуги. У него аж 8 параметров - X1, Y1, X2, Y2, X3, Y3, X4, Y4. Как видишь, это 4 пары координат X и Y, которые указывают 4 точки через которые надо провести дугу.

CopyRect

Этот метод предназначен для копирования указанного прямоугольника из одного объекта ***TCanvas*** в другой. У этого метода три параметра:

Dest: TRect – область, указывающая, куда надо копировать;

Canvas: TCanvas – это объект, из которого надо копировать;

Source: TRect - область, указывающая, откуда надо копировать.

Первый и последний параметр имеют тип ***TRect***. Это простая структура из четырёх целых чисел - *Left, Top, Right, Bottom*. Не трудно догадаться, что это координаты прямоугольника. Для создания переменной такого типа лучше всего использовать функцию *Rect*. Ей нужно передать четыре этих параметра *Left, Top, Right, Bottom* и она вернёт вам готовую структуру.

Давай рассмотрим пример и увидим всё на практике. Допустим, что у нас есть две формы. Мы хотим из второй формы *Form2* скопировать всё её содержимое в первую форму *Form1*. При этом мы отобразим содержимое второй формы на первой в

прямоугольнике размером (10, 10, 110, 110), т.е. На рисунке 12.8.1 показано графически, куда мы будем копировать.

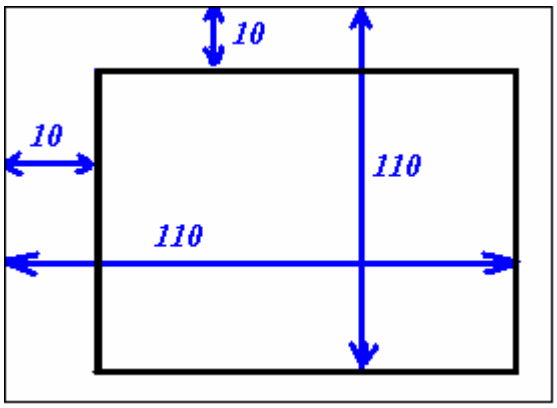


Рис 12.8.1 Область копирования

Этот код будет выглядеть так:

```
var
 SRect, DRect: TRect;// Объявляю две переменные типа TRect
begin
 SRect:=Rect(0, 0, Form2.Width, Form2.Height);
 DRect:=Rect(10, 10, 110, 110);
 Form1.Canvas1.CopyRect(DRect, Form2.Canvas, SRect);
end;
```

В первой строке я заполняю структуру *SRect* с помощью функции *Rect*. При этом я указываю полные координаты окна *Form2* т.е. (0, 0, ширина второй формы, высота второй формы).

Во второй строке я заполняю структуру *DRect* с помощью всё той же функции *Rect*. В принципе, ей можно не пользоваться и заполнять поля как и для любой другой структуры:

```
DRect.Left:=10;
DRect.Top:=10;
DRect.Right:=110;
DRect.Bottom:=110;
```

В этом случае код займёт аж четыре строчки, поэтому я не люблю такие вещи. Лучше всё записать одной строкой.

И последнее - я копирую холст второй формы в первую. Сразу хочу ответить, что если размер области источника больше приёмника, то область будет растянута/сжата до размеров приёмника. Это значит, что если размеры второй формы больше чем 100x100 (именно в такой квадрат на форме 1 мы хотим скопировать вторую форму), то изображения второй формы будет сжато до размеров 100x100.

Draw

Этот метод тоже предназначен для копирования изображений, но другого формата. У него три параметра – X и Y координаты куда копировать и объект типа TGraphic который надо копировать. Этот объект мы ещё не рассматривали и узнаем о нём в следующей главе.

Ellipse

Этот метод предназначен для рисования эллипса (овала). Есть две реализации этого метода:

procedure Ellipse(X1, Y1, X2, Y2: Integer);
procedure Ellipse(const Rect: TRect);

В первом случае нужно передать четыре координаты прямоугольника, в который будет вписан эллипс. Во втором случае достаточно одного параметр типа TRect (как ты уже знаешь, у этой структуры есть все необходимые четыре поля). Какой ты будешь использовать – это твоё дело.

FillRect

У этого метода только один параметр – TRect, указывающий область, которую необходимо залить цветом кисти. В принципе, это то же самое, что и нарисовать прямоугольник.

FloodFill

Заливка. У этого метода четыре параметра – X и Y координаты точки, с которой нужно начинать заливку. Третий параметр – цвет. Последний параметр – способ заливки. Возможны два способа:

fsSurface – залить всю область, где цвет равен цвету указанному в третьем параметре.

fsBorder - залить всю область, где цвет не равен цвету указанному в третьем параметре.

Этих методов пока достаточно. Я не собираюсь переписывать весь файл помощи в своей книге, потому что это бесполезно. Но я показал тебе основные методы, которые тебе могут пригодиться. С ними, и многими другими методами мы познакомимся на практике чуть позже. Возможно уже в следующей главе.

12.9 Компонент работы с графическими файлами (TImage)

Как я уже сказал – первое, с чем мы познакомимся в этой главе, будет компонент для работы с графическими файлами – ***TImage***. Этот компонент ты можешь найти на закладке *Additional* палитры компонентов.

С этим компонентом мы уже немного познакомились в главе 11.5, но тогда мы рассматривали его как компонент украшение, который просто располагает на форме красивое изображение. Тогда я не хотел затрагивать другие возможности компонента, потому что мы ещё не были готовы познакомиться с графикой. Сейчас, когда я уже рассказал всё необходимое, пора разобрать этот компонент по свойствам и методам.

- TImage

Компонент ***TImage*** достаточно универсальный и может отображать картинки разного формата. Но в начальной установке он может загружать только BMP, JPG, JPEG или WMF файлы. Давай посмотрим, как это делается. Создай новое приложение и брось на форму одну кнопку и компонент ***TImage*** с закладки *Additional*.

Теперь брось на форму компонент *OpenPictureDialog* с закладки *Dialogs*. Этот компонент предназначен для отображения на экране стандартного окна открытия картинки. Нам так же понадобиться кнопка, по нажатию которой мы будем отображать окно открытия картинки и потом загружать выбранную.

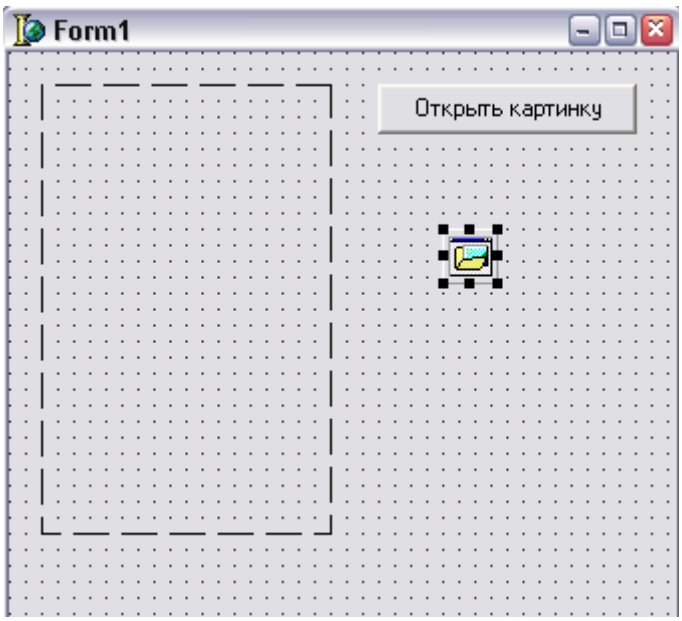


Рис. 12.9.1. Форма будущей программы

На рисунке 12.9.1 ты можешь увидеть форму будущей программы. Но пока готова только форма. Чтобы программа стала полноценной надо написать код загрузки картинки. По нажатию кнопки «Открыть картинку» пишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if OpenPictureDialog1.Execute then
  Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);
end;
```

В первой строке я отображаю стандартное окно открытия картинки. Для этого достаточно вызвать метод *Execute* компонента *OpenPictureDialog1*, т.е. написать *OpenPictureDialog1.Execute*. Этот метод возвращает логическое значение. Если оно равно *true*, то пользователь выбрал файл, иначе нажал отмену. Именно поэтому я проверяю результат вызова метода *Execute* с помощью *if OpenPictureDialog1.Execute then*.

Если файл выбран, то выполняется следующий код:

Image1.Picture.LoadFromFile(OpenPictureDialog1.FileName);

Разберём эту конструкцию по частям. У компонента *Image1* есть свойство *Picture*. Это свойство имеет объектный тип (а значит и свои свойства и методы) ***TPicture***. Этот объект предназначен для работы с изображениями практически любого типа. Он достаточно универсален, в чём мы убедимся достаточно скоро.

Для загрузки изображения я использую метод LoadFromFile (загрузить картинку из файла) объекта *Picture*. В качестве единственного параметра этого метода нужно указать имя открываемого файла или полный путь, если картинка находиться не в той же директории, что и сама программа.

Мы выбираем имя файла с помощью стандартного окна и полный путь к файлу находиться в свойстве FileName компонента OpenPictureDialog1.

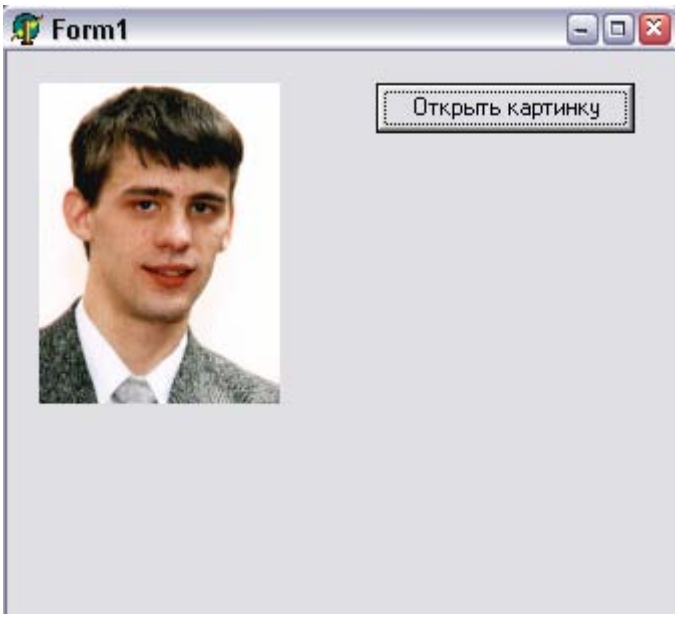


Рис. 12.9.2. Программа в действии

Всё достаточно просто. Попробуй теперь запустить программу и посмотреть на результат её работы. В окне открытия файла посмотри, какие типы файлов ты можешь открывать. В зависимости от версии и установленной комплектации количество типов может быть от 1 до 3 - bmp, ico и wmf.

Давай научим нашу программу работать с jpeg форматом файлов. Не волнуйся, это не сложно и нам не придётся писать сложный алгоритм распаковки изображения. В Delphi уже есть всё необходимое, надо только это необходимое подключить к проекту.

Для начала переместись в раздел **uses** проекта и подключи туда модуль *jpeg*. В этом модуле описано всё необходимое для работы с *jpeg* форматом изображений.

В принципе этого достаточно. Осталось только заставить окно открытия файлов показывать фильтр на данный тип файлов. Для этого выдели компонент *OpenPictureDialog1* и дважды щёлкни по свойству *Filter*.

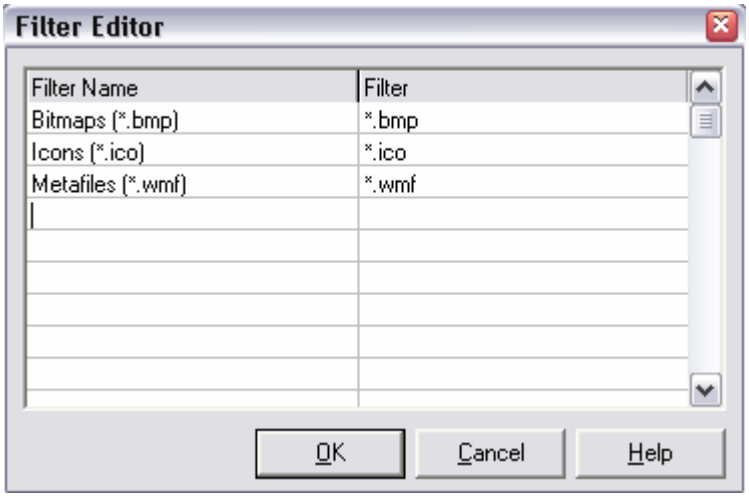


Рис. 12.9.3. Окно редактирования фильтра

В этом окне у тебя есть несколько заполненных строк. У меня (рисунок 12.9.3) только три. В четвёртой строке (первой пустой строке) напиши в первой колонке «*JPEG Files*», а во второй колонке напиши «**.jpg*». Можешь нажимать *OK* и запускать программу. Теперь в окне открытия графического файла можно выбрать тип *JPEG* и открыть нужный файл. Он так же будет загружен в компонент *Image1*, даже не смотря на свой сложный алгоритм сжатия и другой вид хранения данных.

*На компакт диске, в директории **\Примеры\Глава 12\Loading Images** ты можешь увидеть пример этой программы.*

Теперь попробуем модернизировать пример и получить доступ к содержимому картинки. Для этого я буду копировать изображение используя прозрачность. Как? Сейчас узнаешь.

Создай для неё обработчик события *OnPaint* для главной формы в уже созданном нами примере. В созданной процедуре *FormPaint* напиши следующее:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.BrushCopy(Rect(200,16,200+Image1.Width,16+Image1.Height),
        Image1.Picture.Bitmap,
        Rect(0,0,Image1.Width,Image1.Height),
        Image1.Picture.Bitmap.Canvas.Pixels[1,1]);
end;
```

А в процедуре, где мы загружаем изображение нужно в конце добавить вызов метода *Repaint*, чтобы после открытия графического файла форма прорисовалось заново и вызвался обработчик *OnPaint*.

Теперь попробуй запустить программу и загрузить в неё bmp файл. Ты должен увидеть результат подобный рисунку 12.9.4. Слева у нас находиться изображение картинки Image1, а справа мы делаем копию изображения на форму.

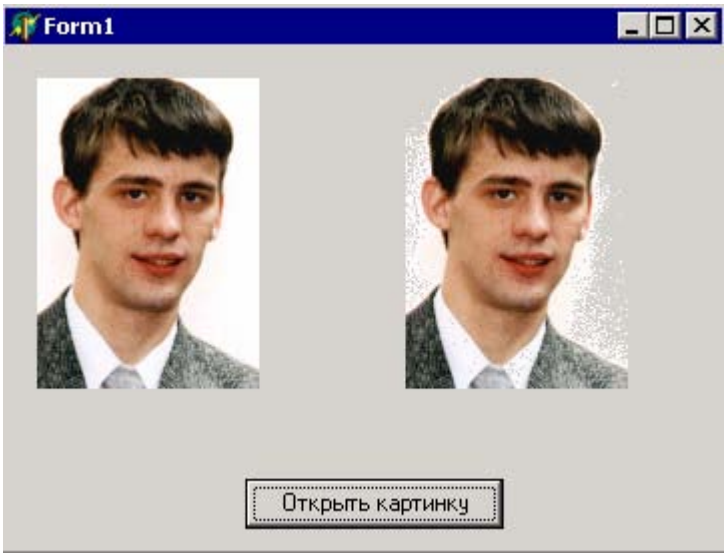


Рис 12.9.4. Результат работы программы

А вот теперь давай посмотрим, что тут происходит. Код кажется немного сложным, но это только на первый взгляд. Давай рассмотрим всё по частям. Мы используем процедуру BrushCopy у уже знакомого Canvas . Эта процедура копирует на Canvas картинку.

```
procedure BrushCopy(
   const Dest: TRect; // Область приёмника
   Bitmap: TBitmap; // Картинка которая будет копироваться
   const Source: TRect; // Область источника
  Color: TColor); // Прозрачный цвет
```

Область приёмника объявлена как *TRect*, который имеет вид *TRect = (Left, Top, Right, Bottom: Integer)*. Что находиться в скобках, я думаю пояснять не надо. То же самое и с областью источника. В качестве картинки мы передаём *Bitmap* из *TImage*.

В качестве прозрачного цвета я использовал цвет пикселя в позиции [1,1] из картинки *TImage*. На это указывает запись *Image1.Picture.Bitmap.Canvas.Pixels[1,1]*. Попробую записать её немного по другому:

TImage1.Его_картинка.Bitmap.Холст.Пиксел[1_по_оси_X, 1_по оси_Y]

*На компакт диске, в директории **\Примеры\Глава 12\Image1** ты можешь увидеть пример этой программы.*

Обрати внимание, что если ты сейчас попытаешься открыть какой-нибудь не bmp файл, то программа ничего не отобразит. Это связано с тем, что только BMP файлы хранят свои изображения в свойстве *Bitmap*, все остальные хранят в свойстве *Graphic*. Так что получить доступ к JPEG изображению таким образом нельзя. Зато можно с помощью метода *Draw*. Для этого нужно подкорректировать обработчик события *OnPaint*:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Draw(200, 16, Image1.Picture.Graphic);
```

```
end;
```

Здесь я уже рисую не *Bitmap*, а *Graphic*, поэтому программа будет работать корректно.

Векторные файлы, такие как wmf хранят свои данные в свойстве *Metafile*. Для их отображения обработчик события *OnPaint* должен быть таким:

```
procedure TForm1.FormPaint(Sender: TObject);
begin
 Canvas.Draw(200, 16, Image1.Picture.Metafile);
end;
```

12.10 Рисование на стандартных компонентах

Очень часто, для лучшего представления данных, тебе будет нужно рисовать внутри компонента TListBox. Что я имею ввиду? Посмотри на рисунок 12.10.1, и ты всё поймёшь.

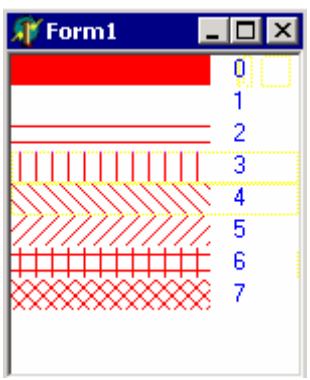


Рис 12.10.1. Пример

Может это покажется странным, но всё это делается за семь строчек кода. Конечно же, в одну строку можно записать и двадцать операций, но я этот случай не учитываю.

Секрет рисования заключается в том, что у компонента *TListBox1* свойство *Style* должен быть *lbOwnerDrawFixed* или *lbOwnerDrawVariable* .

После этого создаёшь обработчик события *OnDrawItem* для этого компонента и в нём пишешь:

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
 with ListBox1.Canvas do
  begin
   Brush.Color:=clRed; // Задаём красный цвет кисти.
   Brush.Style:=TBrushStyle(Index); //Выбираем стиль кисти
   Pen.Style:=psClear;
   Rectangle(Rect.Left,Rect.Top,Rect.Left+100,Rect.Bottom);
```

```
   Brush.Style:=bsClear;
   Font.Color:=clBlue;
   TextOut(Rect.Left+110,Rect.Top,IntToStr(index));
  end;
end;
```

Вот и всё. Твоя прога готова, жми на запуск и наслаждайся. Только давай посмотрим, что тут написано.

Первая строка:

with ListBox1.Canvas do

Оператор "With" говорит, что все последующие операции будут производится с компонентом (объектом) *ListBox1.Canvas*. Для того, чтобы ты лучше понял я приведу код без этой строки:

```
procedure TForm1.ListBox1DrawItem(Control: TWinControl; Index: Integer;
  Rect: TRect; State: TOwnerDrawState);
begin
  ListBox1.Canvas.Brush.Color:=clRed;
  ListBox1.Canvas.Brush.Style:=TBrushStyle(Index);
  ListBox1.Canvas.Pen.Style:=psClear;
  ListBox1.Canvas.Rectangle(Rect.Left,Rect.Top,Rect.Left+100,Rect.Bottom);

  ListBox1.Canvas.Brush.Style:=bsClear;
  ListBox1.Canvas.Font.Color:=clBlue;
  ListBox1.Canvas.TextOut(Rect.Left+110,Rect.Top,IntToStr(index));
end;
```

Как видишь, в каждой строке появились подписи *ListBox1.Canvas*. Код стал очень не красивым. Постоянно нужно говорить, что Brush или ещё что-нибудь нужно взять у *ListBox1.Canvas* .

Поэтому я и использовал оператор *With*

```
With "Объект" do
Begin
// Всё, что находится здесь, будет относиться к объекту "Объект".
// Поэтому не надо писать имя объекта перед каждым используемым
// Свойством или методом.
End;
```

Теперь ещё несколько подводных камней нашей проги. Конструкция *Brush.Style:=TBrushStyle(Index)* выбирает кисть в зависимости от рисуемого в данный момент элемента. Всего существует восемь стилей кисти. Когда вываливается сообщение *OnDrawItem* для первого элемента (об этом говорит параметр *index* передаваемый в процедуру *ListBox1DrawItem*), мы рисуем элемент с кистью первого стиля. Для второго элемента будет использоваться второй стиль кисти и т.д.

Карандаш я выбрал прозрачным *Pen.Style:=psClear*, это для того, чтобы не было никаких оборок. Попробуй убрать эту строку и посмотреть на результат.

Функция *Rectangle(x1,y2,x2,y2)* рисует прямоугольник с соответствующими координатами. Дальше я делал прозрачной кисть и задавал цвет фона. После этого я просто выводил текст строки с помощью функции *TextOut(x, y, текст)*.

Попробуй сделать тоже самое с компонентом *TComboBox*. Не забудь про свойство *Style* у этого компонента. А в остальном, весь код будет таким же.

На компакт диске, в директории ***\Примеры\Глава 12\ListBox*** *ты можешь увидеть пример этой программы.*

Ещё один пример рисования в стандартных компонентах, который я использую с целью обучения работы с графикой – рисование графических подсказок в строке состояния. Что я имел ввиду под выражением "графические подсказки"? Всё очень просто. Ты каждый день встречаешь в программах строку состояния внизу экрана, в которой выскакивают подсказки. Сегодня я покажу тебе, как сделать текст в компоненте *StatusBar* трёхмерным.

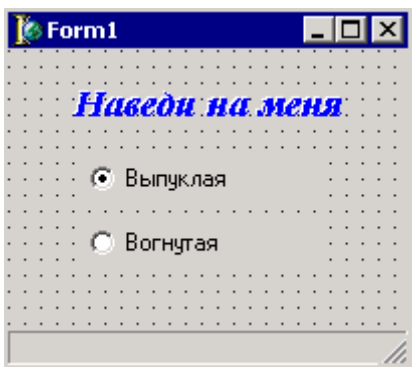


Рис 12.10.2. Форма будущей программы

На рисунке 12.10.2 показана форма, которая будет использоваться нами для вывода графической подсказки. Прежде чем мы приступим, я хочу напомнить, как вообще выводятся подсказки. Вот пример программы (точнее огрызок от программы), которая выводит подсказки:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 Application.OnHint := ShowHint;
end;

procedure TForm1.ShowHint(Sender: TObject);
begin
StatusBar1.SimpleText:=Application.Hint;
end;
```

Вспомним, что здесь происходит. В процедуре *FormCreate* (обработчик события *OnCreate* для главной формы), мы устанавливаем в качестве обработчика события *Application.OnHint* свою процедуру *ShowHint*. Теперь, когда будет происходить событие *OnHint* (т.е. когда нужно вывести подсказку), будет вызываться процедура *ShowHint*. В этой процедуре я просто вывожу подсказку в *StatusBar1*.

Теперь можно переходить к графической подсказке. Вот полный исходник нового вида функции *ShowHint*:

```
procedure TForm1.ShowHint(Sender: TObject);
var
 l,t:Integer;
begin
 StatusBar1.Repaint;
 with StatusBar1.Canvas do
  begin
   Brush.Style:=bsClear;
   Font.Color:=clWhite;
   l:=10;
   t:=1;
   TextOut(l,t,Application.Hint);
   if RadioButton1.Checked then
    begin
     inc(l);
     inc(t);
    end
   else
    begin
     dec(l);
     dec(t);
    end;
   Font.Color:=clBlue;
   TextOut(l,t,Application.Hint);
  end;
end;
```

Здесь ничего сложного нет. Я просто вывожу два раза текст подсказки с разным цветом и небольшим смещением. Это происходит точно так же, как и расположение двух компонентов *TLabel* на форме с небольшим смещением и разным цветом текста.

*На компакт диске, в директории **\Примеры\Глава 12\StatusBar** ты можешь увидеть пример этой программы.*

12.11 Работа с экраном.

Не знаю зачем, но очень часто меня спрашивают о том, как получить доступ к экрану. Такие люди хотят скопировать содержимое экрана в виде картинки и потом использовать это по своему усмотрению. Так как такой вопрос бывает не редко, то я решил добавить описание этого примера в книгу. В любом случае пример интересен и полезен в познавательных целях.

Для этого примера нам понадобиться форма с двумя кнопками и одной картинкой. Создай новый проект и поставь на него две пимпы *TButton* и один штука *TImage*. Для первой кнопки напишем в событии *OnClick*:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 ScreenDC:HDC;
begin
 ScreenDC := GetDC(0);
 Rectangle(ScreenDC, 10, 10, 200, 200);
 ReleaseDC(0,ScreenDC);
end;
```

С помощью этой процедуры я рисую прямо на экране вне области окна своей программы. Во время рисования, я не обращаю внимания на запущенные приложения. Если они попадаются под руку, то рисование происходит поверх них.

Теперь о содержимом. Я объявляю переменную *ScreenDC* типа *HDC*. *HDC* - это тип контекста рисования в *windows*, и работает почти так же, как и *TCanvas* (чуть позже ты увидишь связь). С помощью функции *GetDC(0)* я возвращаю контекст окна указанного в скобках. Но в этих скобках стоит 0 (ноль), а не указатель на реальное устройство или окно. Это значит, что мне нужен глобальный контекст, то есть самого экрана.

Далее, я вызываю функцию *Rectangle*, она похожа на ту, что мы использовали раньше *TCanvas.Rectangle*. Есть только одно отличие - первый параметр теперь, это контекст устройства, а затем идут координаты прямоугольника. Это связано с тем, что раньше мы рисовали через объект *TCanvas*, а сейчас будем рисовать средствами GDI Windows. Если честно, то процедура *TCanvas.Rectangle* всего лишь вызывает *Rectangle* из Windows API и подставляет нужный контекст устройства и размеры, поэтому в ней на один параметр меньше. Сейчас мы сами сделаем это, без помощь *TCanvas*.

После рисования, я освобождаю больше не нужный мне контекст через функцию *ReleaseDC*. Такие вещи обязательно надо освобождать, чтобы не засорять память.

Если ты захочешь рисовать не на экране, а внутри определённого окна, то в этой процедуре нужно поправить только первую строчку. А именно, в качестве параметра *GetDC* передавать указатель на окно. Указатель на наше окно находиться в свойстве *Handle* объекта *TForm*.

Сейчас можно запустить программу и посмотреть на результат, а я пока перейду ко второй кнопке. Для неё мы напишем следующий текст по событию *OnClick*:

```
procedure TForm1.Button2Click(Sender: TObject);
var
 Canvas:TCanvas;
 ScreenDC:HDC;
begin
 ScreenDC := GetDC(0);
 Canvas:=TCanvas.Create();
 Canvas.Handle:=ScreenDC;
 Image1.Canvas.Copyrect(Rect(0,0,Image1.Width,Image1.Height),
      Canvas, Rect(0,0,Screen.Width,Screen.Height));
 ReleaseDC(0,ScreenDC);
 Canvas.Free;
end;
```

Сразу скажу, что здесь я получаю копию экрана и сохраняю её в компоненте Image1.

Первая строка такая же, как и в предыдущей процедуре. Я точно также получаю контекст рисования экрана. Потом я создаю новую переменную *Canvas* типа *TCanvas* (знакомый нам контекст рисования). Потом я связываю их между собой с помощью простого присваивания в *Canvas.Handle:=ScreenDC*. Теперь мой *TCanvas* указывает на экран, и я могу рисовать на нём, привычными нам методами. Теперь видишь связь между холстом Canvas и контекстом рисования *HDC*. Объект холста всегда содержит указатель на контекст рисования *HDC* в свойстве *handle* и использует этот контекст при вызове всех своих методов (таких как *Rectangle*). Для компонентов Delphi это свойство заполняется автоматически и нам не надо о нём заботиться.

Далее, я получаю копию экрана и записываю её в картинку *TImage* с помощью функции *CopyRect* у контекста рисования картинки (*Image1.Canvas.CopyRect*).

После копирования я освобождаю контекст рисования *ScreenDC* и созданный холст *Canvas* чтобы освободить выделенную память.

 *На компакт диске, в директории* ***\Примеры\Глава 12\Screen*** *ты можешь увидеть пример этой программы.*

12.12 Режимы рисования.

У карандаша (свойство *Pen* холста) есть очень много режимов рисования. Благодаря им можно добиться очень интересных эффектов. Режимы рисования устанавливаются в свойстве *Mode* карандаша. Это свойство имеет тип *TPenMode* и может принимать следующие значения: *pmBlack, pmWhite, pmNop, pmNot, pmCopy* и так далее. Здесь я тебе покажу, как можно использовать это свойство в своих целях.

Я пишу не справочник, а книгу, по которой ты должен научиться программировать, познакомиться с некоторыми алгоритмами и научиться правильно мыслить. Именно поэтому я не буду расписывать все возможные варианты режимов, а опишу только один – *pmNotXor*, и пример его использования. Почему этот режим? Да потому что ему легко найти применения, а остальные мне ещё ни разу не понадобились за всю мою программистскую карьеру.

Итак, представим, что нам надо нарисовать пример, в котором пользователь должен иметь возможность рисования на форме прямоугольника. Кода в нашем примере будет немного и логика до предела проста:

1. По нажатию кнопки мыши мы запоминаем текущие координаты точки, где был произведён щелчок.

2. При движении мышки мы должны проверять: если кнопка мыши нажата, то пользователь растягивает прямоугольник и мы должны его нарисовать начиная от стартовой позиции до текущей.

Теперь реализуем это в программе. Создай новый проект и перейди в раздел **private** описания объекта. Там нужно добавить следующие переменные:

```
private
  { Private declarations }
  StartX, StartY:Integer;
  dragging:Boolean;
```

Переменные *StartX, StartY* будут использоваться для запоминания координат начала прямоугольника. Переменная *dragging* будет использоваться для признака растягивания. Если эта переменная равна *true*, то пользователь нажал кнопку мыши и перемещает кнопка мыши нажата и мы должны растягивать прямоугольник.

По событию *OnCreate* для формы мы должны задать переменной *dragging* значение по умолчанию – *false*, ведь после старта приложения мышь ничего не тянет и кнопки не нажата. Вот чтобы избежать случайного попадания в переменную значения *true* мы должны по этому событию написать следующий код:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 dragging:=false;
end;
```

По нажатию кнопки мыши *OnMouseDown* пишем следующий код:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 StartX:=X;
 StartY:=Y;

 dragging:=true;
end;
```

Сначала рассмотрим параметры, которые мы получили вместе с обработчиком. У нас тут пять параметров:

Sender – как всегда, здесь храниться объект, который сгенерировал событие.

Button – здесь храниться признак клавиши, которая была нажата. Этот параметр может быть равен: *mbLeft* (нажата левая кнопка), *mbRight* (нажата правая кнопка) или *mbMiddle* (нажата средняя кнопка).

Shift – состояние дополнительных клавиш клавиатуры. Это набор параметров типа *TShiftState*. Этот параметр может хранить любые из следующих значений ssShift - нажата клавиша *Shift*, *ssAlt* - нажата клавиша Alt, *ssCtrl* - нажата клавиша Ctrl, *ssLeft* - левая кнопка мыши нажата, *ssRight* - правая кнопка мыши нажата, *ssMiddle* – средняя кнопка нажата, *ssDouble* – был двойной щелчок мышкой.

Чтобы проверить, была ли нажата кнопка Shift во время нежетия мышкой можно написать следующий код:

```
if ssShift in Shift then
  ....
```

Почему этот параметр – это набор? Да потому что одновременно на клавиатуре может быть нажато две клавиши *Ctrl* и *Shift* и даже три клавиши.

X,Y – последние два параметра – это координаты, в которых была нажата кнопка мыши.

Внутри самой процедуры я сохраняю координаты в переменных *StartX* и *StartY*, и изменяю переменную *dragging* на *true*.

Теперь напишем обработчик события *OnMouseMove*, который генерируеться при каждом перемещении мышки:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
 if dragging=false then exit;

 Canvas.Rectangle(StartX, StartY, X, Y);
end;
```

Параметры этого обработчика похожи на обработчик события *OnMouseDown*.

В первой строчке происходит проверка, если переменная *dragging* равна *false*, то сейчас нет никакого перемещения и нужно выйти из процедуры. Иначе нужно нарисовать прямоугольник с координатами первой точки *StartX, StartY* и второй точки *X, Y*.

По событию *OnMouseUp* нужно присвоить переменной *dragging* значение *false*. Я думаю, что нет смысла показывать этот код, напиши его сам.

Теперь запусти программу. Попробуй щёлкнуть мышкой и потянуть её. Будет создаваться эффект, как будто ты растягиваешь прямоугольник. А теперь не отпуская мышку попробуй начать уменьшать прямоугольник. Вот тут начинается полный ужас рисунок 12.12.1. Пока мы растягивали прямоугольник, он спокойно накладывался сверху старого. Но как только мы начали уменьшать, новые прямоугольники становятся меньше старого и старый остаётся на экране, а новый оказывается как бы внутри.

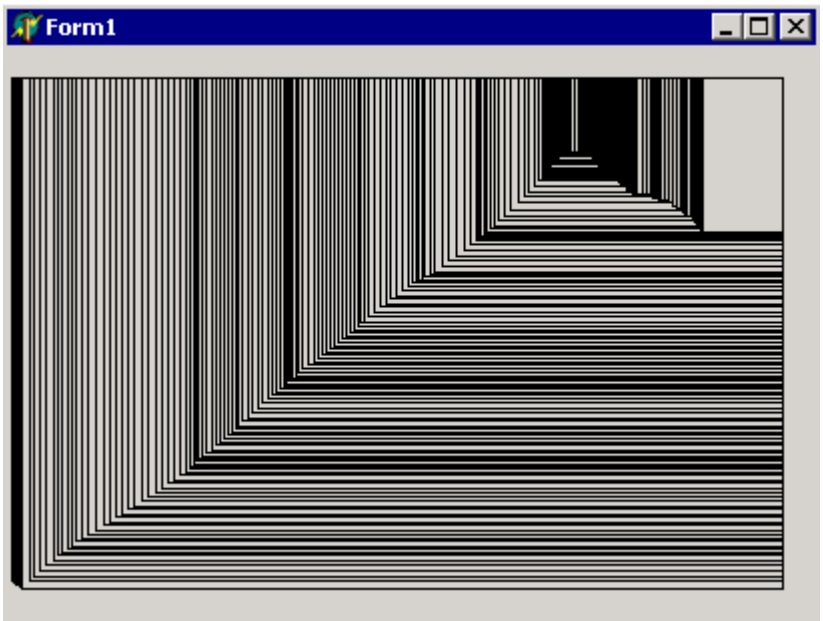


Рис 12.12.1. Пример работы программы

На компакт диске, в директории ***\Примеры\Глава 12\CopyMode1*** *ты можешь увидеть пример этой программы.*

Чтобы избавиться от этого эффекта есть два способа:

1. Перед каждым рисованием запоминать содержимое экрана и потом восстанавливать его. Такой способ связан с большими нагрузками на процессор и лишним расходованием памяти.
2. Затирать только старые прямоугольники и восстанавливать то, что было под линиями.

Все почему-то бояться использовать второй способ, думая, что он сложен. Сейчас я покажу тебе обратное. Мало того, что этот способ абсолютно прост в программировании, но и очень быстр.

Для начала добавим в раздел **private** ещё несколько переменных:

```
private
  { Private declarations }
  OldPenMode:TPenMode;
  StartX, StartY, OldX, OldY:Integer;
  dragging:Boolean;
```

Я добавил переменную *OldPenMode*, в которой будет сохраняться текущее значение режима рисования. Переменные *OldX*, *OldY* нужны для хранения конечных координат старого прямоугольника (начальные координаты *StartX* и *StartY*).

Теперь подправим обработчик события *OnMouseDown*:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
 Canvas.Brush.Color:=clWhite;
 OldPenMode:=Canvas.Pen.Mode;
 Canvas.Pen.Mode:=pmNotXor;

 StartX:=X;
 StartY:=Y;
 OldX:=X;
 OldY:=Y;

 dragging:=true;
end;
```

В самом начале я добавил изменение свойства кисти холста. Я сделал кисть белой, чтобы прямоугольники имели белый фон и ты лучше увидел эффект закраски.

Во второй строке я сохраняю текущий режим рисования в переменной *OldPenMode*. В следующей строке я меняю режим на *pmNotXor*. В таком режиме, когда мы рисуем первый раз какую-то фигуру, то она выводиться в нормальном виде. Если нарисовать второй раз прямоугольник, то он просто стирается и старое изображение восстанавливается на экране.

После этого я просто заполняю текущие координаты *StartX, StartY, OldX* и *OldY*.

Теперь посмотрим обработчик события *OnMouseMove*:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
 if dragging=false then exit;

 Canvas.Rectangle(StartX, StartY, OldX, OldY);
 Canvas.Rectangle(StartX, StartY, X, Y);
 OldX:=X;
 OldY:=Y;
end;
```

Начало обработчика такое же. Потом я рисую прямоугольник в старой позиции, где он был нарисован на прошлом шаге. Так как используется режим *pmNotXor*, то повторное рисования прямоугольника в старой позиции просто восстанавливает старое значение. После этого я рисую фигуру в новой позиции и сохраняю текущую позицию X и Y в переменных *OldX* и *OldY*.

И наконец обработчик события *OnMouseUp*:

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

```
begin
 dragging:=false;
 Canvas.Pen.Mode:=OldPenMode;
 Canvas.Rectangle(StartX, StartY, X, Y);
end;
```

В первой строке я присваиваю переменной *dragging* значение *false*. Во второй восстанавливаю старое значение режима рисования. И в самой последней строке я рисую уже окончательный вариант прямоугольника.

Запусти программу и попробуй нарисовать прямоугольник. Теперь, когда ты его рисуешь никаких накладок не происходит. Обрати внимание, что когда ты растягиваешь прямоугольник, то его фон прозрачный и только когда ты отпускаешь мышку (режим рисования восстанавливается) рисуется прямоугольник с фоном белого цвета.

Попробуй нарисовать сразу несколько прямоугольников на форме и убедиться, что всё работает нормально.

 *На компакт диске, в директории* ***\Примеры\Глава 12\CopyMode2*** *ты можешь увидеть пример этой программы.*

Глава 13. Печать в Delphi

Число 13 действительно не счастливое. Хотя я и не суеверный и не верю в приметы, но именно 13-ю главу я переделывал несколько раз. Сначала она должны была начать рассказ про базы данных. Потом я понял, что поспешил и сдвинул базы данных на более поздний этап, а 13-ю главу переделал в графику. Но не тут-то было. До баз данных надо ещё рассказать про печать в Windows. Пришлось опять немного корректировать план, и 13-я глава получила новое название – «Печать в Delphi».

Но и на этом цифра 13 не закончила свои издевательства. Когда я написал свой первый пример для книги и захотел его протестировать, то вспомнил, что на моём HP400 в картридже закончились чернила. Заправлять картридж уже не имело смысла, потому что он уже несколько раз испытал на себе эту процедуру, да и покупать новый тоже не выгодно (слишком дорого). Поэтому я давно задумал купить новый принтер, но никак не доходили до этого руки. После этого пришлось срочно бежать в магазин и покупать новый принтер, потому что доверять своим знаниям хорошо, а всё же проверять желательно любые примеры. Я человек, а значит мне свойственны ошибки даже в простых примерах (ну бывает не доглядел что-то).

Почему я говорю «Печать в Delphi», а не «Печать в Windows». Во-первых – потому что Delphi сильно упрощает не только сам принцип печати, но и доступ к её возможностям.

13.1 Объект TPrinter

Печать необходима везде и всегда. Ещё со времён первых компьютеров разработчики задумались об устройстве, которое могло бы выводить результаты расчётов на бумагу, чтобы не приходилось переписывать их ручками с монитора. Так и появился принтер, который очень сильно изменился за всё время своего существования, но не потерял актуальности.

Читать с монитора не всегда удобно, да и глаза устают от длительного чтения. Лично я всегда распечатываю всё необходимое для чтения на принтере, а потом изучаю в спокойной обстановке. Когда я путешествую в инете и вижу что-то интересное, то сразу отправляю на принтер. Так я экономлю не только время пребывания в сети, но и зрение.

Ну а офисные приложения вообще не возможно себе представить без возможночти печати. Возьмём те же программы как Word и Excel. Да грош им цена, если они не смогут печатать созданные в них документы. Любые базы данных (о них мы поговорим в следующей главе) должны уметь выводить данные на принтер, формировать отчётность, которая так же должна быть доступна для печати и т.д.

Изначально в Windows задумывался очень удобный и простой механизм вывода информации – контекст устройства. Мы уже познакомились с графикой и увидели, как выводить графику на контекст графического устройства – монитор. Когда нужно вывести информацию на экран, нужно получить контекст рисования монитора (или видеокарты, кому как удобнее) и рисовать на нём. Для удобства в Delphi есть объект TCanvas, который упрощает доступ к функциям отображения данных.

Так вот, функции вывода на контекст устройства универсальны. Им всё равно, куда выводить данные – будь это контекст монитора или принтера или ещё какого-нибудь устройства отображения информации. Я думаю, что пока ещё не совсем ясно, о чём я говорю, но сейчас всё встанет на свои места.

В Delphi есть объект (обрати сразу внимание, что это не компонент), который содержит все необходимые для доступа к принтеру свойства и методы – ***TPrinter***. Если посмотреть на эти свойства (чуть позже я опишу их), то сразу броситься в глаза свойство *Canvas* объектного типа ***TCanvas***.

Да, это тот же самый объект ***TCanvas***, который мы использовали при выводе графики. Я же говорил, что в объекте ***TCanvas*** собраны все необходимые функции для работы с графикой. Я так же сказал, что этим функциям абсолютно параллельно, куда выводить данные – на принтер или на экран монитора. Вот именно поэтому все компоненты способные отображать графику содержат в себе объектное свойство ***TCanvas*** и объект ***TPrinter*** способный выводить информацию на принтер тоже работает через него.

Раз у принтера есть такое свойство *Canvas*, то значит информацию, которую мы хотим вывести на печать надо выводить в виде графики. Даже тест выводиться именно как графика.

Немного теории закончено, теперь давай познакомимся с самим объектом ***TPrinter*** и посмотрим на его свойства и методы.

Свойства объекта ***TPrinter***:

Aborted – переменная типа *Boolean*. Если она равна true, то пользователь прекратил вывод информации на принтер.

Canvas – объект типа ***TCanvas***. Это холст, на который можно выводить информацию в графическом виде.

Copies – количество копий документа необходимых для печати.

Fonts – список шрифтов поддерживаемых принтером.

Handle – здесь храниться контекст принтера. Его ты можешь использовать, когда захочешь воспользоваться напрямую функциями WinAPI. Лично у меня такой надобности пока не было, потому что в объекте ***TPrinter*** уже есть всё необходимое.

Orientation – ориентация страницы. Это свойство может иметь одно из следующих значений: *poPortrait* – книжный или *poLandscape* – альбомный.

PageHeight – высота страницы в пикселях.

PageWidth – ширина страницы в пикселях.

PageNumber – номер печатаемой сейчас страницы.

PrinterIndex – число, которое указывает на номер выбранного сейчас принтера.

Printers – список типа ***TStrings*** установленных в системе принтеров.

Printing – если это свойство равно *true*, то принтер в данный момент печатает.

Title – заголовок или просто текстовое описание печатаемого документа. Этот заголовок будет отображаться во время печати в менеджере печати.

Со свойствами покончено, теперь давай разберёмся с методами объекта ***TPrinter***:

Abort – прерывает текущую печать.

BeginDoc – начало печатаемого документа.

EndDoc – конец документа.

GetPrinter – получить индекс текущего принтера.

NewPage – новая страница документа.

Refresh – обновить информацию о шрифтах и принтерах.

SetPrinter – установить текущий принтер.

Чтобы объект ***TPrinter*** стал доступным твоему проекту ты должен добавить в раздел **uses** модуль *Printers*. Объект ***TPrinter*** не надо инициализировать. Достаточно только подключить модуль и объект становится доступным через переменную *Printer*. Это ты увидишь в примерах этой главы.

Но прежде чем переходить к практике я должен сделать ещё несколько замечаний. Я говорил, что печать похожа на отображение информации на экране и на низком уровне используются одни и те же API функции. Я от своих слов не отказываюсь, но всё же прежде чем печатать, тебе нужно учитывать следующие особенности контекста печати:

1. Если с экрана можно стереть информацию, то изображение, выведенное на Canvas объекта принтера затереть уже не возможно после начала процесса печати.

2. Принтеры имеют большее разрешение, чем экран. Самые простейшие принтеры сейчас могут печатать графику с разрешением до 1200 точек на дюйм. Если ты попытаешься вывести картинку размером 200x200 на современный принтер, без каких либо корректировок, то пользователь этого изображения просто не увидит.

3. Не все принтеры одинаково работают с графикой, поэтому нужно давать пользователю возможность выбирать качество печати. Желательно всегда перед выводом на печать отображать стандартное окно настроек печати. Если на экран вывод производиться только один раз, то на принтере может понадобиться несколько копий. Это замечание не является обязательным, но советую учитывать это, чтобы твоя программа соответствовала признакам хорошего тона. А вдруг пользователю нужно 100 копий, так что ему теперь сто раз нажимать кнопку печати?

4. Ты должен давать возможность прервать печать в любой момент. Например, это может понадобиться, когда закончилась бумага, и её больше нет. Ты же не сможешь заблокировать работу программы, пока пользователь не сбегает в магазин. Конечно же такую возможность всегда предоставляет драйвер печати, но и ты не забывай о хорошем тоне.

13.2 Получение информации об установленном принтере.

Сейчас мы напишем программу, которая пока ещё не будет печатать, но зато она будет вытаскивать из системы информацию об установленных принтерах. Я сделаю её минимально простой, только чтобы ты понял основу работы с объектом ***TPrinter***.

Создай новый проект и сразу же допиши в разделе **uses** модуль *Printers*, чтобы объект ***TPrinter*** стал доступным проекту.

Моя программа будет показывать текущий выбранный принтер и список всех доступных в системе устройств печати. Для этого на форме понадобиться одна строка ввода ***TEdit*** для отображения текущего принтера и один список ***TListBox*** для отображения полного списка. У строки ввода можно установить свойство *ReadOnly* в *true*, потому что информация из этой строки только для отображения и не должна редактироваться, да и смысла нет её редактировать.

На рисунке 13.2.1 показана форма моей будущей программы. Ты можешь расположить компоненты по-другому, но я предпочитаю именно такое расположение. Главное делай подробные подписи для выводимой информации, иначе программа становиться понятной только тебе, и больше уже никому не понадобиться.

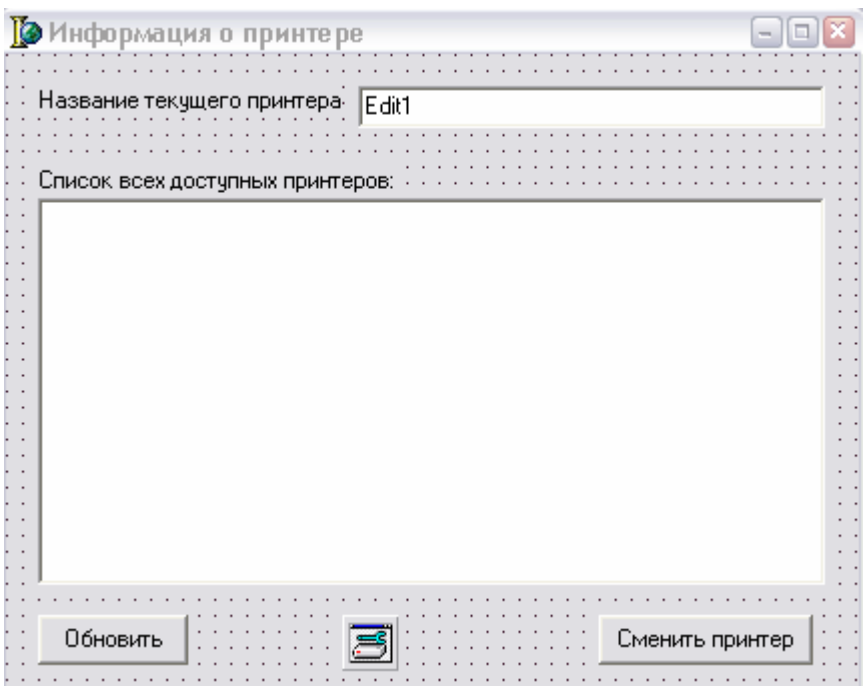


Рисунок 13.2.1 Форма будущей программы

Теперь брось на форму две кнопки: «*Обновить*» (для обновления информации о принтерах) и «*Сменить принтер*» (понятно зачем).

Нам так же понадобиться компонент *PrinterSetupDialog* с закладки *Dialogs* палитры компонентов. Этот компонент предназначен для отображения стандартного окна настроек принтера.

По нажатию кнопки «*Обновить*» пишем следующий код:

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

```
 i:Integer;
begin
 ListBox1.Items.Clear;
 for i:=0 to Printer.Printers.Count-1 do
  ListBox1.Items.Add(Printer.Printers.Strings[i]);

 Edit1.Text:= Printer.Printers.Strings[Printer.PrinterIndex];
end;
```

В первой строке кода я очищаю текущее содержимое списка *ListBox1*. Потом запускаю цикл от 0 до количества установленных в системе принтеров минус 1 (количество принтеров находиться в *Printer.Printers.Count*). Свойство *Printers* объекта ***Tprinters*** имеет тип ***TStrings***, как свойство *Items* компонента ***TListBox***, поэтому их свойства и методы одинаковы и мы с ними уже не раз работали.

Почему я отнимаю единицу? Да потому что цикл начинаю с нуля. Допустим, что у тебя установлено 2 принтера и в переменной *Printer.Printers.Count* будет находиться 2. В этом случае цикл будет выполняться от 0 до 2, т.е. три раза для 0, 1 и 2. Но принтеров только 2, поэтому нужно отнять единицу.

Внутри цикла я добавляю в список названия принтеров. Названия хранятся в переменной *Printer.Printers.Strings[i]*, где i – это индекс принтера, изменяющийся от 0 до значения из *Printer.Printers.Count* минус 1.

После этого я вывожу в строку ввода название текущего принтера. Индекс текущего принтера – это *Printer.PrinterIndex* значит название текущего принтера можно получить так: *Printer.Printers.Strings[Printer.PrinterIndex]*.

Эту же процедуру можно вызвать по событию *OnCreate* или *OnShow* главной формы, чтобы после старта программа сразу же получала необходимую информацию:

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

Нам осталось только написать код для кнопки «*Сменить принтер*» и программа готова. Итак, по нажатию этой кнопки пишем коротко и ясно:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 PrinterSetupDialog1.Execute;
 Button1Click(nil);
end;
```

В первой строке я просто отображаю окно *PrinterSetupDialog1* (окно настроек принтера). Во второй строке я вызываю процедуру, которую мы написали для обновления информации о принтерах.

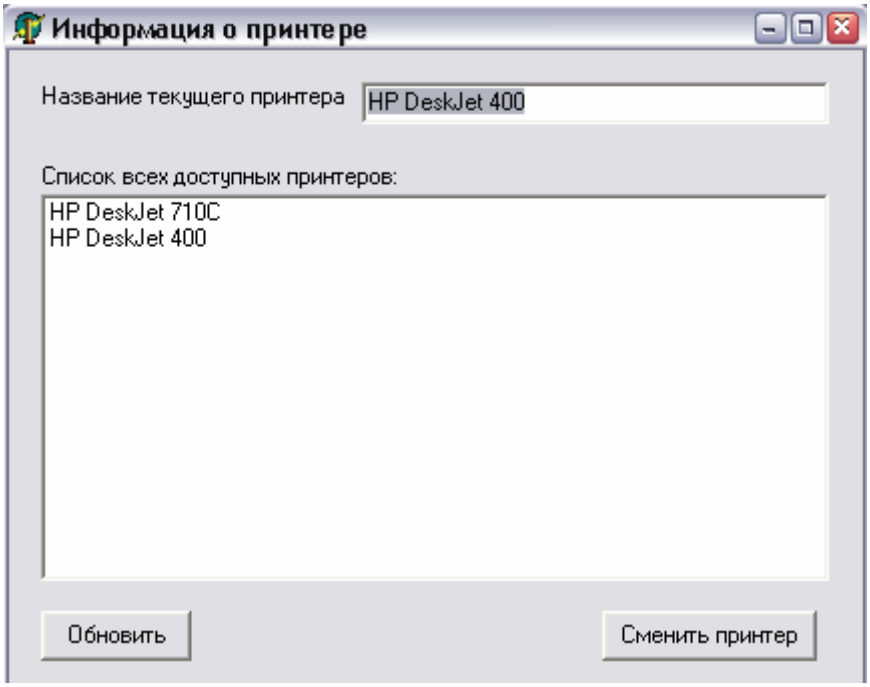


Рисунок 13.2.2 Результат работы программы

Теперь запусти программу и посмотри на результат. Если у тебя в системе только один принтер (ну нет у тебя больше принтеров) или вообще нет ни одного, то зайди в «Панель управления» Windows, выбери там «Принтеры» и установи пару любых принтеров вручную. Система от этого работать хуже не будет, зато ты сможешь полноценно протестировать этот пример.

Нажми на кнопку сменить принтер и перед тобой откроется окно похожее на рисунок 13.2.3. В этом окне, в выпадающем списке «*Имя*» измени имя текущего принтера на другое и закрой окно кнопкой «ОК». Обрати внимание, что в твоей программе, в строке «*Название текущего принтера*» название автоматически изменилось. Это потому что мы после отображения окна сразу перечитали свойства принтеров. А они изменились, потому что окно *PrinterSetupDialog1* тесно связано с системой и автоматически обновляет всю информацию в объекте ***TPrinter***. Поэтому нам не надо самостоятельно читать данные, изменённые в окне свойств принтера, и переносить в объект печати.

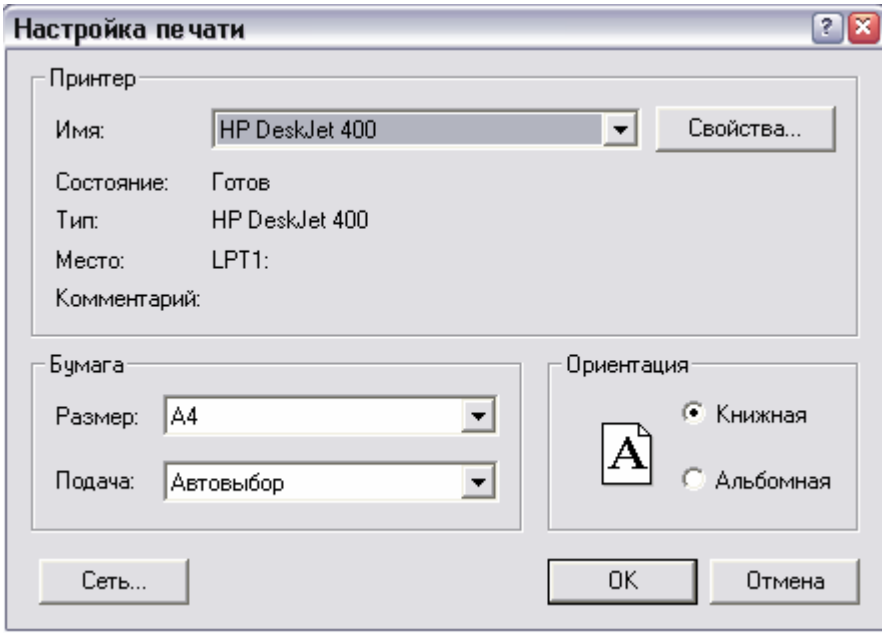


Рисунок 13.2.3 Окно свойств принтера

Советую так же обратить внимание на то, что *Printer* – это объект типа ***TPrinter***, а я всегда говорил, что все объекты должны создаваться с помощью их же метода *Create* и уничтожаться с помощью метода *Free*. В данном случае этого делать не надо, всё делается автоматически. Именно поэтому я даже не показывал тебе метода *Create* и *Free*, когда расписывал объект ***TPrinter***. Вот это тебе обязательно надо помнить!!!

*На компакт диске, в директории **\Примеры\Глава 13\Свойства принтера** ты можешь увидеть пример этой программы.*

13.3 Текстовая печать.

Хотя нынешние принтеры и графические, но вывод на печать в виде текста ещё возможен. Представь себе, как сложно было бы жить, если бы нельзя было вывести содержимое компонента ***TMemo***, как текст. В этом случае нам пришлось бы рисовать каждую строку в контексте рисования принтера (рисовать на *Canvas*-е с помощью его метода TextOut) при этом учитывая расстояния между строками.

Но есть ещё бог на свете и нам не надо так сильно мучиться. С принтером можно работать как с простым текстовым файлом.

Для открытия принтера в текстовом режиме используется процедура *AssignPrn*. В качестве единственного параметра этой процедуре надо передать переменную типа **TextFile**. После этого переменной назначен принтер по умолчанию. Дальше его нужно открыть с помощью процедуры *Rewrite*.

Как только файл открыт, в него можно печатать с помощью процедуры *writeln*, у которой два параметра:

1. Переменная типа **TextFile**, которой назначен принтер.
2. Текст, который надо распечатать.

После печати переменную надо освободить (закрыть файл, ассоциированный с принтером) с помощью процедуры CloseFile.

Вот простейший пример вывода текста «Hello world» в виде строки на принтер:

```
var
 f:TextFile;
begin
 AssignPrn(f);

 try
  Rewrite(f);
  Writeln(f, 'Hello world');
 finally
  CloseFile(f);
 end;
end;
```

В первой строке кода я назначаю переменной *f* принтер. После этого открытие файла ассоциированного с принтером и вывод на печать я заключаю между *try* и *finally*. Это необходимо, потому что если после выполнения процедуры *AssignPrn* переменной *f* не будет назначен принтер (ну нету его в системе, не установлен или вообще отсутствует), то при попытке открыть файл или начать печать произойдёт ошибка.

Между *finally* и *end* (код написанный здесь будет выполняться всегда, вне зависимости от того, была ошибка или нет) я закрываю открытый файл. Если бы я не ставил *try…finally..end*, а во время печати произошла ошибка, то файл, ассоциированный с принтером, остался бы открытым. А это значит, что последующая, нормальная работа принтера уже не гарантируется.

Вот ещё пример вывода на притер содержимого компонента ***TMemo***.

```
var
 f:TextFile;
 i:Integer;
begin
 AssignPrn(f);

 try
  Rewrite(f);
  for i:=0 to Memo1.Lines.Count-1 do
   Writeln(f, Memo1.Lines.Strings[i]);
 finally
  CloseFile(f);
 end;
end;
```

Попробуй сам разобраться, как работает этот пример. Для этого тебе нужно вспомнить свойства компонента ***TMemo*** и как работать с ним.

13.4 Печать содержимого формы.

Мы научились получать информацию о принтере и печатать в текст, теперь пора научиться выводить на печать графику. Хотя сейчас я покажу простейший пример печати, зато в нём будет всё необходимое тебе в будущем при построении сложных сцен.

Этот пример я взял из файла помощи Delphi и просто преподнес тебе в готовом виде. Это достаточно хороший пример, на котором можно научиться основам печати, а уже следующий пример, мы напишем более сложным с более качественной печатью.

Для примера нам понадобиться форма, на которой будет расположен один компонент ***TPageControl***. На нём можно создать несколько закладок (я создал две) и поместить на закладки различные компоненты. Я на первую закладку бросил текст и несколько компонентов ***TShape***. На второй закладке я расположил картинку ***TImage***. На печать я буду выводить содержимое закладок компонента ***TPageControl*** вместе с компонентами и картинкой. Каждая закладка будет печататься как отдельная страница.

Ну и напоследок брось на форму кнопку «Печать» и компонент *PrintDialog* с закладки *Dialogs* палитры компонентов. Второй компонент предназначен для отображения стандартного окна запуска печати (рисунок 14.4.2). Окно печати похоже на окно настроек печати *PrinterSetupDialog*, но имеет свои отличия. Именно такое окно ты видишь каждый раз, когда запускаешь печать в других программах, таких как MS Word, Excel и др.

Вид моей формы ты можешь увидеть на рисунке 14.4.1. Можешь сделать такую же, а можешь попробовать что-нибудь своё.

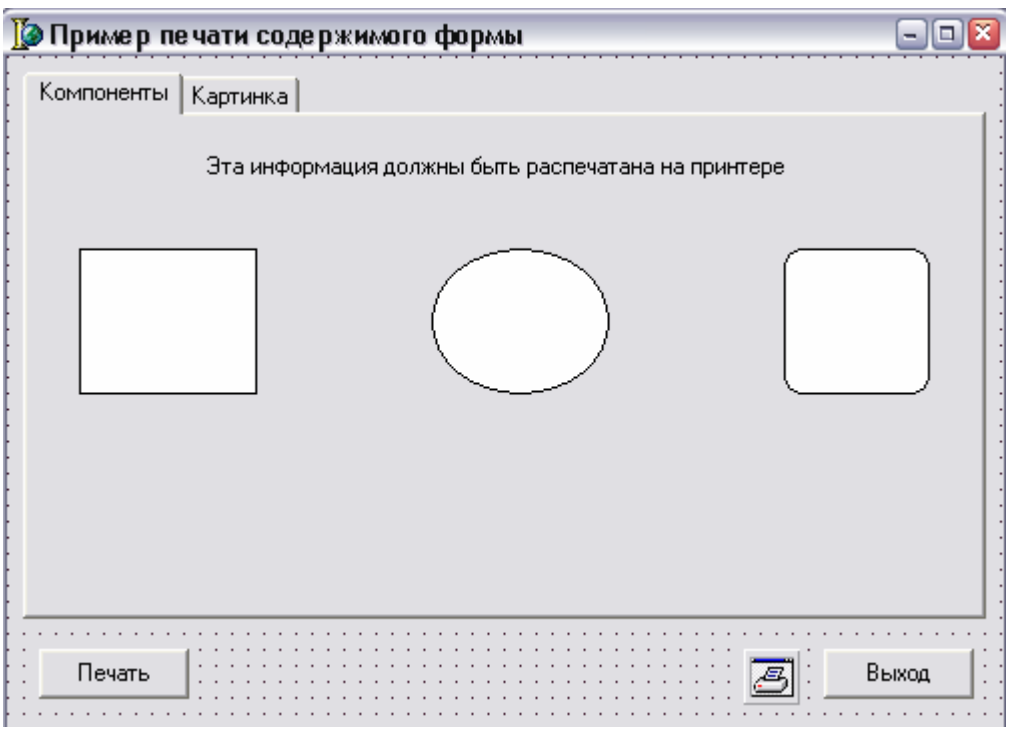


Рисунок 13.4.1 Форма будущей программы

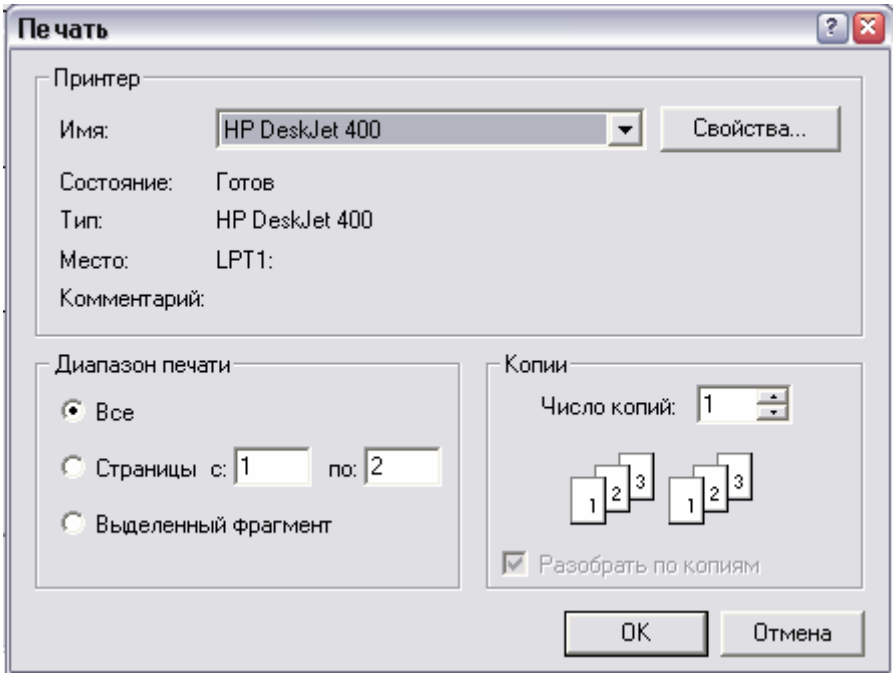


Рисунок 13.4.2 Окно запуска печати

Теперь перейдём к написанию кода. Создай обработчик события *OnClick* для кнопки печать и напиши в нём следующее:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i, Start, Stop: Integer;
begin
 PrintDialog1.Options := [poPageNums, poSelection];
 PrintDialog1.FromPage := 1;
 PrintDialog1.ToPage := PageControl1.PageCount;
 PrintDialog1.MinPage := 1;
 PrintDialog1.MaxPage := PageControl1.PageCount;
 if not PrintDialog1.Execute then exit;

 if PrintDialog1.PrintRange = prAllPages then
  begin
   Start := PrintDialog1.MinPage - 1;
   Stop := PrintDialog1.MaxPage - 1;
  end
 else //Если выбрано отличное от печати «Всё»
  if PrintDialog1.PrintRange = prSelection then
   begin
    Start := PageControl1.ActivePageIndex;
    Stop := Start;
   end
  else /Если выбрано отличное от «Выделенный фрагмент»
   begin
    Start := PrintDialog1.FromPage - 1;
    Stop := PrintDialog1.ToPage - 1;
   end;

 //Начало печати
 Printer.BeginDoc;
 for i := Start to Stop do
  begin
```

```
  PageControl1.Pages[i].PaintTo(Printer.Handle, 10, 10);
  if i <> Stop then
   Printer.NewPage;
 end;
 Printer.EndDoc;
end;
```

В первой строке кода я задаю опции окна печати *PrintDialog1.Options*. В этом свойстве храниться информация о том, что должно отображаться в окне «Печать». Возможны следующие значения:

poDisablePrintToFile – отключить возможность выбора печати в файл. Если ты добавишь в опции это значение, то пользователь не сможет выбрать «Печать в файл».

poHelp – показывать кнопку «Помощь» в окне печати.

poPageNums – разрешить пользователю выбирать диапазон печати (какие страницы нужно распечатать).

poPrintToFile – показывать в окне печати CheckBox, в котором можно выбрать печать в файл.

poSelection – давать возможность пользователю выбрать в окне печати печать выделенного фрагмента.

poWarning – показывать пользователю сообщения, если он пытается запустить работу на не установленный принтер.

Я в раздел настроек добавляю две опции *poPageNums* и *poSelection*:

```
PrintDialog1.Options := [poPageNums, poSelection]
```

Всё это можно было бы сделать проще: просто выделить компонент *PrintDialog1*, и в объектном инспекторе дважды щёлкнуть по свойству *Options*. После этого откроется список из всех этих свойств, где можно указать напротив необходимых свойств значение *true*. Я бы поступил именно так, но я же говорил, что пример взят из файла помощи по Delphi, а там настройки делали программно.

Во второй строке кода я присваиваю свойству *FromPage* компонента *PrintDialog1* начальное значение, с которого должна происходить печать. А в следующей строке я устанавливаю в свойстве *ToPage* количество печатаемых страниц. В это свойство я заношу количество закладок у моего компонента *PageControl1*. Мы же договорились, что каждая закладка будет печататься на отдельной странице, значит, сколько закладок, столько и страниц будет печататься.

После этого я устанавливаю минимальное и максимальное значение страниц доступных для печати:

```
PrintDialog1.MinPage := 1;
PrintDialog1.MaxPage := PageControl1.PageCount;
```

Все, настройки произведены, можно отображать окно печати:

```
if not PrintDialog1.Execute then exit;
```

Здесь используется конструкция *if not*, значит, если пользователь закроет окно через кнопку *Cancel*, то выполниться код написанный после *then*. А там у меня написан выход из процедуры – *exit*. Так что если не будет нажата в окне печати кнопка «*OK*», то процедура не будет выполняться дальше и печать не произойдёт. Ну а если нажат «*OK*», то код процедуры продолжит своё выполнение.

А дальше у меня идёт проверка, какой диапазон печати был выбран пользователем:

```
if PrintDialog1.PrintRange = prAllPages then
```

Этот код проверяет, если выбран диапазон печати всех страниц, то переменным *Start* и *Stop* будут присвоены значения минимального значения страниц и максимального значения соответственно, чтобы были распечатаны все закладки нашего компонента.

Если пользователь выбрал не все страницы, то нужно проверить, а может, он выбрал печать выделенного фрагмента?

```
if PrintDialog1.PrintRange = prSelection then
```

Если пользователь выбрал печать выделенного фрагмента, то переменным *Start* и *Stop* будет присвоено одно и то же - номер выделенной в данный момент закладки *PageControl1.ActivePageIndex*.

Ну и если пользователь не выбрал ни того, ни другого, значит, он выбрал начальную и конечную страницу, которые надо распечатать. В этом случае переменным *Start* и *Stop*, будут назначены значения выделенных пользователем страниц:

```
Start := PrintDialog1.FromPage - 1;
Stop := PrintDialog1.ToPage - 1;
```

Обрати внимания, что я от выделенных пользователем страниц вычитаю единицу. Это потому что пользователь, как нормальный человек будет нумеровать страницы, начиная с единицы, а закладки нашего компонента *PageControl1* нумеруются с нуля.

Всё, теперь наши переменные *Start* и *Stop* содержат диапазон, который надо распечатать в зависимости от выбранных пользователем настроек и можно переходить к самой печати. Для начала распечатки нужно начать новый документ. Для этого нужно вызвать метод *BeginDoc* объекта ***TPrinter***.

После этого запускаем цикл от стартовой страницы, до, последней выделенной:

```
for i := Start to Stop do
```

Внутри цикла выполняется следующий код:

```
PageControl1.Pages[i].PaintTo(Printer.Handle, 10, 10);
if i <> Stop then
 Printer.NewPage;
```

В первой строке я заставляю прорисоваться очередную страницу на определённое устройство. Об этом говорит конструкция *PageControl1.Pages[i].PaintTo*. Метод *PaintTo* заставляет прорисоваться i-ю страницу на указанное устройство. Нужное устройство указывается в качестве первого параметра метода (здесь я указываю принтер). Остальные два параметра указывают отступ слева и сверху.

Далее я проверяю, если i не равна последней печатаемой страницы, то создать новую страницу, на которой будет распечатана следующая закладка. Если не производить этой проверки, то когда распечатается последняя страницы, то будет создана новая, которая вылезет из принтера пустой. Это не ошибка, но будет не приятно, если из принтера в конце печати будет выползать пустой лист бумаги.

Самым последним методом вызывается *EndDoc* объекта ***TPrinter*** после чего принтер начинает печатать весь документ.

На компакт диске, в директории ***\Примеры\Глава 13\Печать содержимого формы*** *ты можешь увидеть пример этой программы.*

13.5 Вывод на печать изображения.

В принципе, уже в прошлом примере я напечатал изображение, потому что я печатал закладки компонента ***TPageControl*** как самые настоящие картинки. Но пример был простой и не учитывал, что разрешение принтера выше, чем у монитора. Поэтому, если ты пытался распечатать примеры, то изображения получались слишком маленькими. В этой части я попытаюсь показать, как учитывать разрешение принтера.

Но сначала, давай взглянём на простейший пример вывода изображения на принтер:

```
begin
 Printer.BeginDoc;
 Printer.Canvas.Draw(10, 10, Image1.Picture.Bitmap);
 Printer.EndDoc;
end;
```

Этот пример не отображает никаких диалогов, а просто начинает новый документ на принтере, потом копирует в его *Canvas* картинку из компонента *Image1* и заканчивает документ.

Это ещё один простейший пример, который не учитывает разрешение принтера. Он прост, но в реальных условиях не применим, потому что никому не нужно изображение, которое на бумаге меньше того, что видно на мониторе.

Теперь пора нам узнать, как определить разрешение принтера. Для этого используется WinAPI функция *GetDeviceCaps*, которая предназначена для получения информации о нужном устройстве. У этой функции два параметра:

1. Устройство, информацию которого мы хотим получить. Нам нужна информация о размере *Canvas* принтера, поэтому нужно будет передавать его указатель *Printer.Canvas.Handle*.

2. Какую именно информацию нам надо. Нам нужно количество пикселей по оси X и Y, поэтому будем указывать **LOGPIXELSX** (разрешение по оси X) и **LOGPIXELSY** (разрешение по оси Y).

Теперь реальный пример с использованием этой функции. Создай новый проект и брось на форму один компонент ***TImage***, который будет хранить картинку для печати (сразу же загрузи туда любое изображение в формате bmp) и кнопка «*Печать*». На рисунке 13.5.1 ты можешь видеть форму моей будущей программы.

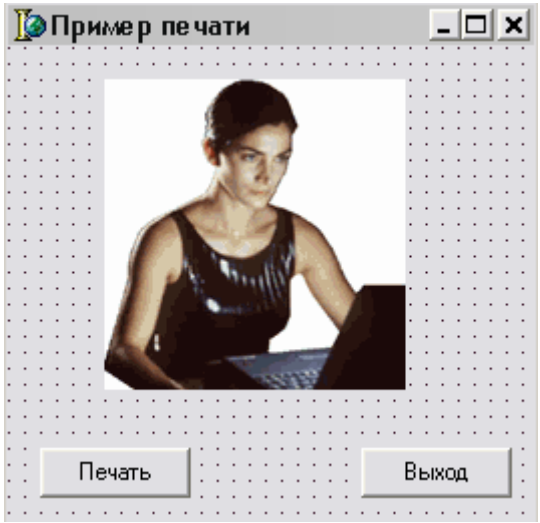


Рисунок 13.5.1. Форма будущей программы

Теперь создай обработчик события *OnClick* для кнопки и напиши там следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 X1,X2,Y1,Y2:Integer;
 PointsX,PointsY:double;
 PrintDlg:TPrintDialog;
begin
 // Создаю и отображаю на экране стандартное окно печати
 PrintDlg:=TPrintDialog.Create(Owner);
 if PrintDlg.Execute then
  begin
  //Начинаю новый документ
   Printer.BeginDoc;
   Printer.Canvas.Refresh;

  //Получаю информацию о разрешении принтера
   PointsX:=GetDeviceCaps(Printer.Canvas.Handle,LOGPIXELSX)/70;
   PointsY:=GetDeviceCaps(Printer.Canvas.Handle,LOGPIXELSY)/70;

  //Расчитываю размеры изображения
   X1:=round((Printer.PageWidth - Image1.Picture.Bitmap.Width*PointsX)/2);
   Y1:=round((Printer.PageHeight - Image1.Picture.Bitmap.Height*PointsY)/2);
   X2:=round(X1+Image1.Picture.Bitmap.Width*PointsX);
   Y2:=round(Y1+Image1.Picture.Bitmap.Height*PointsY);
  //Вывод изображения на печать
   Printer.Canvas.CopyRect(Rect(X1,Y1,X2,Y2),Image1.Picture.Bitmap.Canvas,
         Rect(0,0,Image1.Picture.Bitmap.Width,Image1.Picture.Bitmap.Height));
   Printer.EndDoc;
  end;
 //Уничтожаю созданное окно печати
 PrintDlg.Free;
end;
```

В самом начале я программно создаю компонент *PrintDialog* (стандартное окно печати), потому что на форму я его не бросал. Я сделал это специально, чтобы лишний раз показать тебе, как программно создаются и используются компоненты Delphi. Для

создания я выполняю код *TPrintDialog.Create(Owner)*, который выделит память под объект и возвратит мне ссылку на него. Эту ссылку я сохраняю в переменной *PrintDlg*, которая объявлена у меня в разделе **var** в виде переменной типа ***TPrintDialog***.

После этого я показываю окно печати (*if PrintDlg.Execute then*) и если пользователь нажал «*OK*», то выполниться код между последующими *begin…end*.

Тут с первой строкой кода уже понятно – начало нового документа. После этого я обновляю всю информацию на холсте принтера (Printer.Canvas.Refresh).

Дальше я получаю информацию о разрешении принтера по вертикали и горизонтали с помощью функции *GetDeviceCaps*. Результат делиться на 70 (просто коэффициент масштабирования, можешь подобрать любой подходящий тебе) и сохраняю в переменных *PointsX* и *PointsY*.

После этого идёт расчёт координат картинки. Я её постарался вывести по центру листа бумаги. Отступ слева я вычисляю по такой формуле: (ширина листа принтера – ширина картинки * PointsX)/2. Отступ сверху вычисляю по следующей формуле (высота листа принтера – высота картинки * PointsY)/2.

Отступы слева и сверху рассчитаны. Теперь нужно найти правую и нижнюю сторону изображения. Для этого я просто прибавляю к левой стороне ширину изображения (*Image1.Picture.Bitmap.Width*PointsX*) и к верхней стороне высоту изображения (*Image1.Picture.Bitmap.Height*PointsY*).

Теперь можно выводить картинку на холст принтера. Для этого я копирую изображение из *Image1* на холст принтера с помощью процедуры *Printer.Canvas.CopyRect*. Мы уже пользовались этой процедурой при работе с графикой и ты должен знать, что она умеет масштабировать копируемую картинку.

Всё, можно заканчивать документ (*Printer.EndDoc*) и уничтожать созданное окно (*PrintDlg.Free*). Хотя окно создано как локальное, потому объявлено внутри процедуры, и должно уничтожаться автоматически, я всё же делаю это, не надеясь на компилятор. Не устану напоминать, что все переменные объявленные как локальные (в разделе **var** внутри процедуры) инициализируются в стеке и автоматически уничтожаются сразу после выхода из процедуры. Но всё же те переменные, которым ты выделял память или создавал, как объекты нужно уничтожать самостоятельно, не надеясь на чистку стека. Ведь в стеке храниться только ссылка на объект или выделенную память, а сама память может быть выделена где угодно, но только не в этом стеке.

 *На компакт диске, в директории* ***\Примеры\Глава 13\Печать картинки*** *ты можешь увидеть пример этой программы.*

Глава 14. Delphi и базы данных

Базы данных считаются основным козырем Delphi. Это действительно так. Хотя этот язык не создавался специально под эту сферу, но реализация работы с данными здесь просто поражает. Даже специализированные языки для работы с базами данных (такие, как MS Visual FoxPro) явно уступают по простоте и мощи программирования этого типа приложений.

Delphi скрывает все сложности и в то же время даёт тебе величайшую мощь. Ещё не было такой задачи, которую я не смог бы реализовать на Delphi за короткий промежуток времени. А главное, что всё это реализовано очень удобно и легко для понимания.

Когда я первый раз услышал про базы данных, я сильно испугался. Мне казалось это очень сложным. Но когда я увидел, что в Delphi можно создавать простые приложения, но даже со сложными базами без единой строчки кода, я просто влюбился в эту среду разработки. В этой главе мы познакомимся с основами баз данных и напишем несколько полезных примеров.

Для примеров я буду использовать базы Access и современный формат xml. Я советую использовать эти базы в качестве локальных, потому что они поддерживаются большинством систем и отличаются высокой надёжностью.

В последствии я покажу тебе самые простые и распространённые базы dbf и paradox. Лично я их стараюсь не использовать в своих проектах из-за их ненадёжности, и потому что в них регулярно нарушается индексная целостность, что приводит к неработоспособности программ. Но из-за их распространённости, знать принципы работы с ними просто необходимо. Даже локальная версия 1С Предприятия использует этот ужасный формат DBF. Так что если ты захочешь написать свою программу для работы с чужими данными, то ты просто обязан знать, как работать с этим форматом данных.

14.1 Теория реляционных баз данных

Ещё десять лет назад, программирование баз данных было очень сложным занятием. За какие-либо достижения в этой области многие программисты получили в своё время докторские степени. Сейчас уже такое трудно себе представить, потому что благодаря Delphi, процесс написания программ упростился, а количество разновидностей баз данных уже исчисляется десятками.

Ключ	Фамилия	Имя	Отчество
1	Иванов	Иван	Петрович
2	Петров	Сергей	Владимирович
3	Сидоров	Алексей	Викторович
4	Смирнов	Андрей	Сергеевич

Таблица 14.1 Пример простейшей базы данных

Базы данных делятся на локальные (установленные на компьютере клиента, там же где и работает программа) и удалённые (установленные на сервере, удалённом компьютере). Серверные базы данных располагаются на удалённом компьютере и работают под управлением серверного программного обеспечения. К их главным преимуществам можно отнести возможность работы с одной базой данных одновременно несколькими пользователями, и при этом осуществляется минимальная нагрузка на сеть.

Есть ещё сетевые базы данных, но их мы рассматривать не будем, потому что они создают слишком большую нагрузку на сеть и неудобны в работе, как для программиста, так и для конечного пользователя. Поэтому работать с такими базами мы не будем, и я никому не рекомендую этого делать. Почему? Это я попытаюсь тебе сейчас объяснить.

Когда твоя программа присоединяется к сетевой базе данных, то она выкачивает с сервера практически полную его копию. Если ты внёс изменения, то твоя копия полностью закачивается обратно. Это очень неудобно, потому что создаётся большая нагрузка на сеть из-за излишней перекачки данных.

При клиент-серверной технологии программа клиент посылает простой текстовый запрос на сервер на получение каких-либо данных. Сервер обрабатывает его и возвращает только необходимую порцию данных. Когда нужно изменить какие-то данные, опять посылается запрос к серверу на их изменение и сервер изменяет данные в своей базе. Таким образом, по сети происходит перекачка в основном только текстовых запросов, которые в основном занимают меньше килобайта. Все данные обрабатывает сервер, а значит, машина клиента загружается намного меньше и не так сильно требовательна к ресурсам. Сервер отсылает клиенту только самые необходимые данные, а значит, отсутствует излишняя перекачка копии все базы.

Благодаря всему этому, сетевые базы данных уже устарели и практически не используются. Их практически полностью вытесняет технология клиент-сревер. А вот локальные базы данных будут жить всегда. Может измениться формат их хранения или добавиться какие-то новые функции, но сами базы данных будут существовать.

В этой главе мы затронем только локальные базы данных, а серверные рассмотрим немного позже. Для дальнейшего рассмотрения нам надо определить новое понятие – *таблица*. Пока что я говорил только общие принципы, поэтому использовал общее понятие *баз данных*. Таблица базы данных – это как двух мерный массив, в котором в столбец выстроены данные (яркий пример таблицы – Excel). База данных – грубо говоря это всего лишь файл, в котором может храниться от одной до нескольких таблиц. Большинство локальных баз данных могут хранить только одну таблицу (dBase, Paradox, XML). Но есть представители локальных баз, где в одном файле заключено несколько таблиц (например Access, который мы будем рассматривать в этой главе).

Локальные базы данных

Из локальных баз данных мы будем рассматривать реляционные, как самые распространённые. Что такое реляционная база данных? Это таблица, в которой в качестве столбцов выступают имена хранимых в ней данных, а каждая строка хранит сами данные. Таблица базы данных базы данных похожа на электронную таблицу Excel (если быть точнее, то Excel хранит свои данные в виде собственного формата, построенного на основе технологии баз данных). Локальные таблицы баз данных могут храниться на локальном жёстком диске или централизовано сохраняться на сетевой диск файлового сервера. Эти файлы можно копировать с помощью стандартных средств как любой другой файл, потому что сами таблицы базы данных не привязаны к определённому месту расположения. Главное, чтобы программа могла найти твою таблицу.

В каждой таблице должно быть одно уникальное поле, которое однозначно будет идентифицировать строку. Это поле называется ключевым. Эти поля очень часто используются для связывания нескольких таблиц между собой (с этим мы ещё познакомимся). Но даже если у тебя таблица не связана, ключевое поле всё равно обязательно. Представь, что ты пишешь телефонную базу данных. Сколько у тебя будет "Ивановых"? Как ты будешь отличать их? Вот тут тебе поможет ключ. В качестве ключа желательно использовать численный тип и если позволяет база данных, то будет лучше, если он будет типа "autoincrement" (автоматически увеличивающееся/уменьшающееся число или счётчик).

Имена столбцов в таблице базе данных, также должны быть уникальными, но в этом случае не обязательно числовыми. Их можно называть как угодно, лишь бы было уникально и тебе понятно, а остальное никого не касается.

Каждый столбец (поле базы данных) обязательно должен иметь определённый тип. Количество типов и их разновидности зависит от типа базы данных, например формат dBASE (файлы с расширением DBF) поддерживает только 6 типов, а Paradox уже до 15.

База данных может храниться в одном файле (Access) или в нескольких (Paradox, dBase). Точнее сказать, данные таблицы всегда хранятся в одном файле, а вот дополнительная информация может располагаться в отдельных файлах. В качестве дополнительной информации могут быть индексы, ограничения или список значений по умолчанию для конкретных полей. Если хотя бы один из файлов запортиться или будет удалён, то данные могут стать недоступными для редактирования.

Что такое *индексы*? Очень часто данные из таблиц подвергаются каким-то изменениям, поэтому прежде чем произвести редактирование над какой-либо строкой, необходимо её найти. Даже статические таблицы, использующиеся в качестве справочников, тоже подвергаются операциям поиска перед выводом запрашиваемых данных. Поиск достаточно трудоёмкая операция, особенно если таблица содержит очень много строк. Индексы направлены на ускорение этой процедуры, а так же могут использоваться в качестве отправной точки при сортировке. На данном этапе тебе достаточно знать, что не проиндексированное поле невозможно упорядочить.

Если тебе надо, чтобы какая-то таблица была упорядочена по полю «*Фамилия*», то это поле надо сначала проиндексировать. Затем нужно только указать, что таблица должна работать сейчас с таким-то индексом, и она сортируется автоматически.

Delphi и базы данных

Для работы с базами в Delphi есть несколько наборов компонент. Каждый набор очень хорошо подходит для решения определённого круга задач. Почему такое разнообразие компонентов? Все они используют разные технологии доступа к данным и отличаются по возможностям. В отличие от Microsoft, которая встроила в свои продукты разработки только технологию доступа к данным ADO собственной разработки, фирма Borland дала нам разнообразие средств работающих через разные технологии и не ограничивает нас только своими разработками. Такое положение вещей даёт нам громадные преимущества перед другими программистами.

Помимо этого есть группы, которые могут использоваться в любом случае, и здесь я попробую дать краткий обзор доступных нам средств.

На закладке *Data Access* расположены основные компоненты доступа к данным. Эти компоненты общие для всех и могут использоваться совместно с другими группами компонентов.

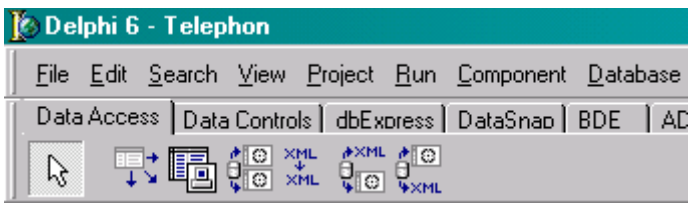


Рис 14.1.1 Закладка Data Access палитры компонентов

На закладке *Data Controls* расположены компоненты для отображения и редактирования данных в таблицах. Эти компоненты так же используются в не зависимости от используемой технологии доступа к данным.

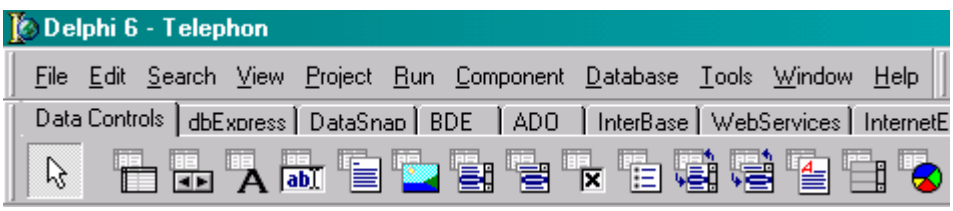


Рис 14.1.2 Закладка Data Controls палитры компонентов

Закладка *BDE* содержит компоненты, позволяющие получить доступ к базам данных по технологии, разработанной фирмой Borland под названием Borland Database Engine. Эта технология сильно устарела и поставляется только для совместимости со старыми версиями. Не смотря на это, она хорошо работает со старыми типами баз данных, такими как Paradox и dBase.

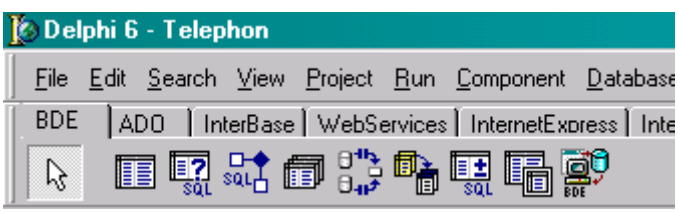


Рис 14.1.3 Закладка BDE палитры компонентов

DBExpress – это новая технология доступа к данным фирмы Borland. Она отличается большей гибкостью и хорошо подходит для программирования клиент серверных приложений, использующих базы данных. Компоненты с одноимённой закладки я советую использовать с базами данных построенных по серверной технологии, например, Oracle, DB2 или MySQL.

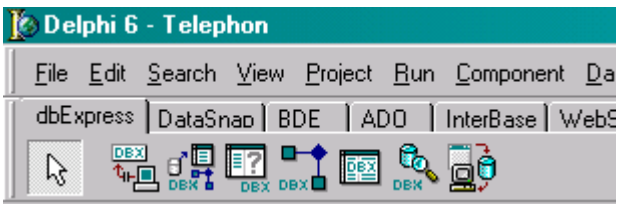


Рис 14.1.4 Закладка dbExpress палитры компонентов

ADO (*Active Data Objects*) – технология доступа к данным, разработанная корпорацией Microsoft. Очень хорошая библиотека, но я рекомендую её использовать только с базами данных Microsoft, а именно MS Access или MS SQL Server. Её так же можно использовать, если у тебя специфичный сервер баз данных, который может работать только через ODBC.

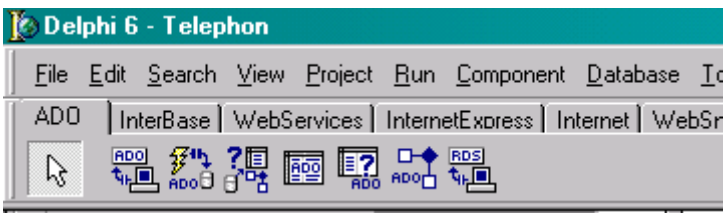


Рис 14.1.5 Закладка ADO палитры компонентов

Работа с базами данных Access идёт через специальную надстройку DAO, которая может устанавливаться на компьютер вместе с программой Office или идти как отдельная установка. Так что если твоя программа не будет работать на компьютере клиента, то надо позаботиться о установке DAO на этот компьютер.

Я не ставлю своей целью расписать абсолютно все эти компоненты, но я постараюсь дать необходимую информацию, чтобы можно было написать профессиональные приложения для работы с базами данных.

14.2 Создание первой базы данных Access

Сейчас я постараюсь подробно рассказать, как создавать и использовать базы данных Access. Для последующей работы необходимо, чтобы на твоём компьютере был установлен MS Office и его компонент MS Access. Именно в нём и будут создаваться базы, а вот работать с ними мы будем уже из Delphi.

Запусти Access и выбери в меню Файл->Создать. В мастере создания базы выбери пункт "База данных" и нажми "ОК" (рисунок 14.2.1). Тебе предложат выбрать имя базы и место расположения, укажи что угодно, а назвал свой файл Database.mdb .

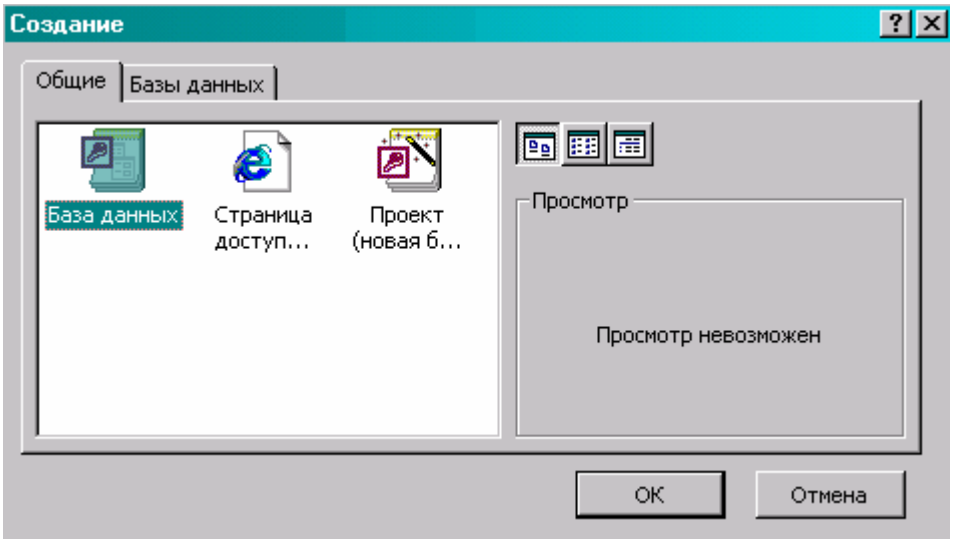


Рис 14.2.1 Окно создания новой базы данных

После этого Access создаст базу и сохранит её по указанному пути. А ты увидишь окно как на рисунке 14.2.2, в котором и происходит работа с базой. С левой стороны окна находится колонка выбора объектов, с которыми ты хочешь работать. Первым находится пункт "Таблицы" (он выделен по умолчанию) который и будет нас интересовать. Если этот объект у тебя не выделен, то выдели его. В окне справа находится три пункта:

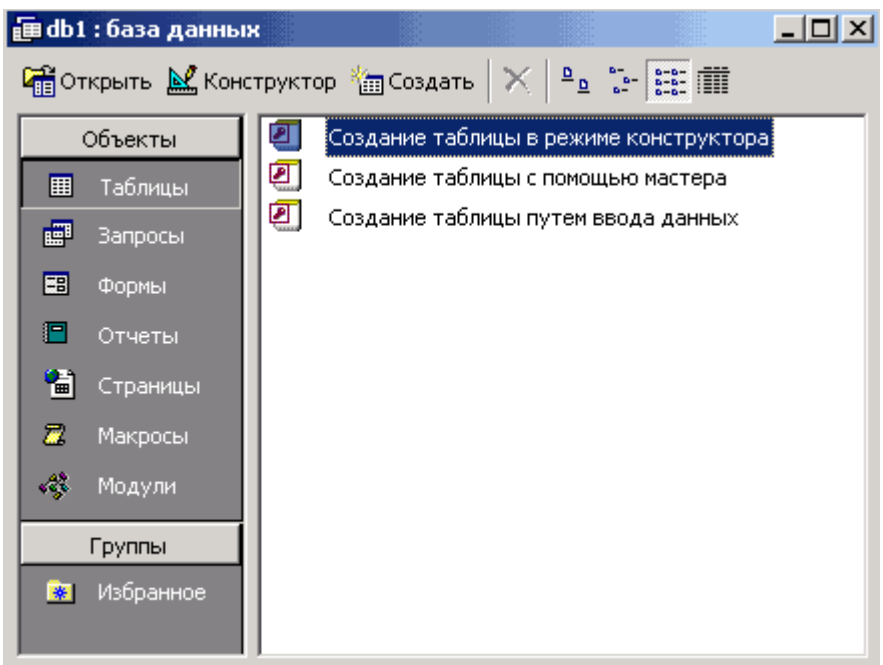


Рис 14.2.2 Окно создания новой базы данных

1. Создание таблицы в режиме конструктора
2. Создание таблицы с помощью мастера
3. Создание таблицы путём ввода данных

С помощью этих команд можно создать таблицы внутри созданной базы данных, Access, которая может хранить в одном файле несколько таблиц.

Все данные в базах данных хранятся в виде двухмерных таблиц. На рисунке 14.2.3 ты можешь видеть пример простой базы данных, состоящей из семи колонок и множества строк.

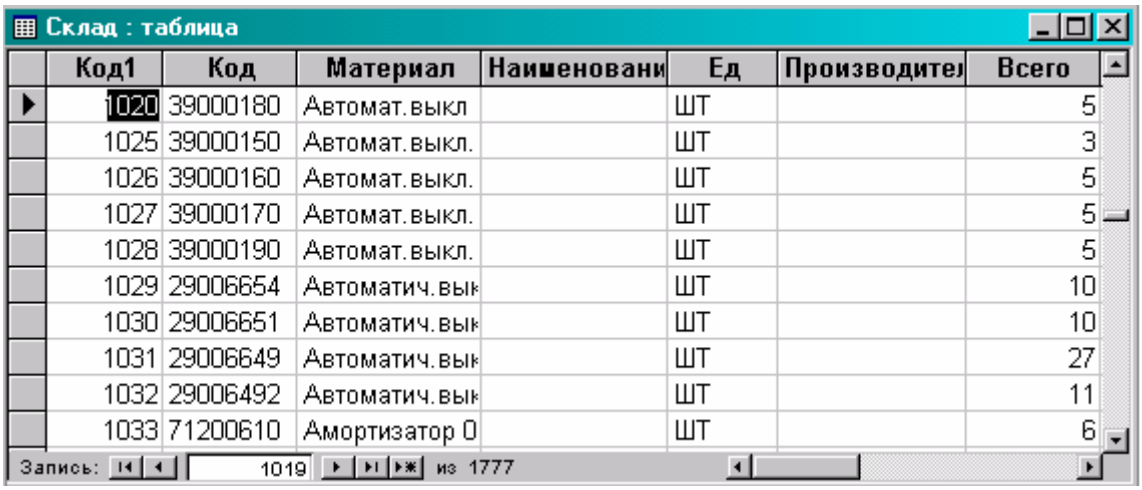


Рис 14.2.3 Пример простой таблицы

Колонки в таблицах называются полями, и по ним определяется, какие именно данные хранятся в таблице. Давай попробуем создать базу данных телефонного справочника, чтобы увидеть всё на практике. Щёлкни по "Создание таблицы в режиме конструктора" чтобы создать новую таблицу в базе данных. Перед тобой откроется окно, как на рисунке 14.2.4.

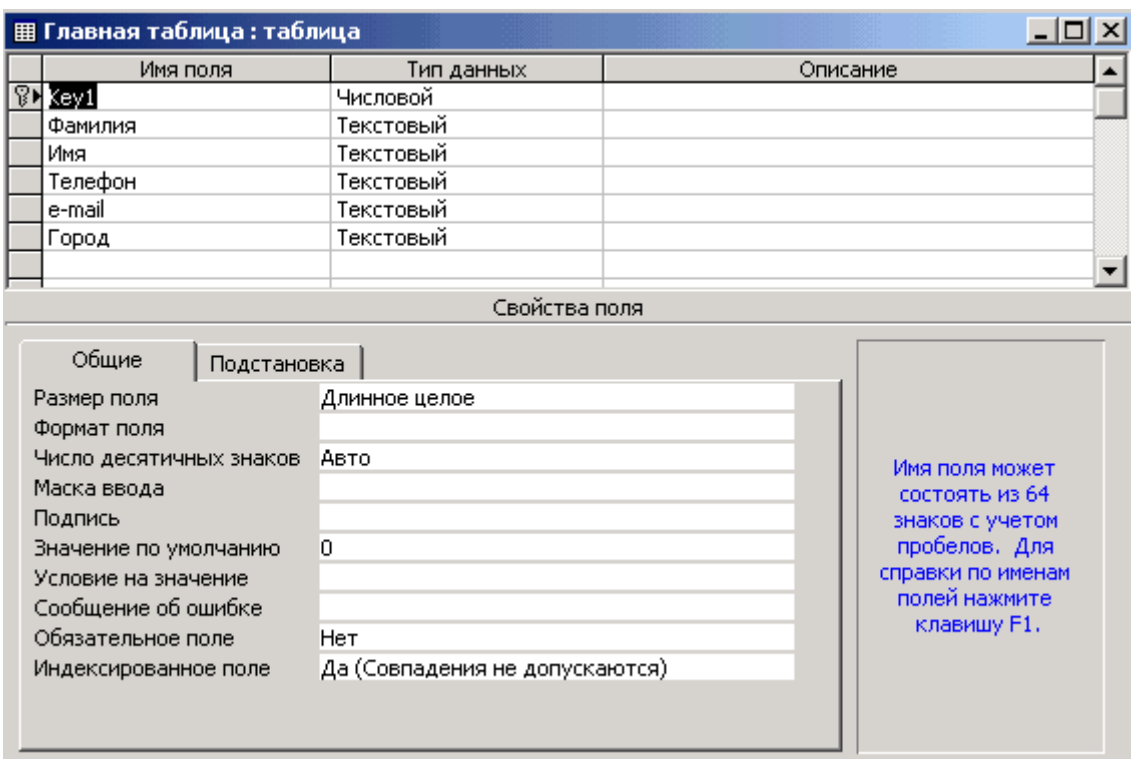


Рис 14.2.4 Окно создания таблицы

Сверху находится сетка, в которой ты вводишь поля таблицы, их тип и описание (последнее не обязательно). Когда ты вписал в сетку имя нового поля и указал тип, внизу окна появляются свойства нового поля. В зависимости от типа поля изменяется и количество свойств. Вот самые основные:

- Масксимальная длинна поля. Для текстового поля размер не может быть больше 255. Если текст длинее, то надо использовать "Поле Memo".

- Формат поля. Здесь ты можешь указать внешний вид данных. Например, поле может выглядить как "Yes/No" для логических полей, или например "mm yyyy" для поля даты.
- Маска ввода. Здесь мы вводим маску, которая отвечает за отображение поля при редактировании. Если ты щёлкнешь на кнопке с точками "..." в строке "Маска ввода", то увидишь мастер создания маски.
- Значение по умолчанию. Умолчание, оно и в африке по умолчанию.
- Обязательное поле. Если пользователь не введёт сюда значение, то появится сообщение об ошибке. Такое поле не может быть пустым.
- Пустые строки. Похоже на предыдущий, потому что это поле тоже не может быть пустым.
- Индексированное поле. Может быть неиндексированным, индексированным с допуском совпадений, и индексированным без допуска совпадений. Основной индекс всегда без допуска совпадений. Остальные желательно с допуском.
- Сжатие Юникод - позволяет сжать данные в соответствии с Юникод.

Создай шесть полей:

1. Имя поля - *Key1*. Тип - счётчик. Это у нас будет ключик. Размер поля - "Длинное целое". Индексированное поле - "Да (Совпадения не допускаются)".
2. Имя поля – *Фамилия*. Тип - текстовый. Размер поля - 50. Индексированное поле - "Да (Допускаются совпадения)".
3. Имя поля – *Имя*. Тип - текстовый. Размер поля - 50. Индексированное поле - "Да (Допускаются совпадения)".
4. Имя поля – *Телефон*. Тип - текстовый. Размер поля - 10. Индексированное поле - "Да (Допускаются совпадения)".
5. Имя поля - *e-mail*. Тип - текстовый. Размер поля - 20. Индексированное поле - "Да (Допускаются совпадения)".
6. Имя поля – *Город*. Тип - числовой. Размер поля - Длинное целое. Индексированное поле - "Нет". Почему город не строковый, ведь названия городов – это текст? Пока я не буду объяснять этого феномена и оставлю его на потом. Чуть позже я покажу, почему город должен быть числовым.

Помимо этого, у всех полей значение "*Обязательно поле*" стоит в "*Нет*", и "*Пустые строки*" выставлено в "*Да*". Если ты сделаешь поле обязательным, то во всех строках обязательно должно быть заполнено соответствующее поле. Если ты запретишь пустые строки (поставишь «*Нет*»), то в указанном поле должно быть обязательно что-то введено, иначе произойдёт ошибки. В реальных условиях, если какое-то поле обязательно должно иметь значение, то лучше сделать его обязательным. Не надо надеяться на добропорядочность пользователя, потому что они слишком часто подводят. Пусть лучше база данных следит за правильностью данных.

Теперь выдели первое поле (Key1), щёлкни правой кнопкой мыши и выбери пункт "Ключевое поле" (рисунок 14.2.5). Задание ключевого поля является обязательным действием, если ты этого не сделаешь, то таблица не сможет редактироваться, а это значит, что в неё нельзя будет добавить строки.

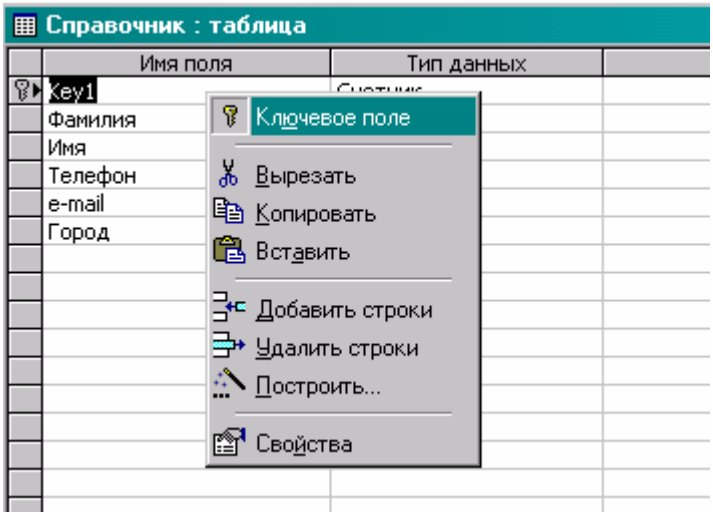


Рис 14.2.5 Задание ключевого поля

Всё, теперь таблицу можно сохранять и закрывать. На вопрос: «Сохранить таблицу» отвечай положительно и сохраняй под именем «*Справочник*».

Наша первая база данных готова. Закрывай её и переходи к следующей части моей книги, где мы напишем первый пример для работы с нашей базой и таблицей.

14.3 Пример работы с базами данных

Мы пишем программу, которая будет работать с базой данных MS Access. Я уже говорил, что для разработок от MS лучше всего использовать ADO. Давай напишем наше первое приложение для работы с базой данных.

- ADOConnection.

Создай новый проект. Теперь брось на форму компонент *ADOConnection* с закладки ADO палитры компонентов. Теперь настроим соединение с сервером, которое должно быть прописано в свойстве *ConnectionString*. Для этого надо дважды щёлкнуть по строке *ConnectionString* и перед нами открывается окно, как на рисунке 14.3.1.

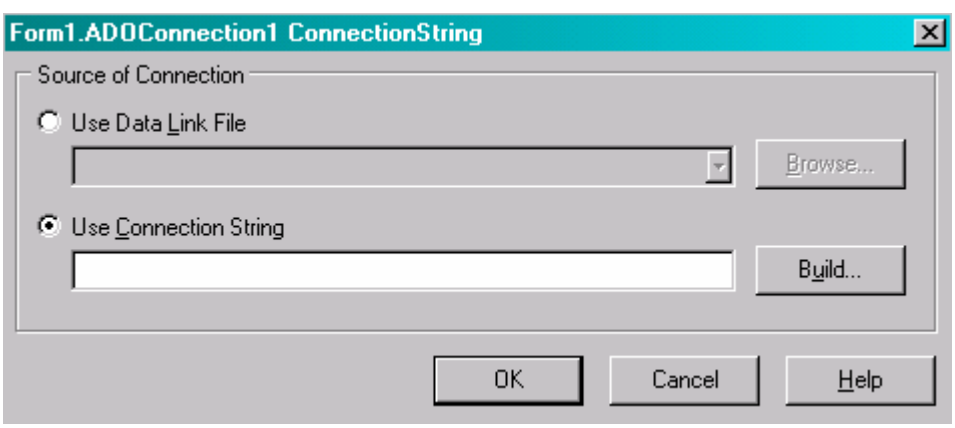


Рис 14.3.1 Окно создания подключения к базе

Здесь перед нами стоит выбор:

1. Использовать специальный файл (*Use Data Link File*);
2. Использовать строку подключения (*Use Connection String*).

Второе, на мой взгляд, более предпочтительно, поэтому я покажу, как создать строку подключения. Для этого щёлкаем кнопку Build и перед нами открывается ещё одно окно, показанное на рисунке 14.3.2.

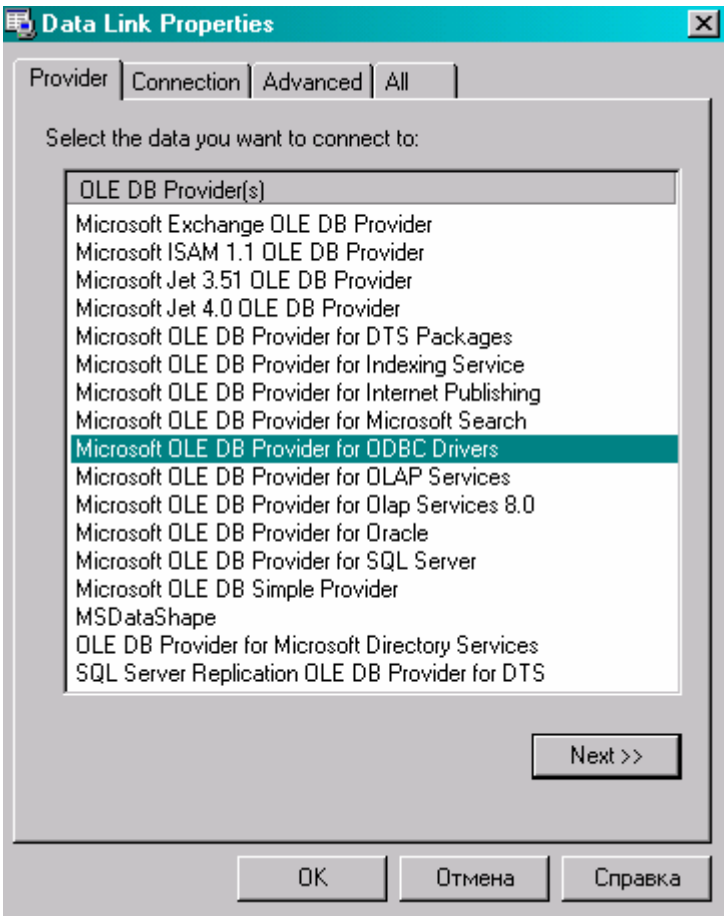


Рис 14.3.2 Окно создания строки подключения

На закладке *Provider* перечислены все доступные ADO драйверы доступа к базам данных. Если какого-то драйвера нет, то можно попробовать выделенный по умолчанию «*Microsoft OLE DB Provider for ODBC Drivers*». Этот драйвер позволяет получить доступ к базе данных через ODBC драйвер, которые есть к большинству существующих баз данных (единственное, он может быть не установленным на твоём компьютере).

В нашем случае, для доступа к базам данных MS Access используется драйвер «*Microsoft Jet OLE DB Provider*». Такой драйвер обязательно устанавливается на машину вместе с MS Office, а в последних версиях Windows он устанавливается по умолчанию.

На моей машине установлено сразу две версии этого драйвера, поэтому я выберу более новый - «*Microsoft Jet 4.0 OLE DB Provider*». После этого нажимаем кнопку Next, или переходим на закладку «Connection».

Вид закладки Connection зависит от выбранного драйвера. В нашем случае она должна выглядеть, как показано на рисунке 14.3.3.

Первым делом, в этом окне надо ввести имя (если надо то и путь) базы данных в строку «*Select or enter a database name*». Если база данных будет располагаться в той же директории, что и запускной файл, то путь указывать не надо. Я вообще советую хранить базы в одной директории с запускными файлами. Если ты будешь держать файлы

отдельно от запускного, то тебе придётся указывать полный путь, а это может вызвать проблемы при переносе программы на другой компьютер. Ведь там программа будет искать базу по казанному пути, который может измениться. Если хочешь держать файлы в другой директории, то указывай относительный путь относительно текущей директории.

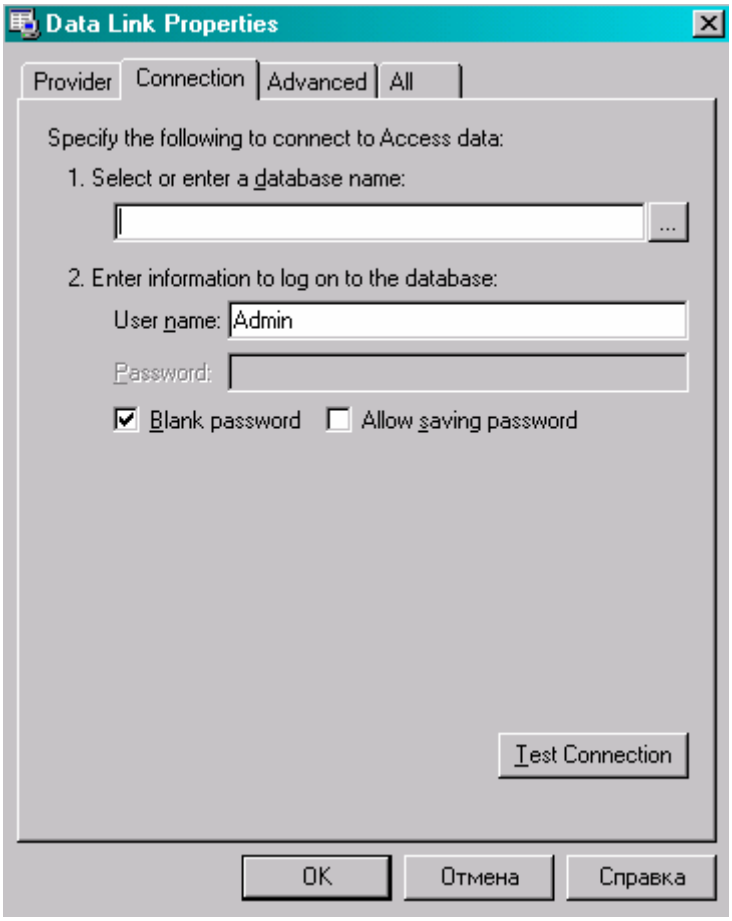


Рис 14.3.3 Закладка Connection

Чтобы легче было выбрать файл базы данных необходимо щёлкнуть по кнопке с точками справа от строки ввода.

Помимо этого нам надо заполнить следующие поля:

1. Имя пользователя (*User name*), можно оставить по умолчанию, если не заданно иное при создании базы в MS Access;
2. Пароль (*Password*) – если база имеет пароль, то его необходимо указать;
3. Пустой пароль (*Blank password*) – если пароль не нужен, то здесь желательно поставить галочку;
4. Позволять сохранять пароль (*Allow saving password*). Если здесь поставить галочку, то пароль может быть сохранён.

Как только выберешь базу данных, нажми кнопку *Test Connection*, чтобы протестировать соединение. Если всё указано правильно, то ты должен увидеть сообщение «*Test connection succeeded*». Всё, можно нажать OK, чтобы закрыть окно создания строки подключения и ещё раз *OK*, чтобы закрыть окно редактора строки подключения (тот, что был на рисунке 14.3.1).

Теперь в свойствах компонента *ADOConnection* отключи свойство *LoginPrompt*, выставив его в *False*. Это нужно для того, чтобы при каждом обращении к базе нас не

грузили окном ввода пароля. А теперь выставим свойство *Connected* в *True*, чтобы произошло соединение с базой.

На этом соединение можно считать оконченным. Теперь нам надо получить доступ к созданной нами таблице «*Справочник*». Для этого брось на форму компонент *ADOTable* с закладки *ADO* палитры компонентов. Сразу измени его свойство *Name* на *BookName*.

 - TADOTable.

В этом компоненте тоже есть свойство *ConnectionString* и его так же можно настраивать. Почему «можно»? Да потому что, чтобы этого не делать мы поставили на форму компонент *ADOConnection*. Теперь мы можем указать у нашего компонента *BookName* в свойстве *Connection*, созданный нами компонент соединения с базой данных. Щёлкни по выпадающему списку в свойстве *Connection* и выбери там единственный пункт *ADOConnection1*. Теперь нам не надо заполнять свойство *ConnectionString*.

Теперь в свойстве *TableName* нужно выбрать имя нашей таблицы (*Справочник*). Всё, таблица и соединение указаны, можно подключатся. Для этого выставь свойство *Active* в *true*.

 - TDataSource

Для отображения данных из таблицы надо ещё установить на форму компонент *DataSource* с закладки *Data Access* палитры компонентов. Теперь этому компоненту надо указать, какую именно таблицу он должен отображать. Для этого в свойстве DataSet нужно из выпадающего списка выбрать нашу таблицу *BookTable*.

 - DBGrid

Всё!!! Все приготовления готовы, можно приступать к реальному отображению данных. Самый простой способ отобразить таблицу – установить компонент DBGrid. Это компонент-сетка, которая может отображать данные в виде таблицы. В этом же компоненте можно добавлять, удалять и редактировать строки нашей таблицы.

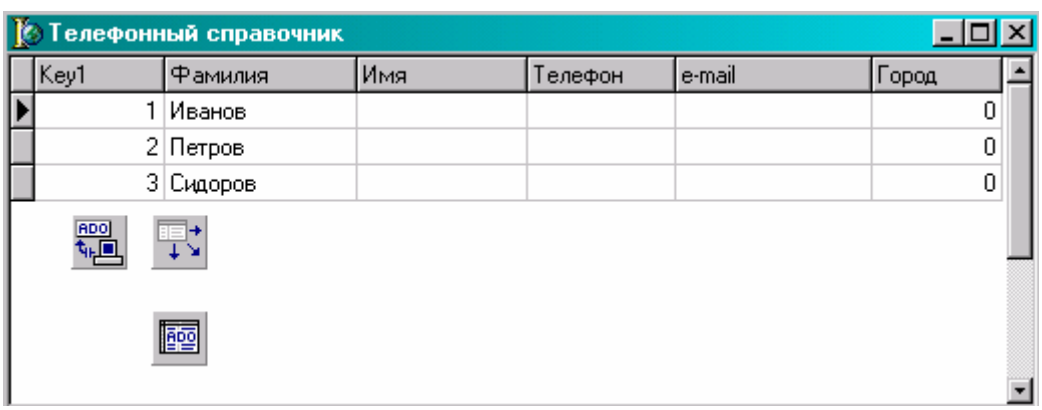


Рис 14.3.4 Форма нашего приложения

И последний этап создания нашего приложения – связывание компонента сетки с компонентом отображения таблицы. Для этого в свойстве *DataSuorce* компонента *DBGrid* нужно указать созданный нами компонент *DataSource1*.

Вот теперь наше приложение готово!!! Может ты не заметил, но мы не написали ни одной строчки кода :). Вот до какой степени Delphi упрощает процесс программирования баз данных, что даже программировать ничего не надо.

Попробуй запустить этот пример и создать несколько строк, отредактировать уже существующие и удалить что-нибудь. Для вставки строки используй клавишу **Ins**, а для удаления **Ctrl+Del**.

 *На компакт диске, в директории* ***\Примеры\Глава 14\Database1*** *ты можешь увидеть пример этой программы.*

14.3.1 Свойства компонента TADOTable

Компонент ***TADOTable*** имеет множество полезных свойств. Большинство из них просты в использовании, поэтому чтобы не писать множество примеров на их использование, я здесь кратко опишу основные из них. В дальнейшем с какими-то мы познакомимся на практике, а какие-то останутся для тебя как дополнительная информация к размышлению.

MasterSource - в этом свойстве указывается главная, по отношению к текущей таблица. Мы рассмотрим это свойство достаточно подробно и на практике, когда будем рассматривать связанные таблицы.

ReadOnly – если это свойство равно *true*, то таблицу нельзя редактировать. В этом случае данные только отображаются. Обязательно устанавливай это свойство для тех таблиц, где данные не должны изменяться и пользователь не должен вносить в них изменения.

TableDirect – это свойство отображает какой будет происходить доступ к таблице. Если этот параметр равен *true* то будет происходить прямой доступ к таблице по имени. Если *false* то незаметно для тебя будет происходить специальный SQL запрос к базе данных (о SQL запросах читай ниже). Не все базы данных позволяют работать через прямой доступ, поэтому это свойство по умолчанию равно *false*.

TableName – имя таблицы, данные которой мы хотим обрабатывать.

CacheSize – размер кэш памяти. Если здесь установить число 50, то при первом подключении к таблице компонент выберет первые 50 строк и поместит их в локальной памяти, что ускорит доступ к ним.

CanModify – свойство похоже на *ReadOnly* и указывает на возможность редактирование данных таблицы.

CommandTimeout – время ожидания выполнения команды. Когда компонент направляет команду базе данных, то он запускает таймер ожидания, по истечению которого (если команда не выполнилась) происходит сообщение об ошибке.

Connection – здесь указывается компонент ***TADOConnection***, через который происходит подключение.

ConnectionString – строка подключения к базе данных.

CursorLocation – расположение курсора, который считывает данные и указывает текущую позицию в таблице. Курсор может находиться на сервере или на машине клиента.

CursorType – тип курсора. Тут возможен один из следующих вариантов:

- ctUnspecified расположение курсора не указано
- ctOpenForwardOnly – курсор может двигаться только вперёд.
- ctKeyset при этом курсоре изменения внесённые одним пользователем не видны остальным пользователям подключённым к этой таблице. Если с одной таблицей

работают одновременно несколько пользователей, то при таком курсоре для отображения изменений других пользователей нужно отключиться от базы и подключиться к ней снова.

- ctDynamic динамический курсор, при котором изменения одного пользователя видят все остальные.

- ctStatic статический курсор. Изменения одного пользователя не видны остальным

Внимание!!! Если курсор расположен на клиенте, то можно использовать только статический курсор. Не все типы курсоров могут работать с определённой базой данных. Одна база данных может поддерживать один тип, а другая может поддерживать всё.

Filter – строка фильтра.

Filtered – является ли таблица фильтруемой. Если здесь установить *false* то строка фильтра (filter) игнорируется.

IndexFieldNames – имя индексированной колонки. Индексы используются для сортировки данных или для связи между таблицами.

RecNo - номер текущей выделенной строки.

RecordCount – количество строк в таблице.

Sort - строка, в которой указывается тип сортировки. Например, для сортировки по полю «Телефон» сюда нужно записать строку: *ADOQuery1.Sort := 'Телефон ASC'*. Оператор *ASC* говорит о том, что надо сортировать в порядке возрастания. Оператор *DESC* говорит о сортировании в порядке убывания.

Active – если это свойство равно *true*, то таблица открыта.

AggFields – здесь хранятся все агрегатные поля.

AutoCalcFields – если здесь *true*, то надо автоматически пересчитывать поля.

Bof – на это свойство влиять нельзя, но если оно равно *true*, то мы находимся в начале файла.

Bookmark - здесь находиться текущая закладка.

Eof - на это свойство влиять нельзя, но если оно равно *true*, то мы находимся в конце файла.

FieldCount – здесь хранится количество полей в таблице.

Fields – через это поле можно получить доступ к значениям полей. Допустим, что тебе надо узнать, какое значение храниться в 4-м поле. Для этого нужно написать *Table.Fields.Fields[4].AsString*. Метод *AsString* говорит о том, что нам надо получить значение в виде строки. Это простой метод, но я его не люблю использовать. В течении всей книги я буду обращаться к полям по именам.

FieldValues – с помощью этого свойства можно легко получить доступ к любому значению указанного поля. Имя поля нужно указывать в квадратных скобках. Например, *Table1.FieldValues['Телефон']:='3346598';*

FilterOption – настройки фильтра. Здесь можно указывать следующие параметры:

- foCaseInsensitive фильтр будет не чувствителен к регистру.

- foNoPartialCompare если стоит этот параметр, то сравнения будут происходить с точной копией указанного значения в фильтре. Если параметр не казан, то в фильтр будут попадать строки содержащие значение в фильтре, но не являющиеся его точной копией. Например, если в фильтре указано показывать слова «са», то в фильтр попадут все слова начинающиеся на «са» (самолёт, самокат).

Modified – если это свойство равно *true*, то в таблице были внесены изменения.

Как видишь, свойств очень много и большинство из них очень полезные. В течении этой главы я буду знакомить тебя с ними и мы увидим большинство свойств на практике. А теперь приготовься получить описание громадного количества методов. Они не менее полезны, и мы так же будем знакомиться с большинством из методов на практике

BookmarkValid – этот метод проверяет правильность закладки. В качестве единственного параметра нужно указать закладку типа ***TBookmark*** и если она является действительной, то результатом будет *true*.

CancelUpdates - отменить обновления сохранённые в кэш памяти

CompareBookmarks.- сравнение двух закладок. У метода два параметра типа ***TBookmark***. Эти две закладки сравниваются. Если закладки равны, то результат равен нулю. Если первая меньше второй, то результат будет –1. Если первая больше второй, то результат равен единице.

DeleteRecords – удалить записи. У метода один параметр – какие записи удалять. Ты можешь указать следующие значения в качестве параметра:

- arCurrent удалить только текущую запись.
- arFiltered удалить записи удовлетворяющие установленному фильтру.
- arAll – все записи.
- arAllChapters удалить записи во всех разделах ADO.

Append – добавить новую запись в конец таблицы.

Cancel – отменить изменения текущей строки, если изменения ещё не были сохранены с помощью метода *Post*.

Close – закрыть таблицу.

Delete – удалить текущую строку.

Edit – перейти в режим редактирования. После этого можно изменять значения полей.

FieldByName – найти поле по имени. В качестве единственного параметра нужно указать имя поля виде строки и в результате получаем ссылку на поле в виде объекта TField.

First – перейти на первую строку в таблице.

Insert – вставить новую строку в таблицу.

IsEmpty – если метод вернёт *true* то в таблице нет записей.

Last - перейти на последнюю запись в таблице.

Next – перейти на следующую запись.

Post – принять все изменения.

Prior - двигаться на предыдущую запись в таблице.

Refresh – обновить информацию о данных.

UpdateRecord – обновить текущую запись.

14.4 Управление отображением данных

В предыдущем примере всё работает прекрасно, только вот после Key пользователю видеть абсолютно не нужно. Это поле – счётчик и его значение увеличивается автоматически. А раз пользователь не может влиять на поле и оно не несёт в себе полезной для него информации, то и видеть это поле он не должен.

Чтобы прятать от пользователя не нужные поля и показывать только то, что мы хотим и в том виде, в котором хотим, нам необходимо научиться управлять отображением данных. Но прежде чем приступить к этому, давай создадим в нашей базе ещё два поля

«*Дата*» и «*Мобильник*». Загрузи нашу базу данных в Access, щёлкни по ней правой кнопкой и в появившемся меню выбери «*Конструктор*».

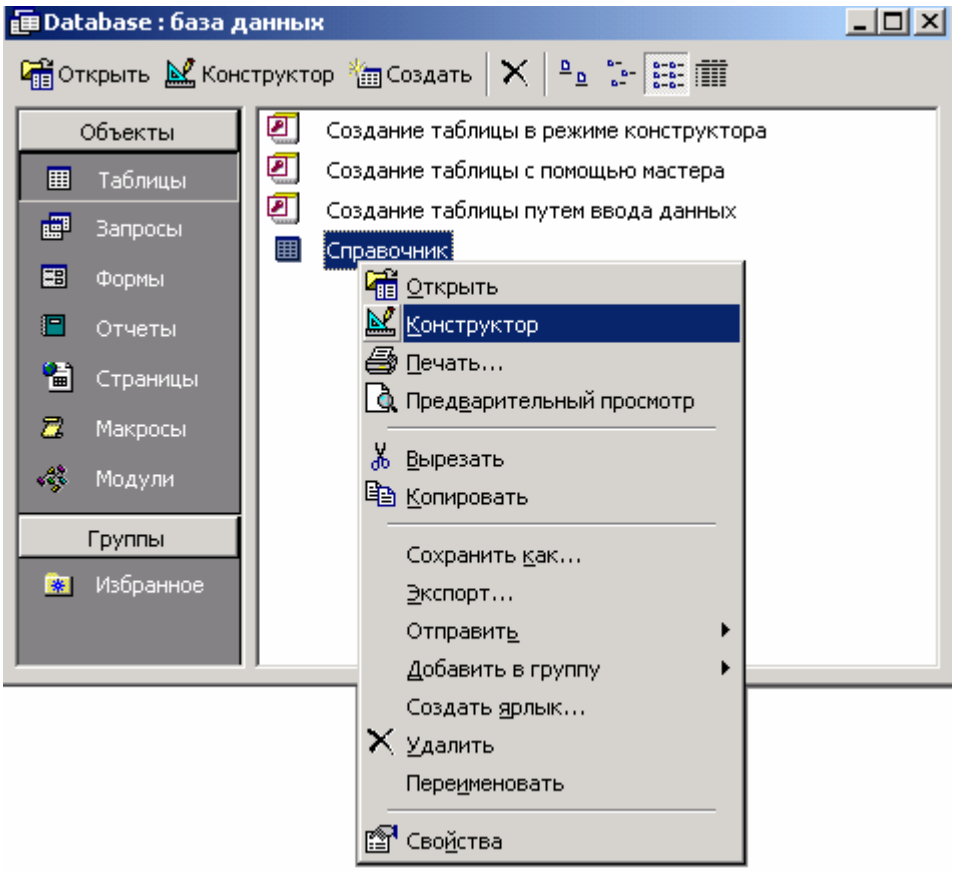


Рис 14.4.1 Редактирование таблицы

1. Добавь поле с именем «*Дата*», тип «*Дата/время*».
2. Добавь поле с именем «*Мобильник*», и тип «*Логический*». Если в строке находиться мобильный телефон, то в этом поле будем ставить **true**, иначе «**false**».

Закрой таблицу. Теперь переходим в Delphi и попробуем отобразить изменения в уже созданном примере.

Для начала давай перенесём компоненты доступа к базе данных в отдельное специальное окно. Выдели компоненты *ADOConnection1*, *DataSource1* и *BookTable*.Теперь выбери из меню *Edit* пункт *Cut*, что бы эти компоненты скопировались в буфер обмена и сразу удалились с формы.

Теперь выбери из меню *File->New->Data Module* (рисунок 14.4.2). Этим ты заставишь Delphi создать специальное окно *Data Module*, которое удобно подходит для хранения компонентов доступа к базам данных.

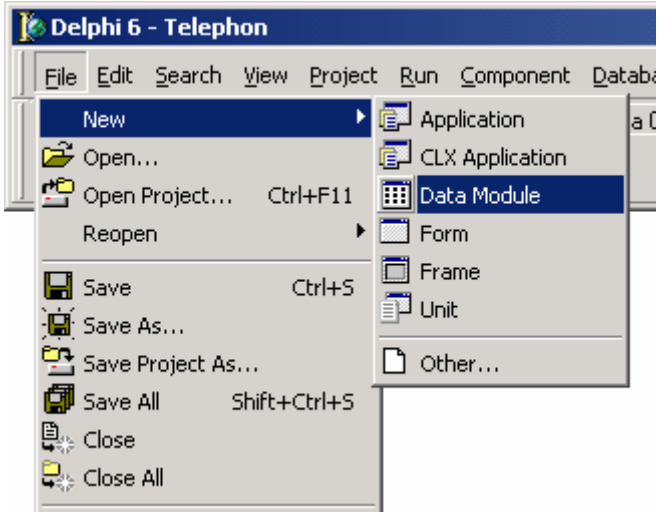


Рис 14.4.2 Создание модуля Data Module

Теперь выбери из меню *Edit* пункт *Paste*, чтобы вставить в это окно вырезанные нами компоненты. Расположи теперь эти компоненты в окне так, как тебе будет удобно. Я сделал это так, как показано на рисунке 14.4.3.

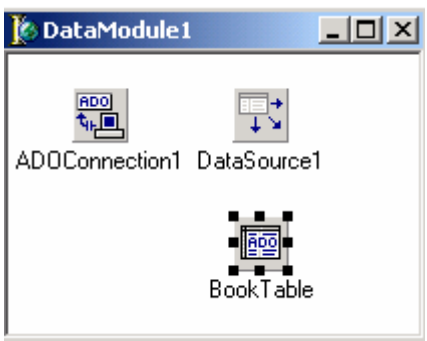


Рис 14.4.3 Окно Data Module

Теперь все компоненты, которые предназначены для доступа к базе данных будем располагать здесь, чтобы с ними удобно было работать. Сохрани новый модуль под именем *DataModuleUnit*.

Теперь открой менеджер проектом (в меню *Project* надо выбрать *Project Manager*) и расположи это окно так, чтобы тебе было удобно в любой момент получить к нему доступ. Я всегда располагаю его в правом нижнем углу экрана.

Теперь, когда тебе надо перейти из главной формы в модуль данных *DataModule* или обратно, ты легко можешь сделать это с помощью менеджера проектов, дважды щёлкнув по нужной форме.

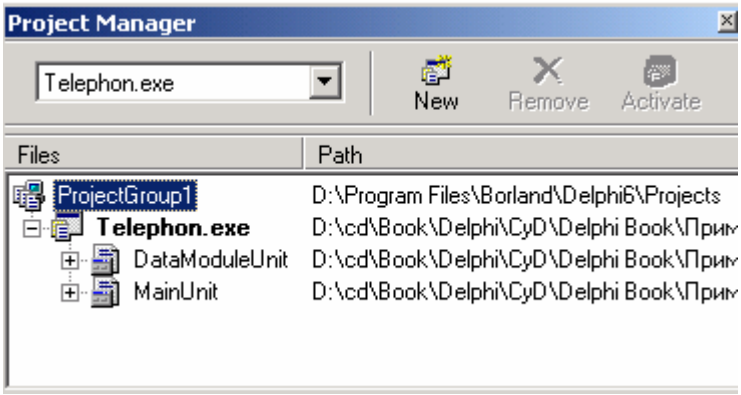


Рис 14.4.4 Менеджер проектов

Если ты хоть раз уже открывал какую-то форму из менеджера проектов и не закрывал, то её можно найти на закладках в окне редактора кода:

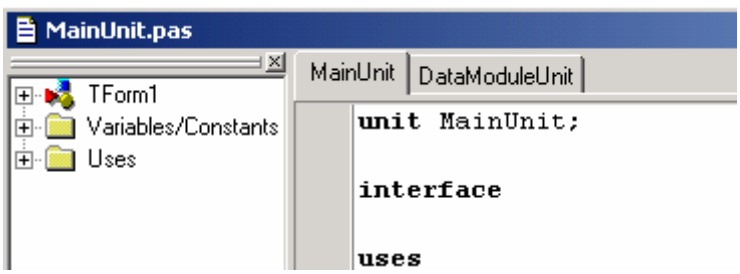


Рис 14.4.5 Закладки форм в редакторе кода

И ещё хочу напомнить, что переключаться между визуальной формой и её кодом очень удобно простым нажатием клавиши F12.

На этом инструкции по работе с оболочкой заканчиваю и пора двигаться дальше.

Перейди в главную форму и ты сразу увидишь, что в нашей сетке *DBGrid1* нет данных. Почему? Да потому что она потеряла связь с компонентами доступа к данным. Выдели сетку и щёлкни по свойству *DataSource*, и ты увидишь, что в выпадающем списке ничего нет. Это потому что все нужные компоненты мы убрали в отдельную форму и главная форма пока об этом не знает.

Чтобы форма узнала о существовании компонентов, ей нужно указать в разделе **uses** наш модуль *DataModuleUnit*. Это можно сделать вручную или выбрать из меню *File* пункт *Use Unit* (в этот момент должно быть выделено окно кода главной формы, потому что мы подключаем новый модуль именно к ней. В появившемся окне (рисунок 14.4.6) нужно выбрать имя нашего нового модуля *DataModuleUnit* (пока оно одно в списке) и нажать **ОК**.

Проверь теперь в редакторе кода, чтобы после ключевого слова **implementation** появилось «***uses** DataModuleUnit;*»:

```
implementation

uses DataModuleUnit;

{$R *.dfm}
```

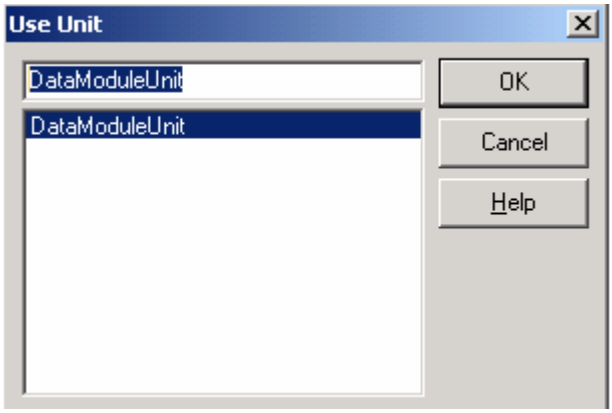


Рис 14.4.6 Окно подключения нового модуля

Вот теперь можно выделять нашу сетку *DBGrid1* и в свойстве *DataSource* указывать компонент *DataSource*, данные которого должны быть отображены в сетке (*DataModule1.DataSource1*).

Теперь переходим в модуль *DataModule* и попытаемся настроить отображение данных. Дважды щёлкни по компоненту *BookTable* и перед тобой появиться окно редактирования полей базы данных (рис 14.4.7).

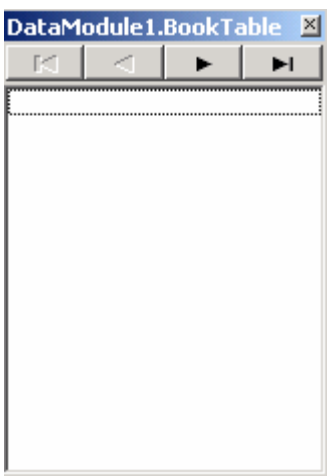


Рис 14.4.7 Окно редактирования полей базы данных

Пока что оно пустое и суда нужно добавить все поля базы данных. Для этого щёлкни в нём правой кнопкой мыши и в появившемся меню выбери пункт *Add All Field* (Добавить все поля). Окно автоматически заполниться именами полей (рис 14.4.8).

Свойства можно переставлять местами двигая мышкой. При этом физическое расположение в базе данных не меняется, зато при отображении данных в сетке, они будут отображаться в том порядке, в котором они выстроены здесь. Так что ты в любой момент можешь изменить порядок отображения данных, не вмешиваясь в саму базу данных.

Ты можешь теперь выделять отдельные поля и в объектном инспекторе редактировать его свойства. Свойства у полей могут быть разные, в зависимости от типа поля. Я сейчас не буду их расписывать, уж лучше мы постепенно познакомимся с ними на практике и увидим их действие.

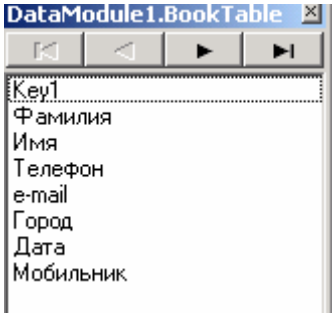


Рис 14.4.8 Заполненное окно редактирования полей базы данных

Первое, что мы должны сделать – убрать из видимости счётчик (поле Key). Мы уже договорились, что пользователю оно не нужно и он не должен его видеть. Выдели это свойство и в объектном инспекторе установи в свойстве *Visible* значение *false* (это свойство есть у всех полей). Сразу же можешь перейти в главную форму или запустить программу, чтобы убедиться в том, что поле *Key1* больше не отображается.

Теперь отредактируем длину отображения колонок. Для этого выдели свойство «*Фамилия*». В базе данных мы выделили под это поле 50 символов (на всякий случай). В сетке ширина колонки будет отображаться по умолчанию на всю длину. Но чаще всего фамилии не превышают 15 символов, поэтому нет смысла отображать всю длину. На много удобней отображать только 15 символов, а если что-то не поместиться, то пользователь программы в любой момент сможет раздвинуть колонку и увидеть недостающие символы.

За ширину колонки отвечает свойство *DisplayWidth* (это свойство есть у всех полей). По умолчанию в нём стоит значение физической ширины поля, но мы укажем там 15. Опять же, на саму базу данных это не влияет и поле всё ещё имеет размер 50, но ширина отображаемой колонки в сетке будет 15. Точно так же сократи ширину поля «Имя».

Ещё несколько интересных свойств:

DefaultExpression – здесь можно указать значение по умолчанию. В дальнейшем, когда будут создаваться новые строки, то в поля будут сразу заноситься указанные здесь значения.

MaxValue – максимально допустимое значение. Если это числовое поле и оно должно изменяться в определённых рамках (например, от 0 до 100), то желательно указать эти ограничения здесь, чтобы сократить вероятность опечатки пользователем. Все люди склонны к ошибкам, так пускай программа автоматически сокращает вероятность таких ошибок.

MinValue – минимально допустимое значение.

ReadOnly –поле только для чтения. Если какой-то поле не должно изменяться, то установи у него в свойстве *ReadOnly* значение *true*. В этом случае ты обезопасишь программу от случайного изменения данного поля пользователем.

Required – если здесь *true*, то поле является обязательным и обязательно должно иметь какое-то значение. Если пользователь ничего не укажет, то программа сообщит об этом. Допустим, что какое-то поле у тебя участвует в расчётах. Если в этом поле не окажется данных, то программа может зависнуть. Есть два пути – при расчёте проверять наличие в поле данных или требовать, чтобы пользователь обязательно что-то ввёл. Второй путь предпочтительней, если это поле действительно важное. Представь запись в телефонном справочнике без телефона. Спрашивается, зачем тогда нужна эта запись если не указан телефон. Так что поле для номера телефона можно делать обязательным.

Tag – просто числовое значение, которое можно использовать по своему усмотрению.

Теперь в нашем окне помещается практически вся необходимая информация и не надо лишний раз использовать полосы прокрутки (рис 14.4.9).

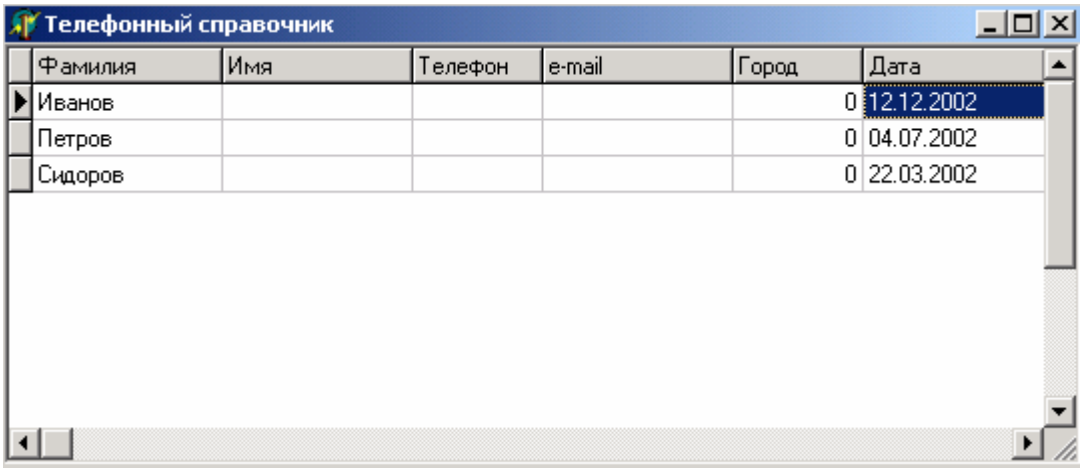


Рис 14.4.9 Улучшенное окно программы

Запусти программу и заполни поле «Дата» у всех записей любыми значениями. При заполнении будь внимателен и указывай реальные даты. Если ты введёшь недопустимое значение, то программа высветит ошибку.

При вводе данных учитывай разделитель чисел. В большинстве русскоязычных ОС Windows в качестве разделителя используется точка или знак косой черты «/». К тому же первым идёт число, потом месяц и потом год (в англоязычной версии первым может идти месяц). Пробелы не допустимы. Этот порядок и разделитель настраиваются в настройках ОС. Войди в «Панель управления» и запусти окно «Язык и стандарты» (рисунок 14.4.10). Здесь ты можешь изменить все настройки ввода даты, времени и чисел.

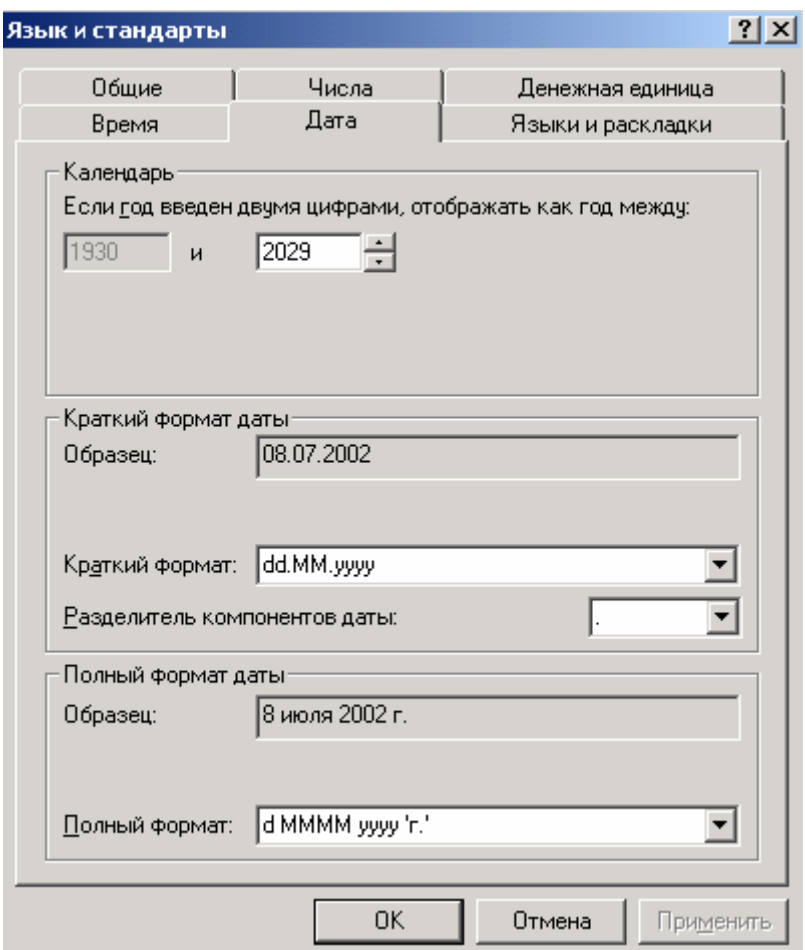


Рис 14.4.10 Настройка формата ввода и отображения даты

Вернёмся к Delphi. Выдели поле «*Дата*». Первое, что мы должны сделать – уточнить, какую именно дату здесь надо вводить. Так как это телефонный справочник, то я собираюсь здесь указывать дату рождения владельца телефона. Поэтому, вполне разумно будет в заголовке сетке отображать не просто «*Дата*», а «*Дата рождения*». Тут можно поступить двумя способами – отредактировать имя поля в базе данных (не совсем разумно) или просто заставить Delphi отображать в заголовке поля нужный текст.

За текст отображаемый в заголовке отвечает свойство *DisplayLabel* (это свойство есть у всех полей). Давай в нём введём текст «Дата рождения». Можешь проверить, что теперь в заголовке отображается нужный текст.

Теперь отредактируем формат отображения даты. За это отвечает свойство *DisplayFormat*. Тут можно указывать текстовый формат, в котором нужно отображать дату. Как отображать? Вспомни функцию *FormatDateTime* и её первый параметр (см главу 10.5 «Преобразование данных»). Вот именно это здесь и можно указывать. Лично я люблю использовать для отображения полный формат – «*dddddd*».

Ну и наконец нужно указать маску ввода для даты. Её нужно указывать в свойстве EditMask и так же, как мы это делали у компонента ***TMaskEdit***. Для даты я всегда указываю маску ввода «99/99/9999».

Если ты теперь запустишь наш пример, то в поле «Дата рождения» все даты будут отображаться в полном формате (рис 14.4.11). Если щёлкнуть дважды по этому полю в любой строке (войти в режим редактирования строки), то дата сразу перейдёт в режим редактирования (см рис 14.4.11 вторая строка.

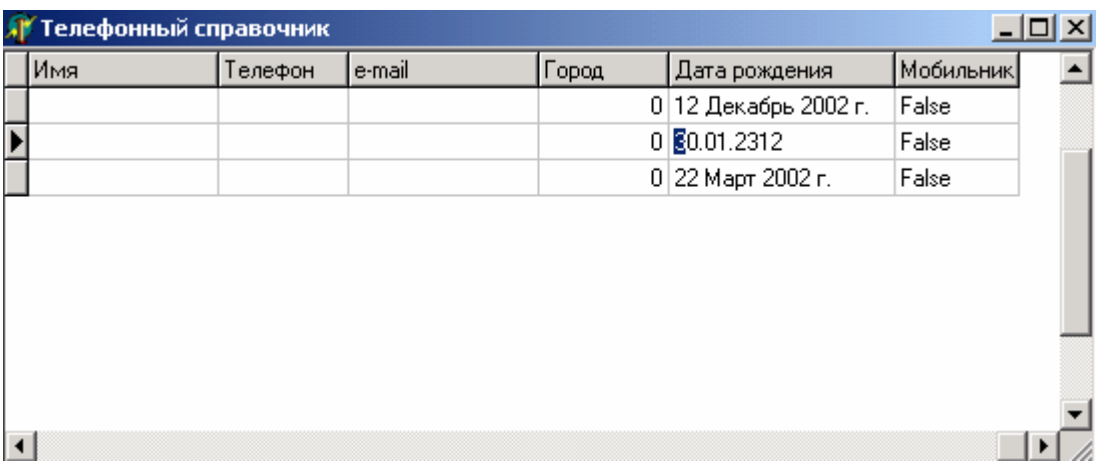


Рис 14.4.11 Улучшенный вид поля «Дата рождения»

Последнее, что нам надо отредактировать – поле «*Мобильник*». Пока что здесь отображаются значения **true** или **false**. Но мы русские люди и для нас будет удобнее видеть родные **Да** или **Нет**. Выдели это поле и в объектном инспекторе найди свойство DisplayValues. Для булевых полей здесь нужно указать два значения в формате «True;False», т.е. сначала указываем положительное значение и после точки с запятой отрицательное (кавычки указывать не надо). Итак, я указал ***Да;Нет***.

Теперь в поле «*Мобильник*» будут отображаться понятные слова, да и при редактировании теперь нужно вводить не **true** или **false**, а родные **Да** или **Нет**.

*На компакт диске, в директории **\Примеры\Глава 14\Database2** ты можешь увидеть пример этой программы.*

14.5 Поисковые поля

Настало время работать наше поле *Город* которое имеет числовой тип и никак не может пока хранить данные о городе. Для этого мы создадим отдельную таблицу в нашей базе данных с полями:

1. ***Key1*** – счётчик (ключевое поле);
2. ***Название города*** – текстовое поле размером в 30 символов.

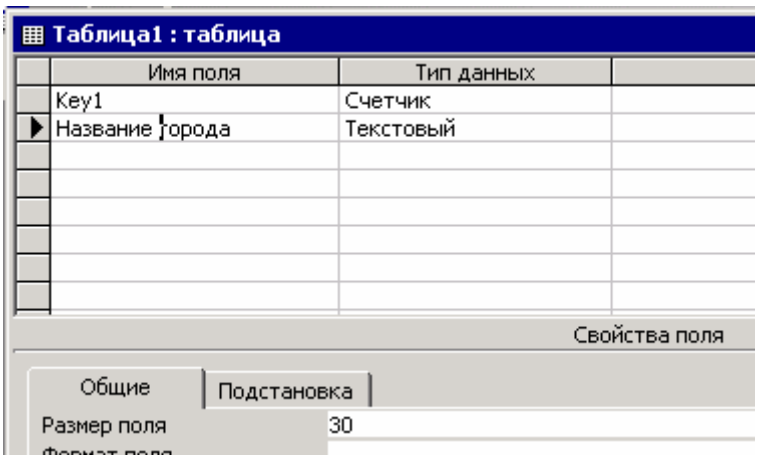


Рисунок 14.5.1 Таблица «Справочник городов»

Сохрани новую таблицу под именем «*Справочник городов*». Теперь твоя база данных должна состоять из двух таблиц:

1. Справочник;
2. Справочник городов.

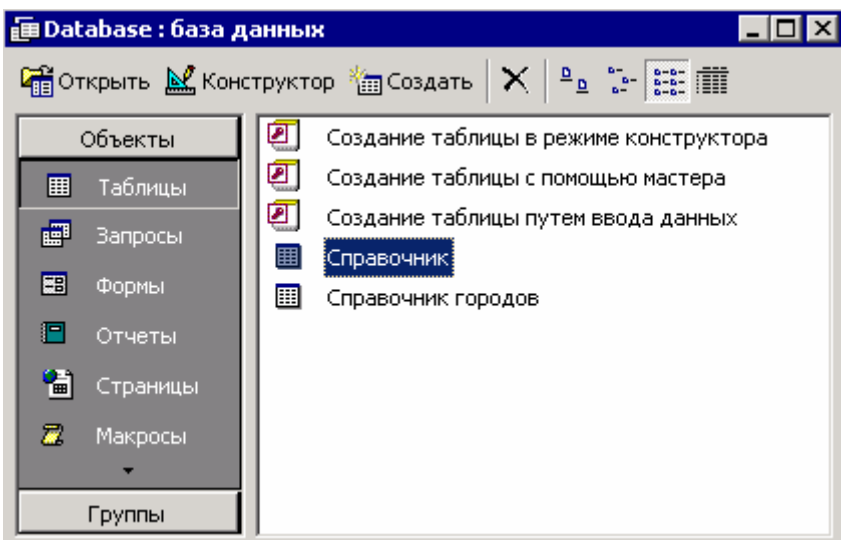


Рисунок 14.5.2 Обновлённый вид базы данных

Теперь давай откроем проект созданный в прошлой части главы и модуль *DataModuleUnit*. Добавь сюда компонент *DataSource* (назовём его *TownSource*) и *ADOTable* (его назовём *TownTable*). После этого у компонента *TownSource* в свойстве *DataSet* укажи таблицу TownTable.

Теперь давай настроим *TownTable* на отображение справочника городов. Для этого:

1. В свойстве *Connection* укажи компонент *ADOConnection1*, который указывает на нашу базу данных.
2. В свойстве *TableName* укажи таблицу *Справочник городов*.
3. Установи свойство *Active* в *True*, чтобы активизировать таблицу.

Войди в редактор полей таблицы *TownTable* и добавь все поля. Сделай поле Key1 невидимым, потому что это счётчик и пользователю он абсолютно не нужен.

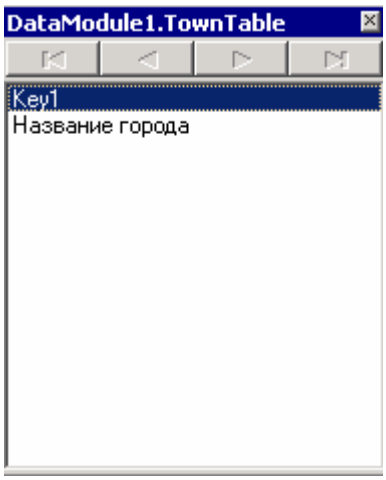


Рисунок 14.5.3 Редактор полей таблицы

Теперь создадим новую форму, для редактирования справочника и сохрани форму в модуле под именем *TownBookUnit*. Саму форму назовём *TownBookForm*. Подключи к новой форме модуль *DataModuleUnit*, чтобы отсюда можно было получить доступ к компонентам для работы с базами данных. Для этого из меню *File* выбери пункт *Use Unit* и в появившемся окне укажи модуль *DataModuleUnit* и нажми *OK*.

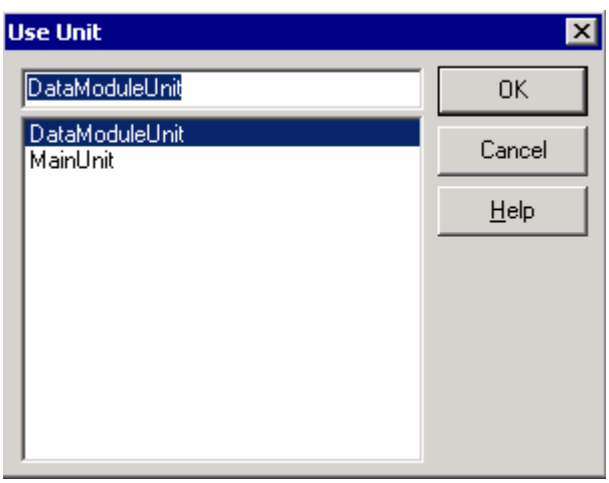


Рисунок 14.5.4 Добавление модуля DataModuleUnit.

Брось на форму сетку *DBGrid* и в свойстве *DataSource* укажи таблицу справочника городов - *DataModule1.TownSource*. Можешь всё это дело красиво оформить и добавить кнопку ОК, для закрытия окна справочника. Моё окно редактора справочника городов ты можешь увидеть на рисунке 14.5.5.

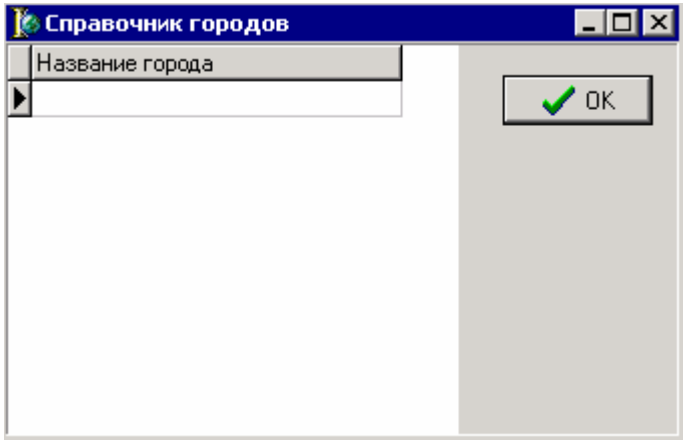


Рисунок 14.5.5 Окно справочника городов

Для большей красоты я ещё добавил на форму кнопки «*Добавить*», «*Сохранить*» и «Удалить» для добавления, удаления и сохранения строк справочника.

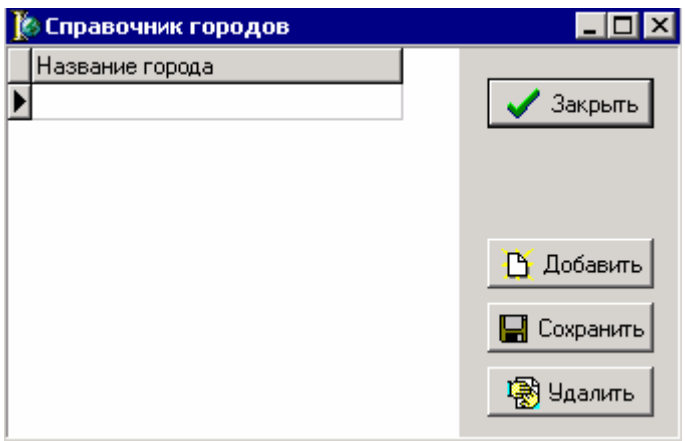


Рисунок 14.5.6 Обновлённая форма справочника

По нажатию кнопки «Добавить» пишем следующий код:

```
procedure TTownBookForm.AddBtnClick(Sender: TObject);
begin
 DataModule1.TownTable.Insert;
 DBGrid1.SetFocus;
end;
```

Метод *Insert* таблицы *TownTable* добавляет новую строку. Во второй строке я вызываю метод SetFocus нашей сетки, чтобы фокус ввода перешёл на него. После нажатия кнопки «Добавить» фокус попадает на неё, но после добавления новой строки, вполне логичным будет перенести фокус на сетку, потому что пользователь будет вводить имя города для новой строки.

По нажатию кнопки «Сохранить» пишем следующий код

```
procedure TTownBookForm.SaveBtnClick(Sender: TObject);
begin
```

```
 if DataModule1.TownTable.Modified then
  DataModule1.TownTable.Post;
end;
```

Если текущая строка претерпела изменения, то в свойстве *Modifies* будет *true*, иначе *false*. А если произошли изменения, то их надо сохранить, иначе если пользователь закроет окно, то данные могут не сохраниться. Для сохранения изменений используется метод *Post*.

По нажатию кнопки «Удалить» пишем следующий код

```
procedure TTownBookForm.DelBtnClick(Sender: TObject);
begin
 DataModule1.TownTable.Delete;
end;
```

Метод *Delete* удаляет текущую строку из таблицы.

Всё, макияж для «*Справочника городов*» закончен. Теперь перейди к главной форме и создай меню или кнопку (я выбрал первое), для вызова справочника городов.

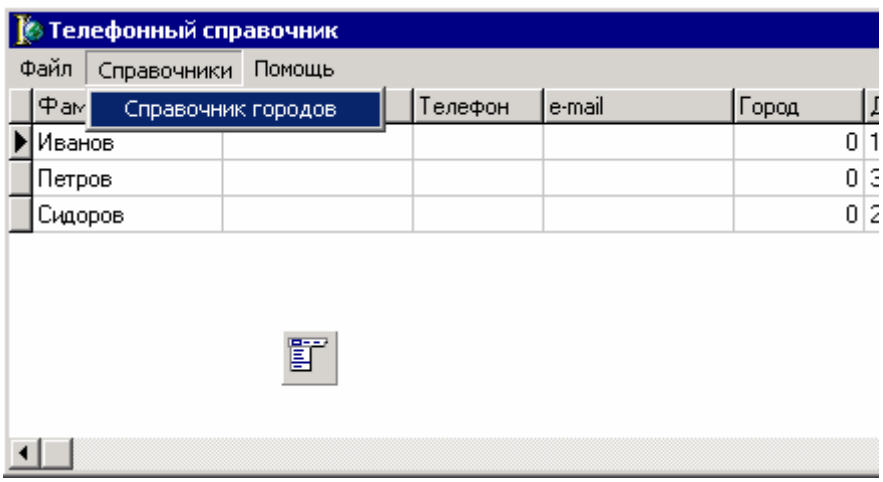
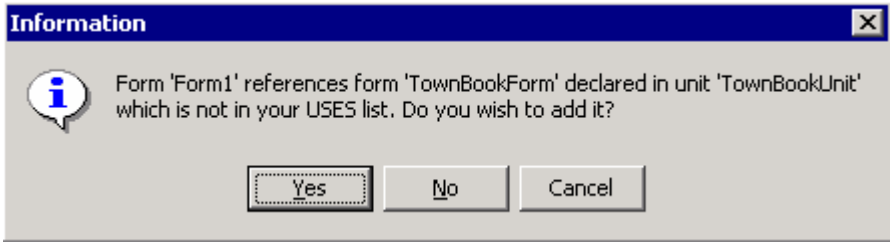


Рисунок 14.5.7 Меню вызова справочника городов

По событию *OnClick* от меню пишем код вызова окна справочника городов:

```
procedure TForm1.TownBookMenuItemClick(Sender: TObject);
begin
 TownBookForm.ShowModal;
end;
```

Если ты не добавил модуль справочника городов к главной форме и попробуешь сейчас откомпилировать проект, то перед тобой появиться следующий запрос:



Здесь говориться о том, что форма *TownBookForm* объявлена в модуле *TownBookUnit* и твоя форма не имеет ссылки на него. Тебе предлагается добавить её автоматически. Выбери *Yes* и модуль будет добавлен автоматически, после этого можно опять компилировать проект не внося никаких изменений. Теперь всё должно пройти удачно. Запусти проект и проверь работу программы.

Запусти программу, вызови «*Справочник городов*» и добавь туда несколько строк. Это будет полезно на будущее, заодно и проверишь правильность работы программы.

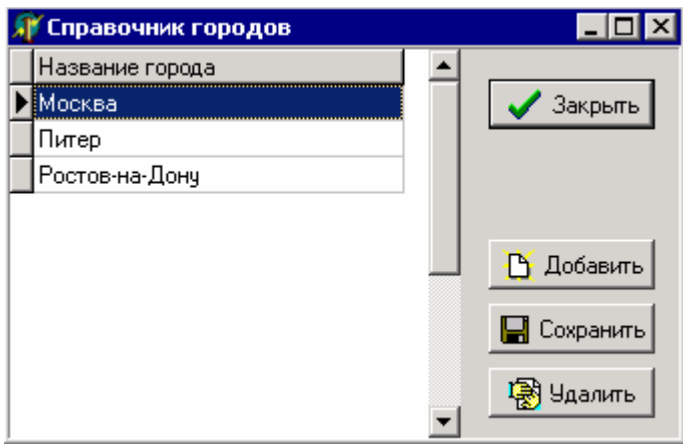


Рисунок 14.5.8 Справочник в действии

Теперь у нас есть справочник городов и мы можем подвязать его данные к основной таблице. Но перед этим немного улучшим форму. Выдели сетку *DBGrid1* на главной форме и в свойстве *Options* отключи возможность редактирования данных в сетке – в dgEditing присвой false:

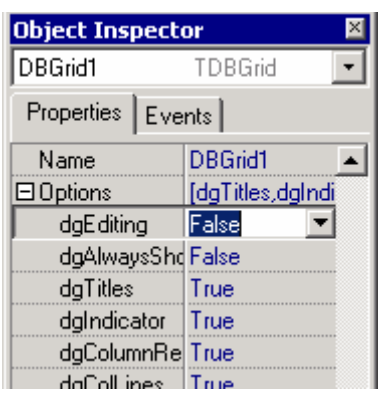


Рисунок 14.5.9 Отключение возможности редактирования

Теперь редактирование данных в сетке невозможно, поэтому мы сделаем для этого отдельные окна. В главном меню создай пункт «*Редактирование*» со следующими подпунктами:

1. Добавить запись;

2. Редактировать запись;
3. Удалить запись.

На рисунке 14.5.10 ты можешь увидеть результат этого меню. Лично я ещё создал панель с кнопками, чтобы к этим командам можно было быстро получить доступ и назначил командам клавиши быстрого вызова.

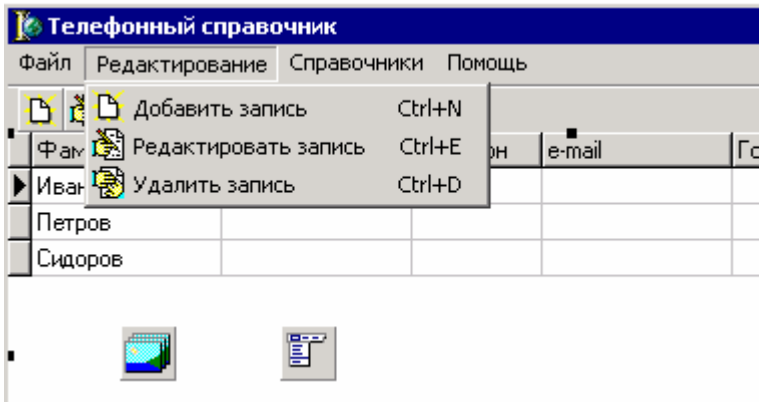


Рисунок 14.5.10 Меню «Редактирование»

Теперь создадим новую форму, которая будет использоваться для редактирования данных каждой записи. Создай форму и сохрани её под именем *EditFormUnit*. Саму же форму назовём *EditRecordForm*. Теперь измени у формы следующие свойства:

BorderStyle *- bsSingle*

Position *– poMainFormCenter.*

Ну и хватит. Этого достаточно, чтобы форма выглядела солидно.

Дальнейшее оформление зависит от твоих пристрастий, а я только покажу самое необходимое. Для начала подключи к новой форме модуль с данными, потому что нам необходимо будет иметь к ним доступ. Для этого выбери из меню *File* пункт *Use Unit*, в появившемся окне выбери *DataModuleUnit* и нажми *OK*.

Теперь посмотри на вид моей формы для редактирования данных (рисунок 14.5.11)

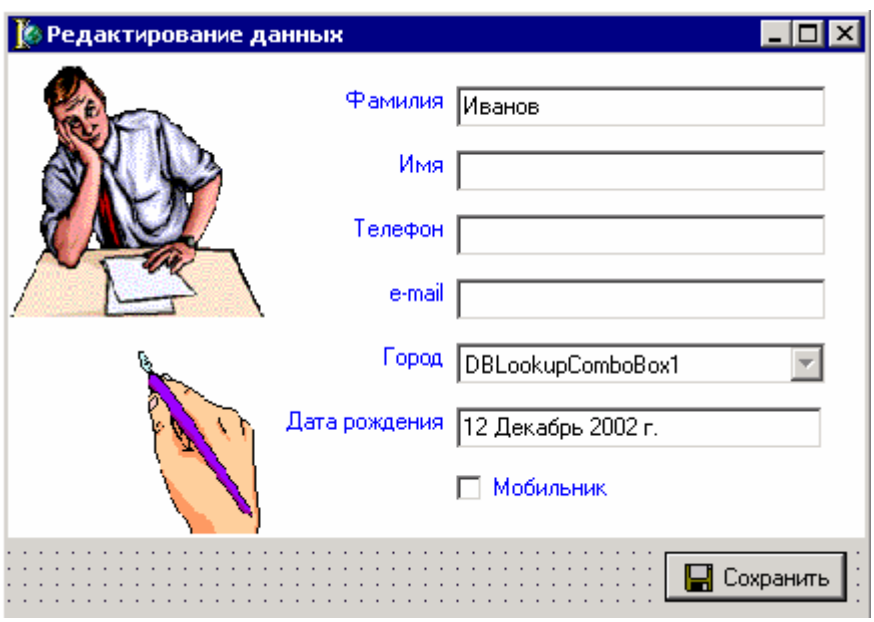


Рисунок 14.5.11 Окно редактирования данных

В этом окне у меня несколько компонентов для украшения вида (картинка, панель белого цвета), но это не главное. Ты можешь повторять и делать подобную форму, а можешь ограничиться только основными компонентами. Основными тут являются – кнопка сохранить, надписи и компоненты доступа к данным.

Напротив надписей *Фамилия*, *Имя*, *Телефон*, *e-mail* и *Дата рождения* находятся компоненты *DBEdit* с закладки *Data Controls*. Эти компоненты представляют собой простые строки ввода типа ***TEdit***, только они умеют автоматически редактировать указанные поля в базе данных. Чтобы компонент видел данные из нужного поля нужно указать у него в свойстве *DataSource* нужную таблицу (*DataModule1.DataSource1*, как мы это делали с сеткой редактирования), а в свойстве *DataField* указать поле, которое надо редактировать. Обязательно попробуй сделать это сам, чтобы подробно разобраться с процессом установки полей.

Для свойства «*Мобильник*» лучше использовать компонент *DBCheckBox*. У него также надо указать поле в таблице, как и у компонентов *DBEdit*.

Самое интересное – поле «*Город*». Названия городов у нас хранятся в отдельном справочнике, а в основной таблице должны храниться только числа – номера строк из справочника городов. Допустим, что у выделенной записи нужно указать город «Москва», который идёт под номером 2 (поле *Key1* для этой строки в справочнике городов равно 2). В этом случае, в основном справочнике в поле «*Город*» нужно указать только цифру 2, а название города в любой момент можно найти в справочнике городов, по полю *Key1*. Так как оно уникально (счётчик), то проблем не возникнет.

Чтобы всё это реализовать достаточно поставить компонент *DBLookupComboBox* с закладки *Data Controls*. Теперь нужно указать у него в свойстве *DataSource* основную таблицу (DataModule1.DataSource1) которая будет редактироваться, а в свойстве *DataField* указать поле, которое надо редактировать – «*Город*».

Компонент *DBLookupComboBox* выглядит как выпадающий список (похож на ***TComboBox***). В качестве элементов выпадающего списка можно указать таблицу. В свойстве *ListSource* нужно указать таблицу, из которой будут браться элементы для выпадающего списка. Давай укажем наш справочник городов - *DataModule1.TownSource*. В свойстве *ListField* укажем поле из этой таблицы, которое будет использоваться для заполнения выпадающего списка – «*Название города*». В свойстве *KeyField* нужно указать поле, значение которого будет вноситься в указанное поле основной таблицы - *Key1*.

Если что-то непонятно, то попробуй перечитать всё заново. Если не поможет, то подожди немного, скоро мы всё увидим на практике.

По нажатию кнопки «*Сохранить*» напиши:

```
procedure TEditRecordForm.BitBtn1Click(Sender: TObject);
begin
 if DataModule1.BookTable.Modified then
  DataModule1.BookTable.Post;
 Close;
end;
```

В первой строке я проверяю, если таблица была изменена (*DataModule1.BookTable.Modified* равна *true*), то принять изменения *DataModule1.BookTable.Post*.

Теперь перейди в основную форму и по нажатию пункта меню «*Добавить запись*» напиши следующее:

```
DataModule1.BookTable.Insert;
EditRecordForm.ShowModal;
```

Здесь в первой строке я вставляю в основную таблицу новую строку. Во второй строке я отображаю окно редактирования данных.

По нажатию пункта меню «*Редактировать запись*» напиши просто код отображения окна редактирования - *EditRecordForm.ShowModal;*.

Теперь запустим программу. Создай новую запись. На рисунке 14.5.12 ты можешь увидеть параметры, которые я забил.

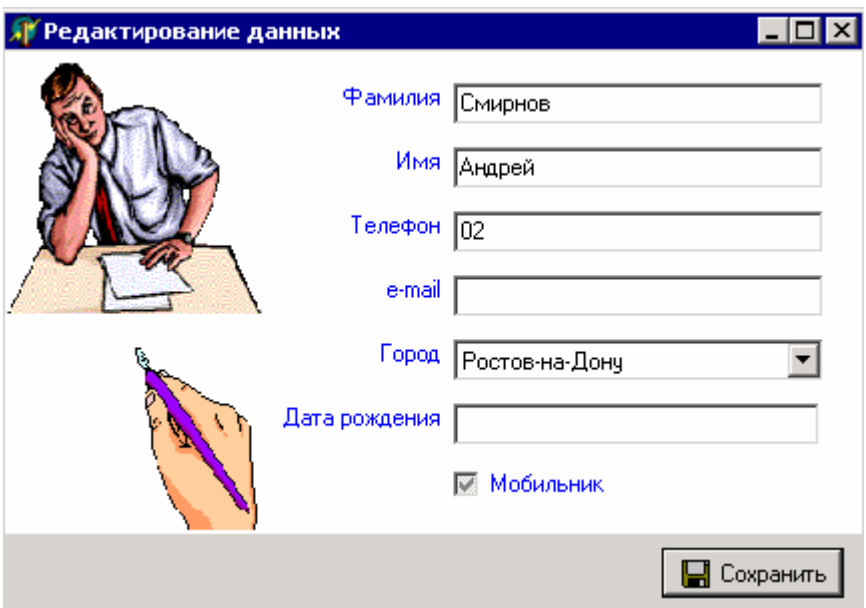


Рисунок 14.5.12 Окно редактирования данных

В поле город я выбрал «Ростов-на-Дону». После нажатия кнопки «*Сохранить*» окно закроется. Посмотри на сетку. В поле «Город» новой строки ты можешь увидеть цифру 3. Это значит, что в справочнике городов есть запись со значением в поле Key1 равным 3 и в поле «Название города» указано название, которое мы выбрали. Давай откроем таблицу через Access и посмотрим:

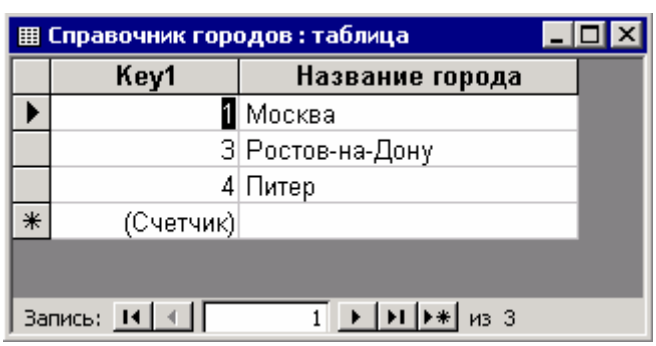


Рисунок 14.5.13 Таблица открытая с Access

Поле с названием «Ростов-на-Дону» действительно имеет в поле Key1 значение 3. Таким образом, в основном справочнике не надо указывать полный текст имени города.

Достаточно только указать нужный ключ справочника городов и мы в любой момент сможем найти название.

Что-то эта глава уже сильно раздулась и пора закругляться. В следующей части мы улучшим пример, а пока попробуй поиграть с уже созданным примером. Попробуй создать несколько строк и потом редактировать их. Обрати внимание, что когда ты открываешь окно редактирования, в компоненте *DBLookupComboBox* отображается правильное название города для указанной записи.

 *На компакт диске, в директории* ***\Примеры\Глава 14\Link*** *ты можешь увидеть пример этой программы.*

14.6 Улучшенный пример с поисковыми полями

Прежде чем двигаться дальше, давай сначала сделаем обработчик для меню «*Удалить запись*». По этому событию нужно вывести запрос на подтверждения удаления и если ответ утвердительный, то можно удалять. Создай такой обработчик и напиши в нём следующее:

```
begin
 if Application.MessageBox(PChar('Ты действительно хочешь удалить '
    +DataModule1.BookTableDSDesigner.AsString), 'Внимание!!!',
     MB_OKCANCEL)=id_OK then
  DataModule1.BookTable.Delete;
end;
```

Здесь я вывожу сообщение уже знакомой функцией *MessageBox*. В первом параметре (текст сообщения) я пишу текст *'Ты действительно хочешь удалить '* плюс значение поля «*Фамилия*» выделенной строки - *DataModule1.BookTableDSDesigner.AsString*. Самое сложное здесь - *DataModule1.BookTableDSDesigner.AsString*. Чтобы понять эту конструкцию перейди в модуль *DataModule*. Здесь щёлкни дважды по компоненту *BookTable*, где у нас подключён основной справочник, и затем по полю «Фамилия». Посмотри в объектном инспекторе имя этого поля, оно должно быть *BookTableDSDesigner*. Теперь ясно? Я пишу имя модуля данных (*DataModule1*), затем через точку имя поля (*BookTableDSDesigner*) и метод *AsString*, который возвращает значение поля в виде строки.

Можешь подняться в раздел ***type*** модуля *DataModule1* и убедиться, что внутри нашего объекта ***TDataModule1*** есть объявление свойства *BookTableDSDesigner* типа ***TWideStringField***.

Надеюсь, что с первой строкой теперь всё ясно. Если нет, то запусти пример и посмотри, что произойдёт если попытаться удалить строку.

Во второй строке я просто удаляю текущую строку с помощью вызова метода *Delete* таблицы - *DataModule1.BookTable.Delete*.

Теперь пример практически готов. Единственный недостаток – в сетке просмотра данных вместо названия города отображается индекс строки в справочнике. Это очень неудобно, поэтому давай исправим этот недостаток.

Перейди в модуль *DataModule1*, и выдели компонент *BookTable*. Сделай его неактивным – в свойстве *Active* установи *false*. Теперь дважды щёлкни по этому компоненту и перед тобой откроется уже знакомый редактор полей Давай создадим новое поле, которое будет содержать текстовое название города для строк таблицы. Для этого

щёлкни внутри окна редактора и в появившемся меню выбери пункт *New Field*. Перед тобой должно открыться окно, как на рисунке 14.6.1.

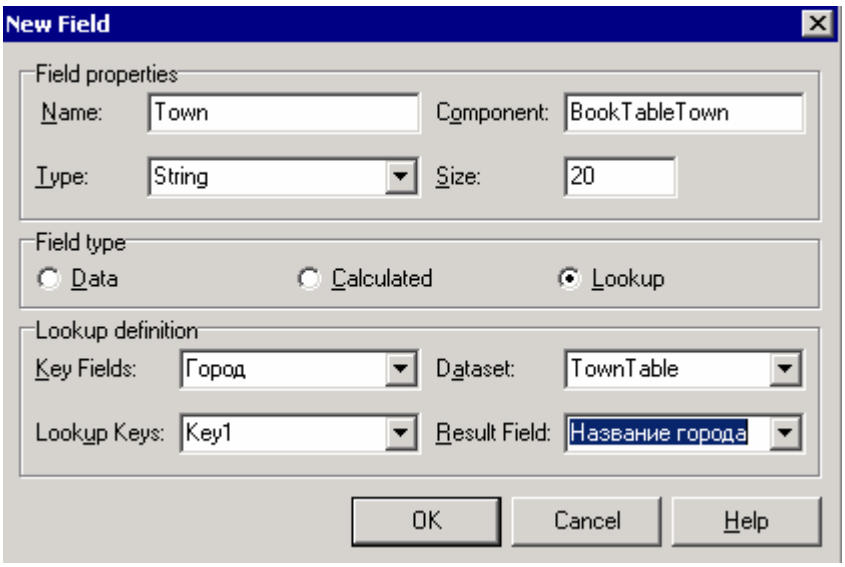


Рисунок 14.6.1 Окно создания нового поля

Создание нового поля возможно только при неактивной таблице, поэтому мы и выставили в свойстве Active значение false.

Заполни поля этого окна следующим образом:

В поле **Name** введи «*Town*»

В поле **Type** укажи тип *String* – строка.

В поле **FieldType** выбери *Lookup* – поисковое поле.

В поле **KeyField** (ключевое поле) выбери поле «Город». Это поле основной таблицы, по значению которого надо будет искать текст в другой таблице.

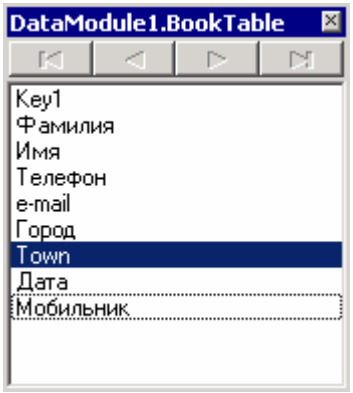
В поле **DataSet** надо указать *TownTable* – это таблица-справочник городов, где нужно искать.

В поле **Lookup Keys** укажи *Key1* – это поле в таблице справочнике, по которому надо искать.

В поле **Result Field** укажи поле «*Название города*» - это поле, текст которого будет подставляться.

Теперь нажми ОК.

В окне редакторе полей появиться новое поле с именем *Town*. В самой базе данных такого поля не будет, потому что оно динамическое и существует только в памяти машины, когда программа запущена. Перетащи его мышкой повыше, ближе к полю *Город*.



Снова сделай таблицу *BookTable* активной и попробуй теперь запустить программу. Посмотри на поле *Town* и ты увидишь теперь там тестовое название города (рисунок 14.6.2).

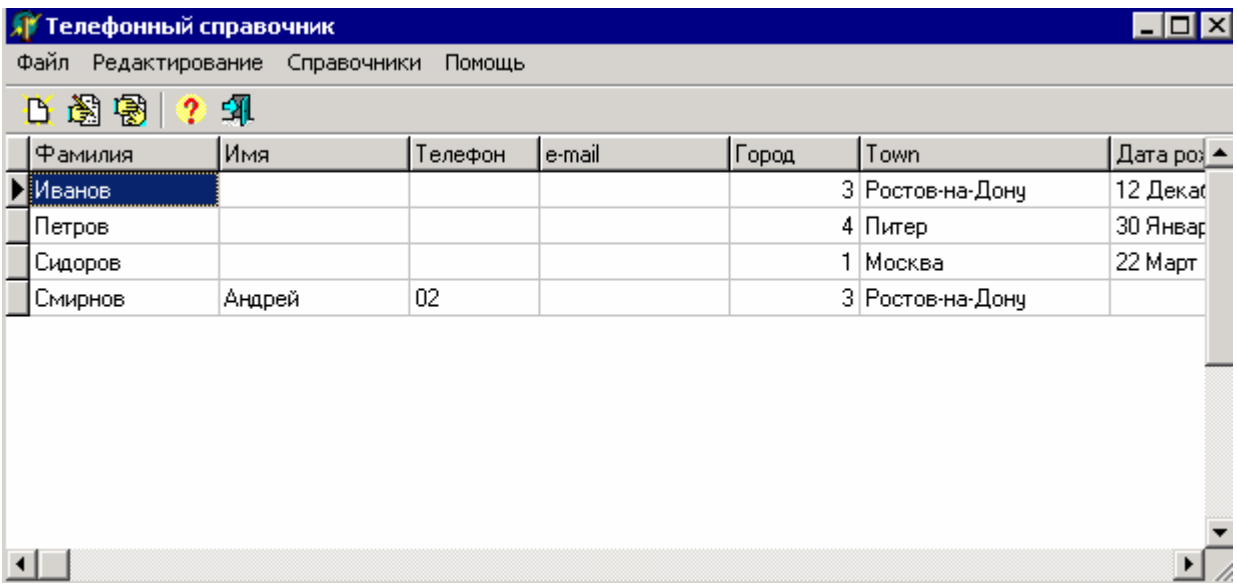


Рисунок 14.6.2 Результат работы программы

Теперь программа выглядит намного красивее и интереснее. Единственное – можно сделать поле «*Город*» невидимым, чтобы пользователь не видел эти непонятные числа, а над полем Town написать надпись «*Город*». Для этого дважды щёлкни по компоненту *BookTable*, выдели поле «*Город*» и установи в свойстве *Visible* значение *false*. Теперь выдели поле *Town* и в свойстве DisplayLabel напиши «*Город*».

Всё!!! С поисковыми полями покончено, можно двигаться дальше, глубже, шире :).

На компакт диске, в директории ***\Примеры\Глава 14\Link1*** *ты можешь увидеть пример этой программы.*

14.7 Сортировка

Наш телефонный справочник уже достаточно хорош. В него можно добавлять новые записи, редактировать или удалять существующие. Но было бы удобней научить нашу программу сортировать записи по определённому

полю. Допустим, что тебе надо отсортировать данные по полю «Фамилия» - как это сделать? Очень просто, для этого существуют индексные поля.

В любой базе данных существует понятие индексного поля. Мы пока создавали только один индекс – главный для поля счётчика. Это обязательный индекс и существует всегда, но ты можешь создавать любое количество дополнительных индексов. Но это не значит, что надо все поля базы данных сделать индексными, ведь индексирование отнимает дополнительное место на диске и если переборщить, то можно наоборот сделать хуже. Поэтому нужно находить золотую середину.

Какие же поля индексировать? Я советую это делать только с теми полями, по которым будет чаще всего происходить поиск. В телефонном справочнике чаще ищут по номеру телефона или по фамилии. В домашнем справочнике это делают чаще по фамилии, а если у тебя будет справочник всего города, то возможно, что чаще будут искать по телефону или даже по адресу. Ну представь себе записную книжку с телефонами твоих друзей. По каким параметрам ты будешь искать номер своего друга? Ну конечно же по фамилии, ведь ты её знаешь, а номер телефона можешь забыть. Вот именно поэтому в нашем справочнике желательно сделать поле «*Фамилия*» индексным.

Индексы увеличивают скорость поиска данных и позволяют сортировать все записи. Если первое практически невозможно увидеть на примерах моей книги (потому что количество записей у нас очень маленькое, а для ощущения скорости нужно большое количество данных), то сортировку можно ощутить без проблем.

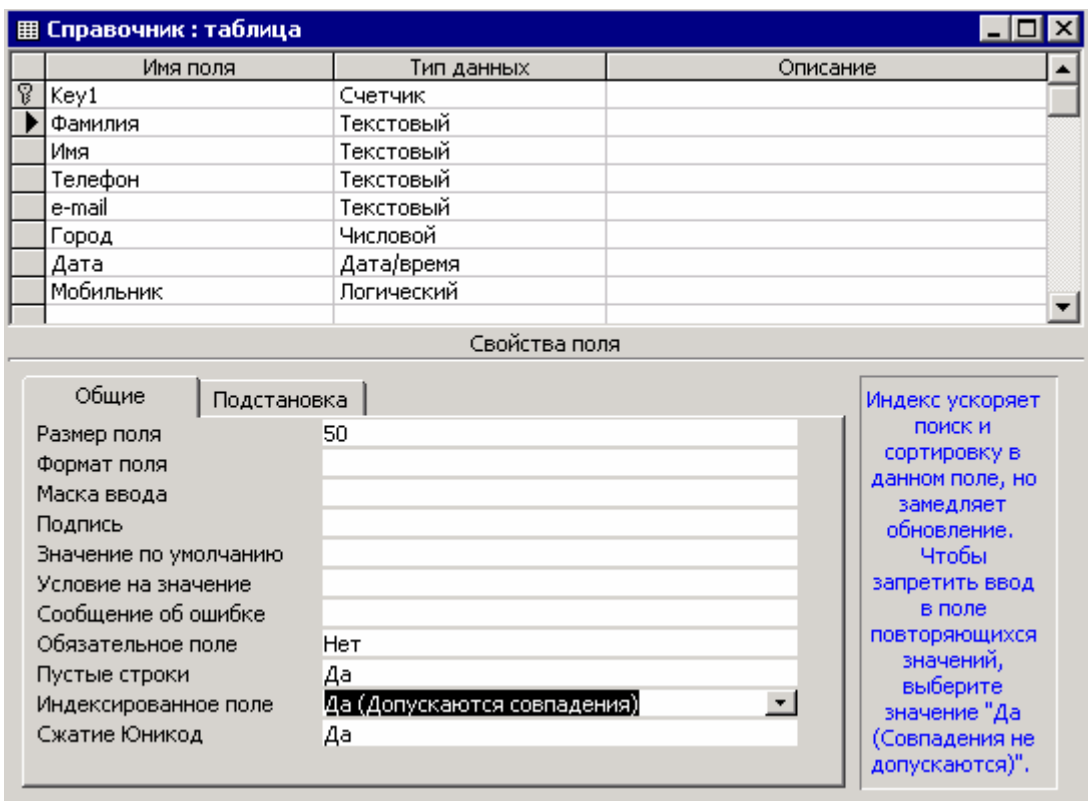


Рисунок 14.7.1 Редактирование свойств полей

Открой нашу базу данных в Access, выдели таблицу «*Справочник*» и нажми кнопку «*Конструктор*». Перед тобой откроется окно редактирование свойств полей (рисунок 14.7.1). Здесь выдели поле «*Фамилия*» и посмотри на свойство «*Индексированное поле*». Свойство имеет выпадающий список для выбора нужного параметра. Тебе доступны три варианта:

1. *Нет* – поле не индексировано.

2. *Да (Допускаются совпадения)* – поле индексировано и две (или более) записи могут иметь одно и то же значение. Это значит, что у тебя может быть две записи, где в поле «*Фамилия*» указано значение «*Сидоров*». Для нашего случая это идеальный вариант, потому что много людей могут быть под одной и той же фамилией.

3. *Да (Совпадения не допускаются)* – если выбран этот параметр, то в проиндексированном поле нельзя хранить одинаковое значение в разных записей. База данных будет сама следить за уникальность поля. Если выбрать этот параметр, то две записи в нашем справочнике не смогут иметь значение «*Сидоров*». Этот параметр очень удобен, когда тебе нужно, чтобы поле было действительно уникальным. Такое бывает в офисных приложениях, например, номер документа часто должен быть уникальным.

Давай сделаем два поля индексными: «Фамилия» и «Телефон». У обоих нужно выбрать в свойстве «*Индексированное поле*» параметр «*Да (Допускаются совпадения)*».

Всё, закрываем базу данных и переходим к программированию. Загружай в Delphi пример, который мы написали в прошлой части, чтобы мы могли добавить к нашему справочнику возможность сортировки.

Для начала улучшим нашу форму. Для этого добавь в меню нашей программы пункт «*Сортировка*» и два его подпункта «*По фамилии*» и «*По телефону*» (рисунок 14.7.2).

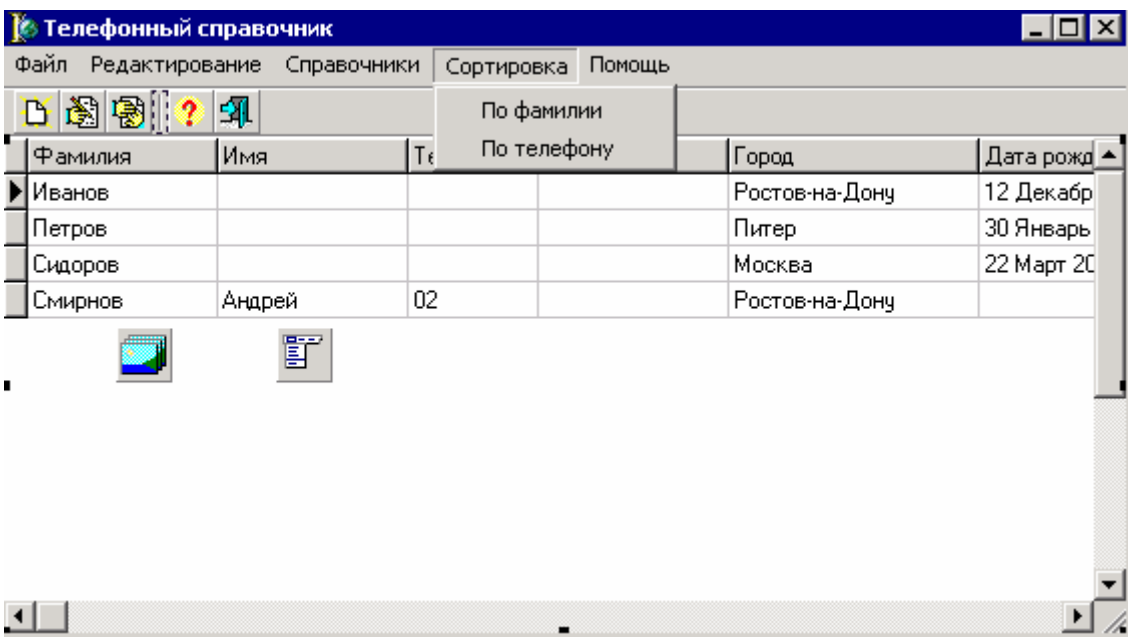


Рисунок 14.7.2 Улучшенная форма нашей программы

По нажатию пункта меню «*По фамилии*» напиши следующий код:

```
procedure TForm1.N8Click(Sender: TObject);
begin
 DataModule1.BookTable.IndexFieldNames:= 'Фамилия';
end;
```

По нажатию пункта меню «*По телефону*» напиши следующий код:

```
procedure TForm1.N8Click(Sender: TObject);
begin
```

```
 DataModule1.BookTable.IndexFieldNames:= 'Телефон';
end;
```

В обоих случаях я присваиваю свойству *IndexFieldNames* таблицы *BookTable* значение поля, по которому нужно сортировать записи.

 *На компакт диске, в директории* ***\Примеры\Глава 14\Index*** *ты можешь увидеть пример этой программы.*

14.8 Фильтрация данных

Наш телефонный справочник достаточно быстро набирается новыми возможностями, но в нём до сих пор нет самого главного – поиска. Представь себе, что у тебя в базе данных находиться телефоны ста человек. Как ты будешь искать телефон определённого человека? А если в базе данных будет 1000 записей? Без возможности поиска тут очень тяжело.

Для поиска в компоненте TADOTable есть свойство *Filter*. В нём можно указывать условие, по которому будут отображаться данные. Например, ты можешь указать там отображение только записей, в которых поле «*Фамилия*» содержит значение «*Сидоров*». Но для того, чтобы фильтр заработал, надо ещё установить свойство *Filtered* нашей таблицы в *true*. После этого можно изменять свойство *Filter* и все изменения сразу же будут вступать в силу.

Свойство *Filter* – это строка. В ней нужно писать текст условия в виде:

Поле [Оператор сравнения] ‘Значение’

Например, если ты хочешь отобразить все записи, в которых поле «*Фамилия*» равно значению «Сидоров», то нужно написать:

AdoTable1.Filter:=’Фамилия=’’Сидоров’’’;

Обрати внимание, что значение нужно указывать в одинарных кавычках. Но так как одинарные кавычки используются в Delphi для ограничения строк, то тут приходиться немного включить соображение. Чтобы внутри строки поставить одинарную кавычку, её нужно поставить дважды:

‘После этого текста будет одинарная кавычка’’ это продолжение текста’

Именно таким способом я ставлю перед значением одинарную кавычку. После значения мне нужно поставить одинарную кавычку и закрыть строку, поэтому я ставлю три одинарных кавычки (две для того, чтобы поставить кавычку для значения и одна для конца строки).

Это был пример простейшего условие на равенство. Ты можешь использовать любые другие операторы сравнения (больше или меньше). А можно даже создавать составные операторы сравнения, в которых сравнивается сразу два или более значений. Например:

AdoTable1.Filter:=’Фамилия=’’Сидоров’’ or Телефон=’’3326523’’’;

В этом пример я ищу все записи, в которых поле «*Фамилия*» равно значению «*Сидоров*» и поле «*Телефон*» имеет значение «*3326523*». Для объединения двух условий используется оператор **or**. Можно также использовать оператор логического «и», т.е. **and**.

При программировании фильтров будь внимателен к кавычкам. Помни, что значения должны выделяться одинарными кавычками и внутри строки для этого приходиться ставить две одинарных кавычки, а не одну двойную!!!

Теперь переходим к программированию. Открывай пример из предыдущей части и будем добавлять к нему возможность поиска.

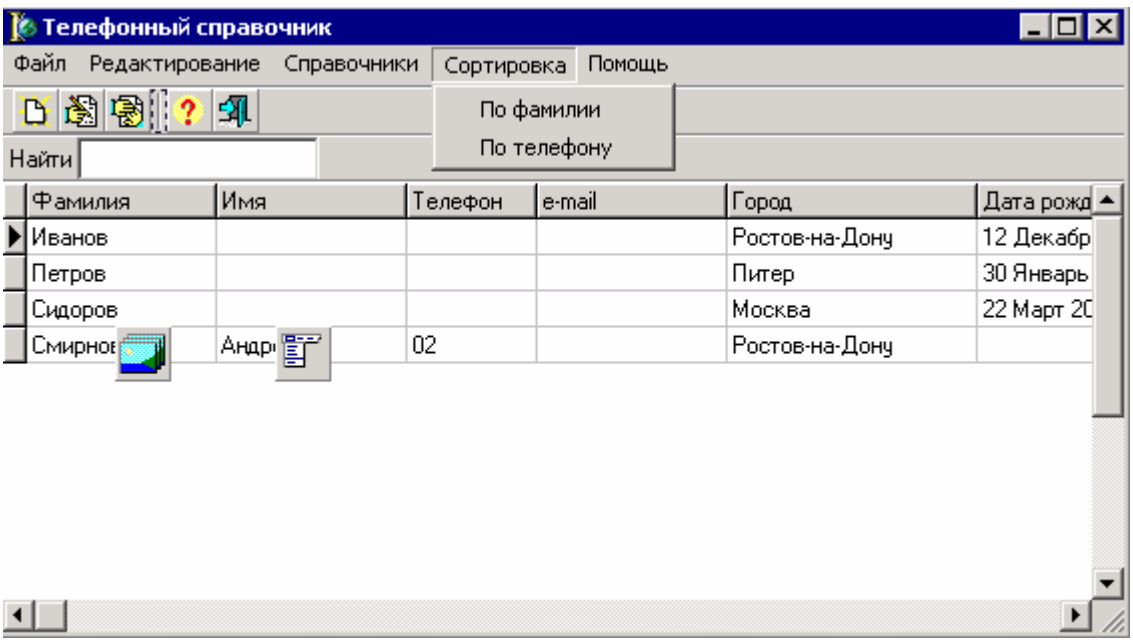


Рисунок 14.8.1 Улучшенная форма нашей программы

Для начала улучшим нашу форму. Для этого добавь панельку, на которой будет располагаться текст «*Найти*» и строка ввода ***TEdit*** с именем *FindEdit* (рисунок 14.8.1). Теперь создай обработчик события *OnChange* для строки ввода. Когда пользователь изменил текст в строке ввода, мы должны изменить и фильтр. Напиши в этом обработчике следующий код:

```
procedure TForm1.FindEditChange(Sender: TObject);
begin
 if Length(FindEdit.Text)>0 then
  DataModule1.BookTable.Filtered:=true
 else
  DataModule1.BookTable.Filtered:=false;

 DataModule1.BookTable.Filter:='Фамилия>'''+FindEdit.Text+'''';
end;
```

Вначале я проверяю, если в строке поиска что-то есть, то включаю фильтр иначе его можно отключить, чтобы показать всю таблицу. После этого создаю условие фильтра: 'Фамилия>'"+FindEdit.Text+"". Я здесь использую знак больше, чтобы отображать все похожие записи на введённый текст. Если установить знак равенства и пользователь введёт букву «*с*», то в таблице ничего отображаться не будет, потому что нет такой фамилии «*с*». А при знаке «*больше*» будут отображаться все фамилии начинающиеся на букву «с».

В конце фильтра я добавляю четыре одинарных кавычки. Почему четыре? Да потому что после значения нам надо добавить одну кавычку. Чтобы её добавить надо добавить строку, содержащую кавычку. Поэтому у меня стоит две одинарных кавычки чтобы открыть и закрыть эту строку. Внутри строки я ставлю эту кавычку, а как ты помнишь, для этого надо ставить две кавычки и в результате получиться одна. Вот так и получается 4 одинарных кавычки.

Вот так мы получили простую возможность поиска. Более эффективные возможности можно получить используя язык SQL – это язык запросов к базам данных. Но это отдельная история и требует отдельного рассмотрения.

 *На компакт диске, в директории* ***\Примеры\Глава 14\Filter*** *ты можешь увидеть пример этой программы.*

14.9 Язык запросов SQL

SQL переводят на русский как Структурированный Язык Запросов . С помощью SQL-запросов можно создавать и работать с реляционными базами данных. Этот язык стал стандартом, поэтому если ты хочешь работать с базами данных, то ты должен знать этот язык как каждую дырку в своих зубах.

SQL определяется Американским Национальным Институтом Стандартов и Международной Организацией по стандартизации (ISO) . Несмотря на это, некоторые производители баз данных вносят изменения и дополнения в этот язык. Эти изменения незначительны и основа остаётся совместимой со стандартом.

Что такое реляционная база данных? Это таблица, в которой в качестве столбцов выступают поля данных, а каждая строка хранит данные. В каждой таблице должно быть одно уникальное поле, которое однозначно будет идентифицировать строку. Это поле называется ключевым. Эти поля очень часто используются для связывания таблиц или для обеспечения уникальности каждой записи. Но даже если у тебя таблица не связана, ключевое поле всё равно обязательно. Представь, что ты пишешь телефонную базу данных. Сколько у тебя будет "Ивановых"? Как ты будешь отличать их? Вот тут тебе поможет ключ. В качестве ключа желательно использовать численный тип и если позволяет база данных, то будет лучше, если он будет типа "autoincrement" (автоматически увеличивающееся/уменьшающееся число).

Столбцы в базе данных, также должны быть уникальными, но в этом случае не обязательно числовыми. Их можно называть как угодно, лишь бы было уникально и тебе понятно, а остальное никого не касается.

SQL может быть двух типов: интерактивный и вложенный . Первый - это отдельный язык, он сам выполняет запросы и сразу показывает результат работы. Второй - это когда SQL язык вложен в другой, как например в C++ или Delphi.

Интерактивный SQL более близок к стандартному, а во вложенном очень часто встречаются отклонения и дополнения. Например, в стандартном SQL различаются только два типа данных: строки и числа, но некоторые производители добавляют свои

типы (Date, Time, Binary и т.д.). Числа в SQL делятся на два типа: целые (INTEGER или INT) и дробные (DECIMAL или DEC). Строки ограничены размером в 254 символа.

Более подробную информацию о SQL ты найдешь в документе «*Язык запросов SQL.doc*» на диске к книге в директории «*Документация*». В этом документе ты найдёшь достаточно подробное описание языка SQL, поэтому прежде чем двигаться дальше советую прочитать этот документ. Особенно если ты хочешь в дальнейшем профессионально писать программы для работы с базами данных.

Давай посмотрим, как можно направить базе данных простейший SQL запрос. В качестве примера я реализую возможность поиска записей по номеру телефона в нашем телефонном справочнике.

Для отправки запросов базе данных используется компонент ***TADOQuery*** с закладки ADO палитры компонентов. Работа этого компонента схожа с таблицей ***TADOTable*** и у них даже много схожих полей. В ***TADOQuery*** ты также должен выбирать строку подключения (свойство *ConnectionString*) или связываться с компонентом ***TADOConnection*** через свойство Connection. У запросов даже есть возможность установки фильтра (свойства *Filtered* и *Filter*). Но мы его рассматривать не будем для экономии места, потому что работа с этим фильтром ничем не отличается от фильтра таблиц ***TADOTable*** который мы рассматривали в главе 14.8.

Давай откроем наш телефонный справочник и дополним его новыми возможностями.

Открой модуль данный *DataModule*, где у нас расположены все компоненты доступа к базе данных. Добавь сюда компонента ***TADOQuery*** (назовём его *FindQuery*) и компонент ***TDataSource*** (назовём его *FindSource*). Теперь надо связать эти компоненты, указав у компонента *FindSource* в свойстве *DataSet* компонент *FindQuery*.

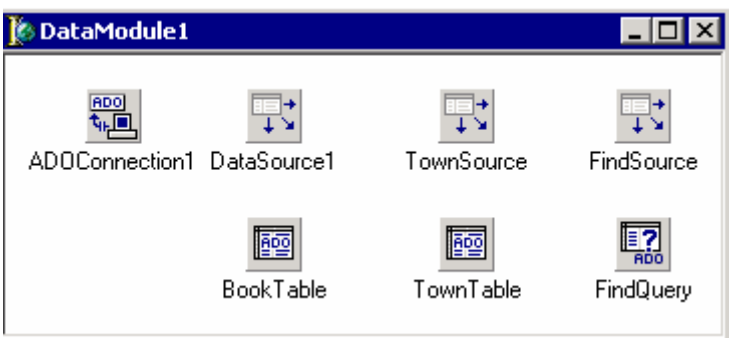


Рисунок 14.9.1 Модуль DataModule

Напоминаю, что ***TDataSource*** отвечает за отображение данных из таблиц. Компонент ***TADOQuery*** предназначен для отправки SQL запросов базе данных. Результат запросов возвращается в виде таблиц и для отображения результата нам будет необходим компонент ***TDataSource***. Именно поэтому мы их установили на форму и связали между собой, чтобы компонент отображения видел данные, которые надо отображать.

Теперь выдели компонент *FindQuery*, здесь нам необходимо указать в свойстве *Connection* наш компонент подключения к базе данных *ADOConnection1*. Этим мы укажем компоненту, какой базе будут отправляться запросы.

Теперь напишем сам запрос. Для этого дважды щёлкни по свойству *SQL* и перед тобой откроется окно редактора запросов (рисунок 14.9.2).

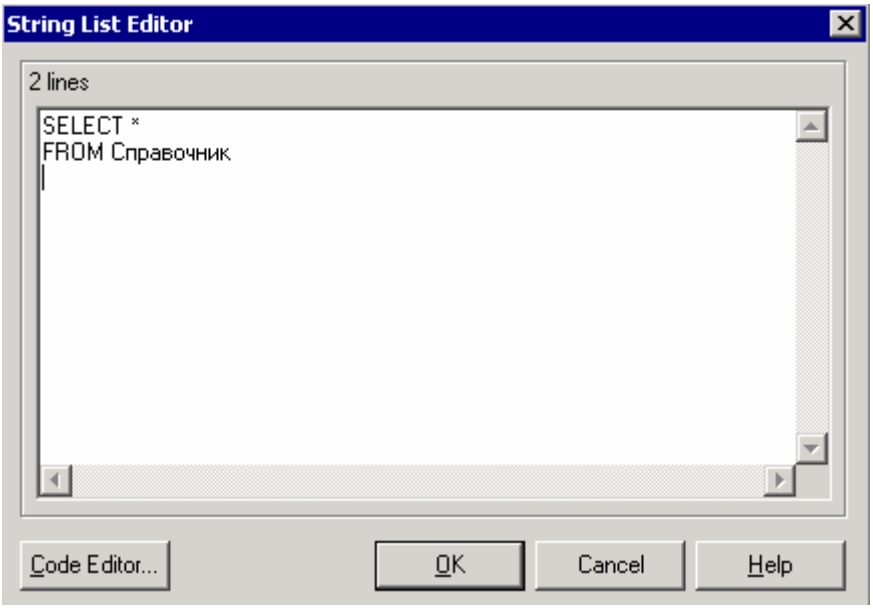


Рисунок 14.9.2 Редактор SQL запросов

В этом редакторе я написал простейший запрос – выбор всех строк и всех столбцов из таблицы «*Справочник*» базы данных, к которой мы подключились:

```
SELECT *
FROM Справочник
```

Мы пока только написали запрос, но ещё не выполнили его. Для его выполнения нужно установить свойство *Active* в *true*. Сделай это.

Теперь перейдём в главный модуль, где у нас храниться основное окно. Выдели сетку *DBGrid1*, которая у нас отображает данные из таблицы *BookTable*. Перейди в объектный инспектор и измени свойство *DataSource* на *DataModule1.FindSource*, чтобы увидеть таблицу результата запроса.

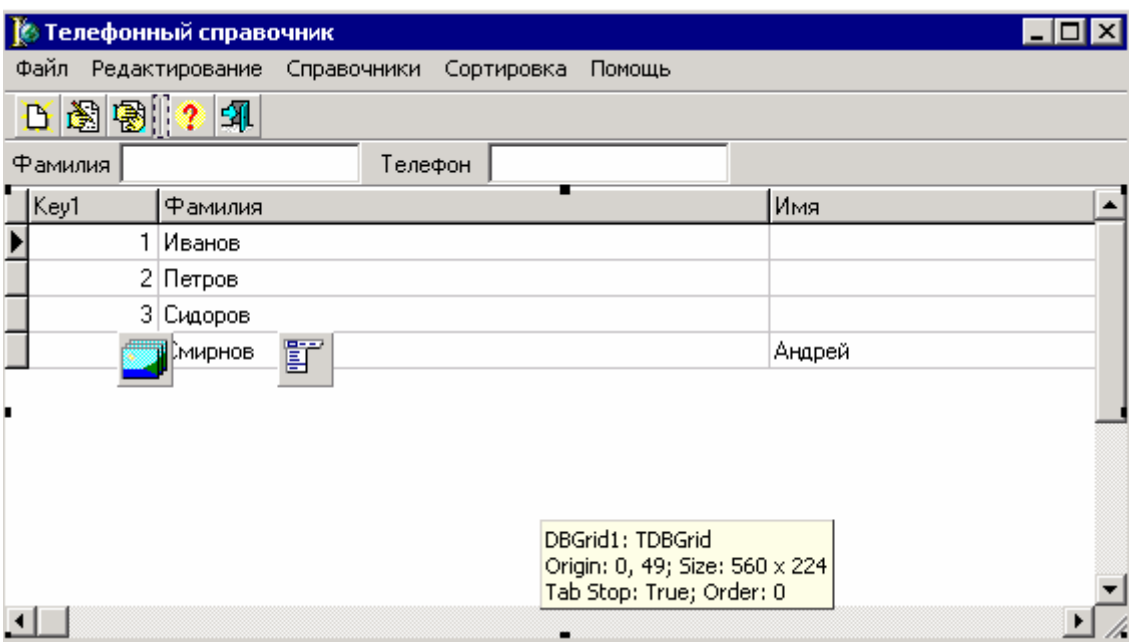


Рисунок 14.9.3 Результат запроса

Посмотри на результирующую таблицу, она похожа на то, что мы видели при работе с компонентом ***TADOTable***, только пока что ты видишь все поля (даже те, которые мы уже скрыли из виду от пользователя в компоненте *BookTable*) и отображение полей не настроено. Но всё это можно исправить, если дважды щёлкнуть по компоненту *FindQuery*. Перед тобой откроется то же самое окно настройки свойств полей. Сейчас мы сделаем это, но сначала снова выдели сетку *DBGrid1*, в главном окне. Перейди в объектный инспектор и измени свойство *DataSource* на *DataModule1.DataSource1*, чтобы вернуть всё на родину.

А вот теперь переходи в модуль данных *DataModule* и дважды щёлкни по компоненту *FindQuery*. В появившемся окне щёлкни правой кнопкой мыши и выбери пункт меню «*Add all fields*». В редакторе должны отобразиться все поля нашей таблицы. Теперь дважды щёлкните по компоненту *BookTable*, чтобы отобразить окно свойств нашей уже настроенной таблицы. Расположи оба окна рядом с друг другом, чтобы ты мог всегда их видеть. Теперь выделяй первое свойство в редакторе свойств таблицы *BookTable*, запоминай их, переходи в редактор свойств полей запроса и делай там те же настройки.

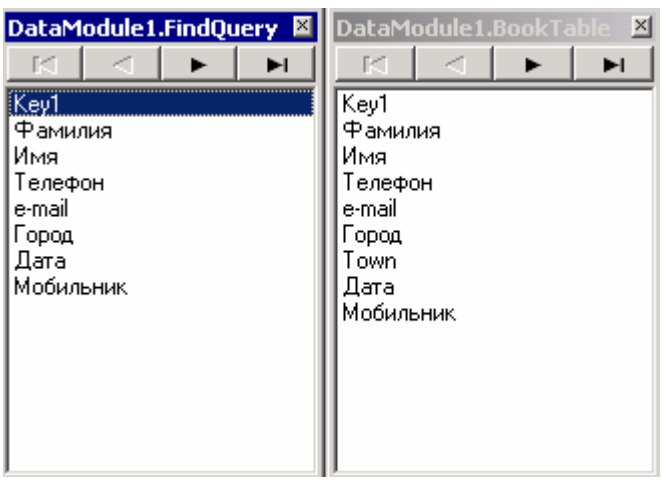


Рисунок 14.9.4 Слева свойства полей запроса, справа свойства полей таблицы

Когда перенесёшь все свойства (не забудь создать поисковое поле для поля «*Город*»), можешь снова посмотреть на результат используя сетку главного окна. Только опять верни всё на родину. Теперь результат вообще не должен отличаться.

Теперь реализуем непосредственно поиск с помощью нашего SQL запроса. Для этого создадим новую форму, в которой будет отображаться результат и назовём её *FindResultForm*. Теперь измени следующие свойства:

Caption – измени на «*Результат поиска*»;

Position – установи в *poMainFormCenter*, чтобы наше окно отображалось по центру главного окна.

Чтобы окно видело таблицы, к нему надо подключить наш модуль данных – *DataModuleUnit*. Для этого используй уже знакомое меню *File->Use Unit*. Теперь брось на форму сетку *DBGrid* с закладки *Data Controls* палитры компонентов и растяни её по всей форме. Перейди в объектный инспектор и измени свойство *DataSource* на *DataModule1.FindSource*, чтобы увидеть в сетке результат запроса.

Всё, можно сохранять форму под именем *FindResultUnit*. Внешний вид моего окна ты можешь увидеть на рисунке 14.9.5.

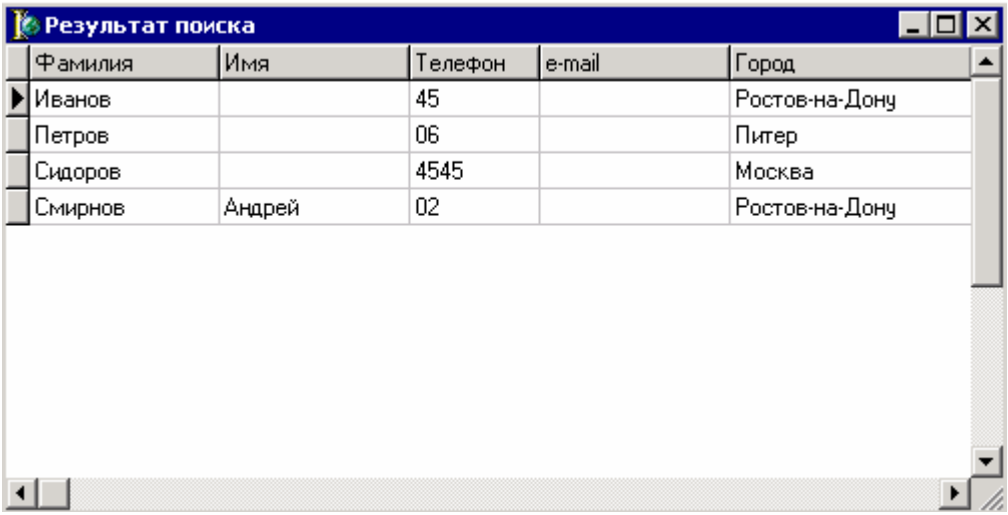


Рисунок 14.9.5 Окно результата поиска.

Теперь перейди на главную форму и на панели поиска добавь надпись и строку ввода для поиска по телефону. Самой последней добавь кнопку, по нажатию которой будет запускаться поиск. На рисунке 14.9.6 ты можешь видеть улучшенное главное окно нашей программы.

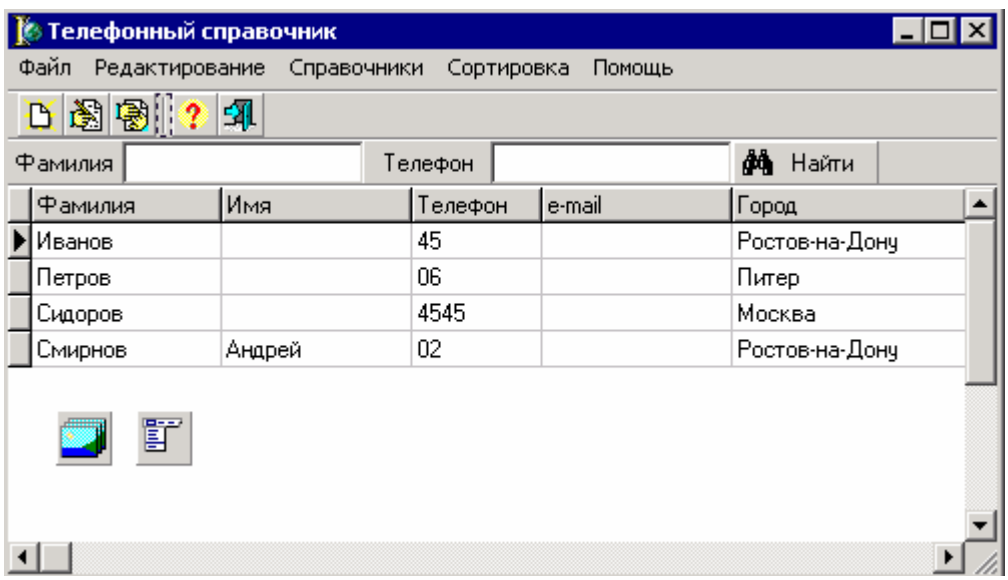


Рисунок 14.9.6 Улучшенная панель поиска.

По нажатию кнопки «*Найти*» пишем следующий код:

```
procedure TMainForm.FindButtonClick(Sender: TObject);
begin
 DataModule1.FindQuery.Active:=false;
 DataModule1.FindQuery.SQL.Clear;
 DataModule1.FindQuery.SQL.Add('SELECT *');
 DataModule1.FindQuery.SQL.Add('FROM Справочник');
 DataModule1.FindQuery.SQL.Add('WHERE Телефон LIKE "'+FindTelephoneEdit.Text+'"');
 DataModule1.FindQuery.Active:=true;

 FindResultForm.ShowModal;
end;
```

В первой строке кода я делаю компонент запроса неактивным. После этого мне надо заполнить свойство *SQL* запросом на поиск данных. Это свойство имеет уже знакомый тип ***TStrings***, который мы использовали при работе с элементами списка ***TListBox***, ***TComboBox*** и др. Но прежде чем заполнять новыми значениями, нужно очистить свойство от старого запроса, который мог остаться после последнего вызова (если пользователь уже нажимал кнопку «*Найти*», то там будет старый запрос, который при следующем нажатии надо очистить). Для очистки вызываем метод *Clear*.

Далее идёт заполнение свойства *SQL* текстом запроса. Нам нужно внести следующий запрос:

```
SELECT *
FROM Справочник
WHERE Телефон LIKE ‘’’+FindTelephoneEdit.Text+’’’’
```

Здесь написано примерно следующее: выбрать все поля из таблицы «*Справочник*», где поле «*Телефон*» равно указанному в компонент *FindTelephoneEdit* тексту. Обрати внимание, что параметр, с которым я сравниваю (текст из *FindTelephoneEdit*) должен быть заключён в кавычки. Поэтому как и при работе с фильтром, мне приходиться использовать множество одинарных кавычек.

Как только текст запроса занесён в свойство *SQL*, его можно выполнять. Для этого я делаю компонент активным. Ну и чтобы отобразить результат, я показываю окно *FindResultForm*.

Для выполнения запроса чаще всего достаточно сделать компонент ADOQuery активным. Это прекрасно работает, если ты запрашиваешь данные из таблицы базы данных. Но если в запросе ты удаляешь строки или изменяешь структуру таблицы (в запросе есть такие операторы как INSERT, UPDATE, DELETE, или CREATE TABLE), то необходимо вызывать метод ExecSQL компонента ADOQuery.

На компакт диске, в директории ***\Примеры\Глава 14\SQL1*** *ты можешь увидеть пример этой программы.*

В принципе пример готов, и на этом можно было бы остановиться, но я хочу показать тебе ещё один вариант использования динамических запросов. Что я понимаю под словом «динамический»? Если мы просто вписали запрос в свойство SQL компонента *ADOQuery* и в течении всей программы его не изменяем, то такой запрос можно назвать статическим. Но если в течении выполнения программы нам надо изменять текст запроса, то его можно назвать динамическим.

При поиске записей нам приходиться изменять запрос, потому что каждый раз используется разный параметр, который необходимо найти. Но зачем же каждый раз изменять весь запрос, когда он не меняется? Не легче ли внести в запрос переменную и изменять только её? Легче, поэтому давай подкорректируем наш пример.

Выдели компонент *FindQuery* и дважды щёлкни по свойству *SQL*. В редакторе запроса введи следующий запрос:

```
SELECT *
FROM Справочник
WHERE Телефон LIKE :Telephone
```

В принципе, здесь написан практически тот же запрос, что мы уже использовали. Единственная разница – вместо параметра *FindTelephoneEdit.Text* стоит *:Telephone*. Что это такое? Это переменная, о чём говорит двоеточие вначале имени. Переменная в *SQL* запросе оформляется как:

```
:ИмяПеременной
```

Закрой окно редактора запроса и дважды щёлкни по свойству *Parameters*. Перед тобой откроется окно редактора параметров (рисунок 14.9.7). Как видишь, описанный в запросе параметр автоматически попадает сюда. Выдели параметр Telephone и посмотри на его свойства в объектном инспекторе.

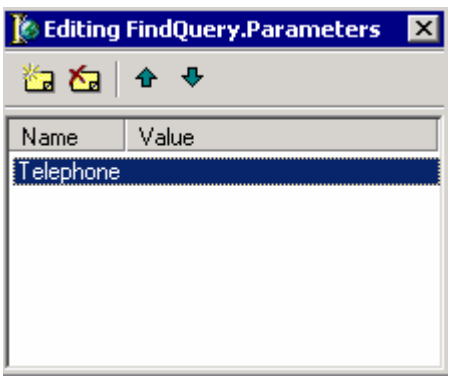


Рисунок 14.9.7 Редактор параметров.

В свойстве *DataType* ты должен указать тип переменной. В нашем случае будет строка с номером телефона, поэтому выбери из списка тип *ftString*. В свойстве *Value* ты можешь указать значение по умолчанию. Попробуй указать там любой номер телефона из твоей базы и сделать компонент *FindQuery* активным. Теперь открой окно, которое должно отображать результат поиска и посмотри на сетку, в которой уже отображаться результат поиска.

Ну и наконец подправим обработчик события кнопки «*Найти*». Там мы полностью формировали запрос, но теперь этого не надо делать. Достаточно только изменить значение параметра. Для этого удали старый код обработчика и напиши следующий:

```
DataModule1.FindQuery.Active:=false;
DataModule1.FindQuery.Parameters.ParamByName('Telephone').Value:=
   FindTelephoneEdit.Text;
DataModule1.FindQuery.Active:=true;

FindResultForm.ShowModal;
```

В самом начале я так же делаю компонент запроса неактивным. После этого надо изменить значение параметра. Для этого я использую конструкцию *DataModule1.FindQuery.Parameters.ParamByName('Telephone').Value*. Сложно? Зато удобно.

Все параметры хранятся в свойстве *Parameters* компонента запроса. В этом свойстве, чтобы найти нужный параметр я использую метод *ParamByName*. В качестве единственного параметра этому методу нужно передать имя нашего параметра. Ну и наконец в свойстве *Value* мы записываем значение для найдённого параметра.

После этого, запрос можно делать активным, чтобы он выполнился и мы увидели результат. Как видишь, такая работа с динамическими запросами на много легче, особенно, если запросы большие, а изменяется только какое-то значение, которое можно заменить на переменную.

 *На компакт диске, в директории* ***\Примеры\Глава 14\SQL1*** *ты можешь увидеть пример этой программы.*

Я думаю, что на этом можно закончить разговор про запросы, осталось лишь добавить пару слов. Как я уже сказал, компонент *ADOQuery* очень похож на *ADOTable*. В ни очень много общего и часто используют в качестве основного доступа к данным именно *ADOQuery*. В нашем случае можно было поступить так же, потому что основное предназначение телефонного справочника – поиск необходимой информации. А вот дополнительные справочники (как, например, справочник городов) можно реализовывать в виде таблицы *ADOTable*.

Компонент *ADOQuery* имеет все необходимые методы необходимые для полноценной работы с базой данных, такие как Insert, Delete, Edit, Post, First, Next, Prev, Last и т.д., которые мы рассматривали для компонента *ADOTable*.

14.10 Связанные таблицы

Ты наверно можешь сказать, что у нас получился полноценный телефонный справочник. В него уже можно не только заносить данные, редактировать, удалять, но и искать по разным параметрам. Я дал уже достаточно информации, чтобы ты сам смог улучшить этот пример, но не всё так просто.

Представь себе ситуацию, когда у одного человека есть два телефона. Один из телефонов может быть простым, а другой может быть сотовым. Как внести такую информацию в нашу базу? Нужно внести две записи, в которых поля Фамилия, Имя, Город, e-mail и Дата рождения будут одинаковыми. Но это же неудобно и сразу заметно, что идёт лишний расход места на диске на хранение двух практически одинаковых строк.

Такая ситуация нарушает рациональность хранения информации. Я же говорил, что строки таблицы должны хранить как можно меньше одинаковой информации. Именно поэтому мы убрали поле Город в отдельный справочник, чтобы сэкономить место на диске и увеличить эффективность нашей базы данных. Но как поступить в данном случае, когда фамилию, имя и др. поля убирать в справочник нет смысла, да и телефоны держать в справочнике нерационально? Очень просто. В этом случае нам помогут связанные таблицы.

В одной таблице надо хранить следующие данные: Фамилия, Имя, Город, e-mail и Дата. В другой таблице будут: Телефон и Мобильник. Обе таблицы будут связаны между собой, как на рисунке 14.10.1.

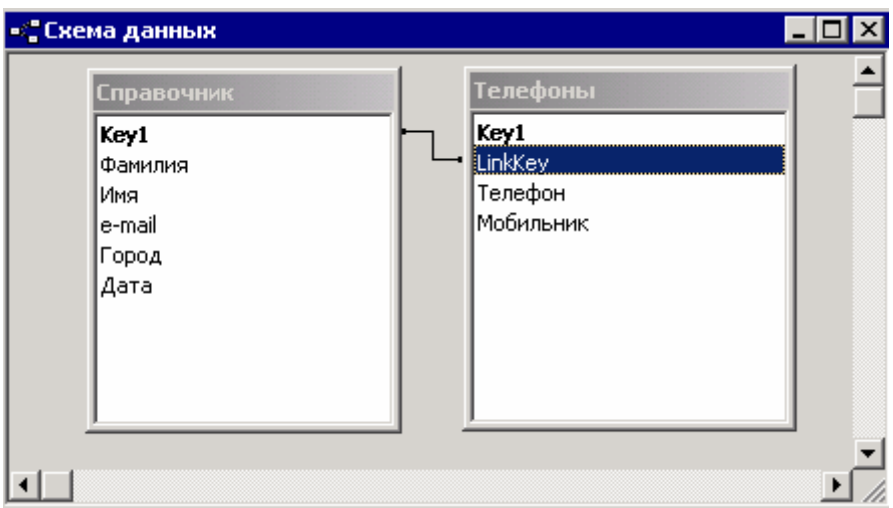


Рисунок 14.10.1 Схема данных

Здесь показаны две таблицы. В справочнике ты видишь все описанные мной поля общих сведений о владельце телефона. Во второй таблице «*Телефоны*» у нас четыре поля: счётчик, *LinkKey*, телефон и мобильник. Поля телефон (строковое поле, хранящее реальный номер телефона) и мобильник (логическое поле) выполняют ту же роль, что и раньше одноимённые поля в основной таблице «*Справочник*». В основной таблице теперь этих полей нет.

Что же такое *LinkKey* и почему между ним и полем «*Key1*» таблицы «*Справочник*» я провёл линию связи? *Key1* – уникальное поле, по которому мы точно можем его найти. *LinkKey1* – связующее поле и будет хранить ссылки на поле *Key1*. Например, взгляни на пример двух таблиц, показанный на рисунке 14.10.2.

Справочник			Телефоны			
Key1	Фамилия	Имя	Key1	Link	Телефон	Мобил
1	Иванов	Сергей	1	1	3351010	Нет
2	Петров	Иван	2	2	3461010	Да
			3	2	2125555	Нет
3	Иванов	Лёша	4	3	4547777	Да
			5	3	6562244	Нет

Рисунок 14.10.2 Схема данных

Слева на рисунке показана таблица «*Справочник*», а справа таблица «*Телефоны*». Если посмотреть на данные, то поле *Key1* постоянно увеличивается на единицу, потому что это счётчик. Теперь давай посмотрим на этом примере, как происходит связь через поля *Key1* таблицы «*Справочник*» и «*LinkKey*» таблицы «*Телефоны*».

В левой таблице у нас первая запись принадлежит Иванову Сергею. Во второй таблице ищем записи, у которых в поле Link стоит цифра 1. Такая запись только одна и она первая.

Вторая запись в левой таблице принадлежит Петрову Ивану. Смотрим в правой таблице записи, у которых в поле Link находиться число 2. Таких записей аж две (2-я и 3-я), значит у Петрова есть два телефона.

Третья запись в левой таблице принадлежит Иванову Алексею. Ищем в правой таблице записи с цифрой 3 в поле Link. Таких записей опять 2, значит и у Алексея тоже есть два телефона.

Таким образом, мы связываем две таблицы с помощью ключей. Когда мы создавали поисковые поля, у нас получалась связь, чем-то похожая на эту, только тогда главная таблица содержала поле *Город* которое подвязывалось под главный ключ справочника городов.

На словах сказано, пора и сделать. Переходим к нашему примеру. Для начала нужно открыть базу данных в Access и отредактировать поля таблицы *Справочник*, а именно убрать поля *Телефон* и *Мобильник*.

Теперь создавай новую таблицу в режиме конструктора со следующими полями:

Key1 – счётчик, ключевое поле.

LinkKey – числовое поле, в свойстве (*Индексированное Поле*) укажи (*Да* (*Допускаются совпадения*).

Телефон – текстовое, размер 10.

Мобильник – логическое.

Сохрани таблицу под именем *Телефоны*.

Теперь запускай Delphi. Открой модуль с компонентами для доступа к данным и дважды щёлкни по компоненту *BookTable*. В редакторе полей выдели поле *Телефон* и удали его кнопкой Del. Потом удали поле *Мобильник*, потому что мы удалили эти поля из базы. Удали эти же поля из компонента *FindQuery* плюс ещё здесь надо удалить поле *Key1*

(это потому что запрос на поиск будет проходить сразу на две таблицы и в обоих есть поле с именем Key1). Потом ты увидишь, почему мы удалили ключевое поле. А пока помни, что поиск у нас не работает, потому что SQL запрос сейчас неправильный. Чуть позже я исправлю его.

Теперь добавь сюда компоненты *DataSource* (назови его *TelephonSource*) и *ADOTable* (назови его *TelephonTable*) для доступа к таблице «Телефоны». Наведи свойство *DataSource* компонента *TelephonSource* на *TelephonTable*.

Теперь установи следующие свойства компонента *TelephonTable*:

Connection – укажи здесь наш компонент присоединения к базе данных *ADOConnection1*.

TableName – здесь укажи имя таблицы *Телефоны*.

Active – установи в *true*, чтобы открыть таблицу.

MasterSource – выбери здесь из выпадающего списка *BookSource*. Этим ты указываешь главную таблицу для таблицы *Телефонов*.

MasterFields – здесь мы должны указать связующие поля. Щёлкни по этому полю дважды и перед тобой откроется окно, как на рисунке 14.10.3. В списке *Detail Fields* (поля подчинённой базы) выбери поле *LinkKey*, а в списке *Master Fields* (поля главной базы) выбери поле *Key1*. Нажми кнопку *Add* и в списке *Joined Fields* (связанные поля) появиться строка отображающая выделенную связь. Закрой окно, кнопкой *OK* чтобы сохранить указанную связь.

Посмотри теперь в объектном инспекторе на свойство *IndexFieldNames*. Как видишь, там появилось имя поля *LinkKey* – поля через которое происходит связь. Так что в связанных таблицах нельзя использовать это поле для обеспечения сортировки, иначе нарушиться связь. ПОМНИ ЭТО!!!

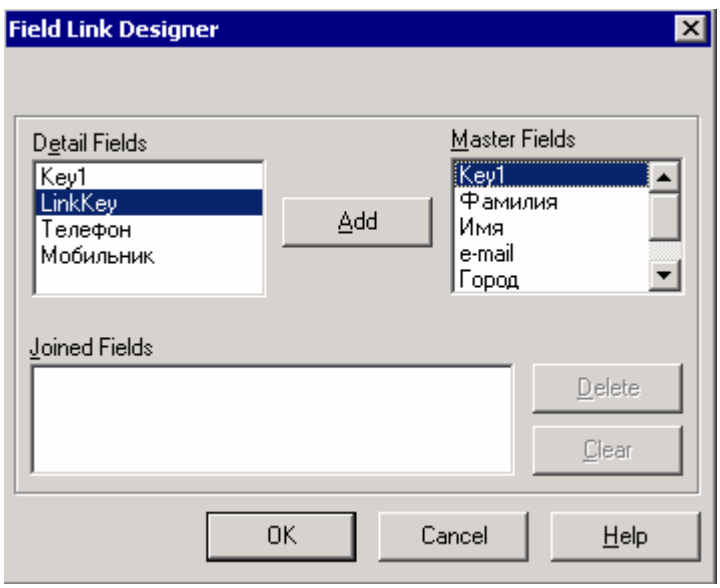


Рисунок 14.10.3 Окно создания связей между главной и подчинённой таблицей

Теперь дважды щёлкни по компоненту *TelephonTable*, чтобы увидеть окно редактирования свойств полей. Здесь тебе надо добавить поля, чтобы ты мог обращаться к ним потом по именам и спрятать первые два поля (ключевые поля, которые не несут пользователю полезной информации).

Теперь можно переходить к программированию или почти к программированию. Открой главную форму и добавь сюда ещё одну сетку *DBGrid*. Я расположил её по правому краю окна, как показано на рисунке 14.10.4. У сетки нужно изменить только свойство *DataSource* указав там нашу таблицу телефонов *DataModule1.TelephonSource*.

Может ты уже заметил, но я люблю ещё устанавливать свойство *BorderStyle* в *bsNone*, так красивее смотрится.

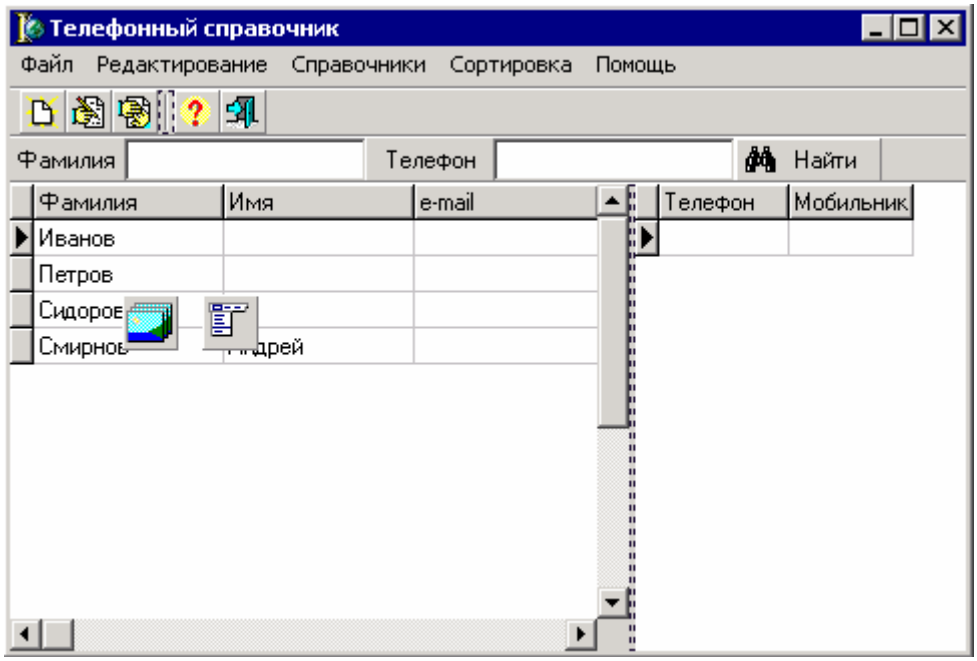


Рисунок 14.10.4 Улучшенная главная форма программы

В принципе, пример готов к запуску. Ты можешь выбирать в левой сетке любого человека, а в правой сетке можно вносить любое количество телефонов для выбранной записи. Для вставки новой записи можно использовать кнопку *Ins*, для удаления *Ctrl+Del*, чтобы сохранить изменения, нужно курсором перейти на другую запись, отменить редактирование записи (если изменения ещё не приняты) – кнопка *Esc*.

Вообще, на счёт сохранения изменений могу сказать следующее. Когда ты пишешь программы не для собственных нужд, то пользователи очень часто забывают или даже не знают о том, что для запоминания введённых изменений надо перейти на другую запись. Именно поэтому я люблю по событию *OnClose* для главной формы писать следующий код для всех таблиц моей программы:

```
if Table1.Modified then
 Table1.Post;
```

Здесь я проверяю, если во время закрытия программы таблица *Table1* изменена, то запоминаю изменения.

Ещё один способ следить за изменениями – устанавливать в сетках *DBGrid* в свойстве *Options* параметр *dgCancelOnExit* в *false*. А лучше использовать сразу оба этих способа.

Теперь давай подправим наш запрос на поиск по телефону. Открой модуль данных и введи в компонент *FindQuery* следующий запрос:

```
SELECT *
FROM Справочник, Телефоны
WHERE Телефон LIKE :Telephone
 AND Справочник.Key1=Телефоны.LinkKey
```

Здесь я уже выбираю все поля из двух таблиц, где есть записи с указанным значением телефона. К тому же здесь указана связь между таблицами. Если ты прочитал мой мануал по SQL, который идёт на диске, то с пониманием этого запроса проблем не будет.

Закрой редактор SQL запроса и дважды щёлкни по компоненту *FindQuery*. В редакторе свойств полей щёлкни правой кнопкой мыши и выбери пункт *Add All Fields*. В редактор будут добавлены все новые поля, которых в нём не было (это поля из таблицы *Телефоны* и ключевое поле таблицы *Справочник*) Обрати внимание, что теперь нет поля *Key1*. У нас две таблицы и в обеих есть поле *Key1*, поэтому для их различия названия полей изменены на следующий вид:

```
Имя Таблицы . Имя поля
```

Спрячь ключевые поля, потому что пользователь не должен их видеть и отсортируй всё так, чтобы удобно было работать. Попробуй запустить программу и посмотреть на результат (рисунок 14.10.5). Несмотря на то, что данные о телефонах хранятся в трёх таблицах, мы их получаем от SQL запроса в виде одной.

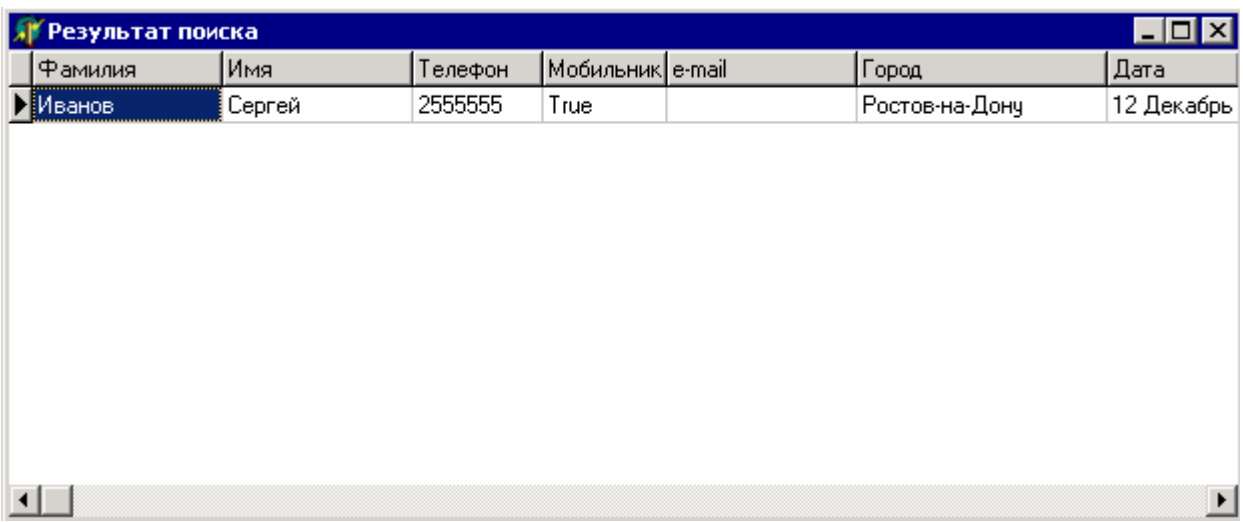


Рисунок 14.10.5 Улучшенная главная форма программы

Обрати внимание, что в окне результата поиска можно редактировать данные и они будут правильно отображены в своих таблицах. Попробуй изменить какое-нибудь поле. Когда ты закроешь окно результата поиска ты пока ничего не увидишь, потому что в сетке главной формы данные нужно обновить методом *Refresh* таблиц. Так что чтобы увидеть изменения, нужно закрыть программу и открыть её снова.

Чтобы не встречаться с такой проблемой, нужно подкорректировать обработчик события на нажатие кнопки «*Найти*». Допиши в конце (после показа окна результата) следующие строки:

```
DataModule1.BookTable.Refresh;
DataModule1.TelephonTable.Refresh;
```

Кстати, в нашей программе можно смело удалять сортировку, потому что использовать её в том виде, в котором мы её написали нельзя. Мы сортировали с помощью индекса, а в связанных таблицах индекс выполняет роль связей. Если мы изменим значение индексного поля, то нарушим связь между таблицами.

Но мы ещё можем воспользоваться свойством *Sort* таблицы. Единственное – мы не сможем сортировать по телефону, потому что он находиться в другой таблице, но можем упорядочить записи по любому полю главной таблицы. По нажатию пункта меню «*По фамилии*» из меню «*Сортировка*» напиши следующий код:

```
DataModule1.BookTable.Sort:='Фамилия ASC';
```

Пункт меню «*По телефону*» можно убирать, а вместо него давай сделаем сортировку по городу. По его нажатию пиши следующий код:

```
DataModule1.BookTable.Sort:='Город ASC';
```

Программу можно считать законченной, если бы не одно но – нужно подкорректировать окно редактирования данных. У нас изменилась главная таблица, значит и это окно должно измениться. Я не буду тратить место в книге на объяснения, как изменить окно, попробуй сделать это сам, потому что все необходимые знания у тебя уже есть. На рисунке 14.10.6 ты можешь видеть моё окно, постарайся привести его к такому виду. В принципе, тут добавлена только одна сетка для телефонов и три кнопки для добавления, редактирования и удаления записей.

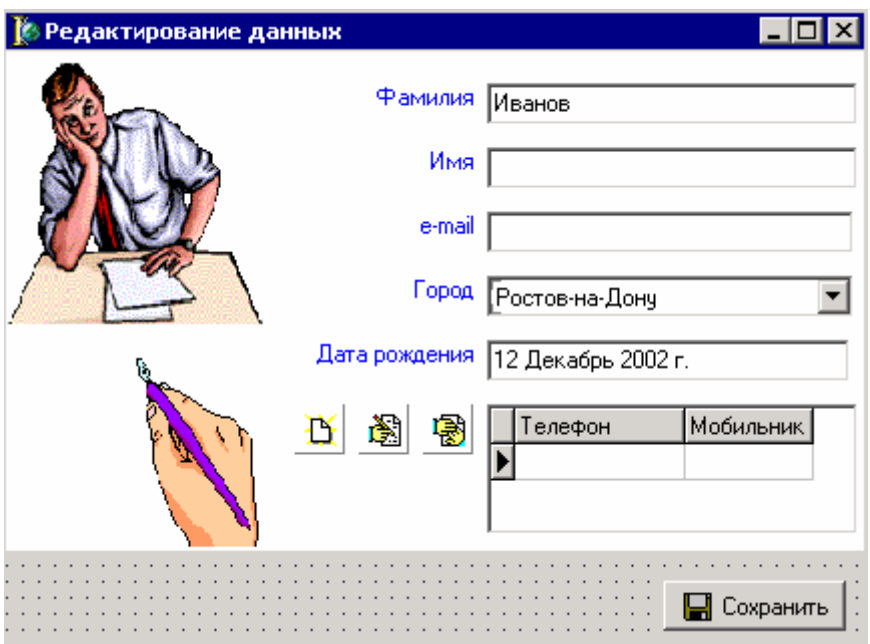


Рисунок 14.10.6 Новая форма редактирования данных о пользователе.

Для добавления и редактирования записей нам нужно ещё одно новое окно. Создай новую форму и назови её *PhoneEditForm*. Её внешний вид ты можешь увидеть на рисунке 14.10.7.

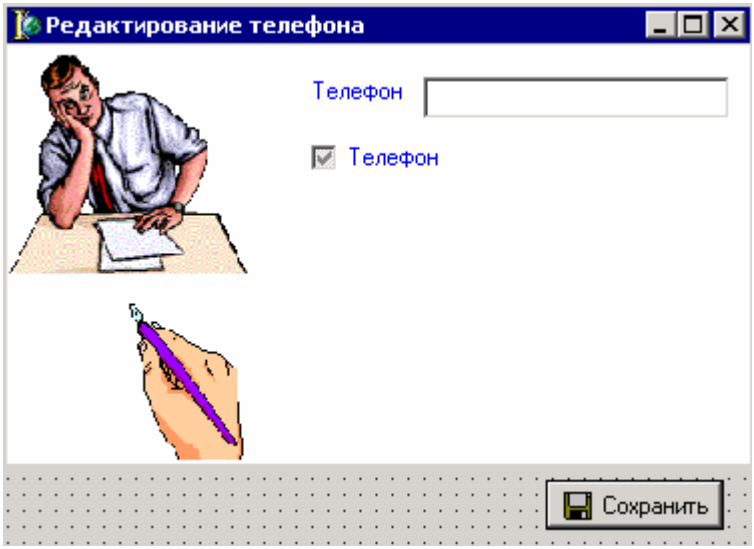


Рисунок 14.10.7 Окно для редактирования данных о телефоне

Код для кнопок добавления, редактирования и удаления записей попробуй написать сам. Он похож на тот, что мы уже писали для окна редактирования данных о пользователе. Если что-нибудь будет непонятно, ты всегда сможешь обратиться к исходникам на диске.

*На компакт диске, в директории **\Примеры\Глава 14\LinkTables** ты можешь увидеть пример этой программы.*

14.11 Вычисляемые поля.

Допустим, что у тебя есть база данных со следующими полями: Наименование, кол-во, цена. А почему «допустим», давай создадим новую базу данных в которой будет таблица с такими полями (рисунок 14.11.1). Сохрани эту таблицу под именем «*Товары*». В ней мы будем хранить наименование покупки, кол-во и цену.

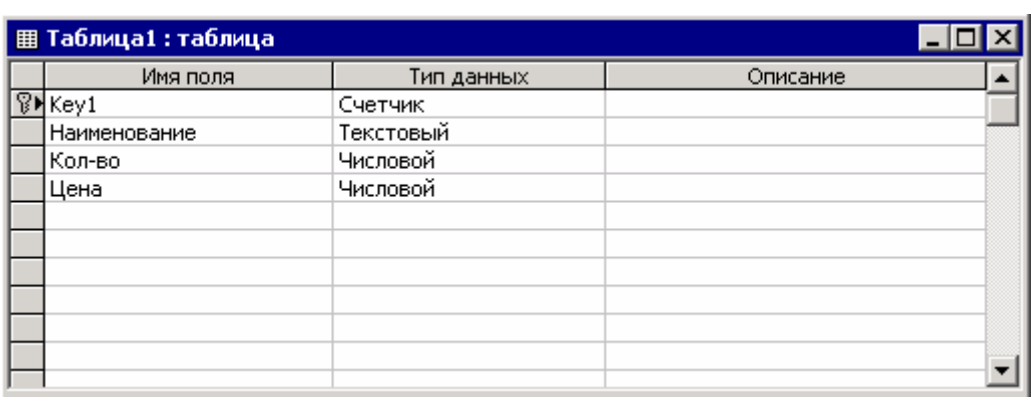


Рисунок 14.11.1 Таблица «Товары» новой базы данных

Цена в таблицы будет указываться за 1 товара, т.е. за штуку, кг или метры. Чтобы узнать общую цену товара, надо цену за единицу умножить на кол-во товара. При этом, итог должен моментально реагировать на любые изменения колонок кол-ва и цены. На первый взгляд задача достаточно сложная, но в программировании она проста, как никогда.

Создай новый проект в Delphi, и сразу добавь в него модуль *DataModule*. В этот модуль кинь компоненты *ADOConnection* (для соединения с базой данных), *DataSource* для возможности отображения данных из таблицы и *ADOTable* для соединения с таблицей (на рисунке 14.11.2 показано моё окно *DataModule*). Подключись к новой базе данных с помощью компонента *ADOConnection1*.

У *DataSource1* в свойстве *DataSet* укажи таблицу *ADOTable1*. В таблице *ADOTable1* в свойстве *Connection* укажи компонент *ADOConnection1*, в свойстве *TableName* нужно указать нашу таблицу «*Товары*». После этого можно делать таблицу активной (в свойстве *Active* нужно указать *true*).

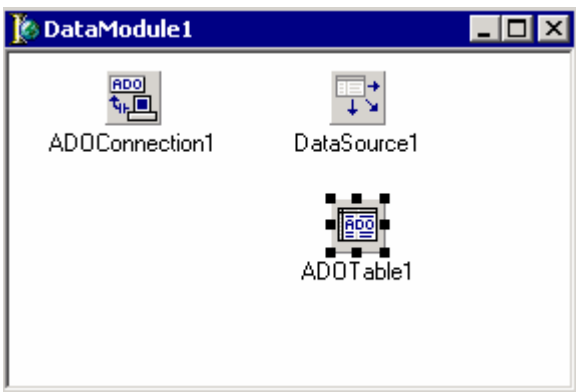


Рисунок 14.11.2 Окно DataModule.

Теперь щёлкай дважды по компоненту *ADOTable1* и в появившемся окне редактора свойств добавляй все поля таблицы. Для начала здесь надо сделать невидимым ключевое поле. Потом нужно установить значения по умолчанию для полей *Кол-во* и *Цена*. Эти поля будут участвовать в математических расчётах, поэтому в них обязательно должны быть какие-нибудь значения. Если в одном из полей не будет данных, то программа во время расчётов выдаст ошибку. Для поля *Кол-во* я указал в свойстве *DefaultExpression* (значение по умолчанию) единицу, а для поля *Цена* в том же свойстве поставил ноль.

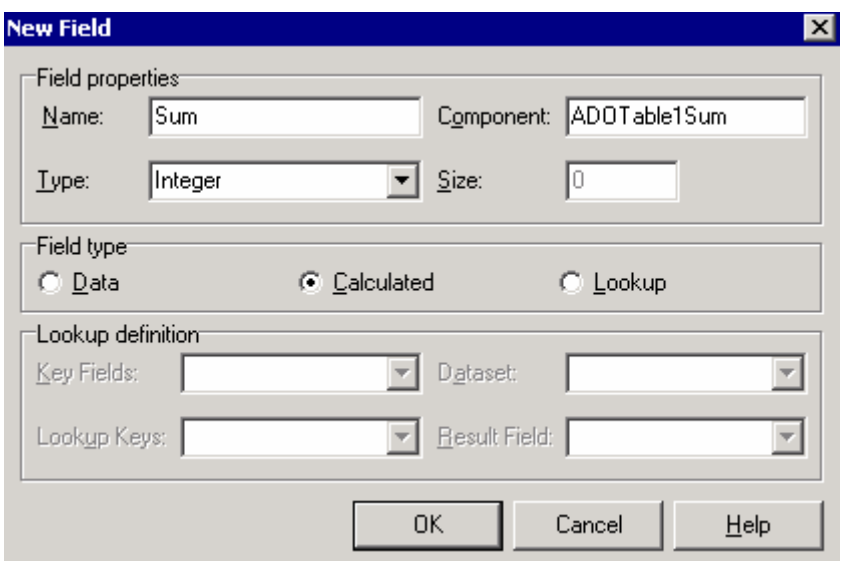


Рисунок 14.11.3 Окно создания нового поля.

Теперь создадим новое поле, которое будет хранить итог расчётов. Но прежде чем это делать, нужно сделать таблицу неактивной. Как ты помнишь, при создании поисковых полей я предупреждал, что новое поле можно создавать только при неактивной таблице. Щёлкой правой кнопкой в окне редактора свойств и выбирай пункт *New Field*. В окне свойств нового поля заполни следующие поля:

Name (имя нового поля) – назовём поле *Sum*.

Type (тип поля) – у нас будет числовая сумма, поэтому выбирай тип *Integer*.

Field Type (тип поля) – выбирай *Calculated*, чтобы создать вычисляемое поле.

На рисунке 14.11.3 ты можешь увидеть заполненное мной окно свойств созданного нового поля. Как только поле создано, таблицу снова можно делать активной.

Теперь выдели компонент *ADOTable1* и создай обработчик события *OnCalcFields*. Это событие вызывается каждый раз, когда надо пересчитать вычисляемые поля. Оно будет вызываться для всех видимых пользователю записей. В этом обработчике напиши следующее:

```
procedure TDataModule1.ADOTable1CalcFields(DataSet: TDataSet);
begin
 ADOTable1Sum.Value:=ADOTable1DSDesigner2.AsInteger*
      ADOTable1DSDesigner3.AsInteger;
end;
```

Прежде чем разбираться с этим кодом, открой окно редактора свойств полей таблицы *ADOTable1* и посмотри имена (свойство *Name*) полей *Кол-во*, *Цена*, и *Sum*. У меня это *ADOTable1DSDesigner2*, *ADOTable1DSDesigner3* и *ADOTable1Sum* соответственно. С помощью этих имён мы можем обращаться к значениям, находящимся в полях. Надо только написать имя поля и вызвать один из его методов, для преобразования значения в нужный формат. Тебе доступны следующие методы полей:

AsInteger – получить значение, хранящееся в данном поле в виде числа.

AsDateTime – в виде объекта *TDateTime*.

AsBoolean – в виде булева значения.

AsCurrency – в виде цены.

AsFloat – в виде вещественного значения.

AsString – в виде строки.

AsVariant – в виде типа *Variant*. Это универсальный тип, который может принимать любые значения, хоть число, хоть строку, в общем любые доступные типы.

Теперь взгляни на код и сразу же встанет всё понятно. Здесь мы записываем в свойство *Value* поля *ADOTable1Sum* результат перемножения значений полей цены и количества. Значения полей я получаю как целые числа – *AsInetger*.

Теперь переходим в главное окно нашей программы. Подключи к нему модуль *DataModule* (File->Use Unit) и брось на форму одну сетку *DBGrid*. Теперь в свойстве *DataSource* сетки выбери нашу таблицу *DataModule1.DataSource1*. Всё!!! Программа готова. Запускай, и попробуй ввести в базу несколько полей. На рисунке 14.11.4 ты можешь увидеть окно результата работы моего примера, но я тебе советую самому попробовать поиграть с ним.

Кстати, поле итога не должно изменяться пользователем вручную, потому что оно вычисляемое. Именно поэтому ты даже не сможешь туда ввести никакого значения. Delphi просто блокирует любые такие попытки, хотя поле и не имеет признака «Только для чтения» (*ReadOnly*).

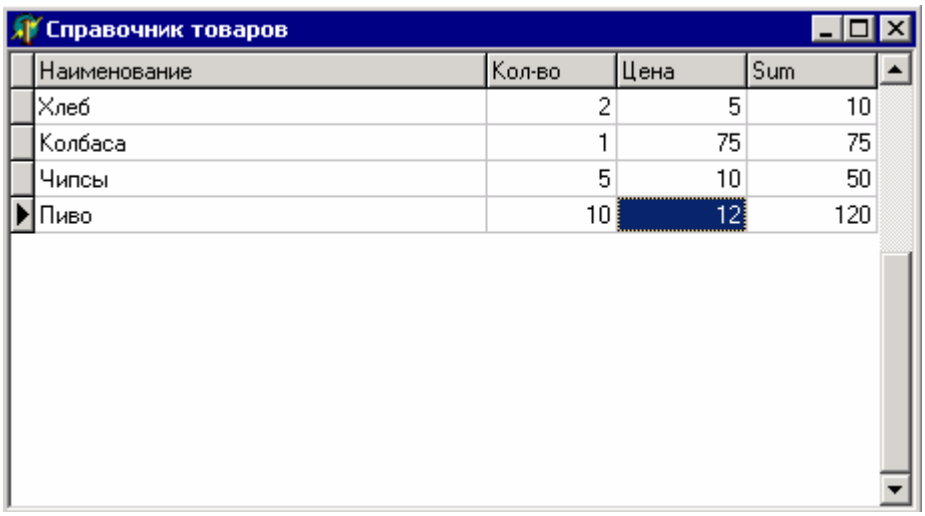


Рисунок 14.11.4 Результат работы программы.

На компакт диске, в директории ***\Примеры\Глава 14\Count*** *ты можешь увидеть пример этой программы.*

14.12 Цветные сетки DBGrid.

Я уже много раз встречался с проблемой, когда пользователи просят, чтобы у них в программе была возможность выделять некоторые записи каким-нибудь цветом. Это действительно удобно, да и в кодинге не сильно сложно. Сейчас я тебе покажу, как это делается.

Открой базу данных из предыдущей главы и добавь туда поле *Color*. Это поле должно иметь текстовый тип, а размер поля достаточно установить в 15 символов.

Теперь открывай Delphi. Загружай тот же пример и в модуле данных дважды щёлкай по компоненту *ADOTable1*. Выбери уже надоевший пункт «*Add All Fields*». В редактор будут загружены новые поля, точнее сказать одно поле – *Color*. Сделай его сразу же невидимым, потому что пользователю не надо видеть его текст, его интересует цвет строки.

Теперь перейди в главное окно программы. Добавь сюда компонент *PopupMenu* (всплывающее меню). Щёлкни по нём дважды, чтобы открыть редактор меню. Создай в нём следующие пункты:

1. Чёрный.
2. Красный.
3. Зелёный.
4. Жёлтый.
5. Синий.
6. Пурпурный.

Имена пунктов меню должны идти именно в таком порядке, а в свойстве *Tag* всех этих пунктов должен находиться порядковый номер пункта. Нумеровать цвета надо с нуля и до 5. Это значит, что у пункта меню *Жёлтый* в свойстве *Tag* будет находиться число 3, а у пункта *Пурпурный* значение 5.

На рисунке 14.12.1 ты можешь увидеть созданное мной меню.

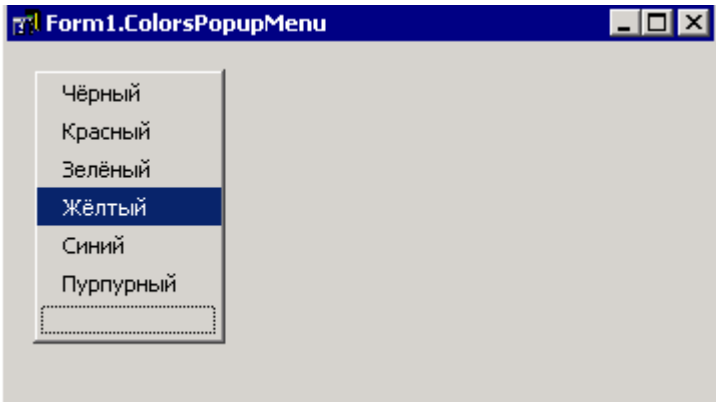


Рисунок 14.12.1 Всплывающее меню выбора цвета.

Теперь выдели все эти пункты меню (щёлкни на первом пункте и удерживая Shift на последнем). Перейди на закладку *Events* объектного инспектора и создай обработчик события *OnClick*. Будет создана процедура-обработчик события, которая будет назначена всем выделенным пунктам меню, т.е. одна процедура отвечает за нажатие любого из пунктов. В этом обработчике напиши следующий код:

```
procedure TForm1.N13Click(Sender: TObject);
const
 MenuColors: array[0..5] of TColor =(clBlack, clRed,
      clGreen, clYellow, clBlue, clPurple);
begin
 //Перейти в режим редактирования поля
 DataModule1.ADOTable1.Edit;

//Заношу в поле Color выбранный цвет
 DataModule1.ADOTable1Color.AsString:=
   ColorToString(MenuColors[TMenuItem(Sender).Tag]);

//Запомнить изменения
 DataModule1.ADOTable1.Post;
end;
```

В разделе **Const** я объявил одну константу – массив из 6 элементов имеющих тип ***TColor***. Так как это массив-константа и в процессе программирования не может менять свои значения, то эти значения нужно обязательно описать здесь. А именно, сразу же после объявления массива ставиться знак равенства и в скобках перечисляются значения элементов массива. Так как массив из 6 элементов, то и в скобках должно быть именно 6 элементов, ни больше, ни меньше. Я указал цвета, которые у меня указаны в пунктах всплывающего меню. Их имена перечислены в том же порядке, что и в меню.

В первой строке кода я перевожу таблицу в режим редактирования с помощью вызова метода *Edit*. Если этого не сделать, то любые попытки изменить данные в полях текущей записи встретят меня шквалом ошибок.

В следующей строке я присваиваю полю *Color* значение выбранного цвета. Выбранный цвет я получаю следующим образом: *MenuColors[TMenuItem(Sender).Tag]*.

Sender – эта переменная передаётся нам в качестве параметра в обработчик события и указывает на объект, который породил данное событие.

TMenuItem(Sender) – переменная *Sender* универсальна, поэтому имеет тип ***TObject*** – родитель всех компонентов. Но в таком виде мы не можем обратиться к свойствам, которых нет у ***TObject***, но есть у пунктов меню (нас интересует свойство *Tag*). Поэтому мы показываем компилятору, что данное событие *Sender* имеет тип ***TMenuItem***.

TMenuItem(Sender).Tag – получаем значение, указанное в свойстве *Tag* пункта меню, породившего это событие.

MenuColors[TMenuItem(Sender).Tag] – получаем из массива *MenuColors* значение цвета соответствующее значению, указанному в свойстве *Tag*.

Значение полученного цвета переводиться в строку с помощью функции *ColorToString* и записывается в поле *Color* в виде строки.

Код написан, теперь надо выделить сетку *DBGrid* и в свойстве *PopupMenu* указать созданное нами меню.

Теперь мы можем запускать приложение, изменять свойство Color в базе данных и осталось только научить программу читать это свойство и выводить текст в зависимости от указанного там значения цвета. Для этого создай обработчик события *OnDrawDataCell* для нашей сетки. Это событие вызывается, когда нужно перерисовать данные какой-нибудь ячейки сетки. В обработчике напиши следующий код:

```
procedure TForm1.DBGrid1DrawDataCell(Sender: TObject; const Rect: TRect;
  Field: TField; State: TGridDrawState);
begin
 try
  DBGrid1.Canvas.Font.Style:=[];
  if (gdSelected in State) or (gdFocused in State)then
   begin
    DBGrid1.Canvas.Brush.Color:=clHighLight;
    DBGrid1.Canvas.Font.Color:=clWhite;
   end
  else
   begin
    DBGrid1.Canvas.Brush.Color:=clWhite;
    DBGrid1.Canvas.Font.Color:=clBlack;

    //Если поле цвета не пустое то использовать цвет из поля
    if DataModule1.ADOTable1Color.AsString<>'' then
     DBGrid1.Canvas.Font.Color:=
       StringToColor(DataModule1.ADOTable1Color.AsString);
   end;
  //Очищаю ячейку
  DBGrid1.Canvas.FillRect(Rect);
  //Вывожу текст ячейки
  DBGrid1.Canvas.TextOut(Rect.Left, Rect.Top, Field.AsString);
 except
  DBGrid1.Canvas.TextOut(Rect.Left, Rect.Top, Field.AsString);
 end;
end;
```

Прежде чем описывать этот код хочу тебя предупредить, что пока что у тебя ничего не откомпилируется. Delphi будет ругаться на тип ***TField***. Этот тип описан в модуле db, поэтому добавь его в раздел **uses**. После этого при компиляции ошибок не должно быть.

Теперь рассмотрим параметры, которые мы получили в обработчик. У нашего обработчика события есть следующие параметры:

Sender – объект, который сгенерировал это событие. У нас это будет сетка.

Rect – здесь хранятся границы области ячейки, которую надо перерисовать. Границы передаются в виде структуры ***TRect***.

Field – этот параметр имеет тип ***TField*** и указывает на поле, которое надо перерисовать.

State – здесь находятся параметры, указывающие на текущее состояние ячейки. Возможны следующие параметры:

gdSelected – ячейка выделена.

gdFocused – ячейка имеет фокус ввода.

gdFixed – ячейка является фиксированной. Такие ячейки используются для названий колонок (сверху сетки) и для индикации текущей строки (слева сетки).

Ну а теперь давай знакомиться с кодом самой процедуры. В первой строке я устанавливая стиль шрифта у холста сетки. Точнее сказать, я очищаю все настройки шрифта, потому что стилю присваивается пустой набор [].

В следующей строке я проверяю, если текущая ячейка выделена или имеет фокус ввода, то цвет кисти изменяю на *clHighLight* (это константа, хранящая цвет, используемый для подсветки). Цвет шрифта я устанавливаю в белый.

Если ячейка не выделена, то цвет фона (цвет кисти) делаю белым, а цвет шрифта чёрным. Далее идёт проверка, если поле цвета текущей строки не пустое, то цвет шрифта меняю на тот цвет, который указан в поле цвета. Единственное, что надо учитывать – цвет храниться в виде строки, поэтому я преобразовываю его в цвет с помощью функции StringToColor.

Все приготовления закончены, можно заняться рисованием. Для начала я очищаю старое значение ячейки с помощью вызова метода *FillRect* холста сетки, который закрашивает указанную область цветом кисти (фона). В качестве единственного параметра я указываю область ячейки в виде структуры *TRect*, которую получаю через параметры обработчика.

После закраски я рисую текст ячейки. Если произошла какая-то ошибка во время подготовки к рисованию (она может возникнуть только при преобразовании StringToColor, если в поле находиться неправильное значение), то я пытаюсь снова вывести текст в блоке *except…end*.

 *На компакт диске, в директории* ***\Примеры\Глава 14\Color*** *ты можешь увидеть пример этой программы.*

14.13 Подключение к базе данных во время выполнения программы.

Использовать заготовленную базу данных очень удобно, но вдруг пользователь захочет выбирать, с какой базой данных ему работать? Зачем это нужно? Допустим, что у тебя программа каждый месяц копирует базу данных в отдельное место и потом очищается, чтобы не содержать устаревших данных. А что если пользователь захотел посмотреть эти старые данные? Ему надо писать отдельную программу для просмотра или придётся пользоваться неудобной программой Access. Но есть выход проще – подключить программу к старой архивной базе данных.

Возьмём пример написанный в прошлой главе. Добавь на форму одну кнопку и по её нажатию напиши следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 DataModule1.ADOConnection1.Close;
```

```
 if EditConnectionString(DataModule1.ADOConnection1) then
  begin
   DataModule1.ADOConnection1.Connected:=true;
   DataModule1.ADOTable1.Active:=true;
  end;
end;
```

Чтобы этот код откомпилировался, нужно в раздел **uses** добавить модуль *ADOConEd*. Вот теперь программа откомпилируется и без проблем запуститься.

Теперь посмотрим, что здесь происходит. В первой строке я выполняю метод *Close* компонента *ADOConnection*, чтобы закрыть соединение с базой данных. После этого я вызываю функцию *EditConnectionString*, которая отображает окно подключения к базе данных, которое ты уже видел и можешь ещё раз увидеть на рисунке 14.13.1. В качестве параметра в эту функцию нужно передать компонент *ADOConnection*, параметры которого будут изменяться.

Если пользователь выбрал новое имя файла, то программа вернёт значение *true* и мы откроем соединение с базой и сделаем единственную таблицу активной. Помни, что после закрытия и открытия базы данных, все таблицы становятся неактивными, поэтому тебе придётся программно восстанавливать их активность.

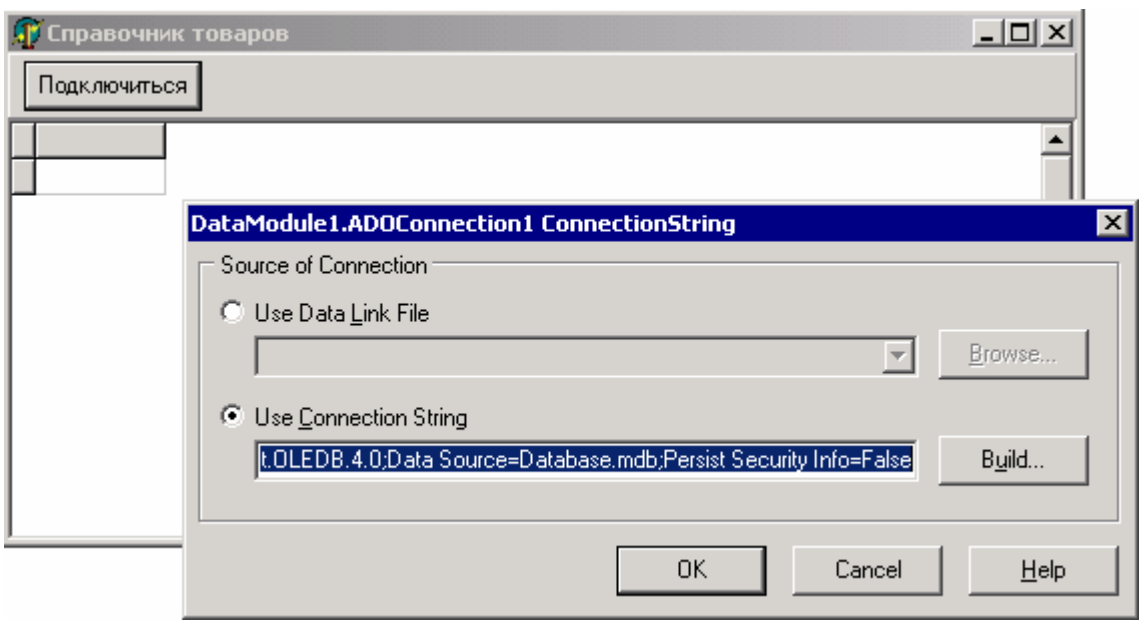


Рисунок 14.13.1 Результат работы программы.

На компакт диске, в директории ***\Примеры\Глава 14\Connect*** *ты можешь увидеть пример этой программы.*

Глава 15. Создание отчётности

С базой данных мы уже научились работать и создали теперь уже полноценный телефонный справочник. Но какая база данных без отчётности? Практически всегда возникает потребность в выгрузке данных в другую программы.

Все примеры будут работать с базами данных, потому что именно при их программировании возникает потребность в создании каких-то выходных документов. Нет смысла вести данные, которые нельзя распечатать или выгрузить в другой формат. В этой главе нам предстоит познакомиться с выгрузкой данных в Excel и научиться формировать красивые выходные документы, которые будут готовы к распечатке.

Но даже если ты пока не думаешь о создании отчётности, я всё же советую тебе ознакомиться с этой главой, потому что здесь будет достаточно много интересных вещей, о которых я пока что не говорил. Например, в прошлой главе я не мог рассмотреть все свойства и методы таблицы ADOTable, поэтому были рассмотрены только основные возможности. В этой главе будет ещё множество пример по работе с данными, так что мы закрепим на практике уже знаковые методы и познакомимся с некоторыми новыми.

15.1 Создание отчётности в Excel

Первое, с чем мы познакомимся – отчётность в Excel. Потребность в выгрузке данных в Excel может возникнуть у каждого программиста баз данных, ведь Office установлен в нашей стране практически на каждом компьютере. А это значит, что нашу отчётность можно смело переносить между компьютерами и быть уверенным, что её смогут прочитать.

В этой части будет не так уж много визуальных манипуляций, зато кода будет предостаточно. Поэтому про мышку можно позабыть и поближе подвинуть клавиатуру.

Но в начале всё таки ткнём мышкой и добавим на форму одну кнопку, по нажатию которой будет создаваться отчётность. Я ещё добавил к программе пункт меню «*Экспорт в Excel*» в меню «*Файл*». Результат можешь увидеть на рисунке 15.1.1.

Вот теперь мышку можно убирать. Переходи в редактор кода и сразу же добавляй в раздел **uses** модуль *ComObj*. В этом модуле описаны все необходимые функции для работы с COM объектами (мы о них пока не говорили, но всё ещё впереди).

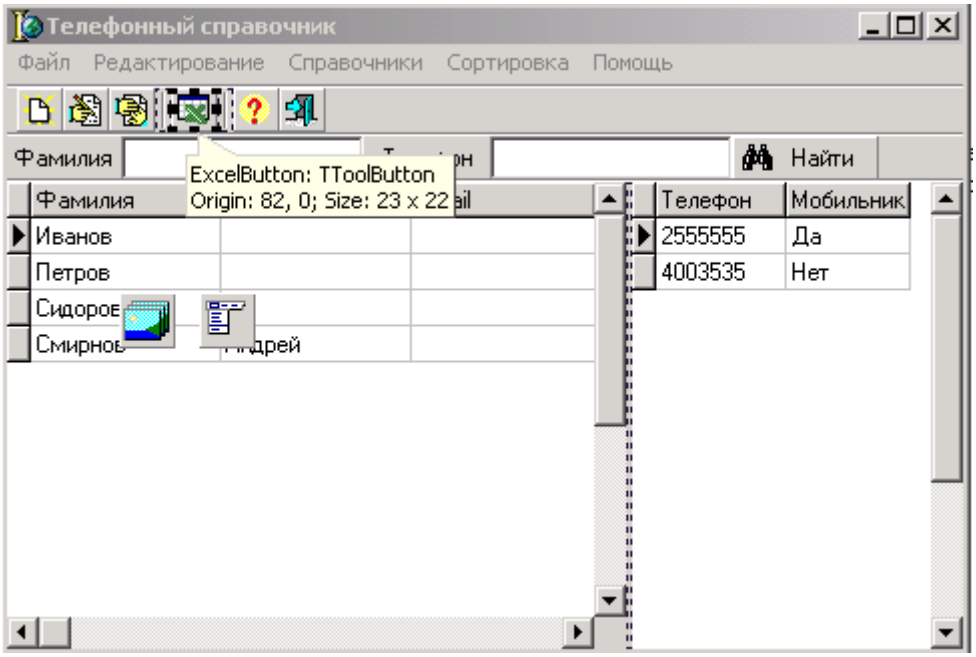


Рисунок 15.1.1 Кнопка создания отчётности в Excel

Теперь создавай обработчик события для кнопки и укажи его же в качестве обработчика для пункта меню (если ты его создал). В этом обработчике нужно написать следующий код:

```
var
 XLApp,Sheet,Colum:Variant;
 index,i:Integer;
begin
 XLApp:= CreateOleObject('Excel.Application');
 XLApp.Visible:=true;
 XLApp.Workbooks.Add(-4167);
 XLApp.Workbooks[1].WorkSheets[1].Name:='Отчёт';
 Colum:=XLApp.Workbooks[1].WorkSheets['Отчёт'].Columns;
 Colum.Columns[1].ColumnWidth:=20;
 Colum.Columns[2].ColumnWidth:=20;
```

```
Colum.Columns[3].ColumnWidth:=20;
Colum.Columns[4].ColumnWidth:=20;
Colum.Columns[5].ColumnWidth:=20;

Colum:=XLApp.Workbooks[1].WorkSheets['Отчёт'].Rows;
Colum.Rows[2].Font.Bold:=true;
Colum.Rows[1].Font.Bold:=true;
Colum.Rows[1].Font.Color:=clBlue;
Colum.Rows[1].Font.Size:=14;

Sheet:=XLApp.Workbooks[1].WorkSheets['Отчёт'];
Sheet.Cells[1,2]:='Телефонный справочник';
Sheet.Cells[2,1]:='Фамилия';
Sheet.Cells[2,2]:='Имя';
Sheet.Cells[2,3]:='e-mail';
Sheet.Cells[2,4]:='Город';
Sheet.Cells[2,5]:='Дата рождения';

index:=3;
DataModule1.BookTable.First;
for i:=0 to DataModule1.BookTable.RecordCount-1 do
 begin
  Sheet.Cells[index,1]:=DataModule1.BookTable.Fields.Fields[1].AsString;
  Sheet.Cells[index,2]:=DataModule1.BookTable.Fields.Fields[2].AsString;
  Sheet.Cells[index,3]:=DataModule1.BookTable.Fields.Fields[3].AsString;
  Sheet.Cells[index,4]:=DataModule1.BookTable.Fields.Fields[5].AsString;
  Sheet.Cells[index,5]:=FormatDateTime('dddddd',
      DataModule1.BookTable.Fields.Fields[6].AsDateTime);
  Inc(index);
  DataModule1.BookTable.Next;
 end;
```

Первая строка создаёт объект *Excel (XLApp:= CreateOleObject('Excel.Application'))*, и записывает его в переменную *XLApp*. Эта переменная типа *Variant*. *Variant* - это тип, который может принимать любые значения: строки, числа, указатели и др.

Функцию *CreateOleObject* я сейчас подробно рассматривать не буду, потому что она не относиться к базам данных или отчётности, и требует отдельного разговора. Единственное, что я сейчас скажу – она позволяет наладить связь с другим приложением по технологии COM. Через эту связь можно передавать данные в чужие приложения. Для этого программа, к которой мы присоединяемся, должна иметь соответствующие возможности для получения данных из вне (как, например, Excel) и тебе должны быть известны функции с которыми можно работать. Чаще всего такие вещи документируются на сайте разработчиков. Я же не могу описать все возможности всех программ, потому что если это сделать, то «Война и Мир» покажутся детской колыбельной. Я думаю, это никому не нужно. Так что я рассмотрю только Excel, чтобы показать возможности передачи данных между приложениями.

Вторая строка (*XLApp.Visible:=true*) заставляет запустить сам *Excel*. Потом я добавляю новую рабочую книгу (*XLApp.Workbooks.Add(-4167)*). Число в скобках - это константа, которая означает создание книги и её изменять нельзя. Подробнее о всех константах ты можешь почитать в руководстве разработчика на сайте MS или в файле excel97.pas, а я не могу тратить на это место книги, потому что констант предостаточно.

Дальше я даю название созданной книге *XLApp.Workbooks[1].WorkSheets[1].Name:='Отчёт'*. Это действие не обязательно, но я всегда это делаю, потому что меня бесит название по умолчанию "*Лист 1*". Всё должно быть понятно с первого взгляда.

Теперь у нас *Excel* запущен и создана новая книга. Можно переходить к впечатыванию данных. Но прежде чем это сделать я отформатирую колонки и строки. Для этого я получаю указатель на колонки рабочей книги (*Colum:= XLApp. Workbooks[1]. WorkSheets['Отчёт']. Columns*), и записываю результат в переменную *Colum* типа *Variant*. Теперь последовательно изменяю ширину колонок (*Colum. Columns[1]. ColumnWidth := 20*). На русском эта команда будет звучать так: *Колонки.Колонка[1].ШиринаКолонки:=20.*

После этого я в ту же переменную записываю указатель на строки рабочей книги (*Colum := XLApp. Workbooks[1]. WorkSheets['Отчёт']. Rows*). Для украшения строк нашего отчёта, я устанавливаю у первых двух строк жирный шрифт (*Colum. Rows[1]. Font. Bold := true*). В квадратных скобках теперь порядковый номер строки. Далее идут две строки, в которых я устанавливаю цвет первой строки в синий и размер шрифта равный 14.

Форматирование окончено, теперь можно выводить данные. Для этого я получаю указатель на лист (*Sheet:=XLApp.Workbooks[1].WorkSheets['Отчёт']*). Для того, чтобы вывести данные, нужно просто присвоить значение в *Sheet.Cells[строка, колонки]*. Давай посмотрим на код вывода данных таблицы:

```
index:=3;
DataModule1.BookTable.First;
for i:=0 to DataModule1.BookTable.RecordCount-1 do
 begin
  Sheet.Cells[index,1]:=DataModule1.BookTable.Fields.Fields[1].AsString;
  Sheet.Cells[index,2]:=DataModule1.BookTable.Fields.Fields[2].AsString;
  Sheet.Cells[index,3]:=DataModule1.BookTable.Fields.Fields[3].AsString;
  Sheet.Cells[index,4]:=DataModule1.BookTable.Fields.Fields[5].AsString;
  Sheet.Cells[index,5]:=FormatDateTime('dddddd',
     DataModule1.BookTable.Fields.Fields[6].AsDateTime);
  Inc(index);
  DataModule1.BookTable.Next;
 end;
```

Сначала я задаю переменной *index* значение 3. Эта переменная будет отображать, в какую строку таблицы Excel мы сейчас должны выводить данные. Первые две строки у нас уже заняты заголовками для отчёта, поэтому данные нужно начинать выводить с третей строки.

Обрати внимание, что строки и таблицы в Excel нумеруются начиная с единицы, а не с нуля, как в остальных таблицах и массивах.

После этого я перехожу на первую строку нашей таблицы с помощью метода *First* компонента *AdoTable*. Это необходимо, потому что пользователь может перед нажатием кнопки отчёта выделить любую строку в середине таблицы и в этом случае отчёт пойдёт от этой выделенной строки.

Теперь мы готовы запускать цикл, в котором будут перебираться все строки и информация из них будет попадать в Excel. Цикл я запускаю начиная с 0 и до количества строк в таблицы. Таким образом я переберу все записи.

Почему в качестве цикла я использую именно конструкцию *for..to..do*? Не знаю, просто иногда удобней так, а иногда я использую цикл *while*. С циклом *while* этот код выгладил бы так:

```
while DataModule1.BookTable.Eof<>true do
 begin
  //Вывод данных в Excel
 Увеличение переменной index;
 Переход на следующею строку;
end;
```

В принципе, всё то же самое, только используется цикл *while*. Этот цикл удобнее использовать, когда нужно вывести не всю таблицу, а, например, только начиная с текущей позиции. В этом случае не надо будет перед циклом переходить на первую строку, а достаточно только запустить этот цикл, который будет выполняться, пока DataModule1.BookTable.Eof не станет равным *true*, т.е. не будет достигнут конец таблицы.

Только не забывай внутри цикла переходить на следующую строку. Если ты забудешь это сделать с циклом *for*, то у тебя в *Excel* попадёт столько же строк, сколько и в таблице, но все они будут одинаковыми (равны первой строке таблицы). Ну а при цикле *while* программа может зависнуть, потому что цикл будет бесконечным, ведь конец таблицы никогда не будет достигнут, если не переходить на следующие строки.

Теперь поговорим о выводе данных. Я последовательно заполняю колонки данными из таблицы, присваивая в *Sheet.Cells[index, номер колонки]*, соответствующие данные. Данные из базы данных я беру по индексу (до этого мы обращались по имени, но здесь я захотел показать тебе, как это делается по индексу). Для этого используется следующая конструкция:

```
DataModule1.BookTable.Fields.Fields[Номер поля].AsString
```

После вывода данных я увеличиваю переменную *Index*, чтобы на следующем этапе выводить данные в следующую строку и перехожу на новую строку.

В принципе, это и всё. На последок я дам несколько замечаний и покажу тебе универсальный способ вывод данных из сетки:

```
for i:=1 to Table.RecordCount do
 begin
  for j:=1 to DBGrid.Columns.Count do
   Sheet.Cells[Index, j]:=DBGrid.Fields[j-1].AsString;

  Inc(Index);
  Table.Next;
 end;
```

Это просто общий пример не связанный с нашей программой. Здесь так же запускается цикл по всем строкам таблицы. Но вывод данных происходит по новому. Для этого запускается ещё один цикл от 1 до количества колонок в сетке *DBGrid*. А внутри этого цикла я присваиваю очередной колонке Excel, значение из такой же колонки сетки *DBGrid*:

```
Sheet.Cells[Номер строки, Номер колонки]:=
   DBGrid.Fields[Номер колонки в сетке].AsString;
```

Этот способ более универсален и может подойти почти всегда, когда надо вывести всё содержимое сетки DBGrid и все данные одного типа. В нашем случае в таблице есть дата, поэтому, чтобы она красиво выгладила я её форматирую с помощью *FormatDateTime*. Так что прямой перенос для этой колонке был бы не очень удобен, но остальные колонки можно было бы перенести указанным способом.

В процессе вывода данных можно изменять цвет строк *Sheet.Rows[строка].Font.Color* или колонок *Sheet.Columns[колонка]*. *Font.Color*, простым присваиванием (как мы это делали при форматировании). Если нужно изменить цвет отдельной ячейки, то это можно сделать, присвоив новое значение в *Sheet.Cells[строка, колонка].Font.Color*.

Вот ещё некоторые параметры, которые ты можешь изменить:

Sheet.Cells[строка,колонка].Font.Italic - курсивный шрифт
Sheet.Cells[строка,колонка].Font.Bold - жирный шрифт
Sheet.Cells[строка,колонка].Font.Underline - подчёркнутый шрифт
Sheet.Cells[строка,колонка].Font.Size - размер шрифта

С помощью всего этого, ты сможешь создавать простые, но эффективные отчёты. В Delphi, в директории Lib есть файлик excel97.pas, в нём ты найдёшь все доступные функции Excel. Если что-то не будет ясно, то пока отложи это занятие, чуть позже ты наберёшься навыков для понимания заголовочных файлов.

А у нашей программы остался один недостаток – она выводит данные только из основного справочника, и не учитывает телефоны. Чтобы избавиться от этого недостатка можно написать отдельный SQL запрос, который будет формировать сводную таблице из наших двух. Этот запрос можно поместить в отдельный компонент *ADOQuery*, выполнить его и брать данные оттуда, а не из таблицы *BookTable*.

Вот пример запроса, который выбирает все строки из двух таблиц формируя отдельную базу данных.

```
SELECT *
FROM Справочник, Телефоны
WHERE  Справочник.Key1=Телефоны.LinkKey
```

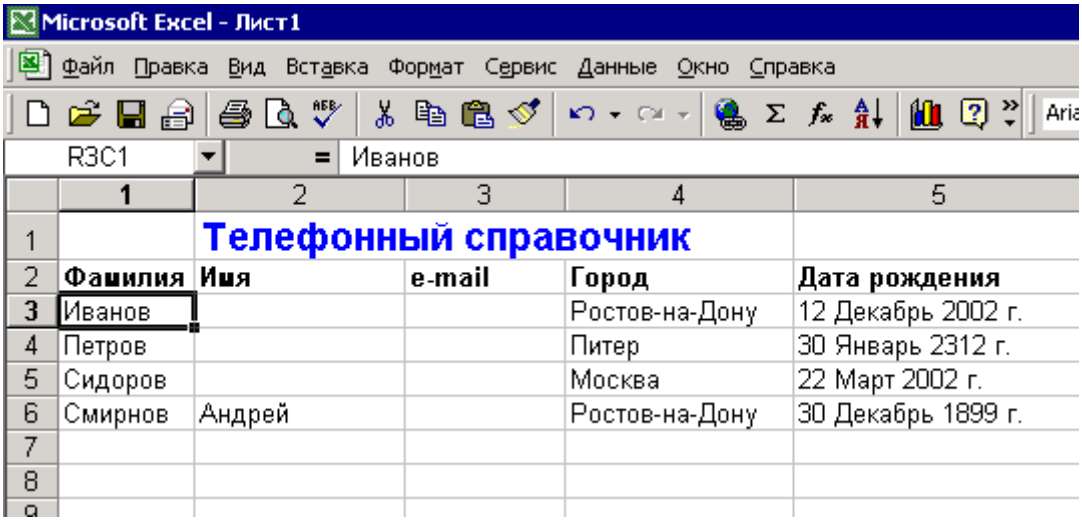


Рисунок 15.1.2 Кнопка создания отчётности в Excel

На рисунке 15.1.2 ты можешь увидеть результат работы программы. Вот так выглядит созданная мной отчётность.

 *На компакт диске, в директории* ***\Примеры\Глава 15\Excel*** *ты можешь увидеть пример этой программы.*

15.2 Отчётность в **Quick Reports**

Теперь я хочу познакомить тебя с мощным средством создания отчётов, которое входит в поставку Delphi – это Quick Reports. Он не является самым быстрым, и в сети Internet можно найти множество более быстрых, как платных, так и бесплатных генераторов отчётов. Но Quick Reports очень мощный и уже установлен в Delphi и готов к работе. Именно поэтому мы будем рассматривать его, Где бы ты не сел за Delphi, этот генератор уже будет установленным и его можно использовать.

Все компоненты Quick Reports находятся на закладке *QReport* палитры компонентов. В Delphi 6 у меня уже 23 компонента, позволяющие создать умопомрачительные документы готовые к печати.

Давай сначала посмотрим на головной компонент Quick Reports – ***TQuickRep***. Этот компонент – основа любого отчёта. Он представляет собой холст листа будущего отчёта. Попробуй дважды щёлкнуть по значку этого компонента в палитре компонентов и он перенесётся на форму во всей своей красе. Тебе надо выровнять края компоненты по форме и ты увидишь готовый белый лист, на котором можно будет размещать будущее твоего документа. По краям листа ты можешь видеть синие пунктирные линии, которые показывают границы документа.

Давай посмотрим на объектный инспектор и разберёмся с полями нашего отчёта:

Bands – здесь ты можешь указать, что должен иметь будущий документ, а он может содержать:

HasColumnHeader – Заголовки колонок. Если твой отчёт будет содержать таблицу, то она должна иметь шапку, где будут описаны названия колонок. Вот именно эту шапку создают в этой части документа. Так что если тебе нужна будет таблица, то этому свойству нужно будет присвоить *true*.

HasDetail – если в отчёте есть таблица, то вид строк делается в этом разделе.

HasPageFooter – в этом разделе создаётся нижний колонтитул.

HasPageHeader – здесь создаётся заголовок документа.

HasSummary –содержимое этого раздела печатается один раз в конце отчёта (на последней странице).

HasTitle – в этом разделе делается заголовок отчёта.

Попробуй создать новый проект и бросить на форму один компонент *QuickRep*. Теперь включи какие-нибудь разделы и посмотри на результат. На форме должны появиться области очерченные пунктирной линией и внизу этой области должно быть написано её предназначение. Так ты легко можешь отличить их между собой. Любую область можно выделить и растянуть или уменьшить (рисунок 15.2.1).

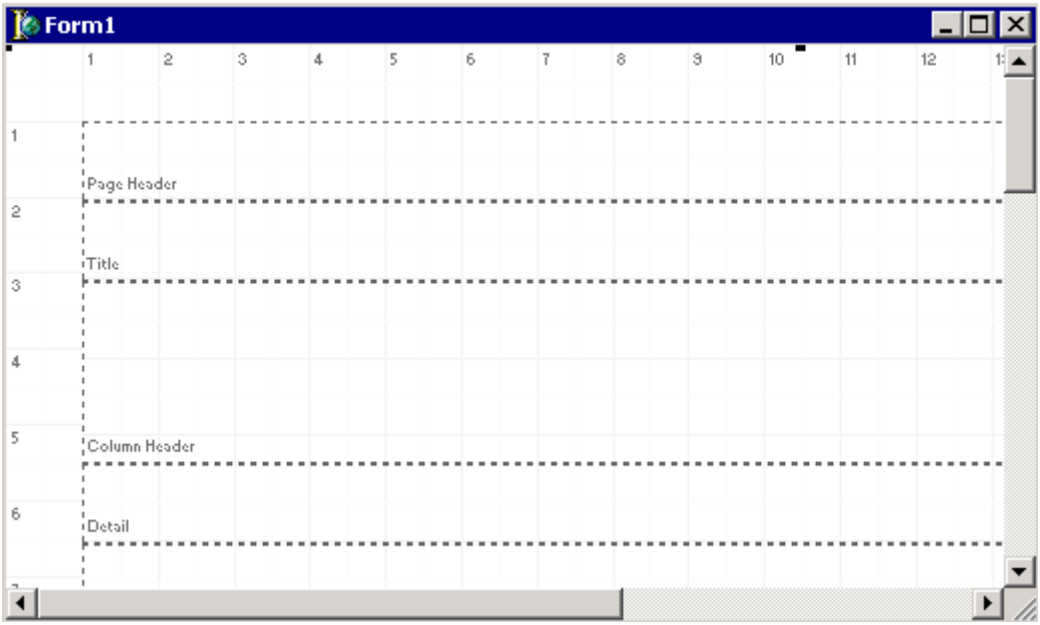


Рисунок 15.2.1 Вид компонента QuickRep с включенными областями

DataSet – здесь указывается набор данных (таблица) из которой отчёт будет брать данные.

Font – шрифт, который будет использоваться по умолчанию.

Frame – здесь ты указываешь параметры рамки.

Options – здесь тебе доступны три параметра. Если *FirstPageHeader* равно *true*, то заголовок печатается только на первой странице отчёта. Если *LastPageFooter* равен *true*, то нижний колонтитул печатается только на последней странице отчёта. Если установить свойство *Compression* в *true*, то отчёт будет сохраняться в сжатом виде.

Page – здесь тебе доступны все необходимые опции для контроля над бумагой отчёта. Ты можешь установить её размеры, отступы и ориентацию.

PrinterSettings – здесь находятся настройки принтера. С принтером мы уже работали, да и настройки практически не требуют пояснения.

ReportTitle – здесь находится заголовок печатаемого документа.

ShowProgress – если этот параметр равен *true* то во время печати тебе будет доступен индикатор хода выполнения печати.

SnapToGrid – нужно ли выравнивать компоненты по установленной сетке.

Zoom – масштаб отображения данных.

Если дважды щёлкнуть по компоненту QuickRep, то перед тобой откроется окно, в котором все эти настройки представлены в одном окне и в очень удобном виде. Они достаточно понятны и легко разобраться со всем самостоятельно (смотри рисунок 15.2.2). Я предпочитаю пользоваться именно этим окном, потому что здесь всегда можно увидеть будущий результат и если что, то все изменения можно отменить нажатием кнопки *Cancel*.

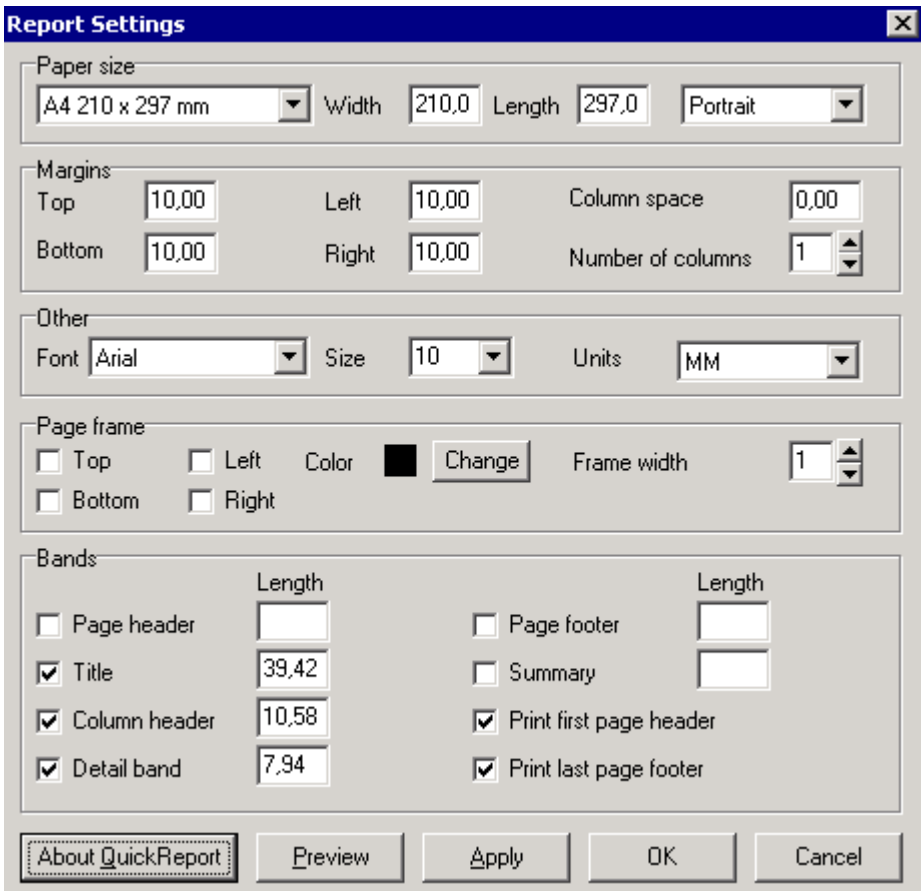


Рисунок 15.2.2 Окно настроек отчёта.

Пока теории хватит, давай посмотрим, как же действует отчётность. Открой телефонный справочник, созданный в прошлой главе. Добавь на панель кнопку печати, а обработчик его события *OnClick* мы напишем чуть позже. Сейчас мы будем создавать форму отчёта.

Создай новую форму (назовём её *ReportForm*) и сохрани в модуле *ReportFormUnit*. Сразу же подключи к этому модулю, модуль данных (*DataModule*), где у нас хранятся компоненты доступа к таблицам базы данных.

Брось на форму компонент *QuickRep*. Выдели этот компонент и в объектном инспекторе включи параметры *HasTitle* и *HasDetail* свойства *Bands*.

Теперь нужно в этих секциях расположить компоненты, которые будут отображать нужную нам информацию. На закладке *QReport* палитры компонентов доступны следующие компоненты, которые можно располагать в этих разделах:

QRLabel – надпись. Этот компонент похож на стандартный компонент *TLabel* и просто отображает нужные данные.

QRDBText – данные. Этот компонент тоже похож на *TLabel*, только он предназначен для отображения значения какого либо поля из базы данных. Тип поля базы данных должен быть совместим с текстом, т.е. может быть целым числом, строкой, датой, но не может быть картинкой или бинарными данными.

QRSysData – системная информация. Это опять копия *TLabel* только с возможностью отображать системную информацию – дату, время, номер страницы, номер строки в таблицы, общее количество страниц и т.д.

QRMemo – набор строк. Этот компонент уже похож на *TMemo* и способен отображать *Memo* данные из базы данных.

QRShape – компонент для создания обрамлений. Он чем то похож на стандартный *TShape*.

QRImage – картинка. Компонент схожий с *TImage*.

Теперь примемся за оформление отчёта. Выдели область заголовка (*Title*) и увеличь её размер где-то в два раза. В правый верхний угол области (именно области *Title*, а не компонента *QuickRep*) помести один компонент *QRSysData*. Выдели его и в свойстве дата выбери значение *qrsDateTime*. Теперь этот компонент будет отображать в правом, верхнем углу дату распечатки документа. Советую всегда это делать, чтобы ты сразу видел, какая версия документа была распечатана последней.

В центре области брось компонент *QRLabel*, увеличь шрифт в свойстве *Font* и напиши в свойстве *Caption* текст «*Распечатка строки из базы Телефонов*». Слева области можешь установить картинку *QRImage*, чтобы убедиться, что работа с ней ничем не отличается от работы с компонентом *TImage*.

Оцени результат!!! Мой результат ты можешь увидеть на рисунке 15.2.3.

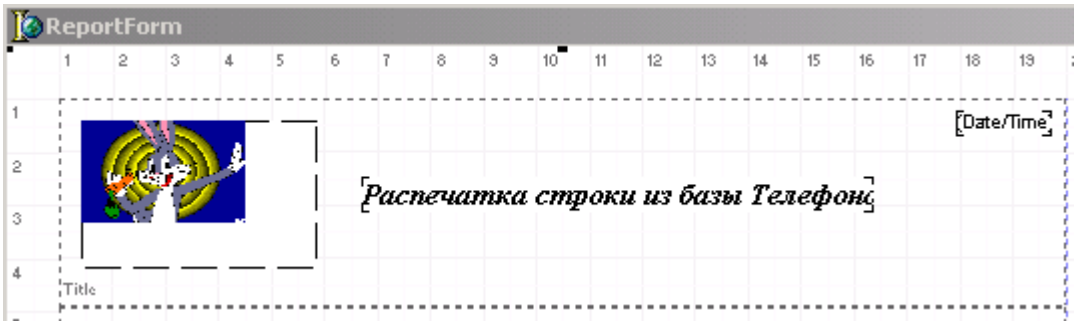


Рисунок 15.2.3 Вид заголовка нашего отчёта.

Теперь переходим к области Detail. Здесь давай выстроим с троку пять компонентов *QRLabel* и дадим им заголовки: Фамилия, Имя, e-mail, Город, Дата рождения. Под ними поставь пять компонентов *QRDBText*. У всех у них установи свойство *DataSet* в *DataModule1.BookTable*, а в свойстве *DataField* укажи соответствующие поля. У тебя должно получиться что-то похожее на рисунок 15.2.4.

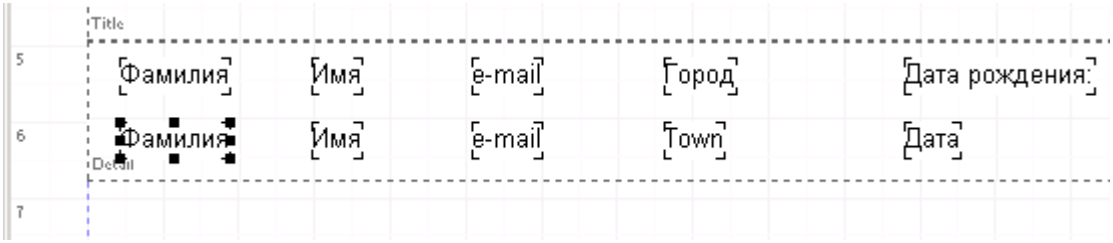


Рисунок 15.2.4 Вид блока Detail.

Теперь переходим в главный модуль и по нажатию кнопки печати пишем следующий код:

```
procedure TMainForm.PrintButtonClick(Sender: TObject);
begin
 ReportForm.QuickRep1.PreviewModal;
end;
```

В этом коде я вызываю метод *PreviewModal* компонента *QuickRep*. Этот метод модально показывает окно предварительного просмотра созданного нами документа.

Попробуй запустить программу, выделить какую-нибудь строку и нажать кнопку печати. Перед тобой откроется окно предварительного просмотра, как на рисунке 15.2.5.

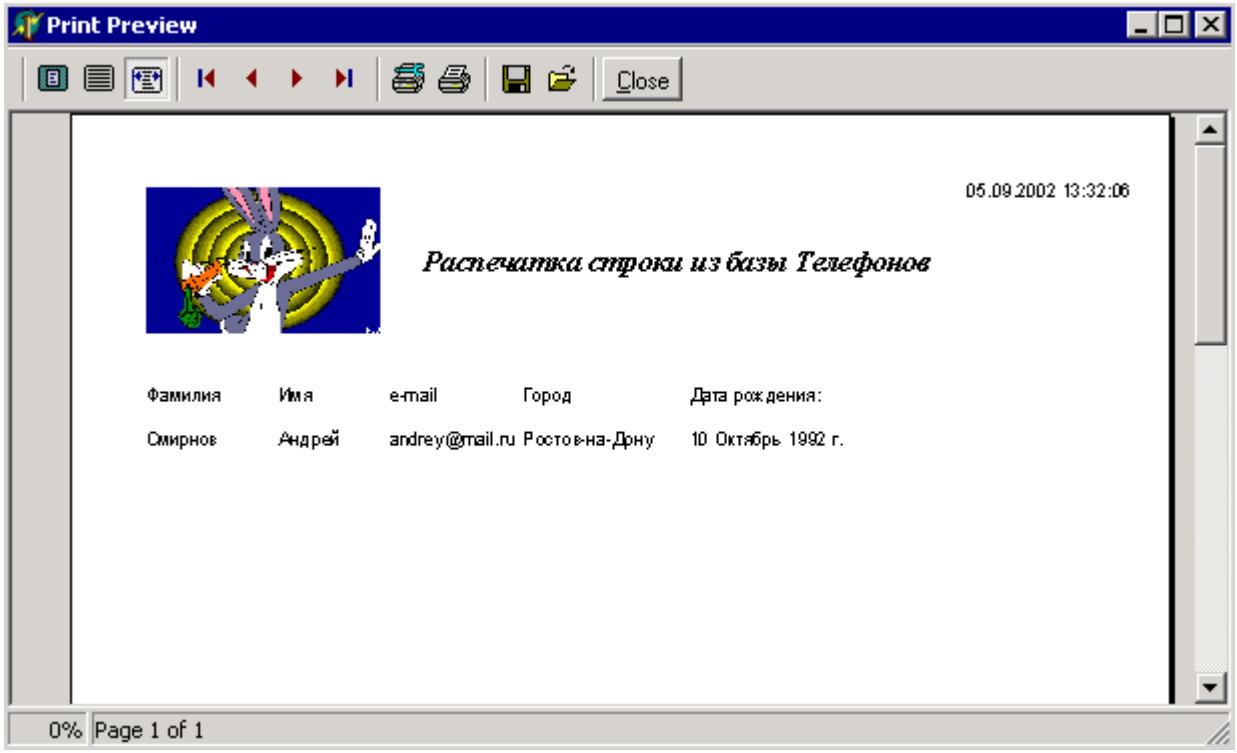


Рисунок 15.2.5 Окно предварительного просмотра.

В этом окне достаточно нажать кнопку печати, и документ будет распечатан на принтере.

*На компакт диске, в директории **\Примеры\Глава 15\QuickRep** ты можешь увидеть пример этой программы.*

15.3 Печать таблиц с помощью **Quick Reports**

Как видишь, создание отчётности с помощью Quick Report задача не сложная. За каких-то 10 минут мы создали красивый и удобный отчёт, который легко распечатать. В нём мы подготавливали к печати одну запись из таблицы. Но что делать, если нужно распечатать все записи базы данных? Неужели нужно для каждой строчки ставить отдельные компоненты? Ну конечно же нет, всё делается очень даже просто.

Открывай пример из прошлой части, сейчас мы его подкорректируем. Открой модуль *ReportForm*, где у нас находятся компоненты отчётности. Выдели *QuickRep1* и в свойстве *Bands* установи *true* у параметра *HasColumnHeader*. На форме появиться новый блок *Column Header*, который можно использовать для создания заголовков таблиц.

Теперь удерживая *Ctrl* обведи все компоненты *QRLabel* в блоке *DetailBand1*. Выбери из меню *Edit* главного меню Delphi пункт *Cut*, чтобы вырезать компоненты в буфер обмена. Теперь выдели блок *ColumnHeaderBand1* и выбери из меню *Edit* главного меню Delphi пункт *Paste*, чтобы вставить вырезанные компоненты из буфера обмена в блок *ColumnHeaderBand1*.

Снова выделяем компонент *QuickRep1* и в свойстве *DataSet* указываем таблицу *DataModule1.BookTable*. Если сделать это, то компонент *QuickRep1* автоматически будет

перебирать все записи из этой таблицы и использовать их в компонентах, которые стоят в блоке *DetailBand1*. Как я уже говорил, этот блок предназначен для создания строк таблиц, так что теперь ты можешь увидеть это на практике.

Запусти программу и нажми кнопку печати. Перед тобой должно открыться окно, похожее на рисунок 15.3.1. Как видишь, блок заголовка и *ColumnHeaderBand1* распечатались только по одному разу, а блок *DetailBand1* был напечатан для каждой строки нашей таблицы. Блок заголовков колонок будет печататься по одному разу вверху каждой страницы, а после него будут идти строки таблиц.

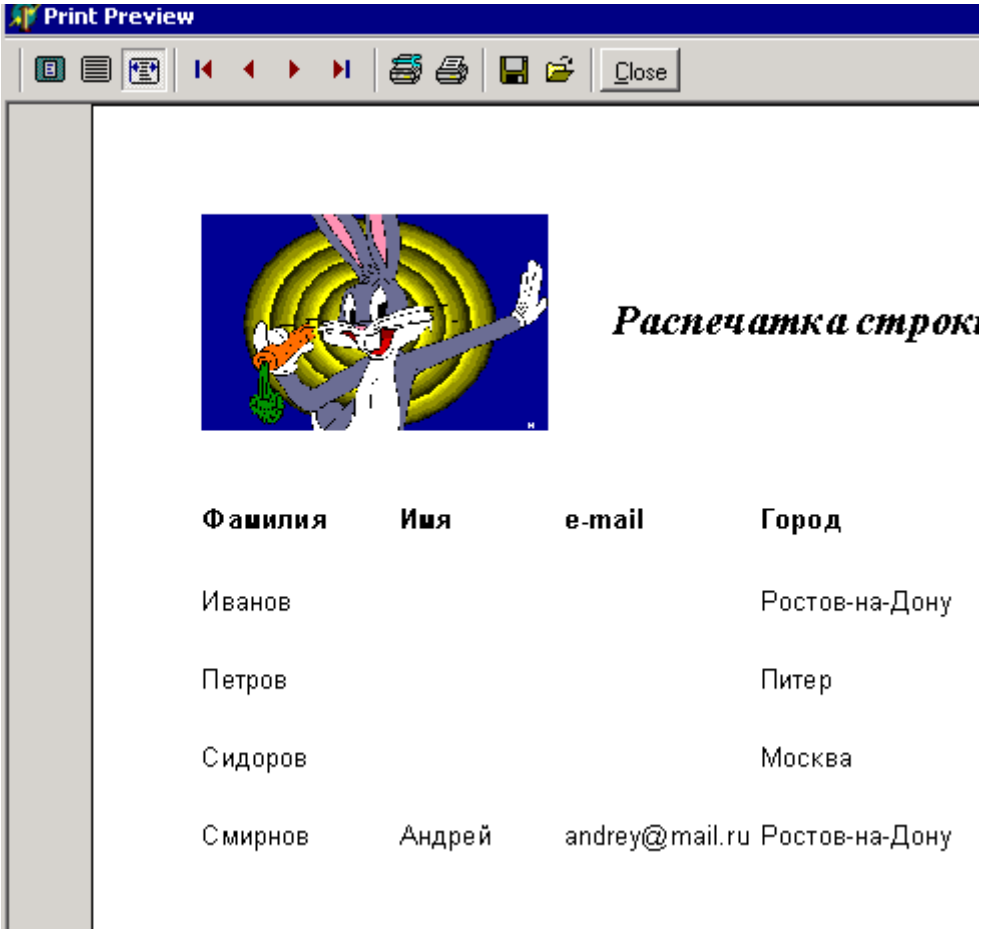


Рисунок 15.3.1 Окно предварительного просмотра.

В принципе, на этом можно было бы закончить разговор но мы же создаём таблицу, а для этого нужно нарисовать сетку. Вот тут можно заметить единственный недостаток отчётности с помощью Quick Report – каждую ячейку придётся рисовать отдельно, с помощью компонентов *QRShape*. В примере на диске я сделал рамку. Правда она получилась немного некрасивая, но у меня просто не было времени её поправлять. Если захочешь, то можешь украсить её на своё вкус, заодно и потренируешься работать с отчётами.

*На компакт диске, в директории **\Примеры\Глава 15\QuickRep1** ты можешь увидеть пример этой программы.*

15.4 Печать связанных таблиц.

У нас уже получился достаточно хороший отчёт, но теперь я хочу ещё более усложнить задачу. У нас же в телефонном справочнике используется связанная таблица и хотелось бы, чтобы в выходном документе печатались все телефоны, принадлежащие людям!!! Можно снова написать SQL запрос, который будет создавать сводную таблицу из двух и потом использовать этот запрос для отчёта, вместо таблиц, но это не выход.

Брось на нашу форму отчёта ещё один компонент – *QRSubDetail* с закладки *QReport*. Этот компонент предназначен для перебора данных относящихся к подчинённым таблицам. Установи у него следующие свойства:

DataSet – здесь установи *DataModule1.TelephonTable*, чтобы связать блок к с таблицей «Телефоны», которая у нас является подчинённой к основному справочнику.

Master – здесь нужно указать главный компонент с основными данными. Выбери в этом свойстве *QuickRep1*.

Теперь брось на этот блок компонент *QRLabel*, чтобы сделать надпись «Телефон». Справа от него брось компонент *QRDBText*. У него установи свойство *DataSet* в *DataModule1.TelephonTable*, а свойство *DataField* в «Телефон».

Рисунок 15.4.1 Окно предварительного просмотра.

Теперь можешь запускать программу. Нажимай кнопку печати, и если ты ввёл в свою базу данных хоть какие-нибудь телефоны, то у тебя должно получиться нечто похожее на рисунок 15.4.1.

*На компакт диске, в директории **\Примеры\Глава 15\QuickRep2** ты можешь увидеть пример этой программы.*

15.5 Дополнительные прибомбасы.

Здесь я покажу некоторые прибомбасы, которые можно добавить в свои отчёты. Я покажу некоторые компоненты, которые не только украшают любой отчёт, но и делают его более удобным и мощным.

Первый компонент в моём обзоре – *QRExpr*. Этот компонент очень удобен для создания вычисляемых полей именно для отчёта. Вычисления будут происходить автоматически и практически не влияют на скорость работы самой программы.

У компонента есть свойство *Expression*. Если дважды щёлкнуть по нему, то перед тобой откроется окно, в котором можно создавать достаточно сложные расчёты. На рисунке 15.5.1 ты можешь увидеть это окно. Если нажать на кнопку *Database field*, то перед тобой откроется окно, в котором можно выбрать таблицу (таблица должна находиться на этой же форме) и поле, которое должно участвовать в расчёте.

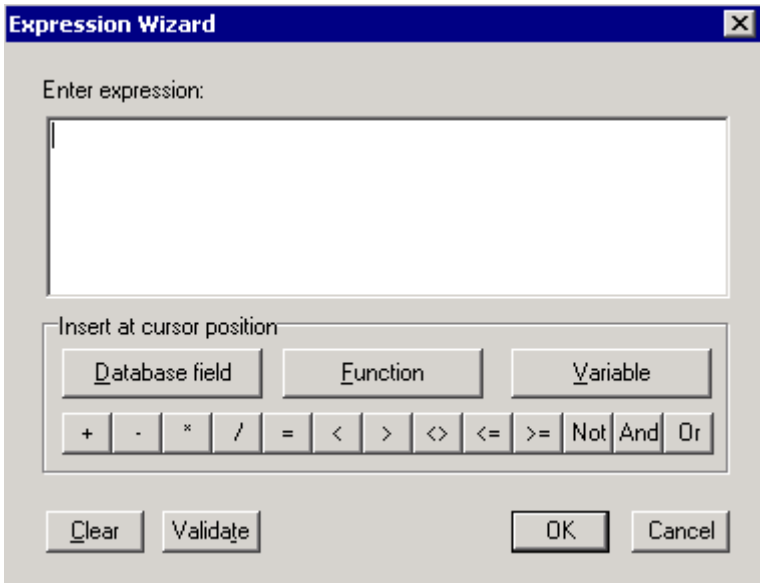


Рисунок 15.5.1 Окно предварительного просмотра.

Если нажать на кнопку *Function*, то ты увидишь громадный список доступных функций. Под кнопкой *Variable* спрятаны переменные, которые могут так же помочь в расчётах.

Всё это расписывать нет смысла, потому что здесь возможностей просто куча плюс ещё тысяча. По работе с Quick Report можно писать отдельную книгу, а моя задача показать тебе основные возможности и направить на путь истинный. Что-то я заговорил, как самый настоящий проповедник.

Теперь посмотрим на компонент QRSysData. Мы уже использовали его и в свойстве *Data* ставили параметр *qrsDateTime*, чтобы можно было видеть дату и время распечатки документа. Но тут есть ещё несколько полезных параметров:

qrsDate – дата распечатки.
qrsDetailCount – количество строк в таблице.
qrsDetailNo – номер строки в таблице.
qrsPageNumber – текущий номер страницы.
qrsReportTitle – заголовок отчёта.
qrsTime – время распечатки отчёта.

Теперь я хочу показать тебе, как можно ещё больше упростить создание отчёта. В Delphi есть мастер, который может облегчить создание отчётов. Для его вызова выбери из главного меню *File* пункт *New* и затем пункт *Other*. В появившемся окне перейди на закладку *Business* и выбери пункт *QuickReport Wizard* (рисунок 15.5.2).

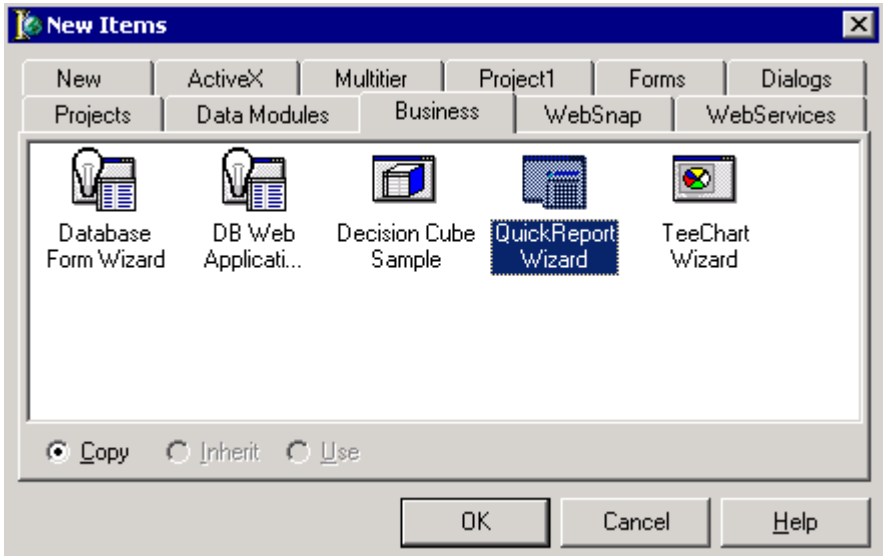


Рисунок 15.5.2 Окно предварительного просмотра.

В принципе, мастер достаточно прост, но при использовании ADO таблиц он абсолютно не подходит. Если ты будешь работать с старыми базами типа dbf или paradox, (об этом в следующей главе) то этот мастер принесёт свои плоды, но при работе с ADO, он практически бесполезен.

Пока что нам больше подходят шаблоны, которые есть в том же окне создания новой формы, но на закладке *Forms* (смотри рисунок 15.5.3). Здесь три шаблона (все они содержат имя QuickReport), содержащие всё необходимое для создания отчёта и хорошо документированные поля. Попробуй посмотреть их сам.

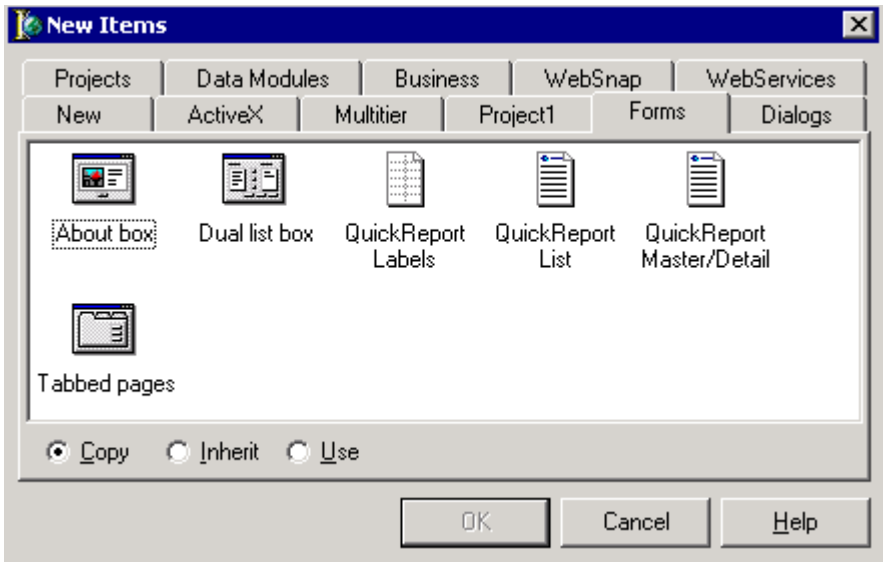


Рисунок 15.5.3 Окно шаблоном.

Глава 16. Работа с DBF, Paradox и XML базами данных.

Я уже сказал, что очень люблю работать с Access базами данных, но моё пристрастие не распространяется на всех. Некоторые программисты любят старые таблицы Paradox и DBF, а некоторые уже перебазировались на новый и перспективный формат XML. Про эти форматы нельзя забывать, поэтому я не могу упустить их из виду.

Если ты думаешь, что в жизни хватит одного только Access, то сильно ошибаешься. Допустим ты пришёл на новую работу, где люди уже работают со старым форматом DBF. Из-за одного тебя никто не будет менять уже устоявшихся традиций и тебе придётся смириться с консерватизмом (ну и словечко), точнее сказать со старьём.

Конечно же, можно попытаться убедить начальство в том, что DBF и Paradox не надёжны и у них регулярно рушатся индексы. Можно убедить в том, что XML ещё не на столько гибок и отстаёт в возможностях, но на это нужно время и хотя бы какой-то промежуток времени тебе придётся работать со старыми форматами. А если не удастся убедить, то будешь работать так вечно, пока не поменяешь работу. Поверь мне, это проверенный вариант.

Так что не советую тебе расслабляться и переворачивать страницу, потому что эта глава достаточно важна. Тем более, что работа с другими форматами данных очень похожа на работу с ADO.

16.1 Создание таблицы Paradox.

Paradox и DBF – это таблицы, а не базы данных. Если в одной базе Access могло храниться несколько таблиц, то у Paradox и DBF в одном файле храниться одна таблица. К тому же индексы хранятся отдельно от таблицы, что накладывает определённые неудобства.

Создание и работа с таблицами Paradox и DBF абсолютно одинаковы, поэтому я буду всё показывать на примере Paradox, потому что к нему отношусь с большим уважением.

Для создания первой базы данных тебе понадобиться запустить отдельную программу *Database Desktop*, который входит в поставку с Delphi. После запуска выполни следующие действия:

1. Выбери меню File затем New и наконец Table
2. В появившемся окне, выбери из списка Paradox 7 .
3. Нажми "OK".

Перед тобой откроется окно как на рисунке 16.1.1. Можно сказать, что первая база данных готова. Теперь ты должен заполнить её поля, но сначала рассмотрим появившийся перед нами диалог.

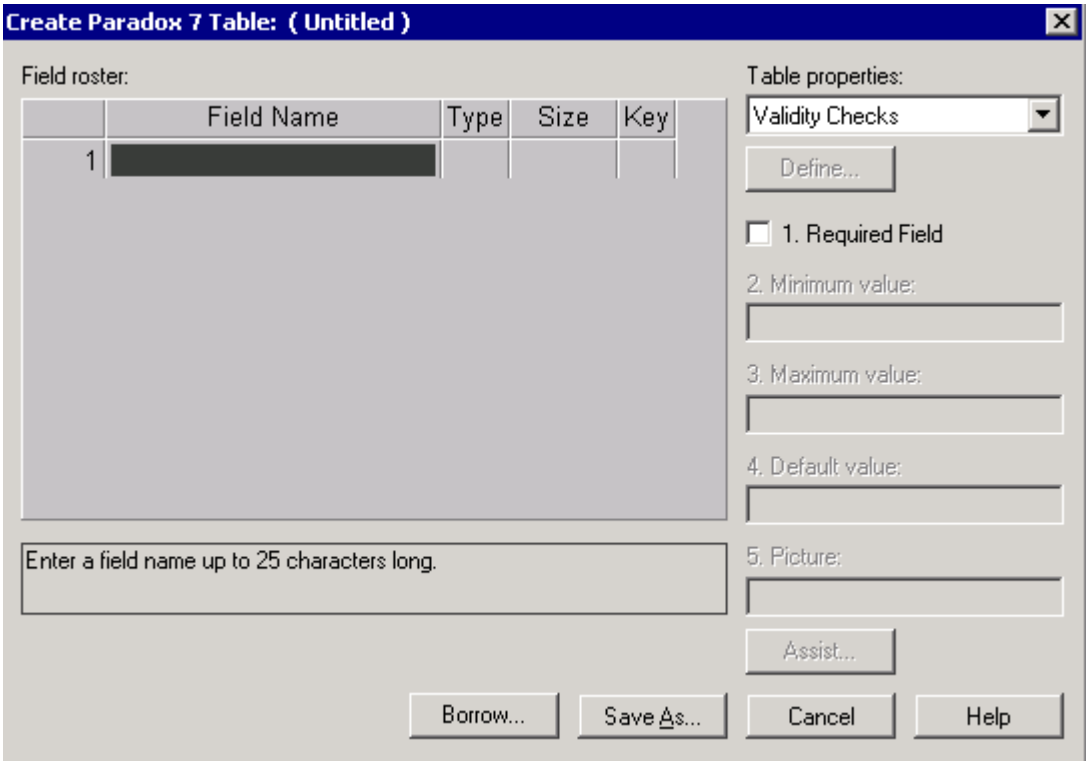


Рисунок 16.1.1. Окно редактирования полей таблицы Paradox

1. *Номер по порядку*. *Database Desktop* генерирует его автоматически и изменять ты его не можешь.

2. *Имя поля*. Здесь ты можешь называть свои поля как угодно, но только английскими буквами и нельзя использовать пробелы. Так что в отличии от Access, здесь могут быть проблемы с нормальным именованием полей.

3. *Тип поля*. Щёлкни в этой колонке правой кнопкой мыши и перед тобой появиться меню со всеми допустимыми типами, тебе необходимо только выбрать нужный.

4. *Размер*. Это размер поля. Не у всех типов полей можно менять размер, у большинства он задан жёстко. Размер в основном меняется у строковых типов (Alpha), бинарных (binary) и др.

5. *Ключ*. Если ты дважды щёлкнешь по этой колонке, то текущее поле станет ключевым, то есть по умолчанию по нему будет отсортирована вся таблица. Ключевыми могут быть только первые поля, то есть второе поле сможет быть ключевым только вместе с первым. Без ключевого поля невозможно добавлять новые записи в таблицу. Точно также, как и в Access создание ключа обязательно.

В качестве примера возьмём классический пример - приём заказов. Самое главное, это правильно представить себе и создать структуру базы данных. В нашем случае будет вестись учёт следующих полей:

1. ФИО Покупатель
2. Адрес
3. Дата заказа товара
4. Наименование заказанного товара
5. Количество заказанного товара

Теперь нужно продумать структуру. Если создать все эти поля в одной таблице, то будет не совсем эффективно. Если один и тот же покупатель возьмёт два товара, то у обеих строчек 1-е и 2-е поле будут содержать одинаковые данные. Будет лучше, если мы вынесем первые два поля в отдельную таблицу и потом будем использовать две связанные таблицы.

Итак, наша база данных будет состоять из двух таблиц. В первой будут следующие поля (после тире стоит тип поля, а в скобках размер):

Ключ 1 - autoincrement (ключевое)
ФИО Покупатель - alpha (размер 50)
Адрес - alpha (размер 50)
И соответственно 2-я:

Ключ 1 - autoincrement (ключевое)
Ключ 2 - Integer
Дата заказанного товара - date
Наименование заказанного товара - alpha (размер 20)
Количество заказанного товара - Integer

"*Ключ 1*" - это будет уникальное ключевое поле в обеих таблицах, поэтому поставь им значок ключевого. "*Ключ 2*" во второй таблице будет связан с "*Ключ 1*" из первой, пока его не делай ключевым, мы это сделаем чуть позже. Создай эти таблицы. Первую назови mast.db , а вторую child.db . Только помни, что имена полей должны писаться английскими буквами.

После того, как создашь вторую таблицу, ты должен сделать "*Ключ 2*" во второй таблице индексным, чтобы можно было связать две таблицы с помощью этого поля. Для этого нужно открыть таблицу child.db и из меню *Table* выбрать пункт *Restructure* . Перед тобой должно открыться окно, которое ты видел при создании полей таблицы (рисунок 16.1.2). теперь мы в этом же окне будем вносить изменения, а именно добавлять индекс.

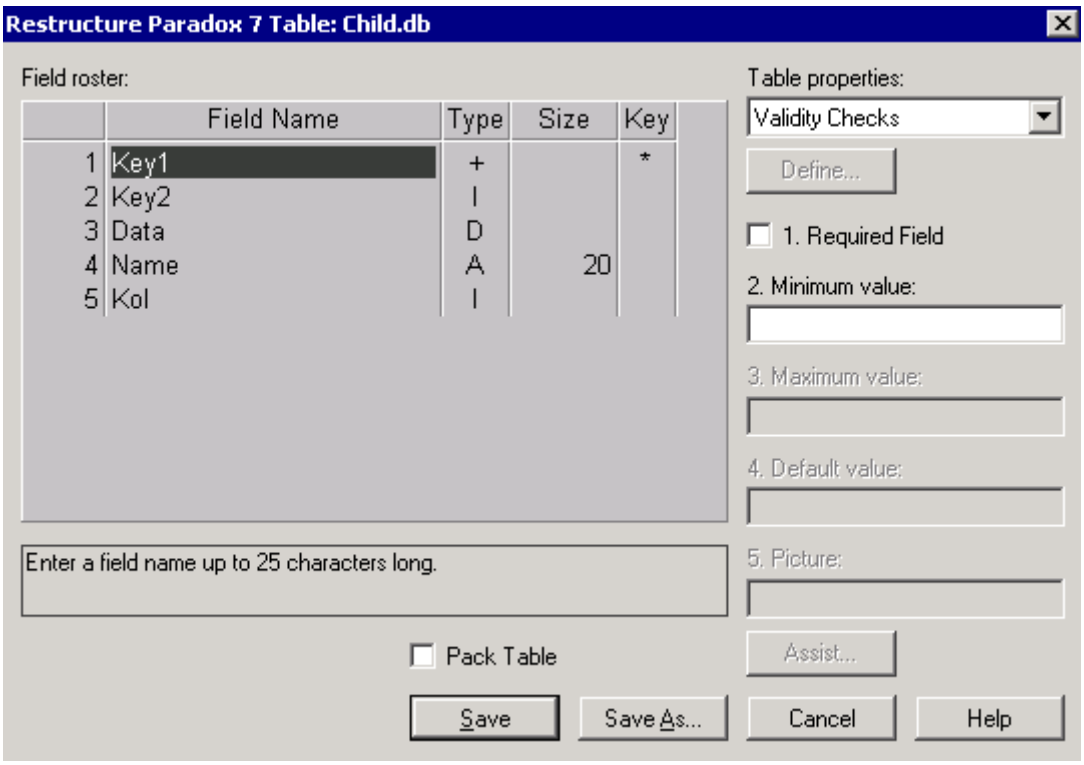


Рисунок 16.1.2. Окно редактирования полей таблицы Paradox

В выпадающем списке *Table properties* выбери *Secondary Indexes* (дополнительные индексы) и нажми кнопку *Define* (определить). Выбери свой второй ключ и перемести его в список *Indexed fields* (индексированные поля). Для этого надо нажать кнопку с изображённой стрелкой вправо (рисунок 16.1.3). Можешь нажимать "OK". У тебя запросят имя индекса, введи, например *hhh* и снова жми "OK". После этого сохраняй таблицу. Индексы готовы, теперь перейдём к последнему этапу подготовки базы.

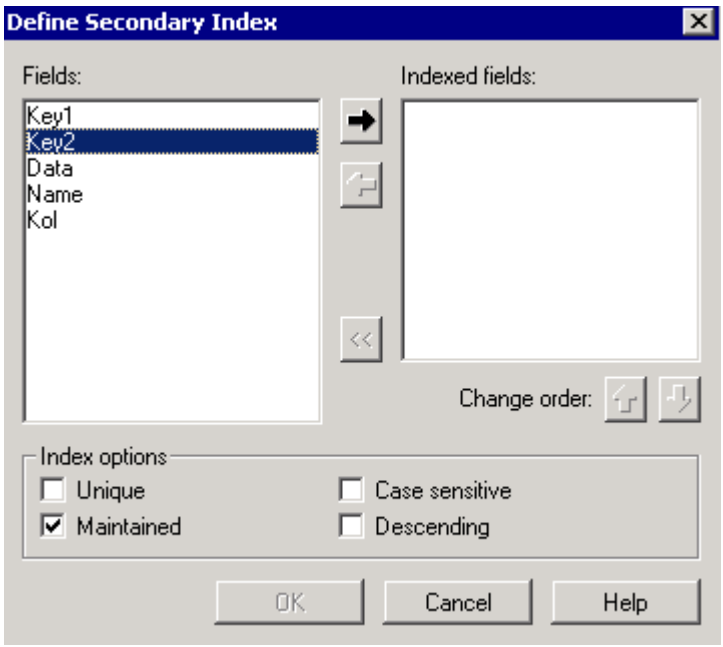


Рисунок 16.1.3. Окно редактирования полей таблицы Paradox

Теперь запусти SQL Explorer . Его главное окно ты можешь увидеть на рисунке 16.1.4. Здесь создаются псевдонимы (Alias) разным директориям с таблицами. Эти

псевдонимы сохраняются в реестре и потом программы при запуске могут по этим псевданимам найти директорию таблицы и прочитать необходимые настройки, которые надо использовать при доступе к данным.

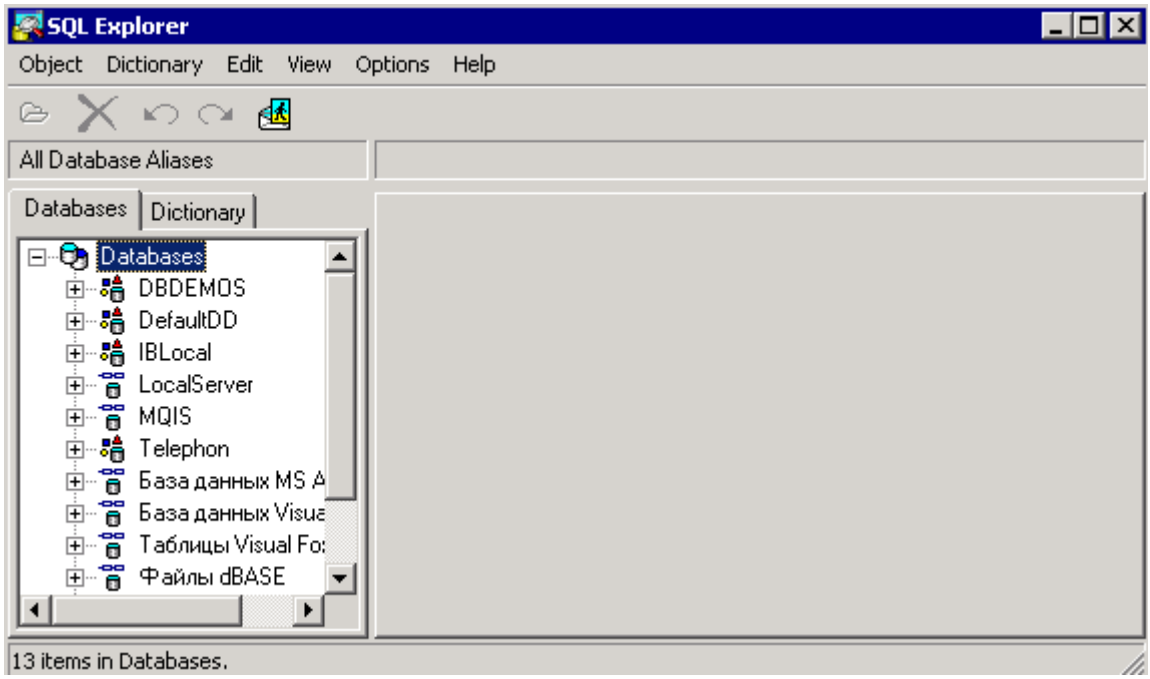


Рисунок 16.1.4. Окно SQL Explorer

Чтобы создать новое имя, нужно из меню *Object* выбрать пункт *New*. Перед тобой откроется окно, в котором *Database driver name* должен быть *STAMDART*. Теперь жми "*OK*". Это создаст новый *Alias* (имя). Переименуй его в "*Sales1*".

Теперь в правой половине окна щёлкни по строке *PATH*. Перед тобой откроется окно выбора директории. Выбери ту, где находятся созданные нами таблицы и нажми "*OK*". Для сохранения того, что ты создал, выбери из меню Object пункт Apply.

Вот теперь таблицы готовы, имя создано и можно запускать Delphi, чтобы написать первую программу, которая будет работать с таблицами *Paradox*.

Создай новый проект. Помести на него из закладки *Data Access* два компонента *DataSource* и с закладки BDE два компонента *TTable* палитры компонентов. Для первого *DataSource* установи свойство *DataSet* в *Table1*, а у второго *Table2*. Теперь у *Table1* измени следующие свойства (желательно в такой последовательности):

DatabaseName выставь *Sales1* (это Alias, который мы создали в SQL Explorer).

TableName выставь в *mast.db*. Если ты всё правильно сделал, то имя этой базы будет в выпадающем списке этой строки.

Active выставь в *true*, чтобы активизировать таблицу.

То же самое сделай и с *Table2*, только в *TableName* выставь *child.db*. После того как ты всё это сделаешь, из палитры компонент *DataControls* поставь на форму два компонента *DBGrid*. В свойстве *DataSource* у одной из них выставь *DataSource1*, а у другой *DataSource2*. Постарайся расположить компоненты, как на рисунке 16.1.5.

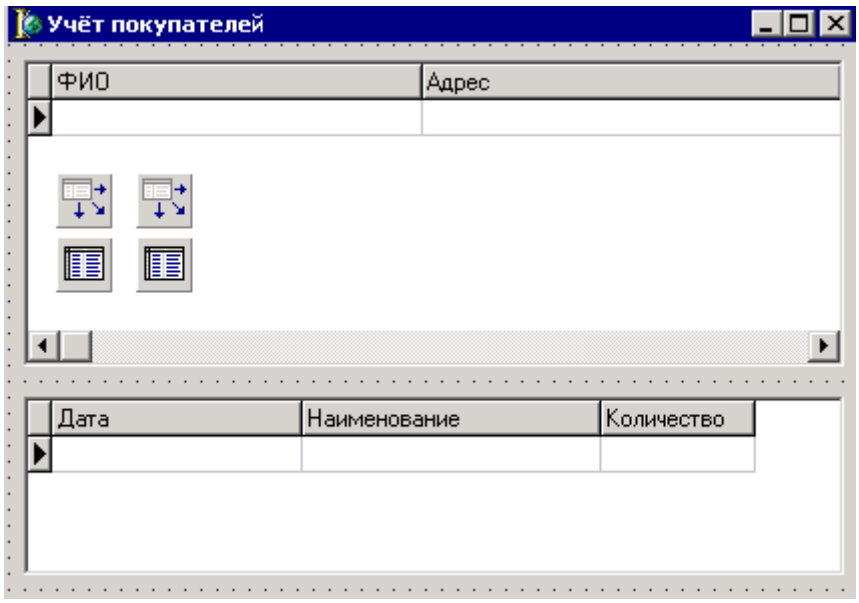


Рис 16.1.5. Расположения компонентов на форме

Уже можно запустить программу и убедиться, что всё работает. Но всё ещё не очень красиво, ведь таблицы ещё не связаны, и никому не нужно видеть ключи, да и подписи на английском. Выдели по *Table2*, здесь у тебя должна находиться child.db. В свойстве *MasterSource* выставь *DataSource1*. После этого дважды щёлкни в поле "MasterFields", перед тобой откроется окно как на рисунке 16.1.6. В верхнем списке *Available Indexes* выбери имя, которое ты задал, когда индексировал *Key2*, в данном случае это *hhh*. Теперь выдели *Key2* в левом окне и *Key1* в правом и нажми кнопку *Add*. Так ты добавил связь, которая будет использоваться между таблицами. Нажимай *OK* и снова ставь в свойстве этой таблицы *Active* в *true*, потому что при наведении связи таблица автоматически закрылась.

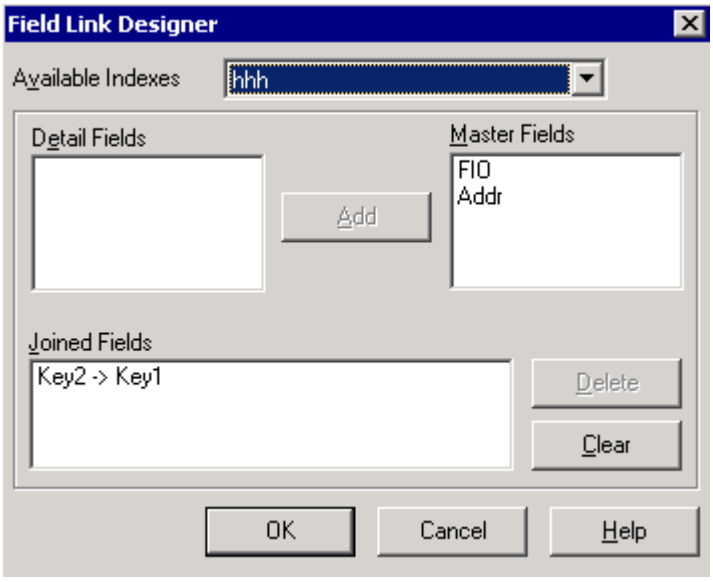


Рис 16.1.6. Окно связывания таблиц

Теперь дважды щёлкни по компоненту *Table2*. Перед тобой откроется окно (на рисунке 16.1.7) редактора свойств полей. С подобным окном мы работали, когда писали примеры с помощью ADO. В этом окне щёлкни правой кнопкой мыши, и в появившемся меню выбери пункт меню *Add All Fields*. После этого, все поля таблицы будут добавлены

в это окно. В свойствах *Key1* и *Key2* установи свойство *Visible* в *false*, чтобы спрятать индексы. В свойстве поля *Data* установи *DisplayFormat* в *dddddd*, а *EditMask* в *99/99/9999*. У всех полей свойство *DisplayLabel* отвечает за имя отображающее в компонентах это поле, поэтому напиши здесь у всех нормальные русские имена.

Рис 16.1.7. Окно свойств полей

После того, как закончишь с этой таблицей, проделай подобные операции со второй. Нужно будет так же спрятать ключевые поля и написать нормальные подписи к полям.

Твоё первое приложение работающее с базой данных Paradox готово. Как видишь, работа с такими базами практически не отличается от работы с ADO. Единственное отличие – используются другие компоненты, а именно закладка BDE. Но большинство свойств компонента *TTable* похожи на свойства компонента *TADOTable*. У них есть все необходимые свойства, такие как *First*, *Next*, *Prev*, *Last*, *Edit*, *Paste* и многие другие.

Для создания запросов к таблицам Paradox и DBF нужно использовать компонент *Query* с закладки BDE. У него так же есть свойство SQL, в котором надо писать SQL запрос. Но так как таблицы Paradox и DBF – это таблицы, а не базы данных, то не надо указывать никаких строк подключение. Достаточно только ввести запрос (имя таблицы ты укажешь в запросе в поле *from*) и выполнить его сделав свойство *Active* равным *true*.

С таблицами Paradox можно так же создавать поисковые и вычисляемые поля и здесь особых отличий от ADO нет. Всё это связано с тем, что в обоих компонентах использована одна и та же основа.

На компакт диске, в директории ***\Примеры\Глава 16\Paradox*** *ты можешь увидеть пример этой программы.*

В примере я не указывал у таблиц псевдонимы (Alias). Это не обязательно, но просто так принято для удобства. Если не указывать у таблиц в свойстве *DatabaseName* никакого псевдонима, то таблица будет искаться в текущей директории, где и запускной файл. Можно ещё указывать полный путь в свойстве *TableName*.

16.2 Русификация таблиц Paradox и DBF.

Для работы с таблицами Paradox и DBF мы используем компоненты с закладки BDE. Это не просто название закладки – это целый набор драйверов, программная надстройка, через которую происходит работа с таблицами. Эта

надстройка устанавливается вместе с Delphi или её можно найти на диске отдельным установочным файлом.

По умолчанию BDE работает с таблицами в кодировке не поддерживающей русский язык. Для русификации нужно запустить программу BDE Administrator. Её главное окно похоже на SQL Administrator. Перейди в этом окне на закладку Configuration (рисунок 16.2.1) и открой в дереве ветку *Configuration – Drivers – Native*. Здесь выбери пункт Paradox и в правой половине окна ты увидишь настройки доступа к таблицам Paradox.

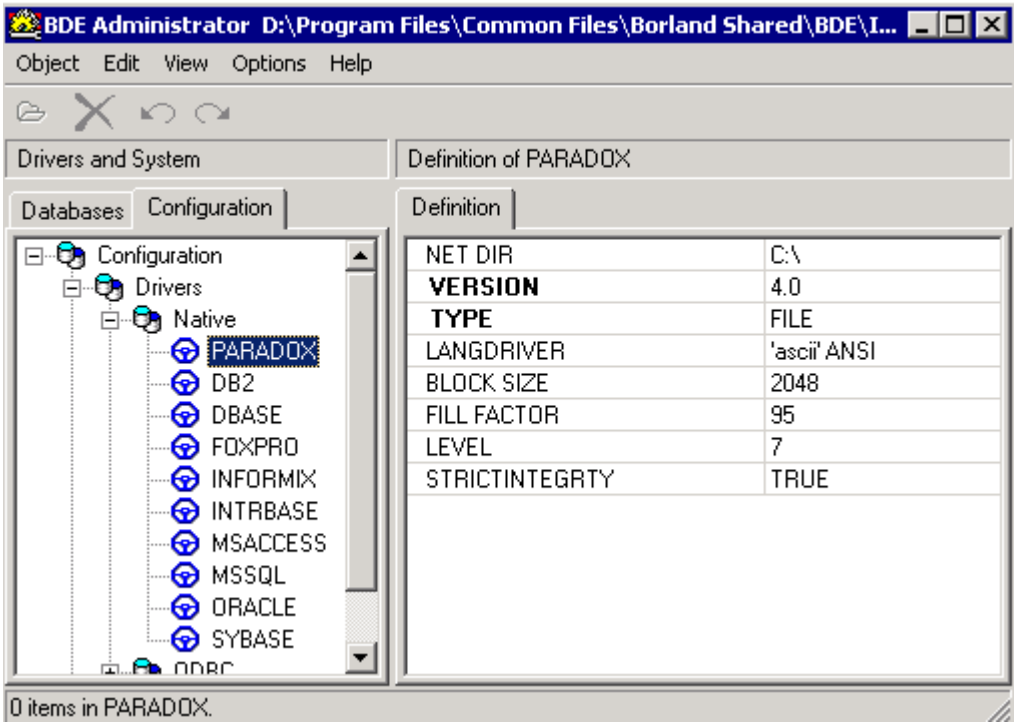


Рис 16.3.1. Обновлённая форма программы

Здесь нужно изменить параметр *LANGDRIVER* – драйвер языка. По умолчанию у меня стоит *asci*, при котором русские буквы превращаются в непонятно что. Выбери у этого параметра в выпадающем списке *Pdox ANSI Cyrillic*. Теперь щёлкни в окне слева (в дереве настроек) по пункту Paradox и выбери в появившемся меню пункт Apply, чтобы сохранить настройки. После этого появиться окно с подтверждением о сохранении данных и после этого предупреждение о том, что для получения эффекта нужно перезапустить все программы работающие с BDE.

Теперь выбери в дереве пункт *DBASE* и у него в настройках выбери драйвер языка *dBASE RUS cp866*. Сохрани эти настройки.

Теперь твои таблицы будут правильно отображать русские буквы и ты сможешь работать с ними на родном и понятном нам языке.

16.3 Моментальный поиск.

Поиск одного поля с помощью SQL запроса или фильтра, просто ужасно глупая затея. Этот поиск будет проходить достаточно долго долго. Но есть способ лучше - Рондо. Точнее сказать поиск по ключевым полям. Этот поиск происходит практически моментально, даже на больших базах данных. Недостатки - отсутствие шаблонов и не возможно использовать привязанную таблицу.

Моментальный поиск мы не рассматривали в главе про ADO, потому что это особенность Paradox и DBF таблиц. Так что я сделаю это сейчас.

За счёт чего достигается большая скорость поиска? Всё очень просто, за счёт индексации. Так можно искать только по индексированным полям. Ну, хватит лишних слов, давай переходить к делу.

Для работы нам понадобиться всё те же таблицы из прошлого примера.

В качестве основы я взял только одну таблицу - mast.db. Как я уже говорил, для быстрого поиска нужно, чтобы поле было проиндексировано, поэтому добавь ещё один вторичный индекс для поля FIO (где у нас храниться фамилия). Теперь у компонента *TTable*, к которой привязана эта таблица (у меня это *Table1*), свойство *IndexFieldName* измени на *FIO*. Всё таблица готова к употреблению.

Теперь давай добавим на форму одну строку ввода, в которой будем вводить данные, которые надо искать и кнопку, по нажатию которой будет запускаться поиск. На рисунке 16.3.1 показана обновлённая форма нашей программы.

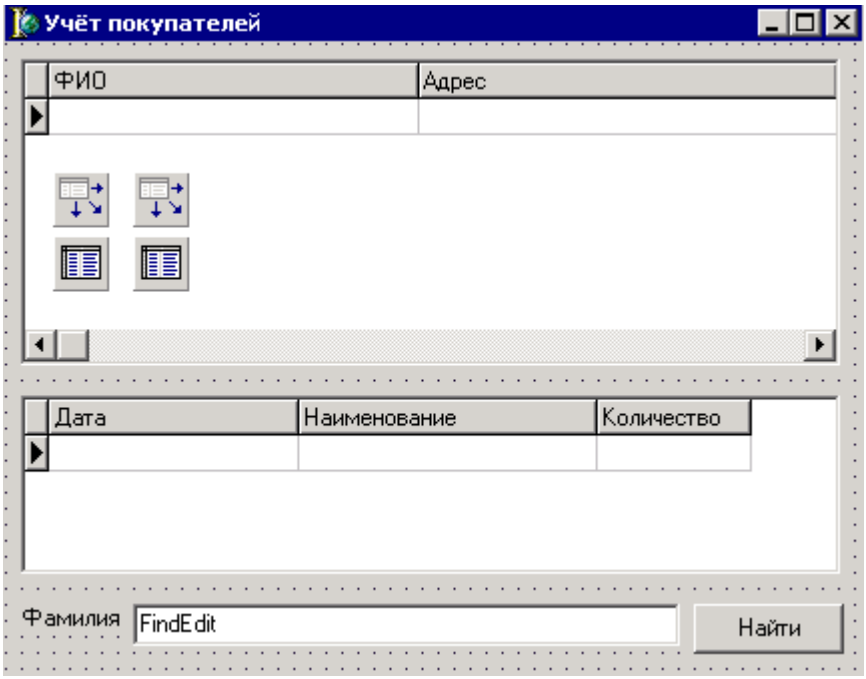


Рис 16.3.1. Обновлённая форма программы

Теперь создаём обработчик события *OnClick* для кнопки и пишем там следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 Table1.SetKey;
 Table1FIO.AsString:=FindEdit.Text;
 Table1.GotoKey;
end;
```

Здесь всё очень просто. Первая строка говорит, что сейчас я буду устанавливать ключ. Вторая строка устанавливает его. Заметь, что это происходит как изменение содержимого поля, но поле не меняется, потому что мы вызвали перед этим SetKey. И, наконец, последняя строка говорит, что надо найти установленный ключ.

Всё время мы создавали одиночные вторичные ключи (состоящие из одного поля), но ты можешь с помощью DatabaseDesktop создавать ключи состоящие из любого количества полей. Для этого в DatabaseDesktop, в выпадающем списке *Table properties* нужно выбрать *Secondary Indexes* и нажать кнопку *Define* . Теперь выбери любое поле, и перемести его в список *Indexed fields*, затем другое поле и снова перемести его в список *Indexed fields*. И так хоть все поля.

Если у тебя установлен ключ из нескольких полей, то ты должен между вызовами *SetKey* и *GotoKey* установить все ключи. Иначе результат может быть ошибкой.

Есть ещё одна процедура, с которой я хочу тебя познакомить - *GotoNearest*. Если *GotoKey* не находит нужного ключа, то генерируется ошибка. *GotoNearest* тоже производит поиск ключевого поля, но если поле не найдено, то ошибка не генерируется, а ищется ближайший похожий ключ.

 На компакт диске, в директории ***\Примеры\Глава 16\FastFind*** *ты можешь увидеть пример этой программы.*

16.4 Создание псевдонимов.

В первом примере работы с таблицами paradox я показал тебе, как с помощью SQL Explorer можно создавать псевдонимы. Но что делать, если в твоей программе используются псевдонимы, а на компьютере пользователя их нет? Напрашивается ответ – создать их. А теперь представь себе, что пользователь живёт в другом городе и абсолютно ничего не понимает в компьютере. В этом случае, ты не сможешь объяснить ему процесс создания псевдонимов по телефону, а из этой ситуации нужно как-то находить выход. А выход простой – создать псевдонимы программно. Пуская твоя программа следит за наличием на компьютере пользователя псевдонимов.

Давай создадим маленькую программу, которая будет создавать какой-нибудь произвольный псевдоним таблиц. Запусти Delphi и создай новый проект.

Брось на форму компонент *TSession* с закладки BDE. Если ты работаешь с базой данных, то этот объект всегда создаётся автоматически без твоего ведома. В нём хранится низкоуровневая информация о BDE, драйверах и многом другом. Если ты хочешь изменить какие-то значения указанные в этом объекте по умолчанию, то тебе необходимо:

1. Поставить этот компонент на форму.
2. Указать любое имя в его свойстве *SessionName*.
3. Всем компонентам *TTable* и *TQuery*, для которых будут действовать установленные тобой значения, в поле *SessionName* нужно указать имя объекта *TSession*.

Таким образом ты сам создашь сессию.

Для нашего примера достаточно первых двух действий, потому что у нас не будет компонентов *TTable* или *TQuery*. Посмотри на рисунок 16.4.1 и сконструируй форму, подобную этой. На моей форме всего лишь две кнопки, один компонент *TListBox* и один компонент *TSession*. Ты можешь расположить их по другому, главное чтобы они присутствовали и тебе удобно было с ними работать.

Рис 16.4.1. Обновлённая форма программы

Создай обработчик события *OnShow* для главной формы и напиши в нём следующее:

```
procedure TForm1.FormShow(Sender: TObject);
var
 str:TStrings;
begin
 Str:=TStringList.Create;
 Session1.GetAliasNames(Str);
 ListBox1.Items.Assign(Str);
 Str.Free;
end;
```

Здесь я получаю список всех доступных в системе псевдонимов и вставляю этот список в компонент *TListBox*. Рассмотрим построчно. Сначала я объявляю переменную *Str* типа *TStrings*.

Функция *GetAliasNames* компонента *TSession* возвращает все доступные системе псевдонимы. В качестве параметра передаётся указатель на объект типа *TStrings*, куда запишутся весь список.

ListBox1.Items.Assign(Str) - копирую список в элементы ListBox1.

И последняя строка освобождает переменную *Str*.

Теперь создадим новый псевдоним стандартного типа. К этому типу относятся таблицы *Paradox* и *DBF*. Я создаю псевдоним по нажатию первой кнопки. Вот соответствующая процедура:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 str:TStrings;
begin
 Session1.AddStandardAlias('VROnline','c:\','Paradox');
 Str:=TStringList.Create;
 Session1.GetAliasNames(Str);
 ListBox1.Items.Assign(Str);
 Str.Free;
end;
```

Создание происходит с помощью функции *AddStandardAlias* компонента *TSession*. В качестве первого параметра, ты должен указать имя нового псевдонима. Второй - путь к базам данных, которые будут связаны с псевдонимом. Третий - драйвер, который будет использоваться по умолчанию.

После добавления, я снова получаю уже обновлённый список доступных псевдонима и копирую его в компонент *TListBox*.

Теперь создадим псевдоним нестандартного типа. Для примера будет использоваться псевдоним для базы данных Interbase. Эти базы данных в книге рассматриваться не будут, но для примера я покажу, как создаются такие псевдонимы. Он будет создаваться по нажатию второй кнопки:

```
procedure TForm1.Button2Click(Sender: TObject);
var
 L: TStringList;
begin
 L := TStringList.Create;
 try
  with L do
   begin
    Add('SERVER NAME=IB_SERVER:/PATH/DATABASE.GDB');
    Add('USER NAME=MYNAME');
   end;
  Session1.AddAlias('NewIB', 'InterBase 4.x Driver by Visigen',L);
 finally
  L.Free;
 end;
end;
```

Процедура *AddAlias* компонента *TSession* создаёт псевдоним нестандартного типа. Первый параметр - имя псевдонима. Второй - имя драйвера. Третий параметр - указатель на *TStringList*, в котором хранятся дополнительные настройки. Дополнительные параметры хранятся в виде строк (например '*SERVER NAME= IB_SERVER: /PATH/DATABASE.GDB*'). В строке указывается имя параметра и после знака равно его значение. Параметры могу отличаться, в зависимости от драйвера. Чтобы узнать, какие параметры доступны, можно создать пробный псевдоним в SQL Explorer и посмотреть, какие там присутствуют параметры.

Внимание!!! - имя драйвера у тебя может отличаться. Чтобы увидеть все доступные драйверы, нужно войти в *BDE Administrator*. Здесь, на закладе *Configuration* на дереве выбираешь *Drivers*, и здесь в разделах *Native* и ODBC есть все имена драйверов.

 *На компакт диске, в директории* ***\Примеры\Глава 16\Alias*** *ты можешь увидеть пример этой программы.*

16.5 Работа с XML таблицами.

Когда мы работаем с базами данных Access, то используем библиотеку DAO. Для доступа к таблицам Paradox или DBF, нужна библиотека BDE. Для XML таблиц нужен всего один файл, который достаточно зарегистрировать в системе и больше никакой головной боле. Из-за такой простоты установки, мы не смоем получить мощь других баз данных, зато получаем простоту, скорость и универсальность XML.

Итак, для того, чтобы ты мог работать с XML тебе понадобиться файл midas.dll. Чаще всего он находиться в системной директории windows. Для Windows 95/98/ME это *windows\system*, а для NT/2000/XP это WinNT/system32. Если у тебя этого файла нет, то можешь взять его с диска этой книги, в директории *dll/midas*. Там же находиться файл regsvr32.exe, который может произвести регистрацию этого dll файла. Для регистрации нужно выполнить команду regsvr32.exe с параметром midas.dll. Например, помести эти файлы в систему *c:\windows\system* (regsvr32.exe уже может находиться там), и нажми кнопку «Пуск» и выбери «Выполнить». Здесь напиши следующее *c:\windows\system\regsvr32.exe c:\windows\system\midas.dll*. Затем нажми ОК и ты должен увидеть сообщение о успешной регистрации.

Создавать XML таблицу мы будем прямо в Delphi. Для этого создай новый проект и сразу же добавь к нему модуль данных. Теперь брось в модуль данных два компонента: *DataSource* и *ClientDataSet1*. Сразу же укажи у компонента *DataSource* в свойстве *DataSet* компонент *ClientDataSet*, чтобы связать их.

Теперь выделяй компонент *ClientDataSet*. Дважды щёлкни по свойству *FieldDefs* в объектном инспекторе чтобы открыть окно определения полей (рисунок 16.5.1). В этом окне можно создавать объявления новых полей. Для этого щёлкни правой кнопкой мышки и выбери пункт *Add*, чтобы создать новое поле. Выдели в окне строку нового поля и посмотри в объектный инспектор. Тут у нас есть три интересных свойства:

DataType – тип поля.

Name – имя.

Size – размер.

Первое поле у нас будет ключевым, поэтому выбирай тип поля *ftAutoInc* и имя *Key1*. Остальное не изменяем. Теперь создай ещё одно поле типа *ftString*, с именем *FIO* и размер поля оставим по умолчанию 20. Для примера этого хватит, хотя ты можешь создать ещё несколько полей.

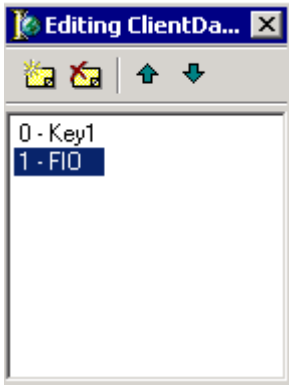


Рис 16.5.2. Обновлённая форма программы

Теперь щёлкни правой кнопкой по компоненту *ClientDataSet* и в появившемся меню выбери пункт *Create DataSet*. Теперь ещё раз щёлкни правой кнопкой по этому же компоненту и перед тобой откроется меню, в котором уже намного больше пунктов. Выбери *Save to MyBase Xml Table* и перед тобой откроется стандартное окно сохранения файла. Введи имя *exapl* и нажми «*Сохранить*».

Всё таблица готова. Теперь можно с ней работать абсолютно так же, как и с ADO или BDE таблицами. Дважды щёлкни по компоненту *ClientDataSet* чтобы открыть редактор полей. Здесь добавь все поля и дай им нормальные, русские заголовки. Кстати, ключевое поле можно вообще спрятать.

Теперь брось на главную форму программы сетку *DBGrid* и свяжи её с нашей таблицей. Программу можно запускать и попробовать с ней поработать. Здесь я не буду

писать полноценного примера и ограничусь только этим, потому что свойства и методы компонента *ClientDataSet* схожи с компонентами *TTable* и *TADOTable*.

Здесь я хочу сделать только одно замечание. Таблицы XML по умолчанию растут достаточно быстро. Это связано с тем, что в них сохраняется журнал изменений. С одной стороны постоянный рост файлов очень быстро есть пространство на диске, а с другой, благодаря этому журналу ты можешь в любой момент отменить последние действия. Для этого нужно вызвать метод *UndoLastChange* компонента *ClientDataSet*. У этого метода есть один параметр – булево значение. Если он равен *true*, то текущий курсор перебежит на строку, изменения которой были отменены, иначе курсор останется на месте.

Чтобы очистить журнал изменений вызови метод *MergeChangeLog*, а если хочешь, чтобы журнал вообще не вёлся, то присвой свойству *LogChanges* значение *false*.

Вот и всё, что я хотел сказать про XML таблицы. В остальном работа с ними ничем не отличается от работы с другими базами данных.

 *На компакт диске, в директории* ***\Примеры\Глава 16\XML*** *ты можешь увидеть пример этой программы.*

Глава 17. Потоки

Операционная система Windows является многопоточной. Это значит, что она может выполнять несколько задач одновременно. Почему я говорю именно задач, а не программ? Да потому что одна программа может состоять из нескольких независимых блоков кода, которые тоже могут выполняться одновременно. Каждый такой блок называется потоком.

Когда ты запускаешь новое приложение, то для него автоматически создаётся главный поток, в котором и будет выполняться код программы. Но это не значит, что ты ограничен этим потоком. В любой момент ты можешь создать дополнительные потоки, которые будут выполняться параллельно с главным.

Таким образом можно добиться многозадачности внутри самой программы.

17.1 Теория потоков.

Я говорю о потоках и ещё ни слова не сказал о том, зачем же нужно разделять программу на несколько потоков. Я часто в этой книге привожу примеры на основе таких программ, как Word и Excel и сейчас снова пример основанный на работе этих программ. Когда ты запускаешь Word и набираешь текст, то встроенный модуль проверки орфографии автоматически следит за тем, что ты пишешь и подправляет орфографические ошибки. Теперь представь логику проверки. После нажатия кнопки, нужно отобразить на экране нужную букву, затем проверить ближайшие слова на изменения и проверить правильность их написания. После проверки слов, проверять всё предложение на наличие пропущенных запятых или других знаков.

На словах алгоритм описывается достаточно просто. Но попробуй представить себе тот большой труд, который надо проделать после каждого нажатия кнопки. Если бы алгоритм проверки орфографии действительно действовал бы так, то буквы появлялись бы на экране не чаще 1 в пару секунд.

К счастью, проверка орфографии работает отдельным процессом. Ты спокойно набираешь текст, а проверка идёт в отдельном потоке не мешая тебе работать с текстом. При этом практически незаметны задержки, и нет никаких неудобств.

Когда ты пишешь новую программу, то не надо пытаться засунуть все функции в отдельные потоки. Каждый поток накладывает на программу дополнительную сложность и неустойчивость, да и отлаживать потоки намного сложнее.

Какой код нужно помещать в отдельный поток? Вот некоторые пример:

1. Если какие-то функции должны выполняться параллельно основному процессу, то тут деваться некуда и нужно обязательно помещать такие вещи в поток.

2. Если какие-то расчёты идут достаточно долго, то многие считают, что их тоже нужно помещать в поток. Просто когда идут такие расчёты программа блокируется и невозможно нажать кнопку «*Отмена*» или что-нибудь подобное. Это неправильное утверждение. Поток тут абсолютно необязателен, потому что можно обойтись и без него. Достаточно внутри расчётов поставить вызов *Application.ProcessMessages* и в этом месте выполнение расчётов будет прерываться на некоторое время и программа будет обслуживать другие сообщения, пришедшие от пользователя. Таким образом получиться простой эффект многозадачности без использования потока.

3. Код критичен к времени выполнения. Допустим, что твоя программа должна принимать какие-то данные по COM порту. Как только на порт пришли какие-то данные, они должны быть моментально обработаны с минимальной задержкой. Вот такие вещи желательно выносить в отдельный поток, потому что если в момент поступления данных программа занята большими расчётами, то данные могут оказаться необработанными.

Истинную многозадачность можно получить только на многопроцессорных системах, где каждый процессор выполняет свою задачу. В домашних компьютерах в основном ставиться только один процессор. Чтобы создать многозадачность на таких процессорах используют псевдомногозадачность. В этом виде один процессор выполняет сразу несколько задач благодаря быстрым переключениям между ними. Например, процессор может выполнять сразу десять задач, при этом каждой из них давать по 10 миллисекунд своего рабочего времени. В этом случае процессор будет через определённые промежутки времени переключаться между задачами и у пользователя будет создаваться впечатление, что они выполняются параллельно. Но это общий вид псевдомногозадачности, реально она реализована по другому.

В 32-х разрядных версиях Windows используется вытесняющая многозадачность (до этого была согласованная). В такой среде ОС разделяет процессорное время между разными приложениями и потоками на основе вытеснения. Разделение происходит в основном благодаря приоритету потока. У каждого потока есть приоритет, по которому определяется его важность. Чем выше приоритет, тем больше процессорного времени

выделяется этому потоку. Потоки с одинаковым приоритетом будут получать одинаковое количество процессорного времени.

У дополнительных потоков приоритет выставляется такой же как и у главного потока программы, но ты его можешь увеличить или уменьшить. Чем выше приоритет потока, тем больше на него отводится процессорного времени.

Снова допустим, что твоя программа должна принимать какие-то данные по COM порту и сразу же их обрабатывать. Для этого создаём новый потоки в нём реализуем код получения и обработки данных. Теперь достаточно поднять приоритет потока, чтобы на него при необходимости выделялось больше процессорного времени и задача решена. Теперь, как только поступают на COM порт новые данные, поток сразу же обработает их, потому что с более высоким приоритетом он получит больше процессорного времени.

В этой книге потоки будут использоваться достаточно часто в главе о программировании звука. Там мы будем создавать отдельный поток, в котором будет выполняться воспроизведение или запись звука, при этом основная программа сможет работать без каких-либо ограничений.

17.2 Простейший поток

Давай попробуем написать простейший поток и в процессе познакомимся с его возможностями и как всё реализовано. На практике этот материал усваивается лучше, поэтому не будем больше тратить время на лишние разговоры и посмотрим на потоки в действии.

Создай новый проект. Поставь на форму компонент *TRichEdit* из палитры Win32 и один компонент *TLabel*. Нам ещё понадобиться пару кнопок – одна для запуска потока, другая для его остановки. Посмотри на рисунок 17.2.1, где показана моя форма. У тебя должно получиться нечто похожее.

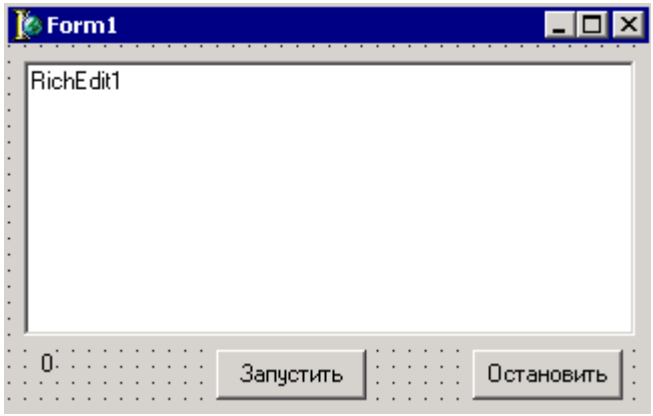


Рис 17.2.1. Главная форма нашей программы.

Теперь создадим модуль для потока. Для этого выбери пункт меню *File->New->Other* для открытия окна создания нового модуля (рисунок 17.2.2). найди в этом окне на закладке *New* пункт *Thread Object*. Выдели его и нажми кнопку "ОК". Появляется окошко, как на рисунке 17.2.3. В этом окне нужно указать имя создаваемого потока. Я назвал свой поток *TCountObj*. Нажимай «ОК» и Delphi создаст модуль-заготовку для нашего будущего потока.

Сохрани весь проект. Главную форму под именем Main, а поток под именем MyThread.

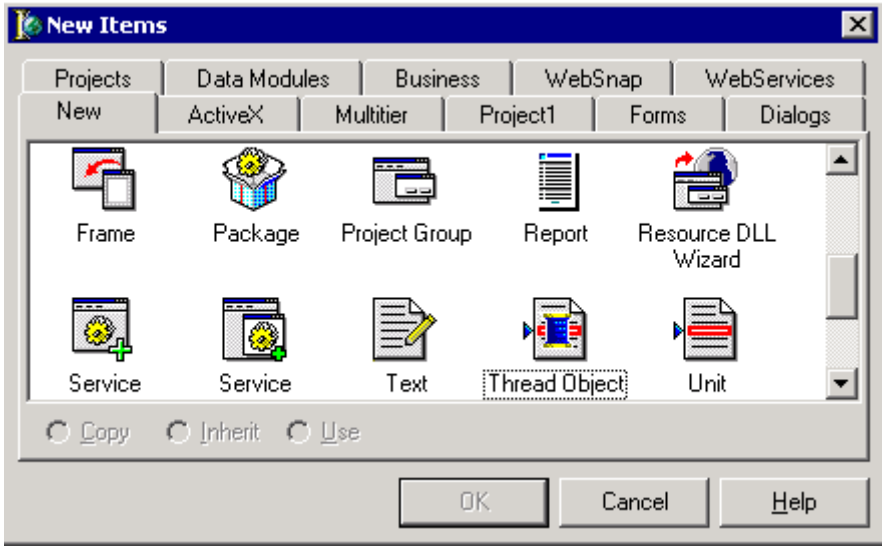


Рис 17.2.2. Создание модуля потока.

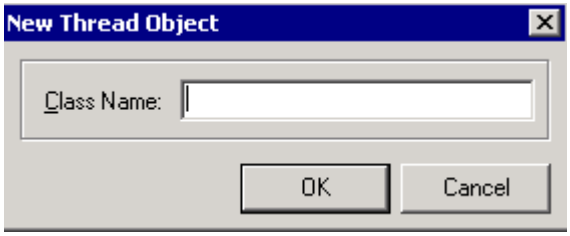


Рис 17.2.3. Задание имени потока.

Теперь посмотрим на код созданного для потока модуля:

```
unit MyThread;

interface

uses
  Classes;

type
  TCountObj = class(TThread)
  private
    { Private declarations }
  protected
    procedure Execute; override;
  end;

implementation

{ Important: Methods and properties of objects in VCL can only be used in a
  method called using Synchronize, for example,
}

     Synchronize(UpdateCaption);

  and UpdateCaption could look like,

    procedure TCountObj.UpdateCaption;
    begin
```

```
    Form1.Caption := 'Updated in a thread';
   end; }

{ TCountObj }

procedure TCountObj.Execute;
begin
  { Place thread code here }
end;

end.
```

Это новый поток. У объекта есть только одна процедура *Execute*. В любых потоках эта процедура обязана быть переопределена, ив ней должен быть написан собственный код. Это связано с тем, что в объекте *TThread*, эта процедура объявлена как абстрактная (*abstract*) – пустая. Это значит, что процедуре дали имя, выделили место, но её код должен быть написан объектами потомками, т.е. нами.

Метод *Execute* – это и есть заготовка для кода потока. То, что мы напишем здесь будет выполняться параллельно основной задаче. Давай напишем здесь следующий код:

```
procedure TCountObj.Execute;
begin
 index:=1;
 //Запускаем бесконечный счётчик
 while index>0 do
  begin
   Synchronize(UpdateLabel);
   Inc(index);
   if index>100000 then
    index:=0;

   //Если поток остановлен, то выйти.
   if terminated then exit;
  end;
end;
```

Переменную *index* я объявил как *integer* в разделе *private* объекта потока. Там же я объявил процедуру *UpdateLabel*. Эта процедура выглядит так:

```
procedure TCountObj.UpdateLabel;
begin
 Form1.Label1.Caption:=IntToStr(Index);
end;
```

И последнее, что я сделал - подключил главную форму в раздел **uses**, потому что я обращаюсь к ней в коде выше (*Form1.Label1.Caption*) для обновления текста компонента *Label1*.

В методе *Execute* у меня запускается цикл *while*, который будет выполняться, пока переменная *index* больше нуля. Внутри цикла я вызываю метод *Synchronize* (о нём чуть позже) и увеличиваю переменную *index*. Если эта переменная становиться больше 100000,

то в *index* присваивается 0 и расчёт начинается с начала. Таким образом цикл будет бесконечно выполнять увеличение переменной *index*от 0 до 100000 и опять сначала.

Самой последней идёт проверка, если свойство *terminated* равно *true*, то выйти из процедуры. Когда мы выйдём, то работа потока закончиться, потому что закончиться код процедуры *Execute*. Свойство *terminated* станет равной *true* тогда, когда будет вызван метод *Terminate* нашего потока.

Теперь о магической функции *Synchronize*. В качестве параметра ей передаётся процедура *UpdateLabel*, которая производит вывод в главную форму. Для чего нужно вставлять процедуру вывода на экран в *Synchronize*? Библиотека VCL имеет один недостаток - она не защищена от потоков. Все пользовательские компоненты разрабатывались так, что к ним может получить доступ только один поток. Если главная форма и поток попробуют одновременно вывести что-нибудь в одну и ту же область экрана или компонент, то программа рухнет как башни близнецы. Поэтому весь вывод на форму нужно выделять в отдельную процедуру и вызывать эту процедуру с помощью *Synchronize*.

Если процедура вызвана в методе *Synchronize*, то выполнение основной программы и потока морозиться и к компонентам окна получает доступ только объект, вызвавший метод *Synchronize*. Этот процесс незаметен для пользователя.

Так что если тебе нужно вывести какие-то данные из потока на экран главного окна, то делай это в отдельной процедуре и вызывай её с помощью метода *Synchronize*.

Всё, наш поток готов. Возвращаемся к главной форме. В раздел **uses** (самый первый, который идёт после **interface**) я добавил модуль потока *MyThread*. Почему именно в этот раздел, а не тот, что расположен ниже? Это связано с тем, что в разделе **private** мне нужно объявить переменную имеющую тип нашего объекта. Если добавить имя модуля во второй раздел **uses**, то он находиться ниже той части кода, где нам нужно написать объявление. Именно поэтому добавлять модуль *MyThread* нужно в первый раздел **uses**.

В разделе **private** я объявил переменную *co* типа *TCountObj* (объект моего потока).

По нажатию кнопки "Запустить" я написал такой код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 co:=TCountObj.Create(true);
 co.Resume;
 co.Priority:=tpLower;
end;
```

В первой строке я создаю поток *co*. В качестве параметра может быть *true* или *false*. Если *false*, то поток сразу начинает выполнение, иначе поток создаётся, но не запускается. Если поток создан не запущенным, то для запуска нужно использовать метод *Resume*, что я делаю во второй строке.

В третьей строке я устанавливаю приоритет потока поменьше, чтобы он не мешал работе основному потоку и выполнялся в фоне. Если установить приоритет повыше, то основной поток начнёт притормаживать, потому что у них будут одинаковые приоритеты.

По нажатию кнопки "*Остановить*" я написал:

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

Здесь я останавливаю выполнение потока с помощью вызова метода *Terminate* объекта потока. После вызова этого метода свойство *terminated* станет равной *true* и выполнение процедуры *Execute* закончиться.

Попробуй запустить эту программу, запустить поток (нажатием кнопки "*Запустить*") и понабирать текст в *RichEdit*. Текст будет набираться без проблем, и в это время в компоненте *TLabel* будет работать счётчик. Если бы ты запустил счётчик без отдельного потока, то ты бы не смог набирать текст в *RichEdit*, потому что все ресурсы программы (основного потока) уходили бы на работу счётчика.

 *На компакт диске, в директории* ***\Примеры\Глава 17\Thread1*** *ты можешь увидеть пример этой программы.*

17.3 Дополнительные возможности потоков.

Теперь я хочу дать краткий обзор ещё некоторых свойств и методов объекта потока, и сделать несколько замечаний по работе с потоком. В принципе, у объекта не так уж и много свойств и методов, и большую часть возможностей мы уже рассмотрели, но всё же я обязан сделать ещё несколько замечаний.

Suspend - приостанавливает поток. Для вызова просто напиши со.*Suspend*. Чтобы возобновить работу с этой же точки нужно вызвать *Resume*.

Priority- устанавливает приоритет потока. Например *Priority:=tpIdle*;

tpIdle - поток будет работать только когда процессор бездельничает.

tpLowest - самый слабый приоритет

tpLower - слабый приоритет

tpNormal - нормальный

tpHigher - высокий

tpHighest - самый высокий

tpTimeCritical - критичный (не советую использовать, потому что может грохнуть систему).

Suspended - если этот параметр *true*, то поток находится в паузе.

Terminated - если *true*, то поток должен быть остановлен, иначе поток должен продолжать работу.

Terminate – остановить выполнение потока.

FreeOnTerminate – если это свойство равно *true*, то по завершении выполнения процедуры *Execute* поток самоуничтожиться. Советую использовать этот параметр, чтобы быть уверенным в том, что поток корректно удалился из памяти.

Давай посмотрим, как будет выглядеть наш метод *Execute* из предыдущего примера с использованием свойства *FreeOnTerminate*:

```
procedure TCountObj.Execute;
begin
 FreeOnTerminate:=true;
 index:=1;
 //Запускаем бесконечный счётчик
 while index>0 do
  begin
   Synchronize(UpdateLabel);
   Inc(index);
   if index>100000 then
```

```
    index:=0;

   //Если поток остановлен, то выйти.
    if terminated then exit;
  end;
end;
```

В принципе, этой информации тебе будет достаточно для написания собственных потоков. У объекта *TThread* есть ещё несколько свойств и методов, но они не так важны и лично я ими никогда ещё не пользовался (не было такой задачи, где бы можно было их применить). Поэтому я не буду тратить место в книге на описания оставшихся возможностей, а лучше дам несколько советов.

Объекты потоков создаются как полноценные объекты. В основной программе мы создаём в памяти отдельный экземпляр потока и потом работаем с ним. Ты можешь создавать по несколько экземпляров одного потока и они будут работать одновременно абсолютно не мешая друг другу. Представим пример программы, копирующей файлы. Ты можешь создать поток, который будет копировать файлы из одного места в другое. В основной программе можно создать два экземпляра таких потоков и каждому из них задать копирования разных файлов в разные места. Оба потока будут копировать свои файлы абсолютно не мешая друг другу.

17.4 Подробней о синхронизации.

В предыдущем примере я использовал процедуру *UpdateLabel*, в которой на главную форму выводиться значение переменной *index*. Если бы мы программировали главное окно, то вполне логичным было бы создать переменную *index* локальной для процедуры *Execute*, а её значение передавать в *UpdateLabel* в качестве параметра. В потоках с этим проблема. Чтобы передать какие-то значения в процедуру, которая должна вызываться методом *Synchronize* нужно пользоваться переменными объекта. Даже не советую пробовать передавать параметры в процедуры которые вызываются методом *Synchronize*.

Но использование синхронизации – не единственный способ обновления параметров окна. Мы можем использовать для этого событийную модель Windows. Каждый раз, когда надо обновить содержимое текста мы можем посылать окну сообщение *SendMessage* с указанием значения, которое надо установить. Главное окно будет получать это сообщение и компонент сам изменит заголовок. В этом случае мы не обращаемся к главному окну из потока, а только отправляем сообщение, поэтому никаких проблем не будет.

Итак, функция *SendMessage* имеет следующие параметры:

1. Указатель на окно (компонент) которому нужно послать сообщение.
2. Тип сообщения.
3. Первый параметр.
4. Второй параметр.

Судя по функции, нам нужен компонент, у которого есть свойство *Handle*. В предыдущем примере у нас был *TLabel*, у которого нет такого свойстваЮ значит он нам не подходит. Замени этот компонент на *TEdit*. Теперь перейдём в поток. Тут в разделе **uses** нужно добавить два модуля: *windows* (здесь объявлена сама функция) и *messages* (здесь находятся все типы сообщений Windows).

Теперь удаляй из потока процедуру *UpdateLabel*, больше она не нужна, потому что мы не будем использовать метод *Synchronize*. Ну и наконец, подкорректируем наш метод *Execute*:

```
procedure TCountObj.Execute;
begin
 index:=1;
 while index>0 do
  begin
   SendMessage(Form1.Edit1.Handle, WM_SETTEXT, 0,
        Integer(PChar(IntToStr(index))));
   Inc(index);
   if index>100000 then
    index:=0;
   if terminated then exit;
  end;
end;
```

Как видишь, теперь у нас вместо метода *Synchronize* генерируется событие на обновления компонента *TEdit*. В качестве второго параметра я указываю тип сообщения *WM_SETTEXT* – обновить информацию. Третий параметр равен нулю. В последнем параметре нужно указать значение, которое нужно установить. Вот тут есть небольшая сложность. У нас значение представлено в виде целого числа, но нужно превратить его в *PChar*. Для этого я сначала конвертирую переменную *index* в строку (*IntToStr*), потом привожу его к типу *PChar* и тут же указываю размер *Integer*. Сложно? Зато не надо ничего синхронизировать.

 *На компакт диске, в директории* ***\Примеры\Глава 17\Thread2*** *ты можешь увидеть пример этой программы.*

Глава 18. Динамически компонуемые библиотеки.

Ты уже наверно много раз слышал заветное выражение «динамически компонуемые библиотеки. Пора познакомиться с ними поближе. В этой главе я постараюсь тебе дать всю необходимую информацию по ним, мы напишем несколько примеров, и ты всё прекрасно увидишь на практике.

В отличии от остальных глав, в этом вступительном слове я больше ничего говорить не буду, потому что если затронуть тему описания предназначения DLL файлов, то тема растянется в долгий разговор, поэтому я посвящу этому первую же часть этой главы.

18.1. Что такое DLL?

Программисты всех стран уже более 30 лет борются с проблемой многоразового использования однажды написанного кода. Так уж повелось, что 30-50% кода в простых офисных приложениях схожи между собой или решают одни и те же задачи. Ни один программист не захочет каждый раз снова писать один и тот же код. Как хорошо, когда можно использовать один раз написанный код многократно ....

Я сам не люблю в каждой новой программе писать одно и то же. Как хорошо, когда написал какой-то универсальный код, а потом только используешь его.

Решение №1.

Самым первым решением этой проблемы стал модульное программирование. Ты пишешь какой-то кусок кода, оформляешь его в виде модуля, а потом просто используешь его в своих программах. Все прекрасно и удобно, а главное, что все довольны. Теперь не надо каждый раз выдумывать велосипед, просто добавил к своей программе определенный модуль и без проблем используй код, когда-то написанный тобой или кем-то другим.

Казалось, что это было самое простое и самое эффективное решение. Но все было прекрасно, пока не появилась многозадачность. Вот тут программисты и простые пользователи заметили, что еще не все так эффективно и полно места, куда можно приложить свои руки для оптимизации выбранного решения.

Проблема №1.

Давай представим ситуацию, когда один добрый человек написал прекрасный модуль размером в 1 мегабайт. Другой добрый человек решил воспользоваться его возможностями и подключил к своей программе. Модуль и программа слились в одно целое. Вроде все нормально, но я же сказал, что программа и модуль слились в одно целое. Это значит, что размер результата увеличился на размер модуля, т.е. на 1 мегабайт. Не фига себе пельмень!!!

А теперь представь, что другой чел написал другую утилиту с использованием этого модуля…. Его программа тоже увеличилась на 1 мегабайт. Получается, что на винте пользователя хранится две программы, в которых по 1 мегабайту кода одинаковых. И кому это нужно?

Ну, конечно же, на счет модуля в 1 мегабайт я немного преувеличил. В те времена даже 100 кило модуль тяжело было найти. Но надо учитывать, что и винты тогда были не бесконечные. Тогда крутым винтом считался диск в 20 мегабайт. Это тебе не нынешние десятки гигабайт на одной пластине. Я сам застал такие машины только на первом курсе института, а это было почти 10 лет назад.

Проблема №2.

Пока существовали только однозадачные операционные системы, проблема с излишней растратой дискового пространства была единственной. Но как только задумались о многозадачности и в мыслях Билла Гейтса появились идеи создать Windows, так сразу возникла другая проблема…. Представь себе ситуацию, когда ты запускаешь обе

этих программы одновременно. При старте любой код грузится в оперативную память и только потом выполняется. Так что получается, что обе программы загрузят в память один и тот же код. Вот это уже абсолютно никому не нужно.

Это только в последнее время память подешевела в несколько раз, и теперь лишние сто кило погоды не сделают. А раньше она стоила достаточно дорого, и люди боролись за каждый байтик потом и кровью. Но если ты думаешь, что если поставить в свой компьютер 500 мегабайт оперативной памяти и проблема уйдёт сама собой, то ты крупно ошибаешься.

Хотя память и дешевая, программы от этого меньше не станут. Если посмотреть на запросы той же Windows 2000, то сразу понятно, что эти 500 мегабайт это только капля в море. Самая простая ОС Windows 2000 Professional отнимет от них около 128 мегабайт. Это что же там такое натворили, что Windows 2000 Server просит для нормальной работы минимум 256 мегабайт? А если учесть, что ещё недавно чипсеты не поддерживали памяти более 512 мегабайт (это сейчас можно от 2 до 3 гигабайт вставить), то о нормальной одновременной работе Windows 2000 Server + 3D Studio Max + MPEG4 можно забыть. Они всю память отберут, как термиты за пять сек.

Решение №2.

И вот тут было найдено вполне солидное решение: не стыковать модули с основной программой, а сохранять их в отдельный файл и пусть любая программа загружает его по мере надобности. Сказали, сделали. Так появились библиотеки DLL, что означает *Dynamic Link Library (DLL)*. Это библиотеки, которые подключаются к программе динамически. В них можно хранить исполняемый код в виде процедур или функций, ресурсы программы, графику или даже видео ролики.

Вот так. Теперь программа не увеличивалась на размер модуля при компиляции, а просто загружала код из DLL файла в память и использовала его. Если одна программа уже загрузила DLL, то следующая не будет уже делать этого. Она воспользуется уже загруженной версией. Таким образом, экономится не только диск, но и оперативная память, которой, как и денег, много не бывает.

Сейчас уже DLL - это не просто динамически подгружаемая библиотека. Ты наверно уже не раз слышал про компоненты *ActiveX*. Они так же могут быть выполнены в виде осх или dll файлов. Да оно и понятно, *ActiveX* используются сейчас достаточно много и занимают места в несколько раз больше чем самая большая DLL библиотека. Так что единственный и нормальный выход экономить место винта и памяти это засунуть ActiveX в динамически подгружаемую библиотеку. Хотя это уже не та DLL, но всё же работает по тем же принципам.

У динамических библиотек есть единственный недостаток - на ее загрузку тратится лишнее время. Если бы код, находящийся в DLL был бы скомпонован с программой, то он грузился бы намного быстрее. Зато если библиотека уже загружена другой программой, то она появляется намного быстрей. Не веришь? Отложи сейчас книгу и возьми в руки секундомер. Теперь запусти Word или Excel. Засеки сколько времени будет проходить загрузка. Теперь закрой эту программу и запусти ее снова. Она появится на экране практически моментально. Это потому что после выхода из программы, DLL файл не выгружается из памяти. Это происходит только тогда, когда операционной системе не хватает памяти и ни одна из программ не использует в данный момент эту библиотеку.

А теперь представь себе, что такое Word!!! Представил? Это и текстовый редактор, и проверка орфографии, и построитель диаграмм, редактор формул и куча еще всякой всячины. Представь себе, что было бы, если все это засунуть в один файл? Нет, ты это не можешь представить. Это был бы один запускной файл размером в 30-50 мегабайт.

А теперь вспомни, что я тебе сегодня говорил: перед запуском, программа загружается в память. Представляешь теперь, сколько бы грузился Word? А сколько

памяти он съёл бы? А тебе ведь и половина его возможностей абсолютно не нужна. И зачем же их грузить в память?

А при использовании динамических библиотек в запускном файле находится только самое основное, а дополнительные возможности подгружаются по мере надобности из DLL-файлов. Например, когда стартует Word, то загружается только модуль текстового редактора. Когда ты выбрал редактор формул или объект WordArt, то Word подгружает из dll файла код выбранного объекта и выполняет его. Таким образом, суммарная скорость загрузки уменьшается, причем очень даже значительно.

Ещё одно большое преимущество динамических библиотек – при их использовании код программы разбивается на несколько файлов (зависит от количества DLL файлов). Допустим, что в одной из функций находящейся в DLL оказался код с ошибкой. В этом случае не надо обновлять всю программу, а достаточно передать всем пользователям только этот DLL файл, и программа получит необходимые обновления.

У динамических библиотек сплошные преимущества и только два недостатка:

1. Код из DLL файла выполняется в том же участке памяти, что и основная программа. Поэтому программа и DLL используют один и тот же стек данных, что иногда накладывает свои ограничения. Например, DLL не может хранить глобальных переменных. Воспринимай динамические библиотеки просто как набор процедур и функций, которые могут хранить только локальные переменные.

2. Изначально динамические библиотеки были процедурными. Хотя сейчас умельцы умудряются использовать их для хранения объектов, но это очень неудобно. Но, несмотря на это, ActiveX (изначально объектные) могут храниться в файлах с расширением dll.

Но всё же динамические библиотеки получили широкое распространение и программисты используют их на каждом углу, когда надо и когда не надо. Никогда нельзя быть уверенным, что какой-то код уже больше никогда не понадобится. Всегда нужно рассчитывать на будущее.

Я надеюсь, что я тебя убедил в великих возможностях динамических библиотек. Это действительно так. Конечно же, ActiveX более продвинуты, но они требуют неудобной регистрации в системе (в реестре) и намного сложнее в программировании, а библиотеки пишутся достаточно просто и их достаточно только скопировать на другой компьютер, чтобы программа смогла её найти.

Из чего же сделан Windows?

Все наверно помнят такую песенку: "Из чего же, из чего же, из чего же, сделаны эти мальчишки?". Глупейшая песня, и я со слезами на глазах вспоминаю, как я в лагере (я имею ввиду пионерский, а не концлагерь) распевал ее вместе с остальными пионерами. Ох, и веселые были времена. Жаль, что сейчас так не развлечешься. О чем это я? Ах да... Я хотел рассказать тебе, из чего состоит Windows.

Большинство думает, что Windows - это все что находится в папке c:\Windows, а ее ядро - это win.com. В какой-то степени это так, но не совсем. Ядро ОС Windows - это простой DLL файл, а если быть конкретнее, то это Kernel32.dll. При старте Windows эта библиотека загружается в память в единственном экземпляре, и любая программа может обращаться к содержащемуся в ней коду и использовать его в своих целях. В этой библиотеке расположены API функции, предназначенные для распределения памяти и многое другое. Мы эти функции не вызываем напрямую, потому что Delphi прячет этот сложный процесс от нас, но иногда тебе может понадобиться воспользоваться ими. Так что помни, если ты выделяешь память, то в этот момент используется Kernel32.dll.

Точно так же, за вывод графики в Windows отвечает GDI32.DLL, которая так же загружается при старте в единственном экземпляре. Все функции для работы с графикой находятся в этой библиотеке. Есть и ещё одна библиотека, User32.dll, которая отвечает за

создание окон и обработку сообщений. Все эти три библиотеки составляют ядро ОС Windows.

В Windows очень много недостатков, но динамические библиотеки это достаточно гениальное решение многократно используемого кода.

Графические движки.

Любой игрок обязан знать про существование OpenGL. Что это такое? Какой-то пакет программ? Какой-то SDK для создания графики? Ничего подобного, это всего лишь две динамические библиотеки opengl.dll (opengl32.dll) и glu.dll (glu32.dll).

Что такое DirectX? Это графическая библиотека, которая состоит из DirectDraw, DirectInput, DirectMusic, DirectPlay и так далее. Все это не что иное, как простые динамически подгружаемые библиотеки. DirectDraw это Ddraw.dll, DirectInput это Dinput.dll, DirectMusic это Dmusic.dll и так далее. Хотя DirectX это не простые библиотеки – это библиотеки созданные на основе технологии COM (та же технология, что и ActiveX).

Любые игровые движки выполнены в виде динамически загружаемых библиотек, потому что их использование очень простое и удобное для любого программиста.

Давай подведём итог тому, что уже было сказано. Динамические библиотеки практически ничем не отличаются от EXE файлов. Это такой же скомпилированный код, только он не может запускаться самостоятельно, потому что в библиотеке нет точки входа (точки, с которой начинает своё выполнение любая программа) В dll файлах хранятся только процедуры и функции, которые можно вызывать из других программ.

Чаще всего динамические библиотеки имеют расширение dll, но можно установить и любое другое расширение или вообще убрать его. Библиотеки имеют свои разновидности, например, драйверы – это тоже динамические библиотеки и им принято давать расширение drv. Возможно так же использование расширения sys для системных файлов. Так что операционная система не накладывает ограничений на расширения динамических библиотек, главное, чтобы тебе было понятно и удобно работать.

Когда одно приложение загружает библиотеку, то она загружается в глобальную память, а потом только проецируется в адресное пространство программы. Это значит, что программа будет видеть функции библиотеки как родные, хотя они расположены совершенно в отдельном адресном пространстве.

Операционная система Windows гарантирует, что в любой момент будет загружена в память только одна версия dll файла. Если две программы обращаются к одной и той же библиотеки, то в памяти будет находиться только одна копия dll файла.

Говоря о библиотеках (dll) я всё время говорил о динамических библиотеках. Но существуют и статические варианты библиотек. Чем они отличаются? В принципе, библиотека одна и та же, поэтому термин статической DLL я считаю неправильным (хотя иногда встречаю в литературе). Но всё же я вынес это название в заголовок, чтобы показать тебе на ошибку.

Библиотеки dll всегда динамические и создаются они с целью динамической загрузки находящихся в них ресурсах. Но не смотря на это, многие компиляторы позволяют присоединять код dll статически. В этом случае при компиляции программы код или данные, находящиеся в dll, становятся неотъемлемой частью исполняемого файла. В этом случае программа и библиотека становятся как единое целое. Это очень удобно, когда библиотека небольшая или тебе необходимо, чтобы программа состояла только из одного запускного файла. Здесь динамическая загрузка не подходит, и надеяться на существование библиотеки на машине клиента нельзя. Такая ситуация может возникнуть, когда ты хочешь показать клиенту демонстрационный файл твоей программы и удобно иметь только один запускной файл, а не множество библиотек.

18.2. Простой пример создания DLL.

Так как DLL – это отдельный файл, то и создаётся он в Delphi как отдельный проект. Для создания новой динамической библиотеки нужно выбрать из меню *File* пункт *New* и затем *Other*. В окне создания нового проекта (рисунок 18.2.1) нужно выбрать на закладке *New* пункт *DLL Wizard*. Выбери этот пункт и нажми *OK*.

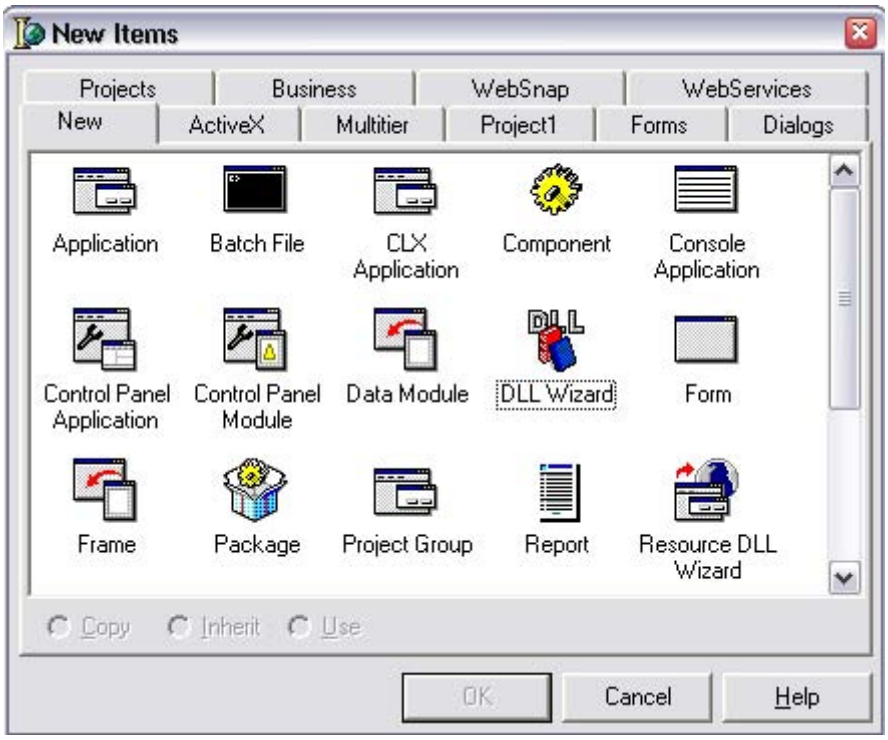


Рисунок 18.2.1 Окно создания нового проекта.

Несмотря на то, что мы выбрали *DLL Wizard* (слово Wizard говорит о том, что должен запуститься мастер), будет просто создан пустой проект с одним только модулем. Модуль будет содержать следующий текст (я убрал только комментарии):

```
library Project2;

uses
  SysUtils,
  Classes;

{$R *.res}

begin
end.
```

Если открыть менеджер проектов (Меню *View->Project Manager*), то в окне вообще не будет видно ни одного модуля. Это потому что код, который мы видели выше относиться к самой библиотеки. Выбери из меню *File* пункт *Save All*, и тебе предложат сохранить только один проект и никаких модулей. Я сохранил проект под именем FirsDLLProject. Потом открыл файл проекта FirsDLLProject.dpr с помощью блокнота и увидел тот же самый код.

Теперь давай добавим в нашу библиотеку одну функцию с именем *Summ*. У этой функции будет два параметра в виде целых чисел, и возвращать она будет сумму этих чисел:

```
library FirsDLLProject;

uses
  SysUtils,
  Classes;

function Summ(X,Y:Integer):Integer; StdCall;
begin
 Result:=X+Y;
end;

exports Summ;

{$R *.res}

begin
end.
```

Обрати внимание, что функция у нас объявлена не так как всегда. В конце строки объявления, после типа возвращаемого значения стоит ключевое слово **StdCall**. Оно говорит о том, что для вызова процедуры нужно использовать стандартный тип вызова.

Я тебе уже говорил, что все параметры, передаваемые в процедуры и в функции, передаются через стек. Если не указать ключевое слово **StdCall**, то параметры будут передаваться способом, заложенным фирмой *Borland*. Этот способ работает быстрее, но он не совместим со стандартными правилами. Если ты уверен, что к процедуре будут обращаться только программы скомпилированные компиляторами фирмы Borland, то можешь не ставить это ключевое слово. Но если библиотека будет выложена на всеобщее использование или к ней будут обращаться программы сторонних разработчиков, то желательно ставить **StdCall**, иначе у программистов на языках Visual C++ или других языках будут проблемы. Я сделал для себя правилом всегда ставить **StdCall**, потому что способ Borland даёт незначительный выигрыш.

В остальном, функция ничем не отличается от тех, что мы уже писали.

После описания функции идёт новое ключевое слово **exports**. После этого ключевого слова должно идти описания процедур, которые должны быть доступны внешним программам. Если нашу функцию *Summ* не описать в разделе **exports**, то мы её не сможем вызвать из внешней программы.

Теперь откомпилируй проект (нажми Ctrl+F9 или выбери из меню *Project* пункт *Compile FirsDLLProject*), чтобы создать нашу динамическую библиотеку. Можешь не пытаться запускать проект, потому что это библиотека, и она не может выполняться самостоятельно. Так что, единственное, что ты можешь увидеть – ошибку.

Теперь напишем программу, которая будет использовать написанную функцию из динамической библиотеки. Создай новый проект простого приложения (*File->New->Application*). На форму брось только одну кнопку и по её нажатия напиши следующий код:

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

```
 r:Integer;
begin
 r:=Summ(10,34);
 Application.MessageBox(PChar(IntToStr(r)), 'Результат функции Summ');
end;
```

В первой строчке я вызываю функцию *Summ* с двумя числовыми параметрами. Результат записывается в переменной *r*. Вторая строка всего лишь выводит окно с результатом.

Если ты попытаешься сейчас откомпилировать проект, то у тебя ничего не выйдет. Компилятор Delphi скажет, что он не знает такой функции *Summ*. Мы должны показать Delphi, что это за функция и где её искать.

Для начала покажем компилятору, что это за функция. Для этого в разделе type, после описания объекта *TForm1* (нашей главной формы) нужно написать следующую строку:

```
function Summ(X,Y:Integer):Integer;StdCall;
```

В принципе, это такое же объявление функции, которое описано в библиотеке, только здесь нет **begin** и **end** и самого кода процедуры. По этой строке Delphi узнаёт, что где-то существует такая функция *Summ*, у неё есть два параметра, и она должна вызываться стандартным вызовом.

Теперь нужно сказать компилятору, где же искать эту загадочную функцию. Для этого после слова **implementation** напиши следующий код:

```
function Summ; external 'FirsDLLProject.dll' name 'Summ';
```

Здесь написано, что есть такая функция *Summ*. После точки с запятой стоит ключевое слово **external**, которое говорит о том, что функция внешняя, не принадлежит программе. После этого слова указывается имя динамической библиотеки, где нужно искать функцию. Далее идёт ключевое слово **name**, которое означает, что функцию надо искать по имени. После этого ключевого слова указывается точное имя функции в библиотеки.

Вот теперь проект готов и его можно компилировать, запускать и проверять результат.

 *На компакт диске, в директории* ***\Примеры\Глава 18\FirstDLL*** *ты можешь увидеть пример этой программы.*

В нашем примере использовался вызов по имени функции. Когда программе нужно выполнить функцию *Summ*, то она просматривает все функции динамической библиотеки и ищет функцию с указанным именем. Это очень неэффективно и перед первым вызовом будет ощущаться большая задержка. Чтобы хоть немного ускорить процесс вызова таких функций можно использовать индексы. Каждой функции в библиотеке может быть назначен индекс, и при вызове можно указывать его.

Давай подкорректируем наш пример на индексы. Открой проект динамической библиотеки *FirsDLLProject.dpr*. Найди ключевое слово **export** и напиши там такой код:

```
exports Summ index 10;
```

После имени функции стоит ключевое слово **index** и числовой индекс функции. Я люблю нумеровать свои функции, начиная с10. Этой я дал десятый индекс (ты можешь попробовать другое число. Индексы и имена должны быть уникальными!!! Вот несколько примеров:

```
exports
  Func1 index 10 name 'Fun',
  Func2 Insert,
  Func3 index 11,
  Func4 index 11,//Ошибка, такой индекс уже существует
  Func5 name 'Don';
```

В объявлении последней процедуры я явно использовал ключевое имя **name**, чтобы указать экспортной функции новое имя. Теперь внутри библиотеки эта функция реализована как *Func5*, но внешние приложения должны обращаться к ней по имени *Don*.

Объявлять можно и так:

```
exports Func1 index 10 name 'Fun',
exports Func2 Insert,
exports Func3 index 11,
```

Перекомпилируй проект, чтобы изменения вошли в силу (нажми Ctrl+F9).

Теперь возвращаемся в проект, где мы используем функцию. В разделе **implementation** подправляем описание нашей функции:

```
function Summ; external 'FirsDLLProject.dll' index 10;
```

Теперь вместо ключевого слова **name** стоит слово **index** и тот же номер.

Запусти проект и убедись, что он работает корректно.

 *На компакт диске, в директории* ***\Примеры\Глава 18\IndexName*** *ты можешь увидеть пример этой программы.*

18.3. Замечания по использованию библиотек.

Когда я сказал, что DLL файлы нельзя запускать, то я и соврал, и нет. В принципе, библиотеки действительно нельзя запускать, но Delphi может сделать это. Открой нашу библиотеку и выбери из меню *Run* пункт *Parameters*. Перед тобой откроется окно, как на рисунке 18.3.1.

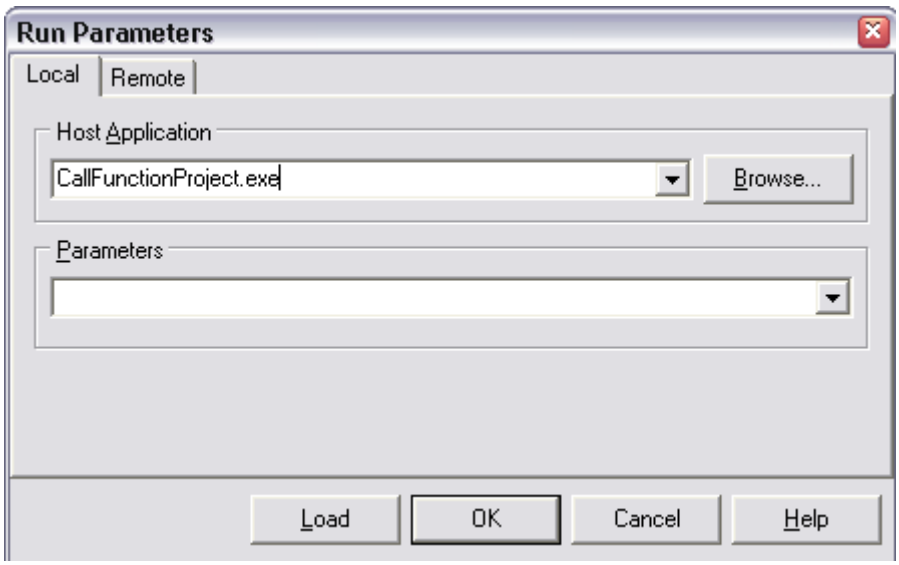


Рисунок 18.3.1 Окно параметров запуска программы.

В строке *Host Application* нужно указать имя приложения, которое умеет загружать библиотеку. Теперь попробуй запустить проект (клавиша F9). Запустить указанная программа.

Зачем нужен этот способ? Если ты попытался запустить программу, и она показала ошибку в коде, где вызывается функция из динамической библиотеки, то можно попытаться запустить библиотеку таким образом. Если снова произойдёт ошибка, то Delphi покажет строку с ошибкой.

Небного позже я покажу тебе, как можно отлаживать программы, выполнять их по шагам. Тогда ты сможешь узнать, что таким образом можно отлаживать и динамические библиотеки. А именно, они могут выполняться в пошаговом режиме (построчно) и ты будешь контролировать весь процесс выполнения.

Пока что я о библиотеках говорил достаточно много хорошего, но не сказал самого главного. Функции и процедуры из динамической библиотеки не могут прямо влиять на ход основной программы. Это значит, что мы не можем получить доступ к окнам основной программы, изменить какие-то переменные или ещё чего-нибудь.

Функции библиотеки – как бы изолированы от всего остального, хотя и выполняются в одном адресном пространстве с основной программой. Они могут использовать только переданные параметры, а результат работы возвращать в качестве результата работы функции.

Имена библиотек пиши полностью, вместе с расширением. Без расширения DLL может быть не найдена в Windows NT/2000/XP, хотя в Windows 98 всё будет работать нормально.

Обязательно соблюдай индексы и параметры процедуры, иначе могут возникнуть ошибки. Лучше лишний раз проверить, чем потом долго искать опечатку.

18.4. Хранения формы в динамических библиотеках.

Теперь я хочу показать, как можно хранить в динамических библиотеках целые окна. Очень удобно, когда редко используемые окна, находятся в динамической библиотеке. В этом случае основной файл очень сильно разгружается от лишнего кода.

Ещё одно преимущество такого кода – библиотека может использовать для вывода информации на экран своё окно. Как я уже сказал, из dll файла нельзя получить доступ к переменным и данным основной программы. Это значит, что из DLL файла нельзя ничего вывести в окна основной программы. Но библиотека может создать собственное окно и использовать для вывода необходимых данных именно его.

Итак, создавай новую DLL библиотеку и сохрани её под именем *ProjectDLL*. Теперь добавим к нашей библиотеки одну экспортную процедуру *ShowAbout*:

```
library ProjectDLL;

uses
  SysUtils, Classes;

{$R *.RES}

exports ShowAbout index 10;

begin
end.
```

Я добавил только одну строку *exports ShowAbout index 10;*. У нас будет только одна процедура *ShowAbout* с индексом 10. Эта процедура будет показывать окно «*О программе*».

Теперь щёлкаем File->New Form , чтобы создать новую форму. Нарисуй на ней что-нибудь, можно даже то, что сделал я (рисунок 18.4.1).

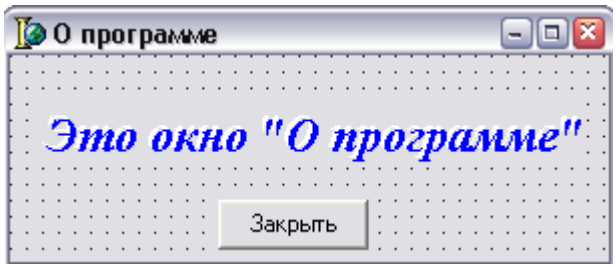


Рисунок 18.4.1. Окно «О программе».

Переходи в текст модуля. В разделе var, после объявления формы опиши процедуру ShowAbout:

```
var
  Form1: TForm1;
  procedure ShowAbout(Handle: THandle);export;stdcall;
```

Опять присутствует ключ **export** и добавлен ещё **stdcall**, указывающий на обязательность использования стандартного вызова процедуры.

Теперь напишем саму функцию после ключевого слова **implementation** и ключа *{$R *.DFM}*:

```
procedure ShowAbout(Handle: THandle);
```

```
begin
 //Установить указатель на приложение
 Application.Handle := Handle;
 //Создать форму
 Form1:= TForm1.Create(Application);
 //Отобразить
 Form1.ShowModal;
 //Очистить
 Form1.Free;
end;
```

Эта процедура получает в качестве параметра указатель на главное приложение. В первой строке я устанавливаю этот указатель в свойство *Handle* объекта *Application*. Этот объект хранит настройки всего приложения, и этим присваиванием мы связали оба приложения.

Во второй строке кода я создаю окно *TForm1.Create(Application)*, в результате чего мне будет возвращён указатель на это окно. Результат я сохраняю в переменной *Form1*. Эта переменная объявлена в разделе **var** проекта.

Следующей строкой я отображаю модально созданное нами окно. Как только оно закроется, будет выполнена последняя строка кода этой процедуры, а именно, окно будет уничтожено из памяти и процедура закончит своё выполнение.

В процедурах DLL библиотек будь более внимателен к высвобождению памяти. По моей практике могу сказать, что ошибки в библиотеках переносятся программами более критично, потому что тут основная программа практически бессильна.

Откомпилируй библиотеку (*Ctrl+F9*) и DLL-файл готов. Можно закрывать этот проект (*File->Close All*) и создавать новое приложение, из которого мы будем вызывать созданную в библиотеке процедуру (*File->New Application*).

В новом проекте переходим в текст формы и объявляем функцию ShowAbout:

```
unit Unit2;

interface

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

procedure ShowAbout(Handle: THandle)stdcall;

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  procedure ShowAbout;external 'ProjectDLL.dll' index 10;

implementation
```

Обрати внимание, что первое описание процедуры я написал не в разделе **type**, а до него:

```
procedure ShowAbout(Handle: THandle)stdcall;
```

Это не является ошибкой, и ты можешь выбрать любой из этих способов. Я чаще всего объявляю внешние процедуры до раздела **var**, чтобы их потом легче было найти.

Теперь ставим на форму кнопочку и пишем по её событию *OnClick* следующий код:

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

Запусти пример и убедись в том, что пример работает корректно. Как видишь окно не так уж трудно засунуть в библиотеку, и в нём могут быть свои события и свои процедуры и функции. К тому же окно может создавать дочерние окна по отношению к себе и той же динамической библиотеки.

*На компакт диске, в директории **\Примеры\Глава 18\Form** ты можешь увидеть пример этой программы.*

18.5. Немодальные окна в динамических библиотеках.

В предыдущем примере я поместил в библиотеку модальное окно. А что, если тебе понадобиться показать немодальное окно? Ведь мы показываем окно и по его закрытию должны освободить память. А как узнать, что окно закрыто? Некоторые ленятся и просто не освобождают память, выделенную под окно. Но это не правильно и просто глупо, потому что показать немодальное окно не намного сложнее.

Давай откроем предыдущий пример и подкорректируем его. Для начала нужно добавить одну экспортную процедуру *FreeAbout* с индексом 11. Теперь у нас будет экспортироваться две процедуры:

```
exports ShowAbout index 10;
exports FreeAbout index 11;
```

Теперь переходим в модуль *Unit1*, где у нас находиться форма динамической библиотеки. Процедуру *ShowAbout* я превращаю в функцию, которая будет возвращать значение типа LongInt. В качестве возвращаемого значения будет идентификатор окна, по которому мы потом будет его закрывать.

Ещё нужно добавить процедуру *FreeAbout* с одним параметром типа *LongInt*.

```
function ShowAbout(Handle: THandle):LongInt;export;stdcall;
procedure FreeAbout(FormRef: LongInt);export;stdcall;
```

Как я уже говорил, динамические библиотеки не могут хранить переменных. Именно поэтому после создания окна мы должны вернуть идентификатор основной программе, чтобы она хранила эту переменную. Когда нужно будет закрыть окно, мы передадим этот идентификатор библиотеке, и она освободит память, выделенную под окно.

Теперь посмотрим на реализацию функции *ShowAbout*:

```
function ShowAbout(Handle: THandle):LongInt;
begin
 Application.Handle := Handle;
 Form1:= TForm1.Create(Application);
 Form1.Show;
 Result:=LongInt(Form1);
end;
```

Здесь всё осталось также, за исключением последней строчки. Если раньше мы освобождали память, то сейчас возвращаем окно *Form1* приведённую к типу *Integer*. Если бы мы тут вызвали метод *Free*, то окно сразу же после появления закрылось бы.

Теперь посмотрим на процедуру *FreeAbout*:

```
procedure FreeAbout(FormRef: LongInt);
begin
 if FormRef>0 then
  TForm1(FormRef).Free;
end;
```

В этой процедуре мы сначала проверяем, если переменная *FormRef* (идентификатор окна) больше нуля, то окно можно уничтожать, иначе оно могло быть уже уничтожено. Во второй строке я вызываю метод *Free* нашего окна. Так как переменная *FormRef* – это числовая переменная и у неё нет методов, то мы должны перевести её обратно к объекту - *TForm1(FormRef)*.

Теперь подкорректируем проект, который использует DLL файл. Для начала подправь объявления процедур библиотеки. Перед разделом **type** напиши следующее:

```
function ShowAbout(Handle: THandle):LongInt;stdcall;
procedure FreeAbout(FormRef: LongInt);export;stdcall
```

В разделе **var** пишем следующее:

```
function ShowAbout;external 'ProjectDLL.dll' index 10;
procedure FreeAbout;external 'ProjectDLL.dll' index 11;
```

Всё это уже должно быть знакомо и не должно вызывать вопросов. Теперь в разделе **private** объекта главной формы добавляем переменную *f* типа *LongInt*.

Подготовка закончено. Осталось только вызвать эти процедуры. Добавь на форму ещё одну кнопку. По нажатию первой, мы будем показывать окно:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 f:=ShowAbout(Handle);
end;
```

Здесь я вызываю функцию показа окна и сохраняю результат в переменной. По нажатию второй кнопки вызываем процедуру освобождения памяти:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
 FreeAbout(f);
end;
```

Запусти пример и убедись, что всё работает корректно. Самое сложное здесь – определить, когда пользователь самостоятельно закрыл окно (например, кнопкой «Закрыть» в нашем окне «О программе»), чтобы мы вызвали процедуру *FreeAbout*. В качестве решения такой проблемы могу посоветовать следующее. При старте приложения присваивать переменной *f* значение 0. В этом случае, мы будем застрахованы от попадания в неё случайного числа. Перед созданием окна вызывать *FreeAbout*. В этом случае, сначала будет происходить проверка переменной f на ноль. Если переменная больше 0, то окно уже создавалось, но память не освободилась. Вот обновлённый код нажатия кнопки показа диалогового окна:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 if f>0 then
  FreeAbout(f);
 f:=ShowAbout(Handle);
end;
```

Здесь идёт проверка, если *f* больше нуля, то надо освободить память от старого окна, а потом пытаться создавать новое.

По событию *OnClose* для главной формы тоже не помешает вызвать процедуру освобождения памяти. Если программа закрывается, то окно из библиотеки уж точно уже не понадобится, значит, переменную *f* можно проверять на 0 и если там большее значение, то освобождать память.

На компакт диске находится пример, в котором уже реализовано всё сказанное и ты можешь увидеть этот код своими глазами и проверить его в действии.

 *На компакт диске, в директории* ***\Примеры\Глава 18\NoModal*** *ты можешь увидеть пример этой программы.*

18.6. Явная загрузка библиотек.

Предыдущие примеры хороши тем, что они просты, но у них есть недостаток – динамическая библиотека загружается автоматически при старте программы. В этом есть два недостатка. Во-первых, загрузка программы немного теряет в скорости (не сильно, но всё же), а функции из библиотеки могут вообще не понадобиться за всё время выполнения программы. Во-вторых, тебе может понадобиться поставлять программу в укороченном варианте без некоторых функций (без каких-либо dll файлов), но это не получиться, потому что программа на этапе загрузке будет выдавать ошибку о том, что библиотека не найдена.

От всего этого можно избавиться, если использовать явную загрузку библиотеки в определённый момент. Для этого надо немного попотеть.

Давай в предыдущем нашем примере будем использовать явную загрузку библиотеки для вызова функции *ShowAbout*. В принципе, если уже делать явную загрузку, то для всех функций и процедур библиотеки, потому что если ты оставишь хотя бы одну неявной, то библиотека всё равно будет грузиться на этапе старта программы. Но для примера я взял только одну функцию, а процедуру попробуй перевести на явную загрузку сам. Тем более, что для этого не надо делать много изменений.

Итак, загружай приложение, написанное в прошлой части, которое использует динамическую библиотеку. В основном модуле убирай объявления функции *ShowAbout*, чтобы ничего не осталось. Теперь в разделе **type** пиши объявление нового типа:

```
ShowA=function (Handle: THandle):LongInt;stdcall;
```

Здесь я объявляю новый тип *ShowA*, который равен функции с параметрами фкнуции *ShowAbout* из динамической библиотеки. Параметры должны быть точными, как при объявлении, иначе могут возникнуть проблемы.

Всё, этого достаточно. Теперь нужно переходить к обработчику события *OnClick* для первой кнопки, где мы показывали окно. Подкорректируй уже имеющийся код до следующего:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 dllHandle:THandle;
 sa:ShowA;
begin
 if f>0 then
  FreeAbout(f);

 dllHandle:=LoadLibrary('ProjectDLL.dll');

 if dllHandle=0 then
  exit;//Библиотека не загрузилась

 @sa:=GetProcAddress(dllHandle, 'ShowAbout');

 if @sa=nil then
  exit;//Функция не найдена

 f:=sa(Handle);
 FreeLibrary(dllHandle);
end;
```

Здесь у меня объявлено две локальные переменные:
dllHandle – здесь будет храниться указатель на загруженную библиотеку.
sa – имеет тип *ShowA*, т.е. тип функции из библиотеки.

В начале кода я выполняю уже знакомую проверку переменной *f*. Если она больше нуля, то окно уже показывалось и нужно освободить память старого окна, прежде чем создавать новое.

Дальше, я вызываю функцию *LoadLibrary*. Эта функция загружает указанную в качестве параметра динамическую библиотеку в память. Результатом выполнения функции является указатель на загруженную библиотеку. Этот указатель я сохраняю в переменной *dllHandle*. После этого нужно проверить, если указатель *dllHandle* равен нулю, то библиотека не загрузилась.

Теперь нам надо получить адрес функции *ShowAbout* в загруженной памяти, чтобы мы могли выполнить процедуру. Для этого я вызываю функцию *GetProcAddress*. Процедуре нужно передать два параметра:

1. Указатель на загруженную библиотеку.
2. Имя искомой процедуры.

Результатом будет адрес искомой функции, и я его сохраняю по адресу переменной *@sa*. Теперь *sa* указывает на адрес, по которому загружена библиотека *ShowAbout*. Единственное, что надо проверить – корректность адреса. Если он равен **nil**, то процедура не найдёна (возможно, это старая версия библиотеки или неправильно указано имя).

Если всё нормально, то я вызываю функцию через переменную *f:=sa(Handle)*, почти так же, как это делалось раньше. Результат выполнения функции сохраняется в переменной *f*.

Последняя строчка кода выгружает динамическую библиотеку из памяти - *FreeLibrary*. Точнее сказать, на этом этапе реальной выгрузки не происходит. Функция только сообщает системе о том, что больше библиотека программе не нужна. Если эту библиотеку использует другая программа (я же говорил, что одну библиотеку может использовать одновременно несколько программ), то она останется в памяти, пока та не сообщит о ненадобности в загруженном DLL файле.

 *На компакт диске, в директории* ***\Примеры\Глава 18\CallFunc*** *ты можешь увидеть пример этой программы.*

18.7. Точка входа.

Ты наверно заметил, что в исходнике библиотеки есть begin и end не относящиеся к ни одной из процедур или функций. Код, описанный здесь, выполняется самым первым при загрузке библиотеки в память. Но зачем это нужно? Здесь можно было бы инициализировать какие-то переменные, но библиотека не может храниться их.

В библиотеках есть одна глобальная переменная, которая существует всегда и её имя *DLLProc*. Это не просто переменная, а указатель на процедуру. По умолчанию он равен **nil**, но если сюда записать адрес реальной процедуры, то эта процедура может вызываться на определённые события, происходящие в библиотеке. В процедуре можно вылавливать следующие события:

DLL_PROCESS_ATTACH – это событие генерируется при загрузке библиотеки.
DLL_PROCESS_DETACH – это событие генерируется при выгрузке библиотеки.
DLL_THREAD_ATTACH – при создании нового потока.

DLL_THREAD_DETACH – при отключении нового потока.

Честно скажу, что всё это тебе может и не пригодиться, но я всё же дам маленький пример, чтобы ты увидел, как это работает.

Открываем нашу библиотеку, написанную в прошлой части. Теперь добавляем в неё следующий код:

```
library ProjectDLL;

uses
  SysUtils,
  Classes,
  Windows,
  dialogs,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

exports ShowAbout index 10;
exports FreeAbout index 11;

procedure DLLEntryPoint(dwReason:DWord);
begin
 case dwReason of
  DLL_PROCESS_ATTACH:ShowMessage('Attach to process');
  DLL_PROCESS_DETACH:ShowMessage('Detach to process');
  DLL_THREAD_ATTACH:ShowMessage('Thread attach to process');
  DLL_THREAD_DETACH:ShowMessage('Thread detach to process');
 end;
end;

begin
 DLLProc:=@DLLEntryPoint;
 DLLEntryPoint(DLL_PROCESS_ATTACH);
end.
```

В разделе **uses** появилось объявление двух новых модулей *windows* и *dialogs*, без них наш код не скомпилируется. Чуть дальше появилась процедура *DLLEntryPoint* с одним параметром, в котором будет передаваться событие, которое произошло. Внутри процедуры я проверяю оператором **case** тип пришедшего сообщения и в зависимости от этого вывожу сообщение.

Между **begin** и **end** библиотеки я назначаю переменной *DLLProc* нашу процедуру. После этого я вызываю её и в качестве параметра указываю событие *DLL_PROCESS_ATTACH*.

*На компакт диске, в директории **\Примеры\Глава 18\Entry** ты можешь увидеть пример этой программы.*

18.8. Вызов из библиотек процедур основной программы.

Теперь я хочу тебе показать ещё один трюк с использованием DLL библиотек. Мы уже познали многое и пора увидеть, как можно из библиотеки DLL вызывать процедуры, описанные в основной программе. Это очень сильный

способ сообщать основной программе, о каких либо событиях происходящих в библиотеке.

Создай новый проект динамической библиотеки. В основном модуле напишем следующий код:

```
library FuncProject;

uses
  SysUtils,
  Classes;

type
 TCompProc= procedure(Str:PChar);StdCall;

procedure CompS(Str:PChar; Proc:TCompProc);StdCall;
begin
 if @Proc<>nil then
  TCompProc(Proc)(Str);
end;

exports CompS index 10;

{$R *.res}

begin
end.
```

Первое, что здесь бросается в глаза – объявление в разделе **type** нового типа – *TCompProc*. Новый тип объявлен как процедура с одним параметром в виде переменной типа *PChar* и имеющей стандартный вызов. Объявление этого типа необходимо, чтобы объяснить динамической библиотеке, какого вида будет процедура в основной программе, которую надо будет вызывать.

Напоминаю, что тип *PChar* – это указатель на строку оканчивающуюся нулём (шестнадцатеричный #0). Сама переменная типа *PChar*, это только указатель на начало строки, а конец строки определяется по наличию значения #0. Но о конце строки нам никогда не придётся заботиться, нас больше будет волновать выделенная память, потому что под строку типа *PChar* нужно резервировать память.

Но всё это небольшое отступление и напоминание уже пройдённого материала, так что продолжим рассматривать наш модуль. После объявление нового типа процедуры идёт процедура CompS, которая будет экспортироваться из модуля. Эту процедуру мы будем вызывать из основной программы, а из неё уже будем обращаться к процедуре основной программы.

В первой строке процедуры я проверяю значение переданного параметра *Proc*. В этом параметре мы должны получать адрес процедуры, которую надо вызвать. Если параметр не равен нулю, то можно вызывать процедуру. Для вызова я пишу следующий код: *TCompProc(Proc)(Str)*. Этим кодом я вызываю процедуру и передаю ей в качестве параметра переменную *Str*, которую мы сами же получили в качестве входного параметра.

В принципе, с процедурой всё. Остальное тебе уже должно быть знакомо. Переходим к написанию основного модуля.

Создай новое приложение. В разделе **type** сразу же объяви следующее:

```
type
```

```
TCompProc= procedure(Str:PChar);StdCall;
procedure CompS(Str:PChar; Proc:TCompProc);export;StdCall;
```

В первой строке я объявляю тот же процедурный тип, что и в динамической библиотеке. Во второй строке объявляется процедура, которую мы экспортируем из библиотеки. Объявление должно быть именно в таком порядке. Если ты попытаешься объявить сначала процедуру из библиотеки, то при компиляции Delphi выдаст ошибку, потому что в качестве второго параметра в процедуре стоит тип *TCompProc* и сначала его нужно описать, а потом использовать.

Теперь напишем процедуру *CallFromDLL*. Эта процедура будет вызываться из динамической библиотеки. Она будет выглядеть так:

```
procedure CallFromDLL(Str:PChar);StdCall;
begin
 ShowMessage('DLL вызвала эту процедуру. Параметр равен: '+Str);
end;
```

Наша процедура должна соответствовать объявленному типу *TCompProc*, а именно, в типе описано, что это процедура, что она имеет один параметр типа *PChar* и вызывается стандартно. Процедура должна соответствовать всему этому описанию, иначе произойдёт ошибка.

Внутри процедуры я вызываю только одну функцию *ShowMessage*, которая показывает на экран окно сообщения. В качестве единственного параметра нужно указать текст сообщения.

Теперь пометим на форму кнопку и по её нажатию напишем следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 CompS('Привет', @CallFromDLL);
end;
```

Здесь я просто вызываю процедуру *CompS*, хранящуюся в динамической библиотеки. Попробуй запустить приложение и проверить результат работы программы.

*На компакт диске, в директории **\Примеры\Глава 18\Call** ты можешь увидеть пример этой программы.*

Глава 19. Разработка собственных компонентов

На протяжении всей книги мы уже использовали достаточно много компонентов Delphi. Как видишь, среда разработки достаточно хорошо продумана и имеет очень много инструментов для создания полноценных приложений. Но готовых компонентов, как и денег никогда не бывает слишком много. Всегда хочется всё больше и больше.

Зайди на сайт www.torry.net и посмотри раздел VCL. Тут целое море компонентов, написанных разными программистами одиночками и маленькими фирмами. Некоторые даже зарабатывают этим деньги (я не думаю, что большие, но всё же).

Допустим, что ты написал какой-то код, который может пригодиться тебе в дальнейшем. Можно оформить этот код в виде DLL файла, но это не всегда возможно. Вдруг твой компонент реализует часы, которые должны выводиться на главной форме, а DLL не может такого позволить.

В этом случае лучше всего подходят компоненты. Ты можешь создать свои компоненты похожие на те, что ты видишь на палитре компонентов и потом многократно использовать написанный там код. Если ты сделаешь так, то в будущем достаточно будет только установить твой компонент на форму и его можно будет использовать так же, как и любой другой компонент.

Я надеюсь, что я заинтересовал в прочтении этой главы, если это так, то можешь приступать к её чтению, чтобы побыстрей узнать, как всё это делается. Но даже если ты не будешь создавать собственные компоненты, я всё же советую эту главу не пропускать, потому что здесь будет описано много теории о внутренностях компонентов. Ты узнаешь из чего они состоят и лучше будешь понимать, как они работают.

19.1 Пакеты.

Любой компонент, который устанавливается в Delphi должен попасть в какой-то пакет. Для этих целей по умолчанию уже есть один пакет, но ты можешь создавать новые пакеты. Таким образом, компоненты можно группировать по смыслу в разные пакеты. Например, те, что работают с графикой складывать в один пакет, а те, что работают с файлами в другой.

Давай посмотрим, как работать с пакетами. Для начала создай где-нибудь у себя на компьютере папку *Components*. В неё ты будешь складывать все созданные или скаченные с интернета компоненты. Внутри этой папки создай ещё одну папку *Other*. Компонент, который мы сейчас проинсталлируем трудно отнести к какой-нибудь категории, поэтому я его поместил в папку с таким названием.

На диске, в директории *Компоненты/Handles* ты найдёшь исходники компонента, который мы будем сейчас устанавливать. Скопируй найденный там файл в папку *Other*. Скопировать нужно файла *Handles.pas* – здесь находиться исходник компонента.

Теперь запусти Delphi и закрой всё, что в нём открыто (*File->Close All*). Чтобы установить компонент выбери из меню *Component* выбери пункт *Install Component*. Перед тобой откроется окно, как на рисунке 19.1.1. В этом окне ты можешь увидеть три строки ввода:

1. *Unit file* name. В первой строке нужно указать имя устанавливаемого компонента. Для этого щёлкни кнопку *Browse* и увидишь стандартное окно открытия файла. Открой файл *Handles.pas*.

2. *Search Path* – здесь находиться список путей, по которым Delphi при компиляции ищет исходники установленных компонентов. В принципе, в конец этой строки мы должны были бы добавить после точки с запятой и путь к нашему компоненту, но пока этого делать не будем.

3. *Package file name*. Это имя пакета, в который будет помещён устанавливаемый компонент. Здесь можно выбирать в выпадающем списке любой пакет.

4. *Package description*. Текстовое описание пакета. Здесь может быть любой текст заданный тобой.

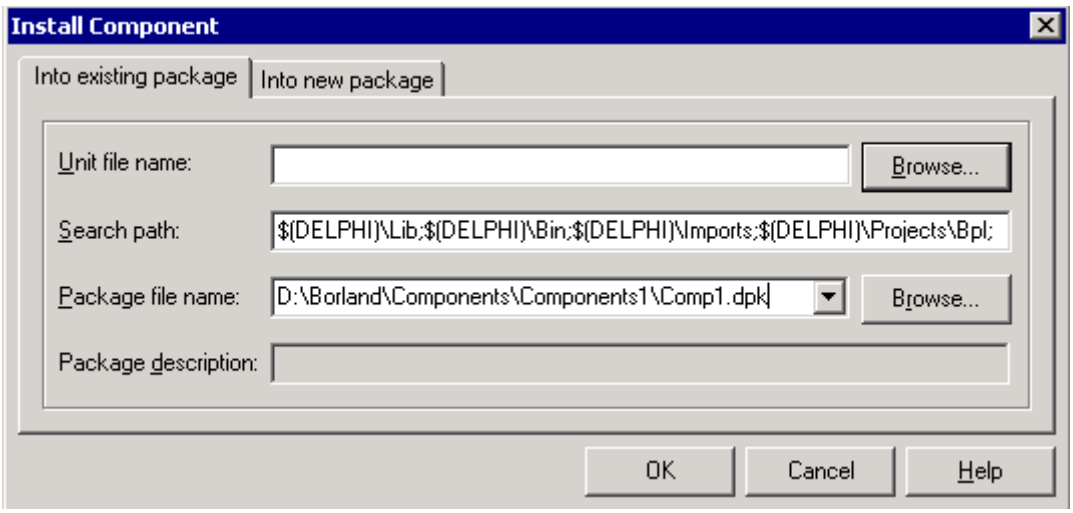


Рисунок 19.1.1. Окно установки компонента.

Будем считать, что у тебя в системе ещё нет своих пакетов и Delphi предложил установить новый компонент в пакет по умолчанию. Мы не будем этого делать, потому что так у нас в системе будет бардак. Давай создадим новый пакет, в который будем

помещать все файлы из директории *Components/Other*. Для этого в этом же окне выбери сверху окна закладку *Into new package* и нажми кнопку *Browse* напротив строки *Package file name*. В появившемся стандартном окне открытия файлов перейди в директорию *Components/Other* и здесь вручную введи имя пакета *OtherComponents* и нажми кнопку *«Открыть»*.

Теперь нажимай кнопку ОК, чтобы закрыть окно установки компонента. Перед тобой автоматически должно появиться сообщение типа: «*Package OtherComponents.bpl will be build then installed. Continue?*». Смысл сообщения следующий «Пакет *OtherComponents* будет откомпилирован и установлен в систему. Продолжить?». Можешь нажать «*Yes*» и Delphi сделает всё необходимое для установки нового компонента в систему. Я нажму «*No*» и покажу как это делать не автоматически и что же произошло.

Для начала у нас появилось новое окно, показанное на рисунке 19.1.2. В центре окна ты можешь видеть дерево из двух веток:

Contains – здесь содержаться модули входящие в пакет. У нас пока один модуль и ты видишь только его файлы.

Requires – здесь находиться список имён пакетов необходимых для компиляции данного пакета. В принципе, эти имена ты можешь удалить (иногда они не нужны), но если при компиляции хотя бы один из этих пакетов понадобиться, то появиться сообщение о необходимости подключения данных пакетов и Delphi снова их вернёт автоматически.

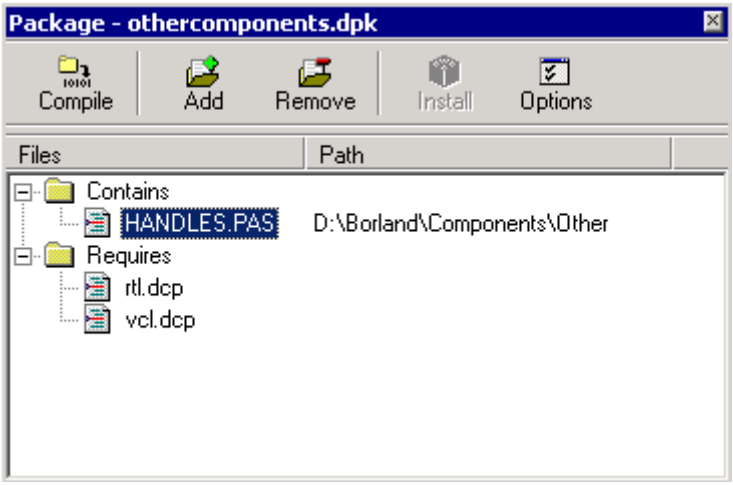


Рисунок 19.1.2. Окно пакета.

Сверху окна ты можешь видеть панели со следующими кнопками:

Compile – компилировать пакет.

Add – добавить новый модуль в этот пакет.

Remove – удалить модуль.

Install – установить пакет в систему. При нажатии этой кнопки, пакет при необходимости будет перекомпилирован.

Options – настройки пакета.

Прежде чем компилировать наш пакет, давай посмотрим его настройки. Нажми кнопку *Options* и ты увидишь окно, как на рисунке 19.1.3.

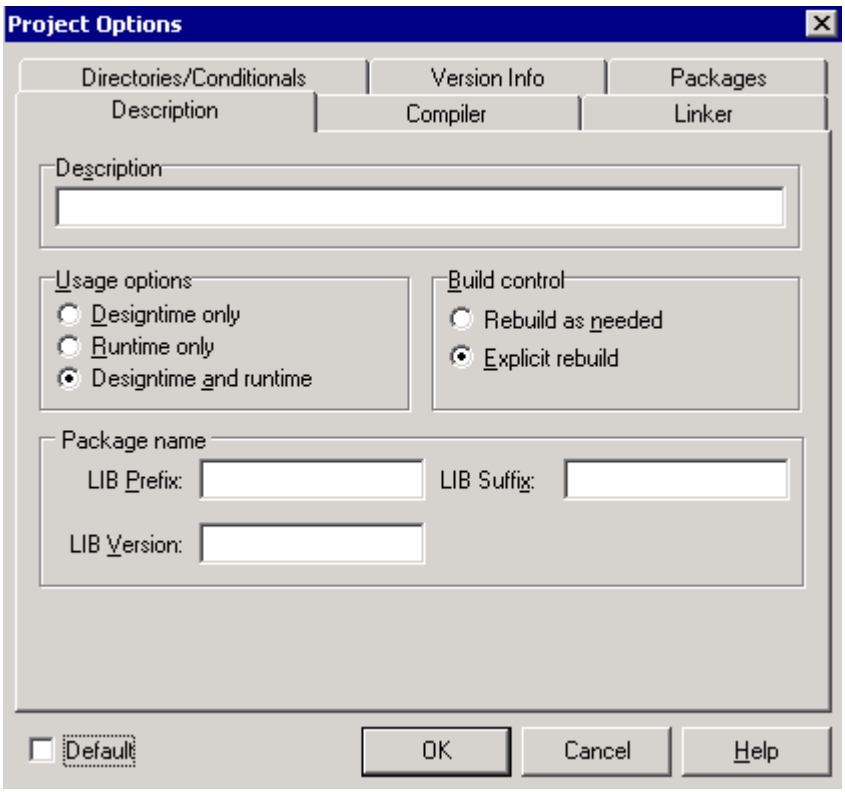


Рисунок 19.1.3. Окно свойств пакета.

В принципе, большинство настроек схоже с настройками запускных файлов, но я не это хотел тебе показать. Обрати внимание на раздел *Usage options*, где ты указываешь, для чего будет использоваться пакет. Тебе доступны тут три варианта

1. *Designtime only* – компоненты пакета могут использоваться только во время проектирования формы в оболочке Delphi.
2. *Runtime only* – компоненты нельзя использовать во время проектирования, а можно только создавать во время выполнения программы. Для этого, на машине запускающей программу, обязательно должен присутствовать откомпилированный файл проекта (файл с тем же именем и расширением bpl).
3. *Designtime and Runtime* – компоненты пакета можно использовать в обоих случаях.

Чаще всего используют именно третий пункт, а второй ставят для коммерческих пакетов, которые распространяются за деньги. В этом случае, пользователь может познакомиться с возможностями твоих компонентов, но только используя их во время выполнения программы. Если он захочет использовать компоненты и во время проектирования формы в оболочке Delphi, то за это можно брать отдельную плату.

Но это всё лирическое отступление, чтобы показать тебе возможности пакетов. Чуть позже я покажу тебе, чем отличаются *Designtime* и *Runtime* пакеты на практике, а пока закрывай окно свойств пакета и возвращайся в окно пакета.

Нажми кнопку *Install* чтобы установить пакет в систему. Если кнопка *Install* недоступна, то сначала откомпилируй проект, а потом установи. Delphi откомпилирует пакет и выведет сообщение об удачной установке. Закрой окно пакета. При закрытии Delphi спросит о необходимости сохранить его, ответь «*Да*», чтобы все изменения сохранились.

Теперь на палитре компонентов у тебя должна появиться новая закладка с именем «CyD», где ты сможешь найти новый компонент. Некоторые компоненты попадают на закладку *Symples*, а для некоторых создаются новые закладки.

Теперь снова закрой всё (*File->Close All*) и создай новый проект приложения. Сейчас я покажу тебе, как используются *Designtime* и *Runtime* пакеты. Выбери пункт меню *Options* из меню *Project*. Перед тобой откроется окно свойств проекта, как на рисунке 19.1.4. В этом окне перейди на закладку *Packages*

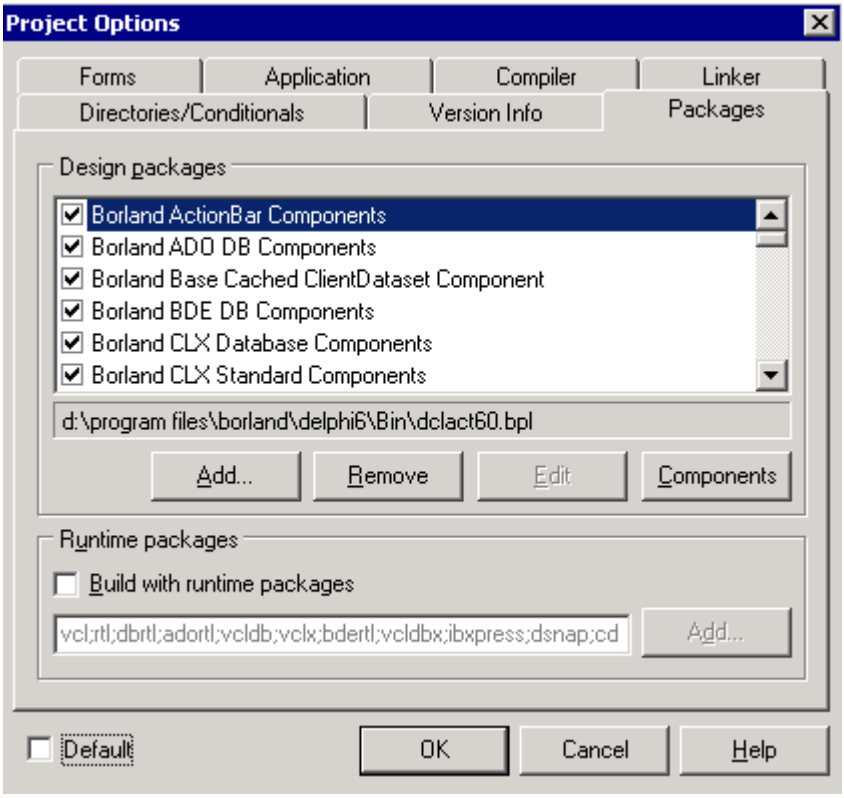


Рисунок 19.1.4. Окно свойств проекта.

Здесь, внизу окна ты можешь видеть *CheckBox* с текстом: «*Build with runtime packages*». Если здесь поставить галочку, то твои программы резко уменьшаются в размере, потому что в них будет храниться только код программы. Все компоненты, которые установлены на форме не попадут в запускной файл.

Когда программа будет запускаться, она будет подгружать эти компоненты из файлов пакетов перечисленных в строке чуть ниже *CheckBox*. Получается, что пакеты bpl могут работать как самые настоящие динамические библиотеки. Так ты можешь сильно экономить размер программ, но при переносе программы на другой компьютер, ты должен заботиться о том, чтобы и необходимые bpl файлы тоже попали на тот компьютер. Их достаточно скопировать в папку System (System32 для Windows NT/2000/XP) директории Windows.

Если убрать этот флажок, то программа становиться полноценной и не нуждается в дополнительных bpl файлах. Исключения составляют только случаи, когда ты использовал *Runtime* пакет, который не может компилироваться в программу и обязательно должен использоваться только на этапе работы программы. Но такие пакеты редкость и я не советую тебе их использовать. Работай с теми компонентами, которые можно установить на форму во время проектировки приложения.

Теперь проверим путь к нашему компоненту. Выбирай из меню *Tools* пункт *Environment Options*. Перед тобой откроется окно настроек Delphi. В этом окне нужно перейти на закладку *Library* (рисунок 19.1.5).

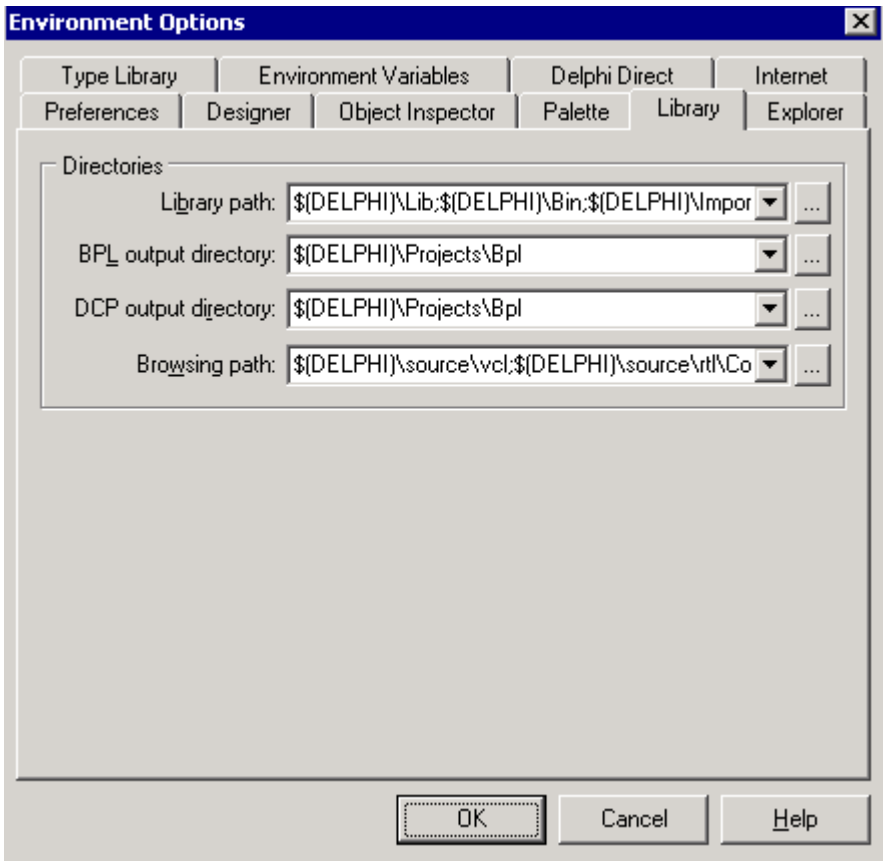


Рисунок 19.1.5. Окно свойств проекта.

В этом окне ты можешь увидеть следующие строки ввода:

Library Path – здесь перечислены пути, где Delphi должен искать исходники компонентов.

BPL output directory – здесь указывается директория, куда будут сохраняться откомпилированные пакеты. По умолчанию здесь указано *$(DELPHI)\Projects\Bpl*. Конструкция *$(DELPHI)* указывает на директорию, куда установлен Delphi. Получается, что bpl файлы будут сохраняться в директорию, где установлен Delphi, поддиректорию *Projects\Bpl*.

DCP output directory – директория, в которую будут помещаться DCP файлы. Это скомпилированные файлы пакета, которые хранят все информацию о всех компонентах скомпилированных в пакет.

Browsing Path – пути для просмотра.

Нас сейчас интересует только первая строка, остальные вообще можно оставлять по умолчанию. Нажми на кнопку с тремя точками справа от строки *Library Path*. Перед тобой откроется окно редактора путей, как на рисунке 19.1.6.

В списке в центре окна ты можешь увидеть список директорий, в которых Delphi будет искать исходники. Когда ты выбираешь любой из них, то этот путь перемещается в строку ввода под списком.

Проверь список на наличие пути к директории, где у нас находиться исходник файла *Handles.pas*. Если такого пути нет, то щёлкни по кнопке с тремя точками справа от строки ввода и ты увидишь окно выбора директории. Найди нужную директорию и нажми OK. Теперь выбранный путь находиться в строке ввода. Чтобы его добавить в список, нужно нажать кнопку “Add”. Сделай это и можешь закрывать окно редактора путей нажатием кнопки «*OK*». После этого закрой и окно настроек Delphi.

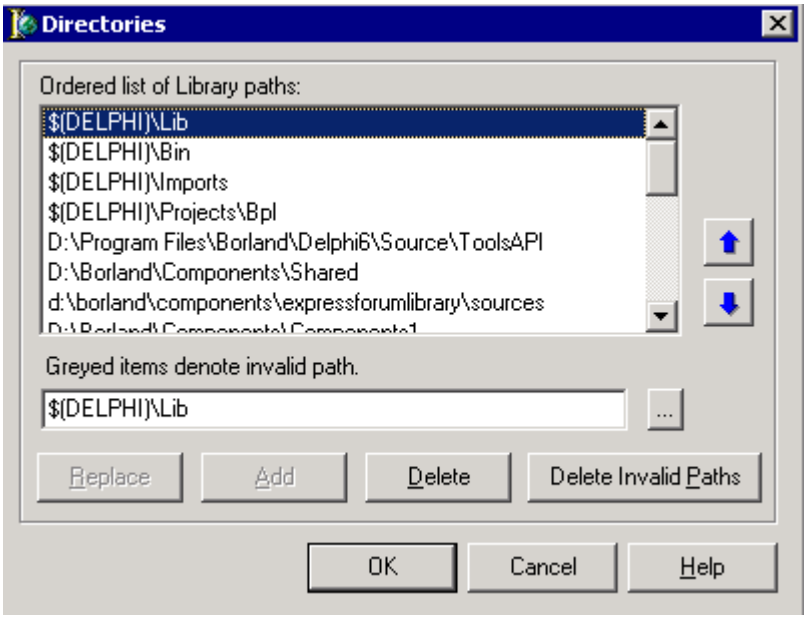


Рисунок 19.1.6. Окно редактора путей.

Если ты не добавишь путь к исходникам в настройки Delphi, то компилятор не найдёт исходный код и при компиляции выдаст ошибку.

Когда ты устанавливаешь *Runtime* пакет, то тебе необходимо найти файл с расширением bpl в директории *$(DELPHI)\Projects\Bpl* и скопировать его в системную папку Windows.

Если у тебя уже есть где-то откомпилированный пакет (файл с расширением bpl), то его можно сразу же установить в Delphi без компиляции. Но даже и в этом случае наличие исходников компонентов из пакета обязательно. Исходники надо будет сложить в доступную для Delphi директорию, или создать новую директорию, но добавить путь в настройках Delphi.

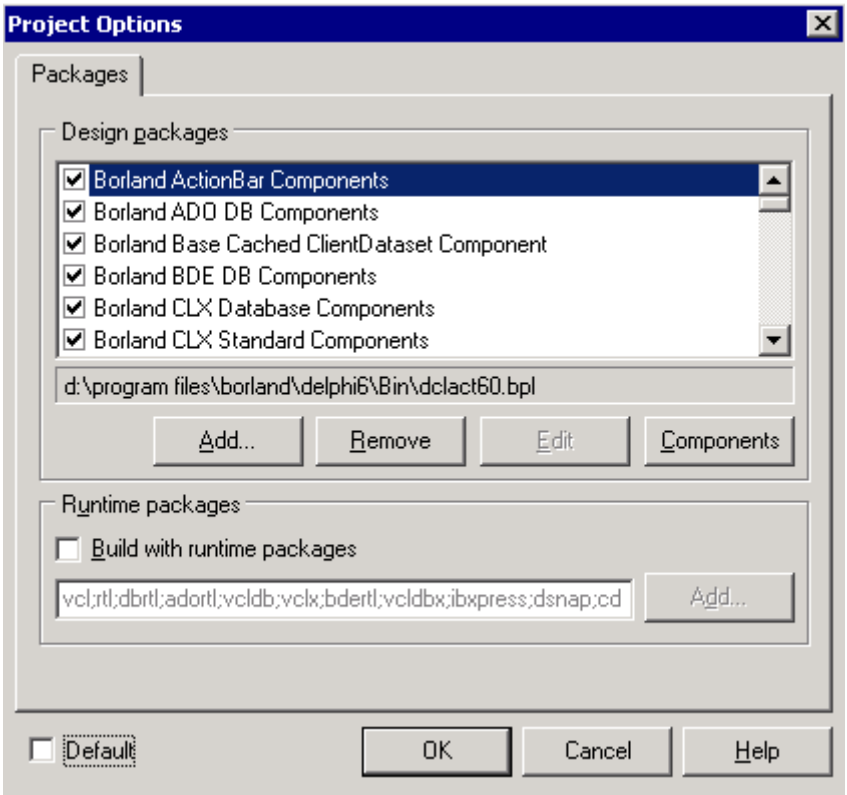


Рисунок 19.1.7. Окно установки пакетов.

Для установки уже скомпилированного пакета выбери из меню *Component* пункт *Install Packages*. Перед тобой откроется окно (рисунок 19.1.7), которое похоже на закладку *Packages* окна свойств проекта, которое ты мог видеть на рисунке 19.1.4.

Для установки нового пакета нужно нажать кнопку *Add*. Перед тобой откроется стандартное окно открытия файлов. Найди нужный bpl файл и открой его. Delphi автоматически установит все компоненты из пакета в палитру компонентов.

19.2 Подготовка к созданию компонента.

Прежде чем начинать создавать собственный компонент, необходимо решить для себя несколько вопросов. Самым первым ты должен решить, от какого компонента/объекта будет происходит твоё детище. Как ты уже заметил, все объекты в Delphi происходят от объекта *TObject*. А все компоненты имеют среди родственников объект *TComponent*. На рисунке 19.2.1 показана иерархия предков компонента *TButton*.

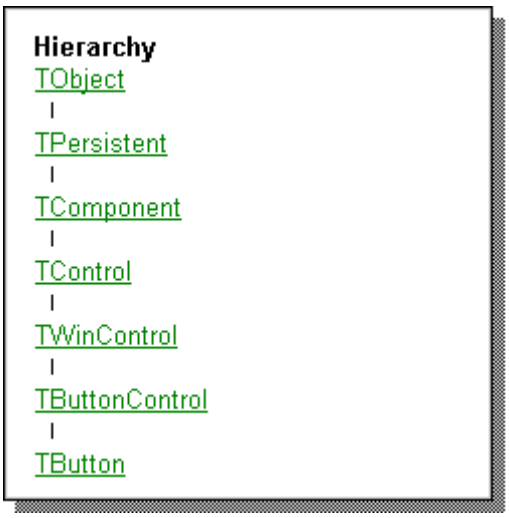


Рисунок 19.2.1. Иерархия предков компонента TButton.

В самом верху иерархии находиться объект *TObject*. Как я уже говорил, абсолютно все объекты происходят именно от него. В *TObject* находятся базовые свойства и методы, которые необходимы любому другому объекту. Именно поэтому, чтобы во всех объектах не прописывать одни и те же свойства и методы, всё уже реализовано в *TObject*. Остальные просто наследуют эти возможности.

Далее по иерархии идёт объект *TPersistent*, который является предком для всех объектов, которые должны уметь назначаться другим объектам. Например, если наш объект должен уметь выполнять метод *Assign* (назначить), то этот объект должен иметь в предках объект *TPersistent*.

Следующим в иерархии идёт объект *TComponent*. Этот объект должен быть предком для любых компонентов Delphi, которые должны уметь ставиться на форму в режиме проектирования. Этот объект наследует все свойства и методы своих предков (*Tpersistent* и *TObject*) и добавляет новые возможности по работе с объектом, как с полноценным компонентом на форме.

Если компонент должен быть видимым во время выполнения программы, то он обязательно должен происходить от компонента *TControl* (следующий в иерархии для

кнопки). Для невидимых компонентов (например компоненты с закладки *Dialogs*, которые не видны во время выполнения), этот объект среди предков не нужен.

Следующий в иерархии идёт *TWinControl*. Этот объект добавляет функции получения фокуса ввода, работы с текстовым буфером, возможность содержания дочерних компонентов. Если твой компонент будет иметь среди предков *TWinControl*, то поверх этого компонента можно будет ставить ещё компоненты (не обязательно, но возможно) в режиме дизайна формы. *TWinControl* – имеет дескриптор (*Handle*) окна и может получать фокус. Вспомни пример с потоками, где мы использовали для вывода текста из потока компонент *TLabel*. Когда мы решили подкорректировать поток и добавить возможность посылать сообщения *SendMessage*, то нам пришлось заменить *TLabel* на *TEdit*, потому что у первого не было дескриптора окна и мы не могли ему отсылать сообщения Windows.

Следующий – *TButtonControl*. Это уже компонент кнопки, в котором реализуется множество необходимых кнопке свойств и методов.

Нужно ещё сказать об одном базовом для некоторых компонентов – *TGraphicControl*. Если твой компонент должен будет иметь метод *Paint*, т.е. уметь рисовать на поверхности графику, то он должен иметь среди предков этот объект.

И последний базовый объект – *TCustomControl* – это сочетание двух объектов – *TGraphicControl* и *TWinControl*, т.е. компонент имеющий дескриптор окна и умеющий метод *Paint*.

Иерархия предков компонента читается сверху вниз. Каждый новый объект иерархии добавляет уже существующую структуру новыми свойствами и методами. Таким образом, в конечном итоге мы получаем полноценный, самостоятельный объект.

Прежде чем создавать свой собственный компонент, ты должен решить от какого уже существующего объекта он будет происходить. Выбор должен зависеть от необходимых будущему компоненту возможностей.

Открой файл помощи в Delphi (меню *Help->Delphi help*). В появившемся окне найди объект *TButton* (рисунок 19.2.2). Для этого введи в строку ввода сверху имя *TButton*.

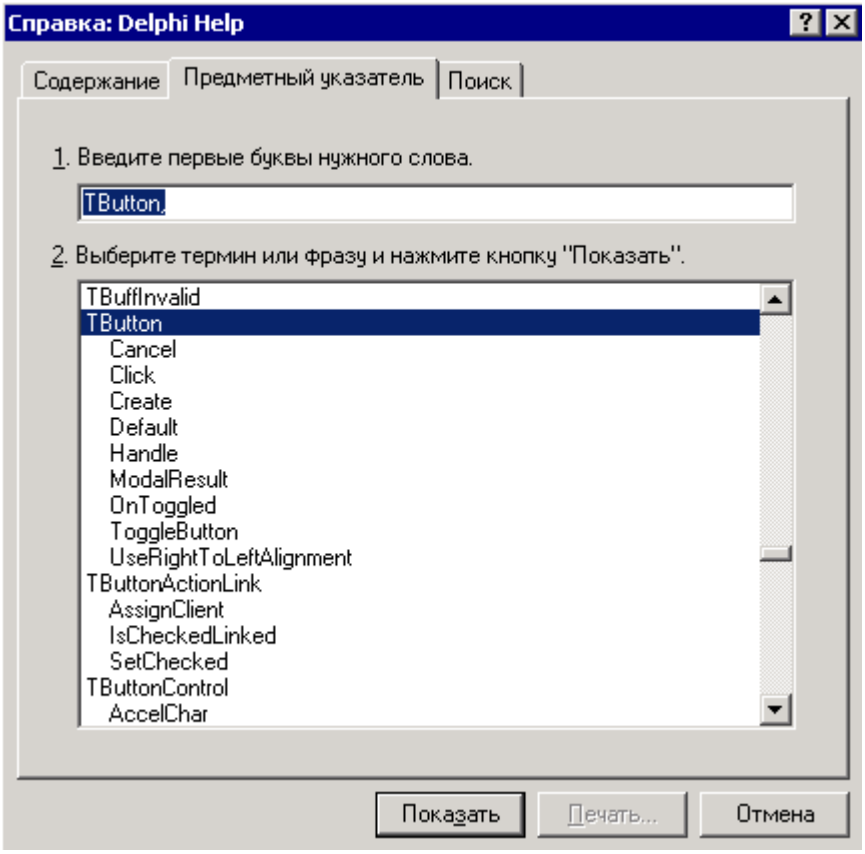


Рисунок 19.2.2. Поиск компонента TButton в файле помощи.

Теперь в большом списке выдели строку *TButton* и нажми кнопку «*Показать*». Перед тобой откроется окно с найденными разделами (рисунок 19.2.3). В данном случае список будет состоять из двух строк:

1. TButton
2. TButton (VCL Reference)

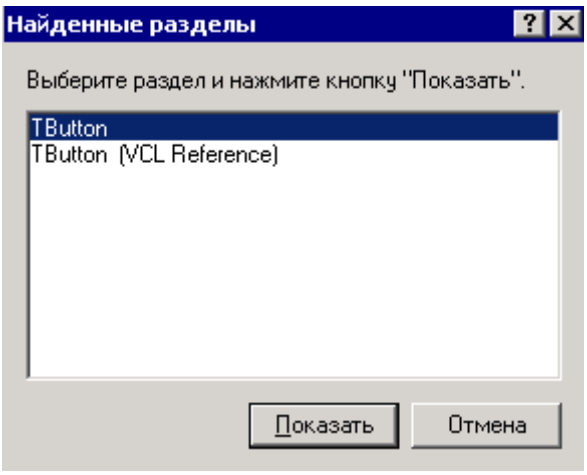


Рисунок 19.2.3. Список найденных разделов.

Первая строка явно относиться к объекту *TButton* из библиотеки CLX (если ничего не указано, то это скорей всего библиотека CLX). Вторая строка – это объект в библиотеке VCL. Оба варианта компонента схожи и имеют очень много одинаковых свойств и методов, но могут быть и отличия в VCL версии специфичные для платформы Windows. Мы в книге рассматриваем платформу Windows, поэтому выбирай VCL вариант и нажимай кнопку «*Показать*».

Перед тобой откроется окно помощи по выбранному объекту и оно должно выглядеть, как на рисунке 19.2.4.

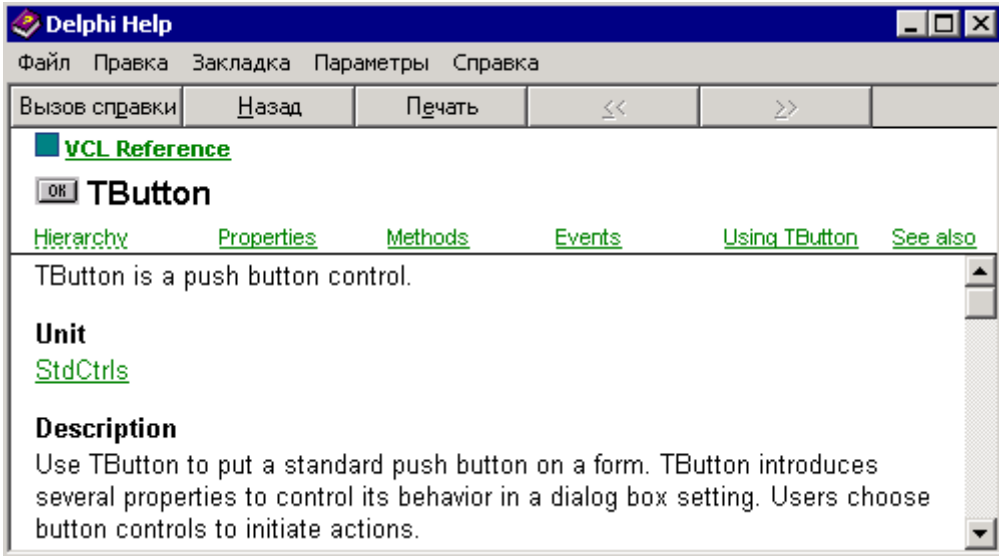


Рисунок 19.2.4. Список найденных разделов.

Сверху окна ты можешь видеть следующие ссылки:

1. *Hierarchy* – если щёлкнуть по этой ссылке, то перед тобой откроется окно, как на рисунке 19.2.1 с иерархией компонента.

2. *Properties* – здесь тебе покажут все свойства компонента.

3. *Methods* – это метода компонента.

4. *Events* – события, которые может генерировать компонент.

Попробуй посмотреть свойства. Щёлкни по ссылке *Properties* и перед тобой откроется окно, как на рисунке 19.2.5. В этом окне ты можешь увидеть все свойства разбитые по разделам. Большим жирным шрифтом написаны имена разделов, а после этого идут ссылки на свойства. Щёлкая по любой ссылке, можно получить её описание.

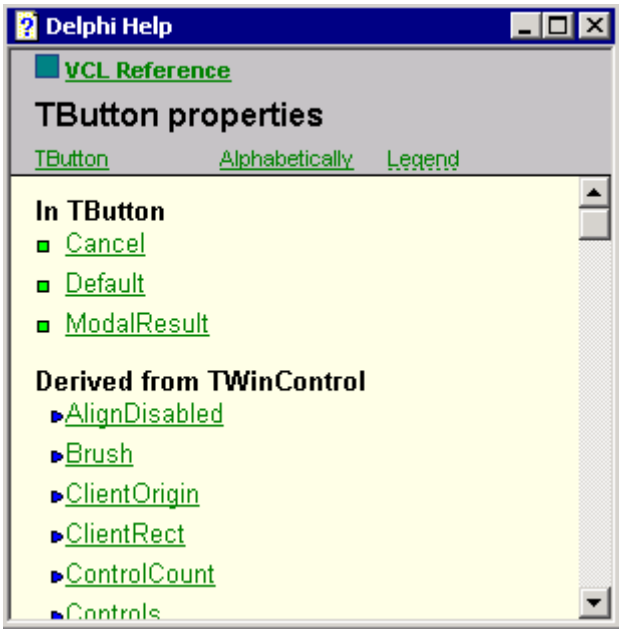


Рисунок 19.2.5. Свойства компонента TButton.

Обрати внимание на имена разделов. Например **Derived from TWinControl**. Судя по названию, в этом разделе будут перечислены все свойства, которые компонент получил от объекта *TWinControl*. Оно так и есть. Выбирая любой компонент ты можешь увидеть его и унаследованные от предков свойства. То же самое и с методами.

19.3 Создание первого компонента.

Теперь давай попробуем создать первый собственный компонент. Во время создания я буду постоянно рассуждать так, как должен это делать ты. К тому же я выбрал не совсем простую задачу, чтобы в очередной раз потренироваться и в программировании.

В качестве примера я выбрал достаточно простой, но сложный математики пример - часы. Наши часы смогут работать как аналоговые и числовые. Так что по этим часам будет Москва сверятся.

Для создания нового компонента выбери из меню *Component* пункт *New Component*. Перед тобой откроется окно, как на рис 19.3.1, где ты должен будешь заполнить основные параметры будущего компонента.

Давай рассмотрим каждое окошечко в отдельности:

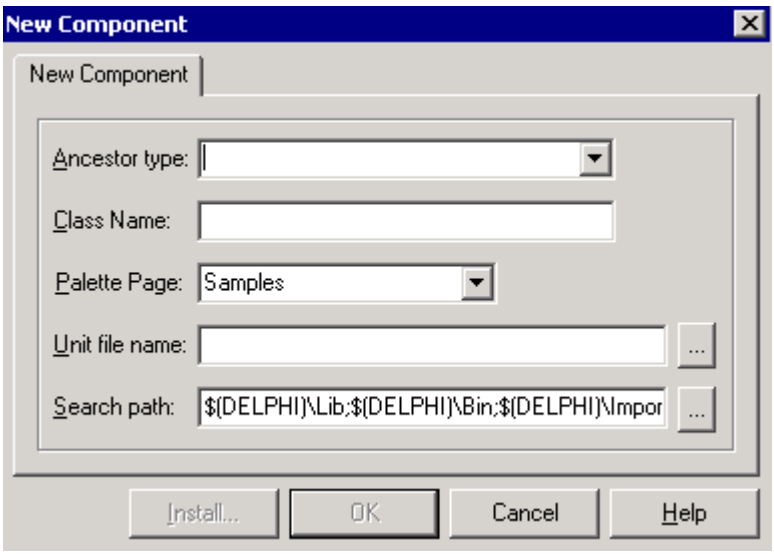


Рис 19.3.1. Окно создания нового компонента

Ancestor type - тип предка. Это имя объекта, от которого мы породим наш объект. Это даст нашему объекту все возможности его предка, плюс мы добавим свои. Для часиков нам понадобится *TGraphicControl*, потому что наш компонент должен будет иметь метод *Paint*, т.к. часы будут графическими.

Class Name - имя нашего будущего компонента. Я его назвал *TGraphicClock*.

Palette Page - имя палитры компонентов, куда будет помещён наш компонент после инсталляции. Я оставил значение по умолчанию "*Samples*". Но ты можешь поместить его даже на закладку "*Standard*".

Unit file name - имя и путь к модулю, где будет располагаться исходный код компонента. Это поле заполняется автоматически, но ты его можешь изменить. Если не хочешь менять, то хотя бы посмотри под каким именем сохранят твой компонент и где.

Search path – здесь перечислены пути, где Delphi ищет исходные коды. Если ты располагаешь компонент в новой директории, о которой Delphi ещё не знает, то обязательно нужно добавить её сюда.

Введи данные о будущем компоненте и нажимай "*OK*" (именно "*OK*", а не *Install*). После этого, Delphi создаст новый модуль с шаблоном для будущего компонента:

```
unit GraphicClock;

interface

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

type
  TGraphicClock = class(TGraphicControl)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;
```

```
implementation

procedure Register;
begin
  RegisterComponents('Samples', [TGraphicClock]);
end;

end.
```

В шаблоне реализована только одна процедура – *Register*. В этой процедуре происходит вызов *RegisterComponents* с двумя параметрами:

1. Имя закладки, на которую нужно будет поместить этот компонент.
2. Имя компонента, которое надо зарегистрировать при установке в системе Delphi.

Процедура *Register* обязательно должна присутствовать в любом модуле компонента. Она вызывается автоматически оболочкой Delphi при установке компонента.

В принципе простейший компонент готов и его можно установить в Delphi. Но этот компонент ничего не умеет и он пока является полным аналогом своего предка *TGraphicControl*. Чтобы он стал отличатся, сейчас мы добавим ему разные свойства.

Начнём написание нашего компонента с конструктора и деструктора. *Конструктор* - это метод объекта, который автоматически вызывается при создании компонента. *Деструктор* - тоже метод, только она автоматически вызывается при уничтожении компонента. Вызывать эти метода напрямую нельзя, а если и можно, то не желательно.

В конструкторе мы проинициализируем все наши переменные, которые понадобятся при его работе, а в деструкторе уничтожим.

Итак, напиши в разделе public:

```
constructor Create(AOwner: TComponent); override;
```

Как видишь, объявление метода похоже на объявление любой другой процедуры или функции, только вместо ключевого слова **procedure** стоит слово **constructor**. Ключевое слово **override**; после имени этих процедур говорит о том, что мы хотим переписать уже существующую у предка функцию с таким именем. У большинства компонентов есть конструктор и когда мы создаём конструктор у потомка, то у нас получается два метода с одним именем (у предка и у нашего объекта).

Теперь нажми сочетание клавиш CTRL+SHIFT+C и Delphi сам создаст заготовку для конструктора:

```
constructor TGraphicClock.Create(AOwner: TComponent);
begin
  inherited;
end;
```

Поправим её до вот такого вида:

```
constructor TGraphicClock.Create(AOwner: TComponent);
begin
//Вызываем конструктор предка
 inherited Create(AOwner);

//Устанавливаем значения ширины и высоты по умолчанию
```

```
 Width := 50;
 Height := 50;

//Устанавливаем переменную ShowSecondArrow в true.
//Она будет у нас отвечать за показ секундной стрелки
 ShowSecondArrow := true;

//Инициализируем остальные переменные
 PrevTime := 0;
 CHourDiff := 0;
 CMinDiff := 0;

//Инициализируем растры TBitmap, в которых будут хранится
//фон и сам рисунок часов.
 FBGBitmap:= TBitmap.Create;
 FFont:=TFont.Create;;

 FBitmap:= TBitmap.Create;
 FBitmap.Width := Width;
 FBitmap.Height := Height;

 //Выставляем формат времени
 DateFormat:='tt';

 //Запускаем таймер
 Ticker := TTimer.Create( Self);
 //Интервал работы таймера - одна секунда
 Ticker.Interval := 1000;
 //По событию OnTimer будет вызыватсья процедура TickerCall
 Ticker.OnTimer := TickerCall;
 //Включаем таймер
 Ticker.Enabled := true;

//Устанавливаем цвета поумолчанию
 FFaceColor := clBtnFace;
 FHourArrowColor := clActiveCaption;
 FMinArrowColor := clActiveCaption;
 FSecArrowColor := clActiveCaption;
end;
```

Ключевое слово **inherited** вызывает конструктор предка (в нашем случае *TGraphicClock*). Это необходимо, потому что предок тоже может делать что-то важное в конструкторе и если мы не вызовем его конструктор, то могут возникнуть проблемы.

В остальном, я надеюсь, что с конструктором всё ясно. Дальше идёт инициализация переменных. Я постарался снабдить код подробными комментариями, чтобы ты смог разобраться с происходящим. Сами переменные мы пока не добавили и я их буду описывать постепенно.

Теперь создадим деструктор. Для этого также опишем его в разделе **public**:

```
 public
   { Public declarations }
   constructor Create(AOwner: TComponent); override;
   destructor Destroy; override;
```

Деструктор тоже объявляется как простая процедура, но здесь стоит ключевое слово **destructor**. Теперь жмём Ctrl+Shift+C и получаем заготовку для деструктора и поправляем её до вида :

```
destructor TGraphicClock.Destroy;
begin
 Ticker.Free;
 FBitmap.Free;
 FBGBitmap.Free;
 inherited Destroy;
end;
```

Здесь я освобождаю всю память выделенную для хранения картинок и объекта *TTimer* в конструкторе. Заметь, что в конструкторе я вызывал предка в самом начале **inherited**, а в деструкторе в самом конце. В конструкторе сначала нужно, чтобы инициализировался предок (он проинициализирует необходимые ссылки), а потом можно инициализировать свои вещи. В деструкторе всё наоборот – сначала уничтожает мы, а потом предок. Если в деструкторе мы сначала вызовем предка, то последующая работа с компонентом уже может быть невозможна, потому что предок уничтожит все ссылки. Поэтому я ставлю этот вызов в самом конце.

Теперь опишем все необходимые нам переменные в разделе **private**:

```
private
  //Для часов обязательно понадобится таймер
  Ticker: TTimer;

  //Картинки часов и фона
  FBitmap, FBGBitmap: TBitmap;

  //События
  FOnSecond, FOnMinute, FOnHour: TNotifyEvent;

  //Центральная точка
  CenterPoint: TPoint;

  //Радиус
  Radius: integer;

  //Переменная отвечающая за показ секундной стрелки
  ShowSecondArrow: boolean;

  //Цвет фона часов
  FFaceColor: TColor;

  //Цвета стрелок часов
  FHourArrowColor,FMinArrowColor, FSecArrowColor: TColor;

  //формат даты
  FDateFormat: String;

  //Стиль шрифта
  FFont: TFont;

  //Остальные параметры, которые мы рассмотрим в процессе.
  LapStepW: integer;
```

```
  PrevTime: TDateTime;
  CHourDiff, CMinDiff: integer;
  FClockStyle:TClockStyle;
```

Теперь в разделе **private** опишем процедуру *TickerCall*. Мы её уже использовали в конструкторе. Она у нас вызывается по событию от таймера, но пока ещё не написали:

```
 protected
   { Protected declarations }
   procedure TickerCall(Sender: TObject);
```

Жмём CTRL+SHIFT+C и модифицируем созданную функцию:

```
procedure TGraphicClock.TickerCall(Sender: TObject);
var
 H,M,S,Hp,Mp,Sp: word;
begin
 //Если компонент создан в дизайнере, то выход
 if csDesigning in ComponentState then exit;

 //Иначе это уже запущеная программа

 //Получить время
 DecodeCTime( Time, H, M, S);
 //Получить предыдущее время.
 DecodeCTime( PrevTime, Hp, Mp, Sp);

 //Сгенерировать событие OnSecond
 if Assigned( FOnSecond) then FOnSecond(Self);
 //Сгенерировать событие  OnMinute
 if Assigned( FOnMinute) AND (Mp < M) then FOnMinute(Self);
 //Сгенерировать событие  OnHour
 if Assigned( FOnHour) AND (Hp < H) then FOnHour(Self);

 //Сохранить текущее время в PrevTime
 PrevTime := Time;

 if ( NOT ShowSecondArrow) AND (Sp <= S) then exit;

 //Прорисовать часы.
 DrawArrows;
end;
```

Процедура *DrawArrows* напичкана математикой и она сейчас не имеет для нас особого значения. Немного ниже я приведу её в полном исходнике, а сейчас я рассказываю о том, как создавать компоненты, поэтому мы рассмотрим события генерируемые в функции *TickerCall* и узнаем, как они генерируются.

У нас уже объявлено три события в разделе **private** компонента:

```
FOnSecond, FOnMinute, FOnHour: TNotifyEvent;
```

Все они появятся на закладке *Events* окна объектного инспектора, когда ты поставишь компонент на форму. Чтобы приложения могло поймать эти события, мы объявили переменные типа *TNotifyEvent* (это тип означает события).

Но всё это пока что переменные, а как же Delphi узнает, что это события, которые надо поместить на закладку *Events* объектного инспектора? Для этого, в разделе *published* мы должны описать само событие *OnSecond* и другие:

```
property OnSecond: TNotifyEvent read FOnSecond write FOnSecond;
property OnMinute: TNotifyEvent read FOnMinute write FOnMinute;
property OnHour: TNotifyEvent read FOnHour write FOnHour;
```

Вначале каждой строки стоит ключевое слово **property**, которое говорит о том, что мы объявляем свойство. После этого идёт имя свойства и после двоеточия стоит тип. Вот как раз по типу, Delphi и узнает, что объявленное свойство на самом деле событие.

После этого идёт ключевое слово **read**, за которым должна идти переменная или функция, которая будет использоваться при чтении события. У меня после этого ключевого слова стоит имя переменной, соответствующей событию. После ключевого слова **write** нужно ставить переменную или функцию, которая будет использоваться для записи в событие. Тут опять стоит соответствующая переменная.

Для генерации события мы пишем следующий код:

FOnSecond(Self) для события *OnSecond*.
FOnMinute(Self) для события *OnMinute*.
FOnHour(Self) для события *OnHour*.

При генерации события в качестве переменной передаётся *Self*. Эта переменная всегда указывает на объект, в котором мы сейчас находимся, в данном случае компонент *TGraphicClock*. Вспомни любой обработчик события. В любом из них есть как минимум один параметр – переменная *Sender*, которая указывает на объект, который сгенерировал событие. Теперь посмотри на код, которым мы генерируем событие. Как видишь, мы при генерации указываем объект *Self*, который генерирует событие и пользователь получит его в переменной *Sender* обработчика события.

Но нельзя вслепую генерировать событие. Прежде чем это делать, нужно проверить, есть ли обработчик события. Для этого нужно вызвать функцию *Assigned* и в качестве параметра указать тип события. Получается, что следующий код проверяет, установлен ли обработчик события OnSecond, и если да, то генерирует событие:

```
if Assigned( FOnSecond) then FOnSecond(Self);
```

Теперь наш компонент сможет генерировать события.

С событиями вроде всё ясно (если нет, то посмотри на исходник на диске и попробуй разобраться, глядя на общую картину). Теперь переходим к свойствам. В разделе **published** мы можем создавать свойства, которые будут отображаться в объектном инспекторе при выделении наших часиков. Все свойства, которые есть у предка нужно просто описать

```
published
  property Align;
  property Enabled;
  property ParentShowHint;
```

```
property ShowHint;
property Visible;
```

Все эти свойства мы получаем от предков *TGraphicControl*, *TControl* и так далее. Слово **property** говорит о том, что мы описываем свойство. Для них не нужны процедуры или функции, потому что эти свойства уже есть у предка. Нам надо только описать их и всё. Я описал только маленькую часть из доступных у *TGraphicControl* функций. Ты можешь добавить любые из доступных. Чтобы узнать, какие функции можно добавлять, открой помощь (меню *Help->Delphi Help*) и найди там объект *TGraphicControl*. Щёлкни по нему дважды и в появившейся справке выбери пункт *Properties* (вверху окна). Появится окно с перечнем всех свойств. Ты можешь добавить любое из них. Например, чтобы добавить свойство Action нужно написать в разделе published:

```
published
 property Action;
```

Чтобы добавить новое свойство, которое не существует у предков, нужно немного попотеть. Например. Добавим возможность, чтобы пользователь мог менять картинку фона. Для этого описываем в разделе **published** свойство *BGBitmap*:

```
property BGBitmap:TBitmap read FBGBitmap write SetBGBitmap;
```

Подобную строку ты видел при описания события, только там свойство имело тип события, а здесь это картинка типа *TBatmap*, так что Delphi воспримет эту запись, как свойство.

Для чтения свойства *BGBitmap* я поставил переменную *FBGBitmap*. Это тоже картинка, и когда мы будем обращаться к свойству с целью прочитать картинку, то она будет просто копироваться из переменной *FBGBitmap*.

Для записи используется процедура SetBGBitmap. В принципе, можно было и для чтения написать функцию, но это не имеет смысла. А вот для записи сложных переменных (для которых выделяется память) лучше использовать функции. Для простых переменных (числа, булевы переменные), писать процедуры не надо, но если ты напишешь, то это ошибкой не будет.

Итак, для записи свойства *BGBitmap* я написал следующую процедуру:

```
procedure TGraphicClock.SetBGBitmap(Value: TBitmap);
begin
 FBGBitmap.Assign(Value);
 invalidate;
end;
```

В первой строке я просто копирую в переменную *FBGBitmap* переданную в качестве параметра картинку. Во второй строке я заставляю наш компонент прорисоваться.

Теперь ты можешь изменять фон простой операцией *GraphicClock1.BGBitmap:=bitmap*.

Если ты хочешь создать свойство с выпадающим списком (как например у свойства *Align*), по щелчку которого выпадает список возможных параметров, то тут уже немного

сложнее. В моих часах есть такой параметр, который делает выбор, какого типа будут часы - аналоговые или цифровые. Объявление делается так:

```
property ClockStyle:TClockStyle read FClockStyle write SetStyleStyle default scAnalog;
```

Мы объявляем свойство *ClockStyle* типа *TClockStyle*. Тип *TClockStyle* мы должны описать в самом начале, до описания нашего объекта *TGraphicClock*, в разделе **type**:

```
type
  TClockStyle = (scAnalog, scDigital);

  TGraphicClock = class(TGraphicControl)
  private
    Ticker: TTimer;
```

Строка *TClockStyle = (scAnalog, scDigital)* - объявляет список переменных, которые и будут выпадать по выбору свойства.

Всё остальное происходит так же, за исключением нового слова *default*, которое устанавливает значение по умолчанию для данного свойство - *scAnalog*.

Остальной код я приводить не буду, потому что там сплошная математика, которой достаточно много, но тебе советую с ним разобраться.

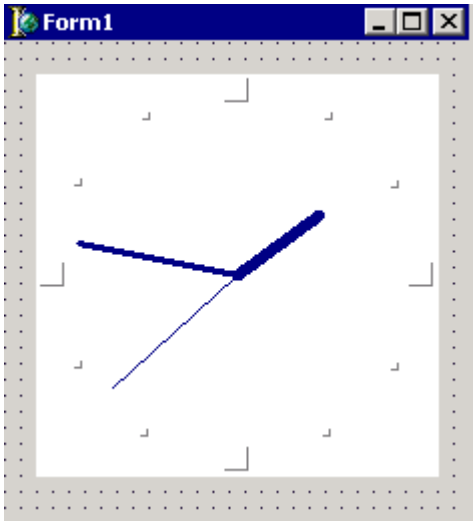


Рис 19.3.2. Пример наших часов.

*На компакт диске, в директории **\Примеры\Глава 19\Component** ты можешь увидеть пример этой программы.*

19.4 Создание иконки компонента.

Если ты установил созданный нами компонент в Delphi, то заметил, что он имеет абсолютно некрасивую картинку. Эта иконка выбирается по умолчанию для всех компонентов, если у них нет своей. У нас пока нет своей иконки и её сейчас предстоит создать.

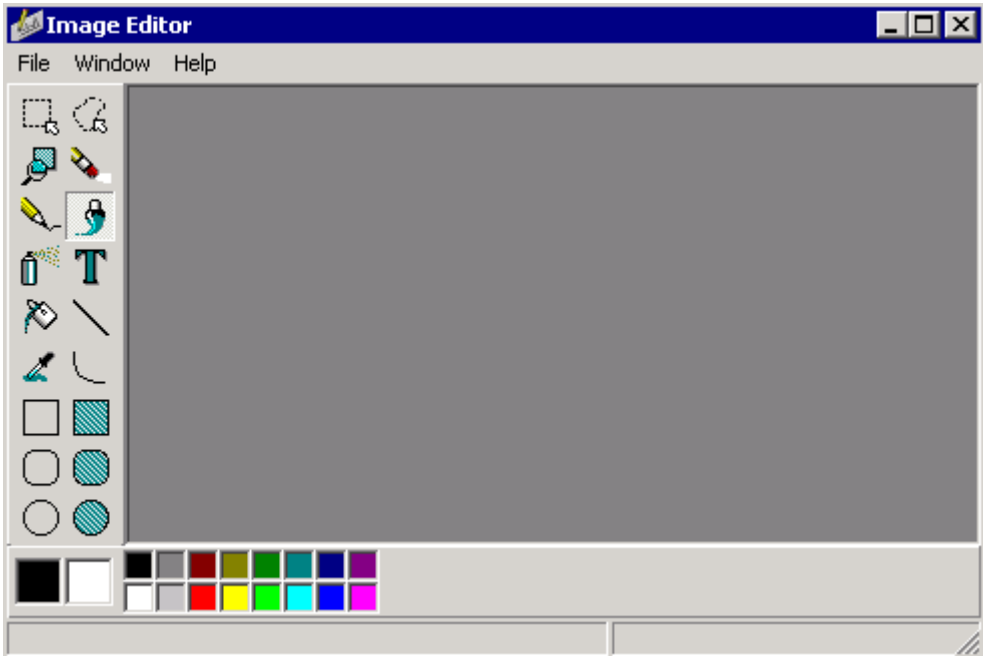


Рис 19.4.1. Пример наших часов.

Для создания иконки будем пользоваться программой Image Editor, которая входит в поставку Delphi. Её главное окно ты можешь увидеть на рисунке 19.4.1. Здесь нужно выбрать из меню *File* пункт *Newi* и затем *Component Resource File (.dcr)* (файл ресурсов для компонентов). Программа создаст новое окно, содержащее дерево, в котором пока только один элемент *Contents*.

Для создания нового элемента, нужно щёлкнуть правой кнопкой мыши в созданном окне проекта ресурса и в появившемся меню выбрать пункт *New->Bitmap*. Перед тобой откроется окно свойств создаваемой картинки (рисунок 19.4.2).

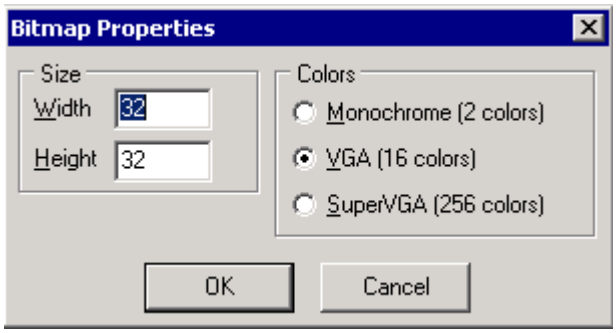


Рис 19.4.2. Пример наших часов.

Ширина и высота (параметры *Width* и *Height*) картинки должны быть равны 24. Режим оставляем VGA, потому что для иконки 16 цветов достаточно. Жми «ОК». Теперь у тебя в дереве элементов ресурсов появился раздел *Bitmap* и в нём наша картинка *Bitmap1* (рисунок 19.4.3). Щёлкни по элементу *Bitmap1* и в появившемся меню выбери пункт *Rename*. Переименуй нашу картинку, дав ей имя нашего компонента *TGraphicClock*. Картинка обязательно должна иметь то же имя, что и компонент, которому она предназначена, потому что в одном файле исходника может быть несколько компонентов и по имени картинки Delphi будет определять, к какому компоненту она относиться.

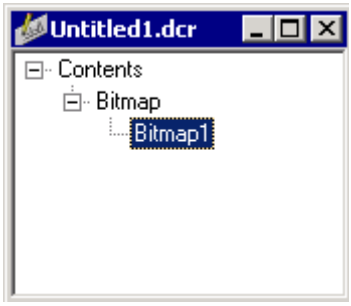


Рис 19.4.2. Пример наших часов.

Теперь дважды щёлкни по элементу картинки и появиться графический редактор, в котором ты можешь нарисовать что угодно. Нарисуй какие-нибудь часы. После этого, файл ресурсов можно закрывать. На вопрос о сохранении файла, сохрани его под именем *GraphicClock.dcr*. Скопируй этот файл в директорию, где у тебя находиться исходник компонента, оба файла должны находиться в одной директории.

Теперь открывай в Delphi наш пакет othercomponents.dpk. Удали из него файл исходника часов и откомпилируй пакет. Это заставит Delphi удалить из оболочки наш компонент. После этого снова добавь файл исходника и откомпилируй пакет ещё раз. Теперь компонент установился обратно, но уже с нашей иконкой.

На компакт диске, в директории ***\Примеры\Глава 19\IconForComponent*** *ты можешь увидеть пример моей иконки.*

Глава 22. OLE, COM, ActiveX

Технология OLE – это сильнейшее изобретение Microsoft, с её же точки зрения. С моей точки зрения – это правда, но только на 10 процентов. Технология OLE включает в себя элементы ActiveX, которые чем-то иногда похожи на компоненты Delphi, но только требуют неоправданных проблем с регистрацией в системе и вечные мучения с контролем версий.

Я не люблю ActiveX и стараюсь использовать их только в крайнем случае. Я даже не хотел тебе рассказывать о них, потому что не хочу тебя загружать ерундой. Но всё же, я иногда сталкиваюсь с ситуацией, когда ActiveX приходиться использовать (иногда использую чужие, а иногда приходиться писать свои компоненты). Поэтому я долго сопротивлялся, но решил всё же рассказать про эту технологию.

Единственное преимущество ActiveX – компоненты, оформленные в таком виде будут работать в любом другом языке программирования. Если компоненты Delphi можно использовать только в Delphi, ну в крайнем случае в C++ Builder или Kylix (обе разработки фирмы Borland), то ActiveX однозначно работают везде. Твои компоненты смогут использовать и в Visual C++, и в Visual Basic и во многих других программах, поддерживающих ActiveX.

И всё же, несмотря на это я всегда пишу VCL компоненты, а если надо их превратить в ActiveX, то это можно сделать с помощью Delphi за пять минут.

22.1. Теория OLE

Технология OLE (Object Link and Embedding) – является стандартом Windows обеспечивает связывание и встраивание объектов на основе технологии COM (Component Object Model). С момента выпуска первой версии OLE 1.0, технология претерпела большие изменение и старая аббревиатура OLE уже не отражает действительности. Теперь эти три буквы используют только по привычке и давно уже пора всё это дело как-нибудь переименовать.

COM – это спецификация, созданная для описания структуры COM-объектов. Как я уже сказал, COM-объекты могут использоваться в любых языках программирования, вне зависимости от того, на каком языке они написаны. Помимо компиляторов, с COM объектами могут работать очень много крупных программных пакетов, например всё те же Word и Excel.

Объекты OLE могут запускаться как в отдельном окне, так и внутри окна вашего приложения (первая версия позволяла запускать объекты только в отдельном окне). Таким образом, ты можешь получать доступ к чужим приложениям и использовать их в своих целях. В главе 15, когда я описывал пример отчётности в Excel, мы уже познакомились с технологией OLE, сами этого не подозревая. Тогда я запускал программу Excel с помощью функции *CreateOleObject* и потом получал к ней доступ. Мы выводили отчёт в чужую программу.

Когда OLE объект запускается внутри твоего окна, то часть твоих меню и панелей заменяется на те, что используются в программе, которую мы загружаем.

Прежде чем говорить о чём-то дальше, давай посмотрим на работу с OLE на маленьком примере. Запусти Delphi и создадим новый проект. Брось на форму один лишь компонент *OleContainer* с закладки *System* палитры компонентов. Теперь дважды щёлкни внутри окна компонента (или щёлкни правой кнопкой и в появившемся меню выбери пункт *Insert Object*) и перед тобой откроется окно, как на рисунке 22.1.1.

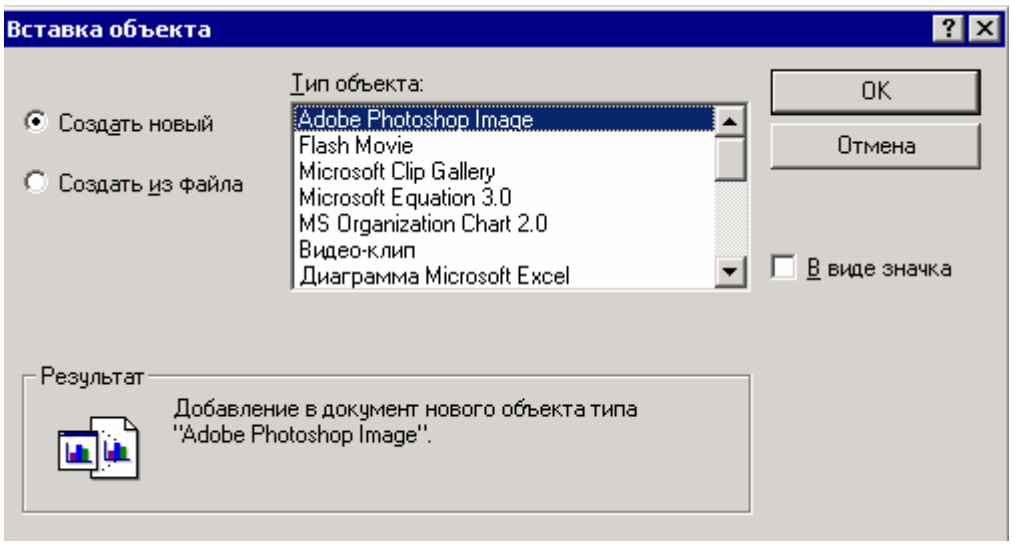


Рисунок 22.1.1. Выбор объекта OLE.

В этом окне, в списке «*Тип объекта*» найди строку «*Рисунок PaintBrush*». Выдели эту строку. Если поставить галочку «*В виде значка*», то на форме ты потом увидишь только значок объекта, а по двойному щелчку будет запускаться выбранное приложение (*PaintBrush*). Если галочки нет, то приложение встраивается прямо в окно.

Попробуй запустить программу и дважды щёлкнуть внутри компонента *OLEContainer*. Компонент будет взят в рамочку и ты сможешь рисовать в нём как в самом PainBrush.

Вот таким нехитрым способом мы встроили стороннее приложение в свою программу. Но у него нет ни меню, ни панелей. Для добавления меню, достаточно просить на форму компонент *MainMenu*. Туда не надо добавлять никаких пунктов, достаточно просто его бросить. Если у формы есть главное меню, то OLE компонент будет автоматически встраиваться в него. Запусти программу и убедись, что меню появляется.

Теперь измени свойство *Align* у компонента *OleContainer1* на *alClient*, чтобы растянуть его по всей форме. Запусти программу и посмотри на результат. Теперь твоё приложение превратилось чуть ли не в полноценный PaintBrush. Единственное, что отличает его – заголовок окна и иконка Delphi.

Рисунок 22.1.2. Результат нашей программы.

На компакт диске, в директории ***\Примеры\Глава 22\OLE*** *ты можешь увидеть пример этой программы.*

Теперь подробней о меню. Попробуй создать в нашем главном меню два пункта *Файл* и *Правка*. У обоих пунктов можешь добавить подпункты (рисунок 22.1.3). Теперь запусти программу и посмотри на результат. Когда ты пытаешься активизировать OLE объект, то меню *Файл* остаётся твоё, а аналогичное из программы PaintBrush исчезает. К тому же у нас получилось два пункта *Правка*. В чём же эффект? Пункты меню, у которых в свойстве *GroupIndex* стоит чётное значение (0, 2, 4 …) – остаются неизменными. Пункты меню, у которых *GroupIndex* нечётное – заменяются аналогичными из OLE объекта.

Попробуй поставить у пункта *Правка* в свойстве *GroupIndex* значение 1 и запусти программу. Теперь при активизации OLE объекта пункт меню «*Правка*» будет заменён аналогичным из объекта. Обязательно учитывай этот эффект при программировании программ работающих с OLE!!!

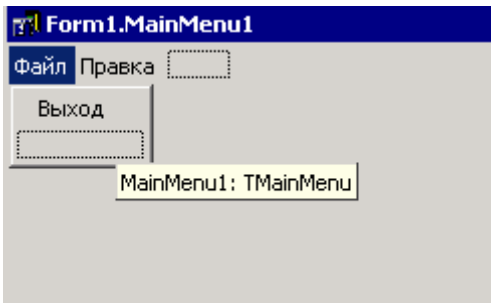


Рисунок 22.1.3. Главное меню.

Написанная нами программа называется *Контейнер OLE*, а программа, которую мы привязываем называется *Объект OLE*.

При работе с OLE выделяются два способа связи:

1. Связывание объекта OLE с контейнером. В этом случае результирующий файл сохраняется отдельно и должен быть создан до того, как контейнер обратиться к нему. Контейнер только ссылается на файл, но не хранит в себе никаких данных. Главное преимущество такого способа – к одному документу может ссылаться множество контейнеров и при изменении документа, все контейнеры получают эти изменения.

2. Встраивание объектов. В этом случае созданный документ храниться в контейнере и другие приложения не могут получить к нему доступ. В этом случае данные хранятся как часть приложения.

22.2. OLE Контейнер

Мы уже написали небольшой пример работы с OLE контейнером (*OleContainer*), но мы задействовали только малую часть его возможностей. Здесь нам предстоит познакомиться с основными свойствами и методами компонента и написать более полезный пример, чем в предыдущей части книги.

Давай, как всегда, начнём с рассмотрения свойств компонента *OleContainer*, которые могут понадобиться при работе с OLE объектами.

AllowInPlace – если это свойство равно *true*, то OLE объект будет создаваться в компоненте, иначе будет запускаться как отдельное приложение.

AutoActivate – определяет способ активизации OLE объекта. Здесь возможны следующие значения:

aaDoubleClick – активизация объекта будет происходить по двойному щелчку.

aaGetFocus – активизация при получении фокуса.

aaManual – активизация вызовом соответствующей функции.

CopyOnSave – если это свойство равно *true* то при попытке сохранения создаётся временный файл, который сжимается для экономии места на диске.

Iconic – если это свойство равно *true*, то в окне контейнера будет отображаться иконка объекта, иначе сам объект.

Linked – если объект связанный, то здесь *true*.

OleClassName – здесь храниться имя OLE объекта.

OleObject – здесь храниться ссылка на сам OLE объект.

OleObjectInterface – здесь храниться ссылка на интерфейс OLE объекта.

Modified – если объект изменён, то это свойство принимает значение *true*.

NewInserted – если объект заново создан командой *Insert Object*, то это свойство равно *true*.

OleStreamFormat – если это свойство равно *true*, то при сохранении будет использоваться старый формат OLE 1.0. Это необходимо, если какое-то программное обеспечение не умеет работать с новым форматом.

SizeMode – управляет размером объекта.

smAutoSize – размер выбирается автоматически.

smCenter – по центру.

smClip – объект показывается реальным размером, отображается та часть, которая поместилась в окно.

smScale – объект масштабируется.

smStretch – объект растягивается.

Теперь посмотрим на основные методы компонента *OleContainer*:

ChangeIconDialog – показать окно смены иконки.

Close – закрыть OLE объект.

Copy – копировать объект в буфер обмена.

CreateLinkToFile – создать ссылку на файл OLE объекта.

CreateObject – создать в контейнере OLE объект. Тут два параметра – имя объекта и булево значение, указывающее на необходимость создания объекта в виде иконки

CreateObjectFromFile создать объект из указанного файла. Тут два параметра – имя файла и булево значение, указывающее на необходимость создания объекта в виде иконки.

DoVerb – передать объекту OLE запрос на выполнение каких-либо действий.

InsertObjectDialog – показать окно вставки нового объекта.

LoadFromFile – загрузить объект из файла. В качестве единственного параметра нужно указать имя файла.

LoadFromStream – загрузить из потока. В качестве единственного параметра нужно указать поток.

ObjectPropertiesDialog – показать окно свойств объекта.

Paste – вставить из буфера обмена.

PasteSpecialDialog – показать специальное окно вставки из буфера обмена.

Run – запустить объект.

SaveAsDocument – сохранить объект в виде OLE документа. В качестве единственного параметра нужно указать имя файла.

SaveToStream – сохранить в поток. В качестве единственного параметра нужно указать поток.

С учётом всего сказанного, давай попробуем улучшить наш пример, написанный в прошлой части. Открой его и добавь одну панель *TToolBar* и брось на неё пять кнопок:

1. Вставить объект.
2. Свойства объекта.
3. Вставить из файла
4. Открыть объект.
5. Сохранить объект.
6. Закрыть объект.

Мою форму ты можешь увидеть на рисунке 22.2.1. Ты можешь создать точно такую же форму, а можешь её реализовать по своему, это зависит от твоих пристрастий.

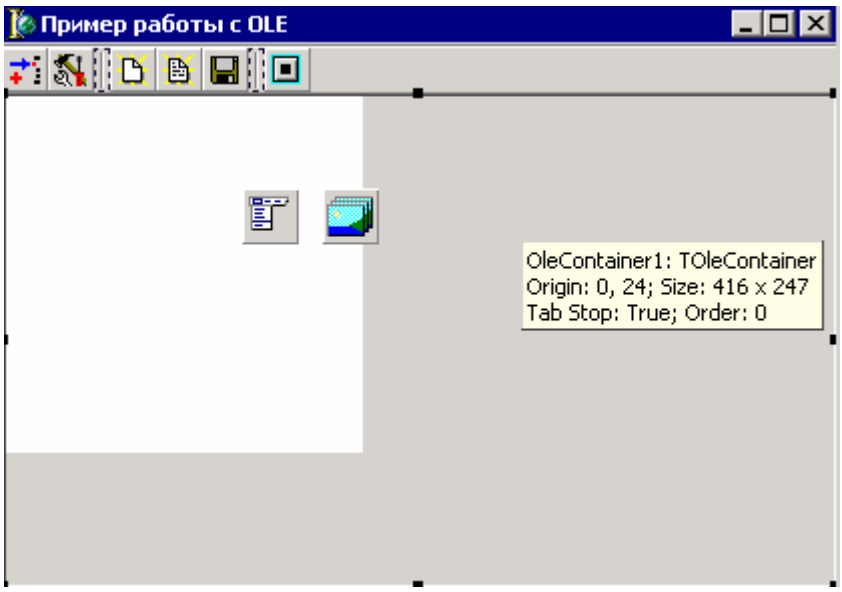


Рисунок 22.2.1. Форма. будущей программы

По нажатию кнопки «Вставить объект» пишем следующий код:

```
procedure TForm1.InsertButtonClick(Sender: TObject);
begin
 OleContainer1.InsertObjectDialog;
end;
```

Теперь, после нажатия этой кнопки, программа будет отображать уже знакомое тебе окно (рисунок 22.1.1), с помощью которого можно будет сменить OLE объект на другой.

По нажатию второй кнопки (свойства объекта) пишем следующий код:

```
procedure TForm1.PropertiesButtonClick(Sender: TObject);
begin
 OleContainer1.ObjectPropertiesDialog;
end;
```

Этим кодом мы будем отображать окно свойств объекта. Пример такого окна ты можешь увидеть на рисунке 22.2.2. В принципе, тут не так уж и много параметров, которые ты можешь изменить. На закладке «*Просмотр*» можно изменить только масштаб (если объект поддерживает масштабирование) или изменить значок.

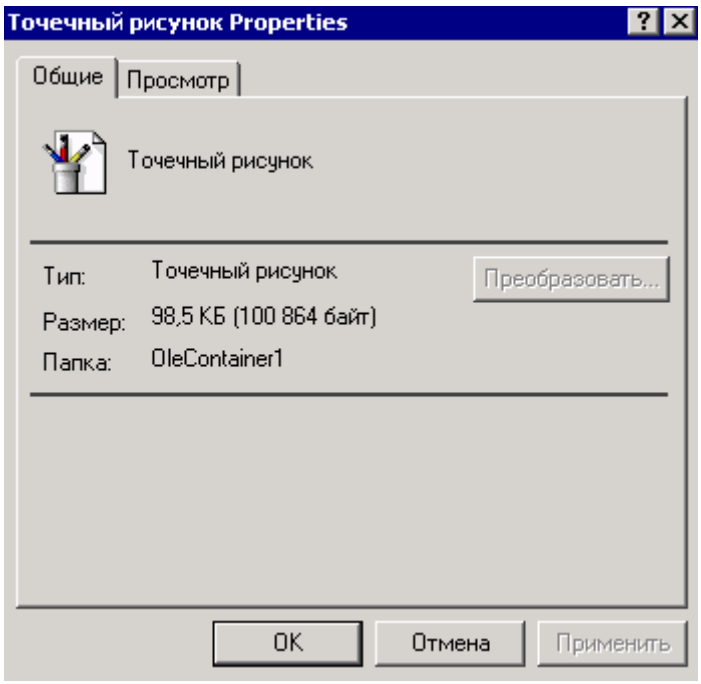


Рисунок 22.2.2. Форма будущей программы

Следующая у нас идёт кнопка «*Вставить из файла*». По нажатию этой кнопки пишем следующий код:

```
procedure TForm1.InsertFromFileToolButtonClick(Sender: TObject);
begin
 if OpenDialog1.Execute then
  OleContainer1.CreateObjectFromFile(OpenDialog1.FileName, false);
end;
```

Вот здесь я добавил окно открытия файла *OpenDialog*. В первой строке я показываю это окно и если пользователь выбрал файл, то во второй строке я создаю объект на основе выбранного файла. Попробуй запустить приложение и открыть bmp, doc или xls файл. Как только ты нажмёшь кнопку «*Открыть*», так сразу перед тобой появиться соответствующий выбранному файлу OLE объект. Если ты выберешь doc файл, то запуститься Word.

По нажатию кнопки сохранить мы пишем следующее:

```
procedure TForm1.OpenButtonClick(Sender: TObject);
begin
 OleContainer1.LoadFromFile('ole.dat');
end;
```

Здесь я открываю объект из указанного файла. В качестве имени файла я использую ‘*ole.dat*’. Ты же можешь добавить на форму компонент *OpenDialog*, чтобы пользователь сам мог выбирать имя файла, который надо открывать.

По кнопке «Открыть файл» пишем следующий код:

```
procedure TForm1.SaveButtonClick(Sender: TObject);
begin
 OleContainer1.SaveToFile('ole.dat');
end;
```

Здесь мы сохраняем объект в тот же файл ‘*ole.dat*’. Здесь тоже можно дать возможность пользователю выбирать имя файла с помощью компонента *OpenDialog*. Только хочу тебя предупредить, что файл созданный OLE объектом не совместим с форматом самой программы объекта. Это значит, что *ole.dat* – это не bmp картинка, хотя и bmp – это родной формат для программы PaintBrush. Так что ты не сможешь открыть файл *ole.dat* с помощью PaintBrush.

По нажатию кнопки закрыть пишем следующее:

```
procedure TForm1.CloseButtonClick(Sender: TObject);
begin
 OleContainer1.Close;
end;
```

Здесь мы закрываем запущенный OLE объект. Если пользователь дважды щёлкнул по контейнеру, то OLE объект запускается и закрыть его можно с помощью этой кнопки.

На компакт диске, в директории ***\Примеры\Глава 22\OLE1*** *ты можешь увидеть пример этой программы.*

22.3. Создание собственного окна вставки OLE объекта

Использование стандартного окна – это хорошо, но можно же написать и своё окошко. Тем более, что это не так уж и сложно, заодно и потренируемся в программировании и вспомним, как работать с реестром. Если ты пока не думаешь, создавать собственные окна, прочтение этой части является обязательным, потому что здесь будет показаны некоторые приёмы работы с реестром, о которых я не говорил.

Откроем предыдущий пример и создадим в нём новую форму. На ней обязательно должен присутствовать компонент *TListBox* с именем *OLEItemsListBox* и две кнопки (*Да* и *Отмена*). Мою форму ты можешь увидеть на рисунке 22.3.1.

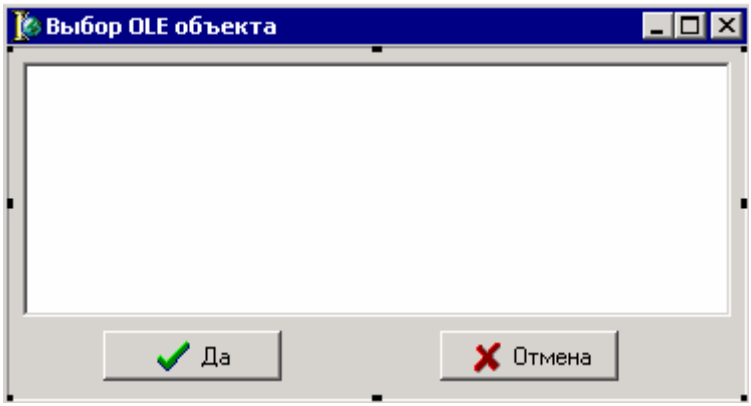


Рисунок 22.3.1. Форма окна загрузки OLE объектов.

В разделе **public** объекта формы нужно объявить одну переменную *ProgramsID* имеющую тип *TStrings*:

```
public
  { Public declarations }
  ProgramsID:TStrings;
```

Почему я объявляю именно в разделе **public**? В этом параметре будет храниться список найденных идентификаторов, по которым мы потом будем создавать OLE объект. К переменной *ProgramsID* нам придется обращаться из основной формы, поэтому переменная должны быть доступна и находиться в разделе **public**.

Эту переменную нужно проинициализировать по событию *OnCreate* формы:

```
procedure TInsertOLEForm.FormCreate(Sender: TObject);
begin
 ProgramsID:=TStringList.Create;
end;
```

Уничтожаться переменная будет по событию *OnDestroy*:

```
procedure TInsertOLEForm.FormDestroy(Sender: TObject);
begin
 ProgramsID.Free;
end;
```

Теперь создаём обработчик события *OnShow*, где будет происходить загрузка из реестра списка доступных в системе OLE объектов:

```
procedure TInsertOLEForm.FormShow(Sender: TObject);
var
 i:Integer;
 CLSID:String;
 Keys:TStrings;
 reg:TRegistry;
begin
 Keys:=TStringList.Create;
 ProgramsID.Clear;
 OLEItemsListBox.Items.Clear;

 reg:=TRegistry.Create;
 reg.RootKey:=HKEY_CLASSES_ROOT;
 reg.OpenKey('\', false);
 reg.GetKeyNames(Keys);

 for i:=0 to Keys.Count-1 do
  begin
   reg.CloseKey;
```

```
  reg.OpenKey(Keys.Strings[i], false);
  if not reg.KeyExists('Insertable') then continue;
  OLEItemsListBox.Items.Add(reg.ReadString(''));
  reg.OpenKey('CLSID', false);
  CLSID:=reg.ReadString('');
  reg.CloseKey;
  if reg.OpenKey('CLSID\'+CLSID, false) then
   begin
    if reg.OpenKey('ProgId', false) then
     ProgramsID.Add(reg.ReadString(''))
    else
     ProgramsID.Add('[Нет идентификатора программы]')
   end;
 end;
 Keys.Free;
end;
```

В первой строке я инициализирую переменную *Keys*, которая имеет тип *TStrings*. Это у нас будет список строк, в котором будут храниться все ключи раздела реестра, с описанием OLE объектов.

Во второй строке я очищаю список из переменной *ProgramsID*, потому что там может что-то остаться после предыдущего вызова окна. В следующей строке очищаю компонент *TListBox*.

Всё, приготовление окончено. Теперь нужно проинициализировать переменную reg, т.е. открыть реестр. Как ты помнишь, реестр открывается на разделе *HKEY_CURRENT_USER*, а информация об установленных OLE объектах находиться в разделе *HKEY_CLASSES_ROOT*. Именно поэтому я тут же меняю имя раздела, чтобы перейти в нужный.

В следующей строке я открываю подраздел ‘\’. В качестве второго параметра метода открытия раздела (*OpenKey*) стоит значение *false*, что говорит о том, что если нужный раздел не существует, то его не надо создавать. В принципе, такой ситуации не может быть, потому что даже в полностью минимальной установке Windows этот раздел существовать будет. Поэтому в данном случае, второй параметр не очень важен.

В следующей строке я получаю список подразделов (с помощью метода *GetKeyNames*) текущего раздела. Результат будет записан в переменную *Keys*. Теперь можно запускать цикл и проверять все найденные ключи на соответствие подключаемым OLE объектам.

Внутри цикла я сначала закрываю текущий раздел и после этого открываю очередной раздел из списка *Keys*. В открытом разделе я проверяю наличие ключа «*Insertable*». Если его нет, то вызывается *continue*, чтобы прервать цикл и перейти на следующий элемент. Если такой ключ есть, то я читаю тип текущего OLE объекта и добавляю его в список *ListBox*:

```
OLEItemsListBox.Items.Add(reg.ReadString(''));
```

Далее мне надо прочитать идентификатор объекта, по которому потом мы будем создавать выбранный объект. Для этого я открываю раздел *CLSID*, в котором храниться уникальный номер объекта и читаю этот номер.

После прочтения *CLSID* я закрываю текущий ключ и открываю раздел с именем ключа. Если он открыт, то происходит попытка чтения идентификатора. Если он прочитан, то добавляем идентификатор в список *ProgramsID*, иначе добавляем строку *[Нет идентификатора программы]*. В принципе, при нормальном раскладе всё должно быть хорошо. Единственная ситуация, когда идентификатор не будет найден – когда OLE объект установлен или удалён из системы неправильно.

После перебора всех строк списка я уничтожаю переменную *Keys*.

Теперь переходим к основному окну и здесь по нажатию кнопки *Вставить объект* пишем следующий код:

```
procedure TForm1.InsertButtonClick(Sender: TObject);
begin
 InsertOLEForm.ShowModal;
 if InsertOLEForm.ModalResult=mrOK then
  OleContainer1.CreateObject(InsertOLEForm.ProgramsID.Strings[
      InsertOLEForm.OLEItemsListBox.ItemIndex], false);
end;
```

В первой строке я показываю созданное нами окно. Если пользователь нажал *OK* (свойство *ModalResult* равно *mrOK*), то выполнить следующую строку, в которой создаётся OLE объект с помощью вызова метода *CreateObject* контейнера. В качестве параметра я указываю идентификатор из списка *ProgramsID* выделенной в списке строки.

InsertOLEForm.OLEItemsListBox.ItemIndex – указывает на выделенную строку в списке.
InsertOLEForm.ProgramsID.Strings[строка] – здесь храниться идентификатор.

На рисунке 22.3.2 ты можешь увидеть моё окно, с результатом работы.

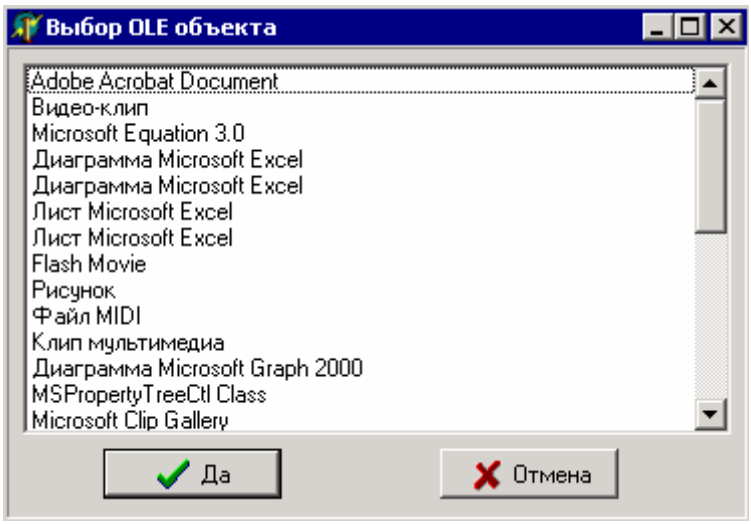


Рисунок 22.3.2. Результат работы.

В принципе, пример рабочий. Единственный его недостаток – нет проверки на ошибки. Возможны ситуации, когда в системе есть объекты с запорченными записями (неправильная установка или удаление записей), в этом случае, при создании программа может зависнуть. Самый простой способ – поставить вызов функции *CreateObject* в блоке *try..except..end*, для исключения подобных ошибок. Второе – у нас нет проверки на выделение элемента. Возможно, что пользователь увидеть окно и нажмёт «*OK*», а при этом ни одна строка списка не будет выделена. Так что перед загрузкой желательно проверять *InsertOLEForm.OLEItemsListBox.ItemIndex* на правильное значение. Если свойство *ItemIndex* больше или равно нулю, то одна из строк выделена, если равно –1, то выделенных строк в списке нет, а пользователь нажал *OK*.

Я намеренно не стал делать полноценный исходник, чтобы ты сам мог его доработать. Я ставлю задачу в этой книги показать тебе основы и научить правильно мыслить, а дальше ты мыслить должен сам.

На компакт диске, в директории ***\Примеры\Глава 22\InsertDialog*** *ты можешь увидеть пример этой программы.*

22.4. Элементы управления ActiveX

Теперь разберёмся с элементами управления ActiveX. Я уже достаточно нехорошего сказал о них в теории, теперь пора познакомиться с компонентами ActiveX на практике. Хотя я уже говорил, что их не особо люблю, но иногда использовать приходиться. Единственный компонент, который я использую регулярно – Internet Explorer. Да, это тоже компонент ActiveX, который использует Windows, хотя ты и видишь его в виде программы. Именно потому что этот компонент встроен в Windows и присутствует на всех машинах, я использую его достаточно часто.

Запускай Delphi. Перейди на закладку *Internet* палитры компонентов. Здесь должен быть компонент *WebBrowser* (он должен быть последний). Если у тебя версия Delphi меньше, чем пятая, то этого компонента может и не быть. Он может отсутствовать и если ты отказался устанавливать интернет компоненты (по умолчанию они ставятся).

Если компонента *WebBrowser* нет, то его надо установить (но даже если есть, читай всё подряд, в будущем пригодиться). Выбери *Import ActiveX Control* из меню *Component*. Перед тобой должно открыться окно, как на рисунке 22.4.1.

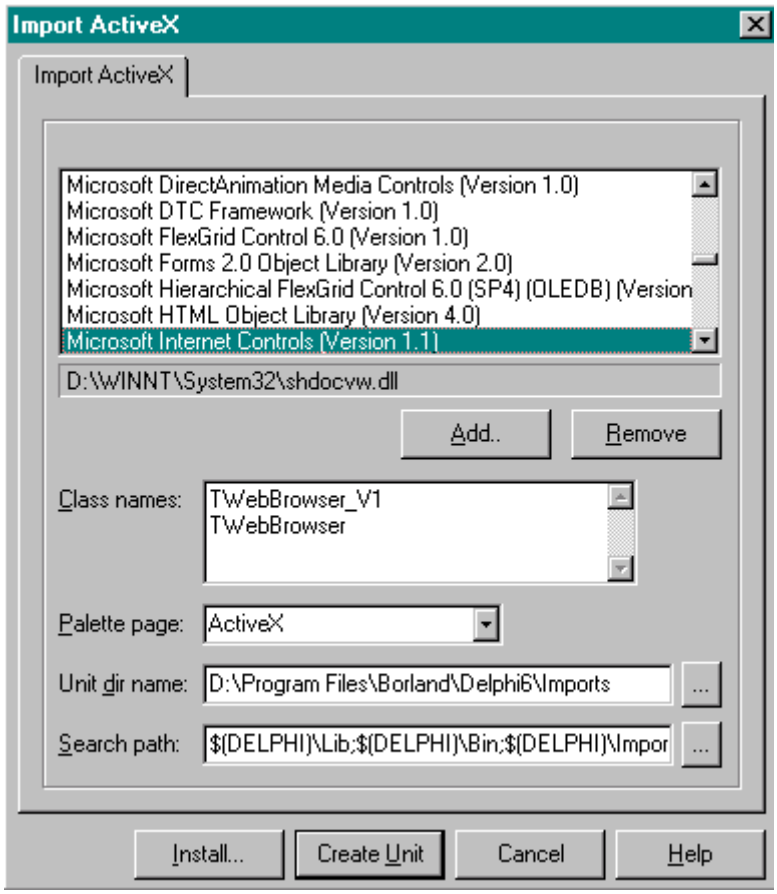


Рисунок 22.4.1 Окно установки ActiveX компонента

В списке выбора этого окна (сверху) найди строку *Microsoft Internet Controls (Version 1.1)*. Версия может отличаться, но это не важно. Теперь нажми кнопку *Install*. Можно

было бы нажать кнопку *Create Unit*, чтобы создать модуль с описанием ActiveX, но это излишне, потому что по нажатию кнопки *Install*, этот модуль создастся автоматически. Нажимай, и перед тобой откроется окно выбора пакета, как на рисунке 22.4.2.

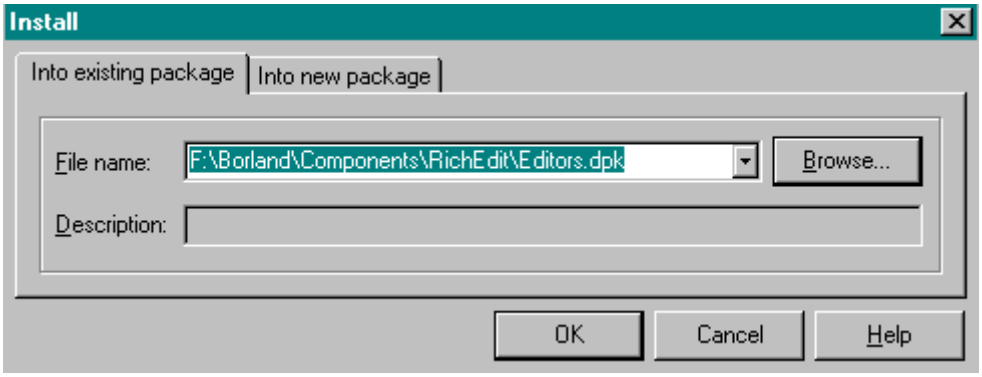


Рисунок 22.4.2 Окно выбора пакета

Окно выбора пакета очень похоже на то, которое мы видели при установки компонентов VCL. Только здесь ты не выбираешь имя файла (он будет создан из компонента ActiveX, а можешь только выбрать пакет, в который будет происходить установка. На закладке *Into new package* ты можешь создать новый пакет точно так же, как и при создании пакетов для VCL компонентов. Точнее сказать, что пакеты одни и те же, разницы никакой. В одном пакете могут храниться VCL и ActiveX компоненты.

Выбери пакет и нажми *OK*. После этого появиться запрос на компиляцию пакета как на рисунке 22.4.3. Соглашайся. Delphi откомпилирует необходимые файлы и установит компонент для работы с браузером.

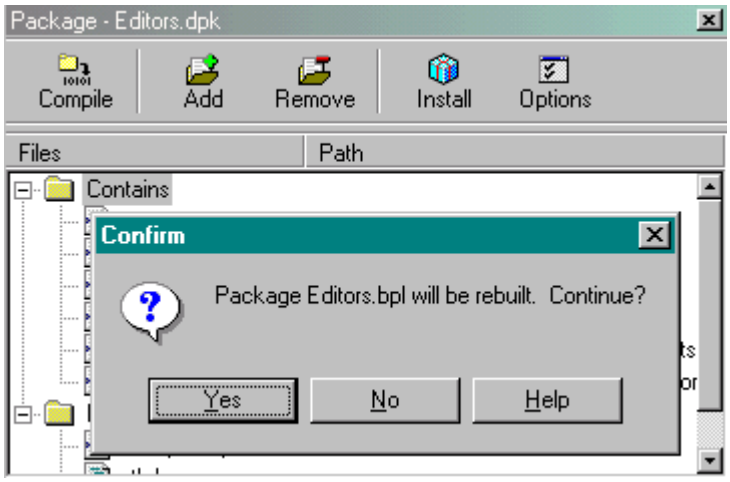


Рисунок 22.4.3 Запрос компиляции пакета

После того, как Delphi прошуршит мозгами, появится окно, которое сообщит об успешной установке нового компонента. Нажми "*OK*" и закрой все, что открыл Delphi. Для этого выбери *Close All* из меню *File*. Теперь и у тебя есть компонент *WebBrowser*, только он расположен на странице *ActiveX* палитры компонентов.

Как ты мог заметить, мы будем использовать *Microsoft Internet Controls*, т.е. движок установленного на твоем компьютере IE. А это значит, что твой браузер подхватит все болезни и глюки своего движка. Единственное, что может успокоить - так это то, что интерфейс не будет таким занудным. Он будет таким, как ты захочешь, потому что сделан твоими руками, а своё всегда приятнее.

Сейчас ты уже готов приступить к программированию. Создай новый проект и сразу измени заголовок и иконку.

Двигаемся дальше. Установи на форму наш компонент *WebBrowser* (он находится на закладке *Internet* или *ActiveX* палитры компонентов) - у тебя появится белый квадрат с именем *WebBrowser1*. После этого брось на форму *CoolBar*, который находится на закладке *Win32* палитры компонентов. Это панелька, которая должна выровняться по верхнему краю на твоей форме. Теперь выдели *WebBrowser1* и перейди в объектный инспектор. Щелкни по свойству *Align* и в выпадающем списке выбери *alClient*. Компонент *WebBrowser* должен растянуться на все свободное место формы. В результате ты увидишь нечто похожее на рисунок 22.4.4.

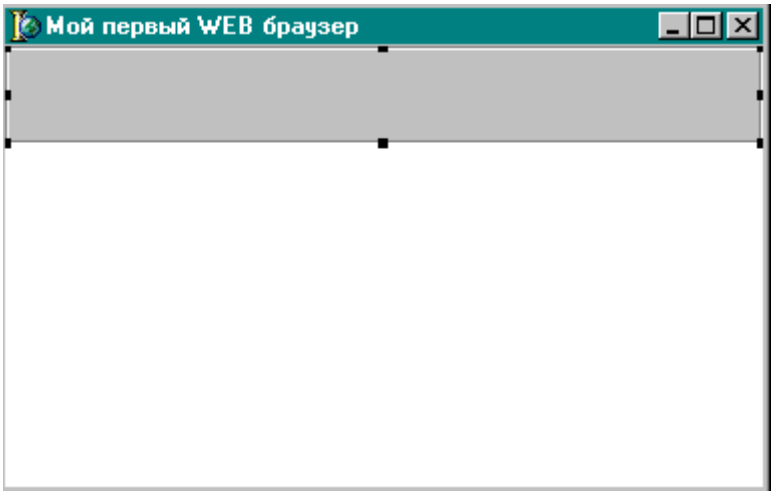


Рисунок 22.4.4. Форма будущей программы

Теперь брось на *CoolBar1* (мы его недавно установили на форму) панель *ToolBar* из закладки *Win32* и *ComboBox* из закладки *Standard* палитры компонентов. Все это ты должен бросить именно внутрь *CoolBar1*, иначе ты получишь некрасивый набор компонентов. После этого нужно выделить *CoolBar1* и перейти в объектный инспектор. Здесь ты должен изменить строку *AutoSize* на *true* (по умолчанию она *false*).

Если что-то не получилось, то читай главу заново. Если все в порядке, то выделяй *ComboBox1* (выпадающий список) и переходи в объектный инспектор. Здесь ты должен выделить закладку *Events* и произвести сложнейшее действие двойного щелчка по строке *OnKeyDown*, чтобы создать обработчик этого события. Как и раньше, Delphi создаст процедуру обработчика. Она будет вызываться каждый раз, когда ты будешь вводить какую-нибудь букву в *ComboBox*. Здесь ты должен написать следующее:

```
procedure TForm1.ComboBox1KeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin
 if Key= VK_RETURN then
  WebBrowser1.Navigate(ComboBox1.Text);
end;
```

Здесь я проверяю, если переменная *Key* равна *VK_RETURN* (т.е. если нажата кнопка *Enter*), то выполнить следующее действие - *WebBrowser1.Navigate(ComboBox1.Text)*. Здесь я вызываю метод *Navigate* нашего браузера и в качестве параметра указываю текст компонента *ComboBox*. Метод *Navigate* заставляет открыть браузер указанную страничку.

Если ты устанавливал компонент *WebBrowser* как ActiveX и он не стоял у тебя сразу же, то возможно, что при компиляции Delphi будет ругаться на недостаточность параметров. Он может запросить аж три дополнительных параметра типа *OleVariant*. В этом случае объяви три переменные такого типа и просто подставь их:

```
procedure TForm1.ComboBox1KeyDown(Sender: TObject; var Key: Word;
var
 p1,p2,p3:OleVariant;
begin
 if Key= VK_RETURN then
 WebBrowser1.Navigate(ComboBox1.Text, p1, p2, p3);
end;
```

Нажми "F9", и твоя прога должна засвистеть. Введи какой-нибудь адрес в строку *ComboBox* и нажми *Enter*. Если ты правильно ввел адрес, то в *WebBrowser1* через несколько минут должен появиться указанный сайт.

Кликни по ToolBar1 и снова переходи в ObjectInspector. Здесь нужно изменить свойства *AutoSize*, *ShowCaption* и *Flat* на *true* (все они по умолчанию равны *false*). Теперь щелкай правой кнопкой по *ToolBar1* и из появившейся меню выбирай пункт *New Button*. На компоненте *ToolBar1* должна появиться новая кнопка с именем *ToolButton1*. Выдели ее и в объектном инспекторе поменяй свойство *Caption* на *Открыть*. Создай еще насколько кнопок с заголовками: *Назад*, *Вперед*, *Стоять*, *Обновить* и *Печать*. Результат должен быть похож на рисунок 22.4.5.

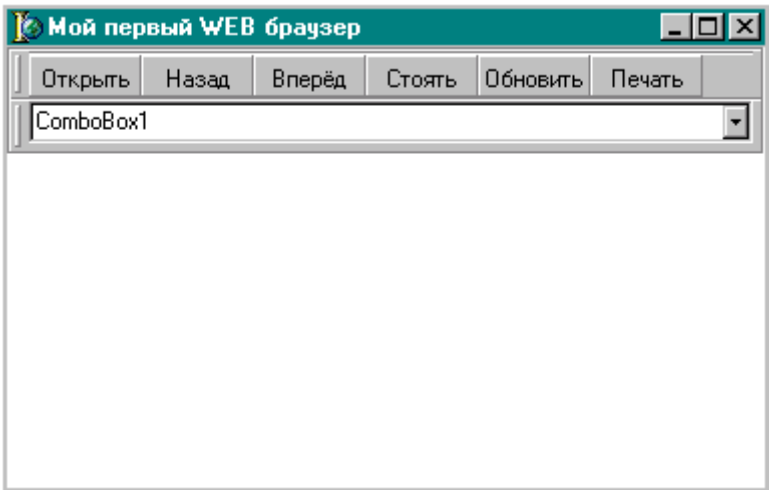


Рисунок 22.4.5. Форма будущей программы

Установи еще на форму *OpenDialog* из закладки *Dialogs* палитры компонентов. Он нам скоро понадобится.

Теперь дважды кликни по кнопке *Открыть*, и Delphi автоматически создаст процедуру, которая будет вызываться при нажатии этой кнопки. В этой процедуре нужно написать следующее:

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
 if OpenDialog1.Execute then
  begin
   WebBrowser1.Navigate(OpenDialog1.FileName);
```

```
   ComboBox1.Text:=OpenDialog1.FileName;
  end;
end;
```

Здесь я заставляю наш браузер загрузить открытый файл и в строке *ComboBox* отображаю его имя, чтобы пользователб знал, какая страница сейчас загружена.

Теперь ты можешь запустить программу и открыть с помощью этой кнопки любой файл на диске. Но, я думаю, что торопиться не надо. Заставим работать остальные кнопки! Дважды кликни по кнопке *Назад*. Какой будет результат, ты уже догадался. Напиши тут следующее:

```
procedure TForm1.ToolButton2Click(Sender: TObject);
begin
 WebBrowser1.GoBack;
end;
```

Я думаю, что здесь ничего объяснять не надо. Мы просто заставляем *WebBrowser1* идти на предыдущую страницу. Повтори те же операции для кнопки *Вперед*, чтобы создать процедуру обработчика *OnClick*. Напиши для нее следующий код:

```
procedure TForm1.ToolButton3Click(Sender: TObject);
begin
 WebBrowser1.GoForward;
end;
```

Для кнопки "Стоять" напиши: "*Стоять на месте, руки по швам*". Шучу. Напиши лучше это:

```
procedure TForm1.ToolButton4Click(Sender: TObject);
begin
 WebBrowser1.Stop;
end;
```

Для кнопки *Обновить*:

```
procedure TForm1.ToolButton5Click(Sender: TObject);
begin
 WebBrowser1.Refresh;
end;
```

И, наконец, для кнопки *Печать*:

```
procedure TForm1.ToolButton6Click(Sender: TObject);
var
```

```
 PostData, Headers:OLEvariant;
begin
 WebBrowser1.ExecWB(OLECMDID_PRINT,OLECMDEXECOPT_DODEFAULT,
   PostData, Headers);
end;
```

Здесь только одна строка, но очень сложная, поэтому я не стану ее объяснять. Скажу только, что в этой строке я посылаю команду через OLE ядру IE. Просто скопируй ее один к одному в свой исходник и поверь мне на слово. Просто это особенности браузера, в которые вникать просто нет смысла, всё равно такое ты наверно нигде больше не увидишь.

Теперь можешь нажать "F9", и твоя программа должна запуститься. Попробуй поиграть с ней. Неплохие ощущения? Закрывай свой браузер, остались последние штрихи! Твой браузер почти готов. Я только наведу небольшой марафет.

Для начала брось на форму *StatusBar* из закладки *Win32* и измени у него свойство *SimplePanel* в *true* (по умолчанию *false*). Теперь выдели *WebBrowser1* и щелкни по закладке *Events* в объектном инспекторе. Дважды кликни по строке *OnStatusTextChange* и напиши в созданной процедуре следующее:

```
procedure TForm1.WebBrowser1StatusTextChange(Sender: TObject;
 const Text: WideString);
begin
 StatusBar1.SimpleText:=Text;
end;
```

Здесь мы присваиваем переменную *Text* (в ней хранится текст подсказки), которую получили в качестве параметра обработчика в *StatusBar1*. Теперь ты сможешь видеть подсказки в строке состояния.

Давай добавим ещё индикатор загрузки. Для этого брось на форму *ProgressBar* из закладки *Win32*. Измени у него свойство *Align* на *alBottom*, чтобы он находился вдоль нижней границы формы. Снова выдели *WebBrowser1* и щелкни по закладке *Events* в объектном инспекторе. Дважды щелкни по строке *OnProgressChange* и напиши в созданной процедуре:

```
procedure TForm1.WebBrowser1ProgressChange(Sender: TObject; Progress,
  ProgressMax: Integer);
begin
 ProgressBar1.Max:=ProgressMax;
 ProgressBar1.Position:=Progress;
end;
```

Здесь мы созданному компоненту *ProgressBar1* (индикатор загрузки) присваиваем максимальное значение (*ProgressMax*) и текущее значение (*Progress*) полученные в качестве параметров.

Теперь надо украсить наши кнопки, а то они смотрятся как сам IE. Для этого брось на форму *ImageList* и произведи по нему двойной щелчок. Перед тобой откроется окно, как на рисунке 22.4.6. Сюда нужно добавить картинки размером 16x16. Для этого нажми кнопку "Add", и перед тобой откроется стандартное окно открытия файла. Найди

картинку и нажми *Открыть*. Повтори эту процедуру 6 раз (6 картинок для 6-и кнопок). После всего этого нажми *OK*.

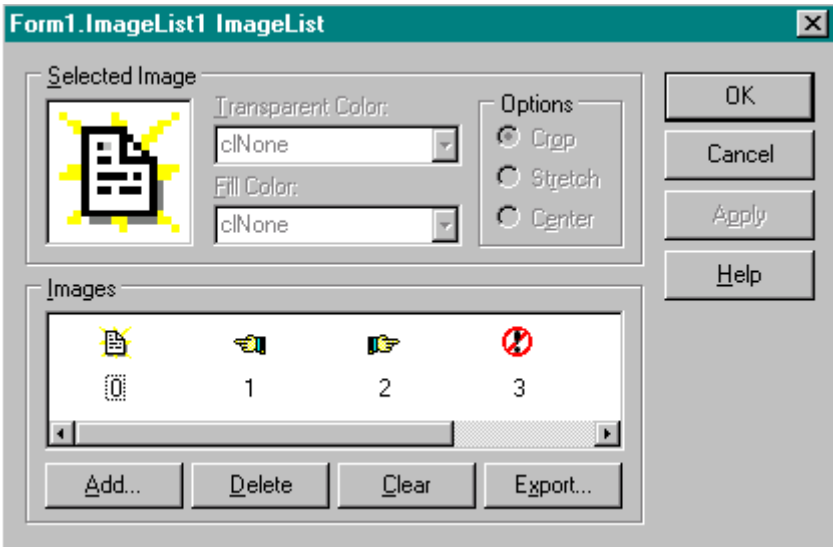


Рисунок 22.4.6 Добавление картинок

Теперь выдели *ToolBar1* и в объектном инспекторе измени свойство *Images* на *ImageList1*. На твоих кнопках должны появиться картинки. Если ты добавлял картинки не в том порядке, как они у тебя стоят на форме, то можешь пересортировать их с помощью свойства *ImageIndex* у кнопки. Например: щелкни по кнопке *Стоять* и измени ImageIndex на 0. На кнопке должна появиться картинка, указанная первой в *ImageList1*.

Можешь создать еще один *ImageList*, который подставляется в *HotImages*. В этом случае картинки из этого компонента будут подставляться на кнопку, когда ты наводишь на нее мышкой.

Все, косметический ремонт окончен. Дави на F9, и Delphi в последний раз создаст тебе окончательную версию. На рисунке 22.4.7 ты можешь увидеть результат сегодняшней работы.

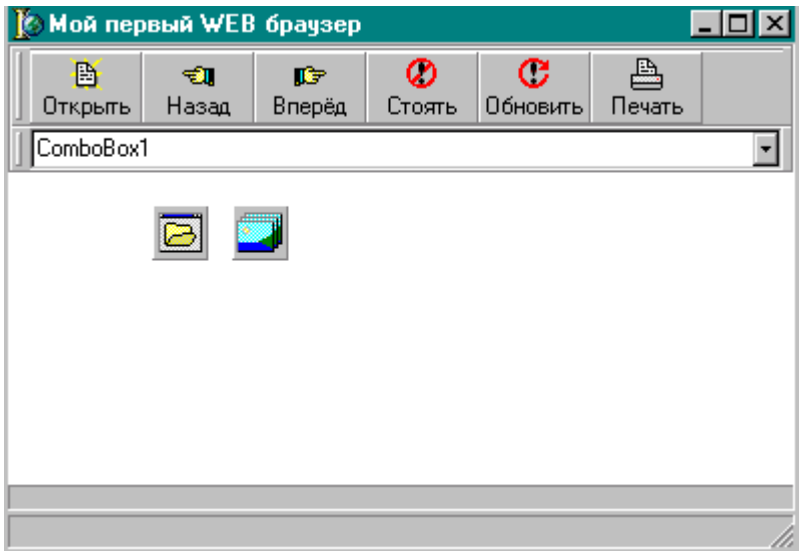


Рисунок 22.4.7 Результат работы

Можешь пользоваться полноценным браузером в свое удовольствие. Конечно же, это не все возможности, которые можно получить из компонента *WebBrowser1*. Сюда еще очень многое можно добавить - главное, чтобы хватило воображения и умений. Но это уже специфические детали, и я их описывать здесь не буду, потому что сейчас у нас идёт совершенно другой рассказ.

 *На компакт диске, в директории* ***\Примеры\Глава 22\WebBrawser*** *ты можешь увидеть пример этой программы.*

22.5. Модель COM

Модель COM (*Component Object Model*) – это независимая от языка программирования спецификация объектов. В спецификации COM объекты называют интерфейсами (потому что у них есть небольшие отличия), но я буду использовать понятие объект, потому что оно мне ближе к телу. Да и нет смысла вводить новое понятие из-за того, что кому-то из Microsoft захотелось выделиться.

Прежде чем приступать к рассмотрению модели, хочу дать пару определений. Взгляни на приложение, написанное в предыдущей главе (наш собственный браузер интернета). Там мы писали программу, которая использует ActiveX компонент ядра IE. Так вот наше приложение называется клиентским, а ядро IE, к которому мы подключились называется COM сервером. Ты должен обязательно понимать эту разницу, когда будешь читать остальной текст этой главы.

Самое большое отличие объектов COM в том, что они реализовываются в виде отдельных файлов. Когда ты хочешь использовать этот объект, то должен подгрузить этот файл объекта.

Файл с объектом называют сервером COM. Такие сервера могут быть выполнены в виде динамических библиотек DLL или запускных EXE файлов. Если сервер реализован в виде динамической библиотеки, то его называют *внутренним*. Ну а если в виде запускного файла, то *локальными* (*внешним*). Локальный сервер функционирует в своём собственном адресном пространстве. Внутренний сервер загружается в адресное пространство клиентского процесса (твоей программы). COM сервера могут быть также и удалёнными, когда они выполняются на другой машине. В этом случае используется расширенная спецификация DCOM (*Distributed COM*).

Клиентское приложение, которое хочет загрузить COM сервер никогда не задумывается о том, где находиться сам сервер. Приложение может узнать реальное расположение (через реестр Windows), но это ему ненужно. За всё это отвечает операционная система. Ей нужно только передать **GUID** (globally unique identifier - глобально уникальный идентификатор) сервера, который нам нужен и ОС все остальные функции инициализации проведёт самостоятельно.

Что значит **GUID**? Это номер, который COM объект получает на этапе проектирования. Он генерируется случайным образом и никакие два объекта не смогут иметь одинаковые **GUID**. За уникальность отвечал сам Билл Гейтс, но при этом делал оговорку, что вероятность совпадения дух номеров **GUID** всё таки есть, хотя и ничтожно мала. А что же будет, если всё же эта вероятность сработает? Неужели вместо запрашиваемого ядра IE мы увидим ядро Excel? Лично я не знаю, и надеюсь, что ничего страшного всё же не произойдет.

Вот здесь я не буду дальше заводить разговор во внутренности технологии COM потому что они достаточно сложны и для разработки приложений абсолютно не нужны. Лучше остановимся и будем изучать то, что нам пригодиться на практике.

Все объекты модели COM происходят от *IUnknown*. Как видишь, в имена объектов COM начинаются с буквы *I* (имена объектов Delphi начинаются с *T*), символизируя слово *Interface*. Объект *IUnknown* действует так же, как *TObject* для объектов Delphi. Это основа, от которого происходят все остальные объекты. В нём реализуются основные метода, которые могут понадобится в последствии для других объектов. В модели COM – это три метода:

QueryInterface – этот метод служит для получения информации об встроенных в COM объектах/интерфейсах. Когда тебе нужно загрузить какой-нибудь объект, то с помощью этого метода проверяется его доступность. Если метод подтверждает возможность использования указанного объекта, то его можно загружать.

_AddRef и *_Release* – эти два метода используются для создания и уничтожения объекта. Объекты COM похожи на динамические библиотеки. В память может быть загружена только одна версия и все клиенты будут обращаться к ней. Чтобы всё это реализовать, необходимо контролировать, сколько клиентов подключено к объекту, чтобы знать, когда можно уничтожать его из памяти.

Когда мы загружаем объект, то вызывается метод *_AddRef*, который увеличивает внутренний счётчик на единицу. При следующем обращении к объекту снова вызывается этот метод и снова счётчик увеличивается на 1. Когда какое-то приложение отключилось от объекта (вызван метод *_Release*), то счётчик уменьшается на 1 и если он равен 0, то объект можно выгружать из памяти, иначе с ним ещё кто-то работает и выгрузка невозможна.

Объект *IUnknown* объявлен в Delphi следующим образом:

```
IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
  function _AddRef: Integer; stdcall;
  function _Release: Integer; stdcall;
end;
```

Как видишь, в первой строке идёт имя нового интерфейса и после знака равно ставиться ключевое слово **interface**. Во второй строке, в квадратных скобках нужно указывать **GUID** интерфейса. Никогда не пиши его вручную. Когда ты будешь писать собственные COM объекты, то этот уникальный номер будет генерировать ОС.

Я показал объект *IUnknown* только для того, чтобы ты смог увидеть самый простейший COM объект. Теперь же давай посмотрим, как создаются COM объекты на практике.

Для создания COM объекта в Delphi нужно выбрать File->New->Other. Перед тобой откроется уже знакомое окно создания нового проекта. Здесь перейди на закладку ActiveX и посмотри, что тут тубе доступно (рисунок 22.5.1). Тут достаточно много разных типов COM объектов и я даже не собираюсь рассматривать их все. Тема COM – это отдельный разговор и если тебе захочется узнать большего, то желательно купить отдельную книгу. Я же дам только основы, чтобы тебе было легче потом читать специализированную литературу.

Если ты не будешь связывать свою жизнь только с COM, то моей информации тебе будет достаточно для написания приложений средней тяжести, потому что Delphi прячет большинство рутины, связанной с программированием COM объектов. Так что использование прямого программирования практически ненужно и я его не буду трогать.

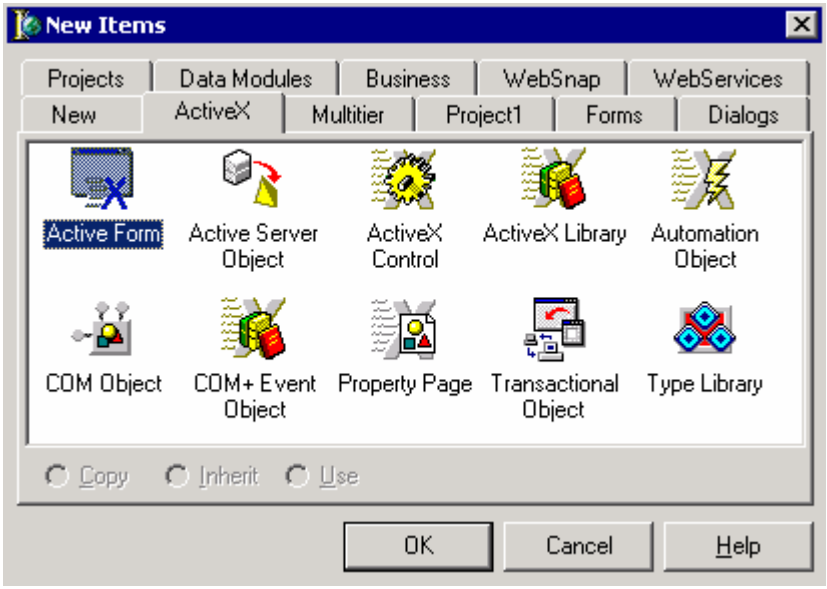


Рисунок 22.5.1 Окно создания COM объектов

22.6. Пример создания ActiveX форм

Первое, с чем мы познакомимся – *Active Form*. Это форма, на которой могут располагаться различные компоненты и элементы управления. Такая форма может быть встроена в любую другую программу, поддерживающую COM технологию или даже выложена в сети Internet. Всё это я покажу на практике в этой главе.

Формы *Active Form* – это файлы с расширением OCX и представляющие собой самый настоящий COM объект. Большинство элементов управления получают расширение OCX, чтобы отличать эти файлы среди массы других файлов.

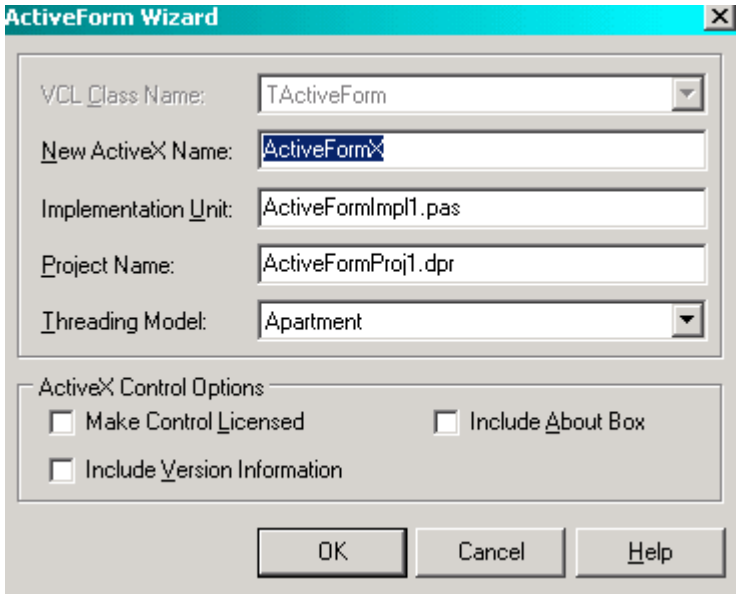


Рис 22.6.1. Свойства нового ActiveX

Для начала создадим новый проект. Для этого выбери File->New->Other и в появившемся окне перейди на закладку *ActiveX*. Здесь выбери пункт *ActiveForm* и нажми *OK*. Перед тобой должно открыться окно, как на рисунке 22.6.1. В нём ты должен указать следующие поля:

New ActiveX Name – имя твоей ActiveX формы. Постарайся дать здесь разумное имя, потому что оно будет потом использоваться для отображения в системе. Не очень приятно будет смотреть на компонент с именем *ActiveFormX*.

Implementation Unit – имя исполнительного модуля.

Project Name – имя проекта, тоже постарайся дать разумное имя, потому что так будет называться файл.

Threading Model – для нас достаточно здесь значения по умолчанию.

Make Control Licensed – создать лицензию для компонента.

Include Version Information – включить в компонент информацию о версии.

Include About Box – включить окно «*О программе*».

Я поменял только имя компонента (*SimpleActiveX*) и проекта (*SimpleActiveProj1*). *CheckBox*-ы оставил без изменений, потому что не хочу иметь контроля версии или окна «*О программе*». Жми ОК и перед тобой появится привычная визуальная форма, на которой можно располагать свои собственные компоненты. Смело располагай на ней компоненты VCL и работай, как с привычным проектом. Ты можешь засунуть сюда целую программу по учёту заработной платы депутатов. Я не стал сильно извращаться, потому что я делаю простой пример. На рисунке 22.6.2 ты можешь увидит моё творение.

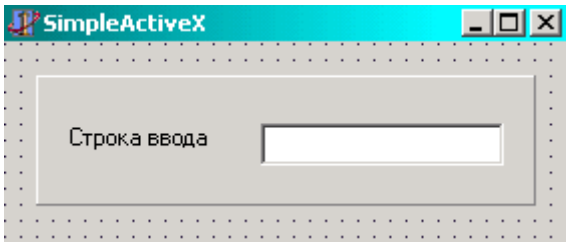


Рис 22.6.2. Свойства нового ActiveX

Визуальную часть я оставляю на тебя, потому что она не должна вызывать проблем. Я же расскажу из чего состоит наш проект. Открой менеджер проектов и посмотри на его содержимое (рисунок 22.6.3). Здесь в дереве находиться наш проект с именем *SimpleActiveProj1.ocx*, который состоит из двух файлов:

1. *SimpleActiveImpl1* – это файл, в котором храниться наша форма.
2. *SimpleActiveProj1.pas* – это файл с описанием возможностей нашего проекта. Не советую в него лазить без особой надобности. Здесь очень много достаточно сложных вещей, поэтому лучше оставить его создание на совесть Delphi. Я же этот файл рассматривать не буду, потому что он пока тебе не нужен.

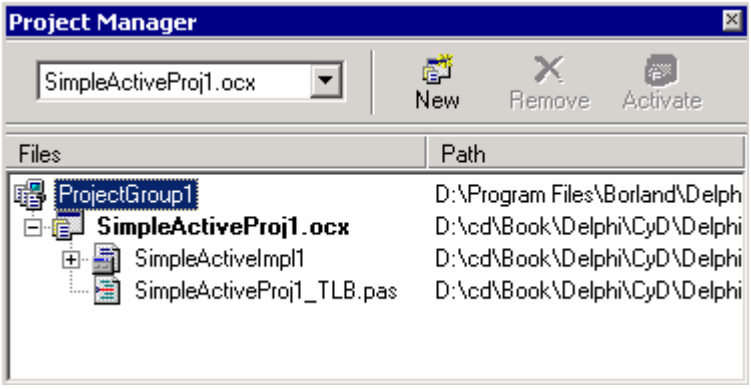


Рис 22.6.3. Менеджер проекта

После того, как ты установил все компоненты, откомпилируй проект нажав Ctrl+F9. Откомпилированный OCX файл готов. Теперь надо зарегистрировать его в системе, чтобы

можно было его протестировать. Я уже говорил, что при вызове COM объектов вся информация о нём находиться в реестре и всё необходимое попадает туда при регистрации. Для этого выбери register ActiveX Server из меню Run . Для регистрации можно так же выполнить в командной строке следующую команду:

Regsvr32.exe ИмяФайла

Можно переходить к тестированию. В качестве тестирования я создам HTML страничку и попробую загрузить форму с помощью IE. Для этого выбери *Web Deployment Option* из меню *Project*, и перед тобой откроется окно публикации компонента в сети Internet, как на рис. 22.6.4.

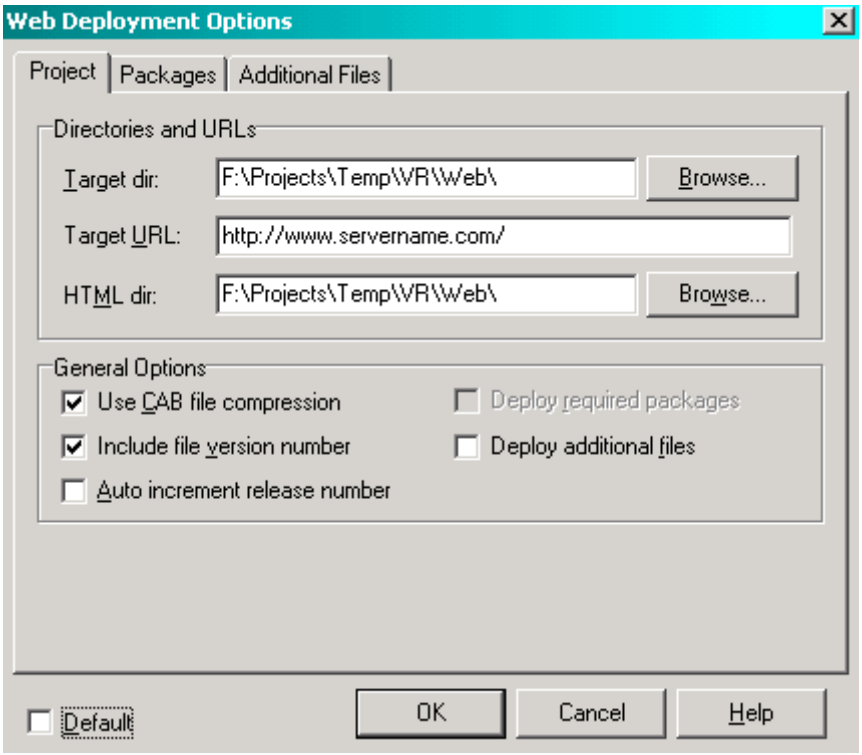


Рис 22.6.4. Окно публикации компонента в интернете

Рассмотрим каждый параметр в отдельности:

Target Dir - директория, в которую попадёт скомпилированный файл OCX.

Target URL - Адрес в инете, откуда будет загружен OCX. Когда пользователь будет загружать страничку, то компонент будет скачан именно с этого адреса.

HTML Dir - директория, в которую попадёт шаблон HTML-файла, который потом можно использовать для публикации.

Use CAB compression - упаковать осх файл в CAB архив. Необходимо указывать только если объект создавался именно для публикации в сети. В этом случае компонент сжимается за счёт чего уменьшается скорость его загрузки по сети.

Auto increment release number – автоматически увеличивать номер релиза.

Заполни поля и нажми *OK*. Заполнение полей *Target Dir*, *Target URL* и *HTML Dir* является обязательным. Я советую тебе выбирать *Target Dir* и *HTML Dir* отличную от той в которой лежит твой проект. Эти директории нужно указать реальные, а вот адрес интернете можно указывать любой. Если компонент уже зарегистрирован в системе, то этот адрес не понадобиться. Браузер будет обращаться к этому адресу только если в системе не найден указанный компонент. В нашем случае мы уже произвели регистрацию в системе и можем использовать компонент по своему усмотрению.

Теперь выбирай пункт меню *Web Deploy* и Delphi и Delphi создаст необходимый для тестирования HTML файл. Запусти этот файл и ты увидишь окно, как на рисунке 22.6.5. Как видишь, достаточно просто создать маленькое приложение, которое сможет работать в сети Internet прямо внутри браузера IE. Только учти, что если ты собираешься выкладывать свои файлы в интернете, то не каждый пользователь сможет воспользоваться услугами твоей программы. В защите ActiveX очень много дыр, поэтому эти компоненты по умолчанию не могут загружаться из сети на компьютеры пользователей. Чтобы пользователь смог увидеть твою программу, он должен понизить уровень безопасности в своём IE до минимума и разрешить загрузку ActiveX по сети.

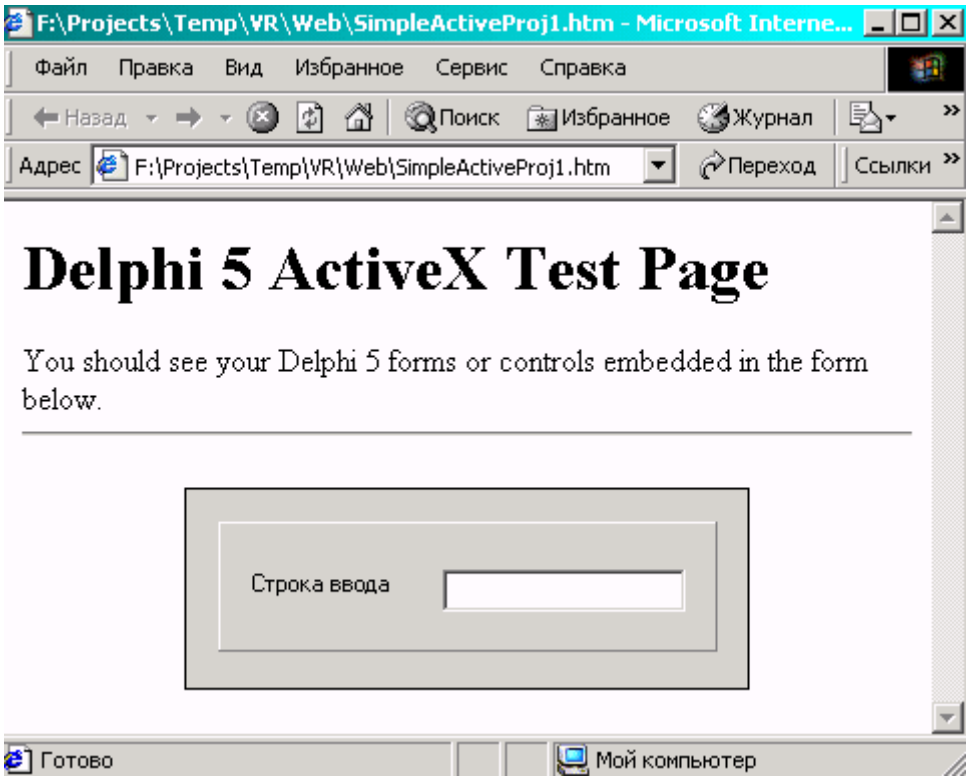


Рис 22.6.5. Результат запуска HTML-файла

Благодаря такой плохой защищённости, компоненты ActiveX не получили должного распространения. Никто не хочет понижать безопасность своего компьютера и надёяться на добропорядочных программистов. Таких программистов не так уж и много, поэтому лучше лишний раз перестраховаться и не включать ActiveX.

И всё же, в локальных сетях ActiveX используют достаточно много, особенно в России. У таких форм достаточно много преимуществ и позволяют решить много проблем:

1. Централизованное обновления. Когда программист внёс в своё продукт обновление, то он должен установить его на все машины, работающие с его программой. А если таких компьютеров много? Можно пытаться известить своих пользователей по почте или каким-либо другим способом. А если неизвестны все пользователи? Вот тут уже задача усложняется. В данном случае достаточно только изменить осх файл и закачать его на сервер, откуда IE качает файл (параметр *Target URL* окна *Web Deployment Option*). При следующем запуске IE сам загрузит обновлённый файл и установит его.

Есть ещё один способ протестировать наш пример. Для этого выбери *Import ActiveX Control…* из меню “Component” и ты увидишь окно, как на рис 22.4.6. Да, именно через это окно я показывал, как установить движок IE. Найди в верхнем списке имя твоего компонента и нажми Install. Delphi установить этот компонент и его иконка появится на

закладке *ActiveX* палитры компонентов. Теперь ты сможешь устанавливать его на любую форму и вообще, использовать как простой компонент.

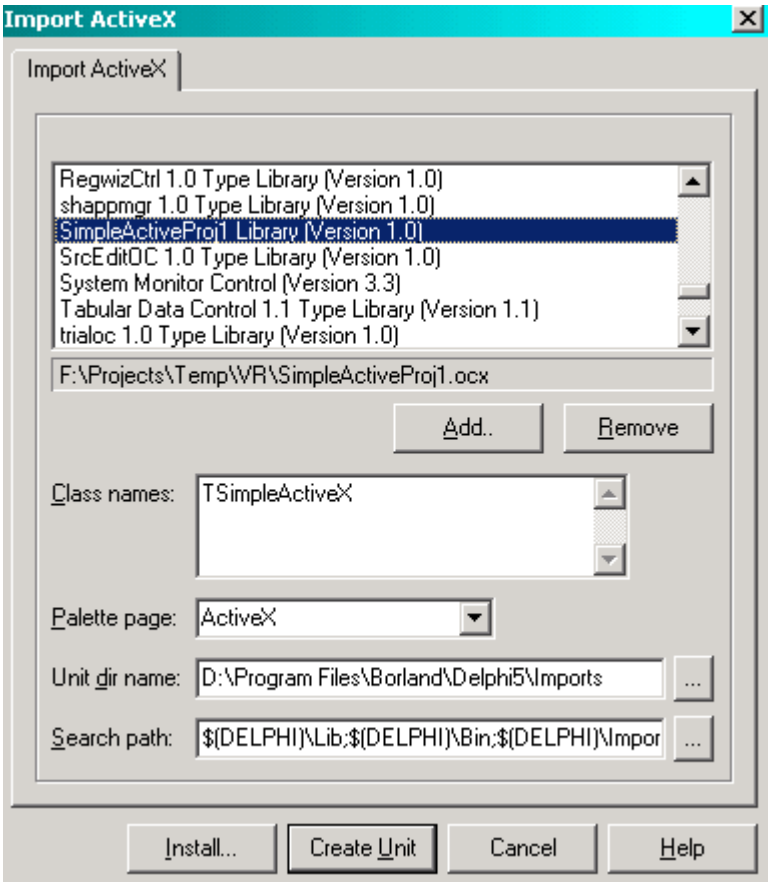


Рис 22.6.6. Результат запуска HTML-файла

На компакт диске, в директории ***\Примеры\Глава 22\ActiveForm*** *ты можешь увидеть пример этой программы.*

22.7. Создание ActiveX компонентов

Ты уже наверно убедился на практике, что ActiveX - это достаточно сложная для понимания и разработки технология. Она является продолжением развития OLE, которая была пополнена технологией COM и переименована в более ёмкое название - ActiveX. Под этим термином скрывается несколько самостоятельных технологий:

1. Формы ActiveForm - мы с ними познакомились в самом начале.
2. Элементы управления
3. Библиотеки
4. Серверы автоматизации
5. Страницы свойств

Всё это объединяется под одним термином ActiveX. С первым мы уже познакомились, теперь предстоит познакомиться со вторым – элементами управления.

Для разработки элементов управления на основе ActiveX нужно иметь достаточно много знаний и навыков. Фирма Borland упростила эту технологию как для понимания, так и для разработки создав свою надстройку - Delphi ActiveX (часто сокращается до DAX). Благодаря DAX ты можешь любой компонент Delphi превратить компонент

ActiveX и использовать его в любой другой среде разработки. Вот именно эти мы и займёмся. Сейчас я покажу, как создать компонент ActiveX с использованием Delphi и надстройки DAX. Хотя о присутствии DAX ты можешь и не знать.

Выбирай меню File->New->Other и на закладке ActiveX выбирай *ActiveX Control*. По нажатию кнопки *OK*. Перед тобой должно открыться окно *ActiveX Control Wizard*, как на рис 22.7.1.

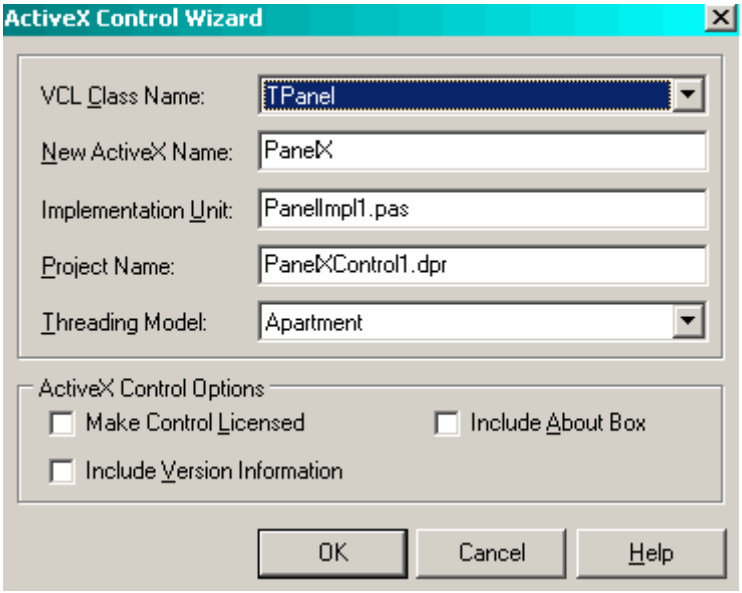


Рис 22.7.1. Мастер создания компонента ActiveX

Давай рассмотрим содержимое окна ActiveX Control Wizard :

VCL Class Name - Имя компонента, который мы хотим превратить в ActiveX. Выбери из списка *TPanel*.

New ActiveX Name - Имя ActiveX компонента (оставь по умолчанию).

Implementation Unit - Имя модуля (оставь по умолчанию)

Project Name - Имя проекта (оставь по умолчанию)

Threading Model - Модель потока.

Make Control Licensed - Лицензия компонента. Включай только если захочешь продавать свой компонент.

Include Version Information - Включить информацию о версии

Include About Box - добавить окно "*О программе*"

После нажатия кнопки "OK", Delphi создаст шаблон для нового твоего компонента, в которой уже готовы к использованию все методы и свойства компонента TPanel. Весь исходный код реализующий компонент будет находится в модуле *PanelImpl1.pas*. Перейди в него с помощью менеджера проектов и посмотри на содержимое.

Здесь полно процедур и функций начинающихся словом *Get_* или *Set_*. Зачем они нужны? В ActiveX нет свойств или переменных, к которым можно было бы обращаться напрямую, как мы это делали с Delphi компонентом. Весь доступ к свойствам происходит через процедуры или функции, поэтому, чтобы прочитать заголовок панели, Delphi создал функцию *Get_Caption*, а чтобы изменить заголовок на новый - *Set_Caption(const Value: WideString)*. Раньше для этого нам достаточно было просто обратиться к свойству *Caption* компонента панели, здесь такой трюк не проходит. Немного позже мы напишем собственную реализацию изменения заголовка.

Вся информация о методах компонента находится в библиотеке типов. Чтобы её увидеть выбери пункт *Type Library* из меню *View* (рис 22.7.2).

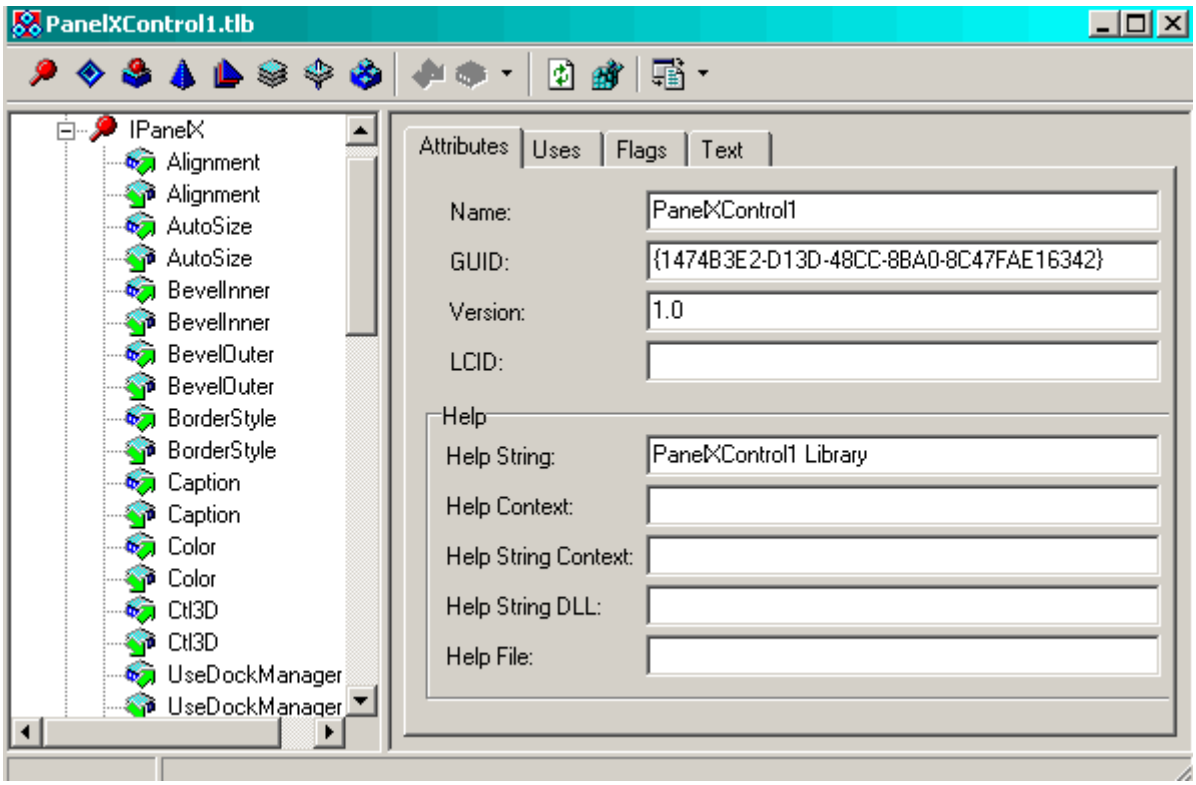


Рис 22.7.2. Библиотека типов

Когда ты выбираешь какой-нибудь элемент библиотеки, в правой стороне окна появляется несколько закладок. На закладке *Attributes* ты можешь видеть:

Name - имя объекта

GUID - уникальный номер в библиотеке (менять не советую)

Version - версия

LCID - Идентификатор языка

Help String - Краткое описание

Help File - Имя Help файла связанного с объектом

Help Context - Идентификатор контекста справки

Вот некоторые из флагов, которые ты тоже можешь тут увидеть:

None - Флаги отсутствуют.

Restricted - Запретить использование библиотеки в средах программирования макросов.

Control - В библиотеке находится компонент ActiveX.

Hidden - Библиотека скрыта от пользователей.

DispInterface - доступ к свойствам и методам производится только через интерфейс *IDispatch*.

Nonextensible - Если выделен, то реализация интерфейса *IDispatch* (основной интерфейс ActiveX) будет включать только те свойства и методы, которые показаны в реализации.

Dual - Методы и свойства интерфейса передаются и через IDispatch, и таблицу виртуальных методов.

OLE Automation - используются только совместимые с автоматизацией типы данных.

Source - указывает, что возвращаемое значение является типа VARIANT, являющееся источником событий.

Bindable - свойство поддерживает связывание данных

Request Edit - свойство поддерживает сообщение *OnRequestEdit*

Небольшое отступление: Программа-клиент может получить доступ к интерфейсам (объектам) через специальный интерфейс *IDispatch*, либо через таблицу виртуальных методов. Интерфейс *IDispatch* позволяет использовать свойства и методы объектов через уникальный идентификатор *DispID*.

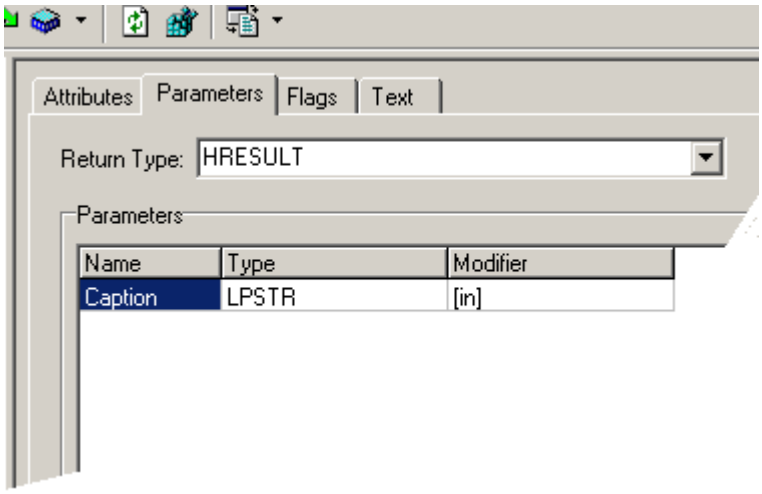


Рис 22.7.3 Закладка Parameters

Давай создадим собственный метод. Для этого выдели ветку *IPanelX* и щёлкни по кнопке. Delphi создаст новое объявление метода *Method1*. Переименуй его в *SetCap*. Delphi создаст процедуру *SetCap*, которая будет является реализацией метода *SetCap*. Перейди на закладку *Parametrs* (рис 22.7.3) и добавь новый параметр для процедуры с именем *Caption* с типом *LPSTR*, и *Modifier* равный [*in*].

Чтобы изменить *Modifier* нужно дважды щёлкнуть по нему и появиться окно, как на рисунке 22.7.4. Здесь тебе доступны:

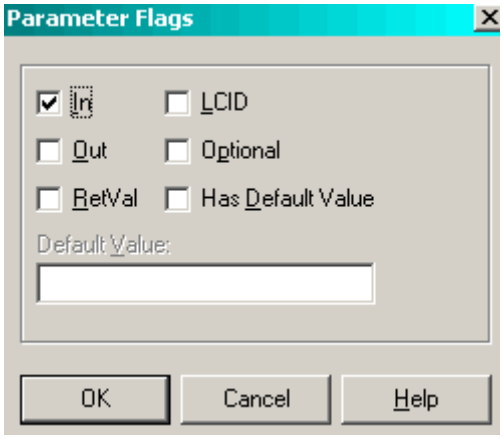


Рис 22.7.4. Изменение параметра Modifier

Здесь ты можешь выставить следующие свойства:

In - означает, что метод является процедурой и используется для установки значений

Out - Говорит о том, что метод будет считывать значение компонента

RetVal - метод будет возвращать значение

Теперь перейди с помощью менеджера проектов в модуль *PanelImpl1.pas* и найди процедуру эту *SetCap* и напиши в ней следующее:

```
procedure TPanelX.SetCap(Caption: PChar);
```

```
begin
 FDelphiControl.Caption:=Caption;
end;
```

FDelphiControl указывает на компонент, с которым мы работаем, в данном случае это *TPanel*. У него мы изменяем свойство *Caption* на то значение, которое указано в качестве параметра в нашей процедуре *SetCap*.

С помощью *SetCap* мы изменяем свойство *Caption* у *TPanel*, то есть наш метод делает то же, что и *Set_Caption*.

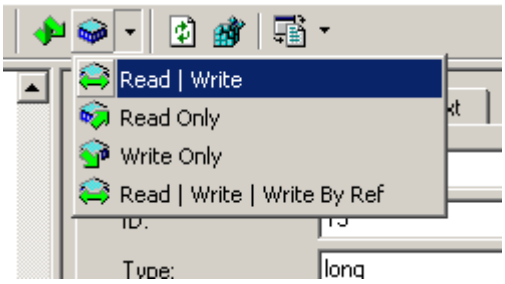


Рис 22.7.5

Теперь создадим свойство. Кликни по кнопке New Property и выбери "read | write" (рис 22.7.5). Delphi создаст два метода: один для чтения свойства, а другой для записи. Переименуй их в *MyProp*. В модуле *PanelImpl1* у тебя появится две новые процедуры:

```
function TPanelX.Get_MyProp: Integer;
begin

end;

procedure TPanelX.Set_MyProp(Value: Integer);
begin

end;
```

Можешь их реализовать, но я не стал. Мне главное было показать, как это делается. Зарегистрируй новый компонент в системе (*Run->Register ActiveX Server*). Теперь можешь установить его на палитру компонентов (*Component->Import ActiveX Control*) и протестировать. После регистрации, компонент будет практически не виден на палитре, потому что у него не будет иконки. Чтобы найти созданную панель, перейди на закладку *ActiveX* и проведи мышкой по палитре компонентов. Самой правой окажется *PanelX*.

Для теста нужно:

1. Создай новый проект
2. Поставь на форму новый компонент
3. Поставь кнопку и напиши по её событию:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
 PanelX1.SetCap('привет');
end;
```

 *На компакт диске, в директории* ***\Примеры\Глава 22\Control*** *ты можешь увидеть пример этой программы.*

На диске, в директории «Документация» ты можешь найти документ под названием «Программирование PWS.doc». В нём описывается процесс написания приложений для WEB сервера. Эти приложения пишутся на основе технологии COM, поэтому я советую тебе прочитать этот документ. Даже если ты не будешь писать свои программы под WEB сервер, документ может оказаться полезным и в будущем может пригодиться.

Глава 23. Буфер обмена

Кнопки «*Копировать*» и «*Вставить*» есть практически в любом полноценном приложении. Я думаю, что ты тоже захочешь вставить такую возможность в свои программы. В этой главе я дам максимум полезной теоретической и практической информации, чтобы ты смог сделать свои программы более привлекательными, добавив возможность переноса данных между приложениями.

Если в твоей программе есть строки ввода *TEdit*, то они уже умеют работать с буфером обмена. Попробуй щёлкнуть в любой такой строке правой кнопкой и перед тобой откроется меню, в котором есть все необходимые пункты. ОС Windows очень много делает за нас и нам не приходиться заботить о таких мелочах.

С другими компонентами Windows дело обстоит немного сложнее. Тут нам придётся немного поработать на клавиатуре. Но эта работа не так уж сложна и ты убедишься в этом, когда закончишь чтение этой главы.

23.1. Буфер обмена и стандартные компоненты Delphi

Большинство компонентов Delphi уже готовы к работе с буфером обмена. В основном это касается тех компонентов, которые содержат какие-либо данные, которые пользователь может захотеть поместить в буфер обмена. Если такая возможность нужна, то у компонента будет методы с именами:

1. *CutToClipboard* – вырезать в буфер обмена;
2. *CopyToClipboard* – копировать в буфер обмена;
3. *PasteFromClopboard* – вставить из буфера обмена.

Давай посмотрим на эти метода в действии. Создай новое приложение, и помести на его форму три кнопки: «Вырезать», «Копировать» и «Вставить». По центру окна растяни компонент *TMemo*. Мою форму ты можешь увидеть на рисунке 23.1.1.

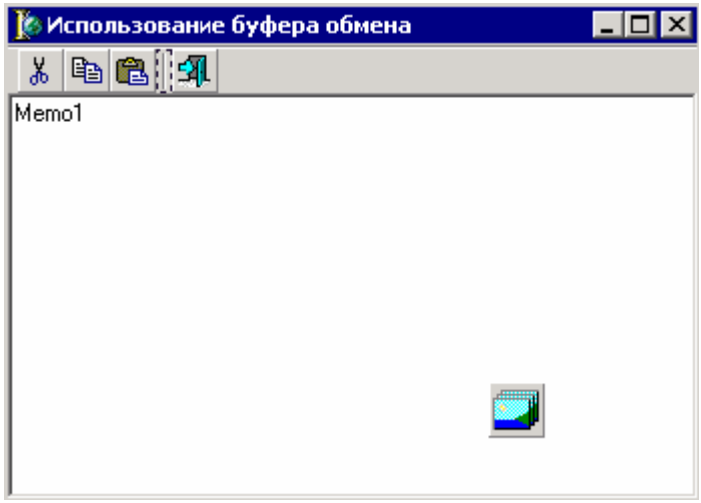


Рисунок 23.1.1 Форма будущей программы

Теперь создадим обработчики событий для кнопок. Для кнопки «Вырезать» напиши следующий код:

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
 Memo1.CutToClipboard;
end;
```

Здесь всего лишь вызывается один метод *CutToClipboard* компонента *Memo1*. Этот метод вырежет выделенный текст в буфер обмена.

По событию *OnClick* для кнопки «*Копировать*» пишем следующий код:

```
procedure TForm1.ToolButton2Click(Sender: TObject);
begin
 Memo1.CopyToClipboard;
end;
```

Здесь происходит копирование выделенного фрагмента текста в буфер обмена с помощью метода *CopyToClipboard*.

Ну и для кнопки вставить пишем вызов метода *PasteFromClipboard* компонента *Memo1*. С помощью этого метода, всё содержимое буфера обмена будет вставлено в компонент *Memo1*.

Теперь попробуй запустить приложение и скопировать какой-нибудь текст в буфер обмена с помощью нашей кнопки «Копировать». После этого нажми на кнопку «Вставить» и скопированный текст появиться в текущей позиции курсора. Если ты скопируешь в буфер какое-нибудь изображение из любого графического редактора и попытаешься вставить его в компонент *Memo1*, то ничего не произойдёт. Система сама следит за форматом данных, хранимых в буфере обмена. Чуть позже я покажу, как сделать так, чтобы кнопка «Вставить» была активна только тогда, когда в буфере есть данные и они соответствуют нужному формату.

Попробуй щёлкнуть правой кнопкой мыши внутри компонента *Memo1*. Перед тобой так же должно появиться всплывающее меню с необходимыми для работы с буфером обмена пунктами.

 *На компакт диске, в директории* ***\Примеры\Глава 23\Memo Clipboard*** *ты можешь увидеть пример этой программы.*

23.2 Объект Clipboard

Для работы с буфером обмена в Delhi есть объект *Clipboard*. Несмотря на то, что это объект, его не надо инициализировать (как и *TApplication* и *TPrinter*), а можно использовать без всяких инициализаций. Достаточно только подключить в разделе **uses** модуль *Clipbrd* и он становиться тебе доступным.

У этого объекта не так уж и много свойств и методов, так что будем их рассматривать на практике. Для начала модернизируем пример, написанный в прошлой части с использованием этого объекта. Открой предыдущий пример и сразу добавь в раздел **uses** модуль *Clipbrd*. Теперь по событию OnClick для кнопки «Копировать» напиши следующий код:

```
Clipboard.SetTextBuf(PChar(Memo1.SelText));
```

Здесь я использую метод *SetTextBuf* объекта *Clipboard*. Этот метод копирует переданный в качестве параметра текст в буфер обмена. В качестве это параметра я передаю выделенный в компоненте *Memo1* текст *Memo1.SelText*. Единственное – это надо привести текст к типу *PChar*.

По нажатию кнопки «Вырезать» пишем следующий код:

```
Clipboard.SetTextBuf(PChar(Memo1.SelText));
Memo1.SelText:='';
```

«Вырезать» - значит скопировать текст в буфер и потом удалить его из компонента *Memo1*. Именно это я и делаю. В первой строке я написал код, который ты видел в обработчике события *OnClick* для кнопки «Копировать». Во второй строке я обнуляю (удаляю) выделенный текст.

По нажатию кнопки «Вставить» пишем следующий код:

```
Memo1.SelText:=Memo1.SelText+Clipboard.AsText;
```

Свойство *AsText* объекта *Clipboard* указывает на содержимое буфера обмена в виде текста. В этом коде я прибавляю это содержимое к выделенному тексту и вставляю это всё в выделенный текст. Вот таким нехитрым способом я реализовал вставку.

Попробуй запустить пример и убедиться, что всё работает. Как видишь, использование объекта *Clipboard* не намного сложнее и бояться тут нечего.

Теперь посмотрим ещё несколько интересных методов объекта *Clipboard*:

Assign – назначить в буфер обмена объект совместимый с *TPersistent*. К таким объектам относятся картинки *TImage*. В качестве единственного параметра нужно указать объект, содержимое которого нужно скопировать в буфер обмена.

Clear – очистить содержимое буфера обмена.

HasFormat – проверка, какого типа данные хранятся в буфере обмена. Возможны следующие типы данных:

CF_TEXT - буфер содержит текст.

CF_BITMAP - буфер содержит картинку.

CF_METAFILEPICT - буфер содержит векторную картинку.

CF_PICTURE - буфер содержит объект типа *TPicture*.

CF_COMPONENT - буфер содержит компонент.

Я добавил в наш пример одну кнопку, по нажатию которой будет выводиться сообщение, показывающее тип данных хранящихся в буфере обмена данных. По нажатию этой кнопки написан следующий код:

```
procedure TForm1.InfoButtonClick(Sender: TObject);
begin
 if Clipboard.HasFormat(CF_TEXT) then
  Application.MessageBox('Буфер содержит текст', 'Внимание!');
 if Clipboard.HasFormat(CF_BITMAP) then
  Application.MessageBox('Буфер содержит картинку', 'Внимание!');
 if Clipboard.HasFormat(CF_METAFILEPICT) then
  Application.MessageBox('Буфер содержит векторную картинку', 'Внимание!');
 if Clipboard.HasFormat(CF_PICTURE) then
  Application.MessageBox('Буфер содержит объект типа TPicture', 'Внимание!');
 if Clipboard.HasFormat(CF_COMPONENT) then
  Application.MessageBox('Буфер содержит компонент', 'Внимание!');
end;
```

SetComponent – назначить в буфер обмена компонент. В качестве единственного параметра нужно указать компонент, который надо скопировать в буфер.

SetTextBuf - назначить в буфер обмена текстовый буфер. В качестве единственного параметра нужно указать буфер *PChar*, который надо скопировать в буфер.

На компакт диске, в директории ***\Примеры\Глава 23\Clipboard*** *ты можешь увидеть пример этой программы.*

23.3 Картинки и буфер обмена.

Давай теперь разберёмся, как работать с изображениями в буфере обмена. Как ты знаешь, у нас основной компонент для работы с картинками является *TImage*. Вот давай напишем пример, в котором будет копироваться и вставляться изображение из буфера обмена в компонент *TImage*.

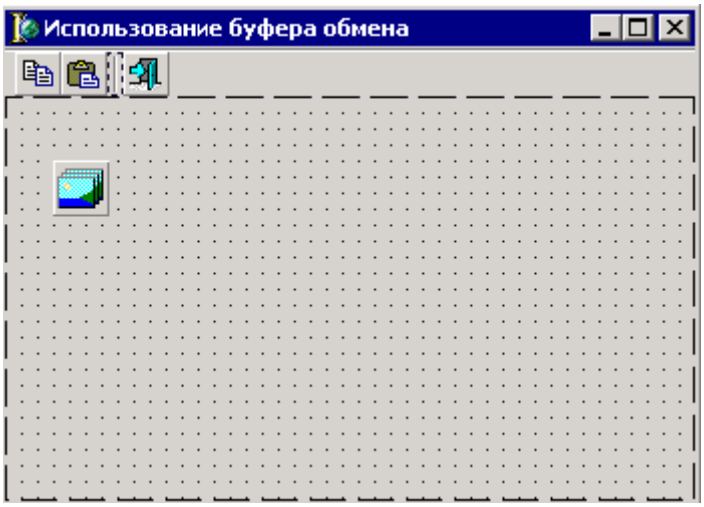


Рисунок 23.3.1 Форма будущей программы

Создай новый проект, и помести на форму две кнопки «Копировать» и «Вставить». По центру окна расположи компонент *TImage*, который будет хранить изображения. Мою форму ты можешь увидеть на рисунке 23.3.1.

По событию *OnClick* для кнопки «Копировать пишем следующий код:

```
procedure TForm1.CopyButtonClick(Sender: TObject);
begin
 Clipboard.Assign(Image1.Picture);
end;
```

Здесь я просто устанавливаю методом *Assign* в буфер обмена содержимое свойства *Picture* компонента *Image1*. Свойство *Picture* имеет тип *TPicture*, который среди своих родителей имеет объект *TPersistent*, поэтому мы можем использовать метод *Assign*. Посмотри на рисунок 23.3.2, на котором показана иерархия объекта *TPicture*.

Рисунок 23.3.2 Иерархия объекта TPicture

Теперь посмотрим на код, который надо написать по нажатию кнопки «Вставить»:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
begin
 Image1.Picture.Assign(Clipboard);
end;
```

Здесь происходит обратное назначение свойству *Picture* содержимое буфера обмена с помощью метода *Assign*.

В принципе, пример готов и на этом можно было бы остановиться, но я здесь же хочу показать, как сделать так, чтобы кнопка «Вставить» была доступна только тогда, когда в буфере обмена находиться именно картинка. Для этого сначала движемся в раздел **private** и объявляем там следующее:

```
private
  { Private declarations }
  FClipboardOwner:HWnd;
  procedure WMDrawClipboard(var Msg: TWMDrawClipboard);
      message WM_DRAWCLIPBOARD;
```

Я объявил здесь переменную *FclipboardOwner* типа *HWnd*. Это тип, который используется для идентификации окна. Вспомни, каждый раз, когда нам нужно было получить или передать указатель на окно, то мы использовали свойство *Handle*. Вот это свойство имеет тип *HWnd*, и здесь мы явно объявили переменную такого же типа, чтобы мы могли работать с окнами.

Здесь так же объявлена процедура *WMDrawClipboard* с одним лишь параметром типа *TWMDrawClipboard*. Это процедура обработчик события *WM_DRAWCLIPBOARD*. Об этом говорит соответствующая надпись после объявления процедуры и точки с запятой. Там у нас стоит ключевое слово **message** и имя сообщение, на которое должна откликаться процедура. Вот таким нехитрым способом мы вручную описали обработчик системного сообщения, которого нет в Delphi. Имена всех сообщений Windows ты можешь найти в директории Delphi, поддиректории *Source/Rtl/Win*, в файле *Messages.pas*. Такие имена всегда начинаются с приставки WM_ (Windows Message – сообщение Windows).

Итак, описанная нами процедура будет вызываться каждый раз, когда изменилось содержимое буфера обмена. Теперь нажимаем Ctrl+Shift+C и Delphi создаёт для нас пустую заготовку описанной процедуры. В ней пишем следующее:

```
procedure TForm1.WMDrawClipboard(var Msg: TWMDrawClipboard);
begin
 SendMessage(FClipboardOwner, WM_DRAWCLIPBOARD, 0, 0);
 Msg.Result := 0;
 ClipboardChanged;
end;
```

Первые две строчки я опущу, и описывать не буду. Поверь мне, они необходимы. В последней строке я вызываю процедуру *ClipboardChanged*. Она должна выглядеть так:

```
procedure TForm1.ClipboardChanged;
var
 I: Integer;
begin
 PasteButton.Enabled := False;
 for I := 0 to Clipboard.FormatCount - 1 do
  begin
```

```
  if Clipboard.HasFormat(CF_BITMAP) then
   begin
    PasteButton.Enabled := True;
    Break;
   end;
  end;
end;
```

Для начала я делаю кнопку «Вставить» неактивной. Потом я запускаю цикл от 0 до количества форматов в буфере обмена *Clipboard.FormatCount*. Внутри цикла происходит проверка, если формат соответствует *CF_BITMAP*, то кнопку «Вставить» можно делать активной и прерывать цикл проверки.

И последнее, что надо сделать в нашей программе – написать обработчик события OnShow для нашей главной формы. В нём пишем следующее:

```
procedure TForm1.FormShow(Sender: TObject);
begin
 FClipboardOwner := SetClipboardViewer(Handle);
 ClipboardChanged;
end;
```

В первой строке я вызываю функцию *SetClipboardViewer*. Она устанавливает указанное в качестве параметра окно (наше главное окно) в системе в качестве наблюдателя за буфером обмена. После этого, как только буфер измениться, нашему окну будет отправлено соответствующее сообщение, и мы его поймаем процедурой *WMDrawClipboard*.

Во второй строке я вызываю процедуру *ClipboardChanged*, чтобы при старте программы произошла проверка, а вдруг там находиться картинка или наоборот. Если этой проверки не производить, то программа после запуска ещё не будет знать, что находится в буфере обмена, пока он не измениться и программа не получит соответствующего сообщения.

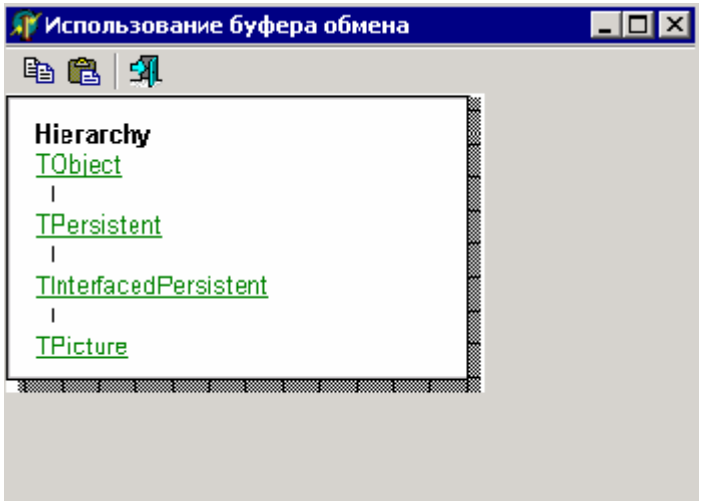


Рисунок 23.3.3 Пример работы программы.

Вот теперь на этом всё. Пример рабочей программы ты можешь увидеть на рисунке 23.3.3.На этом рисунке я вставил в нашу программу изображение иерархии объекта

TPersistent. Попробуй запустить свой вариант программы и последить за кнопкой «Вставить». Запусти любые другие программы и попробуй в них поместить в буфер данные разного типа. Как только ты поместишь туда картинку, так сразу же твоя программа отреагирует на это и сделает кнопку «Вставить» активной.

 *На компакт диске, в директории* ***\Примеры\Глава 23\Image*** *ты можешь увидеть пример этой программы.*

23.4 Создание собственного формата для работы с буфером.

Представь себе ситуацию, когда в твоей программе есть какой-то объект и нужно дать пользователю возможность копировать её в буфер и вставлять в нужное место. Стандартные форматы данных для буфера *CF_TEXT*, *CF_BITMAP*, *CF_METAFILEPICT*, и так далее не подходят, но данные копировать надо. Лично я с такой ситуацией встречаюсь практически в каждой своей программе. В таких случаях нужно создать свой собственный формат данных, с которым и будет работать буфер обмена.

Язык Delphi – это объектный язык и тут я Америки для тебя уже не открываю. А по условиям объектного программирования, чтобы добавить новые возможности к уже существующему объекту нужно создать его потомка. В данном случае мы по идее должны вывести потомка из TClipboard и наделить его великолепными и уникальными возможностями по форматированию нужных нам данных. В данном случае это будет глупо. Объектное программирование хорошее дело, но только когда оно в меру. В данном случае нам не понадобиться наследственность и мы не будем возиться с родителями и детками.

Итак, создай новый проект и сразу же создай в нём новый модуль (File->New->Unit). Delphi создаст пустой модуль, который мы сохраним под именем *ClipboardFormatUnit*. Результат – модуль вот с таким вот содержимым:

```
unit ClipboardFormatUnit;

interface

implementation

end.
```

После ключевого слова **interface** добавляем раздел **type** и объявление структуры *TLineData* и нового объекта *TLineClipboard*.

```
type
  TLineData=record
    Name:String[100];
    LastName:String[100];
    Bothday:String[10];
    Age:Integer;
    Telephone:String[15];
  end;

  TLineClipboard=class
```

```
 public
  LineData:TLineData;
  procedure CopyToClipboard;
  procedure PasteFromClipboard;
 end;
```

Структура *TLineData* состоит из пяти полей. Именно эту структуру мы будем помещать в буфер обмена. Как ты уже понял, объект *Clipboard* не может работать со структурами, и мы сейчас напишем модуль, с помощью которого мы научим его это делать.

После структуры идёт объявление нового объекта. Здесь мы объявляем новый объект вручную описывая все его метода и свойства. Чаще всего за нас это делал Delphi. Обрати внимание на то, что он объявлен, как простой объект без каких либо родителей (*TLineClipboard=class*). Несмотря на это, он будет иметь родителя – *TObject*, потому что все объекты должны иметь родителя и если ничего не указано, то будет использоваться базовый объект *TObject*. У нового объекта будет только одно свойство типа структуры *TLineData* и два метода для копирования и вставки данных в буфер обмена.

Теперь, после раздела **type** напишем **var** и опишем одну переменную:

```
var
 CF_PERSONDATA:word;
```

В этой переменной будет храниться указатель на зарегистрированный формат для буфера обмена. Давай не будем откладывать это дело на потом, а сразу же реализуем регистрацию в системе этого нового формата. Для этого в конце модуля, перед последним «**end.**» пишем:

```
initialization
 CF_PERSONDATA:=RegisterClipboardFormat('CF_PDATA');

end.
```

Здесь мы объявили блок **initialization**, который всегда выполняется автоматически при обращении к модулю или любому его содержимому. В этом блоке я присваиваю переменной *CF_PERSONDATA* результат выполнения функции *RegisterClipboardFormat*. Эта функция регистрирует новый формат для буфера обмена с именем указанным в качестве единственного параметра. Результат выполнения функции – число, идентифицирующее зарегистрированный формат данных. После этого, этот формат можно увидеть в свойстве *Formats* объекта *Clipboard*.

Вот теперь перейдём к реализации функций записи и чтения данных из буфера.

```
procedure TLineClipboard.CopyToClipboard;
var
 Data:THandle;
 DataPtr:Pointer;
begin
 //Выделяем память под данные
 Data:=GlobalAlloc(GMEM_MOVEABLE, SizeOf(LineData));
```

```
 try
  DataPtr:=GlobalLock(Data);
  Move(LineData, DataPtr^, SizeOf(TLineData));

  //Заполняем буфер обмена
  Clipboard.Open;
  Clipboard.SetAsHandle(CF_PERSONDATA, Data);
  Clipboard.AsText:=LineData.Name+#13#10+LineData.LastName+#13#10+
     LineData.Bothday+#13#10+IntToStr(LineData.Age)+#13#10+
     LineData.Telephone;
  Clipboard.Close;
  GlobalUnlock(Data);
 except
  GlobalFree(Data);
 end;
end;
```

В процедуре объявлено две переменные:

Data – в эту переменную мы будем выделять память для хранения структуры, которую надо будет поместить в буфер обмена.

DataPtr – указатель на выделенную для переменной *Data* память.

В самом начале процедуры я присваиваю переменной *Data* результат вызова функции *GlobalAlloc*. Эта функция выделяет нужный кусок памяти в глобальной памяти. У неё два параметра:

1. Параметры (флаги) выделяемой памяти. Здесь я указал флаг *GMEM_MOVEABLE*, который указывает на то, что память может быть перемещаемой. Например, когда ОС не хватает оперативной памяти, то она может выгрузить некоторую часть памяти на диск и по мере надобности вернуть эту память обратно. Такой флаг позволяет ОС производить такие манипуляции.
2. Размер выделяемой памяти. Здесь я указываю размер *SizeOf* структуры *LineData*.

Следующим этапом я блокирую выделенную память с помощью функции *GlobalAlloc* и получаю указатель на выделенную память, который сохраняется в переменной *DataPtr*.

У нас есть выделенная память, и мы её временно заблокировали для работы с ней. Теперь мы должны скопировать в эту память структуру, которая должна быть помещена в буфер обмена. Для этого я пользуюсь процедурой *Move*, которая копирует данные из одного участка памяти (первый параметр – структура LineData) в другой участок (второй параметр – выделенная память DataPtr). Третий параметр – это размер копируемых данных. Я указываю тут размер нашей структуры.

Память подготовлена и в неё занесены данные для буфера обмена. Теперь можно приступать к установке этих данных в буфер. Для начала буфер надо открыть – *Open* для записи. Следующим этапом я заношу в буфер данные в виде структуры (зарегистрированного нами формата *CF_PERSONDATA*). Для этого я использую метод *SetAsHandle*. У этого метода два параметра:

1. Тип заносимых данных.
2. Данные.

Дальше, я заношу данные в виде текста, чтобы программы, которые не знают о существовании моего собственного формата, могли прочитать данные в виде текста. Для этого текст надо занести в свойство *AsText* объекта *Clipboard*. В это свойство я заношу все поля структуры, разделённые символами *#13#10*, Символ #13 означает конец строки, а *#10*

означает перевод каретки на новую строку. Получается, что каждый параметр будет занесён в виде отдельной строки. Чуть позже ты увидишь, как это выглядит на практике.

Данные занесены, можно уничтожать всё, что мы натворили. Для начала закрываю буфер обмена с помощью вызова метода *Close* объекта *Clipboard*. Потом вызываю метод *GlobalUnlock*, который разблокирует память. Уничтожать выделенную память (вызов *GlobalFree*) нужно только если во время записи произошла какая-то ошибка, поэтому я поместил вызов этой процедуры в блоке **except..end**. Если ошибок не было, то память должна остаться целой и невредимой, потому что там храниться структура.

Вот теперь разберёмся с кодом, который получает данные из буфера обмена. Взгляни на процедуру *PasteFromClipboard*:

```
procedure TLineClipboard.PasteFromClipboard;
var
 Data:THandle;
 DataPtr:Pointer;
begin
 Data:=ClipBoard.GetAsHandle(CF_PERSONDATA);
 if Data=0 then exit;

 DataPtr:=GlobalLock(Data);
 Move(DataPtr^, LineData, SizeOf(TLineData));

 GlobalUnlock(Data);
end;
```

В первой строке кода я пытаюсь получить данные из буфера обмена с помощью функции *GetAsHandle*. Этой функции нужно передать формат, в котором мы хотим получить данные. Если значение, которое нам вернула *GetAsHandle* равно нулю, то данных в буфере нет, или они имеют несовместимый формат. В этом случае процедура прерывает своё выполнение.

Если всё нормально, то в переменной *Data* будет указатель на блок памяти с данными буфера обмена. Для их получения я блокирую память и пользуюсь уже знакомой процедурой *Move* для копирования данных из памяти буфера обмена (первый параметр) в структуру *LineData* (второй параметр).

Данные получены, можно разблокировать память с помощью процедуры *GlobalUnlock*.

Всё, модуль готов. Теперь переходим к написанию основной программы, которая будет использовать созданный нами формат данных для копирования и вставки через буфер обмена. У нас уже создан новый проект, в котором мы написали модуль *ClipboardFormatUnit*. Перейди в основное окно, и помести на форму следующие компоненты:

1. Две кнопки «*Копировать*» и «*Вставить*».
2. Компонент *StringGrid*. В нём нужно изменить свойство *goEditing* в разделе *Option* на *true*, чтобы мы могли редактировать ячейки в сетке.
3. *Memo* в котором мы будем отображать содержимое буфера в виде текста.

Мою форму ты можешь увидеть на рисунке 23.4.1. Для реализации кнопок я использовал компонент *ToolBar*, чтобы программа выглядела более эстетично и красиво, но ты можешь поступить и по-другому.

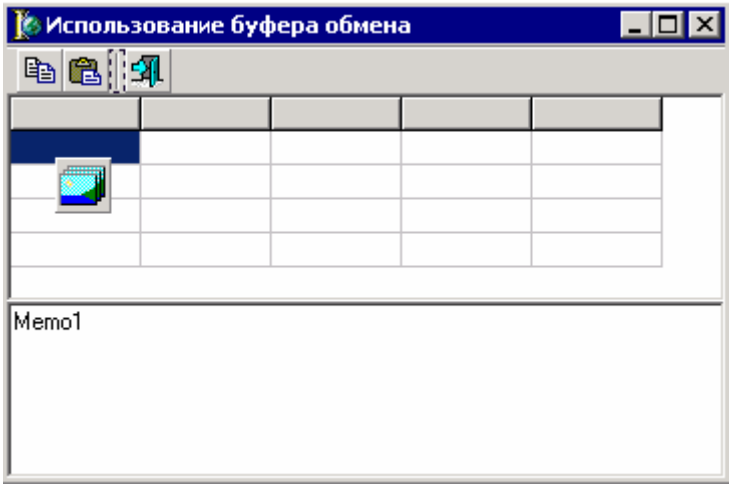


Рисунок 23.4.1 Форма будущей программы

Сразу же добавь в раздел **uses** наш модуль *ClipboardFormatUnit*, чтобы ты мог использовать новый формат из главной формы.

Теперь создадим обработчик события *OnClick* для кнопки «Копировать» и напишем в нём следующее:

```
procedure TForm1.CopyButtonClick(Sender: TObject);
var
 LineClipboard:TLineClipboard;
begin
 LineClipboard:=TLineClipboard.Create;

 LineClipboard.LineData.Name:=StringGrid1.Cells[0, StringGrid1.Row];
 LineClipboard.LineData.LastName:=StringGrid1.Cells[1, StringGrid1.Row];
 LineClipboard.LineData.Bothday:=StringGrid1.Cells[2, StringGrid1.Row];
 LineClipboard.LineData.Age:=StrToInt(StringGrid1.Cells[3, StringGrid1.Row]);
 LineClipboard.LineData.Telephone:=StringGrid1.Cells[4, StringGrid1.Row];

 LineClipboard.CopyToClipboard;

 LineClipboard.Free;
end;
```

В разделе **var** у меня объявлена одна переменная типа *TLineClipboard* – объект для работы с буфером обмена, который мы написали в модуле *ClipboardFormatUnit*. В первой строке кода я инициализирую эту переменную.

Потом я заполняю структуру *LineData* объекта *LineClipboard* данными из выделенной строки сетки. По завершению этого процесса я копирую данные в буфер обмена с помощью вызова метода *CopyToClipboard* объекта *LineClipboard*.

Данные скопированы, значит, объект уже не нужен, и его можно уничтожать. Для этого я вызываю метод *Free*.

По нажатию кнопки «Вставить» пишем следующий код:

```
procedure TForm1.PasteButtonClick(Sender: TObject);
var
 LineClipboard:TLineClipboard;
begin
```

```
LineClipboard:=TLineClipboard.Create;

if Clipboard.HasFormat(CF_PERSONDATA) then
 begin
  LineClipboard.PasteFromClipboard;
  StringGrid1.Cells[0, StringGrid1.Row]:=LineClipboard.LineData.Name;
  StringGrid1.Cells[1, StringGrid1.Row]:=LineClipboard.LineData.LastName;
  StringGrid1.Cells[2, StringGrid1.Row]:=LineClipboard.LineData.Bothday;
  StringGrid1.Cells[3, StringGrid1.Row]:=IntToStr(LineClipboard.LineData.Age);
  StringGrid1.Cells[4, StringGrid1.Row]:=LineClipboard.LineData.Telephone;
 end;

LineClipboard.Free;

Memo1.Lines.Clear;
Memo1.PasteFromClipboard;
end;
```

Опять же, здесь объявлена переменная *LineClipboard*, которая инициализируется в первой строке кода. После этого я проверяю, если буфер обмена содержит информацию в формате *CF_PERSONDATA* (это созданный нами формат), то мы читаем буфер с помощью метода *PasteFromClipboard*. После этого я заполняю поля текущей строки из структуры *LineData* объекта *LineClipboard*.

В самом конце процедуры я очищаю компонент *Memo1* и заставляю его с помощью метода *PasteFromClipboard* прочитать данные из буфера. Этот компонент не знает о существовании нашего формата и читает данные из буфера обмена как текст (это его родной формат). Получается, что мы увидим в компоненте то, что мы записали в свойство *AsText* объекта *Clipboard*. Посмотри на рисунок 23.4.2 и убедись в этом. Там я заполнил поля первой строки, скопировал строку в буфер и потом вставил данные в третью строку. Одновременно со вставкой в компонент *StringGrid* произошла вставка текста буфера в компонент *Memo*.

Рисунок 23.4.2. Результат работы программы

*На компакт диске, в директории **\Примеры\Глава 23\New Format** ты можешь увидеть пример этой программы.*

Глава 24. Дополнительная информация

Я уже рассказал достаточно много и до сих пор не упомянул ни слова об оболочке Delphi. В большинстве книг это первое, с чего начинают обучение программированию, что на мой взгляд грубейшая ошибка. Нет смысла обучать человека тому, чего он абсолютно не собирается применять, потому что не знает зачем и когда.

Сначала нужно заинтересовать человека, что я и пытался сделать на протяжении всей книги. Теперь же, когда у тебя достаточно знаний для написания собственных проектов, пора показать тебе, как работать с оболочкой Delphi, как тестировать и отлаживать написанные тобой приложения. В программах регулярно возникают ошибки даже у именитых профессионалов и для выявления ошибок необходимо много сил и терпения. Я не буду говорить, что Delphi упрощает отладку, потому что это и так понятно. Я лучше всё это покажу на практике.

24.1. Тестирование и отладка

Для начала мы разберёмся с отладкой твоих программ. Отладка – пошаговое выполнение операций программы. В этом режиме, каждая строчка кода выполняется по команде и сразу после её выполнения Delphi останавливает её работу и ждёт следующей команды. Пока программа остановлена, ты можешь просмотреть значения всех переменных на данном этапе и даже изменить их.

Точка прерывания отладки - строка кода, на которой программа должна остановить своё выполнение и перейти в Delphi для продолжения выполнения в пошаговом режиме.

Для того, чтобы поставить точку прерывания, нужно выделить строку и нажать F5. Эта строка должна окраситься в красный цвет. Если она сразу или после компиляции оказалась другого цвета, то на этой строке программа не может останавливаться. На рисунке 24.1.1 ты можешь видеть пример строки, на которой установлена точка прерывания.

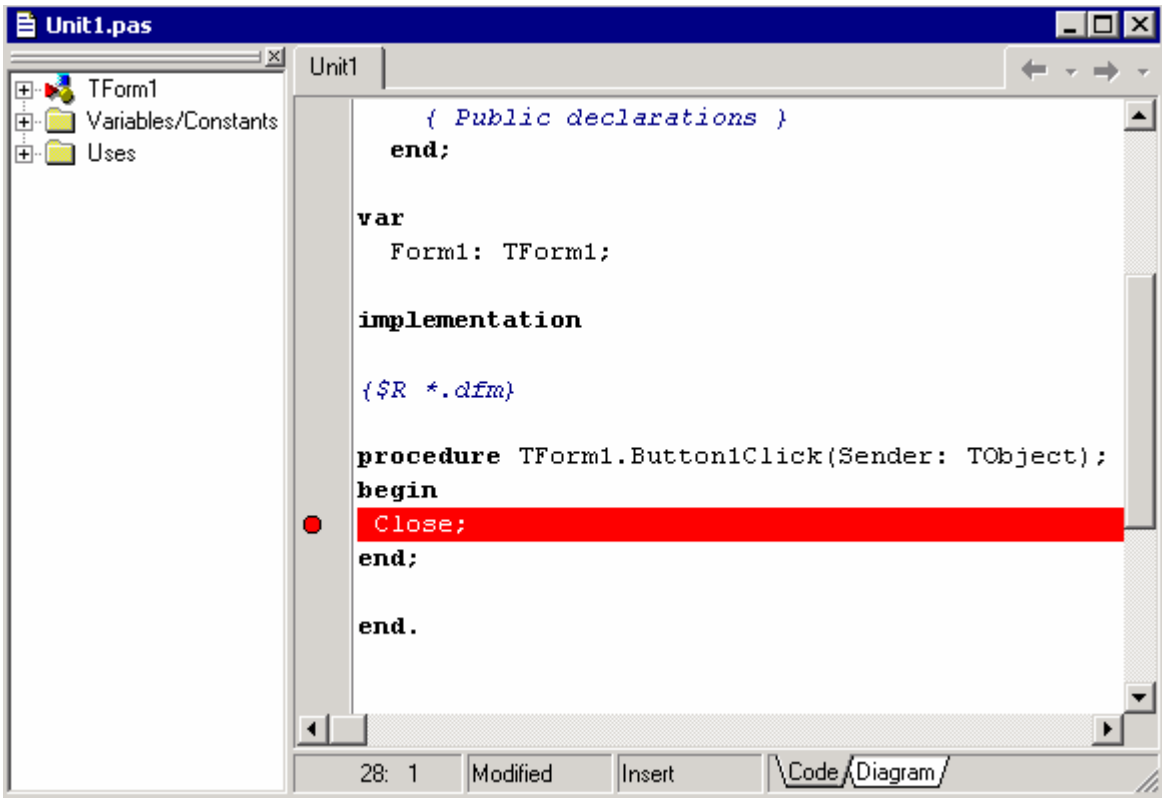


Рис 24.1.1. Точка прерывания.

Если слева от строки стоит синяя точка, то эта строка 100% может стать точкой прерывания. Если такой точки нет, то и прерывания не может быть. Эти точки видны не всегда. Они могут пропадать и появляться только после очередной компиляции программы.

Давай напишем маленький пример и попробуем разобраться с отладкой на нём. Создай новое приложение и установи на форму одну кнопку. По её нажатию напиши следующий код:

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

```
var
 i, j:Integer;
begin
 i:=10;
 j:=20;
 i:=i+j;
end;
```

Теперь откомпилируй программу (Ctrl+F9). После компиляции ты увидишь три сообщения типа *Value assigned to ‘i’ never used* (рисунок 24.1.2). Эти сообщения говорят о том, что значения присвоенные указанным переменным не используются. В таких случаях Delphi оптимизирует код программы и раз значения не используются, то и незачем компилировать этот код.

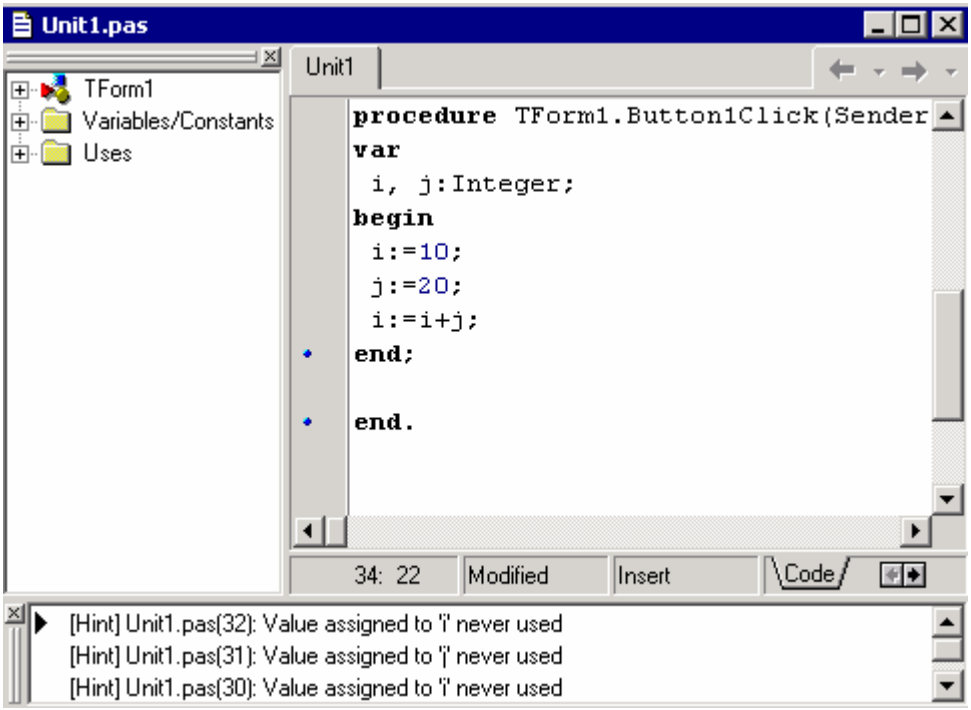


Рис 24.1.2. Список сообщений внизу окна редактора кода.

Обрати внимание, что слева от наших строк кода не появились синие точки. Это значит, что мы не сможем поставить на них прерывания. Точнее сказать сможем, но они не будут работать. Это связано с тем, что Delphi оптимизировал эти строки, потому что они явно не влияют на ход программы. Мы производим расчёты, но они никуда не выводятся. Так что по синим точкам ты можешь проверить, какой код был оптимизирован.

Давай откорректируем наш код до следующего вида:

```
procedure TForm1.Button1Click(Sender: TObject);
var
 i, j:Integer;
begin
 i:=10;
 j:=20;
 i:=i+j;
 if i>0 then exit;
end;
```

Здесь я всего лишь добавил в самом конце процедуры проверку переменной *i*. Если она больше нуля, то произойдёт выход процедуры. Если так подумать, то этот код тоже не влияет на ход работы программы, потому что даже если *i* будет меньше нуля, процедура всё равно заканчивается и произойдёт выход. Но если теперь откомпилировать программу, то сообщений не будет и слева от строк кода появятся синие точки. Вот такой код уже имеет смысл отлаживать.

Запусти программу и как только ты нажмёшь на кнопку, то выполнение остановится и управление получит Delphi. Чтобы продолжить выполнения программы до следующей строки можно нажать F8.

Если выделенная строка – твоя процедура или функция, то можно нажать F7, чтобы отладчик перешёл внутрь этой процедуры и продолжил её выполнение построчно. Если ты нажмёшь F8, то отладчик выполнит процедуру без тебя и перейдёт дальше. В этом случае ты не сможешь посмотреть, что происходит внутри процедуры.

Если ты хочешь, чтобы отладочный режим закончился и программа продолжила выполнятся самостоятельно, то нажми F9. После этого программа продолжит своё выполнение с последней остановленной точки. Если ты хочешь совсем остановить работу программы, то нажми Ctrl+F2 или выбери из меню *Run* пункт *Program Reset* и программа будет выгружена из памяти.

Если ты хочешь посмотреть, какое значение хранится в переменной, то надо её выделить и нажать Ctrl+F7 или выбрать *Evaluate/Modify* из меню *Run*. Перед тобой откроется окно, как на рисунке 24.1.2. Если ты видишь, что значение неправильное и хочешь его изменить, то в строке *New value* нужно ввести нужное значение и нажать кнопку Modify. Чтобы перечитать значение переменной нужно нажать кнопку *Evaluate*.

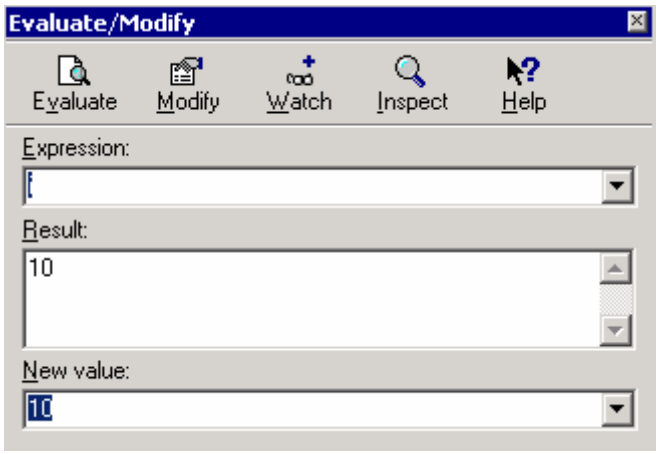


Рисунок 24.1.2 Окно просмотра значений переменной

В окне *Evaluate/Modify* можно вводить не только переменные, но и целые выражения. Например, поставь точку прерывания в примере выше на первой строчке кода процедуры *Button1Click*. Теперь запусти программу и нажми кнопку. Выполнение программы должно прекратиться и управление перейдёт в Delphi. Нажми F8, чтобы выполнить первую строку кода и перейти на вторую. Текущая строка должна быть выделена синим цветом. Теперь нажми Ctrl+F7 и в строке *Expression* окна *Evaluate/Modify* введи следующее выражение: **i+10**. Нажми *Enter* или кнопку *Evaluate*, чтобы просчитать это выражение. В строке *Result* должно появиться значение 20, потому что переменная i равна 10 плюс ещё 10 получаем результат 20.

Введи в строку *Expression* просто переменную i. Перечитай её значение нажатием кнопки *Evaluate*. Теперь с строке *New Value* введи значение 15 и нажми кнопку *Modify*. Теперь переменная i должна равняться 15. Проверь это нажатием кнопки *Evaluate*.

Попробуем ввести в строке *Expression* выражение **i=10**. Здесь используется операция сравнения переменной i и числа 10. Если переменная равна 10, то мы должны увидеть в строке *Result* значение *true*, иначе *false*.

Теперь надо сказать пару слов о видимости переменных. Останови работу программы. Для этого можешь нажать Ctrl+F2, чтобы программа выгрузилась без сохранения данных или продолжи программу нажатием F9 для окончания отладочного режима и закрой окно программы. Снова запусти программу нажатием F9 из Delphi. Нажми кнопку в окне программы и снова Delphi прервёт выполнение и перейдёт в отладочный режим. Выдели переменную i и нажми Ctrl+F7, чтобы увидеть значение переменной. Вместо её значения, в строке *Result* ты увидишь следующий текст: *Variable 'i' inaccessible here due to optimization*. Этот текст гласит примерно следующее (в моём вольном переводе): *переменная i недоступна в этом месте потому что оптимизирована*. В данном случае программа остановила своё выполнение на строке:

```
i:=10;
```

здесь мы присваиваем переменной i значение 10, до этого кода переменная не имеет значения, поэтому мы и увидели такую надпись. Попробуй выполнить эту строку нажатием F8 и снова посмотреть значение переменной i. Теперь значение равно десяти.

Попробуй выполнить процедуру до самого конца. Остановись на следующей строчке:

```
if i>0 then exit
```

Теперь попробуй посмотреть значение переменной *j*. Ты снова должен увидеть сообщение о том, что переменная оптимизирована и не имеет значения. Это связано с тем, что начиная с текущей строчки кода и до конца процедуры уже нет обращений к значению переменной. Значит это значение не нужно и Delphi его снова оптимизировала.

Есть ещё один способ увидеть текущее значение переменной. Ты должен также выделить эту переменную и нажать Ctrl+F5 или выбрать *Add Watch* из меню *Run*. Переменная будет добавлена в специальное окно *Watch*, в котором будет постоянно отображаться её текущее значение (рисунок 24.1.3).

Рисунок 24.1.3. Окно Watch

Если ты делаешь всё, как я говорю, то у тебя программа должна быть остановлена на последней строке кода процедуры. Нажми F9 чтобы продолжить её выполнение, и ты увидишь окно своей программы. Снова нажми кнопку и программа снова остановит выполнение на первой строчке процедуры (если ты не снял точку останова). Добавь переменную *i* в окно *Watch*.

Попробуй дважды щёлкнуть по строке с переменной *i* в окне *Watch* и перед тобой откроется окно параметров просмотра (рис 24.1.4). В этом окне ты можешь сделать достаточно много настроек, но самое интересное – это в центре окна. Здесь находиться большое количество элементов *RadioButton*. Выделяя один из них ты выбираешь тип твоей переменной. В зависимости от выбранного типа ты будешь по разному видеть её в окне *Watch*. Но чаще всего ты не будешь изменять эти настройки и достаточно значений по умолчанию.

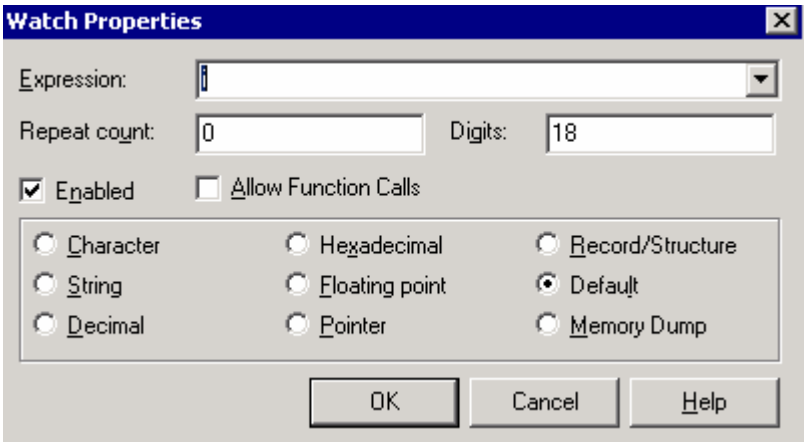


Рисунок 24.1.4. Окно Watch

Теперь попробуй расположить окно *Watch* так, чтобы оно всегда было видно и не мешало видеть код программы. Попробуй построчно выполнить код программы нажатием F8 и понаблюдать за изменением значения переменной *i*.

На компакт диске, в директории ***\Примеры\Глава 24\Отладка*** *ты можешь увидеть пример использованной здесь программы.*

24.2. Работа с редактором

Теперь я хочу познакомить тебя с некоторыми приёмами по работе с редактором кода Delphi. В этой части ты узнаешь, как работать с закладками, как быстро создавать переменные, процедуры и функции и как искать нужный код. Если у тебя маленькая программа и модуль состоит из нескольких строк, то тут у тебя не будет проблем, потому что найти что-то нужное не так уж и сложно. А что если проект большой и модуль из 1000 строк? Вот тут возникает множество проблем, с которыми надо бороться.

Закладки

Встань на какую-нибудь строку текста (выделять не надо) и нажми Ctrl+Shift+ любая цифра. Эти клавиши поставят закладку на строке. Теперь перейди в другое место и нажми Ctrl и ту же цифру. После этого ты вернёшься в строку, где была закладка.

Когда ты устанавливаешь закладку на строку, то слева от строки кода появляется изображение кубика с цифрой внутри. Цифра указывает на номер закладки и именно эта цифра в сочетании с Ctrl моментально перенесёт тебя на выделенную строку. Пример редактора кода с закладкой ты можешь увидеть на рисунке 24.2.1.

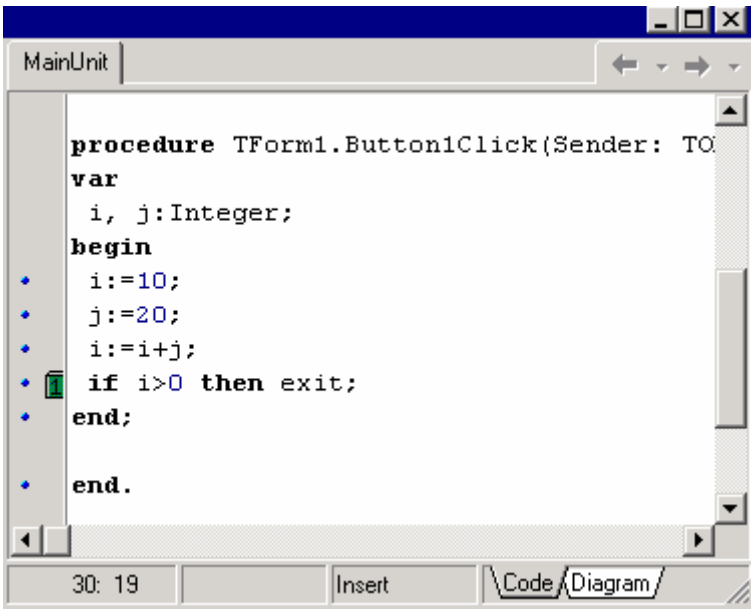


Рисунок 24.2.1. Окно редактора кода с закладкой

Закладки можно устанавливать и из контекстного меню. Щёлкни правой кнопкой мыши в редакторе кода и в появившемся меню наведи на пункт *Toggle Bookmarks*. Откроется подменю в котором ты можешь увидеть изображения кубиков с номерами и подписи. Выбирая любой из пунктов ты можешь установить соответствующую закладку.

Если ты установил закладку с номером 1 на одну строку и потом устанавливаешь на новую строку закладку с этим же номером, то из предыдущего места расположения закладка снимается и переноситься в новое место. Чтобы просто снять закладку достаточно перейти на её строку и повторно установить там закладку с тем же номером. В этом случае закладка просто исчезнет.

Копирование строк

Встань в начало строки текста. Выдели строку с помощью нажатия Shift+стрелка вниз. Это выделит всю строку. Теперь удерживая Ctrl нажми *K*, а затем *C* (буквы латинские). Этот маленький трюк скопирует выделенную строку без использования буфера обмена (clipboard) и установит её чуть ниже копируемой

Code Explorer

Теперь посмотрим, как быстро создавать переменные. В окне кода слева, находиться вытянутое окно *Code Explorer* в котором ты можешь видеть дерево из трёх пунктов:

1. *TForm1* – в этой ветке находиться описание всех компонентов стоящих на твоей форме.
2. *Variables/Constants* – здесь хранятся переменные и константы.
3. *Uses* – здесь храниться список модулей, подключённых к программе.

Давай попробуем создать новую глобальную переменную в разделе **var**. Для этого щёлкни правой кнопкой по пункту *Variables/Constants* и выбери в появившемся меню пункт *New*. В дереве будет создан пункт, в котором нужно ввести следующий текст:

```
H: Integer
```

Таким образом мы создали переменную *H* в разделе **var** и при этом не имеет значения, в какой строчке кода мы находимся. На рисунке 24.2.2 ты можешь увидеть создание нового элемента.

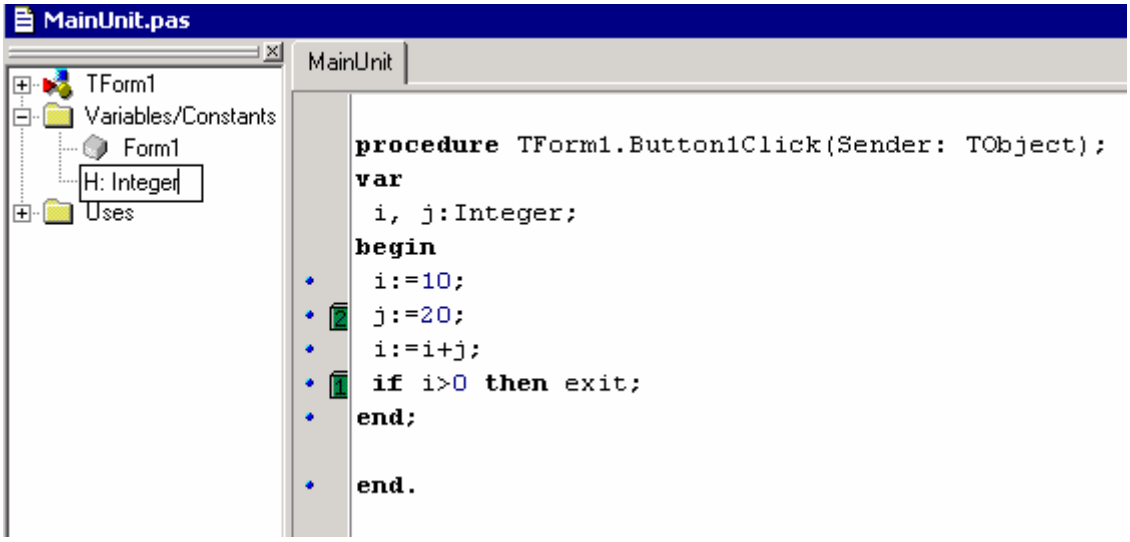


Рисунок 24.2.1. Окно редактора кода с закладкой

Такой приём очень удобен, когда у тебя очень большой модуль и не хочется переходить в самое начало, а потом возвращаться назад.

Точно так же можно создавать не только переменные, но и процедуры и функции для объекта главной формы. Для этого нужно щёлкать правой кнопкой по ветке *TForm1* (имя объекта твоего окна).

24.3. Создание программ инсталляции

Когда твоя программа готова, надо задуматься о том, как бы её установить на компьютер пользователя. Если программа состоит только из одного файла, то никаких проблем. А что если программа состоит из множества файлов? В этом случае используют программы установки, которые запускаясь копируют всё необходимое на компьютер клиента. Именно этот способ я советую тебе использовать и в этой части мы познакомимся с программой *InstallShield Express*.

Программу *InstallShield Express* чаще всего можно найти на том же диске, что и Delphi. Она устанавливается отдельно и если ты ещё не установил её, то сделай это сейчас.

Запустив программу ты увидишь окно, как на рисунке 24.3.1. Главное окно содержит файл помощи, в котором показано, как работать с программой. Слева окна расположено дерево, в котором можно выбирать раздел, который тебя интересует. В центре окна будет отображаться информация, найденная по выбранному разделу. Если у тебя нет проблем с английским, то ты сможешь разобраться по файлу помощи, но если проблемы есть, то лучше почитать эту часть книги, потому что здесь я постараюсь описать всё необходимое для создания собственный программ инсталляции.

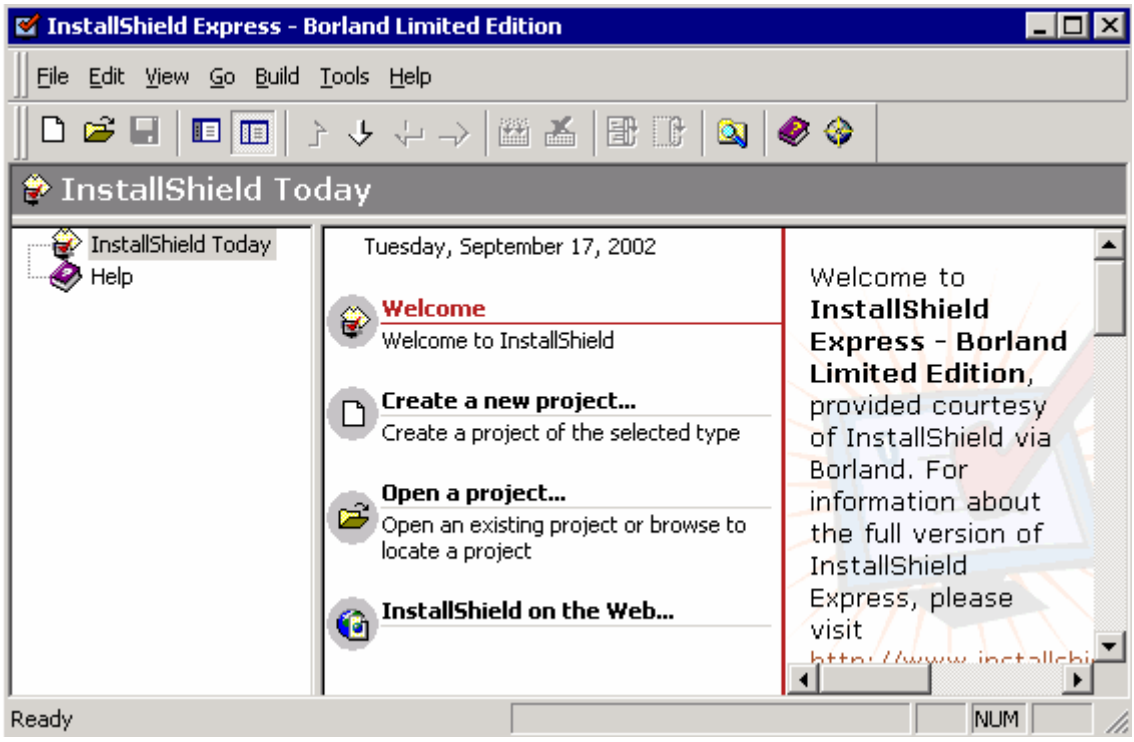


Рис 24.3.1. Главное окно программы InstallShield Express

Для создания нового проекта выбери из меню *File* пункт *New*. Перед тобой откроется окно создания нового проекта. В этом окне нужно указать лишь путь проекта и имя файла. У имени файла должно быть расширение *ism* и старайся ему задавать вполне понятное имя, потому что оно будет использоваться в качестве имени проекта. Указав путь и имя проекта нажимай *OK*.

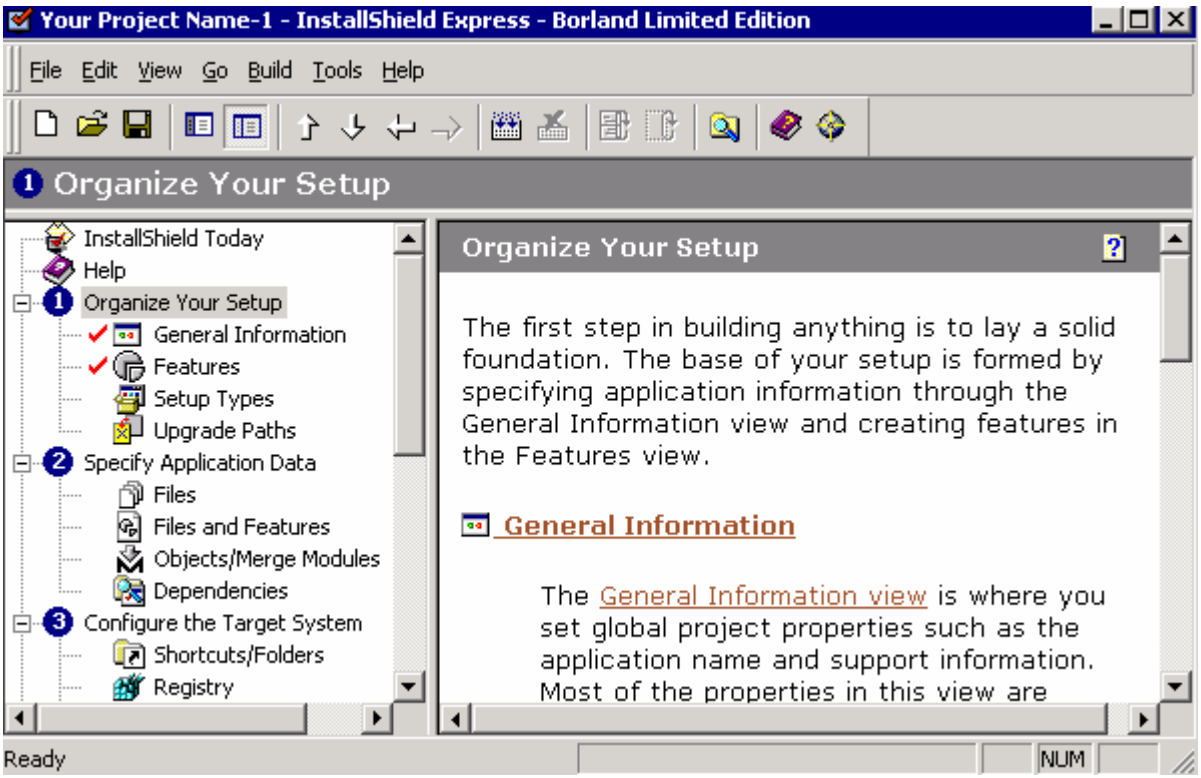


Рис 24.3.2. Главное окно программы после создания нового проекта

Теперь главное окно измениться, и примет вид подобный рисунку 24.3.2. Дерево слева, где были пункты помощи, теперь пополнилось новыми пунктами. Выбирая эти пункты, ты слева будешь указывать параметры будущего инсталлятора. Давай рассмотрим эти пункты более подробно:

Organize Your Setup – это имя раздела, в котором ты найдёшь подпункты основных параметров программы установки. Если щёлкнуть по этому пункту, то в левой половине окна ты увидишь исчёрпывающую информацию о подпунктах, что в них храниться и для чего они предназначены. Далее пойдёт описание подпунктов этого раздела.

General Information – здесь ты должен указать основные сведения о себе, как о разработчике, указать свой сайт в сети интернет и контактную информацию (рисунок 24.3.3). Давай подробно рассмотрим основные свойства, которые ты можешь изменить в этом пункте:

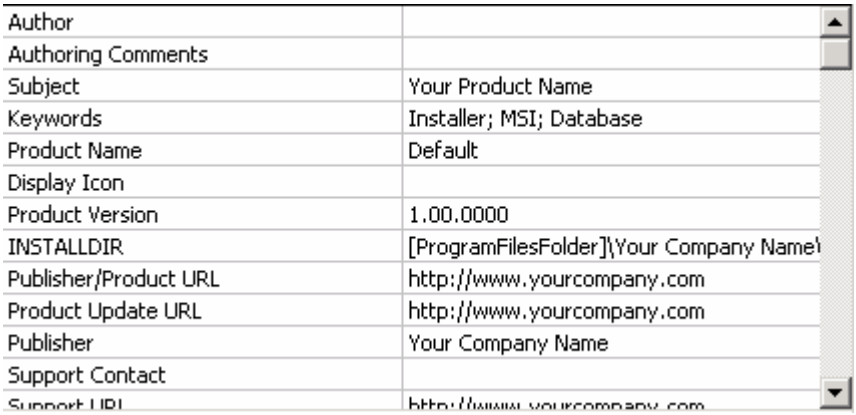


Рис 24.3.3. Свойства пункта General Information

1. *Author* – здесь нужно ввести имя автора программы. Введи своё имя или название своей компании.
2. *Authoring Comments* – Комментарии автора.
3. *Subject* – здесь указывается имя программы, которую нужно инсталлировать.
4. *Product Name* – имя продукта.
5. *Display Icon* – иконка программы.
6. *Product Version* – версия продукта.
7. *INSTALLDIR* – директория, в которую будет установлена программа. По умолчанию используется директория *[ProgramFilesFolder]\Your Company Name\Default*. Здесь *[ProgramFilesFolder]* указывает на то, что программа должна устанавливаться в папку Program Files на компьютере клиента, после чего указывается подкаталоги этой папки. В качестве примера предлагается ввести имя компании (*Your Company Name*).
8. *Publisher/Product URL* и *Product Update URL* – адрес в сети интернет, по которому можно найти программу.
9. *Publisher* – здесь опять укажи своё имя или название компании.
10. *Support Contact* – контактная информация со службой поддержки. Здесь указывается e-mail службы поддержки (твой e-mail).

Остальное – специфичные настройки, которые используются редко. Не советую тебе указывать в качестве службы поддержки свой телефон или реальный почтовый адрес, используй только e-mail.

Features – следующий раздел, в котором ты устанавливаешь возможности инсталятора. Если ты выберешь этот раздел, то в левой части окна увидишь нечто похожее на рисунок 24.3.4.

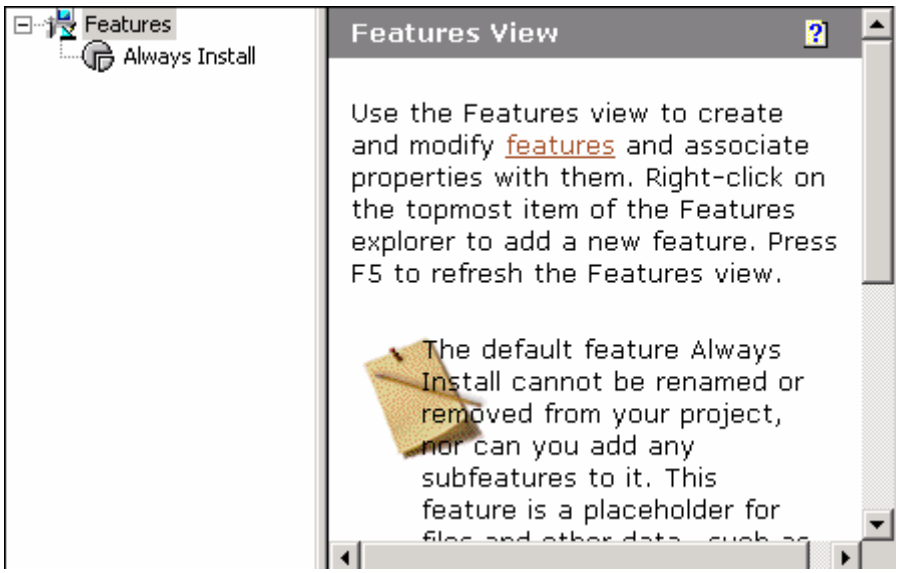


Рис 24.3.4. Свойства пункта Features

Здесь слева находиться дерево, в котором перечислены разные возможности инсталлятора. По умолчанию создана только одна возможность – *Always Install* (инсталлировать всегда). Щёлкни правой кнопкой по верхнему элементу или нажми клавишу Ins, чтобы создать новую возможность – назови её *Organizer*. Допустим, что мы создаём программу установки какой-нибудь программы. Все файлы этой программы мы поместим в пункт *Always Install*. У нашей программы будет и дополнительная возможность – органайзер, который пользователь должен уметь выбирать – устанавливать или нет.

Выбирая одну из возможностей ты можешь указать её свойства:

Description - описание типа установки.

Required – обязательность типа.

Visible – видимость. Для пункта *Always Install* – здесь указывается невидимость, потому что здесь будут указываться пункты, которые должны инсталлироваться всегда. Для пункта *Organizer* – который должен быть виден в окне выборочной установки здесь нужно ставить *Visible and Collapsed*, чтобы пользователь мог выбирать, устанавливать ему дополнительную возможность - *Органайзер* или нет.

Setup Types – типы установки. Если выбрать этот пункт, то в левой половине окна ты увидишь окно, похожее на рисунок 24.3.5. Слева окна находиться список разных типов установки: *Typical* (типичная), *Minimal* (минимальная), *Custom* (выборочная). У каждого пункта есть элемент управления ComboBox, в котором ты можешь выбрать – нужен этот тип установки или нет. По умолчанию во всех типах стоят галочки. Для нашего примера оставим всё так, как есть, пусть будет три типа установки.

Выделяя в левом списке тип установки, ты можешь в правом списке указывать то, что должно устанавливать в данном случае. При выборе *Typical* должно устанавливать все, поэтому в левом списке оставляем выделенными все пункты. При выборе типа *Minimal* должен устанавливаться необходимый минимум, т.е. органайзер не обязателен и с него снимаем галочку. При выборе *Custom* опять же оставляем всё выделенное, чтобы пользователь сам мог убрать то, что ему надо.

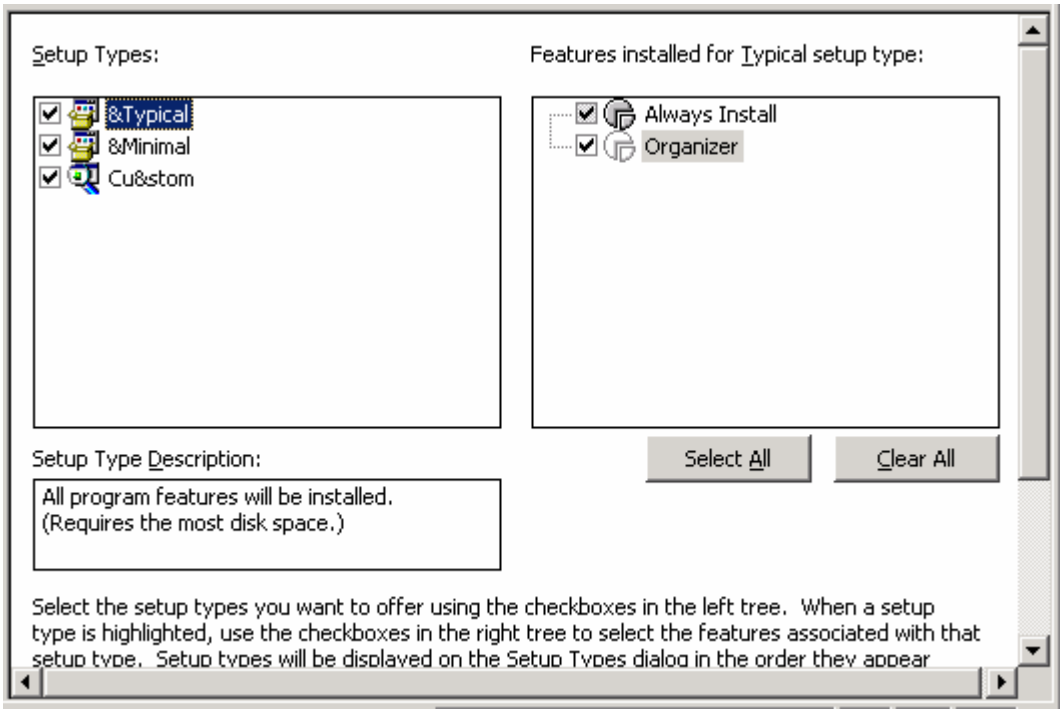


Рис 24.3.5. Свойства пункта Setup Types

Specify Application Data – это следующий раздел, в котором нужно указать какие файлы нужно устанавливать на компьютер клиента. Этот раздел состоит из следующих пунктов:

Files – здесь ты указываешь, какие файлы нужно устанавливать на компьютер клиента. Если выбрать этот пункт, то в левой половине окна ты увидишь рисунок 24.3.6.

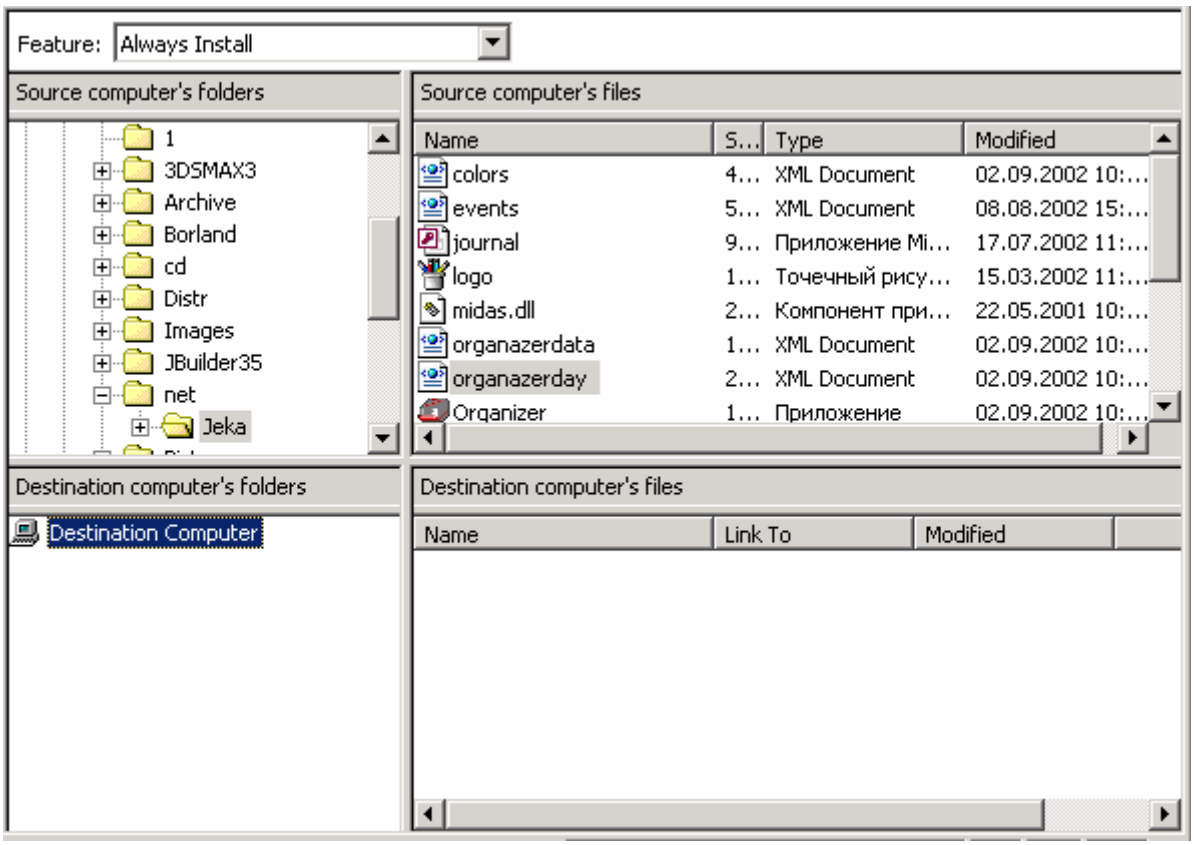


Рис 24.3.6. Свойства пункта Files

Сверху ты можешь выбирать в выпадающем списке тип установки. У нас должно быть там два пункта:

1. *Always Install.*
2. *Organizer.*

Чуть ниже слева находиться список дисков и директорий твоего компьютера. Выбирая директорию ты справа будешь видеть файлы выбранной папки. Перейди в директорию, где находятся файлы твоей программы, для которой ты создаёшь программу установки.

Внизу слева ты можешь видеть окно, в котором мы должны выбирать папку на компьютере клиента, в которую нужно устанавливать файлы. Щёлкни в этом окне правой кнопкой мыши по пункту *Destination Computer* и в появившемся меню выбери пункт *Show Predefined Folder* и затем выбери пункт *[INSTALLDIR]*. В дереве этого окна появиться соответствующий пункт. В этот пункт надо переместить основные файлы, которые надо поместить в папку, в которую устанавливается программа. Если какие-то файлы должны быть помещены в папку Windows, то щёлкни в этом окне правой кнопкой мыши по пункту *Destination Computer* и в появившемся меню выбери пункт *Show Predefined Folder* и затем выбери пункт *[WindowsFolder]*. В созданный пункт дерева можно переносить файлы, которые должны быть в папке Windows.

Теперь в выпадающем списке *Features* (сверху окна) выбери пункт *Organizer*. Снова создай в левом нижнем окне пункт *[INSTALLDIR]* и перенеси в него файлы органайзера.

Files and Features – здесь ты можешь просмотреть информацию о копируемых файлах в виде списка (рисунок 24.3.7).

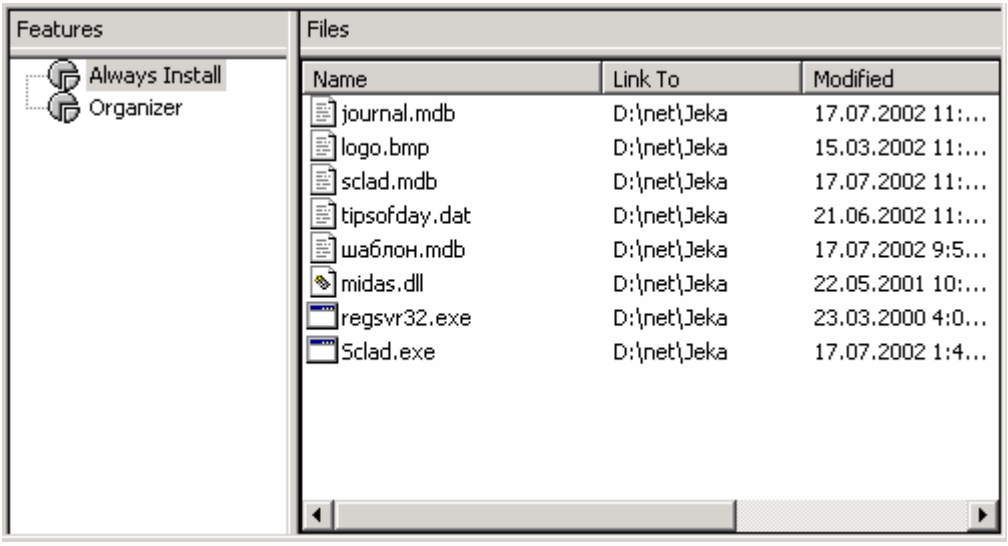


Рис 24.3.7. Свойства пункта Files and Features

Objects/Merge Modules – здесь ты можешь указать, какие модули нужно перенести на компьютер клиента. Выделив этот пункт, перед тобой откроется список установленных на твоём компьютере модулей, которые можно перенести на компьютер клиента (рисунок 24.3.8).

Допустим, что твоя программа использует базы данных MS Access. В этом случае на компьютере клиента должен быть установлена надстройка DAO. Твоя программа установки может автоматически перенести эту надстройку на компьютер клиента. Достаточно только найти её в списке левого верхнего окна и поставить напротив этой строки галочку. Программа сама определит файлы, которые надо скопировать на компьютер клиента и при установке внесёт необходимые изменения в реестр.

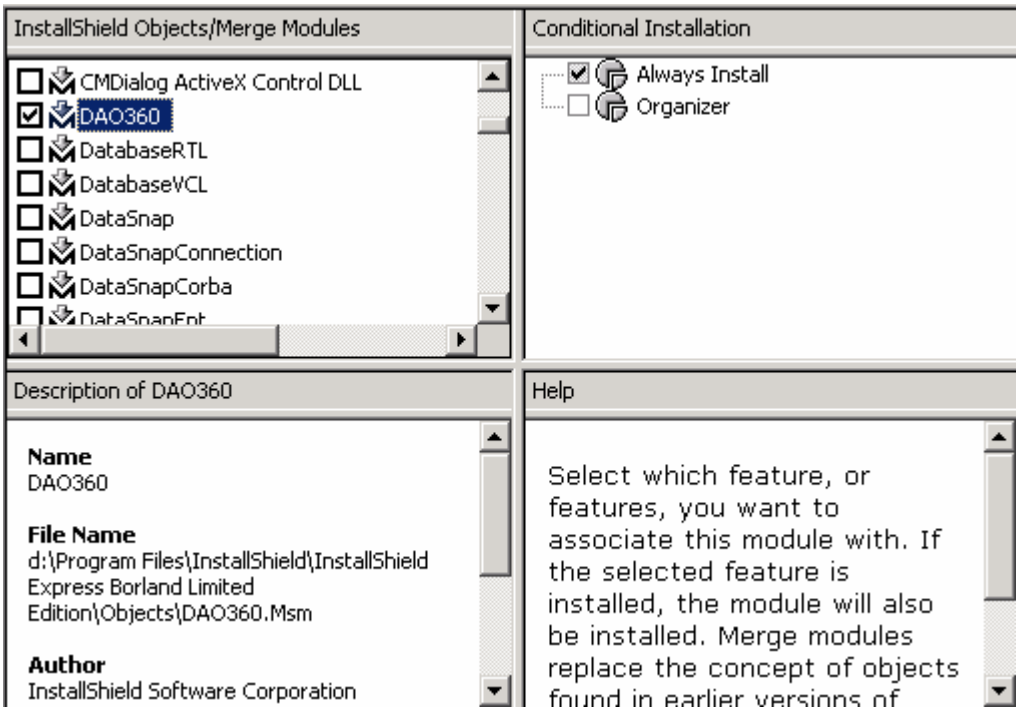


Рис 24.3.8. Свойства пункта Objects/Merge Modules

Если тебе необходимо установить на компьютер клиента какой-нибудь компонент ActiveX, то опять же его можно найти в этом списке. Созданная программа установки

сама зарегистрирует этот компонент во время инсталляции и ты избавишься от всех проблем по установке и регистрации ActiveX компонентов.

Configure the Target System – это следующий раздел, в котором ты можешь указать, какие изменения надо произвести на компьютере клиента.

Shortcuts/Folders – здесь мы указываем ярлыки, которые нужно создать в программном меню для вызова программ. Выдели этот пункт и ты увидишь в правой половине окна нечто похожее на рисунок 24.3.9.

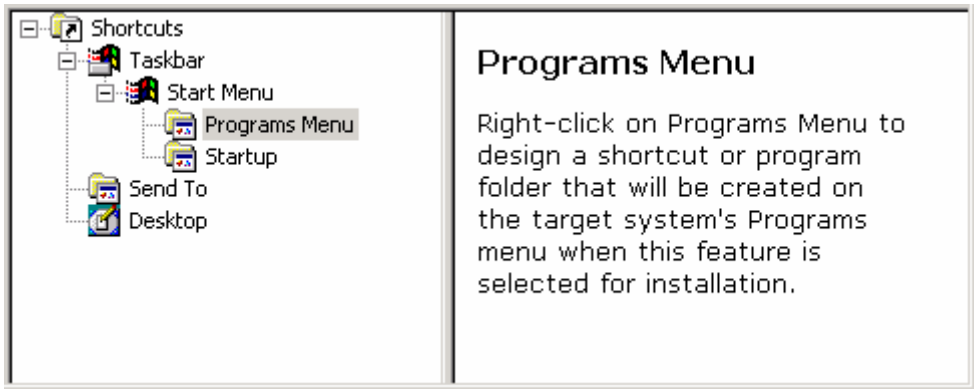


Рис 24.3.9. Свойства пункта Shutcuts/Folders

Если ты хочешь создать ярлык для вызова программы из меню *Пуск*, то выдели пункт *Program Menu* и щёлкни по нему правой кнопкой. В появившемся меню выбери пункт *New Folder*, чтобы создать отдельную папку для своей программы. Теперь выдели эту папку и щёлкни правой кнопкой по ней. Здесь выбери пункт *New Shortcut*, чтобы создать ярлык для программы.

Выделив имя созданной иконки ты увидишь следующие её свойства:

Description – описание программы.

Feature – когда создавать ярлык. По умолчанию стоит «Всегда», но ты можешь выбрать пункт *Organizer*, если иконку нужно создавать только если пользователь требует установку органайзера.

Arguments – здесь указываются параметры, которые нужно передавать программе.

Target – файл, который надо запускать при выборе этой иконки. Здесь нужно указывать имя запускного файла, относительно компьютера клиента, например *[INSTALLDIR]\organizer.exe*.

Icon File – файл иконки для ярлыка.

Icon Index – если у программы есть своя иконки, то ты можешь указать её индекс. Например, если здесь указать 0, то будет использоваться первая иконка программы.

Run – здесь указываются параметры запуска программы. По умолчанию стоит нормальный запуск – *Normal Window*.

Working Directory – рабочая директория программы. Здесь так же нужно указывать директорию относительно компьютера клиента, например, *[INSTALLDIR]*.

Registry – здесь ты можешь указывать изменения, которые надо произвести в реестре. Окно разделено на две части: в верхней ты видишь свой текущий реестр, снизу ты видишь изменения, которые надо произвести.

Для создания нового параметра, который нужно будет создать на машине клиента ты должен щёлкнуть правой кнопкой по нужному разделу и в появившемся меню выбрать пункт *Key* для создания нового ключа. В этом ключе ты можешь создать ещё один ключ или параметр (снова щёлкнув правой кнопкой).

ODBC Resources – здесь ты можешь увидеть в виде дерева все установленные к тебя ODBC драйвера. Эти драйвера используются для доступа к базам данных с через

компоненты ADO. Мы в этой книге использовали MS Jet драйвер, который относиться к DAO, но тебе могут понадобиться и ODBC драйвера. Если нужен такой драйвер, то найди его в левом верхнем окошке и поставь галочку напротив имени.

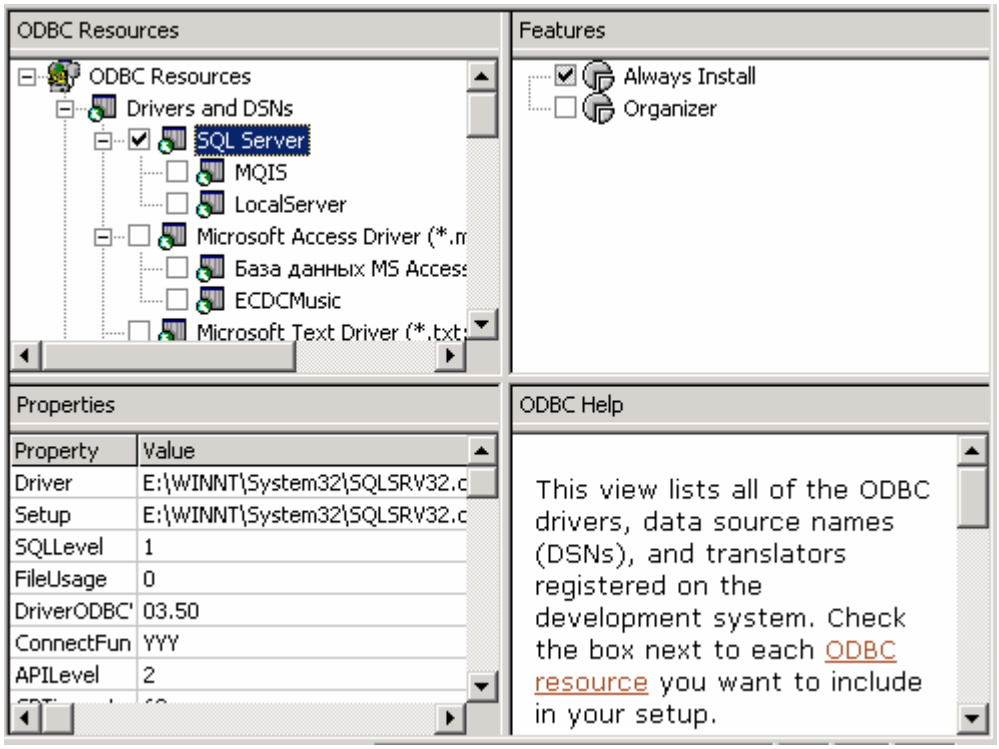


Рис 24.3.10. Выбор ODBC драйвера

Как только ты выбрал ODBC драйвер, так сразу активизируются правое верхнее и левое нижнее окна. В правом верхнем ты можешь указать в каких типах установки нужно устанавливать этот драйвер. В левом нижнем видны свойства драйвера.

INI File Changes – здесь ты можешь указать, какие изменения нужно произвести в ini файлах. Я эти фалы не использую по причине их устарелости и тебе не советую. Поэтому я не буду останавливаться на этом пункте и двинусь дальше.

File Extensions – здесь ты указываешь, какие расширения нужно зарегистрировать под твою программу. Выбери этот пункт и слева появиться окно, как на рисунке 24.3.11.

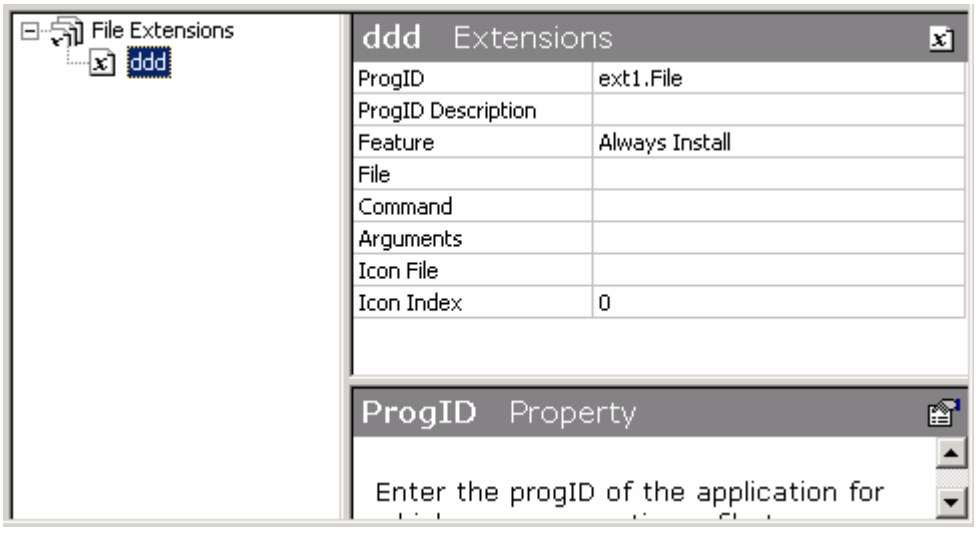


Рис 24.3.11. Окно регистрации нового расширения

Чтобы создать новое расширение, щёлкни правой кнопкой по пункту *File Extensions* в дереве слева. В дереве появиться новый пункт в котором надо ввести имя расширения. В правой половине окна ты должен в свойстве *File* указать имя твоей программы, которая должна запускаться. Ты так же можешь указать дополнительные аргументы программы и иконку, которая будет отображаться для всех файлов этого типа.

Customize the Setup Appearance – в этом разделе находятся пункты настроек, в которых ты можешь указать, как должна выглядеть программа инсталляции.

Dialogs – здесь ты указываешь, какие диалоги должны быть видны во время инсталляции. Выбери этот пункт и ты увидишь окно, как на рисунке 24.3.12. В левом верхнем окне находиться дерево, в котором перечислены все доступные диалоги. В левом нижнем окне ты сможешь видеть примерный внешний вид выделенного окна. В правом верхнем окне будут отображаться свойства, которые ты можешь изменить.

Slash Bitmap – если напротив этого пункта стоит галочка, то вначале загрузки будет отображаться рисунок. В свойстве *Splash Bitmap* этого окна ты можешь указать картинку, которая должна отображаться в окне.

Install Welcome – окно приглашения в программу установки. Это окно невозможно отключить и оно обязательно должно присутствовать в программе установке. Ты можешь только изменять свойства окна:

1. Bitmap Image – картинка, которая будет отображаться в окне.
2. Show Copyright – показывать строку Copyright.
3. Copyright Text – текст строки Copyright.

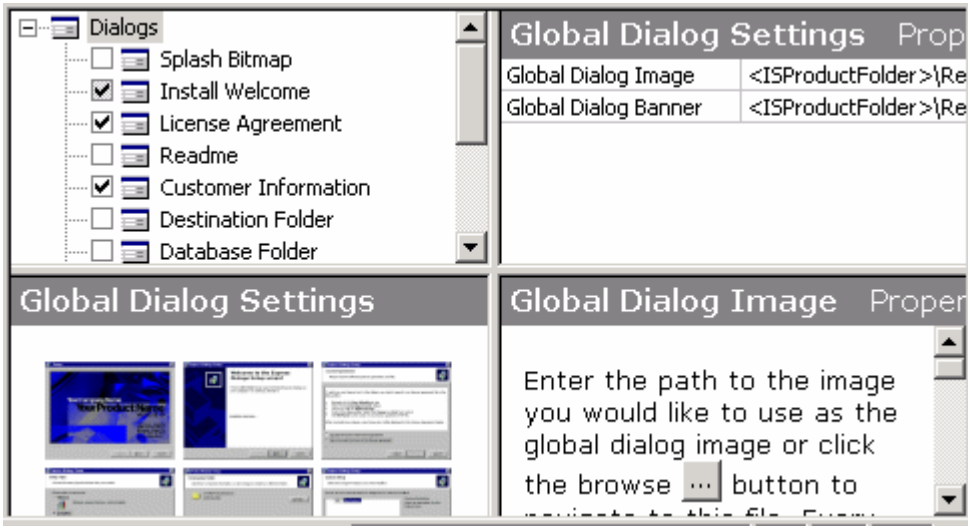


Рис 24.3.12. Окно регистрации нового расширения

License Agreement – окно лицензии. У этого окна два свойства:

1. Banner Bitmap – картинка иконки.
2. License File – файл с текстом лицензии.

Readme – окно с дополнительной текстовой информацией об устанавливаемой программе. У этого окна есть два свойства:

1. Banner Bitmap – картинка иконки.
2. Readme File – файл с текстом.

Customer Information – окно, в котором пользователь должен вводить данные о себе – имя, название компании. У этого окна много свойств и все они достаточно интересны:

1. Banner Bitmap – картинка иконки.
2. Show Serial Number – показывать строку для ввода серийного номера продукта.
3. Serial Number Template – шаблон серийного номера. Здесь ты можешь указать шаблон, по которому будет вводиться серийный номер. Например, у тебя серийный номер состоит из двух чисел (каждое по три цифры) между которыми стоит знак тире. В этом случае шаблон будет выглядеть так «###-###». Если вначале должны быть какие-то обязательные буквы, например «*sernum*», то шаблон можно записать так: «sernum###-###».
4. Serial Number Validation DLL – здесь можно указать DLL файл, который будет отвечать за проверку правильности введённого серийного номера.
5. Validate Function – здесь указывается функция из библиотеки, которая будет проверять серийный номер.
6. Success Return Value – значение, которое должно возвращаться при положительном результате проверки.
7. Retry Limit – количество попыток, которые пользователь может ввести при регистрации.
8. Show All Users Option – показать возможность выбора, для каких пользователей будет доступна программа. В Windows NT/2000/XP есть возможность давать доступ к программе только тому пользователю, который её устанавливает или всем пользователям компьютера.

Destination Folder – показывать окно выбора папки, в которую надо устанавливать программу.

Database Folder – окно выбора папки, куда будут устанавливаться базы данных.

Setup Type – окно выбора типа установки. В этом окне пользователь сможет выбирать, какая установка ему нужна – типичная, минимальная или выборочная.

Custom Setup – окно, в котором будут отображаться компоненты твоей программы, если пользователь выбрал выборочную установку. Если у этого окна в свойстве Show Change Destination установить true, то пользователь сможет менять директорию установки из этого окна.

Ready to Install – окно, предупреждающее о начале копирования файлов.

Setup Progress – показывать окно хода установки. Если у этого окна в свойстве Show Progress Bar установить false, то пользователь не будет видеть Progress Bar, в котором отображается ход установки.

Setup Complete Success – окно окончания установки. У этого окна целая куча интересных свойств:

1. Banner Bitmap – картинка иконки.
2. Show Launch Program – показывать возможность запуска программы по окончанию процесса установки.
3. Program File – имя файла программы, которую надо запустить.
4. Command Line Parameters – параметры командной строки, которые надо передать программе.
5. Show Readme – показывать текстовый файл с дополнительной информацией.
6. Readme file - текстовый файл с дополнительной информацией.

Define Setup Requirements and Actions – в этом разделе мы будем делать последние настройки нашей программы установки.

Requirements – здесь можно указать ОС, тип процессора, количество памяти и др. параметры необходимые твоей программе. Программа установки при старте будет проверять эти параметры и если что-то не совпадает, то установка не пройдёт.

Prepare For Release – в этом разделе ты создаёшь программу установки.

Build You Release – здесь ты должен выбрать носитель, на котором будет распространяться твоя программа. В зависимости от типа носителя программа установки может отличаться. Для запуска процесса создания программы установки нужно нажать F7 или выбрать из меню *Build* пункт *Build*.

Test You Release – тут можно протестировать созданную тобой программу установки.

Distribute You Release – в этом разделе ты можешь скопировать программу установки на носитель.

24.4. Как писать и распространять Shareware программы

Сегодня я решил рассказать тебе о том, как писать программы и что именно надо писать. Многие программисты говорят: "Дайте мне точку опоры, и я переверну весь мир". Где-то я это уже слышал, и возможно это сработает, но не в нашем случае. Самое главное при написании программы это не идея. А что же можно назвать главным фактором успеха программного продукта? Вот именно это я постараюсь тебе сегодня рассказать.

Какую программу всё же написать? Для этого не надо далеко идти. Просто подумай, что тебе нужно? Если ты работаешь с базами данных, то займись написанием этих баз. Если ты работаешь с графикой, то напиши простенькую программу, которая будет делать несколько крутых выкрутасов или эффектов. Твоя будущая программа должна удовлетворять нескольким критериям:

1. Программа должна быть нужна тебе. Если она не нужна даже разработчику, то ею не будет пользоваться никто. Если ты дизайнер, то кто, как не ты лучше знает, что нужно настоящему художнику? Вот я, например, в графике не особо смыслю. Я умею её программировать, но ничего не понимаю в художествах. Поэтому я никогда не лезу в эту сферу, хотя мне очень хочется и нравится работать с различными алгоритмами.

2. Программа должна быть уникальна. Если в твоей программе есть какая-то изюминка, то она точно найдёт своего пользователя.

3. Она должна быть простой в использовании. Не стоит загромождать программу лишними действиями. Если ты гений в графических фильтрах, то напиши простенький plug-in к PhotoShop, но не надо писать целый графический редактор ради одного эффекта. Это только усложнит твоё детище и отпугнёт потенциальных пользователей. Программа должна выполнять только самые необходимые действия и содержать как можно меньше лишних функций. Поэтому не стоит встраивать в графический редактор, текстовый процессор, такие большие продукты уже есть и ты всё равно не сможешь с ними конкурировать. Чем проще пользоваться программой, тем больше у неё шансов. Вот ещё один пример: CyD GIF Studio Pro - отличный GIF редактор, но в нём очень много функций, которыми не все пользуются, поэтому многие переходят на более дорогую

программу, но выполняющую только самое необходимое. Это принцип жизни всех американцев и европейцев. Я этого не понимаю, но приходится к нему прислушиваться. Для меня лучше взять программу немного более сильную и дешевле, чем дороже, но менее функциональную. Приведу ещё один пример: Linux - дешёвая и навороченная, поэтому она не получила распространения, а Windows простая и дорогая и стоит на большинстве компьютеров. Парадокс, но это так.

4. Интерфейс - должен быть удобным и симпатичным. Цвета желательно выбирать не сильно яркие, чтобы они не резали глаз. На счёт цвета, я бы вообще-то посоветовал бы использовать только системные, чтобы они изменялись в зависимости от выбранной в компьютере цветовой схемы. Все часто используемые функции, нужно выносить из меню на панель, чтобы можно было получить к ним быстрый доступ. Короче, посмотри на все программы Windows и старайся не сильно выделятся, а то это иногда шокирует и люди не очень охотно используют такие вещи.

5. Документация - она должна быть полной и желательно с конкретными примерами. Пользователи любят, когда описаны конкретные действия при работе с программой. Не надо ограничиваться простым описанием возможностей. Постарайтесь дать как можно больше информации и конкретных примеров.

Этот список можно продолжать бесконечно, но я остановлюсь. В течении всей статьи я ещё вернусь к уже описанным требованиям и укажу ещё некоторые вещи.

Прежде чем писать программу, поставьте перед собой конкретные цели. Главное, чтобы твоя цель была достижима и как можно в кратчайшие сроки. Не надо ставить перед собой задачу написать текстовый редактор, потому что их уже достаточно, ты не напишешь лучше чем Microsoft, но даже если это и так, то пока ты будешь писать, Microsoft выпустит уже десять версий. Ты не сможешь угнаться за гигантами. Поэтому делай свою цель менее фантастической.

Я уже говорил, что желательно сделать программу маленькой. Для этого есть несколько причин:

1. Пользователи не любят слишком навороченные программы. Об этом я уже сказал и привёл несколько примеров.

2. Первая версия твоей программы должна быть готова максимум через месяц. Если ты затянешь написание программы на год, то через этот период может пропасть необходимость в ней или кто-то уже реализует твою идею, и ты потратишь время зря.

Именно поэтому ставь реальные и быстро достижимые цели.

После того, как ты поставил перед собой цель, начинай писать программу. Во время написания нужно чётко придерживаться поставленной цели. Не в коем случае нельзя обращать внимание на мысли: "надо бы добавить вот такую возможность или вот такую". Если ты хоть раз обратишь на неё внимание и начнёшь выполнять, то тебя затянет. Ты будешь вечно добавлять новые возможности и так и не выпустишь свою программу в свет. А если программа и появиться, то в недоработанном варианте, потому что ты просто выбросишь промежуточный вариант, в надежде, что в ближайшее время будет полный, а сам снова уйдёшь в постоянные доработки. Лучше записывай появляющиеся мысли на бумагу и засовывай подальше в карман.

Когда ты закончишь выполнять первоначальный план, то протестируй готовую программу и начинай заниматься раскруткой (об этом я расскажу чуть ниже). На своём сайте можешь указать всё, что у тебя накопилось на бумажках (то что ты собираешься добавить) и указать, что это можно увидеть в следующей версии программы. Потенциального пользователя это заинтересует, и он запомнит ссылку на твою страничку среди "Избранных", чтобы вернуться за новой версией с обещанными возможностями.

Пока все работают с твоей первой версией программы, ты достаёшь из кармана все свои мысли и начинаешь их выполнять, оформляя новую версию.

Постарайся хорошенечко тестировать свою работу. Для этого можно набрать группу тестеров среди российских пользователей. Для этого раздай им бесплатные лицензии, они всё равно не заплатят (в России ещё не привыкли платить за программы). Чем больше будет ошибок в твоей программе, тем хуже будут к тебе относится. Я в своё время очень сильно запустил своё детище и ко мне потеряли интерес все пользователи. После этого мне понадобилось два дня, чтобы исправить все ошибки и пол года, чтобы снова завоевать доверие. Так что выводы делай сам.

Я постарался дать тебе самые основные (на мой взгляд) правила написания программы. Не думай, что что-то из этого можно не учитывать, в нашем деле важно всё, особенно при работе с иностранными клиентами. Так что выучи наизусть всё, что я тебе рассказал.

Где размещать

Твоя программа готова и теперь её надо где-то разместить, чтобы потенциальные клиенты могли скачать твоё творение. Для этого опять же полно серверов бесплатно предоставляющих место для твоих страничек. Конечно же, это не самый лучший шаг, потому что в цивилизованных странах не очень любят пользоваться такими страницами. Но для тебя этого будет достаточно. Ты всё равно сможешь заработать свои кровные даже при использовании бесплатного хостинга.

Советую выбирать тебе сервер заграницей, чтобы ничего в его адресе не указывало на происхождение твоей программы. Желательно, чтобы никто не знал, что ты русский. Твой почтовый ящик тоже должен быть не в зоне RU. И я надеюсь, что твоя страничка сделана без ошибок в английском языке. Если ошибки есть, то твои шансы заработать падают на 99%.

Если не можешь писать на английском, то сделай перевод с помощью любого электронного переводчика и отправляйся сюда: *http://members.home.net/djosborne1/*. Здесь тебе за небольшую плату исправят все ошибки. Воспользуйся этим, не жалей сотню долларов. Потрать их и ты уже через месяц сможешь ощутить живую прибыль, если твоя программа хоть чего-то стоит.

Я немного отклонился от темы. Вернёмся к нашим серверам. Вот тебе несколько адресов, которые тебе помогут.

http://www.crosswinds.net/ - хороший сервер, но до него очень трудно добиться, чтобы закачать файлы. Скорость закачки ужасная, меньше 1 кило в секунду. Зато скорость скачивания моментальная.

http://www.webjump.com/ - добиться легко, скорость закачки хорошая, но вешают большой банер. Есть специальная поддержка программистов распространяющих Shareware программы, таких как ты.

http://www.freeservers.com/ - Дают мало места. Я им не пользовался и больше ничего сказать не могу.

http://www.tripod.com/ - говорят хороший, сам не пробовал, поэтому утверждать не буду.

http://www.virtualave.net/ - третий сорт не уродство. Полно возможностей и достаточно много места.

Как рекламировать

После того, как ты выложил свою программу в глобальной сети, её надо начать рекламировать. Я тебе советую зайти на следующий адрес: http://download.cnet.com/. Зайди туда и жми на ссылку *Submint file*. Перед тобой появится простая форма для описания программы, которую надо заполнить (Рисунок 24.4.1).

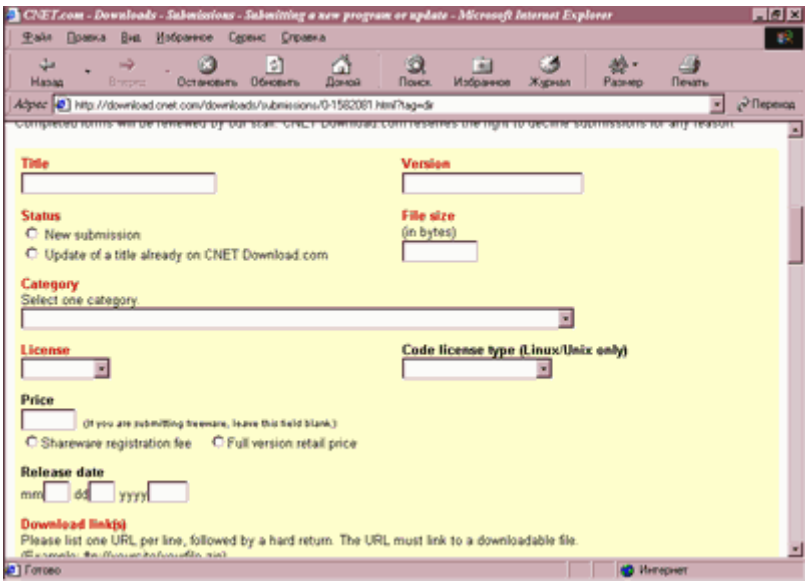


Рис 24.4.1. Форма регистрации программы на www.download.com

Заполни эту форму сведениями о своей программе и через некоторое время на этом сервере будет вывешено её описание. На момент написания книги, время обновления сведений сильно увеличилось (до 2-х месяцев), а моментальное обновление доступно за деньги. Если есть лишние деньги, то можешь потратить их на обновление информации. Я уже говорил, что скорость в нашем деле очень важна, но и про качество забывать не надо.

Только на *download.com* твою программу будут скачивать по 100 копий в день. Только не думай, что на этом сервере будет располагаться твоя программа. Её будут качать с твоей страницы, а *download.com* хранит только описание и ссылку на твой файл.

Если ты всё правильно сделал и программа оказалась на download.com, то через некоторое время другие архивы программ будут сами делать ссылку на твоё детище. Так что можно не напрягать свои мозги регистрацией на всех серверах подряд. Тем более, что только download.com даёт ощутимый трафик, остальные смогут дать только 5-10 скачиваний в день (по моему личному наблюдению).

Как получать деньги

Твоя программа готова, выложена в интернете и её качают. Возможно, что её скоро захотят купить. Ты должен заранее приготовиться к этому моменту. Сам ты не сможешь получать деньги от клиентов. Как же тогда решить эту проблему? Есть несколько серверов, которые предоставляют услуги по оплате счетов через интернет, почту и даже телефон. Ты просто регистрируешься у них, и они сами будут получать деньги с твоих клиентов. Берут они за это немного, всего 9-15 процентов. Не надо жадничать, это действительно очень мало. Воспользуйся этими услугами.

Я могу посоветовать тебе использовать *www.regnow.com*. Он очень удобен, а главное прост в использовании. После регистрации тебе приходит письмо, в котором находятся логин и пароль доступа. Когда ты войдёшь в свою зону, ты сможешь легко разобраться с работой сервера. Просто попробуй, там всё понятно, если ты хоть немного понимаешь английский язык или умеешь пользоваться переводчиками.

24.5. Создание программ маленького размера

Очень часто нам приходиться задумываться о написании программ маленького размера. Если при написании офисных приложений мы можем забыть про оптимизацию размера, то для программ постоянно находящихся в памяти размер кода критичен. Допустим, что тебе надо написать маленькую утилиту – будильник. Эта программа должна постоянно находиться в памяти и следить за временем. Я думаю, что будет неприятно, если программа будет занимать мегабайт в оперативной памяти на какой-то будильник.

Программы созданные Delphi получаются достаточно большого размера. С чем это связано? А с тем, что Delphi является объектным языком. В нем каждый элемент выглядит как объект, который обладает своими свойствами, методами и событиями. Любой объект вполне автономен и может работать без твоего ведома. Это значит, что тебе нужно только подключить его к своей форме, изменить нужные свойства и все готово. После этого все будет работать без какого-либо внешнего вмешательства.

Но в этом есть и свои недостатки. В объектах реализовано большинство возможных действий, которые ты можешь производить с ним. Но реально, в любой программе мы пользуемся двумя-тремя из всех этих свойств. Все остальное для программы лишний груз, который никому не нужен.

Но как же тогда создать компактный код, чтобы мой будильник занимал минимум места на винте и как можно меньше занимал памяти? У тебя есть несколько вариантов:

1. Не использовать визуальную библиотеку VCL (для любителей Visual C++ это библиотека MFC), которая упрощает программирование. В этом случае придется все делать вручную и работать только с WinAPI. Код в этом случае получается очень маленьким и быстрым. Но тут ты лишаешься визуальности и можешь ощутить все неудобства программирования на чистом WinAPI.

2. Сжимать программы с помощью компрессоров. Такой код сжимается в несколько раз и программа с использованием VCL может превратиться из 300 кило в 30-50. Главное преимущество тут в том, что ты не лишаешься возможностей объектного программирования и можешь спокойно забыть про неудобства WinAPI.

Здесь я постараюсь, как можно подробнее рассмотреть оба этих метода.

Программирование на WinAPI

Если ты хочешь создать программу маленького размера, то ты должен забыть про все удобства. Ты не сможешь подключать визуальные формы или другие примочки написанные фирмой Barland для упрощения жизни программиста. Только API функции и ничего больше.

Для того, чтобы создать маленькую программу в Delphi, нужно создать новый проект и зайти в менеджер проектов (меню *View->Project Manager*). Здесь нужно удалить все формы, чтобы остался только файл самого проекта (по умолчанию его имя Project1.exe). Никаких модулей в проекте не должно быть.

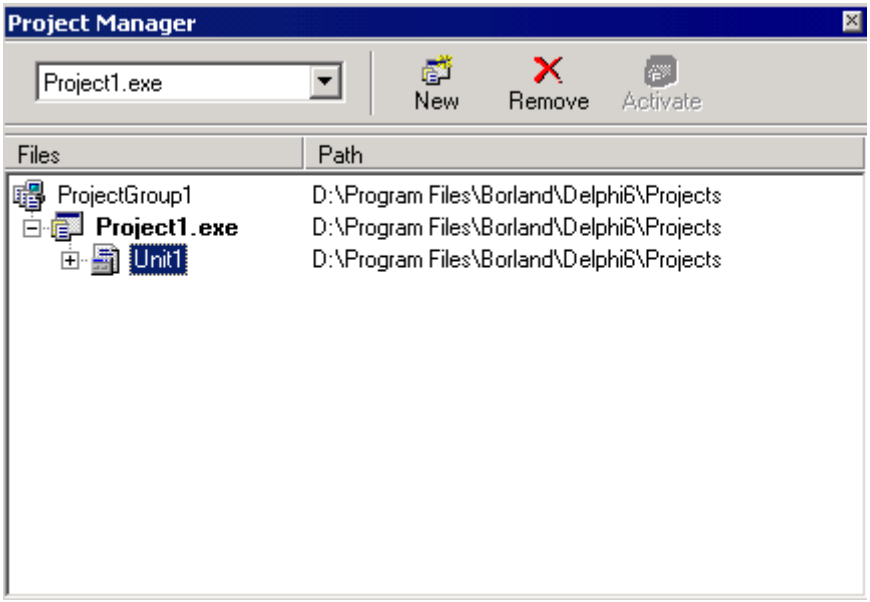


Рисунок 24.5.1. Project Manager

Теперь щелкни правой кнопкой по имени проекта и выбери из появившегося меню пункт *View Source* (или из главного меню *Project* выбери пункт меню *View Source*). В редакторе кода откроется для редактирования файл проекта Project1.*dpr*. Если ты уже удалил все модули, то его содержимое должно быть таким:

```
program Project1;

uses
  Forms;

{$R *.res}

begin
  Application.Initialize;
  Application.Run;
end.
```

Я удалил все визуальные формы и теперь могу скомпилировать абсолютно пустой проект. Я решил попробовать сделать это. После компиляции я выбрал из меню Project пункт *Information for Project1*. Передо мной появилось окно с информацией о проекте. Моё окно ты можешь увидеть на рисунке 24.5.2.

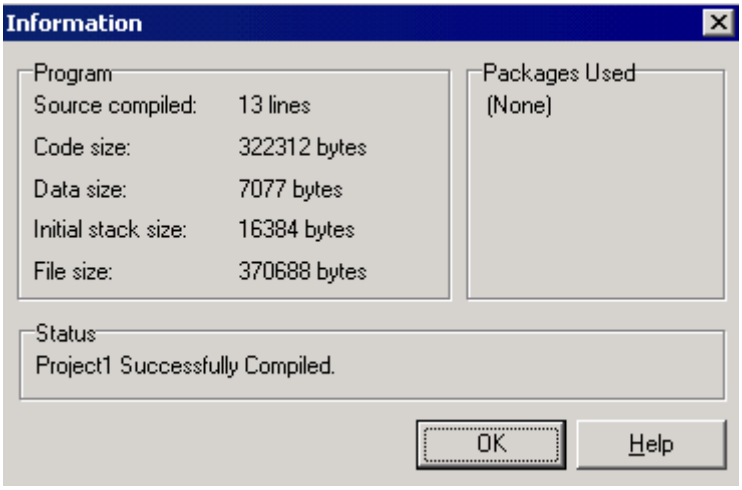


Рисунок 24.5.2. Окно информации о проекте

В правой части окна должны быть описаны используемые пакеты. Мы все удалили, значит там точно должна красоваться надпись *None*. А вот с левой стороны должна быть описана информация о скомпилированном коде. Самая последняя строка показывает размер файла и у меня он равен 370688 байт. Ничего себе пельмень!!! Мы же ничего еще не писали. Откуда же тогда такой большой код?

Давай разберем, что осталось в нашем проекте, чтобы обрезать все, что еще не обрезано. Сразу обрати внимание, что в разделе **uses** подключен модуль *Forms*. Это объектный модуль, написанный дядей Борманом, а значит, его использовать нельзя, потому что именно он увеличивает размер нашей программы. Между **begin** и **end** используется объект *Application*. Его тоже использовать нельзя, потому что это объект.

Все накладки большого кода, даже у пустой программы, как раз и связаны с объектом *Application*, который объявлен в модуле *Forms*. Хотя мы используем только два метода *Initialize* и *Run*, при компиляции в запускной файл попадает весь объект *TApplication*, а он состоит из сотен, а может и тысяч строчек кода.

Чтобы избавится от накладных расходов нужно заменить модуль *Forms* на *Windows*. Второй – это модуль который описывает только WinAPI и не связан с объектами Delphi. Его подключение обязательно, иначе мы не сможем вызвать ни одной функции из набора WinAPI. А между **begin** и **end** вообще все можно удалять. Самый минимальный код проги будет выглядеть так:

```
program Project1;

uses Windows;

Begin

end.
```

Снова откомпилируй проект. Зайди в окно информации и посмотри на размер получившегося файла. У меня получилось 8192 байта (смотри рисунок 24.5.3). Вот это уже по человечески.

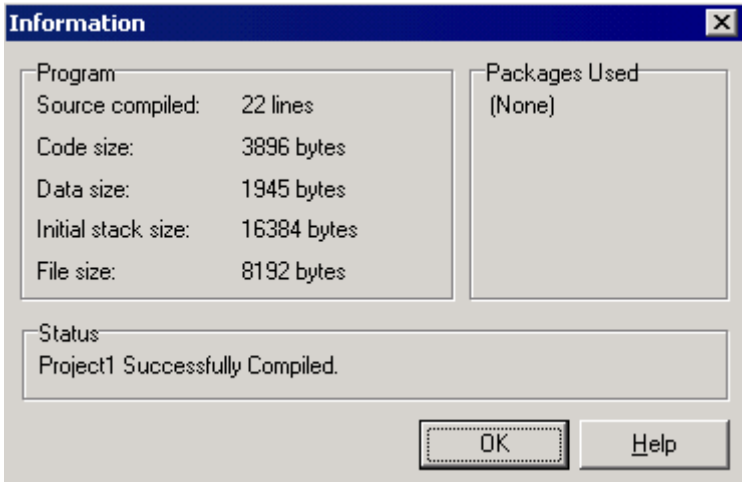


Рисунок 24.5.3. Информации о проекте

Заготовка минимальной программы с использованием WinAPI готова. Теперь ты можешь смело добавлять свой код. Мне только надо объяснить тебе какие модули можно подключать к своему проекту в раздел **uses**. Тут все очень просто и не займет много времени.

Если при установке **Delphi** ты не отключал копирование исходников библиотек, то перейди в директорию, куда ты установил Delphi. Здесь перейди в папку *Source*, затем в *Rtl* и наконец *Win*. Вот здесь расположены исходники модулей, в которых описаны все API функции Windows. Именно эти модули ты можешь подключать к своим проектам, если хочешь получить маленький код. Если ты подключишь что-то другое, то я уже не гарантирую тебе минимум размера твоей программы (хотя есть и исключения).

Сразу же пример. Если ты хочешь, чтобы в твоей программе были возможности работы с сетью, то тебе нужно подключить к нему библиотеку сокетов. Среди модулей WinAPI есть файл с именем *winsock.pas*. Значит, ты должен в раздел **uses** написать *winsock* (расширение писать не надо) и твоя программа сможет работать с сетью.

Пока что я описал минимальный проект, в который можно добавлять свой код. Но код, который ты вставишь, выполниться один раз и программа выгрузится из памяти. А что если тебе надо, чтобы твоя программа постоянно висела в памяти и что-то делала? Для этого используй следующий шаблон для своих программ:

```
program Project1;

uses Windows;

var
  Msg: TMsg;
Begin

//Сюда можешь добавлять свой код

// Дальше идет код, который заставит прогу висеть в
// памяти вечно и не будет сильно грузить систему.
    while GetMessage( Msg, HInstance, 0, 0) do
     begin
       TranslateMessage(msg);
       DispatchMessage(msg);
     end;
end.
```

Сжатие программ

Если тебя не устраивает программирование на чистом WinAPI, то твоим лучшим другом должна стать какая-нибудь программа для сжатия размера файлов. Я очень люблю ASPack, которую ты можешь забрать по адресу www.cydsoft.com/vr-online/download.htm или на компакт диске в директории *Programs* (файл установки называется ASPack.exe). Она прекрасно сжимает запускные файлы .exe и динамические библиотеки .dll.

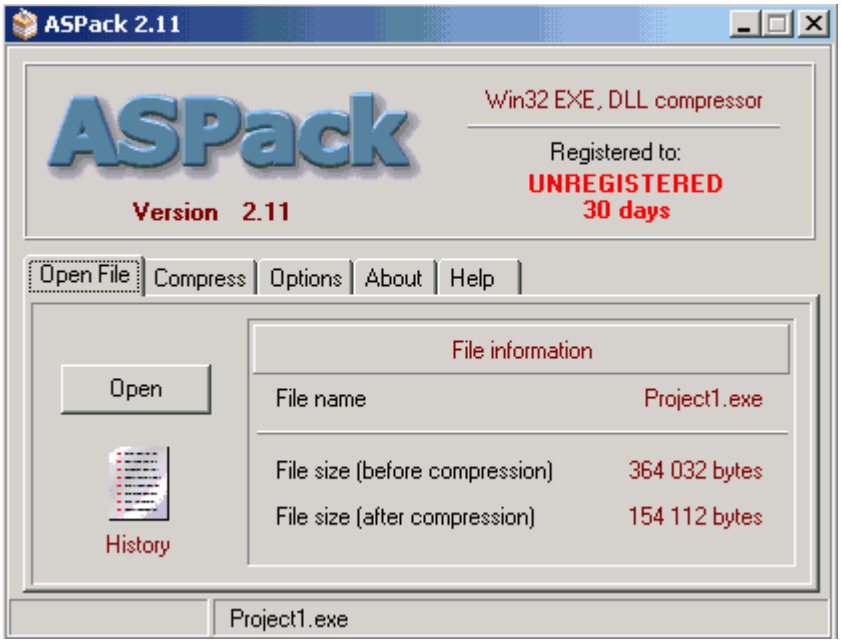


Рисунок 24.5.4. Главное окно ASPack

Я не буду объяснять установку ASPack, потому что там абсолютно ничего сложного нет. Только одно нажатие на кнопке *Next* и все готово. Теперь запусти установленную программу и ты увидишь окно, как на рисунке 24.5.4. Главное окно состоит из нескольких закладок:

1. Open File.
2. Compress.
3. Options.
4. About.
5. Help.

На закладке *Open File* есть только одна кнопка - *Open*. Нажми на нее и выбери файл, который ты хочешь сжать. Как только ты выберешь файл, программа перескочит на закладку *Compress* и начнет сжатие (рисунок 24.5.5).

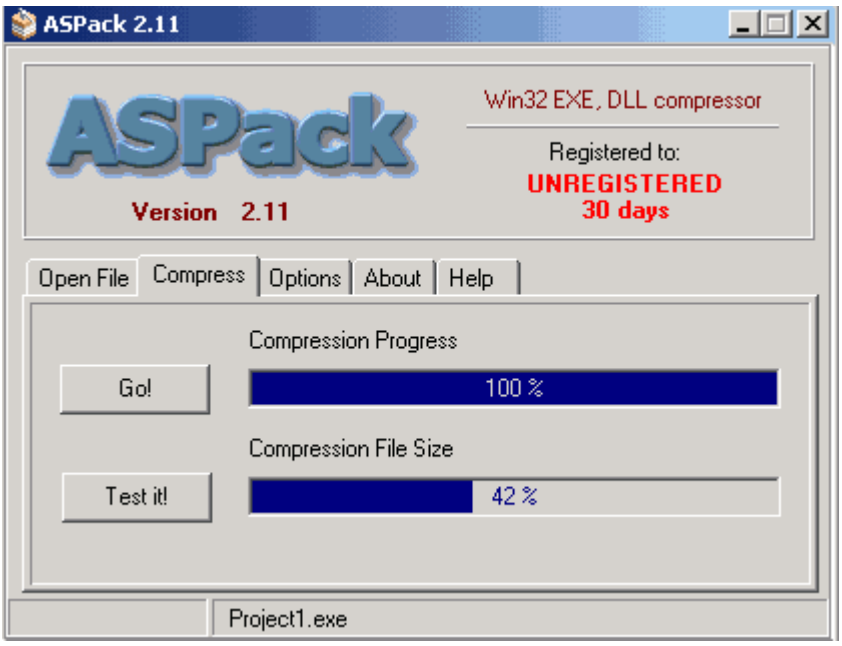


Рисунок 24.5.5. Сжатие файла

Сжатый файл сразу перезаписывает существующий, а старая несжатая версия сохраняется на всякий случай под тем же именем, только с расширением *bak*. Настроек у ASPack не так уж много (рисунок 24.5.6), и с ними ты сможешь разобраться без меня. Уж лучше я расскажу тебе, как все работает.

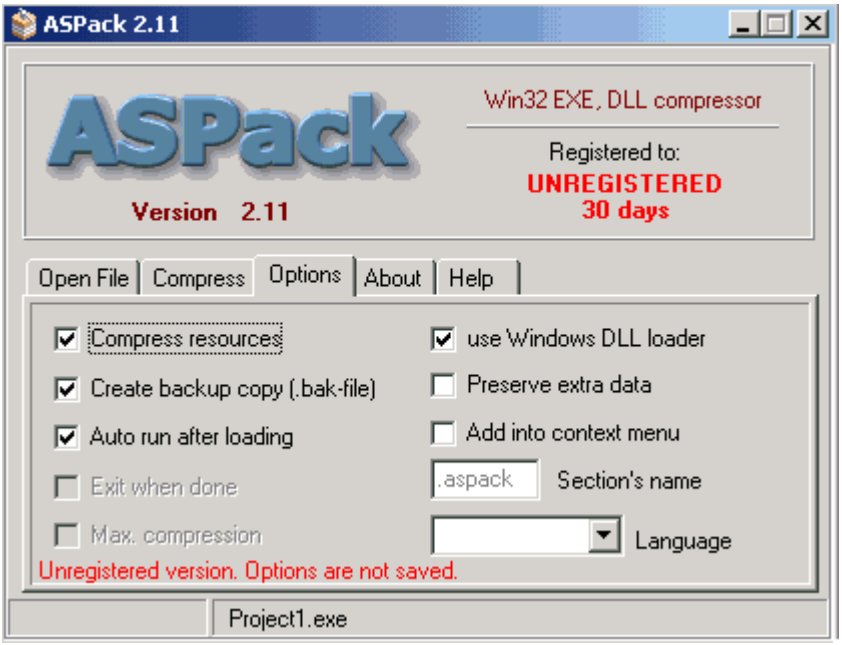


Рисунок 24.5.6. Настройки ASPack

Давай разберемся, как работает сжатие. Сначала весь код программы сжимается архиватором. Если ты думаешь, что он какой-то навороченный, то сильно ошибаешься. Для сжатия используется простой архиватор типа WinZIP, только оптимизированный для сжатия байт-кода. После этого, в конец сжатого кода добавляется код разархиватора,

который будет все это дело разжимать обратно. И в самом конце ASPack изменяет заголовок запускного файла так, чтобы при старте, сначала запускался разархиватор.

Теперь, когда ты запустишь сжатую программу, сначала запуститься разархиватор, который разожмет байт-код программы и аккуратно выложит его в памяти машины. Как только этот процесс пройдет, разархиватор передаст управление твоей программе.

Некоторые считают, что из за расходов на распаковку программа будет работать медленней!!! Я бы не сказал, что ты это заметишь. Даже если и будут какие-то потери, то они будут неощутимы. Это потому что архивация хорошо оптимизирована под байт-код. К тому же, размер программы уменьшится, а значит, она загрузится в память намного быстрее. В результате потери на скорости неощутимы даже с секундомером.

Конечно же, программирование на WinAPI слишком сложный процесс, но программы получаются очень маленького размера, и никакой архиватор не сможет сжать байт-код до такой степени. А если еще и сжать программу написанную на WinAPI, то ее размеры будут меньше некуда.

При нормальном программировании с использованием всех навороченных возможностей типа визуальности и объектного программирования код получается большим, но его можно сжать на 60-70% спец архиватором. К тому же такой код писать намного легче и быстрее.

Ещё одно «за» использование сжатие - сжатый код труднее взломать, потому что не каждый disassembler сможет прочитать упакованные команды. Так что, помимо уменьшения размера ты получаешь защиту способную отпугнуть большинство взломщиков. Конечно же, профессионала не отпугнешь даже этим, но взломщик средней руки не будет мучиться со сжатым байт-кодом.

24.6. Оптимизация скорости выполнения программы

Вся наша жизнь это борьба с тормозами и нехваткой времени. Каждый день мы тратим по несколько часов на оптимизацию. Каждый из нас старается оптимизировать все, что попадает под руку. А ты уверен, что ты это делаешь правильно? Может быть, есть возможность что-то сделать еще лучше?

Я понимаю, что все сейчас зажрались и выполняют свои обязанности лениво и не принужденно. Лично я до такой степени привык, что за меня все делает железный друг, что даже забыл, как выглядит шариковая ручка. Недавно мне пришлось писать заявление на отпуск на простой бумаге, так я забыл, как пишется буква "ю". Пришлось подглядывать, как она выглядит на клавиатуре :).

Даже для того, чтобы написать текст из двух строк, мы включаем свой компьютер и загружаем MS Word, тратя на это драгоценное время. А может легче было бы написать этот текст вручную? Я тебя понимаю - не солидно!!!

Программисты - так это вообще полное бесстыдство, тра-та-та (дальше все вырезано цензурой). Если они считают, что раз их творение (в виде исходника) никто не увидит, то можно писать что угодно? Так это они ошибаются. С этой точки зрения программы с открытым исходником в большом преимуществе, потому что они намного чище и быстрей. Создавая код, мы ленимся его оптимизировать не только с точки зрения размера, но и с точки зрения скорости. Глядя на такие творения, хочется выругаться матом, так ведь опять цензура обрежет.

Хакеры далеко не ушли. Если раньше, глядя на программиста или хакера создавался образ прокуренного, заросшего и немытого молодого человека, то сейчас это цифровое существо, залитое пивом Балтика по самые уши за которого все выполняют машины. Тебе медсестра в поликлинике не говорила, что у тебя вместо крови одно только пиво льется? Не, я ничего против пива не имею, я и сам его люблю.

Все это деградация по методу MS!!! Мы берем в руки мышку и начинаем тыкать ей где попало, забывая про клавиатуру и горячие клавиши. Я считаю, что надо бороться с этим. В последнее время меня самого посещает такая лень, что я убираю клавиатуру, запускаю экранную клавиатуру и начинаю работать только крысой. Осталось только покрыть мое тело шерстью и посадить в клетку к таким же ленивым шимпанзе.

Не надо тратить большие деньги на апгрейд компьютера!!! Начните лучше апгрейд с себя. Давайте, оптимизируем свою работу и то, что мы делаем.

В своем зародыше, эта часть книги задумывалась как рассказ об оптимизации кода программ, но в последствии я перенёс в неё свой труд, который можно найти и в инете на моём сайте, потому что оптимизировать надо всё. Я буду говорить про теорию оптимизации, а ее законы действуют везде. По тем же законам ты можешь оптимизировать свой распорядок дня, чтобы успевать все сделать, и свою ОС, чтобы она работала быстрей. Но основа все же будет относиться к коду программ.

Как всегда я постараюсь давать как можно больше реальных примеров, чтобы ты смог убедится в том, что тебе не вешают очередную лапшу на уши, и ты мог применить все сказанное на практике.

Начну я с законов, которые работают не только в программировании, но и в реальной жизни. Ну а напоследок я оставлю только то, что может пригодиться только при оптимизации кода.

ЗАКОН №1

Оптимизировать можно все. Даже там, где тебе кажется, что все и так работает быстро, можно сделать еще быстрее.

Это действительно так. И этот закон очень сильно проявляется в кодинге. Идеального кода не существует. Даже простую операцию сложения 2+2, тоже можно оптимизировать. Чтобы достичь максимального результата, нужно действовать последовательно и желательно в том порядке, в котором я буду описывать.

ЗАКОН №2

Первое с чего нужно начинать - это с поиска самых слабых и тормознутых мест. Зачем начинать оптимизацию с того, что и так работает достаточно быстро. Если ты будешь оптимизировать сильные места, то можешь нарваться на неожиданные конфликты.

Тут же я вспоминаю пример из своей собственной жизни. Где-то в 1995-96-м году меня посетила одна невероятная идея - написать собственную игру в стиле Doom. Я не собирался ее делать коммерческой, а хотел только потренировать свои мозги на сообразительность. Четыре месяца невероятного труда, и нечто похожее на движок уже было готово. Я создал один голый уровень, по которому можно было перемещаться, и с чувством гордости побежал по коридорам.

Никаких монстров, дверей и атрибутики на нем не было, а тормоза ощущались достаточно значительные. Тут я представил себе, что будет, если добавить монстров и атрибуты, да еще и наделить все это AI.... Вот тут чувство достоинство поникло. Кому нужен движок, который при разрешении 320x200 (тогда это было круто) в голом виде тормозит со страшной силой? Вот именно....

Понятное дело, что мой виртуальный мир нужно было оптимизировать. Целый месяц я бился над кодом и вылизывал каждый оператор моего движка. Результат - мир стал прорисовываться на 10% быстрей, но тормоза не исчезли. И тут я увидел самое слабое место - вывод на экран. Мой движок просчитывал сцены достаточно быстро, а пробоиной тормозов был именно вывод изображения. Тогда еще не было шины AGP и я

использовал простую PCI видюху от S3 с 1 мегом памяти. Пару часов колдовства, и я выжал из PCI все возможное. Откомпилировав движок, я снова загрузился в свой виртуальный мир. Одно нажатие клавиши вперед и я очутился у противоположной стены. Никаких тормозов, сумасшедшая скорость просчета и моментальный вывод на экран.

Как видишь, моя ошибка была в том, что я неправильно определил слабое место своего движка. Я месяц потратил месяц на оптимизацию математики и что в результате? Мизерные 10% прироста в производительности. Но когда я реально нашел слабое звено, то смог повысить производительность в несколько раз.

Слабые места компьютера

Меня поражают люди, которые гонятся за мегагерцами процессора и сидят на доисторической видюхе от S3, винте на 5400 оборотов и с 32 мегами памяти. Посмотри в корпус своего железного друга и оцени его содержимое. Если ты увидел, что памяти у тебя не более 32 мегов, то встань и громко произнеси: "Уважаемый DIMM, команда выбрала вас. Вы сегодня самое слабое звено и должны покинуть мой компьютер" :). После этого, закупаешь себе 128, а лучше 256 мегов памяти и наслаждаешься ускорением работы Delphi, Photoshop и других тяжелых программ.

В данном случае, наращивание мегагерцов у процессора даст более маленький прирост в скорости. Если ты используешь тяжелые приложения при нехватке памяти, то процессор начинает тратить слишком много времени на загрузку и выгрузку данных. Ну а если в твоем железном друге достаточно оперативки, то процессор уже занимается только расчетами и не расходуется по лишним загрузкам-выгрузкам.

То же самое с видюхой. Если она у тебя слабенькая, то процессор будет просчитывать сцены быстрей, чем они будут выводиться на экран. А это грозит простоями и минимальным приростом производительности.

ЗАКОН №3

Следующим шагом ты должен разобрать все операции по косточкам и выяснить, где происходят регулярно повторяющиеся операции. Начинать оптимизацию нужно именно с них.

Опять начнем рассмотрение этого закона с кодинга. Допустим, что у тебя есть следующий код (я приведу просто логику, а не реальную программу):

1. А:=А*2;
2. Б:=1;
3. Х:=Х+Б;
4. Б:=Б+1;
5. Если Б<100 то перейти на шаг 3;

Любой программист скажет, что здесь слабым местом является первая строка, потому что там используется умножение. Это действительно так. Умножение всегда выполняется дольше, и если заменить его на сложение (А:=А+А) или еще лучше на сдвиг, то ты выиграешь пару тактов процессорного времени. Но это только пару тактов и для процессора это будет незаметно.

Теперь посмотри еще раз на наш код. Больше ничего не видишь? А я вижу. В этом коде используется цикл: "*Пока Б<100 будет выполнятся операция Х:=Х+Б*". Это значит, что процессору придется выполнить 100 переходов с шага №5 на шаг №3. А это уже не мало. Как же можно здесь что-то оптимизировать? Очень легко. Здесь у нас внутри цикла выполняется две строки 3 и 4. А что если мы внутри цикла размножим их 2 раза:

2. Б:=1;

3. X:=X+Б;
4. Б:=Б+1;

5. X:=X+Б;
6. Б:=Б+1;

7. Если Б<50 то перейти на шаг 3;

Здесь я разложил цикл на более маленький. Вторую и третью операцию я повторил два раза. Это значит, что за один проход цикла я выполню два раза строки 2 и 3 и только после этого перейду на строку 1, чтобы повторить операцию. Такой цикл уже нужно повторить только 50 раз (потому что за один раз выполняется два действия). Это значит, что я сэкономил 50 операций переходов. Нехило? А это уже несколько сотен тактов процессорного времени.

А что если внутри цикла написать строки 2 и 3 десять раз. Это значит, что за один проход цикла строки 2 и 3 будут вычисляться 10 раз и мне понадобится повторить такой цикл только 10 раз, чтобы получить в результате 100. А это уже экономия 90 операций переходов.

Недостаток этого подхода - увеличился код моей программы, зато повысилась скорость и очень значительно. Этот подход очень хорош, но им не стоит злоупотреблять. С одной стороны увеличивается скорость, а с другой увеличивается размер. А большой размер это враг любой программы.

В жизни таких примеров намного больше. Любую циклическую операцию можно оптимизировать. Хочешь пример? Пожалуйста. Допустим у твоего провайдера интернета есть несколько телефонов доступа. Ты каждый день перезваниваешь на каждый из них в ожидании найти свободный. Начинающий тут же скажет, что провайдер обязан оптимизировать свои пулы модемов в один, чтобы не надо было трезвонить по всем номерам сразу. Но продвинутый должен знать, что не у каждого хорошая связь с любой станцией города. Поэтому провайдеры держат пулы на разных станциях, чтобы ты мог выбрать тот с которым у тебя лучший коннект. Тогда что же оптимизировать? Очень просто - поставь прогу дозвонщик (таких сейчас полно в интернете) которая сама будет перебирать номера телефонов.

А теперь другой пример - тебе досталась карточка на 1 час какого-то нового провайдера. Заносить ее в прогу-дозвон не имеет смысла, потому что ты можешь больше никогда не позвонить ему. Из-за этой одноразовой операции тебе придется перенастраивать свой дозвонщик на нового провайдера и потом обратно, а выигрыш практически нулевой, потому что пока ты меняешь настройки уже можно было дозвониться.

ЗАКОН №4.

(этот закон как бы расширение предыдущего). Оптимизировать одноразовые операции - это только потеря времени. Сто раз подумай, прежде чем начать мучится с редкими операциями.

Пол годика назад я прочитал рассказ в интернете "Записки жены программиста" (http://www.exler.ru/novels/wife.htm). Очень даже некислый и жизненный рассказ. Когда я его читал, у меня было ощущение, что его написала моя жена :). Слава "Красной Шапочке", что она на такую подлость не способна. Так вот там была такая ситуация:

"Очаровашка выходит замуж за программера и им надо разослать приглашения на свадьбу. Вместо того, чтобы набрать их на печатной машинке, программер кричит, что он крутой и пишет специальную прогу. Написание проги заняло один день и столько же ее отладка".

Главная ошибка - неправильная оптимизация своего труда. Легче набрать шаблон в любом текстовом редакторе и потом только менять фамилии приглашенных на этот траурный день (это я сужу по себе :)). Но даже если нет текстового редактора, писать прогу действительно нет смысла. Затраты большие а пользоваться ей будешь только один раз. Здесь действительно легче будет даже набрать на печатной машинке.

Получается, что одноразовые операции оптимизировать нет смысла. Затраты тут себя не окупают, поэтому не стоит тратить свои нервы на этот бессмысленный труд.

В самом начале этой части я раскритиковал тебя как лентяя, который ленится что-то делать. Так вот именно здесь ты можешь проявлять все свои ленивые качества в полном объеме. В данном случае крутым считается не тот, кто целый день промучился и ничего не добился, а тот, кто выполнил свою работу наиболее эффективно. И эти две вещи путать нельзя.

ЗАКОН №5

Нужно знать внутренности компьютера и принципы его работы. Чем лучше ты знаешь, каким образом компьютер будет выполнять твой код, тем лучше ты сможешь его оптимизировать.

Это последний закон, который я хотел бы тебе сказать, и относится он только к кодингу. Тут трудно привести полный набор готовых решений, но некоторые приемы я постараюсь описать.

1. Старайся поменьше использовать вычисления с плавающей запятой. Любые операции с целыми числами выполняются в несколько раз быстрее.

2. Операции умножения и тем более деления так же выполняются достаточно долго. Если тебе нужно умножить како-то число на 3, то для проца будет легче три раза сложить одно и то же число, выполнить умножение.

А как же тогда экономить на делении? Вот тут нужно знать математику. У проца есть такая вещь как сдвиг. Ты должен знать, что процессор думает с помощью нулей и единиц. Это значит, что числа в нем хранятся в двоичной системе. Например, число 198 для процессора будет выглядеть как 11000110. Теперь посмотрим, как работают операции сдвига.

Сдвиг вправо: Если сдвинуть число 11000110 вправо на одну позицию, то последняя цифра исчезнет и останется только 1100011. Теперь загони это число в калькулятор и переведи его в десятичную систему. Твой результат должен быть 99. Как видишь - это ровно половина числа 198. Вывод - когда ты сдвигаешь число вправо на одну позицию, то ты делишь его на 2.

Сдвиг влево: Возьмем то же самое число 11000110. Если сдвинуть его влево на одну позицию, то с правой стороны освободится место, которое сразу заполняется нулем 110001100. Теперь переведи это число в десятичную систему. У тебя должно получится 396. Ни что не напоминает тебе? Это 198 умноженное на 2.

Вывод: Когда ты сдвигаешь число вправо, то ты делишь его на 2. Когда сдвигаешь влево, то умножаешь его на 2. Так что используй это свойство сдвигов везде, где это возможно, потому что сдвиги работают в десятки раз быстрее.

3. Когда создаешь процедуры, то не пичкай их большим количеством входных параметров. Перед каждым вызовом процедуры ее параметры подымаются в специальную область памяти (стек), а после входа изымаются оттуда. Чем больше параметров, тем больше расходы на общение со стеком.

Тут же нужно сказать, что ты должен быть аккуратным и с самими параметрами. Не вздумай пересылать процедурам переменные, которые могут содержать данные большого объема в чистом виде. Лучше передай адрес ячейки памяти, где хранятся данные, а внутри процедуры работай с этим адресом. Ты представь себе ситуацию, когда тебе нужно передать текст размером одного тома "Войны и мир".... Перед входом в процедуру, программа попытается вогнать все это в стек. Если ты не схватишь его переполнение, то тормоза ощутишь значительные.

4. В самых критичных моментах (как, например вывод на экран) можно воспользоваться языком Assembler. Даже встроенный в Delphi или C++ ассемблер на много быстрее. Ну а если скорость в каком-то месте уж слишком критична, то код ассемблера можно вынести в отдельный модуль. Там его нужно откомпилировать с помощью компиляторов TASM или MASM, и подключить к своей проге.

Ассемблер достаточно быстрая и компактная вещь, но писать достаточно большой проект только на нем это чистый геморрой. Поэтому я не советую им увлекаться, и используй его только в самых критичных для скорости местах.

Если ты прочитал все внимательно, то можешь считать, что с основами оптимизации ты уже знаком. Но это только основы и тут есть куда развиваться. Я бы мог рассказать больше, но не вижу особого смысла, потому что оптимизация - это процесс творческий и в каждой отдельной ситуации можно подойти с разных сторон. И все же, те законы, которые я сегодня описал, действуют в 99,9% случаев.

Если ты хочешь познать теорию оптимизации более глубоко, то тебе нужно больше изучать принципы работы процессора и операционных систем. Главное, что законы ты уже знаешь, а остальное прейдет со временем и навыками.

Глава 25. Сплошная практика

Я уже рассказал все необходимые основы. Теперь ты готов к самостоятельному изучению тонкостей программирования, а я только дам несколько практических примеров. В этой главе я покажу, как можно создавать собственные программы ScreenSaver и закрепим максимум из пройдённого материала на полезных в реальной жизни примерах.

Те примеры, которые я писал на протяжении всей книги, очень хороши и удобны в образовательных целях. Но для полной готовности к реальной жизни надо ещё немного попрактиковаться, потому что описанные примеры были маленькие (в целях экономии места в книги) и теперь надо научиться всё сказанное собирать в единое целое.

Достаточно много дополнительного материала ты найдёшь на диске к этой книге. Я понимаю, что читать с монитора не так уж удобно, но я не хочу, чтобы эта книга была слишком дорогой. Моя Библия должна стать доступной каждому. При этом я постарался дать в ней максимум информации, при этом не просто заполнить компакт диск всякой ерундой, а наполнить его полезными вещами. Так что не поленись, и после прочтения книги загляни в директорию «Документация».

25.1. Создание ScreenSaver

Если ты до сих пор считаешь, что Delphi это язык, предназначенный для работы с базами данных, то в этой статье я окончательно опровергну твоё мнение. Здесь я покажу, что на Delphi можно написать даже хранитель экрана. Самое главное, что никаких особых усилий не понадобится.

С одной стороны, эта часть должна была идти в главе по графике, потому что мы будем много рисовать. Но я решил её вставить сюда, потому что используемые здесь графические функции мы уже изучили. А вот само программирование будет очень интересным.

Для того, чтобы Delphi мог создать хранитель экрана, ты должен сделать несколько установок для своей формы:

1. Перед объявлением типов вставить вот такую строку: *{$D SCRNSAVE Saver}*. Здесь *Saver* – имя нашего проекта, я его сохранил под именем *Saver.dpr*.

2. Свойство формы *BorderStyle* поменять на *bsNone*. Это означает, что у формы не должно быть никаких заголовков и обрамлений.

3. Все параметры в *Border Icons* установить в False.

4. Свойство формы *WindowState* поменять на *wsMaximized*, чтобы окно появлялось максимизированным на весь экран.

5. Создать событие *OnKeyPress* и вставить туда всего лишь одну процедуру - *Close()* – закрытие хранителя экрана по нажатию любой клавиши.

6. На событие *FormActivate* значения *Left* и *Top* нужно установить в ноль.

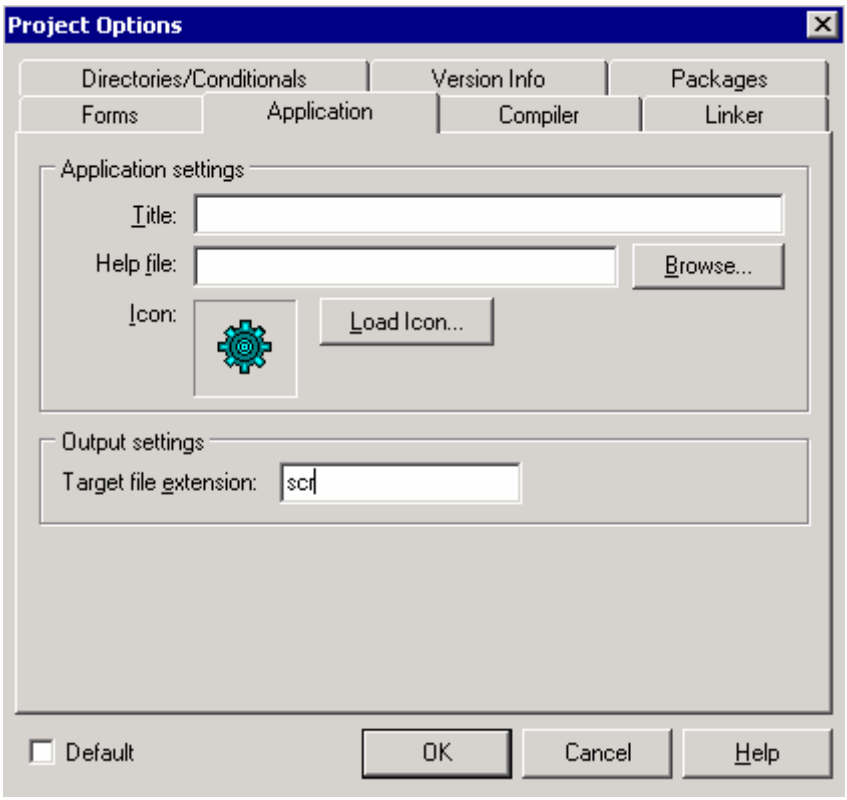


Рисунок 25.1.1 Свойства проекта

Теперь Delphi будек компилировать запускной файл, совместимый с хранителем экрана. Что такое ScreenSaver? Это та же программа, только с расширением *scr*. Так что

остаётся одна деталь – изменить расширение на *scr*. Ты можешь после компиляции программы переименовать файл в *.scr или возложить этот тяжёлый труд на Delphi. Для этого необходимо выбрать пункт Option из меню Project и на закладке *Application* в строке *Target file* extension написать scr (рисунок 25.1.1). В этом случае Delphi сам подставить это расширение.

Подготовительные работы закончены. Можно приступать к написанию хранителя экрана. Ты уже знаешь достаточно много, и сможешь сам написать что-нибудь интересное. А я здесь покажу простейший пример.

Для моего примера понадобится объявить три переменные в разделе **private**:

```
 private
  { Private declarations }
  BGbitmap:TBitmap;
  DC : hDC;
  BackgroundCanvas : TCanvas;
```

Затем посмотрим на мой обработчик события *OnCreate*:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 BGbitmap:=TBitmap.Create;//Инициализация

// Выставляем размеры картинки как у экрана
 BGbitmap.Width := Screen.Width;
 BGbitmap.Height := Screen.Height;

 DC := GetDC (0);
 BackgroundCanvas := TCanvas.Create;
 BackgroundCanvas.Handle := DC;

 BGBitmap.Canvas.CopyRect(Rect (0, 0, Screen.Width, Screen.Height),
                  BackgroundCanvas,
                  Rect (0, 0, Screen.Width, Screen.Height));
 BackgroundCanvas.Free;
 randomize;
end;
```

Что здесь творится? В самом начале я инициализирую *BGbitmap* и устанавливаю ему размер как у экрана. Объект *Screen* – это объект, который создаётся автоматически при старте программы и содержит в себе информацию об экране. В свойствах *Width* и *Height* находятся ширина и высота экрана.

Потом в переменную *DC* заноситься указатель на контекст вывода экрана. На первый взгляд это происходит не явно, но всё очень просто. Функция *GetDC* возвращает указатель на контекст указанного в качестве параметра устройства или формы. Если указать 0, то *GetDC* вернёт указатель на контекст воспроизведения экрана. Если ты захочешь получить контекст воспроизведения твоей формы, то ты должен написать *GetDC(Handle)*. В качестве параметра выступает *Handle* (указатель) окна. Но ты в таком виде не будешь никогда использовать *GetDC*, потому что у тебя уже есть контекст воспроизведения формы - это *Canvas*, а указатель на него - *Canvas.Handle*.

Дальше я создаю *BackgroundCanvas* - это *TCanvas*, который будет указывать на экран. Я делаю это только для удобства. Я мог бы использовать для рисования *DC*, но всё

же *TCanvas* более понятен и мы с ним достаточно подробно разобрались в главе посвящённой графике.

С помощью следующей строки, я сохраняю копию экрана:

```
BGBitmap.Canvas.CopyRect(Rect (0, 0, Screen.Width, Screen.Height),
                         BackgroundCanvas,
                         Rect (0, 0, Screen.Width, Screen.Height));
```

Затем я уничтожаю указатель на контекст экрана *BackgroundCanvas.Free*, потому что больше он нам не понадобиться.

Самая последняя функция - *randomize* инициализирует таблицу случайных чисел. Если ты этого не сделаешь, то после запроса у системы случайного числа, тебе вернут значение из текущей таблицы, которая не всегда удачна. Тебе не надо будет видеть таблицу случайных чисел, она прекрасно будет работать без твоих глаз.

В обработчике события *OnDestroy* я уничтожаю картинку:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
 BGBitmap.Free;
end;
```

Последняя функция - это обработчик таймера, который я поставил на форму:

```
procedure TForm1.Timer1Timer(Sender: TObject);
const
 DrawColors: array[0..7] of TColor =(clRed, clBlue, clYellow, clGreen,
                        clAqua, clFuchsia, clMaroon, clSilver);
begin
 BGBitmap.Canvas.Pen.Color:=DrawColors[random(7)];
 BGBitmap.Canvas.MoveTo(random(Screen.Width),random(Screen.Height));
 BGBitmap.Canvas.LineTo(random(Screen.Width),random(Screen.Height));
 Canvas.Draw(0,0,BGBitmap);
end;
```

В разделе констант я объявляю массив *DrawColors*, который хранит восемь цветов. Объявление происходит следующим образом:

Имя_Массива : array [Индекс_первого_значения .. Индекс_последнего_значения] of Тип_значения_Массива = (Перечисление_значений)

В "перечислении значений" должно быть описано "Индекс последнего значения" - "Индекс первого значения"+1 параметров. В моём случае это 7-0+1=8 значений цвета.

Дальше меняю цвет у контекста рисования формы *BGBitmap.Canvas.Pen.Color* случайным цветом из массива *DrawColors*. Функция *random* возвращает число из таблицы случайных чисел, при этом, это число будет больше нуля и меньше значения переданного в качестве параметра. Это значит, что *random(7)* вернёт случайное число от 0 до 7. С помощью *BGBitmap.Canvas.MoveTo* я перемещаюсь в случайную точку внутри *BGBitmap* и с помощью *BGBitmap.Canvas.LineTo* рисую из этой точки в новую точку линию. *Canvas.Draw(0,0,BGBitmap)* выводит моё произведение на экран.

Попробуй запустить пример, и ты увидишь, как экран засыпается линиями со случайными координатами. Но если ты протестируешь пример, то заметишь несколько недостатков:

1. Если скопировать хранитель экрана в системную директорию (windows или winnt) и попытаться установить этот хранитель, то будут заметны блики.

2. При нажатии на кнопку «*Настроить*» запускается хранитель, а не окно настройки.

Как же нашему хранителю экрана узнать, что надо отображать окно настройки, а не запускаться самому? Очень просто. При старте хранителя экрана нам в командной строке передаются параметры. Они могут быть следующими:

/s – надо запустить хранитель экрана.

/p – хранитель экрана должен отображаться в окне свойств экрана на рисунке монитора (рисунок 25.1.2).

/c:xxxxx – нужно отобразить окно настроек. Здесь после двоеточия, вместо xxxx стоит число.

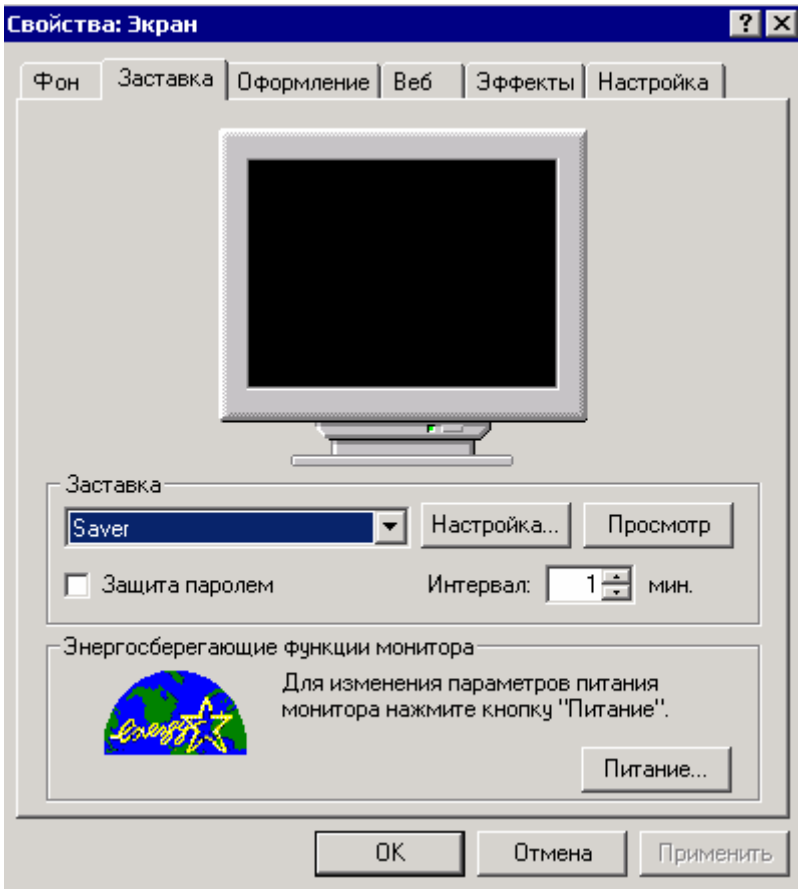


Рисунок 25.1.2 Свойства экрана

Количество параметров переданных программе можно узнать, вызвав функцию *ParamCount*. Эта функция вернёт только количество параметров. Сами параметры можно получить с помощью функции *ParamStr*. Этой функции нужно только передать номер параметра, который мы хотим получить. Чтобы перебрать все переданные программе значения можно использовать следующий код:

```
var
 i:Integer;
begin
```

```
 for i:=0 to ParamCount-1 do
  Переменная:=ParamStr(i);
end;
```

Теперь забудем на минутку про параметры и для начала избавимся от бликов. Для этого по событию *OnCreate* в самом начале присваиваем ширине и высоте окна значения 0, чтобы окно было минимальным. В этом случае окно не будет мерцать, потому что его просто нет на экране из-за нулевой ширины и высоты.

```
 Width:=0;
 Height:=0;
```

А вот теперь вернёмся к нашим параметрам. По событию *OnShow* для главной формы пишем следующий код:

```
procedure TSaverForm.FormShow(Sender: TObject);
begin
 if ParamCount>0 then
  begin
   if ParamStr(1)='/p' then
    begin
     Close;
     exit;
    end;
   if ParamStr(1)[2]='c' then
    begin
     OptionsForm.ShowModal;
     Close;
     exit;
    end;
  end;

 Timer1.Enabled:=true;
 Width:=Screen.Width;
 Height:=Screen.Height;
end;
```

В самом начале я проверяю, если количество параметров больше нуля, то нужно будет проверить, что нам передали. Если параметр равен ‘/p’ то мы просто будем закрывать хранитель экрана и ничего отображать не будем. Если параметр равен ‘/c:xxxx’, то будем отображать окно настроек. Но здесь с проверкой небольшое затруднение, ведь мы не знаем, какое число будет вместо *xxxx*. Поэтому я просто проверяю второй символ параметра и если он равен букве «c», то значит это «/c».

Окно свойств я сделал простейшим (смотри рисунок 25.1.3), в нём всего лишь запрашивается пароль для хранителя экрана. Саму реализацию работы с паролем я опустил, потому что здесь всё просто. Тебе нужно сохранять этот пароль в реестре, а при попытке выхода из программы запрашивать ввод пароля. Если пользователь ввёл правильный пароль, то хранитель экрана можно закрывать. Попробуй всю эту логику написать сам. Если что-то не получиться, то на диске сможешь найти мой пример, в котором реализовано всё необходимое.

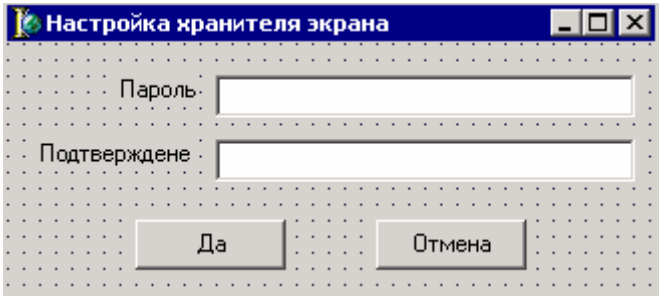


Рисунок 25.1.3 Окно настроек хранителя экрана

Теперь вернёмся к нашему обработчику события *OnShow*. После проверки всех параметров, я делаю таймер *Timer1* активным. Если у тебя на форме стоит активный таймер (свойство *Enabled* равно *true*), то сделай его неактивным.

После активизации таймера я выставляю ширину и высоту окна в значения ширины и высоты всего экрана, чтобы моё окно покрыло весь экран.

Остальной код практически не изменился, за исключением того, что ты должен добавить проверку пароля. Я могу тебе ещё посоветовать добавить возможность управления частотой таймера в окне настроек, но это уже по желанию.

*На компакт диске, в директории **\Примеры\Глава 25\ScreenSaver** ты можешь увидеть пример этой программы.*

25.2. Компоненты в runtime

Здесь я опишу маленький пример, который ответит сразу на два поставленных мне вопроса: как создавать компоненты во время выполнения программы и как ими управлять. Что такое runtime? Твоя прога может находиться в двух состояниях - designtime (время создания проекта) и realtime (время выполнения проекта). Мы сегодня будем создавать компоненты не рисованием на форме, а чистым кодом уже во время выполнения программы.

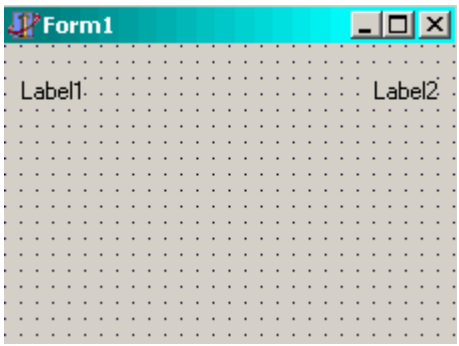


Рис 25.2.1. Форма будущей программы

Создай новый проект и брось на него два компонента *TLabel* рисунок 25.2.1. Всё остальное будем делать ручками. Для начала, в разделе **private** объявим переменную *CompList* типа *TList*. *TList* - это "объект-контейнер", который может хранить в себе кучу других. Точнее сказать, он хранит только ссылки, но это не главное. Главное - *TList* позволяет хорошо управлять хранящимися в нём объектами.

На событие *OnCreate* напиши:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
 CompList:=TList.Create;
end;
```

Здесь мы инициализируем нашу переменную *CompList* с помощью объекта *TList*. Во время инициализации выделяется память под нашу переменную. Сразу же на событие OnDestroy пишем:

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
 CompList.Free;
end;
```

Здесь мы освобождаем выделенную память для переменной *CompList*.

Теперь создадим обработчик события нажатия мышкой *OnMouseDown*. По нажатию кнопкой, мы будем создавать на форме новый компонент *TPanel*. Давай в нём напишем следующий код:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
 TempPanel:TPanel;//Объявляю переменную для панели
begin
 //Создаю панель. В скобках у Create указан будущий владелец
 TempPanel:=TPanel.Create(Form1);

 TempPanel.Left:=X; //Устанавливаю левую и правую координату
 TempPanel.Top:=Y;  //в X и Y позицию, где нажата кнопка мыши

 TempPanel.Width:=20;  //Устанавливаю ширину
 TempPanel.Height:=20; //Устанавливаю высоту

 //Далее устанавливаю обработчик нажатия на эту панель
 TempPanel.OnMouseDown:=PanelMouseDown;

 //Добавляю панель в контейнер CompList (CompList.Add)
 //и сохраняю результат в TempPanel.Tag
 TempPanel.Tag:=CompList.Add(TempPanel);

 Form1.InsertControl(TempPanel); //Вставляю панель на форму
end;
```

Для начала вспомним, что это за свойство *Tag* у компонента *TPanel*. Это просто целое значение, которое ты можешь использовать по своему усмотрению. Именно этим свойством мы и будем часто пользоваться во время программирования нашего примера.

Теперь разберём написанный код. В разделе **var** я объявил одну переменную *TempPanel* типа *TPanel*. Это временная переменная, в которой будет инициализироваться новая панель. В первой же строчке кода обработчика я инициализирую эту переменную, как панель. В качестве параметра методу *Create* я должен передавать имя объекта,

который будет являться родителем создаваемого компонента. Я передаю нашу главную форму, потому что компонент будет размещаться именно на нём.

Следующим этапом, я устанавливаю левую и правую позицию панели в координаты, где мы щёлкнули мышкой (X и Y, которые нам переданы в обработчике, указывают на точку, в которой была нажата кнопка мышки). Далее, я устанавливаю ширину и высоту панели. Я решил занести туда значение 20 (просто так захотелось).

Теперь об обработчике события *TempPanel.OnMouseDown*. Я туда засунул имя функции *PanelMouseDown*. Но такой функции нет среди стандартных функций и среди моего проекта. Поэтому мы должны её создать сами. Как это сделать эффективно? Вот тебе мой совет:

1. Мы создаём обработчик для *TPanel*, поэтому временно поставь один экземпляр панели на форму в произвольное место.

2. Создай для него обработчик на *OnMouseDown* и переименуй его в *PanelMouseDown*.

3. Напиши нужный текст (я его покажу ниже) и можно удалять временно созданный на форме экземпляр *TPanel*.

Таким образом, ты можешь быть уверен, что ошибок не будет, потому что Delphi сама пропишет функцию PanelMouseDown где надо и укажет все необходимые параметры.

Если захочешь объявлять эту функцию вручную, то напиши в разделе **private**:

```
procedure PanelMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
```

Объявлять можно и до **private**, там где объявляет Delphi обработчики событий. А ниже опиши саму функцию

```
procedure TForm1.PanelMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin

end;
```

Обязательно нужно следить, чтобы количество и тип параметров точно совпадали с необходимыми. У каждого обработчика свои параметры и при объявлении процедуры вручную, нужно относиться к этому вопросу очень внимательно. Именно поэтому я советую тебе первый способ, с временной панелью, когда Delphi сам создаст обработчик для одной панели, а ты будешь использовать его для других.

Всё, панель готова и её надо сохранить в нашем контейнере *CompList*. Для этого нужно выполнить метод *Add* нашего контейнера, в качестве параметра передать ему нашу панель:

```
CompList.Add(TempPanel)
```

Этот метод добавит панель в контейнер и вернёт нам индекс компонента в контейнере. Этот индекс я сохраняю в свойстве *Tag* нашей панели *TempPanel*. Это свойство абсолютно не влияет на сам компонент, а мне этот индекс пригодится.

Теперь давай посмотрим на функцию *PanelMouseDown*, которая должна быть такой:

```
procedure TForm1.PanelMouseDown(Sender: TObject; Button: TMouseButton;
```

```
  Shift: TShiftState; X, Y: Integer);
begin
 Label1.Caption:=IntToStr(TPanel(CompList.Items[TPanel(Sender).Tag]).Left);
 Label2.Caption:=IntToStr(TPanel(Sender).Left);
end;
```

Здесь две строки. Обе строки выполняют одно и тоже, но по-разному. Обе строки записывают в свой TLabel левую позицию панели, по которой ты щёлкнул.

Первая строка, чтобы получить левую позицию панели использует CompList, а вторая работает с панелью напрямую. Рассмотрим сначала вторую строку. В ней основным является выражение *TPanel(Sender).Left*. *Sender* - передаётся нам процедурой обработчиком *PanelMouseDown*. В нём записан указатель на объект, который сгенерировал событие *OnMouseDown*. В нашем случае это будет указатель на панель, по которой ты щёлкнул. Так как мы точно уверены, что это панель, то мы так и показываем *TPanel(Sender)*. Этим мы приводим *Sender* к *TPanel* и теперь ты можешь использовать все свойства и методы панели, для примера нам достаточно свойства *Left*. Если бы мы знали точное имя панели, то этого писать не пришлось бы. Но это невозможно, потому что все создаваемые в Runtime панели (а их можно создать любое количество) у нас используют один обработчик нажатия мышкой, и мы не знаем, по какой именно панели был произведён щелчок. Получив значение левой позиции, мы переводим целое значение левой позиции в строку с помощью *IntToStr*.

Первоя строка очень похожа на вторую, только внутри *TPanel()* мы используем не *Sender*, а *CompList.Items[TPanel(Sender).Tag]*, т.е. значение из конейнера. Чтобы получить первое значение из контейнера, нужно написать *CompList.Items[0]*, для второго *CompList.Items[1]*, для третьего *CompList.Items[2]* и т.д. Но по какой именно панели произведён щелчок? Чтобы это узнать я пишу *TPanel(Sender).Tag*, то есть получаю свойство *Tag* (там хранятся индекс панели) панели сгенерировавшей событие. Далее, всё происходит так же.

Запусти пример и пощёлкай по форме. По каждому щелчку будут создаваться панели. Потом попробуй пощёлкать по самим панелям. На двух TLabel будут появляться значения левой позиции панель, по которым ты щёлкал.

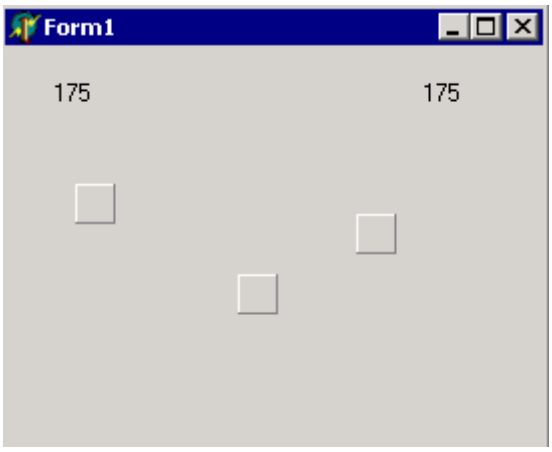


Рисунок 25.2.2 Пример работы программы.

Теперь я хочу тебе показать ещё несколько интересных свойств и методов, которые есть у контейнера *TList*:

Count – в этом свойстве храниться количество элементов в контейнере.

Items – здесь хранятся ссылки на элементы контейнера. Для доступа к ссылкам нужно написать *Items[Индекс элемента]*.

Clear – очистить список.

Delete – удалить элемент из списка. В качестве единственного параметра нужно указать индекс удаляемого элемента.

Exchange – поменять в контейнере местами два элемента. Здесь два параметры – индексы меняемых местами компонентов.

First – получить указатель на первый элемент списка. Это то же самое, что и записать *Items[0]*.

IndexOf – получить индекс указанного в качестве параметра объекта. Допустим, что ты знаешь объект (*TPanel*) и хочешь узнать, под каким индексом он расположен в контейнере. В этом случае ты можешь написать следующий код: *CompList.IndexOf(Panel1)*. Если такой панели не найдено в списке, то тебе будет возвращено значение –1, иначе правильный индекс указанной панели.

Insert – вставить новый элемент. У этого метода два параметра – индекс, под которым надо вставить элемент и сам элемент.

Last – получить последний элемент списка. Это то же самое, что использовать свойство *Items[Count - 1]*.

Move – переместить элемент в новое место. У метода два параметра – индекс элемента, который надо переместить и индекс, который должен получить элемент.

Pack – при удалении элементов из контейнера, они просто помечаются, как нулевые. Выполняя этот метод, все нулевые элементы уничтожаются, и занятая ими память освобождается.

Remove – удалить элемент. В качестве параметра нужно указывать элемент, который надо удалить, например *CompList.Remove(Panel1)*.

Давай добавим в наш пример возможность удаления панели с формы и из контейнера. Это будет происходить по нажатию правой кнопки мышки по компоненту. Добавь с конец процедуры *PanelMouseDown* следующий код:

```
if Button=mbRight then
  begin
   index:=TPanel(Sender).Tag;
   TPanel(CompList.Items[index]).Free;
   CompList.Delete(index);

   for i:=index to CompList.Count -1 do
    TPanel(CompList.Items[i]).Tag:=TPanel(CompList.Items[i]).Tag-1;
  end;
```

В разделе **var** этой процедуры нужно объявить две переменные *index* и *i*. Обе они будут числами целого типа.

Теперь разберём код. Сначала я проверяю, если нажата правая кнопка мыши, то нужно удалить компонент, по которому щёлкнули. Для этого, я сначала сохраняю индекс компонента *TPanel(Sender).Tag* в переменной *index*. Это необходимо, потому что после уничтожения компонента *TPanel(Sender)* не будет существовать, и я не смогу получить доступ к его свойству *Tag*.

Следующим этапом происходит удаление. Сначала я уничтожаю сам компонент - *TPanel(CompList.Items[index]).Free*, а после это уничтожаю ссылку на него в контейнере - *CompList.Delete(index)*. Вроде бы всё удалили, но программа после этого будет работать неправильно. Допустим, что у нас в контейнере было 5 элементов, и мы удалили 3-й. По

идее, в контейнере должны остаться элементы с индексами 1, 2, 4, 5, а 3-й должен отсутствовать. Но реально индексы перестроятся, и мы увидим индексы 1, 2, 3, 4. Если мы попытаемся оставить всё так, как есть, то программа будет нестабильна. Если ты щёлкнешь, по последней созданной панели, то программа обратиться к элементу в контейнере под номером 5, потому что в свойстве *Tag* панели находиться цифра 5. Но такого элемента не существует и произойдёт ошибка. Поэтому нам надо подкорректировать свойства Tag у всех панелей начиная с удаляемой. Для этого я запускаю цикл от индекса удаляемой панели до последнего элемента списка и уменьшаю их свойство *Tag*:

```
for i:=index to CompList.Count -1 do
 TPanel(CompList.Items[i]).Tag:=TPanel(CompList.Items[i]).Tag-1;
```

Вот теперь у меня в контейнере будут элементы с индексами от 1 до 4 и у всех панелей на форме в свойстве *Tag* будут правильные значения.

На компакт диске, в директории ***\Примеры\Глава 25\Runtime*** *ты можешь увидеть пример этой программы.*

25.3. Тест на прочность.

Ко мне очень часто приходят вопросы, хоть как-то связанные с написанием программы теста. Видимо преподаватели в институтах начинают любить компьютеры, и используют их для тестирования знаний учеников. Но так как в наших школах есть проблемы с программным обеспечением, то преподаватели дают задания своим ученикам-программистам написать какой-нибудь определённый тест.

В этой книге я решил подробно описать процесс написания программы теста, потому что в такой программе будет множество интересных приёмов программирования и в для обучения такая программа будет просто идеальной.

Итак, нам придётся написать де программы:

1. Редактор тестов. В ней будут создаваться тесты, заполняться вопросы, на которые надо будет отвечать и варианты правильных ответов.
2. Программа тестирования. Это оболочка, которая будет загружать созданные тесты, и в ней пользователь должен будет отвечать на появляющиеся вопросы.

Итак, начнём мы с редактора, потому что сначала нужно научиться создавать тесты, а потом уже будем их отображать и работать с ними. Для начала я создал главную форму, как показано на рисунке 25.3.1. Здесь у меня главное окно, в котором есть главное меню программы, панель кнопок быстрого вызова команд ToolBar и строка состояния, которая будет отображать подсказки. Попробуй создать что-то подобное.

На форме у меня созданы следующие кнопки (и соответствующий пункты меню):

1. Создать;
2. Открыть;
3. Сохранить;
4. Печать;
5. Настройки программы;
6. Помощь;
7. О программе;
8. Выход.

У каждой кнопки в свойстве Hint указана соответствующая ей подсказка, которая должна появляться в строке состояния при наведении на кнопку. Чтобы эта подсказка появлялась и рядом с кнопкой, установи свойство *ShowHint* у нашей главной формы в значение *true*.

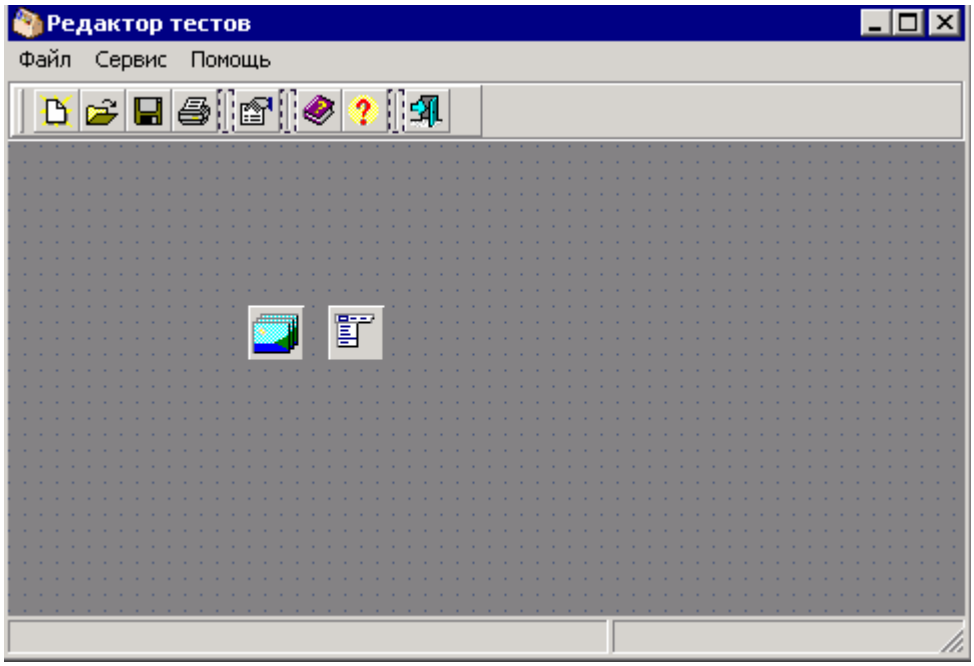


Рисунок 25.3.1. Главная форма редактора теста.

Первым делом, давай сразу же сделаем возможность отображения подсказок *Hint* в строке состояния. Для этого в разделе **private** опиши новую процедуру:

```
private
  { Private declarations }
  procedure ShowHint(Sender: TObject);
```

Теперь нажми Ctrl+Shift+C, чтобы создать заготовку для этой процедуры. В ней нужно написать следующее:

```
procedure TTestEditorForm.ShowHint(Sender: TObject);
begin
 StatusBar.Panels.Items[0].Text := Application.Hint;
end;
```

Здесь я отображаю текст текущей подсказки, который находиться в свойстве *Hint* объекта *Application* на нулевой панели строки состояния. У меня строка состояния состоит из двух панелей.

Теперь сделай нашу главную форму многодокументной. Для этого в свойстве *FormStyle* нужно установить значение *fsMDIForm*.

Можешь сразу же создать окно «О программе». Я это не буду описывать, потому что тут фантазия у каждого работает по-разному, а моё окно ты сможешь увидеть на диске вместе с исходником. Можешь ещё сразу создать обработчик события OnClick для кнопки

выхода и написать там вызов метода *Close*, чтобы по нажатию этой кнопки наша программа закрывалась. Потом, эту же процедуру обработчик нужно назначит пункту меню «Выход».

На этом первые приготовления закончены. Теперь двинемся дальше. Создай новую форму (я её назвал *NewTestForm*), которая будет отображаться по нажатию кнопки «Создать». В этом окне нужно расположить строку ввода имени теста и выпадающий список для выбора типа теста. Я разместил все компоненты так, как показано на рисунке 25.3.2

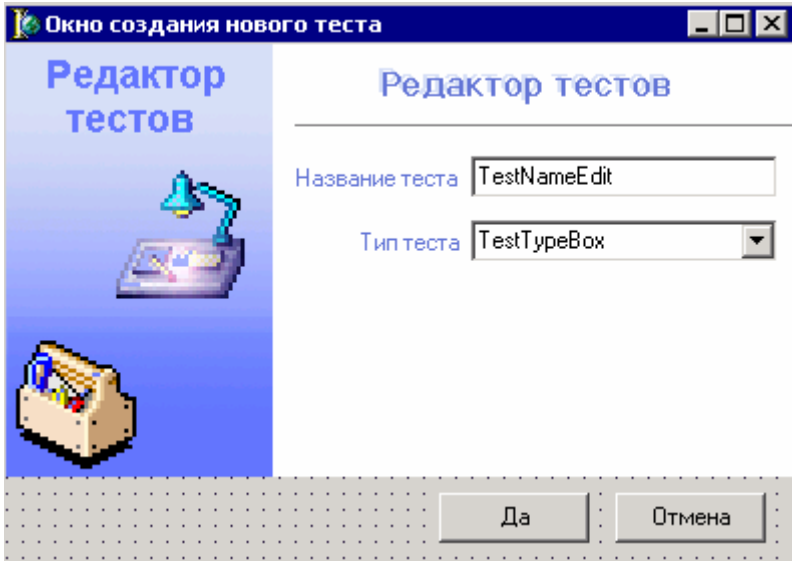


Рисунок 25.3.2. Окно создания нового теста.

У выпадающего списка я изменил свойство *Style* на *csDropDownList* для того, чтобы пользователь мог только выбирать уже имеющееся значение в списке и не мог вводить новое. В свойстве *Items* (список элементов) у меня будет только одна строка – «Вопрос - варианты ответа». В принципе, можно было бы не делать этот выпадающий список, раз там только один элемент, но я его поставил, чтобы ты потом мог расширить возможности программы.

Для кнопок «*Да*» и «*Нет*» я установил только свойство *ModalResult* в значения *mrOk* и *mrCancel* соответственно.

У самой формы я изменил свойства:

Position на *poMainFormCenter*, чтобы окно показывалось по центру главного окна.

BorderStyle на *bsSingle*, чтобы пользователь не мог изменять размеры окна.

Эти свойства я буду менять у всех форм, которые будут отображаться модально. Кнопки «*Да*» и «*Нет*» у таких окон тоже всегда будут иметь указанные выше значения свойства *ModalResult*, поэтому я больше не буду останавливаться на этих вещах. Просто помни, это.

Теперь возвращаемся в главную форму и по событию *OnClick* для кнопки создать пишем код отображения окна *NewTestForm*. Это окно нужно отображать как модальное.

На этом остановимся и сделаем промежуточную паузу.

*На компакт диске, в директории **\Примеры\Глава 25\Test1\Редактор** ты можешь увидеть исходник уже написанного примера.*

Теперь создадим окно создания нового теста вопросов ответов. Создай новую форму и установи у неё следующие свойства:

Caption – «*Тест "Вопрос - варианты ответов"*»

FormStyle *– fsMDIChild*, это у нас будет дочернее окно.
Name *– QuestionResultForm*.

По событию *OnClose* пишем следующий код:

```
procedure TQuestionResultForm.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
 Action:=caFree;
 QuestionResultForm:=nil;
end;
```

Здесь, в первой строке я устанавливаю переменной *Action* (эту переменную мы получаем в качестве параметра) значение *caFree*, чтобы окно могло закрыться. Дочерние окна по умолчанию не закрываются, а сворачиваются. Во второй строке я обнуляю переменную *QuestionResultForm*, которая указывает на объект окна.

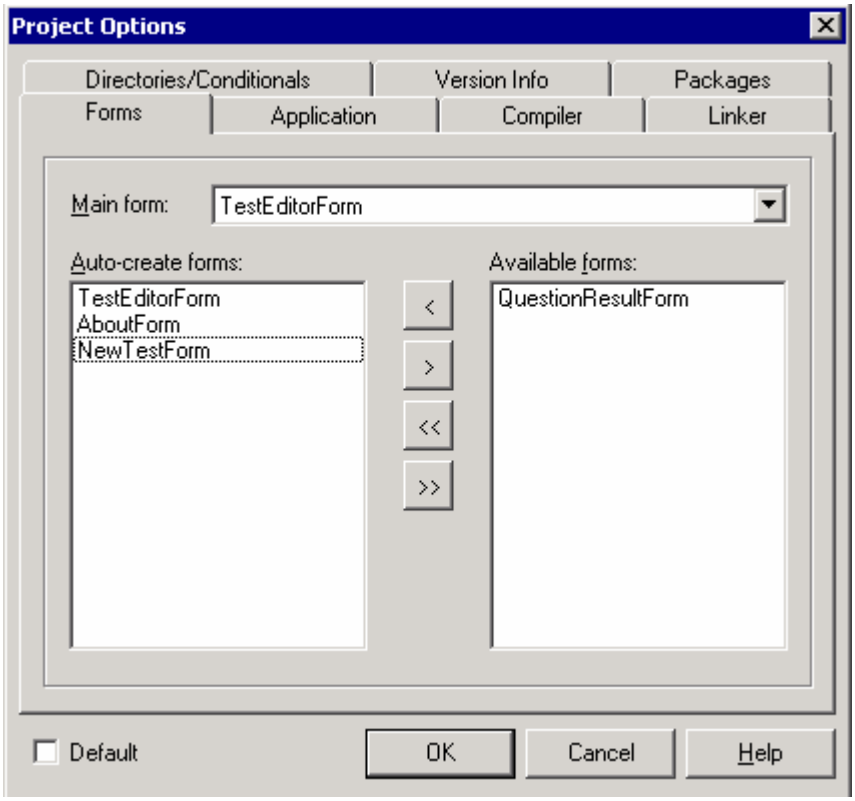


Рисунок 25.3.3 Окно свойств проекта

Дочерние окна при старте автоматически становятся видимыми и отображаются в окне главного окна. Нам это не надо, потому что это окно должно появляться только после того, как пользователь нажмёт кнопку создания нового теста и подтвердит его создание. Для того, чтобы отключить форму от автоматического создания, мы должны войти в свойства проекта (Project->Options) и в появившемся окне переместить имя формы *QuestionResultForm* из списка *Auto-create forms* в список *Available forms* (рисунок 25.3.4).

Вот теперь перейдём к дизайну формы создания теста. Мою форму ты можешь увидеть на рисунке 25.3.4. В центре окна у меня находиться компонент *TreeView*

растянутый по левому краю и *ListView* растянутый по всему оставшемуся контуру (clClient).Сверху окна расположена панель ToolBar со следующими кнопками:

1. Создание вопроса.
2. Редактирование.
3. Удаление.
4. Выход.

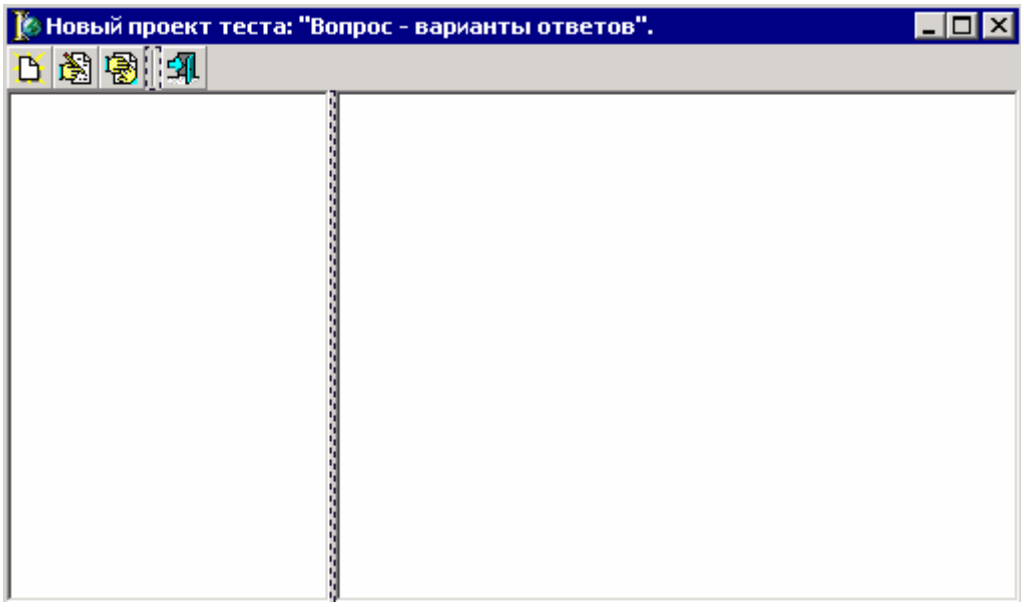


Рисунок 25.3.4 Форма окна создания теста

Выдели компонент ListView и установи у него следующие свойства:

1. *GridLines – true.*
2. *ViewStyle – vsReport.*
3. *Name - ResultView.*

После этого дважды щёлкни по свойству *Columns* и в появившемся окне создай две колонки с именами «Верно» и «Вариант ответа». У второй колонки установи свойство *AutoSize* в значение *true*.

Теперь объявим в разделе **type** следующую структуру:

```
PQuestion=^TQuestion;
TQuestion=record
 Name: String[255];
 ResultCount:Integer;
 ResiltText: array[0..10] of String[255];
 ResiltValue: array[0..10] of boolean;
end;
```

Я объявил структуру *TQuestion* со следующими полями:
Name – здесь будет храниться вопрос.
ResultCount - количество вариантов ответов.
ResiltText – массив из строк для десяти вариантов ответов.
ResiltValue – массив из булевых переменных, указывающих, какие ответы верные.

Тут же объявлена переменная *PQuestion*, которая является указателем на структуру *TQuestion*.

Теперь в разделе **public** объявим следующие переменные:

```
public
 { Public declarations }
 ProjectName:String[255];
 QuestionList:TList;
```

В переменной *ProjectName* будем хранить имя проекта. Список *QuestionList* будет использоваться для хранения структур типа *PQuestion*. Этот список должен инициализироваться по событию *OnShow* и уничтожаться по событию *OnClose*. Как это делать ты уже должен знать.

Теперь создаём форму для редактирования элементов, которую мы будет отображать на экране по нажатию кнопки «Создать вопрос» и «Редактировать вопрос». Мою форму ты можешь увидеть на рисунке 25.3.5.

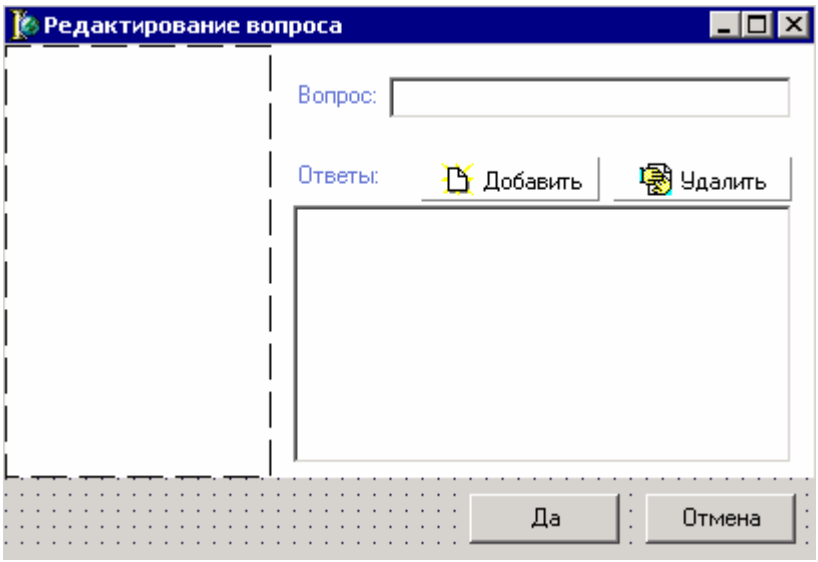


Рисунок 25.3.5 Форма окна ввода вопросов

Здесь находиться строка *TEdit*, для ввода текста вопроса и *TCheckListBox* для ввода вариантов ответа. По нажатию кнопки «*Добавить*» я буду показывать окно ввода нового варианта ответа:

```
procedure TEditQuestionForm.NewResultButtonClick(Sender: TObject);
var
 Str:String;
begin
 Str:='';
 if InputQuery('Новый ответ', 'Введите текст ответа:', Str) then
  ResultListBox.Items.Add(Str);
end;
```

Здесь я объявил одну строковую переменную. В первой строке кода ей присваивается пустая строка. Затем я отображаю на экране стандартный диалог ввода текстовой строки с помощью функции *InputQuery*. На рисунке 25.3.6 ты можешь видеть пример такого окна. У этой функции три параметра:

1. Текст, который будет отображаться в заголовке окна.
2. Текст, который будет отображаться в окне, рядом со строкой ввода.
3. Строковая переменная, через которую мы можем передать значение по умолчанию и получить результат ввода.

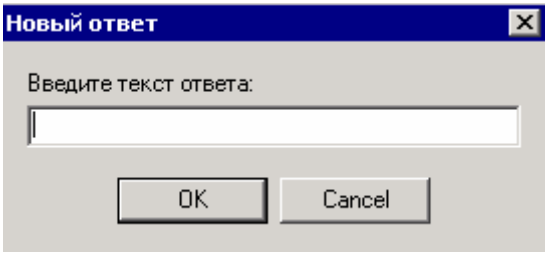


Рисунок 25.3.6 Форма окна создания теста

Если функция возвращает *true*, то пользователь после ввода нажал кнопку *OK*, и в этом случае я добавляю введённый текст в список ответов *ResultListBox*.

По нажатию кнопки «*Удалить*» я пишу следующий код:

```
procedure TEditQuestionForm.SpeedButton1Click(Sender: TObject);
begin
 if ResultListBox.ItemIndex<>-1 then
  ResultListBox.Items.Delete(ResultListBox.ItemIndex);
end;
```

В первой строчке кода я проверяю свойство *ItemIndex*, которое указывает на выделенный элемент в списке. Если оно не равно –1, значит, в списке есть выделенный элемент, и я его должен удалить. Для этого я выполняю *ResultListBox.Items.Delete* указывая в качестве параметра выделенную строку.

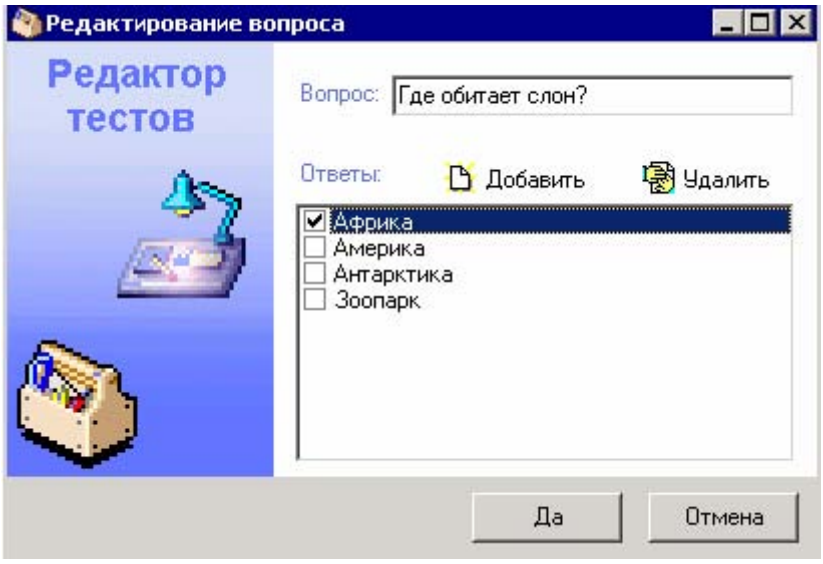


Рисунок 25.3.7 Окно редактирования вопроса в действии.

Теперь вернёмся к нашему окну *QuestionResultForm*. Здесь создадим обработчик события OnClick для кнопки создания нового вопроса:

```
procedure TQuestionResultForm.NewButtonClick(Sender: TObject);
var
 NewQuest:PQuestion;
 i:Integer;
begin
 //Очищаю содержимое окна EditQuestionForm
 EditQuestionForm.ResultListBox.Items.Clear;
 EditQuestionForm.QuestionEdit.Text:='';

 //Отображаю окно на экране
 EditQuestionForm.ShowModal;
 if EditQuestionForm.ModalResult<>mrOK then exit;

 //Создаю в памяти новую структуру
 NewQuest:=New(PQuestion);
 NewQuest.Name:=EditQuestionForm.QuestionEdit.Text;
 NewQuest.ResultCount:=EditQuestionForm.ResultListBox.Items.Count;

//Добавляю в структуру варианты ответов
 for i:= 0 to NewQuest.ResultCount-1 do
  begin
   NewQuest.ResiltText[i]:=EditQuestionForm.ResultListBox.Items.Strings[i];
   NewQuest.ResiltValue[i]:=EditQuestionForm.ResultListBox.Checked[i];
  end;
 QuestionList.Add(NewQuest);

//Добавляю новый элемент в дерево вопросов
 with QuestionTreeView.Items.Add(nil, NewQuest.Name) do
  begin
   ImageIndex:=0;
   Data:=NewQuest;
  end;
end;
```

В самом начале я очищаю элементы управления окна *EditQuestionForm*. Потом я отображаю это окно, и если пользователь ввёл название вопроса и нажал *OK*, то нужно обработать введённую информацию. Для начала выделяется память под переменную *NewQuest*. Эта переменная объявлена как *PQuestion*, а это указатель на структуру *TQuestion*. Как ты знаешь, любые указатели создаются пустыми и чтобы они на что-то указывали, им надо выделять память. Я выделяю память с помощью функции *New*. Этой функции нужно передать в качестве параметра подо что нужно выделять память. Я указываю наш указатель *PQuestion*, по которому функция определит, сколько памяти надо выделить. Результат выполнения функции – указатель на выделенную память, который я сохраняю в переменной *NewQuest*.

Если нужно уничтожить выделенную память, то мы должны вызвать процедуру *Dispose* и передать ей переменную, которую нужно уничтожить, например *Dispose(NewQuest)*. Но мне не надо уничтожать эту переменную, потому что она потом будет использоваться и я её добавляю в список *QuestionList* типа *TList*.

После того, как я выделил память для структуры *NewQuest*, я заполняю её поля в зависимости от введённой пользователем информации. Как только всё заполнено, я добавляю структуру в список:

```
QuestionList.Add(NewQuest);
```

После этого происходит самое интересное. Я должен создать новый элемент в дереве вопросов. Для этого выполняется следующий код:

```
with QuestionTreeView.Items.Add(nil, NewQuest.Name) do
 begin
  ImageIndex:=0;
  Data:=NewQuest;
 end;
```

Давай разберём этот код по частям. В первой строке я добавляю в дерево новый элемент с помощью вызова *QuestionTreeView.Items.Add*. В качестве параметров методу *Add* нужно передать указатель на родительский элемент в дереве и текст элемента. В качестве родительского элемента я передаю **nil** потому что я буду создавать дерево без вложенных элементов. В качестве текста элемента я передаю текст вопроса.

Выполненный метод *Add* возвращает указатель на созданный элемент. И тут у меня стоит оператор **with**, который заставляет выполнять следующие действия с указанным объектом. Получается, что следующие действия между **begin** и **end** будут выполняться с созданным элементом дерева вопросов. А у меня тут выполняется два действия:

ImageIndex:=0 – индексу иконки присваивается значение 0. Я бросил на форму список картинок *ImageList*, загрузил туда несколько картинок и указал этот список в свойстве *Images* нашего дерева. В этом коде я назначаю элементу первую картинку из созданного списка.

Data:=NewQuest – Свойство *Data* элемента дерева – это такое же свойство, как Tag у всех компонентов. Оно так же не влияет на работу компонента и его элементов и может использоваться в наших собственных целях. Это свойство имеет значение указателя, и мы можем в него вносить любые указатели. Я указываю здесь указатель на структуру *NewQuest*, которая связана с созданным элементом.

Теперь создадим обработчик события *OnChange* для нашего дерева:

```
procedure TQuestionResultForm.QuestionTreeViewChange(Sender: TObject;
  Node: TTreeNode);
var
 i:Integer;
begin
 //Очищаю список
 ResultView.Items.Clear;

 //Если не выделен элемент, то выход
 if Node=nil then exit;

//Запускаю цикл, по которому заполняются данные списка
 for i:=0 to PQuestion(node.Data).ResultCount-1 do
  with ResultView.Items.Add do
   begin
    Caption:=PQuestion(node.Data).ResiltText[i];
    if PQuestion(node.Data).ResiltValue[i]=true then
     begin
      SubItems.Add('Да');
      ImageIndex:=2;
     end
    else
     begin
      SubItems.Add('Нет');
      ImageIndex:=1;
```

```
    end;
  end;
end;
```

Это событие генерируется каждый раз, когда пользователь выбрал какой-нибудь элемент. По выбору вопроса мы должны заполнить ответы в списке *ListView*. Но прежде чем заполнять, я очищаю список, потому что он уже мог быть заполненным данными другого вопроса.

В обработчик события нам передаётся параметр *Node* типа *TTreeNode*, который указывает на выделенный элемент. Второй строчкой кода я проверяю, если выделенный элемент равен **nil** (ничего не выбрано), то нужно выходить из процедуры.

Чтобы получить доступ к структуре *PQuestion*, в которой хранятся данные об ответах выделенного вопроса, мы должны обратиться к свойству *Data* выделенного элемента *Node*. Как ты помнишь, в это свойство мы поместили указатель на структуру PQuestion. Но программа не может знать какого типа этот указатель, поэтому мы должны явно указывать это - *PQuestion(node.Data)*.

Далее, я запускаю цикл от 0 до количества вариантов ответов в данном вопросе *PQuestion(node.Data).ResultCount* минус 1. Внутри цикла я выполняю код *ResultView.Items.Add*, который создаёт очередной элемент списка. Здесь метод *Add* также возвращает указатель на созданный элемент, вместе с которым и будет выполняться дальнейший код (об этом говорит оператор **with**). А внутри кода я выполняю следующее:

1. Заполняю заголовок элемента *Caption*.
2. Если *PQuestion(node.Data).ResiltValue[i]* равно *true*, то есть ответ верный, то я добавляю дочерний элемент (текст этого элемента будет отображаться во второй колонке списка) SubItems.Add('Да') и присваиваю 2-й индекс иконки. Иначе текст дочернего элемента будет равен «*Нет*» и иконка будет иметь индекс единицы.

Таким образом, внутри цикла будет обработаны все варианты ответов, и все они будут добавлены в список. Попробуй сейчас запустить программу и создать пару вопросов с различными вариантами и посмотреть, как всё будет выглядеть. Моё окно программы ты можешь увидеть на рисунке 25.3.8.

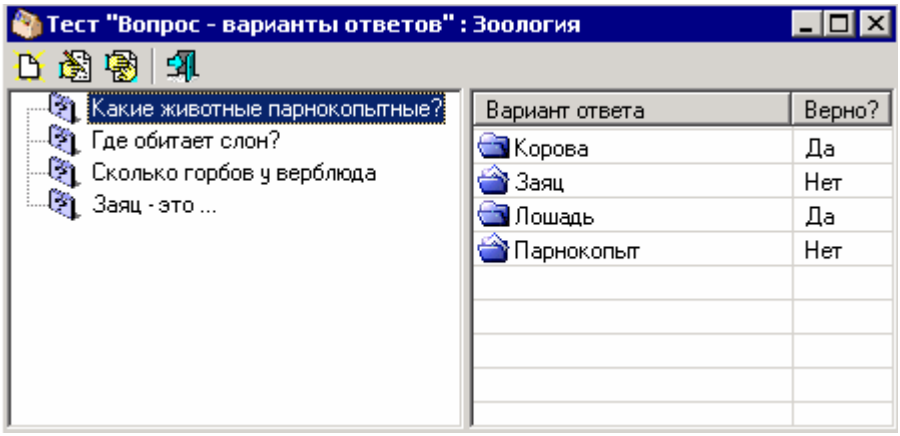


Рисунок 25.3.8 Рабочее окно прогрыммы.

Совсем забыл описать код, по которому мы будем отображать окно, показанное мной на рисунке 25.3.8. Для этого нам надо подкорректировать обработчик события *OnClick* для кнопки создания нового проекта теста:

```
procedure TTestEditorForm.NewButtonClick(Sender: TObject);
```

```
begin
 NewTestForm.ShowModal;
 if NewTestForm.ModalResult<>mrOK then exit;

 if NewTestForm.TestTypeBox.ItemIndex=0 then
  begin
   QuestionResultForm:=TQuestionResultForm.Create(Owner);
   QuestionResultForm.ProjectName:=NewTestForm.TestNameEdit.Text;
   QuestionResultForm.Caption:=QuestionResultForm.Caption+' : '
       +QuestionResultForm.ProjectName;
  end;
end;
```

Здесь в первой строке я показываю окно создания нового проекта. Если пользователь выбрал первый тип теста «*Вопрос - варианты ответа*» (в моей книге он будет описан как единственный), то создаётся окно, в котором мы создаём вопросы. Потом я сохраняю имя выбранного проекта и изменяю заголовок окна.

Обработчики события кнопок «*Редактировать*» и «*Удалить*» вопросы я расписывать не буду, а только приведу их код с комментариями. Ты уже должен разобраться с этим кодом:

```
procedure TQuestionResultForm.EditButtonClick(Sender: TObject);
var
 i:Integer;
begin
 //Здесь QuestionTreeView.Selected указывает на выделенный элемент
 //в дереве. Если он равен nil, то ничего не выделено, и нужно выйти
 if QuestionTreeView.Selected=nil then exit;

 //Заполняю компонент QuestionEdit в окне редактирования вопросов
 EditQuestionForm.QuestionEdit.Text:=PQuestion(QuestionTreeView.Selected.Data).Name;

 //Очищаю список вариантов ответов в окне редактирования вопросов
 EditQuestionForm.ResultListBox.Clear;
 for i:=0 to PQuestion(QuestionTreeView.Selected.Data).ResultCount-1 do
  begin
   //Заполняю список вариантов ответов в окне редактирования вопросов
   EditQuestionForm.ResultListBox.Items.Add(
         PQuestion(QuestionTreeView.Selected.Data).ResiltText[i]);

   //Если ответ верный, то ставлю галочку
   if PQuestion(QuestionTreeView.Selected.Data).ResiltValue[i]=true then
    EditQuestionForm.ResultListBox.Checked[i]:=true;
  end;

 //Отображаю окно редактирования вопроса
 EditQuestionForm.ShowModal;
 if EditQuestionForm.ModalResult<>mrOK then exit;

//Записываю информацию обратно в структуру
 PQuestion(QuestionTreeView.Selected.Data).Name:=EditQuestionForm.QuestionEdit.Text;
 PQuestion(QuestionTreeView.Selected.Data).ResultCount:=
     EditQuestionForm.ResultListBox.Items.Count;
 for i:= 0 to PQuestion(QuestionTreeView.Selected.Data).ResultCount-1 do
  begin
   PQuestion(QuestionTreeView.Selected.Data).ResiltText[i]:=
      EditQuestionForm.ResultListBox.Items.Strings[i];
   PQuestion(QuestionTreeView.Selected.Data).ResiltValue[i]:=
```

```
    EditQuestionForm.ResultListBox.Checked[i];
 end;

 //Вызываю процедуру QuestionTreeViewChange, которая должна обновить
 //информацию в ResultView. Первый параметр нас не интересует, а второй
 //мы обязаны указать, потому что внутри процедуры QuestionTreeViewChange
 //мы используем его. Я указываю выделенный элемент.
 QuestionTreeViewChange(nil, QuestionTreeView.Selected);
end;
```

Единственное, что я здесь хочу отметить, так это то, что здесь я обращаюсь к структуре связанной с элементом через свойство *Data* выделенного элемента. Как я уже говорил, там храниться указатель на структуру. То же самое можно было бы делать, обращаясь через контейнер, просто в данном случае это будет не так удобно. Но всё же это возможно и на всякий случай я приведу пример, как можно обратиться к свойству *Name*:

```
PQuestion(QuestionList[QuestionTreeView.Selected.Index]).Name
```

Здесь я использую контейнер *QuestionList*. В квадратных скобках у него я указываю индекс элемента из контейнера, который мне нужен. Здесь я указываю индекс выделенного в дереве элемента *QuestionTreeView.Selected.Index*.

По нажатию кнопки «*Удалить*» ты должен написать следующий код:

```
procedure TQuestionResultForm.DeleteButtonClick(Sender: TObject);
var
 index, i:Integer;
begin
 if QuestionTreeView.Selected=nil then exit;

 //Подтверждение удаления
 if Application.MessageBox(PChar('Вы действительно хотите удалить - '+
  QuestionTreeView.Selected.Text), 'Внимание!!!',
   MB_OKCANCEL+MB_ICONINFORMATION)<>idOk then Exit;

 //Сохраняю индекс выделенного элемента
 index:=QuestionTreeView.Selected.Index;

 //Удаляю выделенный элемент из дерева
 QuestionTreeView.Items.Delete(QuestionTreeView.Selected);

 //Удаляю из контейнера
 QuestionList.Delete(Index);
end;
```

 *На компакт диске, в директории **\Примеры\Глава 25\Test2\Редактор** ты можешь увидеть исходник уже написанного примера.*

25.4. Сохранение и загрузка теста.

Твоя программа уже умеет создавать тесты, пора бы её научить и сохранять их и тем более загружать потом созданные проекты для редактирования. Я вынес кнопки открытия и сохранения проекта из дочернего окна в основное. Если честно, то сохранение легче сделать внутри дочернего окна, а открытие в главном. Но я не пошёл простым путём, потому что хочу тебе показать, как работать с дочерними окнами.

Итак, по нажатию кнопки «Сохранить» проект пишем следующий код:

```
procedure TTestEditorForm.SaveButtonClick(Sender: TObject);
begin
 //Если актиыное дочернее окно равно нулю
 //(нет активных окон), то выход
 if ActiveMDIChild=nil then exit;
 //Если окно имеет имя QuestionResultForm, то это
 //вопрос-варианты ответов и вызываем для сохранения
 //процедуру SaveTest1.
 if ActiveMDIChild.Name='QuestionResultForm' then
  SaveTest1;
end;
```

Свойство *ActiveMDIChild* всегда указывает на активное в данный момент дочернее окно. Прежде чем использовать это свойство, его желательно сравнивать со значением **nil**, потому что в данный момент может вообще не быть ни одного дочернего окна. В этом случае, при обращении к свойству может произойти критическая ошибка.

Процедура *SaveTest1* должна выглядеть следующим образом:

```
procedure TTestEditorForm.SaveTest1;
var
 fs:TFileStream;
 i:Integer;
 Str:String[5];
begin
 //Если у активного окна в свойстве FileName пусто,
 //то нет имени файла и нужно вызвать обработчик события
 //меню "Сохранить как...", чтобы появилось окно ввода
 //имени файла
 if TQuestionResultForm(ActiveMDIChild).FileName='' then
  begin
   SaveAsMenuClick(nil);
   exit;
  end;

 //Создаю новый файл. Если он уже существовал, то его
 //содержимое будет уничтожено
 fs:=TFileStream.Create(TQuestionResultForm(ActiveMDIChild).FileName, fmCreate);

 //Сохраняю в начале файла текст "Тест", чтобы по нему потом
 //определить к чему относиться данный файл.
 Str:='Тест';
 fs.Write(Str, SizeOf(Str));

 //Сохранить имя проекта
 fs.Write(TQuestionResultForm(ActiveMDIChild).ProjectName,
      sizeof(TQuestionResultForm(ActiveMDIChild).ProjectName));
 try
  //Сохранить количество вопросов
  fs.Write(TQuestionResultForm(ActiveMDIChild).QuestionList.Count,
```

```
      sizeof(TQuestionResultForm(ActiveMDIChild).QuestionList.Count));

  //Запускаю цикл, в котором сохраняются все вопросы.
  for i:=0 to TQuestionResultForm(ActiveMDIChild).QuestionList.Count-1 do
   fs.Write(PQuestion(TQuestionResultForm(ActiveMDIChild).QuestionList[i])^,
     sizeof(TQuestion));
 finally
  //Закрыть файл
  fs.Free;
 end;
end;
```

Здесь всё очень просто и с кодом можно разобраться по комментариям. Единственное, на что я хочу обратить внимание – наша структура *PQuestion* находится в динамической памяти, поэтому при сохранении нужно указывать знак разыменования ^. Если этого знака не указать, то в файл сохраниться адрес структуры, а не сама структура. В этом случае при чтении данных из файла мы прочитаем адрес, но по этому адресу ничего хорошего не будет, потому что после первой же перезагрузки программы память очиститься и сама структура уничтожиться. Поэтому, для сохранения данных по адресу, а не самого адресу нужно указывать знак ^.

Обработчик события для пункта меню «*Сохранить как…*» ещё проще:

```
procedure TTestEditorForm.SaveAsMenuClick(Sender: TObject);
begin
 if SaveDialog1.Execute then
  begin
   TQuestionResultForm(ActiveMDIChild).FileName:=SaveDialog1.FileName;
   SaveButtonClick(nil);
  end;
end;
```

Здесь я отображаю окно выбора имени файла. Если пользователь что-то выбрал, то сохраняю имя файла в свойстве *FileName* активного окна, и вызываю обработчик события кнопки «*Сохранить*», где происходит сохранение.

Теперь посмотри на обработчик события *OnClick* для кнопки «Открыть» проект:

```
procedure TTestEditorForm.OpenButtonClick(Sender: TObject);
var
 fs:TFileStream;
 i, Count:Integer;
 Str:String[5];
 NewQuest:PQuestion;
begin
 //Показать окно открытия файла
 if not OpenDialog1.Execute then exit;

 //Открыть файл для чтения
 fs:=TFileStream.Create(OpenDialog1.FileName, fmOpenRead);

 //Перейти в начало файла и прочитать заголовок
 fs.Seek(0,soFromBeginning);
 fs.read(Str, SizeOf(Str));

 //Если заголовок равен тексу "Тест", значит это "вопрос-
```

```
//варианты ответов".
if Str='Тест' then
 begin
  //Создать новое окно теста
  QuestionResultForm:=TQuestionResultForm.Create(Owner);

  //Сохранить имя открытого файла в объекте окна
  QuestionResultForm.FileName:=OpenDialog1.Filename;

  //Прочитать имя проекта
  fs.Read(QuestionResultForm.ProjectName, sizeof(QuestionResultForm.ProjectName));

  try
   //Прочитать количество вопросов
   fs.Read(Count, sizeof(Count));

   //Запустить цикл чтения вопросов
   for i:=0 to Count-1 do
    begin
     //Создаю новую структуру в памяти для вопроса
     NewQuest:=New(PQuestion);
     //Читаю структуру
     fs.Read(NewQuest^, sizeof(TQuestion));

     //Добавляю структуру в контейнер
     QuestionResultForm.QuestionList.Add(NewQuest);

     //Создаю новый элемент в дереве
     with QuestionResultForm.QuestionTreeView.Items.Add(nil, NewQuest.Name) do
      begin
       ImageIndex:=0;
       Data:=NewQuest;
      end;
    end;
  finally
   //Закрываю файл
   fs.Free;
  end;
 end;
end;
```

В чтении файла так же ничего сложного нет. Всё очень похоже на запись и со всеми методами ты уже должен быть знаком. Здесь так же мы читаем данные в указатель на структуру *PQuestion*, поэтому при чтении нужно разыменовывать указатель *NewQuest^*, чтобы данные записались «по адресу», а не в адрес.

*На компакт диске, в директории **\Примеры\Глава 25\Test3\Редактор** ты можешь увидеть исходник уже написанного примера.*

Вот на этом наш редактор можно считать законченным. Хотя ещё не реализованы обработчики события для кнопок печати и свойств проекта. Но свойства проекта нам не нужны, а вот печать я оставлю тебе. Попробуй сам добавить вывод на печать нашего проекта.

25.5. Тестер.

Теперь напишем программу тестирования, которая будет загружать наши проекты, отображать вопросы и собирать статистику правильных ответов. Для этого у нас будет отдельная программа, поэтому создай новый проект и установи на форму следующие компоненты (мою форму ты можешь увидеть на рисунке 25.5.1):

1. Панель *ToolBar* с тремя кнопками «Открыть», «Запустить» и «Выход».
2. Компонент *StaticText*, где будем отображать вопросы. В свойстве *Name* укажи *QuestionLabel* и свойство *AutoSize* установи в *false*.
3. Список *CheckListBox* в котором будут отображаться варианты ответов. В свойстве *Name* укажи *QuestionCheckList*.
4. Ну и последнее - кнопку «Дальше».

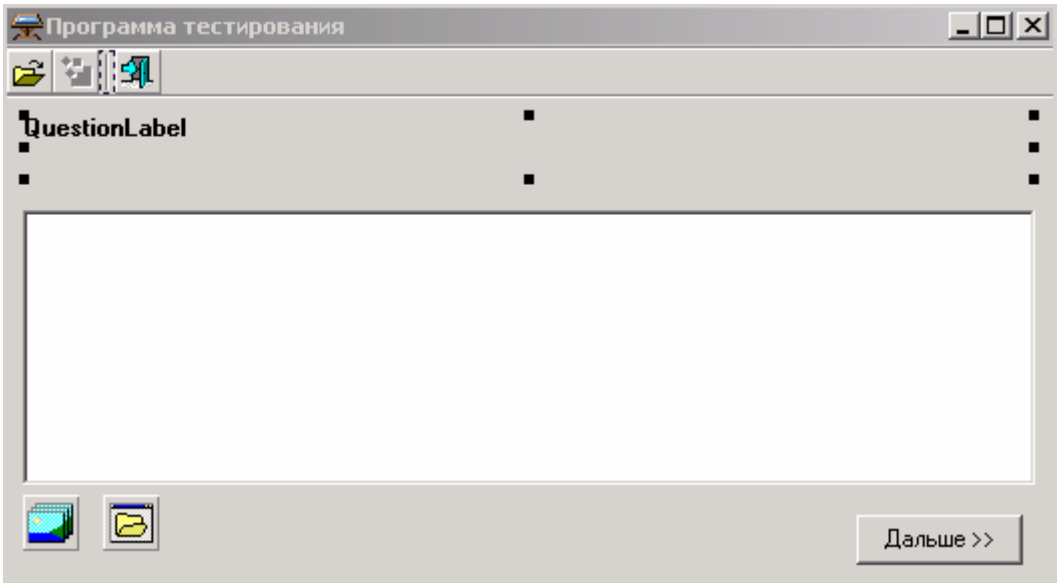


Рисунок 25.5.1 Форма будущей программы.

В разделе **type** объяви структуру *TQuestion*, такого же вида, как и в редакторе вопросов. Количество и размерность полей структуры должно быть одинаково, потому что мы будем использовать её для загрузки данных из файла. Если хоть какое-то поле будет отличаться, то при загрузке данных произойдёт ошибка.

```
type
  PQuestion=^TQuestion;
  TQuestion=record
   Name: String[255];
   ResultCount:Integer;
   ResiltText: array[0..10] of String[255];
   ResiltValue: array[0..10] of boolean;
  end;
```

В разделе **private** объяви следующие переменные:

```
private
 { Private declarations }
 QuestionList:TList;
 Question, QuestionNumber, FalseNumber:Integer;
```

```
 FileName:String;
```

Разберём, для чего нужны эти переменные:
QuestionList – здесь будет храниться список вопросов, как и у редактора вопросов.
Question – будет отображать текущий вопрос, на который отвечает испытуемый.
QuestionNumber – здесь мы будем хранить количество вопросов, на которые уже даны ответы. Нам же надо иметь счётчик, после которого тест должен закончиться.
FalseNumber – количество неправильных ответов.

Теперь создадим обработчик события *OnShow* для главной формы. В этом обработчике нужно инициализировать список *QuestionList*:

```
procedure TTestForm.FormShow(Sender: TObject);
begin
 QuestionList:=TList.Create;
end;
```

По событию *OnDestroy* мы должны уничтожить этот объект:

```
procedure TTestForm.FormDestroy(Sender: TObject);
begin
 QuestionList.Free;
end;
```

Теперь для кнопки открытия пишем следующий код:

```
procedure TTestForm.OpenButtonClick(Sender: TObject);
begin
 //Показать окно открытия файла
 if not OpenDialog1.Execute then exit;
 FileName:=OpenDialog1.Filename;
 RunButton.Enabled:=true;
end;
```

В первой строке я отображаю окно открытия файла. Если пользователь нажал на кнопку «Отмена», то происходит выход из процедуры. Иначе, в переменной *FileName* сохраняется имя выбранного файла. В принципе, этого можно было и не делать, потому что имя файла хоть как останется в свойстве *OpenDialog1.Filename*, но я всё же завёл отдельную переменную, и буду хранить имя файла там.
В последней строке я делаю кнопку «Запустить» *RunButton* доступной. Кстати, на форме эта кнопка должна быть не доступной, чтобы при старте программы, пользователь не мог нажать кнопку «Запустить», пока не выберет файл.
Теперь пишем обработчик события *OnClick* для кнопки «Запустить»:

```
procedure TTestForm.RunButtonClick(Sender: TObject);
begin
```

```
 LoadFile;
 QuestionNumber:=0;
 FalseNumber:=0;
 NextButton.Enabled:=true;
 NextQuestion;
end;
```

В первой строке я вызываю процедуру *LoadFile*, которую я напишу чуть позже, и она будет загружать список вопросов из выбранного файла проекта. Почему я должен загружать вопросы каждый раз при старте программы? Да потому что тест будет происходить следующим образом:

1. Из списка вопросов случайным образом выбирается первый попавшийся вопрос.
2. Пользователь отвечает на него, и мы удаляем его из списка. Таким образом, в следующий раз, когда мы будем выбирать вопрос из списка, то мы уже точно уверены, что в списке нет вопроса, на который бы уже отвечал пользователь.
3. При следующем старте теста список вопросов инициализируется заново (мы снова загружаем весь список) и все вопросы возвращаются на свои места.

После загрузки вопросов я обнуляю все переменные, и делаю доступной кнопку *NextButton* (это кнопка «Дальше», по нажатию которой будет выбираться следующий вопрос). При старте программы кнопка «Дальше» должна быть недоступной.

В последней строке я вызываю процедуру *NextQuestion*, которая и будет выбирать случайный вопрос и отображать его в окне программы.

Теперь посмотрим на процедуру загрузки вопросов *LoadFile*. Она идентична уже написанной процедуре загрузки в программе редактора вопросов:

```
procedure TTestForm.LoadFile;
var
 fs:TFileStream;
 i, Count:Integer;
 Str:String[5];
 ProjectName:String[255];
 NewQuest:PQuestion;
begin
 QuestionList.Clear;
 //Открыть файл для чтения
 fs:=TFileStream.Create(FileName, fmOpenRead);

 //Перейти в начало файла и прочитать заголовок
 fs.Seek(0,soFromBeginning);
 fs.read(Str, SizeOf(Str));

 //Если заголовок равен тексу "Тест", значит это "вопрос-
 //варианты ответов".
 if Str='Тест' then
  begin
   //Прочитать имя проекта
   fs.Read(ProjectName, sizeof(ProjectName));
   Caption:=ProjectName;

   try
    //Прочитать количество вопросов
    fs.Read(Count, sizeof(Count));

    //Запустить цикл чтения вопросов
    for i:=0 to Count-1 do
     begin
```

```
    //Создаю новую структуру в памяти для вопроса
    NewQuest:=New(PQuestion);
    //Читаю структуру
    fs.Read(NewQuest^, sizeof(TQuestion));

    //Добавляю структуру в контейнер
    QuestionList.Add(NewQuest);
   end;
  finally
   //Закрываю файл
   fs.Free;
  end;
 end;
end;
```

Тут всё должно быть понятно, но я на всякий случай снабдил весь код подробными комментариями.

Теперь посмотрим на процедуру *NextQuestion*, которая должна случайным образом выбирать вопрос из списка:

```
procedure TTestForm.NextQuestion;
var
 i:Integer;
begin
 Randomize;
 Question:=Random(QuestionList.Count-1);

 QuestionLabel.Caption:=PQuestion(QuestionList[Question]).Name;

 QuestionCheckList.Items.Clear;
 for i:=0 to PQuestion(QuestionList[Question]).ResultCount-1 do
  QuestionCheckList.Items.Add(PQuestion(QuestionList[Question]).ResiltText[i]);

 Inc(QuestionNumber);
end;
```

В первой строке процедуры я вызываю процедуру *Randomize*, которая инициализирует таблицу случайных чисел. Если ты опустишь вызов этой процедуры, то ничего страшного не произойдёт, и когда ты будешь запрашивать случайное число, то оно будет случайным, но всё же лучше инициализировать таблицу. Просто в этом случае считается, что случайность будет более случайной (во как звучит!!!).

Во второй строке я вызываю функцию *Random*, которая возвращает случайное число. Ей нужно передать в качестве параметра максимально допустимое число. Я передаю *QuestionList.Count–1*, т.е. количество вопросов в нашем списке. Функция вернёт мне случайное число от 0 до указанного числа. Я сохраняю это число в переменной *Question*.

В следующей строке кода я показываю в компоненте *QuestionLabel* вопрос соответствующий вопрос. Затем очищаю список ответов в компоненте *QuestionCheckList* и заполняю его вариантами ответов, относящихся к данному вопросу. В последней строке кода я увеличиваю переменную *QuestionNumber*, в которой у нас храниться количество отвеченных вопросов.

По нажатию кнопки «*Далее*» пишем следующий код:

```
procedure TTestForm.NextButtonClick(Sender: TObject);
var
 OK:Boolean;
 i:Integer;
begin
 OK:=true;

 for i:=0 to PQuestion(QuestionList[Question]).ResultCount-1 do
  if PQuestion(QuestionList[Question]).ResiltValue[i]<>QuestionCheckList.Checked[i] then
   OK:=false;

 if OK=false then
  Inc(FalseNumber);

 //Удаление вопроса из списка
 QuestionList.Delete(Question);

 if QuestionNumber<5 then
  NextQuestion
 else
  begin
   Application.MessageBox(PChar('Вы закончили тест с количеством ошибок = '+
      IntToStr(FalseNumber)), 'Внимание!!!');
   NextButton.Enabled:=false;
  end;
end;
```

В первой строке здесь устанавливается логическая переменная *OK* в значение *true*. В этой переменной мы будем хранить состояние результата ответа. По умолчанию будем считать, что ответ правильный, поэтому и устанавливаем значение *true*.

Далее, запускаю цикл от 0 до количества вариантов ответов в списке. Внутри цикла я сравниваю значение правильных ответов с состоянием свойство *Checked* компонента *QuestionCheckList*. Если хоть что-то не совпадает, то тестируемый где-то ошибся и нужно установить переменную *OK* в значение *false*, т.е. ответ неверный. После цикла происходит проверка, если переменная *OK* равна *false*, то увеличиваем счётчик неправильных ответов *FalseNumber* на единицу.

Всё, текущий вопрос нам больше в списке не нужен, и его нужно удалить, чтобы он больше не появился, когда мы будем случайным образом получать следующий вопрос.

Дальше происходит проверка, если количество отображённых вопросов меньше 5, то выбираем следующий вопрос (вызываем процедуру *NextQuestion*), иначе отображаем сообщение с состоянием пройденного теста и делаем кнопку «*Далее*» недоступной.

Как видишь, мой тест состоит из 5 вопросов, если тебе нужно больше, то можешь увеличить это значение. Но у нас в редакторе вопросов есть кнопка свойств, по нажатию которой можно отображать окно свойств проекта. Я бы сделал возможность в этом окне выбирать количество вопросов, на которые должен ответить испытуемый. Потом эти свойства можно сохранить в файл проекта и загружать в нашей программе теста. Но всё это я делать не буду, потому что у тебя уже есть достаточно знаний, чтобы попробовать всё это сделать самому.

 *На компакт диске, в директории* ***\Примеры\Глава 25\Test4*** *ты можешь увидеть исходник уже написанного примера.*