Объектно-ориентированное программирование на языках Delphi и C++

Учебное пособие для студентов

Авторы: А.Н. Вальвачев, К.А. Сурков,
Д.А. Сурков, Ю.М. Четырько

Содержание

Часть 1. Программирование на языке Delphi

Предисловие

Программисты всегда старались сделать свою жизнь более легкой, изобретая и совершенствуя технологии программирования, и на этом поприще им удалось одержать ряд действительно крупных побед. Попытаемся проследить развитие технологий программирования, чтобы читатель понял, почему авторы взялись за написание книги по языку Delphi.

Краткий экскурс в историю

Языки программирования

Пожалуй, наиболее важной вехой в истории программирования, сравнимой по значимости разве что с изобретением письменности, можно считать переход от машинных кодов (тарабарщины типа 0110110101111...) к понятным простому смертному языкам программирования (типа ALGOL, FORTRAN, PL/1, Pascal), а также к широкому использованию методов структурного программирования. Программы стали модульными, состоящими из подпрограмм. Появились библиотеки готовых подпрограмм, облегчающие многие задачи, но все равно программистам хватало трудностей, особенно при разработке пользовательского интерфейса.

Объектно-ориентированное программирование

Качественным шагом в развитии методов структурного программирования стало изобретение объектно-ориентированного программирования (языков SmallTalk, C++, Turbo Pascal и др.). Программы стали строиться не из чудовищных по размеру процедур и функций, перерабатывающих громоздкие структуры данных, а из сравнительно простых кирпичиков-объектов, в которых были упрятаны данные и подпрограммы их обработки. Гибкость объектов позволила очень просто приспосабливать их для собственных целей, прилагая для этого минимум усилий. Программисты обзавелись готовыми библиотеками объектов, но, как и раньше, создание пользовательского интерфейса требовало уйму времени и сил, особенно когда программа должна была работать под управлением популярной операционной системы Windows и иметь графический пользовательский интерфейс.

Визуальное программирование

С изобретением визуального программирования, первой ласточкой которого была среда разработки Visual Basic, создание графического пользовательского интерфейса стало под силу даже новичку. В среде Visual Basic можно было быстро создать приложение для операционной системы Windows, в котором были все присущие графическому пользовательскому интерфейсу элементы: окна, меню, кнопки, поля ввода и т.д. Все эти элементы превратились в строительные блоки программы — компоненты — объекты, имеющие визуальное представление на стадии проектирования и во время работы.

Проектирование пользовательского интерфейса упростилось на порядок, однако, для профессиональных программистов язык Basic оказался явно слабоват. Отсутствие в нем контроля типов данных и механизма их расширения оказалось камнем преткновения на пути создания серьезных программ. Создание нестандартных компонентов в среде Visual Basic было крайне затруднено (для этого приходилось прибегать к другим средствам разработки, в частности, к языку C++). В общем, среда Visual Basic отлично подходила для создания прототипов приложений, но не для разработки коммерческих программных продуктов.

Среда программирования Delphi

Мечта программистов о среде программирования, в которой бы простота и удобство сочетались с мощью и гибкостью, стала реальностью с появлением среды Delphi. Она обеспечивала визуальное проектирование пользовательского интерфейса, имела развитый объектно-ориентированный язык Object Pascal (позже переименованный в Delphi) и уникальные по своей простоте и мощи средства доступа к базам данных. Язык Delphi по возможностям значительно превзошел язык Basic и даже в чем-то язык C++, но при этом он оказался весьма надежным и легким в изучении (особенно в сравнении с языком C++). В результате, среда Delphi позволила программистам легко создавать собственные компоненты и строить из них профессиональные программы. Среда оказалась настолько удачной, что по запросам любителей C++ была позже создана среда C++Builder — клон среды Delphi на основе языка C++ (с расширенным синтаксисом).

Среда Delphi стала, по сути, лучшим средством программирования для операционной системы Windows, но программистов ждало разочарование, если возникало желание перенести программу в другую операционную систему, в частности, в операционную систему Unix.

Технология Java

Практически одновременно со средой программирования Delphi на свет появилась технология Java, включавшая три составляющих: одноименный язык программирования, очень похожий на язык C++, но более простой и безопасный; универсальный байт-код, в который компилировались программы на языке Java; интерпретатор (виртуальную машину) для выполнения байт-кода в любой операционной системе. Благодаря автоматическому

управлению памятью — так называемой «сборке мусора» — резко повысилась надежность программ и скорость их разработки.

Поначалу на технологию Java возлагались большие надежды. Программные библиотеки для языка Java стали единым стандартом, поэтому написанные на нем программы оказались по-настоящему переносимыми. Однажды написанная и компилированная в байт-код программа могла работать на любой платформе без ограничений (единственное требование — наличие на этой платформе виртуальной машины Java).

Безграничная переносимость Java-программ родила идею сетевого компьютера и сетевых вычислений, суть которой в том, что все программы хранятся в байт-коде на серверах сети Интернет. Когда подключенный к сети пользователь запускает программу, то она сначала загружается к нему на компьютер, а затем интерпретируется. Охваченные этой идеей крупные фирмы ринулись осваивать новый рынок Java-приложений. Для языка Java появились средства визуального программирования, такие как JBuilder и Visual Age for Java. Казалось бы, бери и используй их для разработки пользовательского интерфейса и серверных программ. Но практически пропускная способность сети Интернет в лучшем случае обеспечивала оперативную загрузку на клиентские компьютеры лишь небольших по размеру программ. Кроме того, созданный на языке Java пользовательский интерфейс хронически отставал от возможностей операционной системы Windows и раздражал своей медлительностью. Поэтому технологию Java стали применять главным образом для разработки серверных приложений. Однако и здесь цена переносимости программ оказалась очень высокой — представленные в байт-коде программы работали на порядок медленнее аналогичных программ, компилированных напрямую в команды процессора. Применение динамической компиляции, при которой программа перед выполнением преобразуется из байт-кода в команды процессора и попутно оптимизируется, улучшило положение дел, но скорость работы Java-приложений так и не смогла приблизиться к скорости работы традиционных приложений. Иными словами, переносимость программ шла в ущерб их производительности и удобству. Многие начали задумываться над целесообразностью такой переносимости программ вообще.

Тем временем назревала революция в области серверных платформ — небывалыми темпами росла популярность операционной системы Linux.

Среда программирования Kylix

В связи со стремительным распространением операционной системы Linux возникла необходимость в эффективных средствах создания для нее программ. Таким средством стала среда Kylix (произносится «киликс») — первая среда визуального программирования для операционной системы Linux. Среда Kylix явилась полным аналогом среды Delphi и была совместима с ней по языку программирования и библиотекам компонентов. Программу, созданную в среде Delphi, можно было без изменений компилировать в среде Kylix, и наоборот. Эта возможность достигалась за счет новой библиотеки компонентов, которая взаимодействовала с операционной системой не напрямую, а через промежуточный программный слой, скрывающий разницу в работе компонентов в той или иной операционной системе. Программисты получили возможность создавать программы сразу для двух самых популярных операционных систем: Windows и Linux. Фактически вместо принципа абсолютной переносимости программ была предложена идея разумной переносимости.

Постепенно пришло понимание того, что в эпоху Интернет способность программ к взаимодействию в сети не менее (а порой более!) важна, чем возможность их переноса на различные платформы. Такая способность была обеспечена за счет стандартизации протоколов обмена данными в сети Интернет и форматов этих данных. Развитие протоколов и стандартов Интернет привело к рождению технологии Web-сервисов, которая ставила своей задачей максимально упростить создание программ, взаимодействующих по принципу

клиент-сервер в глобальной сети. Поддержка технологии Web-сервисов была изящно встроена в системы Delphi и Kylix, в результате разработчики программ получили в руки еще один очень важный инструмент.

Технология .NET

Несмотря на трудности и уроки Java-технологии, программисты не желали отказываться от идеи создания полностью переносимых программ. Вместе с тем их совершенно не устраивала необходимость платить производительностью и удобством программ за переносимость. Работы по разрешению этого противоречия привели к появлению на свет технологии под названием .NET (произносится «дот-нет»).

Технология .NET по сути явилась новой платформой, надстроенной над другими операционными системами, и этим походила на технологию Java. Однако у технологии .NET имелся ряд существенных концептуальных отличий. В частности, платформа .NET хотя и имела свой собственный новый язык программирования C# (произносится «си-шарп»), но не была привязана только к нему, позволяя писать программы на других языках. Кроме того, программы для платформы .NET компилировались не в байт-код, а в универсальный промежуточный язык, который сохранял семантику программы и был близок к ее исходному тексту (байт-код, напротив, близок к командам процессора). Программы на промежуточном языке вообще не интерпретировались, а всегда компилировались в команды процессора при запуске программы или при ее первоначальной установке на компьютер пользователя. Выполняемый код получался очень эффективным и оказывался сравнимым по быстродействию с выполняемым кодом, полученным прямой компиляцией с языка высокого уровня в команды процессора. Немаловажно и то, что на платформе .NET стало возможным использование любых (а не только стандартных) библиотек подпрограмм и компонентов, а также всех функций операционной системы. Все это обеспечило создание быстрых и удобных программ.

Поначалу технология .NET была доступна только для семейства операционных систем Windows, но со временем этот недостаток был устранен, и на свет появилась платформа Mono — клон технологии .NET для операционных систем Linux и Unix.

... и опять среда Delphi

Платформы .NET и Mono имеют большое будущее, поэтому фирма Borland адаптировала для них язык и среду программирования Delphi. В итоге, разработчики получили уникальную возможность — применять один и тот же язык Delphi для создания профессиональных программ для любых операционных систем и платформ: Windows, Linux, .NET, Mono. Этим, кстати, язык Delphi выгодно отличается от модного ныне языка C#, который применяется лишь для программирования на платформах .NET и Mono.

У языка Delphi есть еще одно очень важное преимущество перед остальными коммерчески успешными языками — он великолепно подходит для обучения программированию. Поэтому авторы рекомендуют его в качестве первого языка для всех учеников и студентов, собирающихся стать профессиональными программистами.

Что вы найдете в этой книге

Уважаемый читатель, мы смеем утверждать, что в этой книге есть именно то, что нужно человеку, чтобы научиться писать программы на языке Delphi и стать профессионалом в этой области. Почерпнутые из книги знания понадобятся вам независимо от того, для какой платформы (Windows, Linux, .NET или Mono) вы будете программировать.

В этой книге вы найдете:

- понятное каждому объяснение принципов визуального программирования, которого нет ни в одной другой книге. Оно поможет вам понять современный подход к

программированию и технологию создания графического пользовательского интерфейса;

- не имеющее аналогов по полноте и простоте описание языка Delphi и объектно-ориентированного программирования. Это фундамент среды Delphi, без знания которого заниматься программированием не имеет смысла;
- готовые решения многих проблем (с исходными текстами!), с которыми каждый программист рано или поздно встретится на практике: от построения простейших меню до пошаговой реализации мультимедиа-систем и создания приложений, работающих с базами данных. Принцип подачи материала везде один: от простого — к сложному.

При написании этой книги авторы использовали материалы своих предыдущих книг по системам программирования Delphi и C++Builder. По сравнению с предыдущими книгами существенной переработке подверглась терминология. Описание языка Delphi расширено и теперь охватывает такие аспекты, как динамические массивы и их физическое представление в памяти, перегрузку процедур и функций, новые форматы кодирования символов строк и многое другое. Новый набор стандартных подпрограмм рассмотрен самым подробным образом. Объектно-ориентированное программирование объясняется на более понятном сквозном примере. Отдельная глава посвящена расширению возможностей объектно-ориентированного программирования с использованием технологии интерфейсов. Глава, посвященная созданию динамически подключаемых библиотек, охватывает дополнительно вопросы создания динамически подключаемых пакетов компонентов. Если вы посмотрите оглавление, то получите более полное представление об обширной тематике этой книги.

Книга может использоваться как для самообучения, так и для чтения лекций и организации лабораторных занятий, поскольку методика подачи материала проверена на тысячах студентов и доказала свою эффективность.

Возможные трудности освоения

Основная проблема в освоении среды Delphi — это ее гигантский объем. Новичок просто теряется в этом море поистине безграничных возможностей. Еще одна сложность — это понимание объектно-ориентированного программирования. Практика показывает, что многие осваивают эту тему только после второго-третьего прочтения и глубокого анализа примеров. Но паниковать не стоит. Читайте книгу не спеша, выполняйте на компьютере все примеры, и успех обязательно придет.

Глава 1. Основы визуального программирования

Перед тем, как отправиться в дальнюю дорогу, мудрый путник намечает цель путешествия, рассчитывает свои силы, прикидывает маршрут и готовит снаряжение. После этого он совершает маленький тренировочный поход, чтобы проверить надежность снаряжения и получить необходимый навык. Давайте уподобимся мудрецу и, начиная виртуальное путешествие в страну Delphi, сделаем то же самое: поставим перед собой цель научиться писать программы в среде Delphi и убедимся в том, что она нам по силам. После этого подготовим "снаряжение" — установим на компьютер Delphi, и пройдем курс "молодого бойца" — напишем простую, но полезную и вполне работоспособную программу.

1.1. Краеугольные камни визуального программирования

В основе создания графических приложений лежат несколько очень простых понятий. Это те краеугольные камни, которые заложены архитектором в фундамент системы Delphi. Разобравшись с ними, вы быстро поймете суть визуального программирования.

Начиная работу со средой Delphi, вы должны:

1. Уяснить задачу, которую собираетесь решать на компьютере;
2. Нарисовать на бумаге все то, что предполагаете увидеть на экране в процессе решения. Это может быть один или несколько рисунков. Если задача сложная, ее следует разбить на этапы и для каждого этапа сделать отдельный рисунок;
3. Написать сценарий работы будущей программы. Местом развертывания действия является экран, а зритель не просто смотрит, но и участвует в “спектакле”. В сценарии должно быть учтено все: что выводится на экран вначале, что делается потом, как программа завершается, т.д. Декорациями “спектакля” служат сделанные в пункте 2 рисунки.

Не теряя драгоценного времени, реализуем все эти пункты для какой-нибудь простой и полезной задачи. Например, думая о своем здоровье, давайте создадим программу вычисления оптимального веса человека. Алгоритм решения выберем самый простой:

Оптимальный вес (кг) = Рост (см) – 100 – 10 (не слишком жестоко?)

Пункт 1 выполнен, задача абсолютно понятна, алгоритм решения имеется.

Теперь выполним пункт 2 — нарисуем то, что мы хотим видеть на экране в процессе решения задачи (рисунок 1.1): два редактируемых поля — для ввода роста (Specify your height) и вывода веса (Your ideal weight); две кнопки — для запуска вычислений (Compute) и выхода из программы (Close); текстовые надписи.

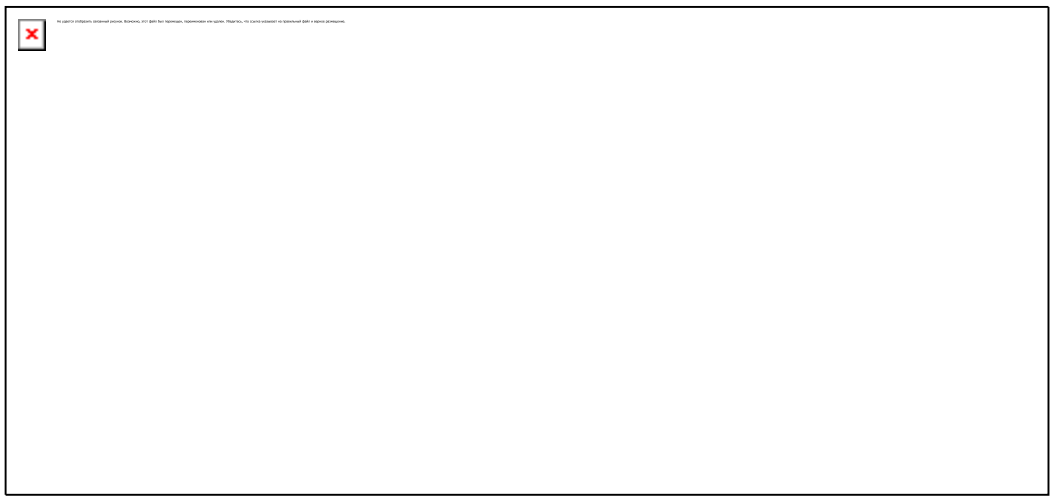


Рисунок 1.1. Форма и компоненты

Пора дать название тому, что мы тут нарисовали. Рисунок в целом называется *формой*, а поля ввода, вывода, кнопки и все прочее, что располагается на форме — *компонентами*. Нетрудно заметить, что компоненты на нашем рисунке — разные: редактируемые поля, кнопки, надписи. Они могут иметь разные размеры, их текст может отличаться высотой, шрифтом, цветом. Короче говоря, каждый компонент характеризуется рядом признаков, которые называются *свойствами*.

Для решения задачи может потребоваться несколько форм. Та форма, из которой вызываются все остальные, называется *главной*. Все другие формы — *второстепенные*. Главная форма в задаче присутствует всегда, второстепенных форм может быть несколько или не быть вообще. В нашем случае достаточно одной формы.

Вооруженные теорией и рисунками, выполним пункт 3 — напишем сценарий работы нашей будущей программы. Сразу после старта программы на экране появляется форма. Пользователь начинает вычисления: активизирует редактируемое поле с надписью Specify your height и вводит значение роста, затем нажимает кнопку Compute. Программа реагирует на это *событие*: вычисляет идеальный вес и выводит результат в поле с надписью Your ideal

weight. Когда пользователь определит идеальный вес всех своих знакомых (и сообщит им об этом по телефону), он нажмет кнопку Close. В ответ на это событие программа уберет с экрана свою форму и закончит работу.

Кстати, так ярко описанный нами процесс *событие-отклик-событие-отклик* называется *событийным управлением*, он лежит в основе работы всех современных графических программ. Простейшая аналогия для тех, кто не понял: любимый компьютер упал на любимый мозоль — событие, ваш тихий вопль в ночи — отклик.

Теперь решим крайне важный вопрос: что в этом сценарии будет делать среда Delphi, а что вы. Дело, в общем, обстоит так:

- Среда Delphi строит по вашим указаниям форму со всеми компонентами (редактируемыми полями, кнопками, надписями) и формирует исходный код соответствующей программы. По объему это львиная доля работы, но она выполняется за несколько минут.
- Программист дописывает на языке Delphi детали программы — процедуры обработки событий. Он делает это во встроенном в среду *редакторе кода*. Главное событие нашей программы — нажатие кнопки Compute. Обработка этого события — кодирование формулы Weight = Height – 100 – 10;
- Среда Delphi по команде программиста компилирует весь исходный код и запускает программу.

Такое распределение работы программистам нравится, всегда бы так! Кстати, оно отражено уже в структуре самой программы, которая состоит из нескольких частей. Важнейшие из них: файл исходного кода на языке Delphi (подготовленная средой основа + ваши детали) и файл формы (его тоже создает среда). Эти файлы и ряд других файлов, которые также нужны для решения задачи, составляют *проект*. Для каждой отдельной задачи создается свой проект. Сейчас важно просто о нем знать, все подробности вы узнаете в главе 7.

Только что изученные вами понятия являются ключом к пониманию используемой в среде Delphi технологии визуального программирования. Имея столь основательную теоретическую подготовку, можно спокойно готовить «походный инвентарь» — устанавливать на свой компьютер систему Delphi — и приступать к тренировочному «походу» — писать первую программу.

1.2. Подготовка к работе

Система программирования Delphi продается в нескольких редакциях:

- Delphi Personal — минимальный набор для любителей, изучающих программирование;
- Delphi Professional — набор для профессиональных разработчиков-индивидуалов;
- Delphi Enterprise — полный набор инструментальных средств для фирм, занимающихся изготовлением программного обеспечения на заказ.
- Delphi Architect — самый «упакованный» вариант, добавляющий к набору Delphi Enterprise средства коллективной разработки и средства моделирования данных.

Все редакции имеют одинаковую основу — интегрированную среду Delphi, но отличаются друг от друга дополнительными инструментальными средствами и лицензиями на их распространение вместе с вашими программами. При написании книги авторы ориентировались на обладателей редакции Delphi Enterprise. Тем не менее, книгой в равной мере смогут воспользоваться обладатели менее мощных редакций.

Будем считать, что вы успешно разобрались в своих задачах и купили именно ту редакцию, которая требуется. Теперь самое время открыть коробку. Вы обнаружите в ней компакт-диски со всем необходимым программным обеспечением и многотомное руководство пользователя. Отложим книги в сторону и приступим к приятному ритуалу: переносу системы Delphi с компакт-диска на жесткий диск вашего компьютера.

Вставьте первый компакт-диск в устройство CD-ROM — заставка установочной программы запустится автоматически. Если этого не произошло, запустите Windows Explorer, откройте компакт-диск и запустите программу Install. Следуйте указаниям, которые установочная программа будет время от времени выводить для вас на экран. (Новичкам советуем внимательно прочитать файл Readme.txt, который содержит подробные инструкции по установке и находится на компакт-диске в корневом каталоге.) Установочная программа сама пообщается с операционной системой, организует в стартовом меню **Start | Programs** папку **Borland Delphi 7** и поместит в нее ярлыки соответствующих программ (рисунок 1.2):

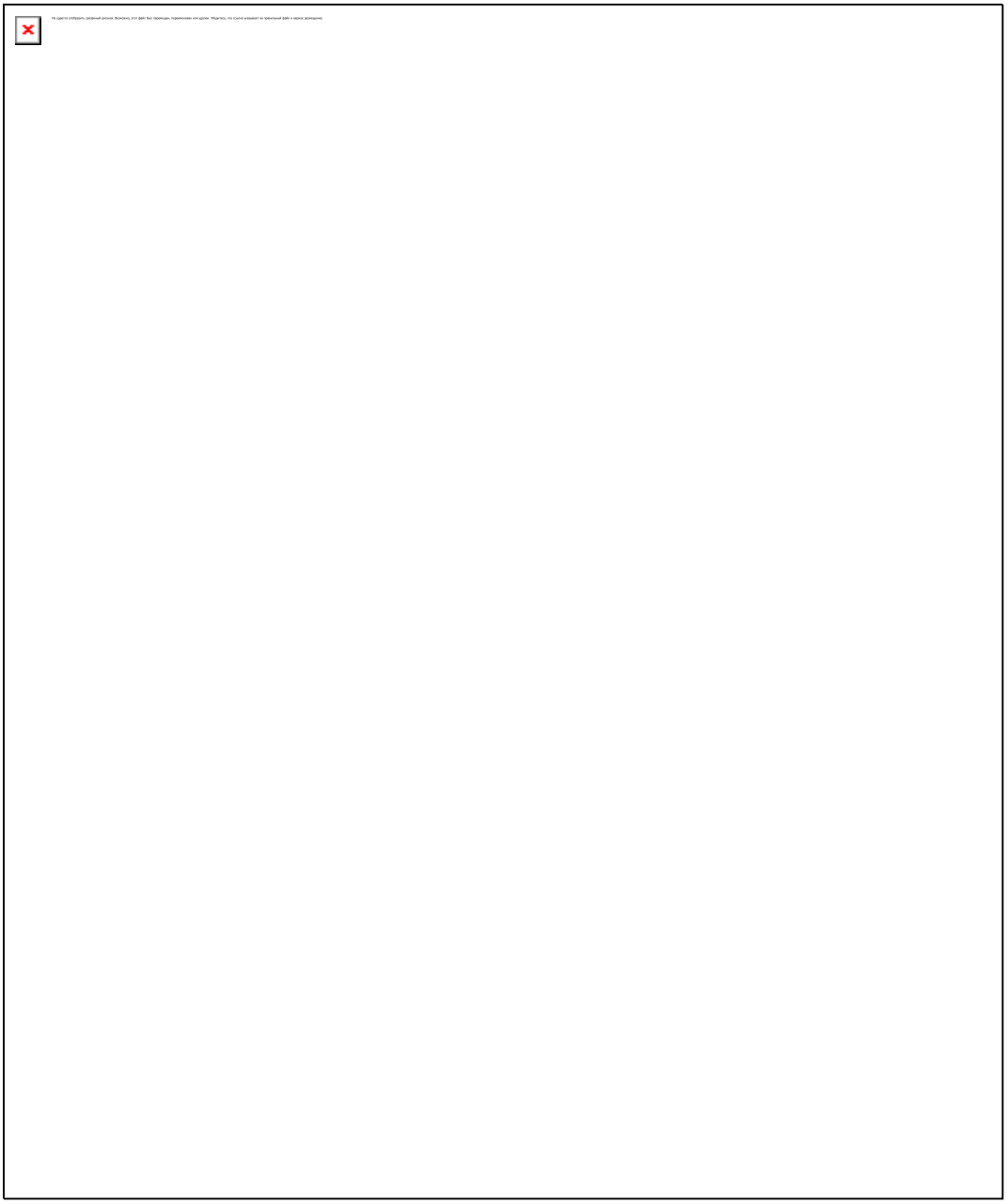


*Рисунок 1.2. Папка **Borland Delphi 7** в стартовом меню*

Давайте беглым взглядом посмотрим, что же мы установили:

- Delphi 7 — интегрированная среда разработки приложений;

- Image Editor — средство создания и редактирования точечных рисунков, значков, указателей мыши;
- BDE Administrator — программа-администратор ядра баз данных Borland Database Engine;
- Database Desktop — средство создания и редактирования таблиц в базах данных;
- SQL Explorer — интегрированное в среду Delphi средство для просмотра и редактирования таблиц в базах данных;
- SQL Monitor — отладочное средство, которое позволяет программисту отслеживать SQL-запросы к базам данных;
- WinSight32 — отладочное средство, которое позволяет программисту отслеживать сообщения Windows;
- XML Mapper — программа подготовки схем преобразования обычных XML-документов в пакеты данных, с помощью которых происходит обмен информацией с базами данных и другими приложениями.
- Register Now — программа, с помощью которой вы можете зарегистрировать свою копию системы Delphi у фирмы-разработчика.
- Справочники по различным вопросам; их список впечатляет (рисунок 1.3).

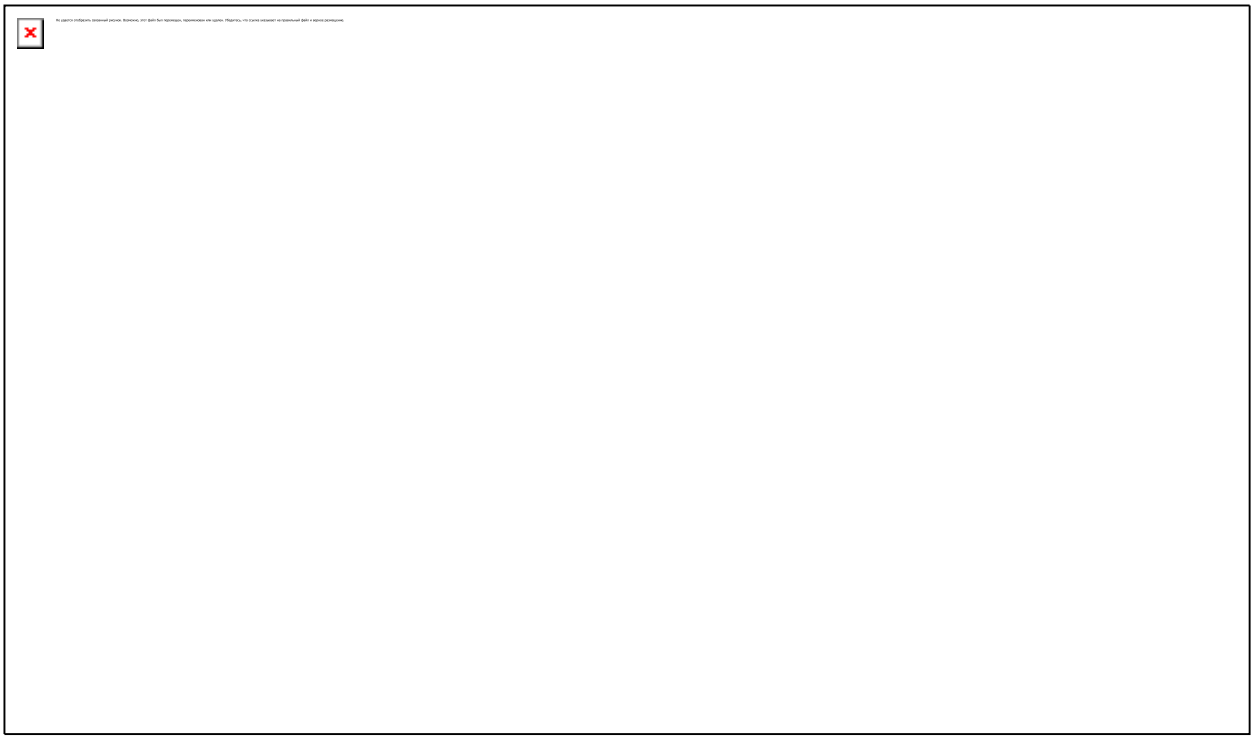


Рисунок 1.3. Справочники среды Delphi

Вот теперь мы готовы начать работу.

1.3. Первый запуск среды Delphi

Запустите среду разработки, выбрав соответствующий ярлык из главного меню операционной системы. Мир Delphi — перед вами (рисунок 1.4):

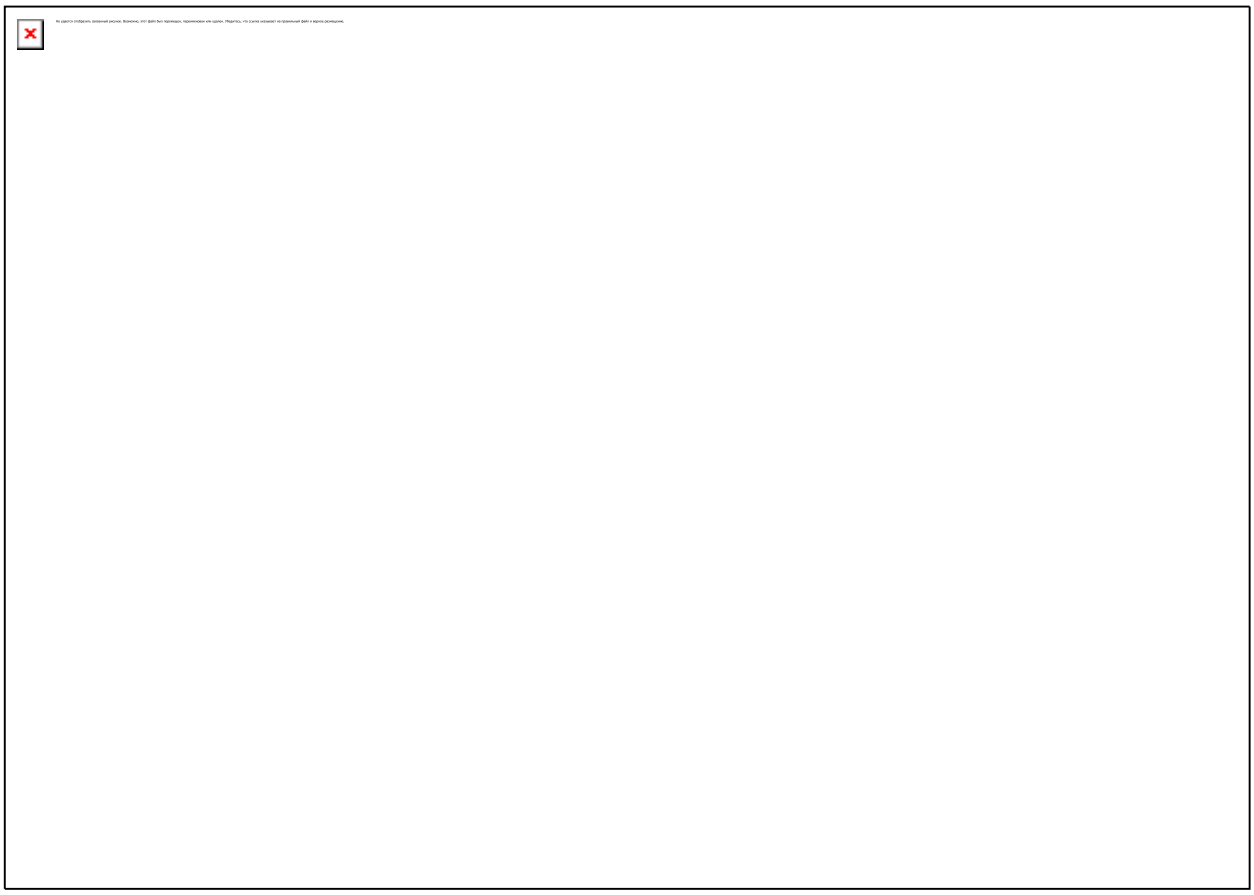


Рисунок 1.4. Вид среды Delphi при первом запуске

Что же вы видите? Окна, кнопки, списки... В общем — хаос. Давайте попробуем навести порядок, и посмотрим на среду Delphi издалека. Ба! Да это же навсегда запечатленные в памяти читателя краеугольные камни: *форма*, *компоненты* и *свойства* (рисунок 1.5).

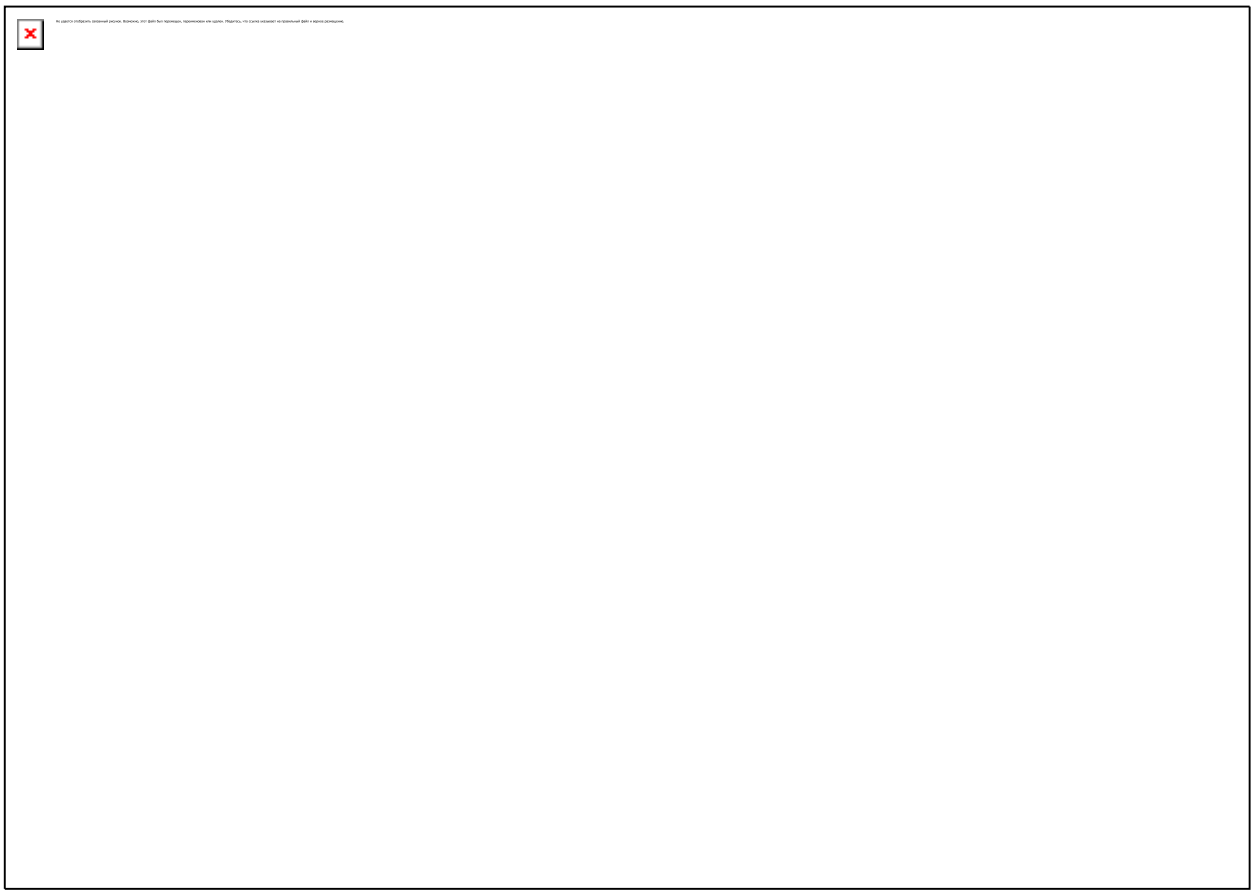


Рисунок 1.5. Главные части среды Delphi

Ура! Неизвестное оказалось хорошо известным! Хаос исчез, и сообразительный читатель уже все понял: из области “палитра компонентов” надо с помощью мыши выбрать компонент (кнопку, надпись, редактор текста и т.д.), поместить на "форму" и задать значения его свойств в области “свойства“. Среда Delphi проанализирует содержимое формы, создаст соответствующий программный код, а программисту останется только внести в него детали решения задачи — отклики на события. В общем, назначение интегрированной среды понятно, теперь можно спуститься с небес и заняться деталями.

1.4. Исследуем среду разработки программ

Снова посмотрим на интегрированную среду разработки (рисунок 1.6) и дадим название каждой ее части:

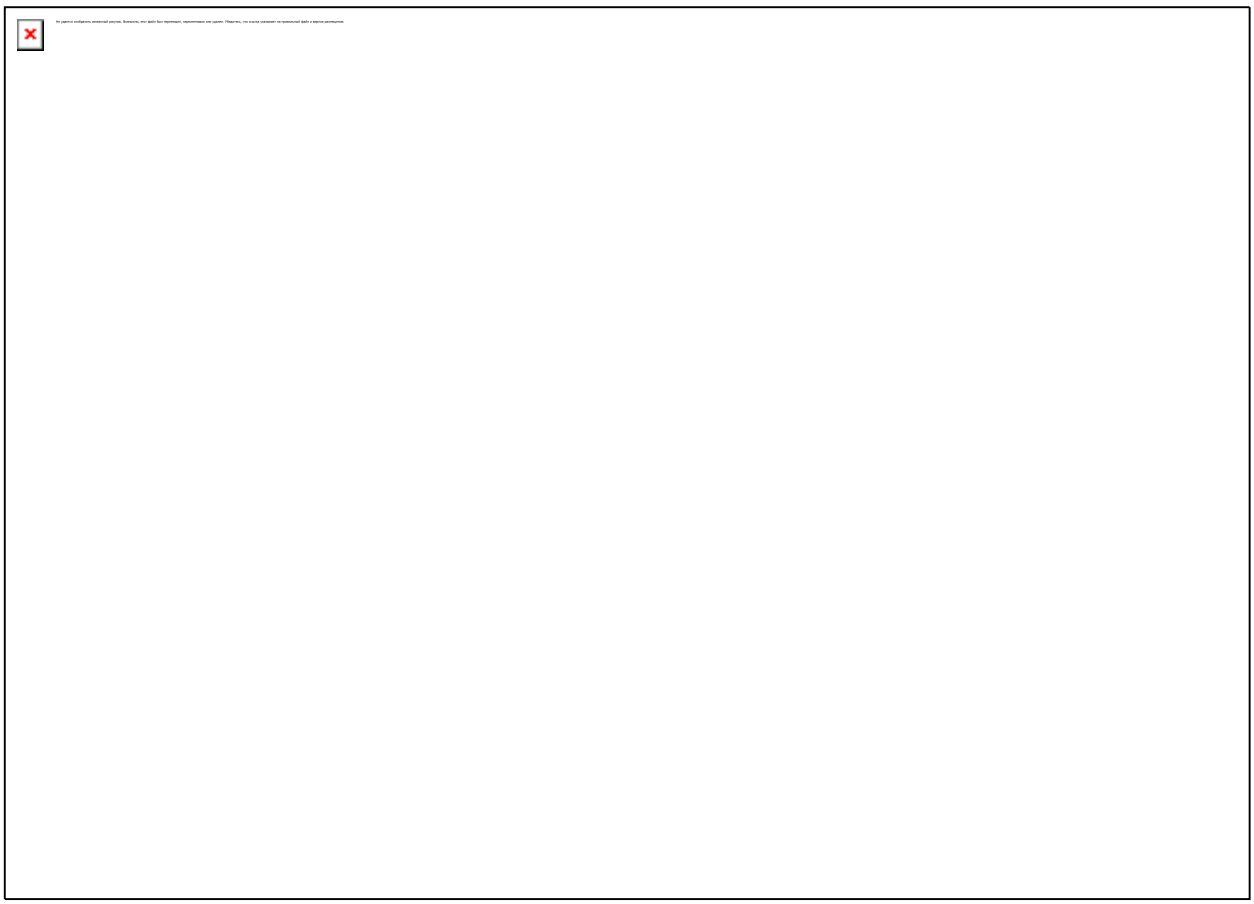


Рисунок 1.6. Среда Delphi в деталях

Обсудим кратко каждую из составных частей. Важнейшая часть — *форма* (рисунок 1.7). Она имеет заголовок Form1 и пока пуста (это аналог чистого листа, на котором вы собираетесь что-то рисовать):

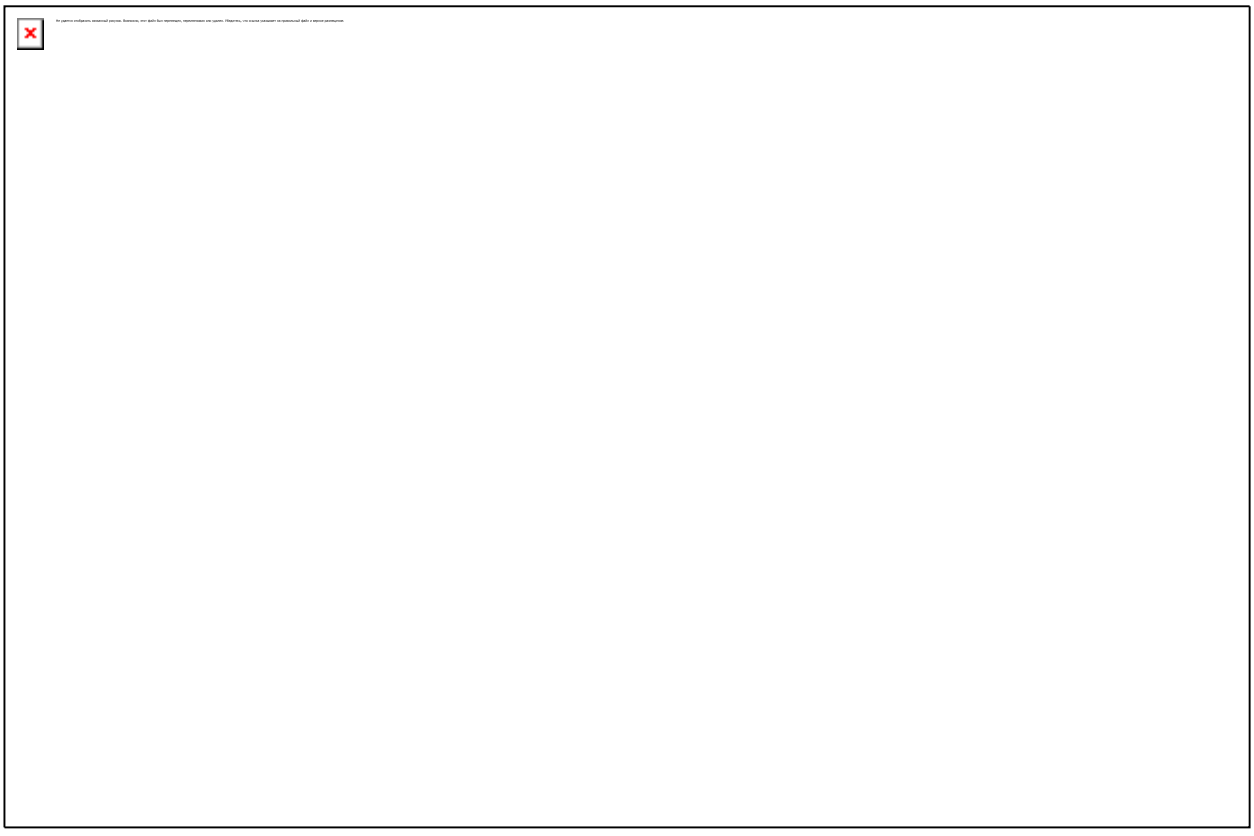


Рисунок 1.7. Форма

Обратите внимание, что форма имеет все признаки “главного окна” традиционных графических приложений: значок, заголовок, кнопки "Свернуть", "Развернуть", "Закрыть", размерную рамку и, конечно, управляется мышью. Умудренный опытом читатель-программист подумает: сколько надо попотеть, чтобы сделать все это самому, а здесь основа интерфейса практически готова...

Под формой спрятан *редактор кода* (рисунок 1.8):

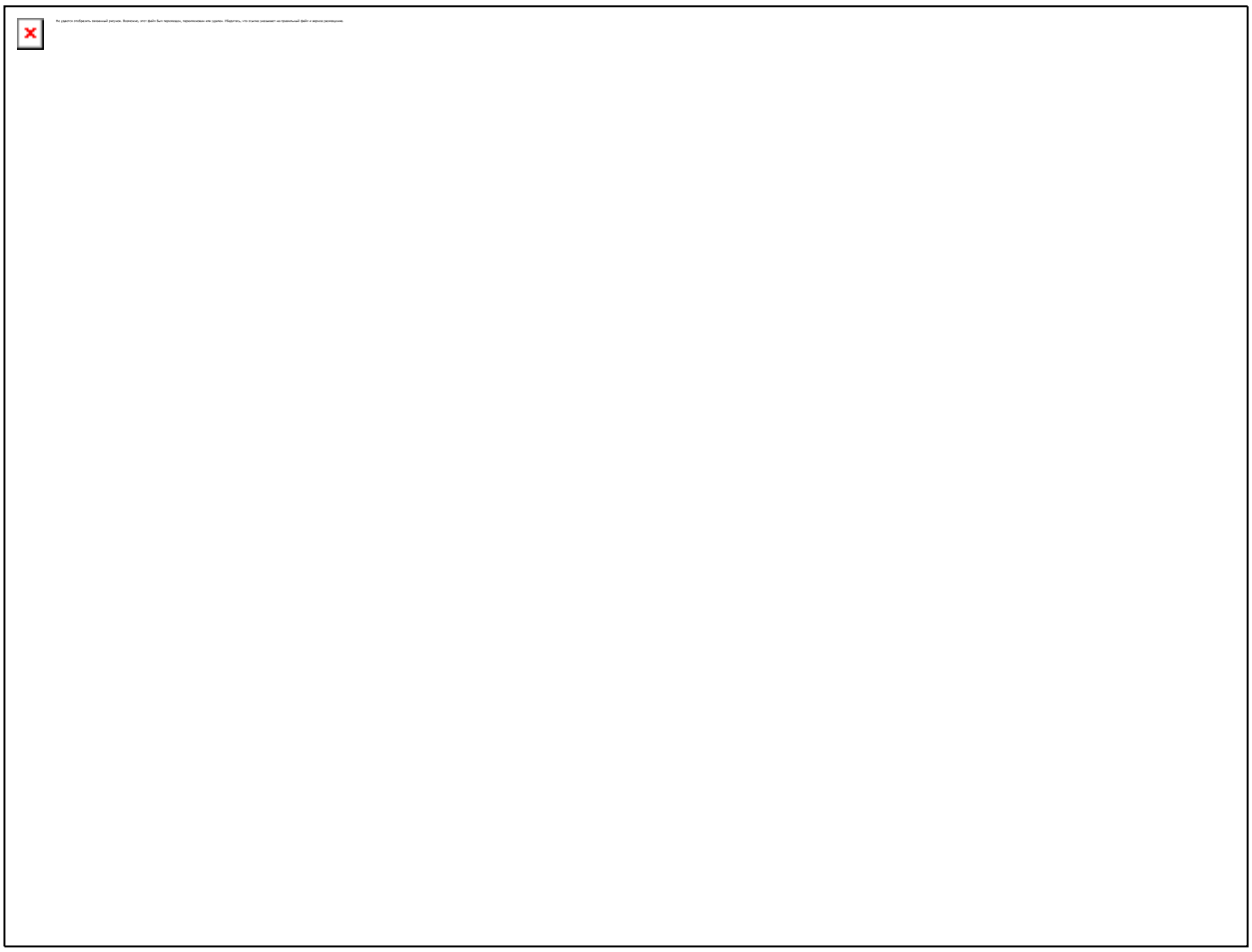


Рисунок 1.8. Редактор кода

Именно там размещается программный код на языке Delphi, соответствующий содержанию формы. Активизация редактора кода осуществляется щелчком мыши на части окна, которая выглядывает из-под формы, либо с помощью клавиши F12 на клавиатуре. В редакторе кода вы выполняете свою часть работы — дописываете детали решения задачи. Для возврата к форме достаточно нажать на клавиатуре клавишу F12 еще раз.

С формой все ясно, займемся компонентами, которые будем на ней размещать. Они находятся в области главного окна интегрированной среды, которая называется *палитрой компонентов* (рисунок 1.9).

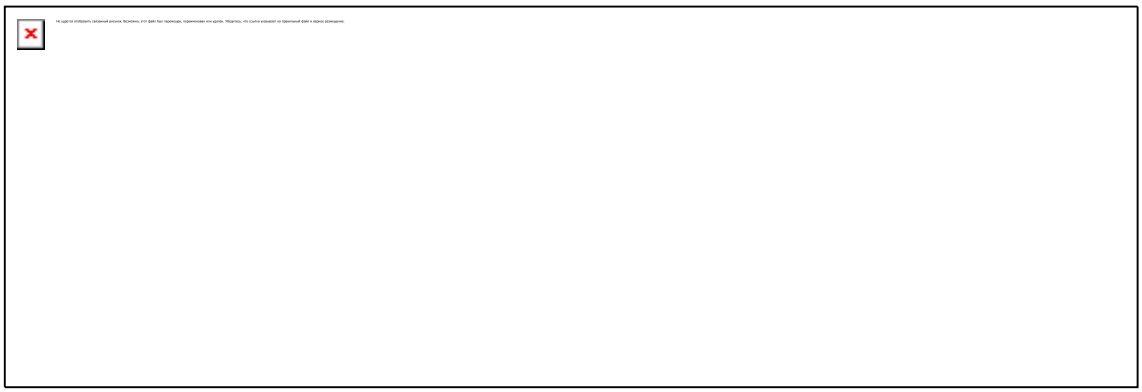


Рисунок 1.9. Палитра компонентов

Разработчики среды Delphi поместили в палитру компонентов то, что считают оптимальным набором “строительных кирпичиков“, достаточным для создания любых приложений. Среди компонентов вы найдете меню, кнопки, надписи, стандартные диалоговые окна и др.

Как видно на рисунке 1.9, все множество компонентов разделено на группы. Каждая группа размещена в палитре компонентов на своей вкладке: Standard — стандартные компоненты пользовательского интерфейса, Additional — дополнительные компоненты пользовательского интерфейса, Common Controls — общепринятые для Windows компоненты пользовательского интерфейса и т.д. Описание каждой вкладки палитры компонентов приведено в приложении А.

Выбрать нужный компонент из палитры и поместить его на форму очень просто:

- Перейдите к нужной вкладке в палитре компонентов;
- Выберите нужный компонент;
- Отметьте на форме то место, где будет находиться компонент, — он мгновенно окажется на форме;
- Придайте компоненту нужные размеры, растягивая по высоте и ширине, и скорректируйте его местоположение, используя имеющуюся на форме сетку (рисунок 1.10).

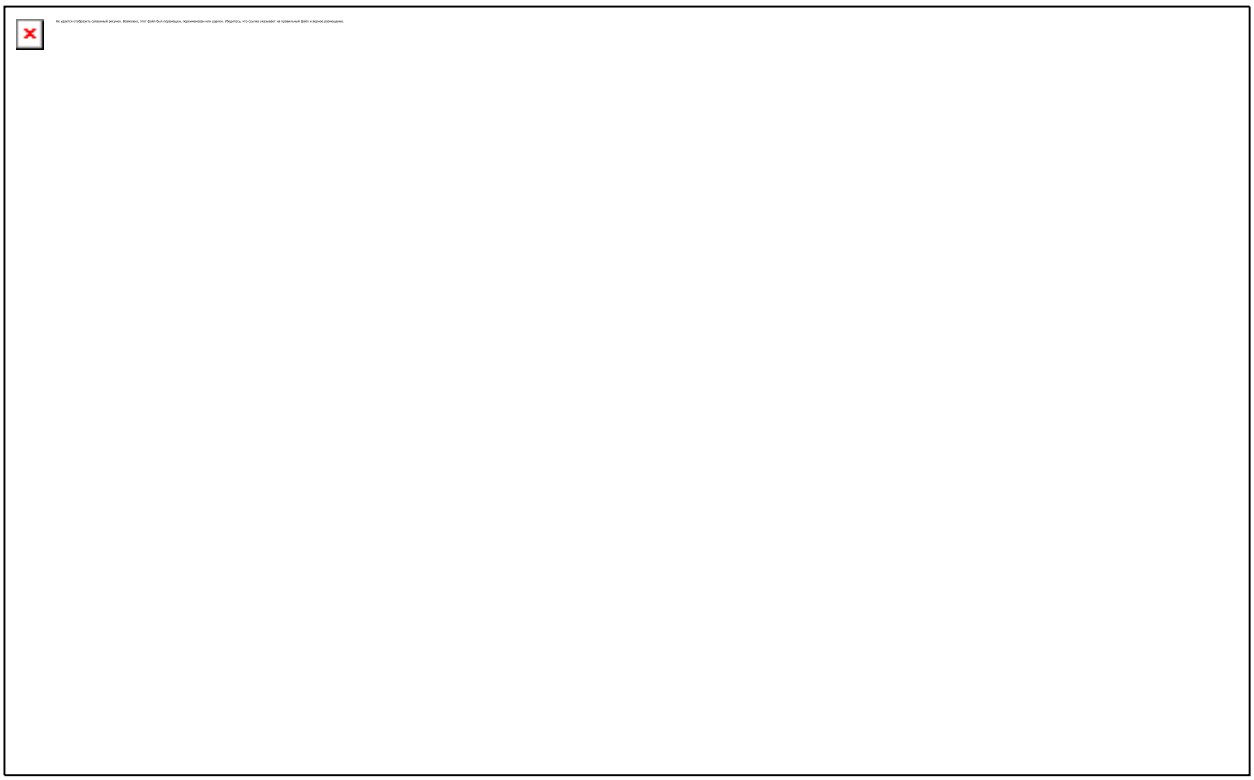


*Рисунок 1.10. Компонент **Button** на форме*

Компонент — на форме, пора задать его свойства. Для этого обратимся к окну с заголовком **Object Inspector** — *окну свойств* (рисунок 1.11). Оно расположено слева снизу от формы и активизируется с клавиатуры нажатием клавиши F11.

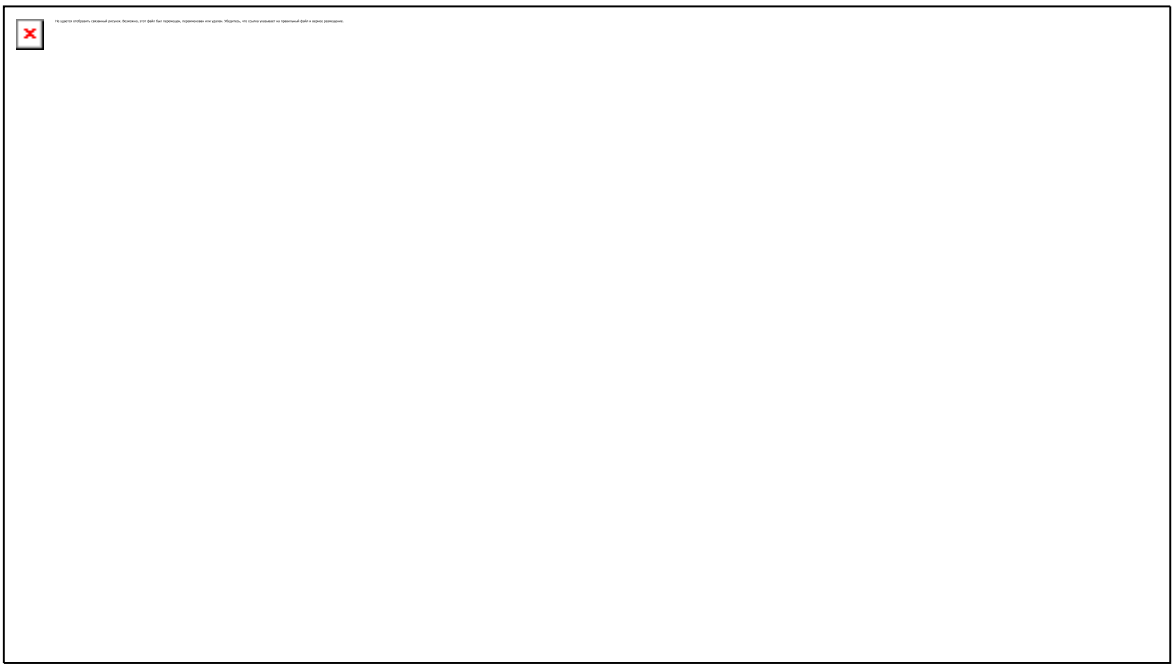


Рисунок 1.11. Окно свойств

Как только компонент оказывается на форме, в окне **Object Inspector** отображается список его свойств. Ваша задача — присвоить свойствам нужные значения. Например, чтобы написать на кнопке слово **Compute**, достаточно изменить значение свойства **Caption**, которое изначально содержит текст **Button1**. По мере набора строки каждая буква будет автоматически появляться на кнопке.

Нетрудно заметить, что окно **Object Inspector** состоит из двух вкладок: *вкладки свойств* — **Properties** и *вкладки событий* — **Events**. На вкладке **Properties** устанавливаются свойства компонента. Когда значения свойств определены, нужно активизировать вкладку **Events**. Вы тут же увидите список событий, на которые данный компонент может реагировать. В качестве примера приведем список событий, на которые может реагировать кнопка (рисунок 1.12):

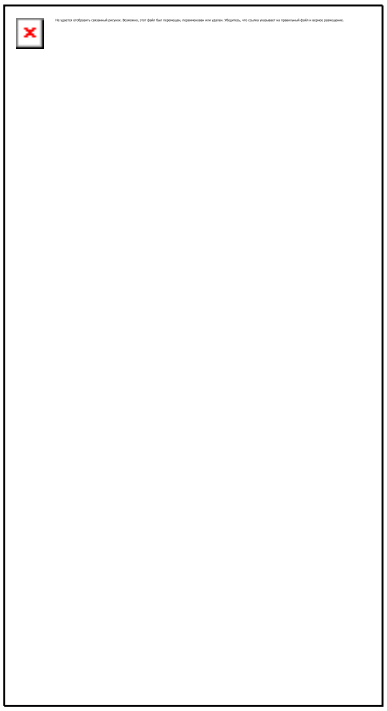


*Рисунок 1.12. Список событий компонента **Button***

В представленном списке нас может интересовать событие **OnClick** — нажатие кнопки.

Некоторые компоненты подобно форме умеют содержать другие компоненты. Например, панель (компонент **Panel**) может содержать кнопки, надписи, другие панели и т.д. По внешнему виду формы не всегда можно определить, какие компоненты содержит интересующий вас компонент или на каком компоненте он содержится. Для ответа на эти вопросы, обратитесь к окну **Object TreeView** — *окну компонентов формы* (рисунок 1.13). Оно расположено слева вверху от формы и активизируется нажатием комбинации клавиш Shift+Alt+F11. В окне **Object TreeView** компоненты представлены в виде дерева, отражающего вложенность компонентов на форме. Сейчас на форме находится лишь одна единственная кнопка, поэтому дерево компонентов имеет очень простой вид: корневой элемент **Form1** и один вложенный элемент **Button1**.

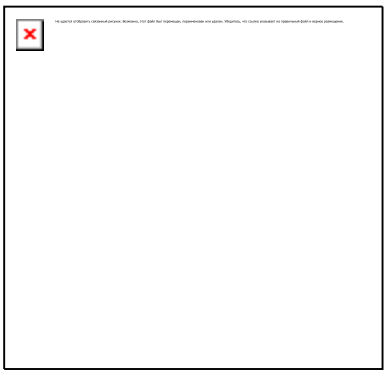


Рисунок 1.13. Окно компонентов формы

Надеемся, что важнейшие элементы интегрированной среды — форма, редактор кода, палитра компонентов, окно компонентов формы, окно свойств — навсегда запечатлелись в вашей памяти, и переходим к другим ее частям.

Для управления процессом создания приложения в целом служит *главное меню*. Оно расположено в главном окне среды Delphi и выполняет множество служебных функций. Меню, в общем-то, стандартно и понятно каждому, кто имел дело с компьютером. Поэтому мы предельно кратко опишем назначение важнейших разделов главного меню:

File — работа с файлами.

Edit — работа с областью обмена, размещение компонентов на форме.

Search — поиск, замена заданного символа или строки в тексте.

View — отображение различной информации.

Project — управление проектом: добавление и удаление файлов, сборка проекта, установка параметров проекта.

Run — запуск и отладка программы.

Component — разработка новых компонентов, установка готовых компонентов.

Database — запуск программ, облегчающих построение приложений баз данных.

Tools — настройка параметров интегрированной среды разработки, запуск вспомогательных программ.

Window — активизация нужного окна интегрированной среды разработки.

Help — получение справочной информации.

Для ускорения доступа к некоторым командам служит *панель кнопок* (рисунок 1.14).

Рисунок 1.14. Панель кнопок среды Delphi

На ней вы обнаружите шестнадцать кнопок-аналогов основных команд меню. Этот список можно расширить, добавив кнопки доступа к своим любимым командам. Для этого достаточно навести указатель мыши на панель кнопок, вызвать вспомогательное меню щелчком правой кнопки мыши и выбрать команду Customize.

Справа от главного меню есть небольшая панель (рисунок 1.15) для сохранения и восстановления внешнего вида среды Delphi. Расположите окна на экране на свой вкус, подберите наиболее удобные для себя кнопки. Затем, нажав кнопку с подсказкой **Save current desktop**, сохраните внешний вид среды Delphi. В следующем сеансе работы вы сможете мгновенно восстановить его из списка.

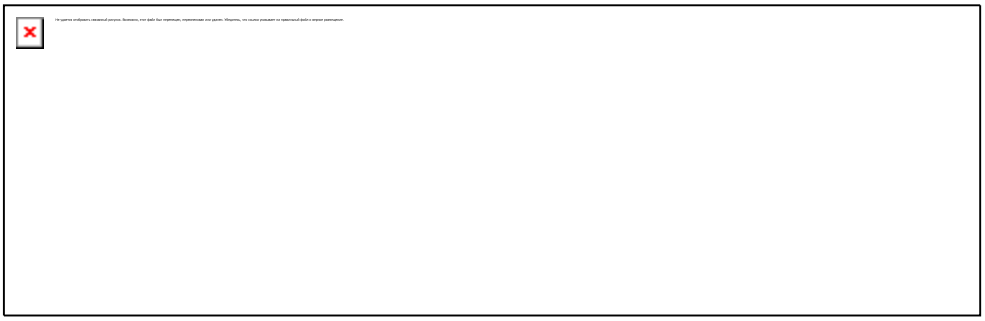


Рисунок 1.15. Панель для сохранения и восстановления внешнего вида среды Delphi

Вот вы и изучили основные элементы среды визуального программирования! Вы познакомились со средой Delphi только в самых общих чертах, но этого достаточно, чтобы попробовать написать первую программу.

1.5. Первая программа

Пора обрадовать ваших родственников и знакомых приятным сообщением о том, сколько они должны весить для поддержания хорошего здоровья. Для этого напишем программу вычисления оптимального веса по известному вам сценарию. Как вы помните, форма для задачи содержит две кнопки (для запуска вычислений и завершения работы) и два поля (для ввода значения роста в сантиметрах и вывода веса в килограммах).

Вы, конечно, уже запустили среду Delphi, и видите перед собой пустую форму. Начнем с кнопок. Наведите указатель мыши на палитру компонентов и щелкните на значке с подсказкой **Button** (рисунок 1.16).

*Рисунок 1.16. Значок компонента **Button***

Затем наведите указатель мыши на форму и щелкните еще раз. Заготовка кнопки с надписью **Button1** окажется на форме (рисунок 1.17):

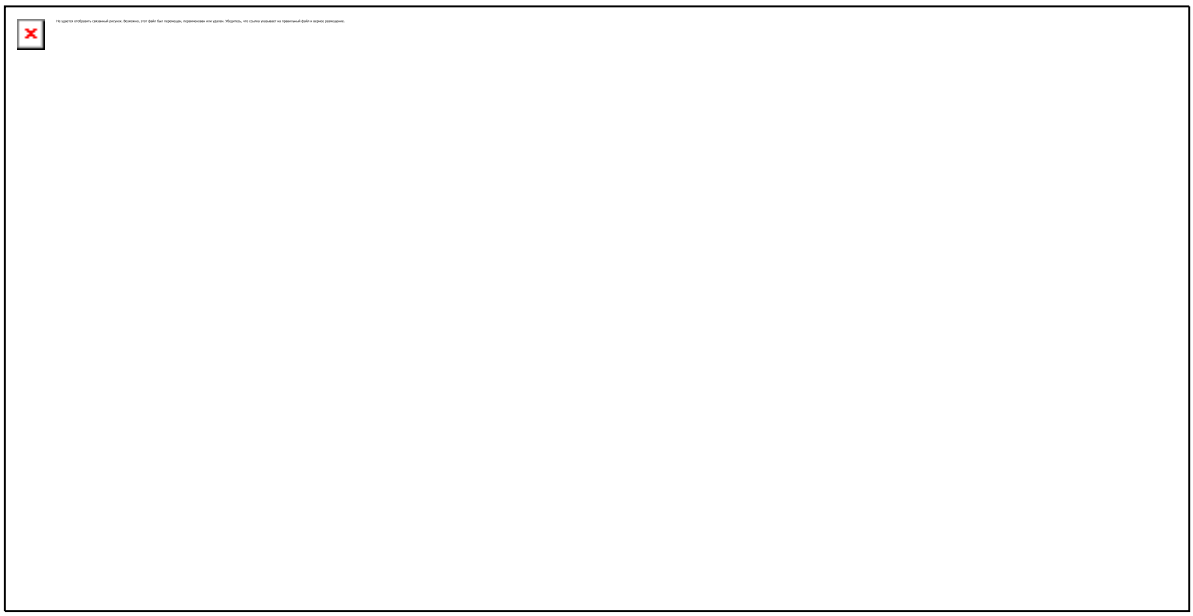


Рисунок 1.17. Заготовка кнопки на форме

Ваше дело — сделать из заготовки "конфетку"! С помощью мыши подправьте ее местоположение. После этого перейдите к окну **Object Inspector**. Там вы увидите список свойств компонента **Button**. В нем нас интересует свойство **Caption**, так как именно оно определяет содержимое надписи. Заменим в свойстве **Caption** стандартное значение **Button1** на **Compute** (рисунок 1.18).

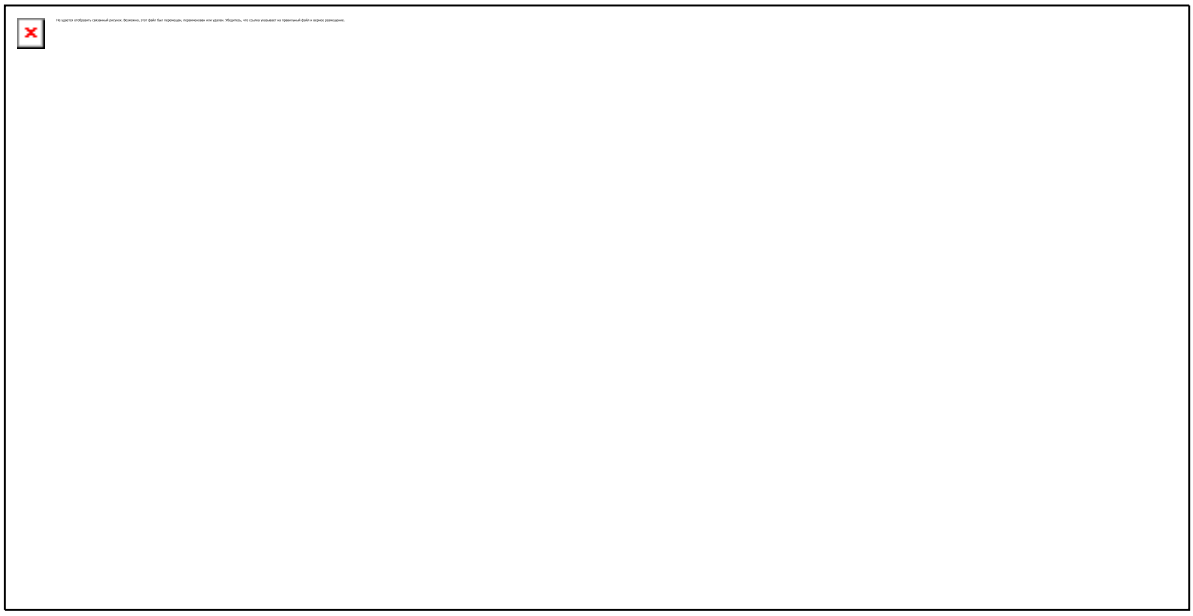


*Рисунок 1.18. Кнопка **Compute***

Точно так же на форму помещается вторая кнопка с надписью **Close** (рисунок 1.19):

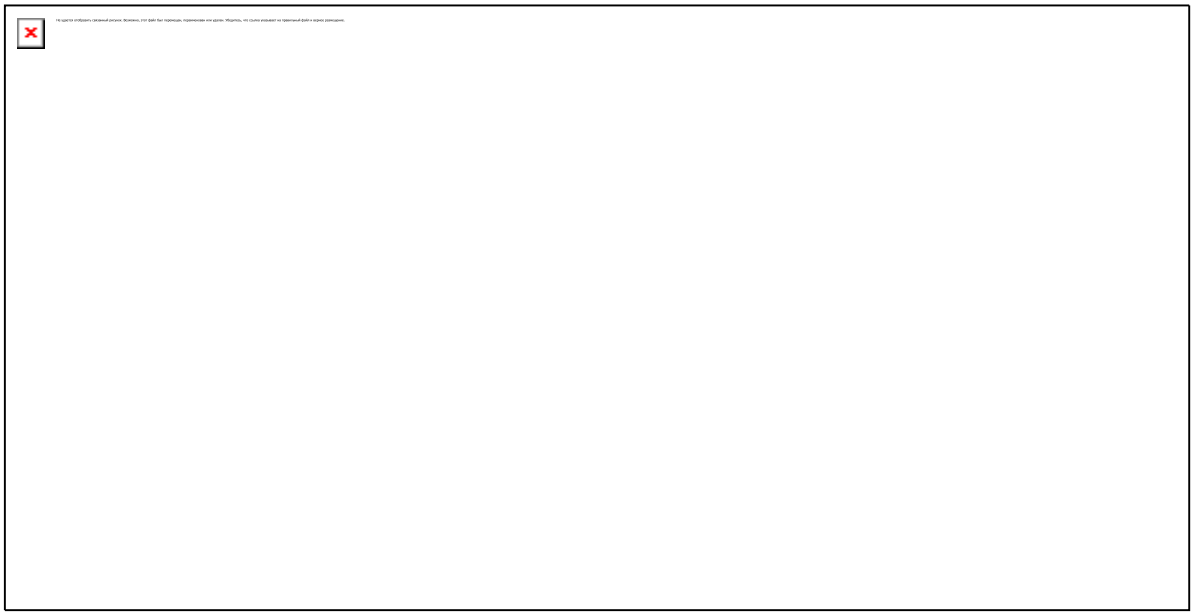


*Рисунок 1.19. Кнопка **Close***

Теперь возьмемся за поля ввода и вывода. Для этого покинем на время окно **Object Inspector** и перейдем к палитре компонентов. Там вы найдете компонент **Edit** (рисунок 1.20), который лучше всего отвечает нашим целям.

*Рисунок 1.20. Значок компонента **Edit***

Щелкните на нем, затем наведите указатель мыши в нужное место формы и щелкните еще раз. На форме появится поле ввода. Изначально оно содержит текст **Edit1** (рисунок 1.21):

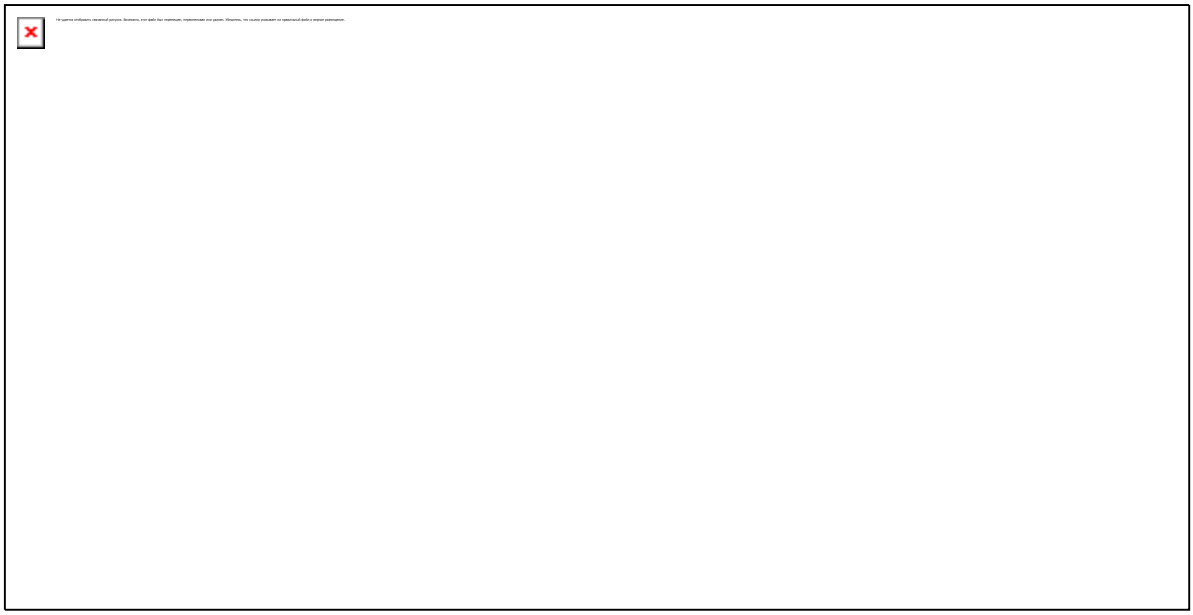


Рисунок 1.21. Поле ввода исходных данных

Придайте компоненту нужные размеры и откорректируйте местоположение. Готово? Теперь снова перейдите к окну **Object Inspector**. Нас интересует свойство **Text**. Удалите в нем ненужное значение **Edit1** и содержимое поля на форме сразу же очистится.

Точно так же приготовьте поле для вывода результата вычислений. Вот вы и получили форму, полностью отвечающую сценарию работы программы (рисунок 1.22):

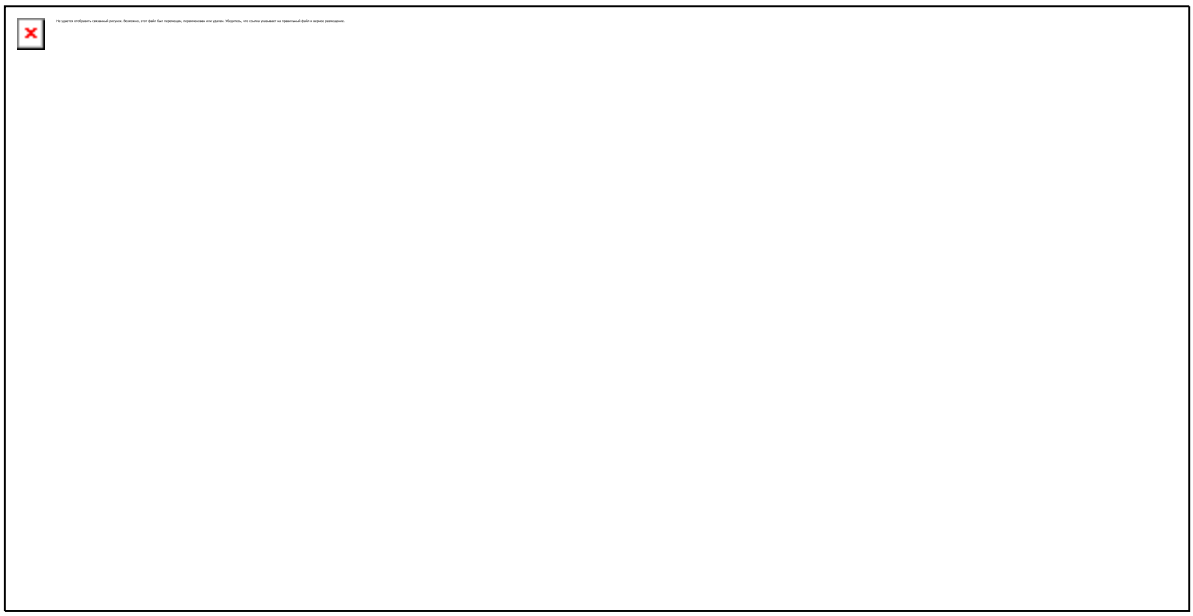


Рисунок 1.22. Поле вывода результата вычислений

Ба! Да мы забыли дать название нашей прекрасной форме и надписать поля для ввода и вывода! Исправим этот досадный пробел: присвоим свойству **Caption** для формы в целом значение **Weight Calculator** и оно появится в заголовке формы. Теперь давайте озаглавим редактируемые поля. Над окном ввода данных напишем **Specify your height**, а над окном вывода результата **Your ideal weight**. Вы помните, что строка текста на форме — такой же

компонент, как, например, кнопка. Поэтому обратитесь к палитре компонентов, выберите компонент **Label** (рисунок 1.23)

*Рисунок 1.23. Значок компонента **Label***

и поместите его над полем ввода. Отрегулируйте местоположение компонента с помощью мыши и в значении свойства **Caption** наберите текст первой надписи. Затем повторите те же манипуляции для поля вывода. В результате получилась очень неплохая основа для будущей программы (рисунок 1.24):

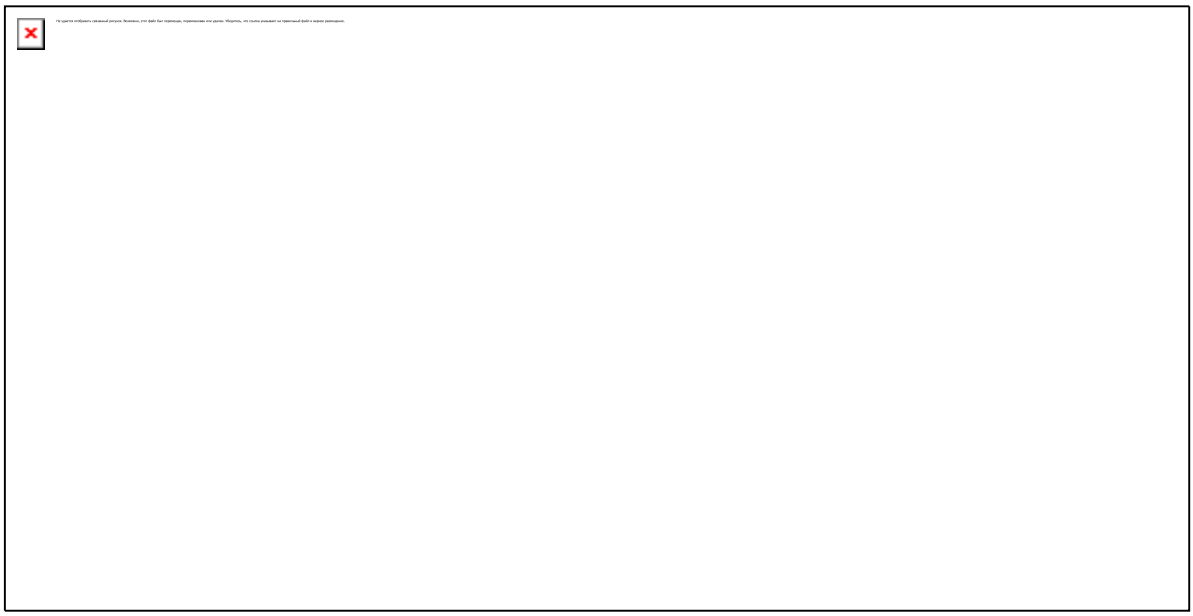


Рисунок 1.24. Форма для задачи вычисления оптимального веса человека

Все это хорошо, но работает ли на практике? Давайте проверим: на панели кнопок нажмите кнопку с подсказкой **Run** (Запуск) или выберите в меню команду **Run | Run**. Этим вы одновременно выполните компиляцию и запустите приложение. Понажимайте кнопки **Compute** и **Close**. Работают? Да. Перейдите в поле ввода и понажимайте цифровые клавиши. Нормальный ввод? Да. Однако вычислений никаких нет. Почему? Потому что события (нажатия кнопок) есть, а вот отклика на них нет — мы просто этим вопросом еще не занимались.

Закрыв приложение, приступим к обработке событий. Начнем с нажатия кнопки **Compute**. Активизируйте кнопку, с которой будем работать, затем перейдите к окну **Object Inspector** и переключитесь на вкладку **Events** (события). На ней вы обнаружите список всех возможных событий для активного компонента (в данном случае кнопки). Нас интересует событие **OnClick**, возникающее при нажатии кнопки. Чтобы запрограммировать обработчик этого события, сделайте двойной щелчок мыши в поле значения. Появится окно редактора кода с заготовкой для нашего обработчика (рисунок 1.25):

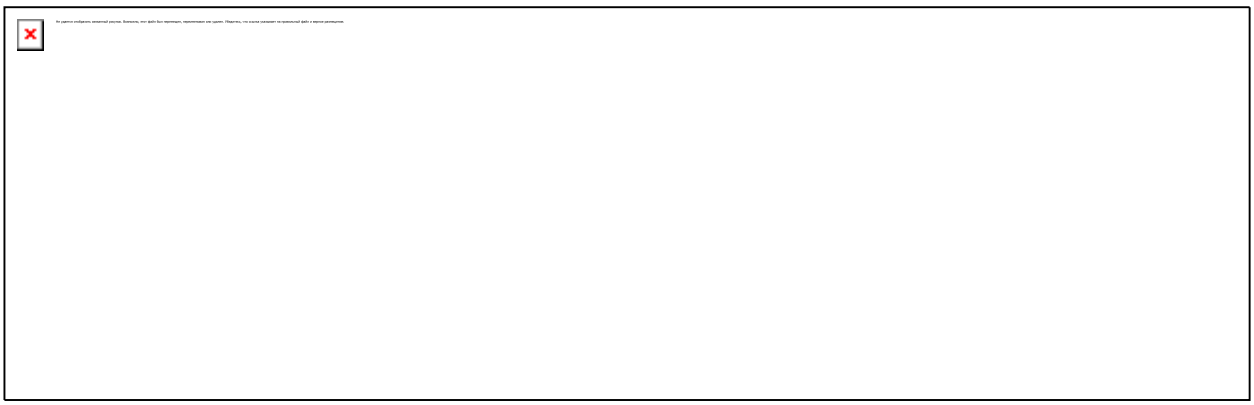


Рисунок 1.25. Заготовка в редакторе кода для обработки щелчка кнопки ***Compute***

Вычисление оптимального веса делается в одну строку:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit2.Text := IntToStr(StrToInt(Edit1.Text) - 100 - 10);
end;
```

Теперь займемся кнопкой **Close**. Обработчик события для нее устанавливается аналогично: кнопка активизируется на форме, в окне **Object Inspector** выбирается вкладка **Events** и на значении события **OnClick** делается двойной щелчок мыши. Текст этого обработчика еще проще, чем текст предыдущего:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

Процедура Close, вызываемая при нажатии кнопки **Close**, закрывает форму и завершает программу.

Теперь давайте сохраним результат нашей работы. Для этого выберите команду меню **File | Save All**. Сначала среда Delphi предложит ввести имя для модуля формы, а потом — имя для всего проекта. Модуль назовем Unit1.pas (рисунок 1.26),

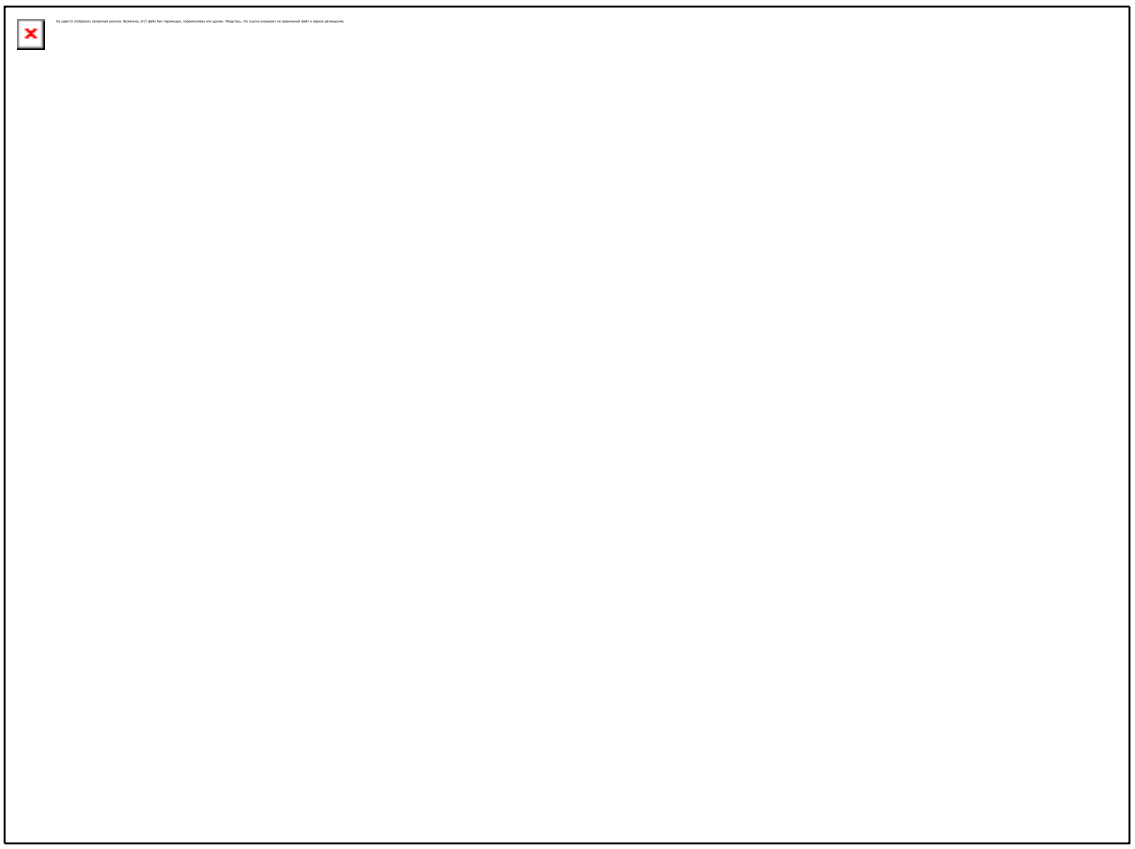


Рисунок 1.26. Окно, выдаваемое при сохранении нового модуля

а проект — Project1.dpr (рисунок 1.27):

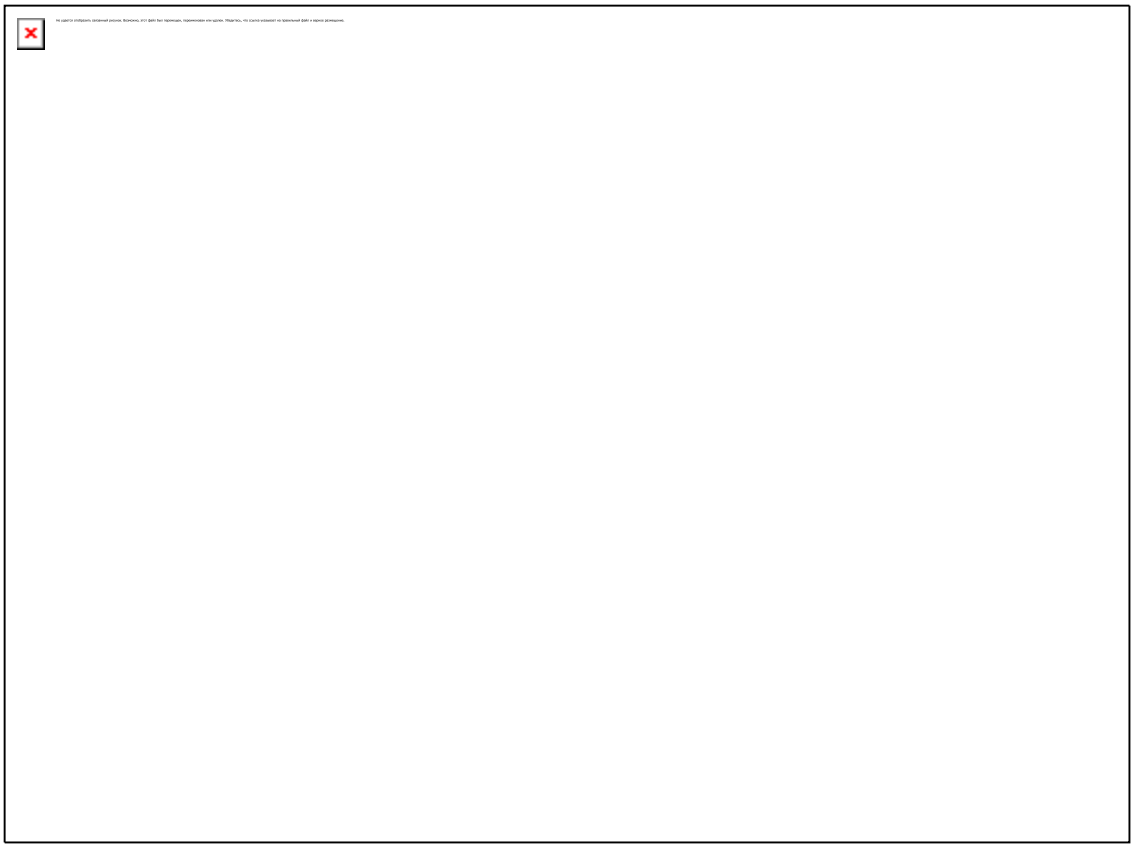


Рисунок 1.27. Окно, выдаваемое при сохранении нового проекта

Все файлы, которые относятся к решению задачи и составляют проект, будут записаны на диск в заданный каталог. Сохраненный проект может быть открыт для доработки в любой момент.

Теперь давайте поработаем с программой по-настоящему. Запустите ее (с помощью команды меню **Run | Run**), активизируйте поле с надписью **Specify your height** и введите значение роста своего любимого дядюшки Джо (170 см). Исходный материал есть, пора начать вычисления. Нажмите кнопку **Compute**. В выходном поле появится значение 60 (рисунок 1.28):

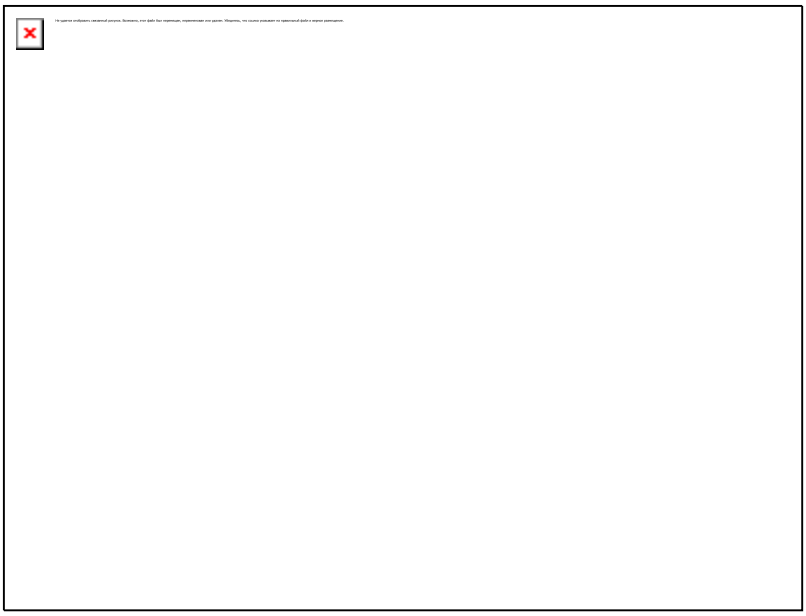


Рисунок 1.28. Работающая программа вычисления оптимального веса человека

Что-то здесь не то. Дядя Джо весит 90 кг... Может быть, наша программа неправильно считает? Нет, дело здесь не в программе, просто дядюшке нужно немного похудеть — килограмм на 30.

Очевидно, что программа работает. А как насчет отказоустойчивости? Давайте проверим: запустим программу и вместо значения роста введем какую-нибудь строку, например, JOHN. Что случилось? На экране появилось сообщение об ошибке (рисунок 1.29) и программа временно прекратила работу.

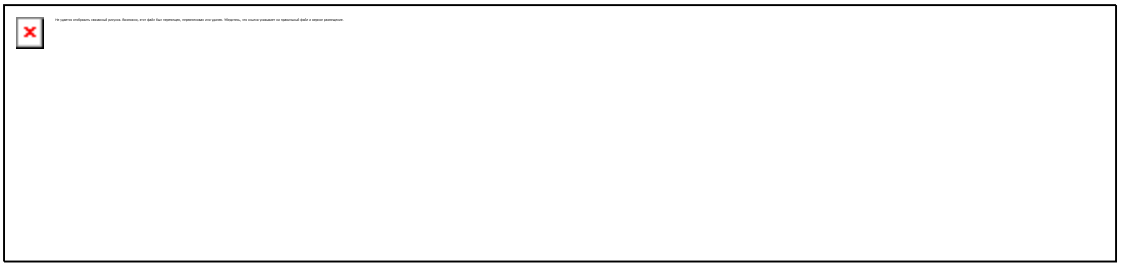


Рисунок 1.29. Сообщение об ошибке

Причина: при попытке обработать строку вместо числа, система попала в *исключительную ситуацию*. В нашем случае программа запускалась из среды Delphi, которая перехватывает исключительную ситуацию и показывает вам приведенное выше окно **Error**. Нажмите **OK** и вы увидите то место в программе, где возникла исключительная ситуация. Программа приостановлена, продолжить ее выполнение можно, нажав в среде Delphi кнопку **Run**. Приложение выдаст сообщение о том, что JOHN — это неверное числовое значение (рисунок 1.30):

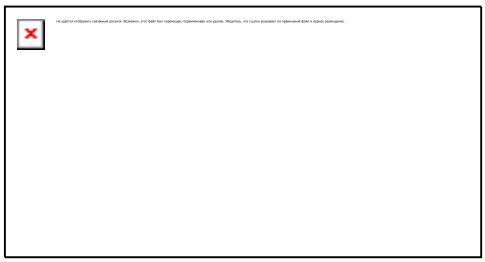


Рисунок 1.30. Сообщение о неверном числовом значении

и после нажатия **OK** восстановится. Это работают автоматически встроенные в приложение защитные механизмы Delphi. В данном случае программа справилась с ошибкой без посторонней помощи, но бывают исключительные ситуации, которые в среде Delphi не оговорены. Об обработке таких ситуаций должен позаботиться программист. Однако всему свое время — обрабатывать исключительные ситуации вы научитесь, прочитав главу 4. Сейчас достаточно знать, что они бывают и что их можно обрабатывать.

1.6. Итоги

Пора подвести первые итоги. Они очень даже не плохи: вы изучили все основные понятия визуального программирования; умеете подойти к решению любой задачи; представляете как свою роль, так и роль среды Delphi в процессе решения; познакомились с проектом приложения; создали и выполнили свою первую программу, и даже испытали ее на прочность. Эйфория от столь значительных успехов не должна затмевать вставшую перед вами первую серьезную и совершенно очевидную проблему: интерфейс — форму с компонентами — вы знаете, как сделать, а вот как самому написать обработчик событий кнопки или другого компонента, — вам пока не известно. А эта работа — ваша, и никуда от нее не деться. Чтобы ее выполнить, нужно знать не только среду программирования, но и язык Delphi. Ему мы посвятим следующие несколько глав.

Глава 2. Основы языка Delphi

В основе среды Delphi лежит одноименный язык программирования — Delphi, ранее известный как Object Pascal. При разработке программы среда Delphi выполняет свою часть работы — создает пользовательский интерфейс согласно вашему дизайну, а вы выполняете свою часть — пишите обработчики событий на языке Delphi. Объем вашей работы зависит от программы: чем сложнее алгоритм, тем тяжелее ваш труд. Необходимо заранее усвоить, что невозможно заставить средство разработки делать всю работу за вас. Некоторые задачи среда Delphi действительно полностью берет на себя, например создание простейшей программы для просмотра базы данных. Однако большинство задач не вписываются в стандартные схемы — вам могут понадобиться специализированные компоненты, которых нет в палитре компонентов, или для задачи может не оказаться готового решения, и вы вынуждены будете решать ее старым дедовским способом — с помощью операторов языка Delphi. Поэтому мы настоятельно рекомендуем вам не игнорировать эту главу, поскольку на практике вы не избежите программирования. Мы решили изложить язык в одной главе, не размазывая его по всей книге, чтобы дать вам фундаментальные знания и обеспечить быстрый доступ к нужной информации при использовании книги в качестве справочника.

2.1. Алфавит

2.1.1. Буквы

Изучая в школе родной язык, вы начинали с букв, слов и простейших правил синтаксиса. Для постижения основ языка Delphi мы предлагаем вам сделать то же самое.

Текст программы на языке Delphi формируется с помощью букв, цифр и специальных символов.

Буквы — это прописные и строчные символы латинского алфавита и символ подчеркивания:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _
```

Цифры представлены стандартной арабской формой записи:

```
0 1 2 3 4 5 6 7 8 9
```

Специальные символы

```
+ - * / = < > [ ] , . : ; ' ( ) { } @ # $ & ^
```

применяются в основном в качестве знаков арифметических операций, разделителей, ограничителей и т.д. Из специальных символов формируются *составные символы*:

```
<>  <=  >=  ..  (.  .)  (*  *)  //  :=
```

Они служат, в частности, для обозначения операций типа "не равно", "больше или равно", указания диапазонов значений, комментирования программы, т.д.

Все перечисленные знаки отражены на клавиатуре и при нажатии соответствующих клавиш появляются на экране. Как вы видите, среди них нет русских букв, хотя на клавиатуре вашего компьютера они наверняка присутствуют. Дело в том, что такие буквы в языке Delphi несут чисто информационную нагрузку и используются только в качестве данных или при написании комментария к программе.

2.1.2. Числа

Одно и то же число можно записать самыми разными способами, например:

```
15          { целое }
15.0        { вещественное с фиксированной точкой }
1.5E01      { вещественное с плавающей точкой }
$F          { шестнадцатиричное }
```

В языке Delphi имеется возможность применять все способы записи, но чаще всего используют целые и вещественные числа.

Целые числа состоят только из цифр и знака + или –. Если знак опущен и число не равно 0, то оно рассматривается как положительное, например:

```
0           { 0 интерпретируется как целое число }
17          { положительное целое число }
-39         { отрицательное целое число  }
```

Вещественные числа содержат целую и дробную части, разделенные точкой:

```
0.0         { 0 интерпретируется как вещественное число }
133.5       { положительное вещественное число }
-0.7        { отрицательное вещественное число  }
```

Вещественные числа могут быть представлены в двух формах: с фиксированной и плавающей точкой.

Форма с *фиксированной точкой* совпадает с обычной записью чисел, например:

```
27800       { точка в конце числа опущена }
0.017
3.14
```

Форма с *плавающей точкой* используется при работе с очень большими или очень малыми числами. В этой форме число, стоящее перед буквой E, умножается на 10 в степени, указанной после буквы E:

```
7.13E+14    { 7.13 x 10^14 }
1.7E-5      { 1.7 x 10^-5 }
3.14E00     { 3.14 x 10^0 = 3.14}
```

Число, стоящее перед буквой E, называется *мантиссой*, а число после буквы E — *порядком*.

В этой книге мы чаще будем использовать форму с фиксированной точкой, так как она воспринимается лучше второй формы и совпадает с привычной математической записью чисел.

2.1.3. Слова-идентификаторы

Неделимые последовательности символов алфавита образуют *слова* (*идентификаторы*). Идентификатор начинается с буквы и не должен содержать пробелов. После первого символа допускаются буквы и цифры. Напоминаем, что символ подчеркивания считается буквой.

При написании идентификаторов могут использоваться как прописные, так и строчные буквы (между ними не делается различий). Длина идентификатора может быть любой, но значимы только первые 255 символов (вполне достаточный предел, не так ли). Примеры написания идентификаторов приведены ниже:

Правильно	*Неправильно*
RightName	Wrong Name
E_mail	E–mail
_5inches	5inches

Все идентификаторы подразделяются на зарезервированные слова, стандартные директивы, стандартные идентификаторы и идентификаторы программиста.

Зарезервированные (ключевые) слова составляют основу языка Delphi, любое их искажение вызовет ошибку компиляции. Вот полный перечень зарезервированных слов:

```
and
as
asm
array
begin
case
class
const
constructor
destructor
dispinterface
div
do
downto
else
end
except
exports
file
finally
finalization
for
function
goto
if
implementation
in
inherited
inline
initialization
interface
is
label
library
mod
nil
not
object
of
or
out
packed
procedure
program
property
raise
record
repeat
resourcestring
set
shl
shr
string
then
threadvar
to
try
type
unit
until
uses
var
while
with
xor
```

Стандартные директивы интерпретируются либо как зарезервированные слова, либо как идентификаторы программиста в зависимости от контекста, в котором используются. Вот они:

```
absolute
abstract
assembler
at
automated
cdecl
default
dispid
dynamic
export
external
far
forward
index
message
name
near
nodefault
on
overload
override
pascal
private
protected
public
published
read
register
reintroduce
resident
stdcall
stored
virtual
write
```

Стандартные идентификаторы — это имена стандартных подпрограмм, типов данных языка Delphi, т.д. В качестве примера приведем имена подпрограмм ввода и вывода данных и нескольких математических функций. Вы, без сомнения, сами угадаете их назначение:

```
Read    Write    Sin    Cos    Exp    Ln
```

Идентификаторы программиста определяются программистом, т.е вами, и носят произвольный характер. Если идентификатор состоит из двух или более смысловых частей, то для удобства их лучше выделять заглавной буквой или разделять символом подчеркивания:

```
LowProfit          AverageProfit        HighProfit
Price_One          Price_Two            Price_Sum
```

Имя идентификатора обязательно должно нести смысловую нагрузку, тогда вы сможете читать программу как книгу и не потратите время на расшифровку непонятных обозначений.

2.1.4. Комментарии

С помощью *комментариев* вы можете пояснить логику работы своей программы. Комментарий пропускается компилятором и может находиться в любом месте программы. Комментарием является:

```
{ Любой текст в фигурных скобках }
(* Любой текст в круглых скобках со звездочками *)
// Любой текст от двойной наклонной черты до конца строки
```

Если за символами **{** или **(*** сразу идет знак доллара **$**, то текст в скобках считается не комментарием, а директивой компилятора. Большинство директив компилятора являются переключателями, которые включают или выключают те или иные режимы компиляции, оптимизацию, контроль выхода значений из допустимого диапазона, переполнение, т.д. Примеры таких директив:

```
{$OPTIMIZATION ON}
{$WARNINGS ON}
{$RANGECHECKS OFF}
```

2.2. Данные

2.2.1. Понятие типа данных

Программа в процессе выполнения всегда обрабатывает какие-либо данные. Данные могут представлять собой целые и дробные числа, символы, строки, массивы, множества и др. Так как компьютер всего лишь машина, для которой данные — это последовательность нулей и единиц, он должен абсолютно точно "знать", как их интерпретировать. По этой причине все данные в языке Delphi подразделены на типы. Для описания каждого типа данных существует свой стандартный идентификатор: для целых — Integer, для дробных — Real, для строк — string и т.д. Программист может образовывать собственные типы данных и давать им произвольные имена (о том, как это делается, мы поговорим чуть позже).

Тип данных показывает, какие значения принимают данные и какие операции можно с ними выполнять. Каждому типу данных соответствует определенный объем памяти, который требуется для размещения данных. Например, в языке Delphi существует тип данных Byte. Данные этого типа принимают значения в целочисленном диапазоне от 0 до 255, могут участвовать в операциях сложения, вычитания, умножения, деления, и занимают 1 байт памяти.

Все типы данных в языке Delphi можно расклассифицировать следующим образом:

- простые типы данных. Они в свою очередь подразделяются на порядковые и вещественные типы данных. К порядковым типам относятся целочисленные, символьные, булевские, перечисляемые и интервальные типы данных;
- временной тип данных. Служит для представления значений даты и времени;
- строковые типы данных. Служат для представления последовательностей из символов, например текста;
- составные типы данных (в некоторых источниках — структурированные типы данных). Формируются на основе всех остальных типов. К ним относятся массивы, множества, записи, файлы, классы и ссылки на классы;
- процедурные типы данных. Позволяют манипулировать процедурами и функциями как данными программы;
- указательные типы данных. Данные этих типов хранят адреса других данных, с их помощью организуются различные динамические структуры: списки, деревья и т.д.;
- тип данных с непостоянным типом значений. Служит для представления значений, тип которых заранее неизвестен; с его помощью легко организуется работа со списком разнотипных значений;

Некоторые предопределенные типы данных делятся на фундаментальные и обобщенные типы. Данные *фундаментальных типов* имеют неизменный диапазон значений и объем занимаемой памяти на всех моделях компьютеров. Данные *обобщенных типов* на различных моделях компьютеров могут иметь разный диапазон значений и занимать разный объем памяти. Деление на фундаментальные и обобщенные типы характерно для целых, символьных и строковых типов данных.

По ходу изложения материала мы рассмотрим все перечисленные типы данных и более подробно объясним их смысл и назначение в программе.

2.2.2. Константы

Данные, независимо от типа, имеют некоторое значение и в программе предстают как константы или переменные. Данные, которые получили значение в начале программы и по своей природе изменяться не могут, называются *константами*. Константами, например, являются скорость света в вакууме и соотношение единиц измерения (метр, сантиметр, ярд, фут, дюйм), которые имеют научно обоснованные или традиционно принятые постоянные значения. Константы описываются с помощью зарезервированного слова **const**. За ним идет список имен констант, каждому из которых с помощью знака равенства присваивается значение. Одно присваивание отделяется от другого с помощью точки с запятой. Тип константы распознается компилятором автоматически, поэтому его не надо указывать при описании. Примеры констант:

```
const
  DelphiLanguage = 'Object Pascal';
  KylixLanguage = DelphiLanguage;
  Yard = 914.4;
  Foot = 304.8;
```

После такого описания для обращения к нужному значению достаточно указать лишь имя соответствующей константы.

Значение константы можно задавать и выражением. Эту возможность удобно использовать для комплексного представления какого-либо понятия. Например, временной промежуток, равный одному месяцу, можно задать так:

```
const
  SecondsInMinute = 60;
  SecondsInHour = SecondsInMinute * 60;
  SecondsInDay = SecondsInHour * 24;
```

Очевидно, что, изменив базовую константу SecondsInMinute, можно изменить значение константы SecondsInDay.

При объявлении константы можно указать ее тип:

```
const
  Percent: Double = 0.15;
  FileName: string = 'HELP.TXT';
```

Такие константы называются типизированными; их основное назначение — объявление константных значений составных типов данных.

2.2.3. Переменные

Переменные в отличие от констант могут неограниченное число раз менять свое значение в процессе работы программы. Если в начале программы некоторая переменная X имела значение 0, то в конце программы X может принять значение 10000. Так бывает, например, при суммировании введенных с клавиатуры чисел.

Переменные описываются с помощью зарезервированного слова **var**. За ним перечисляются идентификаторы переменных, и через двоеточие указывается их тип. Каждая группа переменных отделяется от другой группы точкой с запятой. Например:

```
var
  Index: Integer;         // переменная целого типа данных
  FileName: string;       // переменная строкового типа данных
  Sum, Profit: Double;    // группа переменных вещественного типа данных
```

В теле программы переменной можно присвоить значение. Для этого используется составной символ **:=**, например:

```
Sum := 5000.0;            // переменной Sum присваивается 5000
Percent := 0.15;          // переменной Percent присваивается 0.15
Profit := Sum * Percent; // вычисляется произведение двух переменных
                          // и его результат присваивается переменной
                          // Profit
```

Вы можете присвоить значение переменной непосредственно при объявлении:

```
var
  Index: Integer = 1;
  Delimiter: Char = ';';
```

Объявленные таким образом переменные называются *инициализированными*. На инициализированные переменные накладывается ограничение: они не могут объявляться в подпрограммах (процедурах и функциях). Если переменная не инициализируется при объявлении, то по умолчанию она заполняется нулем.

Каждый используемый в программе элемент данных должен быть описан в разделе **const** или **var**. Исключение составляют данные, заданные непосредственно *значением*, например:

```
Write(100, 200); // 100 и 200 - данные, заданные значением
```

2.3. Простые типы данных

2.3.1. Целочисленные типы данных

Целочисленные типы данных применяются для описания целочисленных данных. Для решения различных задач могут потребоваться различные целые числа. В одних задачах счет идет на десятки, в других — на миллионы. Соответственно в языке Delphi имеется несколько целочисленных типов данных, среди которых вы можете выбрать наиболее подходящий для своей задачи (таблица 2.1).

Тип данных	Диапазон значений	Объем памяти (байт)
Фундаментальные типы данных		
Byte	0..255	1
Word	0..65535	2
Shortint	–128..127	1
Smallint	–32768..32767	2
Longint	–2147483648..2147483647	4
Longword	0.. 4294967295	4
Int64	–2^63..2^63–1	8
Обобщенные типы данных		
Cardinal	0.. 4294967295	4*
Integer	–2147483648..2147483647	4*

Таблица 2.1. Целочисленные типы данных

* Примечание: количество байт памяти, требуемых для хранения переменных обобщенных типов данных, приведено для 32-разрядных процессоров семейства x86.

Пример описания целочисленных данных:

```
var
  X, Y: Integer;
  TextLength: Cardinal;
  FileSize: Longint;
```

Позволим себе дать небольшой совет. При программировании алгоритмов предпочтение следует отдавать обобщенным типам даных, поскольку они позволяют достичь максимальной производительности программ при переходе на другие модели компьютеров (например, при переходе на компьютеры, построенные на основе новых 64-разрядных процессоров). Переменные обобщенных типов данных могут храниться в памяти по-разному в зависимости от конкретной модели компьютера, и для работы с ними компилятор может генерировать наиболее оптимальный код. Однако при использовании переменных обобщенных типов данных ни в коем случае нельзя полагаться на формат их хранения в памяти, в частности на размер.

2.3.2. Вещественные типы данных

Вещественные типы данных применяются для описания вещественных данных с плавающей или с фиксированной точкой (таблица 2.2).

Тип данных	Диапазон значений	Мантисса	Объем памяти (байт)
Real	$5.0*10^{-324}..1.7*10^{308}$	15–16	8*
Real48	$2.9*10^{-39}..1.7*10^{38}$	11–12	6
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	$3.4*10^{-4932}..1.1*10^{4932}$	19–20	10
Comp	–9223372036854775808 .. 9223372036854775807	19–20	8
Currency	–922337203685477.5808 .. 922337203685477.5807	19–20	8

Таблица 2.2. Вещественные типы данных

* Примечание: количество байт памяти, требуемых для хранения переменных обобщенных типов данных, приведено для 32-разрядных процессоров семейства x86.

Пример описания вещественных данных:

```
var
  X, Y: Double;
  Z: Extended;
```

Необходимо отметить, что тип Real является обобщенным типом данных и по отношению к нему справедливо все то, что было сказано о типах Integer и Cardinal.

2.3.3. Символьные типы данных

Символьные типы применяются для описания данных, значением которых является буква, цифра, знак препинания и другие символы. Существуют два фундаментальных символьных типа данных: AnsiChar и WideChar (таблица 2.3). Они соответствуют двум различным

системам кодировки символов. Данные типа AnsiChar занимают один байт памяти и кодируют один из 256 возможных символов расширенной кодовой таблицы ANSI, в то время как данные типа WideChar занимают два байта памяти и кодируют один из 65536 символов кодовой таблицы Unicode. Кодовая таблица Unicode — это стандарт двухбайтовой кодировки символов. Первые 256 символов таблицы Unicode соответствуют таблице ANSI, поэтому тип данных AnsiChar можно рассматривать как подмножество WideChar.

Тип данных	Диапазон значений	Объем памяти (байт)
Фундаментальные типы данных		
AnsiChar	Extended ANSI character set	1
WideChar	Unicode character set	2
Обобщенный тип данных		
Char	Same as AnsiChar's range	1*

Таблица 2.3. Символьные типы данных

* Примечание: Тип данных Char является обобщенным и соответствует типу AnsiChar. Однако следует помнить, что в будущем тип данных Char может стать эквивалентным типу данных WideChar, поэтому не следует полагаться на то, что символ занимает в памяти один байт.

Пример описания переменной символьного типа:

```
var
  Symbol: Char;
```

В программе значения переменных и констант символьных типов заключаются в апострофы (не путать с кавычками!), например:

```
Symbol := 'A';  // Переменной Symbol присваивается буква A
```

2.3.4. Булевские типы данных

Булевские типы данных названы так в честь Георга Буля (George Boole), одного из авторов формальной логики. Диапазон значений данных булевских типов представлен двумя предопределенными константами: True — истина и False — ложь (таблица 2.4).

Тип данных	Диапазон значений	Объем памяти (байт)
Boolean	False (0), True (1)	1
ByteBool	False (0), True (не равно 0)	1
WordBool	False (0), True (не равно 0)	2
LongBool	False (0), True (не равно 0)	4

Таблица 2.4. Булевские типы данных

Пример описания булевских данных:

```
var
  Flag: Boolean;
  WordFlag: WordBool;
  LongFlag: LongBool;
```

Булевские типы данных широко применяются в логических выражениях и в выражениях отношения. Переменные типа Boolean используются для хранения результатов логических выражений и могут принимать только два значения: False и True (стандартные идентификаторы). Булевские типы данных ByteBool, WordBool и LongBool введены в язык Delphi специально для совместимости с другими языками, в частности с языками C и C++. Все булевские типы данных совместимы друг с другом и могут одновременно использоваться в одном выражении.

2.3.5. Определение новых типов данных

Кроме стандартных типов данных язык Delphi поддерживает типы, определенные программистом. Новый тип данных определяется с помощью зарезервированного слова **type**, за которым следует идентификатор типа, знак равенства и описание. Описание завершается точкой с запятой. Например, можно определить тип, *тождественный* существующему типу:

```
type
  TUnicode = WideChar; // TUnicode тождественен типу WideChar
  TFloat = Double;     // TFloat тождественен типу Double
```

Нетрудно заметить, что идентификаторы новых типов в примере начинаются заглавной буквой **T** (первая буква слова **type**). Такое соглашение о типах программиста принято разработчиками среды Delphi, но оно не является строгим. Тем не менее, мы рекомендуем его придерживаться, так как оно способствует более легкому восприятию исходного текста программы.

Синтаксическая конструкция **type** позволяет создавать новые порядковые типы: *перечисляемые типы* и *интервальные типы*.

2.3.6. Перечисляемые типы данных

Перечисляемый тип данных представляет собой список значений, которые может принимать переменная этого типа. Каждому значению поставлен в соответствие идентификатор, используемый в программе для указания этого значения.

```
type
  TDirection = (North, South, East, West);
```

На базе типа TDirection можно объявить переменную Direction и присвоить ей значение:

```
var
  Direction: TDirection;
begin
  Direction := North;
end.
```

На самом деле за идентификаторами значений перечисляемого типа стоят целочисленные константы. По умолчанию, первая константа равна 0, вторая — 1 и т.д. Существует возможность явно назначить значения идентификаторам:

```
type
  TSizeUnit = (Byte = 1, Kilobyte = 1024 * Byte, Megabyte = Kilobyte * 1024,
    Gigabyte = Megabyte * 1024);
```

2.3.7. Интервальные типы данных

Интервальный тип данных задается двумя константами, ограничивающими диапазон значений для переменных данного типа. Обе константы должны принадлежать одному из стандартных порядковых типов (но не вещественному и не строковому). Значение первой константы должно быть обязательно меньше значения второй. Например, определим интервальный тип TDigit:

```
type
  TDigit = 0..9;
var
  Digit: TDigit;
begin
  Digit := 5;
  Digit := 10; // Ошибка! Выход за границы диапазона
end.
```

В операциях с переменными интервального типа данных компилятор генерирует код проверки на принадлежность диапазону, поэтому последний оператор вызовет ошибку. Это очень удобно при отладке, но иногда отрицательно сказывается на скорости работы программы. Для отключения контроля диапазона откройте окно **Project Options**, выберите страницу **Compiler** и снимите пометку пункта **Range Checking**.

Данные перечисляемых и интервальных типов занимают в памяти 1, 2 или 4 байта в зависимости от диапазона значений типа. Например, если диапазон значений не превышает 256, то элемент данных занимает один байт памяти.

2.3.8. Временной тип данных

Для представления значений даты и времени в среде Delphi существует тип TDateTime. Он объявлен тождественным типу Double. Целая часть элемента данных типа TDateTime соответствует количеству дней, прошедших с полночи 30 декабря 1899 года. Дробная часть элемента данных типа TDateTime соответствует времени дня. Следующие примеры поясняют сказанное:

Значение	*Дата*	*Время*
0	30.12.1899	00:00:00
0.5	30.12.1899	12:00:00
1.5	31.12.1899	12:00:00
–1.25	29.12.1899	06:00:00
35431.0	1.1.1997	00:00:00

2.3.9. Типы данных со словом type

Если в программе создается новый тип данных, тождественный уже существующему типу данных, то компилятор не делает никаких различий между ними (ни на этапе компиляции, ни на этапе исполнения программы). По сути, создается не новый тип данных, а псевдоним для уже существующего типа данных.

```
type
  TFileName = string;
```

В приведенном выше примере тип данных TFileName является псевдонимом для стандартного типа данных string.

Для того чтобы создать действительно новый тип данных, обладающий свойствами уже существующего типа данных, но не тождественный ему, необходимо использовать зарезервированное слово **type**:

```
type
  TFileName = type string;
```

Различие между таким способом создания типа и обычным (без слова **type**) проявится при изучении массивов, записей и классов. Чтобы подготовленный читатель уже сейчас понял, в

чем оно состоит, забежим вперед и приведем поясняющий пример (новичкам советуем пропустить пример и вернуться к нему позже после изучения массивов):

```
type
  TType1 = array [1..10] of Integer;
  TType2 = type TType1;
var
  A: TType1;
  B: TType2;
begin
  B := A; // Ошибка!
end.
```

В примере переменные A и B оказываются несовместимы друг с другом из-за слова **type** в описании типа TType2. Если же переменные A и B принадлежат простым типам данных, то оператор присваивания будет работать:

```
type
  TType1 = Integer;
  TType2 = type TType1;
var
  A: TType1;
  B: TType2;
begin
  B := A; // Работает
end.
```

2.4. Операции

2.4.1. Выражения

Переменные и константы всех типов могут использоваться в выражениях. *Выражение* задает порядок выполнения действий над данными и состоит из операндов, круглых скобок и знаков операций. *Операнды* представляют собой константы, переменные и вызовы функций. *Операции* — это действия, выполняемые над операндами. Например, в выражении

```
(X + Y) / 2;
```

X, Y, 2 — операнды; '+', '/' — знаки операций; скобки говорят о том, что сначала выполняется операция сложения, потом — деления.

В простейшем случае выражение может состоять из одной переменной или константы. Круглые скобки используются, как и при записи обычных математических выражений, для управления порядком выполнения операций.

Операции в языке Delphi подразделяются на арифметические, операции отношения, логические (булевские), строковые, операцию получения адреса и другие. Выражения соответственно называются арифметическими, отношения, булевскими, строковыми и т.д. в зависимости от того, какого типа операнды и операции в них используются.

2.4.2. Арифметические операции

Арифметические операции наиболее часто используются в выражениях и выполняют арифметические действия над значениями операндов целочисленных и вещественных типов данных (таблица 2.5).

Операция	Действие	Тип операндов	Тип результата
+	Сложение	Целый, вещественный	Целый, вещественный
–	Вычитание	Целый,	Целый,

		вещественный	вещественный
*	Умножение	Целый, вещественный	Целый, вещественный
/	Деление	Целый, вещественный	Вещественный
Div	Целочисленное деление	Целый	Целый
Mod	Остаток от деления	Целый	Целый

Таблица 2.5. Арифметические операции

Операции сложения, вычитания и умножения соответствуют аналогичным операциям в математике. В отличие от них операция деления имеет три формы: обычное деление (/), целочисленное деление (**div**), остаток от деления (**mod**). Назначение каждой из операций станет понятным после изучения следующих примеров:

Выражение	*Результат*
6.8 – 2	4.8
7.3 * 17	124.1
–(5 + 9)	–14
–13.5 / 5	–2.7
–10 div 4	–2
27 div 5	5
5 div 10	0
5 mod 2	1
11 mod 4	3
–20 mod 7	–6

2.4.3. Операции отношения

Операции отношения выполняют сравнение двух операндов и определяют, истинно значение выражения или ложно (таблица 2.6). Сравниваемые величины могут принадлежать к любому порядковому типу данных. Результат всегда имеет булевский тип.

Эта группа операций специально разработана для реализации алгоритмических элементов типа “больше”, “больше или равно” и т.п., которые имеются практически в каждой программе.

Операция	Действие	Выражение	Результат
=	Равно	A = B	True, если A = B

<>	Не равно	A <> B	True, если A < B или A > B
<	Меньше	A < B	True, если A < B
>	Больше	A > B	True, если A > B
<=	Меньше или равно	A <= B	True, если A < B или A = B
>=	Больше или равно	A >= B	True, если A > B или A = B

Таблица 2.6. Операции отношения

Типичные примеры операций отношения:

Выражение	*Результат*
123 = 132	False
123 <> 132	True
17 <= 19	True
17 > 19	False
7 >= 7	True

2.4.4. Булевские операции

Результатом выполнения логических (булевских) операций является логическое значение True или False (таблица 2.7). Операндами в логическом выражении служат данные типа Boolean.

Операция	Действие	Выражение	A	B	Результат
Not	Логическое отрицание	not A	True		False
			False		True
and	Логическое И	A and B	True	True	True
			True	False	False
			False	True	False
			False	False	False
or	Логическое ИЛИ	A or B	True	True	True
			True	False	True
			False	True	True
			False	False	False

xor	Исключающее ИЛИ	A xor B	True	True	False
			True	False	True
			False	True	True
			False	False	False

Таблица 2.7. Логические операции

Результаты выполнения типичных логических операций:

Выражение	*Результат*
not (17 > 19)	True
(7 <= 8) or (3 < 2)	True
(7 <= 8) and (3 < 2)	False
(7 <= 8) xor (3 < 2)	True

2.4.5. Операции с битами

Если операнды в булевской операции имеют целочисленный тип, то операция выполняется над битами операндов и называется побитовой. К побитовым операциям также относятся операции сдвига битов влево (**shl**) и вправо (**shr**).

Операция	Действие	Тип операндов	Тип результата
not	Побитовое отрицание	Целый	Целый
and	Побитовое И	Целый	Целый
or	Побитовое ИЛИ	Целый	Целый
xor	Побитовое исключающее ИЛИ	Целый	Целый
shl	Сдвиг влево	Целый	Целый
shr	Сдвиг вправо	Целый	Целый

Таблица 2.8. Побитовые операции

Примеры побитовых операций:

Выражение	*Результат*
not $FF00	$00FF
$FF00 or $0FF0	$FFF0
$FF00 and $0FF0	$0F00
$FF00 xor $0FF0	$F0F0

$FF00 shl 4	$F000
$FF00 shr 4	$0FF0

2.4.6. Очередность выполнения операций

При выполнении выражений одни операции выполняются раньше других. Например, в выражении

20 + 40 / 2

сначала произойдет деление (ибо скобок, меняющих естественный порядок выполнения операций, нет) и только потом — сложение. Выполнение каждой операции происходит с учетом ее приоритета. Не зная приоритета каждой операции, крайне трудно правильно записать даже самое простое выражение. Значения приоритетов для рассмотренных выше операций представлены в таблице 2.9.

Операция	Приоритет	Описание
–, not	Первый	Унарный минус, отрицаиие
*, /, div, mod, and	Второй	Операции типа умножение
+, –, or, xor	Третий	Операции типа сложение
=, <>, <,>, <=, >=	Четвертый	Операции отношения

Таблица 2.9. Приоритет операций

Чем выше приоритет (первый — высший), тем раньше операция будет выполнена.

2.5. Консольный ввод-вывод

2.5.1. Консольное приложение

Решение самой простой задачи на компьютере не обходится без операций ввода-вывода информации. Ввод данных — это передача данных от внешнего устройства в оперативную память для обработки. Вывод — обратный процесс, когда данные передаются после обработки из оперативной памяти на внешнее устройство. Внешним устройством может служить консоль ввода-вывода (клавиатура и монитор), принтер, гибкий или жесткий диск и другие устройства.

Сейчас мы рассмотрим лишь средства консольного ввод-вывода данных. Консоль — это клавиатура плюс монитор. С клавиатуры данные вводятся в программу, а на монитор выводятся результаты ее работы. Консольная модель ввода-вывода, при которой данные представляются потоком символов, не позволяет использовать графических средств. Однако она очень подходит для изучения языка Delphi, так как не загромождает примеры программ излишней информацией о среде и библиотеках программирования.

Итак, давайте последовательно создадим консольное приложение:

4. Запустите среду Delphi, выберите в главном меню команду **File | Close All**, а затем — команду **File | New**.
5. Выберите “Console Application” и нажмите “OK” (рисунок 2.1).

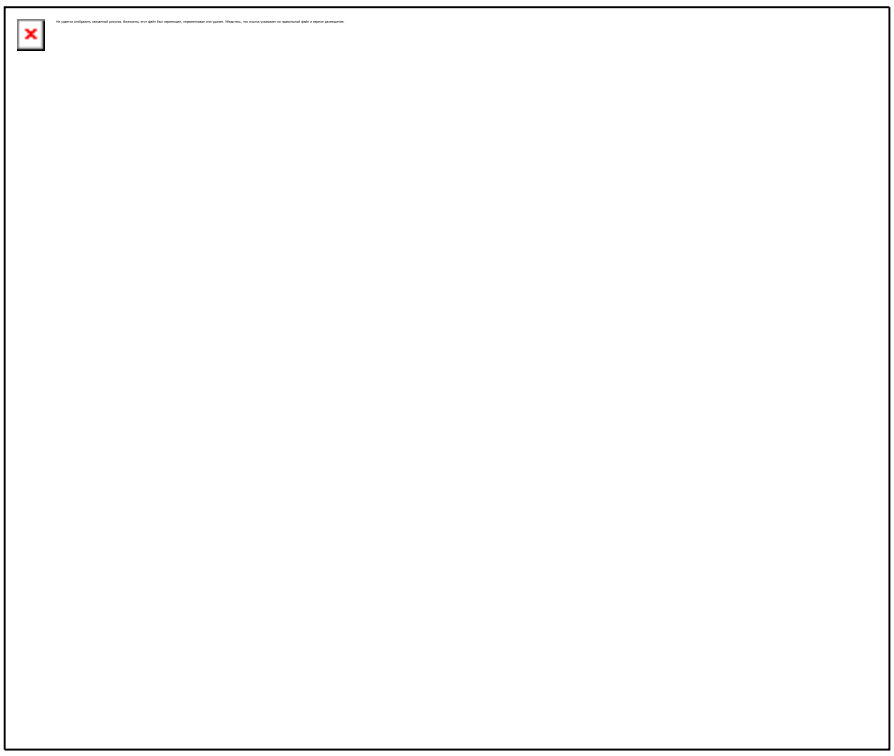


Рисунок 2.1. Окно среды Delphi для создания нового проекта

6. В появившемся окне между ключевыми словами BEGIN и END введите следующие строчки (рисунок 2.2):

```
Writeln('Press Enter to exit...');
ReadLn;
```

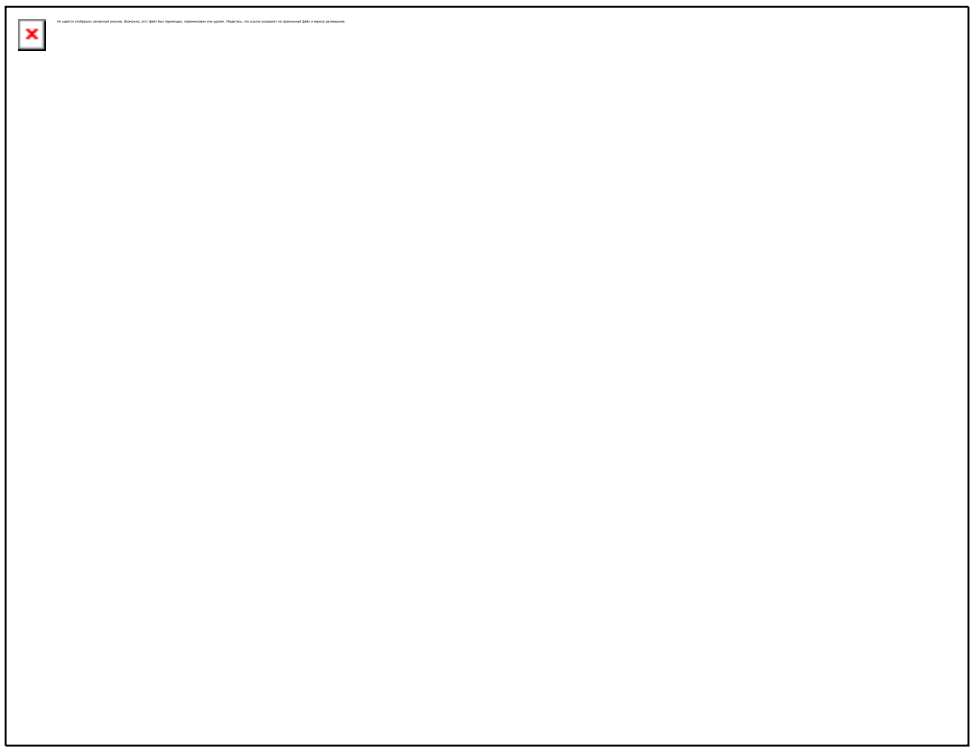


Рисунок 2.2. Текст простейшей консольной программы в окне редактора кода

7. Скомпилируйте и выполните эту программу, щелкнув на пункте **Run | Run** главного меню среды Delphi. На экране появится черное окно (рисунок 2.3), в левом верхнем углу которого будет содержаться текст "Press ENTER to exit..." ("Нажмите клавишу Enter ...").

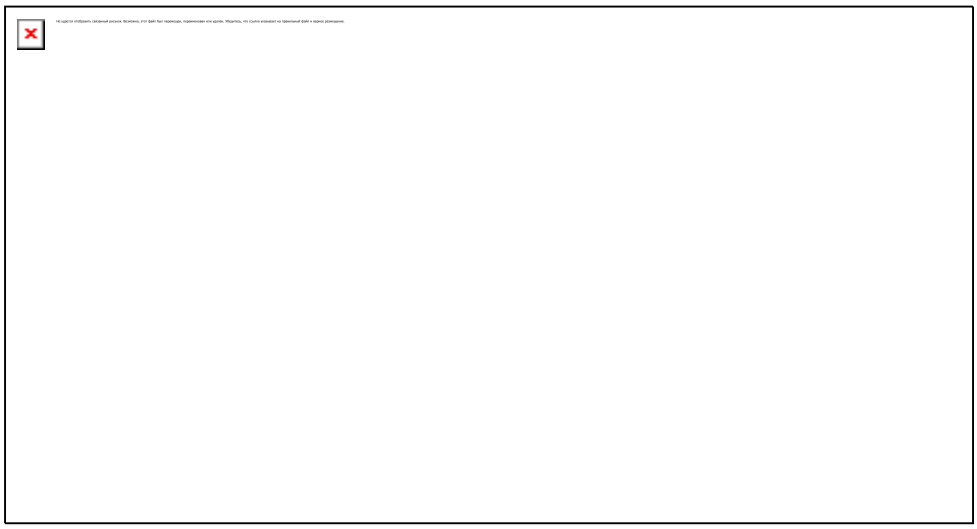


Рисунок 2.3. Окно работающей консольной программы

8. Нажмите в этом окне клавишу Enter — консольное приложение завершится.

Теперь, когда есть основа для проверки изучаемого материала, рассмотрим операторы консольного ввода-вывода. К ним относятся Write, Writeln, Read, Readln.

2.5.2. Консольный вывод

Инструкции Write и Writeln служат для вывода чисел, символов, строк и булевских значений на экран. Они имеют следующий формат:

```
Write(Y1, Y2, ... ,Yn);
Writeln(Y1, Y2, ... ,Yn);
```

где Y1, Y2,..., Yn — константы, переменные и результаты выражений. Инструкция Writeln аналогична Write, но после своего выполнения переводит курсор в начало следующей строки.

Если инструкции Write и Writeln записаны без параметров:

```
Write;
Writeln;
```

то это вызывает пропуск на экране соответственно одной позиции и одной строки.

2.5.3. Консольный ввод

Инструкции ввода обеспечивают ввод числовых данных, символов, строк для последующей обработки в программе. Формат их прост:

```
Read(X1, X2, ... ,Xn);
Readln(X1, X2, ... ,Xn);
```

где X1, X2, ..., Xn — переменные, для которых осуществляется ввод значений. Пример:

```
Read(A);   // Вводится значение переменной A
Readln(B); // Вводится значение переменной B
```

Если одна инструкция вводит несколько значений:

```
Read(A, B);
```

то все эти значения надо набрать на клавиатуре, отделяя одно значение от другого пробелом, и нажать клавишу Enter.

Если вводится одно значение:

```
Read(C);
```

то его следует набрать и нажать клавишу Enter. С этого момента программа может обрабатывать введенное значение в соответствии с алгоритмом решаемой задачи.

Инструкция Readln отличается от Read только одним свойством: каждое выполнение инструкции Readln переводит курсор в начало новой строки, а после выполнения Read курсор остается в той же строке, где и был (потренеруйтесь — и вы быстро поймете разницу).

В простейшем случае в инструкциях Read и Readln параметры можно вообще не указывать:

```
Read;
Readln;
```

Оба этих оператора останавливают выполнение программы до нажатия клавиши Enter.

2.6. Структура программы

Читатель уже достаточно много знает об отдельных элементах программы, пора изучить ее общую структуру. Синтаксически программа состоит из заголовка, списка подключаемых к программе модулей и программного блока:

```
Заголовок программы                program <имя программы>;
    Директивы компилятора          {$<директивы>}

Подключение модулей                uses <имя>, ..., <имя>;

Программный блок
    Константы                      const ...;
    Типы данных                    type  ...;
    Переменные                     var   ...;
    Процедуры                      procedure <имя>; begin ... end;
    Функции                        function  <имя>; begin ... end;
    Тело программы                 begin
                                     <операторы>
                                   end.
```

Любая секция в программном блоке кроме тела программы может отсутствовать. Секции описания констант, типов данных, переменных, процедур и функций могут встречаться в программе любое количество раз и следовать в произвольном порядке. Главное, чтобы все описания были сделаны до того, как они будут использованы (иначе компилятор просто не поймет того, что вы написали).

2.6.1. Заголовок программы

Заголовок программы должен совпадать с именем программного файла. Он формируется автоматически при сохранении файла на диске и его не следует изменять вручную. Например, заголовок программы в файле Console.dpr выглядит так:

```
program Console;
```

Одним заголовком сказать можно немного, поэтому для подробного рассказа о назначении программы, нюансах алгоритма и других вещах применяют комментарий, например:

```
{************************************************}
{    Демонстрационный пример                     }
{    A.Valvachev, K.Surkov, D.Surkov, Yu.Chetyrko    }
{************************************************}
```

После сведений о программе и разработчиках принято размещать директивы компилятора. Например, следующая директива всегда включается в текст консольного приложения:

```
{$APPTYPE CONSOLE}
```

2.6.2. Подключение модулей

Секция подключения модулей предназначена для встраивания в программу стандартных и разработанных вами библиотек подпрограмм и классов (о подпрограммах и классах читайте ниже). Эта секция состоит из зарезервированного слова **uses** и списка имен подключаемых библиотечных модулей. При написании программ, эмулирующих текстовый режим, подключается по крайней мере модуль SysUtils. В нем содержатся определения часто используемых типов данных и подпрограмм:

```
uses
  SysUtils;
```

С момента подключения все ресурсы модуля (типы данных, константы, переменные, процедуры и функции) становятся доступны программисту.

2.6.3. Программный блок

Важнейшим понятием в языке Delphi является так называемый блок. По своей сути *блок* — это программа в целом или логически обособленная часть программы, содержащая описательную и исполнительную части. В первом случае блок называется *глобальным*, во втором — *локальным*. Глобальный блок — это основная программа, он присутствует всегда; локальные блоки — это необязательные подпрограммы (они рассмотрены ниже). Локальные блоки могут содержать в себе другие локальные блоки (т.е. одни подпрограммы могут включать в себя другие подпрограммы). Объекты программы (типы, переменные и константы) называют глобальными или локальными в зависимости от того, в каком блоке они объявлены.

С понятием блока тесно связано понятие области действия программных объектов. *Область действия* трактуется как допустимость использования объектов в том или ином месте программы. Правило здесь простое: объекты программы можно использовать в пределах блока, где они описаны, и во всех вложенных в него блоках. Отсюда следует вывод — с глобальными объектами можно работать в любом локальном блоке.

Тело программы является исполнительной частью глобального блока. Именно из него вызываются для выполнения описанные выше процедуры и функции. Тело программы начинается зарезервированным словом **begin** (начало), далее следуют операторы языка, отделенные друг от друга точкой с запятой. Завершает тело программы зарезервированное слово **end** (конец) с точкой. Тело простейшей консольной программы выглядит так:

```
begin
  Writeln('Press Enter to exit...');
  Readln;
end.
```

На этом мы заканчиваем рассмотрение структуры программы и переходим к содержимому тела программы — операторам.

2.7. Операторы

2.7.1. Общие положения

Основная часть программы на языке Delphi представляет собой последовательность операторов, выполняющих некоторое действие над данными, объявленными в секции описания данных. Операторы выполняются строго последовательно в том порядке, в котором они записаны в тексте программы и отделяются один от другого точкой с запятой.

Все операторы принято в зависимости от их назначения разделять на две группы: простые и структурные. Простые операторы не содержат в себе никаких других операторов. К ним относятся операторы присваивания, вызова процедуры и безусловного перехода. Структурные операторы содержат в себе простые или другие структурные операторы и подразделяются на составной оператор, условные операторы и операторы повтора.

При изучении операторов мы рекомендуем вам обратить особое внимание на наши рекомендации по поводу того, где какой оператор надо применять. Это избавит вас от множества ошибок в практической работе.

2.7.2. Оператор присваивания

Оператор присваивания (:=) вычисляет выражение, заданное в его правой части, и присваивает результат переменной, идентификатор которой расположен в левой части. Например:

```
X := 4;
Y := 6;
Z := (X + Y) / 2;
```

Во избежании ошибок присваивания необходимо следить, чтобы тип выражения был совместим с типом переменной. Под *совместимостью типов данных* понимается возможность автоматического преобразования значений одного типа данных в значения другого типа данных. Например, все целочисленные типы данных совместимы с вещественными (но не наоборот!).

В общем случае для числовых типов данных действует следующее правило: выражение с более узким диапазоном возможных значений можно присвоить переменной с более широким диапазоном значений. Например, выражение с типом данных Byte можно присвоить переменной с типом данных Integer, а выражение с типом данных Integer можно присвоить переменной с типом данных Real. В таких случаях преобразование данных из одного представления в другое выполняется автоматически:

```
var
  B: Byte;
  I: Integer;
  R: Real;
begin
  B := 255;
  I := B + 1;     // I = 256
  R := I + 0.1;   // R = 256.1
  I := R;         // Ошибка! Типы данных несовместимы по присваиванию
end.
```

Исключение составляет случай, когда выражение принадлежит 32-разрядному целочисленному типу данных (например, Integer), а переменная — 64-разрядному целочисленному типу данных Int64. Для того, чтобы на 32-разрядных процессорах семейства x86 вычисление выражения происходило правильно, необходимо выполнить явное преобразование одного из операндов выражения к типу данных Int64. Следующий пример поясняет сказанное:

```
var
  I: Integer;
  J: Int64;
begin
  I := MaxInt;         // I =  2147483647 (максимальное целое)
  J := I + 1;          // J = -2147483648 (неправильно: ошибка переполнения!)
  J := Int64(I) + 1;   // J =  2147483648 (правильно: вычисления в формате Int64)
end.
```

2.7.3. Оператор вызова процедуры

Оператор вызова процедуры представляет собой не что иное, как имя стандартной или пользовательской процедуры. О том, что это такое, вы узнаете чуть позже, а пока достаточно просто наглядного представления. Примеры вызова процедур:

```
Writeln('Hello!'); // Вызов стандартной процедуры вывода данных
MyProc;            // Вызов процедуры, определенной программистом
```

2.7.4. Составной оператор

Составной оператор представляет собой группу из произвольного числа операторов, отделенных друг от друга точкой с запятой и заключенную в так называемые операторные скобки — **begin** и **end**:

```
begin
  <оператор 1>;
  <оператор 2>;
  …
  <оператор N>
end
```

Частным случаем составного оператора является тело следующей программы:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  X, Y: Integer;

begin
  X := 4;
  Y := 6;
  Writeln(X + Y);
  Writeln('Press Enter to exit...');
  Readln; // Точка с запятой после этого оператора не обязательна
end.
```

Хотя символ точки с запятой служит разделителем между операторами и перед словом **end** может опускаться, мы рекомендуем ставить его в конце каждого оператора (как в примере), чтобы придать программе более красивый вид и избежать потенциальных ошибок при наборе текста.

Составной оператор может находиться в любом месте программы, где разрешен простой оператор. Он широко используется с условными операторами и операторами повтора.

2.7.5. Оператор ветвления if

Оператор ветвления **if** — одно из самых популярных средств, изменяющих естественный порядок выполнения операторов программы. Вот его общий вид:

```
if <условие> then
  <оператор 1>
else
  <оператор 2>;
```

Условие — это выражение булевского типа, оно может быть простым или сложным. Сложные условия образуются с помощью логических операций и операций отношения. Обратите внимание, что перед словом **else** точка с запятой не ставится.

Логика работы оператора **if** очевидна: выполнить оператор 1, если условие истинно, и оператор 2, если условие ложно. Поясним сказанное на примере:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  A, B, C: Integer;

begin
  A := 2;
  B := 8;
  if A > B then
    C := A
  else
    C := B;
  Writeln('C=', C);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

В данном случае значение выражения A > B ложно, следовательно на экране появится сообщение C=8.

У оператора **if** существует и другая форма, в которой **else** отсутствует:

```
if <условие> then <оператор>;
```

Логика работы этого оператора **if** еще проще: выполнить оператор, если условие истинно, и пропустить оператор, если оно ложно. Поясним сказанное на примере:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  A, B, C: Integer;

begin
  A := 2;
  B := 8;
  C := 0;
  if A > B then C := A + B;
  Writeln('C=', C);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

В результате на экране появится сообщение C=0, поскольку выражение A > B ложно и присваивание C := A + B пропускается.

Один оператор **if** может входить в состав другого оператора **if**. В таком случае говорят о вложенности операторов. При вложенности операторов каждое **else** соответствует тому **then**, которое непосредственно ему предшествует. Например:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  A: Integer;

begin
  Readln(A);
  if A >= 0 then
    if A <= 100 then
      Writeln('A попадает в диапазон 0 - 100.')
    else
      Writeln('A больше 100.')
  else
    Writeln('A меньше 0.');
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Конструкций со степенью вложенности более 2–3 лучше избегать из-за сложности их анализа при отладке программ.

2.7.6. Оператор ветвления case

Оператор ветвления **case** является удобной альтернативой оператору **if**, если необходимо сделать выбор из конечного числа имеющихся вариантов. Он состоит из выражения, называемого *переключателем*, и альтернативных операторов, каждому из которых предшествует свой *список допустимых значений переключателя*:

```
case <переключатель> of
  <список №1 значений переключателя>: <оператор 1>;
  <список №2 значений переключателя>: <оператор 2>;
      ...
  <список №N значений переключателя>: <оператор N>;
  else <оператор N+1>
end;
```

Оператор **case** вычисляет значение переключателя (который может быть задан выражением), затем последовательно просматривает списки его допустимых значений в поисках вычисленного значения и, если это значение найдено, выполняет соответствующий ему оператор. Если переключатель не попадает ни в один из списков, выполняется оператор, стоящий за словом **else**. Если часть **else** отсутствует, управление передается следующему за словом **end** оператору.

Переключатель должен принадлежать порядковому типу данных. Использовать вещественные и строковые типы в качестве переключателя не допускается.

Список значений переключателя может состоять из произвольного количества констант и диапазонов, отделенных друг от друга запятыми. Границы диапазонов записываются двумя константами через разграничитель в виде двух точек (**..**). Все значения переключателя должны быть уникальными, а диапазоны не должны пересекаться, иначе компилятор сообщит об ошибке. Тип значений должен быть совместим с типом переключателя. Например:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Day: 1..31;

begin
  Readln(Day);
  case Day of
    20..31: Writeln('День попадает в диапазон 20 - 31.');
    1, 5..10: Writeln('День попадает в диапазон 1, 5 - 10.');
    else Writeln('День не попадает в заданные диапазоны.');
  end;
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Если значения переключателя записаны в возрастающем порядке, то поиск требуемого оператора выполняется значительно быстрее, так как в этом случае компилятор строит оптимизированный код. Учитывая сказанное, перепишем предыдущий пример:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Day: 1..31;

begin
  Readln(Day);
  case Day of
    1, 5..10: Writeln('День попадает в диапазон 1, 5 - 10.');
    20..31: Writeln('День попадает в диапазон 20 - 31.');
    else Writeln('День не попадает в заданные диапазоны.');
  end;
  Writeln('Press Enter to exit...');
  Readln;
end.
```

2.7.7. Операторы повтора — циклы

Алгоритм решения многих задач требует многократного повторения одних и тех же действий. При этом суть действий остается прежней, но меняются данные. С помощью рассмотренных выше операторов трудно представить в компактном виде подобные действия в программе. Для многократного (циклического) выполнения одних и тех же действий предназначены *операторы повтора* (*циклы*). К ним относятся операторы **for**, **while и repeat**. Все они используются для организации циклов разного вида.

Любой оператор повтора состоит из *условия повтора* и *повторяемого оператора* (*тела цикла*). Тело цикла представляет собой простой или структурный оператор. Оно выполняется столько раз, сколько предписывает условие повтора. Различие среди операторов повтора связано с различными способами записи условия повтора.

2.7.8. Оператор повтора for

Оператор повтора **for** используется в том случае, если заранее известно количество повторений цикла. Приведем наиболее распространенную его форму:

```
for <параметр цикла> := <значение 1> to <значение 2> do
  <оператор>;
```

где <параметр цикла> — это переменная любого порядкового типа данных (переменные вещественных типов данных недопустимы); <значение 1> и <значение 2> — выражения, определяющие соответственно начальное и конечное значения параметра цикла (они вычисляются только один раз перед началом работы цикла); <оператор> — тело цикла.

Оператор **for** обеспечивает выполнение тела цикла до тех пор, пока не будут перебраны все значения параметра цикла от начального до конечного. После каждого повтора значение параметра цикла увеличивается на единицу. Например, в результате выполнения следующей программы на экран будут выведены все значения параметра цикла (от 1 до 10), причем каждое значение — в отдельной строке:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  I: Integer;

begin
  for I := 1 to 10 do Writeln(I);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Заметим, что если начальное значение параметра цикла больше конечного значения, цикл не выполнится ни разу.

В качестве начального и конечного значений параметра цикла могут использоваться выражения. Они вычисляются только один раз перед началом выполнения оператора **for**. В этом состоит важная особенность цикла **for** в языке Delphi, которую следует учитывать тем, кто имеет опыт программирования на языках C/C++.

После выполнения цикла значение параметра цикла считается неопределенным, поэтому в предыдущем примере нельзя полагаться на то, что значение переменной I равно 10 при выходе из цикла.

Вторая форма записи оператора **for** обеспечивает перебор значений параметра цикла не по возрастанию, а по убыванию:

```
for <параметр цикла> := <значение 1> downto <значение 2> do
  <оператор>;
```

Например, в результате выполнения следующей программы на экран будут выведены значения параметра цикла в порядке убывания (от 10 до 1):

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  I: Integer;

begin
  for I := 10 downto 1 do Writeln(I);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Если в такой записи оператора **for** начальное значение параметра цикла меньше конечного значения, цикл не выполнится ни разу.

2.7.9. Оператор повтора repeat

Оператор повтора **repeat** используют в тех случаях, когда тело цикла должно быть выполнено перед тем, как произойдет проверка условия *завершения* цикла. Он имеет следующий формат:

```
repeat
  <оператор 1>;
      ...
  <оператор N>;
until <условие завершения цикла>;
```

Тело цикла выполняется до тех пор, пока условие завершения цикла (выражение булевского типа) не станет истинным. Оператор **repeat** имеет две характерные особенности, о которых нужно всегда помнить:

- между словами **repeat** и **until** может находиться произвольное число операторов без операторных скобок **begin** и **end**;
- так как условие завершения цикла проверяется после выполнения операторов, цикл выполняется, по крайней мере, один раз.

В следующем примере показано, как оператор **repeat** применяется для суммирования вводимых с клавиатуры чисел. Суммирование прекращается, когда пользователь вводит число 0:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  S, X: Integer;

begin
  S := 0;
  repeat
    Readln(X);
    S := S + X;
  until X = 0;
  Writeln('S=', S);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Часто бывает, что условие выполнения цикла нужно проверять перед каждым повторением тела цикла. В этом случае применяется оператор **while**, который, в отличие от оператора **repeat**, содержит условие выполнения цикла, а не условие завершения.

2.7.10. Оператор повтора while

Оператор повтора **while** имеет следующий формат:

```
while <условие> do
  <оператор>;
```

Перед каждым выполнением тела цикла происходит проверка условия. Если оно истинно, цикл выполняется и условие вычисляется заново; если оно ложно, происходит выход из цикла, т.е. переход к следующему за циклом оператору. Если первоначально условие ложно, то тело цикла не выполняется ни разу. Следующий пример показывает использование оператора **while** для вычисления суммы S = 1 + 2 + .. + N, где число N задается пользователем с клавиатуры:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  S, N: Integer;

begin
  Readln(N);
  S := 0;
  while N > 0 do
  begin
    S := S + N;
    N := N - 1;
  end;
  Writeln('S=', S);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

2.7.11. Прямая передача управления в операторах повтора

Для управления работой операторов повтора используются специальные процедуры-операторы Continue и Break, которые можно вызывать только в теле цикла.

Процедура-оператор Continue немедленно передает управление оператору проверки условия, пропуская оставшуюся часть цикла (рисунок 2.4):

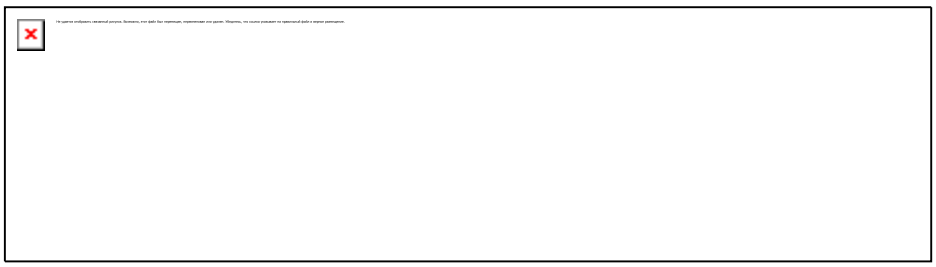


Рисунок 2.4. Схема работы процедуры-оператора Continue

Процедура-оператор Break прерывает выполнение цикла и передает управление первому оператору, расположенному за блоком цикла (рисунок 2.5):

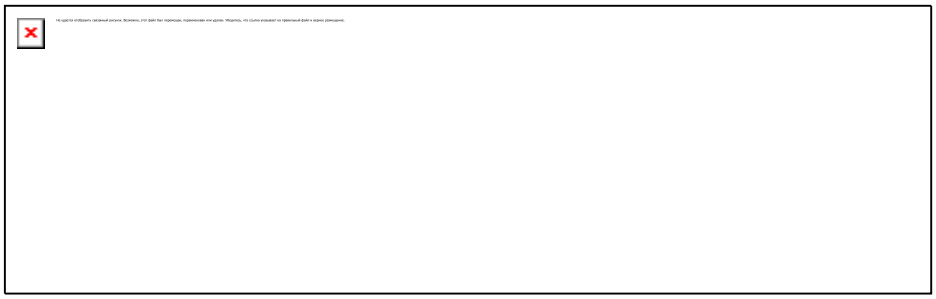


Рисунок 2.5. Схема работы процедуры-оператора Break

2.7.12. Оператор безусловного перехода

Среди операторов языка Delphi существует один редкий оператор, о котором авторы сперва хотели умолчать, но так и не решились. Это *оператор безусловного перехода* **goto** ("перейти к"). Он задумывался для того случая, когда после выполнения некоторого оператора надо выполнить не следующий по порядку, а какой-либо другой, отмеченный меткой, оператор.

Метка — это именованная точка в программе, в которую можно передать управление. Перед употреблением метка должна быть описана. Раздел описания меток начинается зарезервированным словом **label**, за которым следуют имена меток, разделенные запятыми. За последним именем ставится точка с запятой. Типичный пример описания меток:

```
label
  Label1, Label2;
```

В разделе операторов метка записывается с двоеточием. Переход на метку выполняется с помощью зарезервированного слова **goto**, за которым следует имя метки:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

label
  M1, M2;

begin
  M1:
  Write('Желаем успеха ');
  goto M2;
  Write('А этого сообщения вы никогда не увидите!');
  M2:
  goto M1;
  Writeln('в освоении среды Delphi!');
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Эта программа будет выполняться бесконечно, причем второй оператор Write не выполнится ни разу!

Внимание! В соответствии с правилами структурного программирования следует избегать применения оператора **goto**, поскольку он усложняет понимание логики программы. Оператор **goto** использовался на заре программирования, когда выразительные возможности языков были скудными. В языке Delphi без него можно успешно обойтись, применяя условные операторы, операторы повтора, процедуры Break и Continue, операторы обработки исключений (последние описаны в главе 4).

2.8. Подпрограммы

2.8.1. Общие положения

В практике программирования часто встречается ситуация, когда одну и ту же группу операторов требуется выполнить без изменений в нескольких местах программы. Чтобы избавить программиста от многократного дублирования одинаковых фрагментов, была предложена концепция подпрограмм. В этом разделе мы расскажем о том, как эта концепция реализована в языке Delphi.

Подпрограммой называется именованная логически законченная группа операторов, которую можно вызвать по имени (т.е. выполнить) любое количество раз из различных мест программы. В языке Delphi подпрограммы оформляются в виде процедур и функций.

Процедура — это подпрограмма, имя которой не может использоваться в выражениях в качестве операнда. Процедура состоит из заголовка и тела. По структуре ее можно рассматривать как программу в миниатюре. Когда процедура описана, ее можно вызвать по имени из любой точки программы (в том числе из нее самой!). Когда процедура выполнит свою задачу, программа продолжится с оператора, следующего непосредственно за оператором вызова процедуры. Использование имени процедуры в программе называется оператором вызова процедуры.

Функция также является подпрограммой, но в отличие от процедуры ее имя может использоваться в выражениях в качестве операнда, на место которого подставляется результат работы этой функции.

Все процедуры и функции языка Delphi подразделяются на две группы: встроенные и определенные программистом.

Встроенные процедуры и функции являются частью языка и могут вызываться по имени без предварительного описания. В данной главе рассматриваются лишь базовые группы

встроенных процедур и функций, остальные будут рассмотрены в других главах по ходу изложения материала.

Процедуры и функции программиста пишутся программистом, т.е. вами, в соответствии с синтаксисом языка и представляют собой локальные блоки. Предварительное описание процедур и функций программиста обязательно.

2.8.2. Стандартные подпрограммы

Арифметические функции

Abs(X)	Возвращает абсолютное значение аргумента X.
Exp(X)	Возвращает значение e^x.
Ln(X)	Возвращает натуральный логарифм аргумента X.
Pi	Возвращает значение числа π.
Sqr(X)	Возвращает квадрат аргумента X.
Sqrt(X)	Возвращает квадратный корень аргумента X.

Примеры:

Выражение	*Результат*
Abs(–4)	4
Exp(1)	2.71828182845905
Ln(Exp(1))	1
Pi	3.14159265358979
Sqr(5)	25
Sqrt(25)	5

Тригонометрические функции

ArcTan(X)	Возвращает угол, тангенс которого равен X.
Cos(X)	Возвращает косинус аргумента X (X задается в радианах).
Sin(X)	Возвращает синус аргумента X (X задается в радианах).

Примеры:

Выражение	*Результат*
ArcTan(Sqrt(3))	1.04719755119660
Cos(Pi/3)	0.5

Sin(Pi/6)	0.5

Заметим, что в состав среды Delphi входит стандартный модуль Math, который содержит высокопроизводительные подпрограммы для тригонометрических, логорифмических, статистических и финансовых вычислений.

Функции выделения целой или дробной части

Frac(X)	Возвращает дробную часть аргумента X.
Int(X)	Возвращает целую часть вещественного числа X. Результат принадлежит вещественному типу.
Trunc(X)	Возвращает целую часть вещественного числа X. Результат принадлежит целому типу.
Round(X)	Округляет вещественное число X до ближайшего целого. Если число X находится строго посередине между целыми числами, то округление выполняется до ближайшего четного целого числа (см. примеры ниже). Такое округление называется "округлением банкира", оно применяется в банках и бухгалтериях при работе с деньгами. Для других расчетных задач это округление может не подойти.

Примеры:

Выражение	*Результат*
Frac(1.5)	0.5
Int(1.5)	1.0
Trunc(1.5)	1
Round(1.5)	2
Round(2.5)	2

Функции генерации случайных чисел

Random	Возвращает случайное вещественное число в диапазоне 0 <= X < 1.
Random(I)	Возвращает случайное целое число в диапазоне 0 <= X < I.
Randomize	Заново инициализирует встроенный генератор случайных чисел новым значением, полученным от системного таймера.

Подпрограммы для работы с порядковыми величинами

Chr(X)	Возвращает символ, порядковый номер которого равен X.
Dec(X,	Уменьшает целую переменную X на 1 или на заданное

[N])	число N.
Inc(X, [N])	Увеличивает целую переменную X на 1 или на заданное число N.
Odd(X)	Возвращает True, если аргумент X является нечетным числом.
Ord(X)	Возвращает порядковый номер аргумента X в своем диапазоне значений.
Pred(X)	Возвращает значение, предшествующее значению аргумента X в своем диапазоне.
Succ(X)	Возвращает значение, следующее за значением аргумента X в своем диапазоне.

Примеры:

Выражение	*Результат*
Chr(65)	'A'
Odd(3)	True
Ord('A')	65
Pred('B')	'A'
Succ('A')	'B'

Подпрограммы для работы с датой и временем

Date	Возвращает текущую дату в формате TDateTime.
Time	Возвращает текущее время в формате TDateTime.
Now	Возвращает текущие дату и время в формате TDateTime.
DayOfWeek(D)	Возвращает день недели по дате в формате TDateTime.
DecodeDate(...)	Разбивает значение даты на год, месяц и день.
DecodeTime(...)	Разбивает значение времени на час, минуты, секунды и милисекунды.
EncodeDate(...)	Формирует значение даты по году, месяцу и дню.
EncodeTime(...)	Формирует значение времени по часу, минутам, секундам и милисекундам.

Процедуры передачи управления

Break	Прерывает выполнение цикла.
Continue	Начинает новое повторение цикла.
Exit	Прерывает выполнение текущего блока.
Halt	Останавливает выполнение программы и возвращает управление операционной системе.
RunError	Останавливает выполнение программы, генерируя ошибку времени выполнения.

Разные процедуры и функции

FillChar(...)	Заполняет непрерывную область символьным или байтовым значением.
Hi(X)	Возвращает старший байт аргумента X.
High(X)	Возвращает самое старшее значение в диапазоне аргумента X.
Lo(X)	Возвращает младший байт аргумента X.
Low(X)	Возвращает самое младшее значение в диапазоне аргумента X.
Move(...)	Копирует заданное количество байт из одной переменной в другую.
ParamCount	Возвращает количество параметров, переданных программе в командной строке.
ParamStr(X)	Возвращает параметр командной строки по его номеру.
SizeOf(X)	Возвращает количество байт, занимаемое аргументом X в памяти. Функция SizeOf особенно нужна для определения размеров переменных обощенных типов данных, поскольку представление обощенных типов данных в памяти может изменяться от одной версии среды Delphi к другой. Рекомендуем всегда использовать эту функцию для определения размера переменных любых типов данных; это считается хорошим стилем программирования.
Swap(X)	Меняет местами значения старшего и младшего байтов аргумента.
UpCase(C)	Возвращает символ C, преобразованный к верхнему регистру.

Примеры:

Выражение	*Результат*
Hi($F00F)	$F0
Lo($F00F)	$0F
High(Integer)	32767
Low(Integer)	–32768
SizeOf(Integer)	2
Swap($F00F)	$0FF0
UpCase('a')	'A'

2.8.3. Процедуры программиста

Очевидно, что встроенных процедур и функций для решения большинства прикладных задач недостаточно, поэтому приходиться придумывать собственные процедуры и функции. По своей структуре они очень напоминают программу и состоят из заголовка и блока. *Заголовок процедуры* состоит из зарезервированного слова **procedure**, имени процедуры и необязательного заключенного в круглые скобки списка формальных параметров. *Имя процедуры* — это идентификатор, уникальный в пределах программы. *Формальные параметры* — это данные, которые вы передаете в процедуру для обработки, и данные, которые процедура возвращает (подробно параметры описаны ниже). Если процедура не получает данных извне и ничего не возвращает, формальные параметры (в том числе круглые скобки) не записываются. *Тело процедуры* представляет собой локальный блок, по структуре аналогичный программе:

```
procedure <имя процедуры> ( <список формальных параметров> ) ;
const ...;
type  ...;
var   ...;
begin
  <операторы>
end;
```

Описания констант, типов данных и переменных действительны только в пределах данной процедуры. В теле процедуры можно использовать любые глобальные константы и переменные, а также вызывать любые подпрограммы (процедуры и функции).

Вызов процедуры для выполнения осуществляется по ее имени, за которым в круглых скобках следует список *фактических параметров*, т.е. передаваемых в процедуру данных:

```
<имя процедуры> ( <список фактических параметров> );
```

Если процедура не принимает данных, то список фактических параметров (в том числе круглые скобки) не указываются.

Понятие процедуры является чрезвычайно важным, так как именно оно лежит в основе одной из самых популярных технологий решения задач на языке Delphi. Технология эта внешне проста: задача разбивается на несколько логически обособленных подзадач и решение каждой из них оформляется в виде отдельной процедуры. Любая процедура может содержать в себе другие процедуры, их количество ограничено только объемом памяти вашего компьютера.

Приведем пример небольшой программы, использующей процедуру Power для вычисления числа X в степени Y. Результат вычисления процедура Power заносит в глобальную переменную Z.

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  Z: Double;

procedure Power(X, Y: Double); // X и Y - формальные параметры
begin
  Z := Exp(Y * Ln(X));
end;

begin
  Power(2, 3);                  // 2 и 3 - фактические параметры
  Writeln('2 в степени 3 = ', Z);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

2.8.4. Функции программиста

Функции программиста применяются в тех случаях, когда надо создать подпрограмму, участвующую в выражении как операнд. Как и процедура, функция состоит из заголовка и блока. *Заголовок функции* состоит из зарезервированного слова **function**, имени функции, необязательного заключенного в круглые скобки списка формальных параметров и типа возвращаемого функцией значения. Функции возвращают значения любых типов данных кроме Text и **file of** (см. файлы). *Тело функции* представляет собой локальный блок, по структуре аналогичный программе.

```
function <имя функции> ( <список формальных параметров> ): <тип результата>;
const ...;
type  ...;
var   ...;
begin
  <операторы>
end;
```

В теле функции должен находиться по крайней мере один оператор, присваивающий значение имени функции или неявной локальной переменной Result. Если таких присваиваний несколько, то результатом функции будет значение последнего из этих операторов. Преимущество от использования переменной Result состоит в том, что она может участвовать в выражениях как операнд.

В качестве примера заменим явно неуклюжую процедуру Power (см. выше) на функцию с таким же именем:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

function Power(X, Y: Double): Double;       // X и Y - формальные параметры
begin
  Result := Exp(Y * Ln(X));
end;

begin
  Writeln('2 в степени 3 = ', Power(2, 3)); // 2 и 3 - фактические параметры
  Writeln('Press Enter to exit...');
  Readln;
end.
```

2.8.5. Параметры процедур и функций

Параметры служат для передачи исходных данных в подпрограммы и для приема результатов работы этих подпрограмм.

Исходные данные передаются в подпрограмму с помощью *входных* параметров, а результаты работы подпрограммы возвращаются через *выходные* параметры. Параметры могут быть входными и выходными одновременно.

Входные параметры объявляются с помощью ключевого слова **const**; их значения не могут быть изменены внутри подпрограммы:

```
function Min(const A, B: Integer): Integer;
begin
  if A < B then Result := A
  else Result := B;
end;
```

Для объявления *выходных* параметров служит ключевое слово **out**:

```
procedure GetScreenResolution(out Width, Height: Integer);
begin
  Width := GetScreenWidth;
  Height := GetScreenHeight;
end;
```

Установка значений выходных параметров внутри подпрограммы приводит к установке значений переменных, переданных в качестве аргументов:

```
var
  W, H: Integer;
begin
  GetScreenResolution(W, H);
  ...
end;
```

После вызова процедуры GetScreenResolution переменные W и H будут содержать значения, которые были присвоены формальным параметрам Width и Height соответственно.

Если параметр является одновременно *и входным, и выходным*, то он описывается с ключевым словом **var**:

```
procedure Exchange(var A, B: Integer);
var
  C: Integer;
begin
  C := A;
  A := B;
  B := C;
end;
```

Изменение значений **var**-параметров внутри подпрограммы приводит к изменению значений переменных, переданных в качестве аргументов:

```
var
  X, Y: Integer;
begin
  X := 5;
  Y := 10;
  ...
  Exchange(X, Y);
  // Теперь X = 10, Y = 5
  ...
end;
```

При вызове подпрограмм на место **out**- и **var**-параметров можно подставлять только переменные, но не константы и не выражения.

Если при описании параметра не указано ни одно из ключевых слов **const**, **out**, или **var**, то параметр считается входным, его можно изменять, но все изменения не влияют на фактический аргумент, поскольку они выполняются с копией аргумента, создаваемой на время работы подпрограммы. При вызове подпрограммы на месте такого параметра можно использовать константы и выражения. Пример подпрограммы:

```
function NumberOfSetBits(A: Cardinal): Byte;
begin
  Result := 0;
  while A <> 0 do
  begin
    Result := Result + (A mod 2);
    A := A div 2;
  end;
end;
```

Параметр A в приведенной функции является входным, но при этом он используется в качестве локальной переменной для хранения промежуточных данных.

Разные способы передачи параметров (**const**, **out**, **var** и без них) можно совмещать в одной подпрограмме. В следующем законченном примере процедура Average принимает четыре параметра. Первые два (X и Y) являются входными и служат для передачи исходных данных. Вторые два параметра являются выходными и служат для приема в вызывающей программе результатов вычисления среднего арифметического (M) и среднего геометрического (P) от значений X и Y:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

procedure Average(const X, Y: Double; out M, P: Double);
begin
  M := (X + Y) / 2;
  P := Sqrt(X * Y);
end;

var
  M, P: Double;

begin
  Average(10, 20, M, P);
  Writeln('Среднее арифметическое = ', M);
  Writeln('Среднее геометрическое = ', P);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Существует разновидность параметров без типа. Они называются *нетипизированными* и предназначены для передачи и для приема данных любого типа. Нетипизированные параметры описываются с помощью ключевых слов **const** и **var**, при этом тип данных опускается:

```
procedure JustProc(const X; var Y; out Z);
```

Внутри подпрограммы тип таких параметров не известен, поэтому программист должен сам позаботиться о правильной интерпретации переданных данных. Заметим, что при вызове подпрограмм на место нетипизированных параметров (в том числе и на место нетипизированных const-параметров) можно подставлять только переменные.

Передача фактических аргументов в подпрограмму осуществляется через специальную область памяти — *стек*. В стек помещается либо значение передаваемого аргумента (*передача значения*), либо адрес аргумента (*передача ссылки на значение*). Конкретный способ передачи выбирается компилятором в зависимости от того, как объявлен параметр в заголовке подпрограммы. Связь между объявлением параметра и способом его передачи поясняет таблица 2.10:

Ключевое слово	Назначение	Способ передачи
<отсутствует>	Входной	Передается копия значения
const	Входной	Передается копия значения либо ссылка на значение в зависимости от типа данных
out	Выходной	Передается ссылка на значение
var	Входной и выходной	Передается ссылка на значение

Таблица 2.10. Способы передачи параметров

Если передается значение, то подпрограмма манипулирует копией аргумента. Если передается ссылка на значение, то подпрограмма манипулирует непосредственно аргументом, обращаясь к нему через переданный адрес.

2.8.6. Опущенные параметры процедур и функций

В языке Delphi существует возможность задать параметрам процедур и функций стандартные значения. Они указываются через знак равенства после типа параметра. Например, опишем процедуру, которая заполняет некоторую область памяти заданным значением:

```
procedure Initialize(var X; MemSize: Integer; InitValue: Byte = 0);
```

Для параметра InitValue задано стандартное значение, поэтому его можно опустить при вызове процедуры Initialize:

```
Initialize(MyVar, 10); // Эквивалентно Initialize(MyVar, 10, 0);
```

Подпрограмма может содержать любое количество параметров со стандартными значениями, однако такие параметры должны быть последними в списке. Другими словами, после параметра со стандартным значением не может следовать обычный параметр, поэтому следующее описание будет воспринято компилятором как ошибочное:

```
procedure Initialize(var X; InitValue: Byte = 0; MemSize: Integer); // Ошибка!
```

2.8.7. Перегрузка процедур и функций

В некоторых случаях возникает необходимость в написании подпрограмм, которые выполняют одинаковые логические действия, но над переменными разных типов данных. Например:

```
procedure IncrementInteger(var Value: Integer);
procedure IncrementReal(var Value: Real);
```

В языке Delphi существует возможность дать двум и более процедурам (функциям) одинаковые идентификаторы при условии, что все такие процедуры (функции) отличаются списком параметров. Такая возможность называется *перегрузкой*. Для указания того, что процедура (функция) перегружена, служит стандартная директива **overload**. С ее помощью вышеприведенный пример можно переписать следующим образом:

```
procedure Increment(var Value: Integer); overload; // процедура 1
procedure Increment(var Value: Real); overload;    // процедура 2
```

Какую именно процедуру использовать в том или ином случае компилятор будет определять на этапе компиляции программы по типам фактических аргументов, передаваемых при вызове.

```
var
  X: Integer;
  Y: Real;
begin
  X:=1;
  Y:=2.0;
  Increment(X); // Вызывается процедура 1
  Increment(Y); // Вызывается процедура 2
end.
```

При перегрузке процедур и функций существует особенность, связанная с целочисленными типами данных. Допустим, имеются две процедуры:

```
procedure Print(X: Shortint); overload; // процедура 1
procedure Print(X: Longint); overload;  // процедура 2
```

Если мы попробуем вызвать процедуру Print, указав в качестве фактического аргумента целочисленную константу, то увидим, что выбор компилятором варианта процедуры зависит от значения константы.

```
Print(5);    // Вызывается процедура 1
Print(150);  // Вызывается процедура 2
Print(-500); // Вызывается процедура 2
Print(-1);   // Вызывается процедура 1
```

Очевидно, что одно и то же число может интерпретироваться и как Longint, и как Shortint (например, числа 5 и –1). Логика компилятора в таких случаях такова: если значение фактического параметра попадает в диапазон значений нескольких типов, по которым происходит перегрузка, то компилятор выбирает процеудуру (функцию), у которой тип параметра имеет меньший диапазон значений. Например, вызов Print(5) будет означать вызов того варианта процедуры, который имеет тип параметра Shortint. А вот вызов Print(150) будет означать вызов того варианта процедуры, который имеет тип параметра Longint, т.к. число 150 не вмещается в диапазон значений типа данных Shortint.

Поскольку в нынешней версии среды Delphi обощенный тип данных Integer совпадает с фундаментальным типом данных Longint, следующий вариант перегрузки является ошибочным:

```
procedure Print(X: Integer); overload;
procedure Print(X: Longint); overload; // Ошибка!
```

Такая же ошибка возникает при использовании пользовательских типов данных, определенных через общий базовый тип.

```
type
  TMyInteger = Integer;

procedure Print(X: Integer); overload;
procedure Print(X: TMyInteger); overload; // Ошибка!
```

Что делать в тех случаях, когда такая перегрузка просто необходима? Для этого пользовательский тип данных необходимо создавать с использованием ключевого слова **type**:

```
type
  TMyInteger = type Integer;

procedure Print(X: Integer); overload;
procedure Print(X: TMyInteger); overload; // Правильно
```

Необходимо заметить, что при использовании перегруженных процедур (функций), у которых есть параметры, имеющие стандартные значения, нужно быть очень внимательным, т.к. могут возникнуть ситуации, когда компилятор просто не будет знать, какую именно процедуру (функцию) вы хотите вызвать. Например:

```
procedure Increment(var Value: Real; Delta: Real = 1.0); overload; // процедура 1
procedure Increment(var Value: Real); overload;                    // процедура 2
```

Вызов процедуры Increment с одним параметром вызовет неоднозначность:

```
var
  X: Real;
begin
  Increment(X, 10); // Вызывается процедура 1
  Increment(X);     // Ошибка! Неоднозначность
end.
```

Запрещается также перегружать функции, которые отличаются лишь типом возвращаемого значения.

```
function SquareRoot(X: Integer): Single; overload;
function SquareRoot(X: Integer): Double; overload; // Ошибка!
```

2.8.8. Соглашения о вызове подпрограмм

В различных языках программирования используются различные правила вызова подпрограмм. Для того чтобы из программ, написанных на языке Delphi, возможно было

вызывать подпрограммы, написанные на других языках (и наоборот), в языке Delphi существуют директивы, соответствующие четырем известным соглашениям о вызове подпрограмм: **register**, **stdcall**, **pascal**, **cdecl**.

Директива, определяющая правила вызова, помещается в заголовок подпрограммы, например:

```
procedure Proc; register;
function Func(X: Integer): Boolean; stdcall;
```

Директива **register** задействует регистры процессора для передачи параметров и поэтому обеспечивает наиболее эффективный способ вызова подпрограмм. Эта директива применяется по умолчанию. Директива **stdcall** используется для вызова стандартных подпрограмм операционной системы. Директивы **pascal** и **cdecl** используются для вызова подпрограмм, написанных на языках Delphi и C/C++ соответственно.

2.8.9. Рекурсивные подпрограммы

В ряде приложений алгоритм решения задачи требует вызова подпрограммы из раздела операторов той же самой подпрограммы, т.е. подпрограмма вызывает сама себя. Такой способ вызова называется *рекурсией*. Рекурсия полезна прежде всего в тех случаях, когда основную задачу можно разделить на подзадачи, имеющие ту же структуру, что и первоначальная задача. Подпрограммы, реализующие рекурсию, называются *рекурсивными*. Для понимания сути рекурсии лучше понимать рекурсивный вызов как вызов другой подпрограммы. Практика показывает, что в такой трактовке рекурсия воспринимается значительно проще и быстрее.

Приведенная ниже программа содержит функцию Factorial для вычисления факториала. Напомним, что факториал числа определяется через произведение всех натуральных чисел, меньших либо равных данному (факториал числа 0 принимается равным 1):

X! = 1 * 2 * ... * (X – 2) * (X – 1) * X

Из определения следует, что факториал числа X равен факториалу числа (X – 1), умноженному на X. Математическая запись этого утверждения выглядит так:

X! = (X – 1)! * X, где 0! = 1

Последняя формула используется в функции Factorial для вычисления факториала:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

function Factorial(X: Integer): Longint;
begin
  if X = 0  then // Условие завершения рекурсии
    Factorial := 1
  else
    Factorial := Factorial(X - 1) * X;
end;

begin
  Writeln('4! = ', Factorial(4)); // 4! = 1 * 2 * 3 * 4 = 24
  Writeln('Press Enter to exit...');
  Readln;
end.
```

При написании рекурсивных подпрограмм необходимо обращать особое внимание на условие завершения рекурсии, иначе рекурсия окажется бесконечной и приложение будет прервано из-за ошибки переполнения стека.

Бывает встречается такая рекурсия, когда первая подпрограмма вызывает вторую, а вторая — первую. Такая рекурсия называется *косвенной*. Очевидно, что записанная первой подпрограмма будет содержать еще неизвестный идентификатор второй подпрограммы (компилятор не умеет заглядывать вперед). В результате компилятор сообщит об ошибке использования неизвестного идентификатора. Эта проблема решается с помощью упреждающего (предварительного) описания процедур и функций.

2.8.10. Упреждающее объявление процедур и функций

Для реализации алгоритмов с косвенной рекурсией в языке Delphi предусмотрена специальная директива предварительного описания подпрограмм **forward**. Предварительное описание состоит из заголовка подпрограммы и следующего за ним зарезервированного слова **forward**, например:

```
procedure Proc; forward;
function Func(X: Integer): Boolean; forward;
```

Заметим, что после такого первичного описания в полном описании процедуры или функции можно не указывать список формальных параметров и тип возвращаемого значения (для функции). Например:

```
procedure Proc2(<формальные параметры>); forward;

procedure Proc1;
begin
  ...
  Proc2(<фактические параметры>);
  ...
end;

procedure Proc2; // Список формальных параметров опущен
begin
  ...
  Proc1;
  ...
end;

begin
  ...
  Proc1;
  ...
end.
```

2.8.11. Процедурные типы данных

Наряду с уже известными типами данных в языке Delphi введен так называемый *процедурный тип,* с помощью которого обычные процедуры и функции можно интерпретировать как некоторую разновидность переменных. Определение процедурного типа состоит из зарезервированного слова **procedure** или **function**, за которым следует полное описание параметров. Для функции дополнительно указывается тип результата. Символические имена параметров никакой роли не играют, поскольку нигде не используются.

```
type
  TProc = procedure (X, Y: Integer);
  TFunc = function (X, Y: Integer): Boolean;
```

Определив процедурный тип, можно непосредственно перейти к так называемым *процедурным переменным*. Они объявляются точно так же, как и обычные переменные.

```
var
  P: TProc;
  F: TFunc;
```

При работе с процедурной переменной важно понимать, что она не дублирует код подпрограммы, а содержит лишь ее адрес. Если обратиться к такой переменной как к подпрограмме, произойдет выполнение подпрограммы, адрес которой записан в переменной.

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

function Power(X, Y: Double): Double;
begin
  Result := Exp(Y * Ln(X));
end;

type
  TFunc = function (X, Y: Double): Double;

var
  F: TFunc;

begin
  F := Power; // В переменную F заносится адрес функции Power
  Writeln('2 power 4 = ', F(2, 4)); // Вызов Power посредством F
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Обращение к процедурной переменной следует выполнять только после установки ее значения. Чтобы установка значения была корректной, процедура либо функция справа от знака присваивания не должна быть встроенной подпрограммой модуля System.

2.9. Программные модули

2.9.1. Структура модуля

Логически обособленные группы процедур и функций чрезвычайно удобно объединять в специализированные библиотеки — *модули*. Для этого язык Delphi предлагает специальные средства и доступную каждому технологию. Приведем общую структуру программного модуля:

```
Заголовок модуля                  unit <имя модуля>;
    Директивы компилятора         {$<директивы>}

Интерфейсная часть                interface
    Подключение модулей           uses <имя>, ..., <имя>;
    Константы                     const ... ;
    Типы данных                   type  ... ;
    Переменные                    var   ... ;
    Заголовки процедур            procedure <имя> (<параметры>);
    Заголовки функций             function  <имя> (<параметры>): <тип>;

Часть реализации                  implementation
    Подключение модулей           uses <имя>, ..., <имя>;
    Константы                     const ... ;
    Типы данных                   type  ... ;
    Переменные                    var   ... ;
    Реализация процедур           procedure <имя>; begin ... end;
    Реализация функций            function  <имя>; begin ... end;

Код инициализации                 initialization <операторы>
Код завершения                    finalization <операторы>
                                  end.
```

После слова **unit** записывается *имя модуля*. Оно должно совпадать с именем файла, в котором находится исходный текст модуля. Например, если файл называется MathLib.pas, то модуль должен иметь имя MathLib. Заголовок модуля формируется автоматически при сохранении файла на диске, поэтому его не следует изменять вручную. Чтобы дать модулю другой заголовок, просто сохраните его на диске под другим именем.

В разделе **interface** описываются *глобальные* данные, процедуры и функции, доступные для использования в основной программе и других модулях.

В разделе **implementation** реализуется программный код глобальных процедур и функций и описываются *локальные* данные, процедуры и функции, недоступные основной программе и другим модулям.

Блок **initialization** является необязательным. Он состоит из операторов и выполняется автоматически непосредственно перед запуском основной программы. Блоки инициализации подключенных к программе модулей выполняются в том порядке, в котором они упоминаются в секции **uses**.

Блок **finalization** тоже является необязательным. Он состоит из операторов и выполняется автоматически непосредственно после завершения основной программы. Блоки завершения подключенных к программе модулей выполняются в порядке, обратном порядку подключения модулей в секции **uses**.

Если модуль не нуждается в инициализации и завершении, блоки **initialization** и **finalization** можно опустить.

В качестве упражнения давайте создадим модуль и подключим его к основной программе (для этого сначала запустите среду Delphi):

1. Выберите в главном меню команду **File | New...**, в появившемся диалоговом окне активизируйте значок с подписью **Unit** и щелкните на кнопке **OK** (рисунок 2.6).

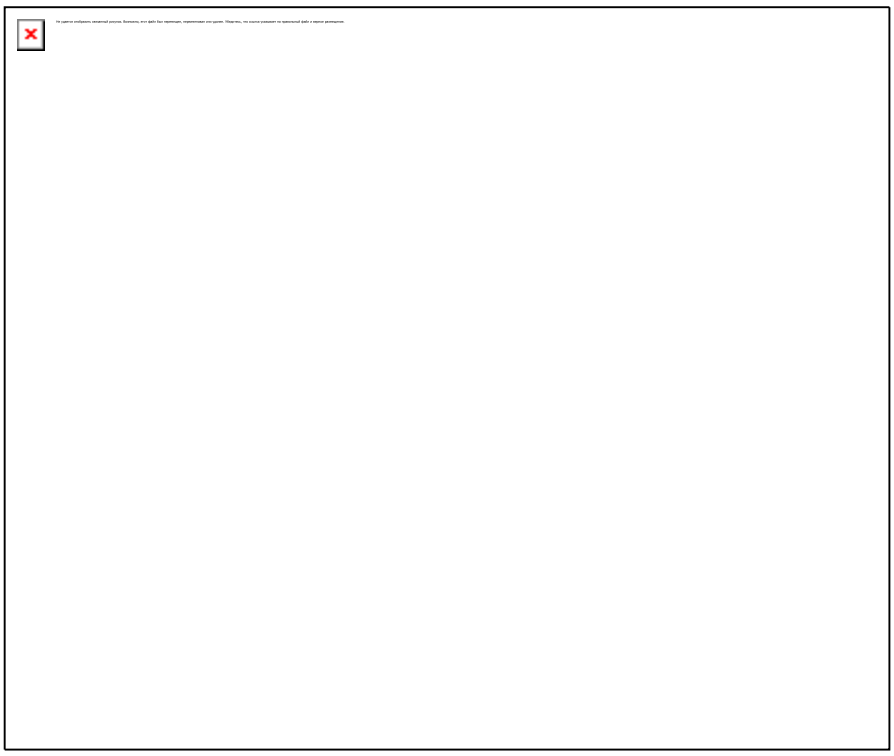


Рисунок 2.6. Окно среды Delphi для создания нового модуля

2. Вы увидите, что среда Delphi создаст в редакторе кода новую страницу с текстом нового модуля **Unit1** (рисунок 2.7):

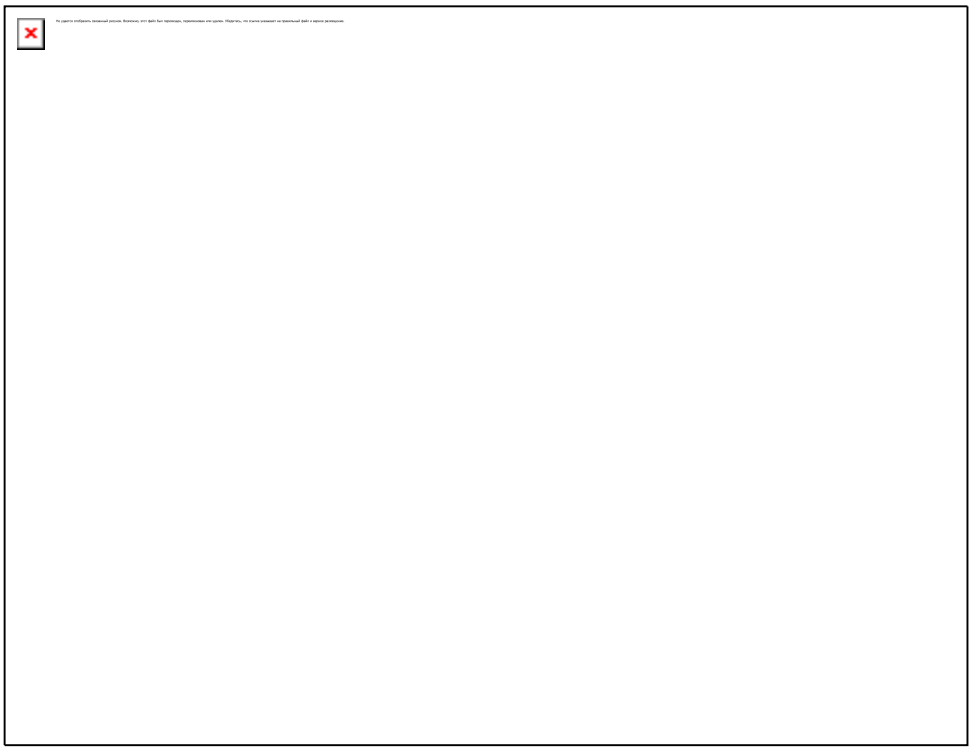


Рисунок 2.7. Текст нового модуля в редакторе кода

3. Сохраните модуль под именем MathLib, выбрав в меню команду **File | Save** (рисунок 2.8):

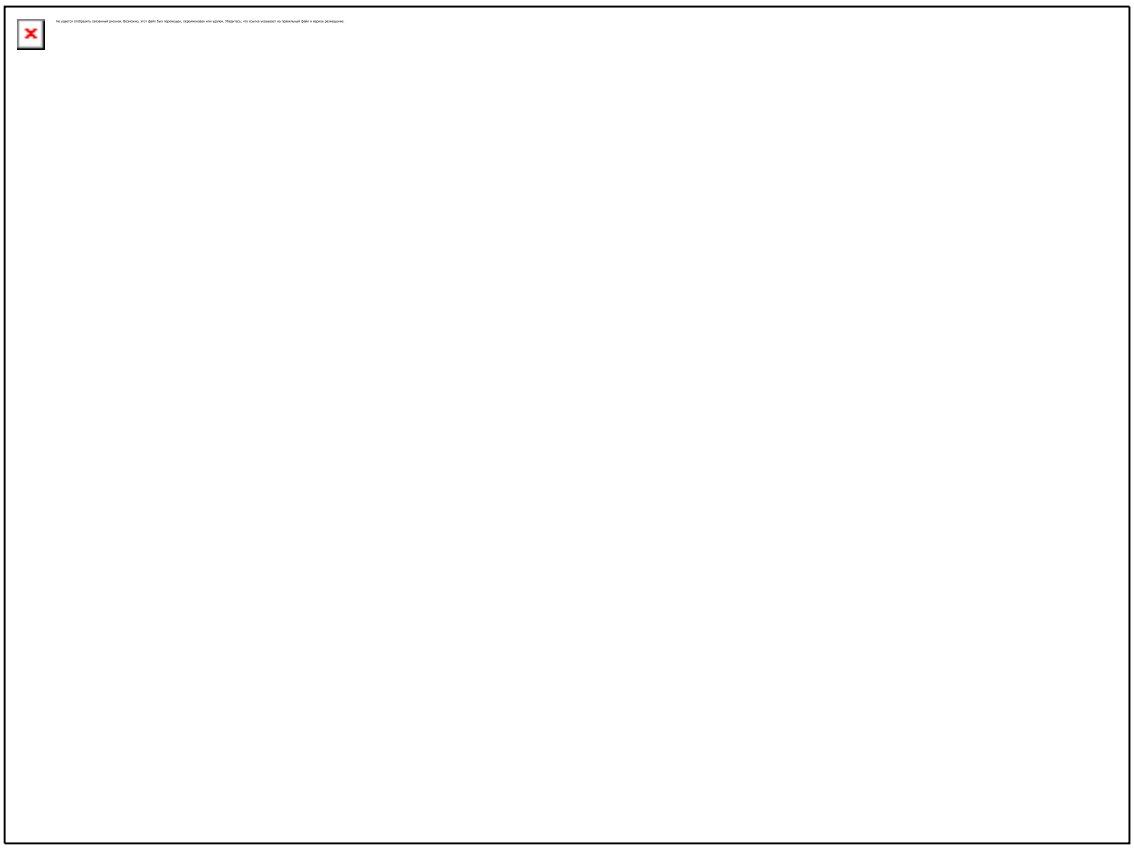


Рисунок 2.8. Окно сохранения модуля

4. Заметьте, что основная программа Console изменилась: в списке подключаемых модулей появилось имя модуля MathLib (рисунок 2.9). После слова **in** среда Delphi автоматически помещает имя файла, в котором находится модуль. Для стандартных модулей, таких как SysUtils, это не нужно, поскольку их местонахождение хорошо известно.

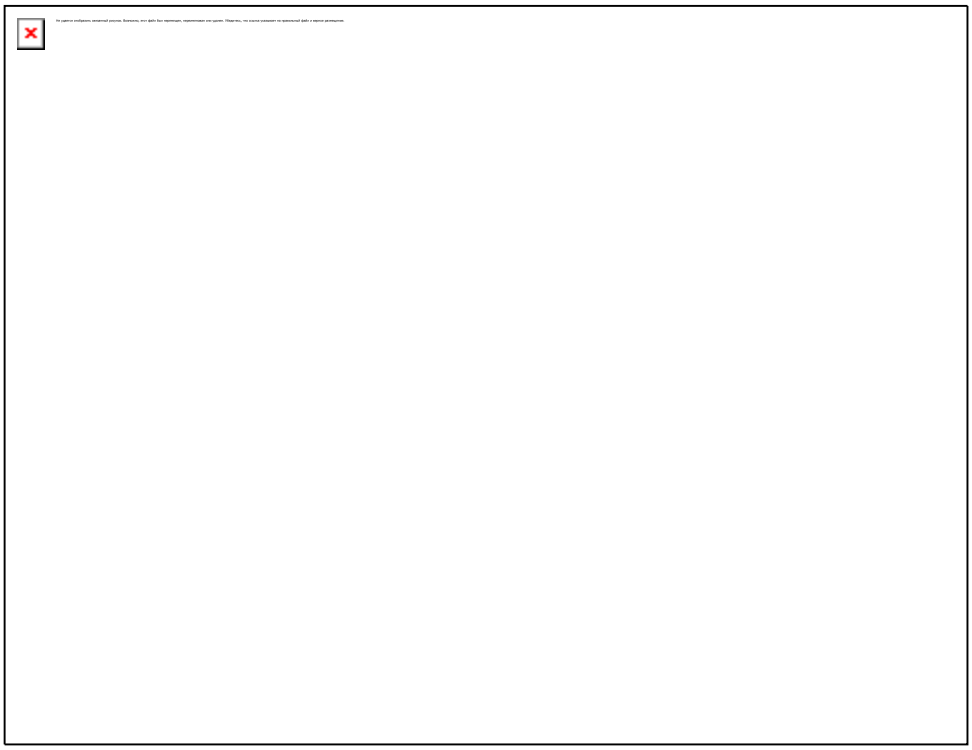


Рисунок 2.9. Текст программы Console в окне редактора

Теперь перейдем к содержимому модуля. Давайте объявим в нем константу Pi и две функции: Power — вычисление степени числа, и Average — вычисление среднего арифметического двух чисел:

```
unit MathLib;

interface

const
  Pi = 3.14;

function Power(X, Y: Double): Double;
function Average(X, Y: Double): Double;

implementation

function Power(X, Y: Double): Double;
begin
  Result := Exp(Y * Ln(X));
end;

function Average(X, Y: Double): Double;
begin
  Result := (X + Y) / 2;
end;

end.
```

Вот как могла бы выглядеть программа, использующая модуль Math:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  MathLib in 'MathLib.Pas';

begin
  Writeln(Pi);
  Writeln(Power(2, 4));
  Writeln(Average(2, 4));
  Writeln('Press Enter to exit...');
  Readln;
end.
```

После компиляции и запуска программы вы увидите на экране три числа (рисунок 2.10):

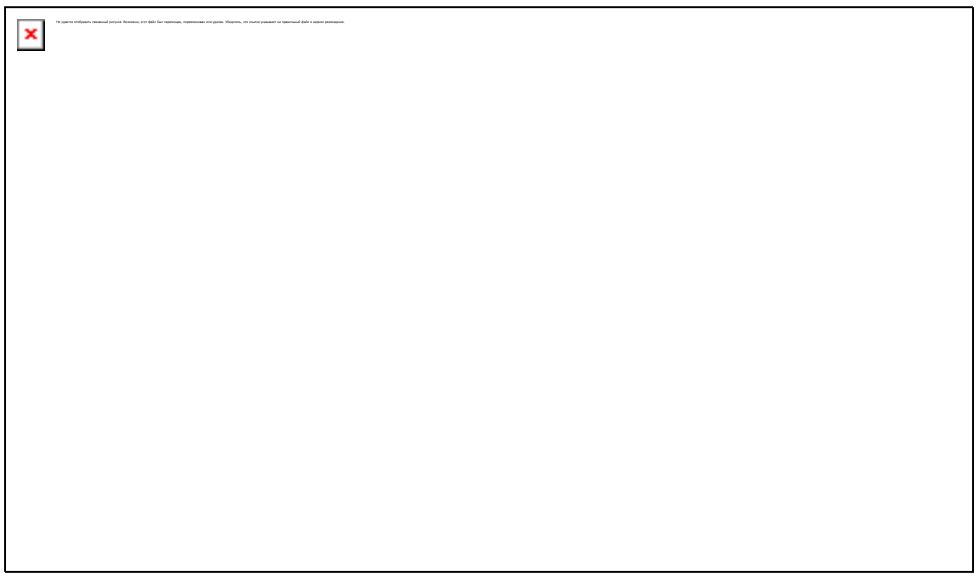


Рисунок 2.10. Результат работы программы Console

2.9.2. Стандартные модули языка Delphi

В состав среды Delphi входит великолепный набор модулей, возможности которых удовлетворят даже самого привередливого программиста. Все модули можно разбить на две группы: системные модули и модули визуальных компонентов.

К *системным модулям* относятся System, SysUtils, ShareMem, Math. В них содержатся наиболее часто используемые в программах типы данных, константы, переменные, процедуры и функции. Модуль System — это сердце среды Delphi; содержащиеся в нем подпрограммы обеспечивают работу всех остальных модулей системы. Модуль System подсоединяется автоматически к каждой программе и его не надо указывать в операторе **uses**.

Модули визуальных компонентов (VCL — Visual Component Library) используются для визуальной разработки полнофункциональных GUI-приложений — приложений с графическим пользовательским интерфейсом (Graphical User Interface). Эти модули в совокупности представляют собой высокоуровневую объектно-ориентированную библиотеку со всевозможными элементами пользовательского интерфейса: кнопками, надписями, меню, панелями и т.д. Кроме того, модули этой библиотеки содержат простые и эффективные средства доступа к базам данных. Данные модули подключаются автоматически при помещении компонентов на форму, поэтому вам об этом заботиться не надо. Их список слишком велик, поэтому мы его не приводим.

Все основные модули среды Delphi, включая модули визуальных компонентов, поставляются вместе с их исходными текстами на языке Delphi. По мере роста вашего профессионального опыта мы рекомендуем чаще обращаться к этим исходным текстам. Во-первых, в них вы найдете ответы на многие вопросы о внутреннем устройстве среды Delphi, а во-вторых, они послужат образцовым примером профессионального подхода в решении широкого круга задач. И, в-третьих, что не менее важно, это поможет научиться красиво и правильно (в рамках устоявшегося стиля) оформлять тексты Ваших собственных программ так, чтобы их с легкостью читали и понимали другие программисты.

Исходные тексты стандартных модулей среды Delphi находятся в каталоге Delphi/Source.

2.9.3. Область действия идентификаторов

При программировании необходимо соблюдать ряд правил, регламентирующих использование идентификаторов:

- каждый идентификатор должен быть описан перед тем, как он будет использован;
- областью действия идентификатора является блок, в котором он описан;
- все идентификаторы в блоке должны быть уникальными, т.е. не повторяться;
- один и тот же идентификатор может быть по-разному определен в каждом отдельном блоке, при этом блоки могут быть вложенными;
- если один и тот же идентификатор определен в нескольких вложенных блоках, то в пределах вложенного блока действует вложенное описание;
- все глобальные описания подключенного модуля видны программе (подключающему модулю), как если бы они были сделаны в точке подключения;
- если подключаются несколько модулей, в которых по-разному определен один и тот же идентификатор, то определение, сделанное в последнем подключенном модуле перекрывает все остальные;

- если один и тот же идентификатор определен и в подключенном модуле, и в программе (подключающем модуле), то первый игнорируется, а используется идентификатор, определенный в программе (подключающем модуле). Доступ к идентификатору подключенного модуля возможен с помощью уточненного имени. *Уточненное имя* формируется из имени модуля и записанного через точку идентификатора. Например, чтобы в предыдущем примере получить доступ к стандартному значению числа π, нужно записать System.Pi.

2.10. Строки

2.10.1. Строковые значения

Строка — это последовательность символов. При программировании строковые значения заключаются в апострофы, например:

```
Writeln('Я тебя люблю');
```

Так как апостроф является служебным символом, для его записи в строке как значащего символа применяются два апострофа, следующих непосредственно друг за другом:

```
Writeln('Object Pascal is Delphi''s and Kylix''s language');
```

Для записи отсутствующих на клавиатуре символов используется символ #, за которым следует десятичный номер символа в кодовой таблице ASCII, например:

```
Writeln('Copyright '#169' А.Вальвачев, К.Сурков, Д.Сурков, Ю.Четырько');
```

Строка, которая не содержит символов, называется *пустой*:

```
Writeln('');
```

Теперь, когда известно, что представляют собой строковые значения, займемся строковыми переменными.

2.10.2. Строковые переменные

Строковая переменная объявляется с помощью зарезервированного слова string или с помощью идентификатора типа данных AnsiString, например:

```
var
  FileName: string;
  EditText: AnsiString;
```

Строку можно считать бесконечной, хотя на самом деле ее длина ограничена 2 ГБ. В зависимости от присваиваемого значения строка увеличивается и сокращается динамически. Это удобство обеспечивается тем, что физически строковая переменная хранит не сами символы, а адрес символов строки в области динамически распределяемой памяти (о динамически распределяемой памяти мы расскажем ниже). При создании строки всегда инициализируются пустым значением (''). Управление динамической памятью при операциях со строками выполняется автоматически с помощью стандартных библиотек языка Delphi.

Вы конечно же можете описывать *строковые типы данных* и использовать их при объявлении переменных и типизированных констант, например:

```
type
  TName = string;
var
  Name: TName;
const
  FriendName: TName = 'Alexander';
```

Символы строки индексируются от 1 до N+1, где N — реальная длина строки. Символ с индексом N+1 всегда равен нулю (#0). Для получения длины следует использовать функцию **Length**, а для изменения длины — процедуру **SetLength** (см. ниже).

Для того чтобы в программе обратиться к отдельному символу строки, нужно сразу за идентификатором строковой переменной или константы в квадратных скобках записать его номер. Например, FriendName[1] возвращает значение ‘A’, а FriendName[4] — ‘x’. Символы, получаемые в результате индексирования строки, принадлежат типу Char.

Достоинство строки языка Delphi состоит в том, что она объединяет в себе свойства строки самого языка Delphi и строки языка C. Оперируя строкой, вы оперируете значением строки, а не адресом в оперативной памяти. В то же время строка не ограничена по длине и может передаваться вместо C-строки (как адрес первого символа строки) в параметрах процедур и функций. Чтобы компилятор позволил это сделать, нужно, записывая строку в качестве параметра, преобразовать ее к типу PChar (тип данных, используемый в языке Delphi для описания нуль-терминированных строк языка C). Такое приведение типа допустимо по той причине, что строка всегда завершается нулевым символом (#0), который хоть и не является ее частью, тем не менее всегда дописывается сразу за последним символом строки. В результате формат строки удовлетворяет формату C-строки. О работе с нуль-терминированными строками мы поговорим чуть позже.

2.10.3. Строки в формате Unicode

Для поддержки работы со строками формата Unicode в язык Delphi имеется строковый тип данных WideString. Работа со строками типа WideString почти не отличается от работы со строками типа AnsiString; существуют лишь два отличия.

Первое отличие состоит в представлении символов. В строках типа WideString каждый символ кодируется не одним байтом, а двумя. Соответственно элементы строки WideString — это символы типа WideChar, тогда как элементы строки AnsiString — это символы типа AnsiChar.

Второе отличие состоит в том, что происходит при присваивании строковых переменных. Об этом вы узнаете чуть позже, прочитав параграф "Представление строк в памяти".

2.10.4. Короткие строки

Короткая строка объявляется с помощью идентификатора типа ShortString или зарезервированного слова **string**, за которым следует заключенное в квадратные скобки значение максимально допустимой длины, например:

```
var
  Address: ShortString;
  Person: string[30];
```

Короткая строка может иметь длину от 1 до 255 символов. Предопределенный тип данных ShortString эквивалентен объявлению **string**[255].

Реальная длина строки может быть меньше или равна той, что указана при ее объявлении. Например, максимальная длина строки Friend в примере выше составляет 30 символов, а ее реальная длина — 9 символов. Реальную длину строки можно узнать с помощью встроенной функции **Length**. Например, значение Length(Friend) будет равно 9 (количество букв в слове Alexander).

Все символы в строке типа ShortString пронумерованы от 0 до N, где N — максимальная длина, указанная при объявлении. Символ с номером 0 — это служебный байт, в нем содержится реальная длина короткой строки. Значащие символы нумеруются от 1. Очевидно, что в памяти строка занимает на 1 байт больше, чем ее максимальная длина. Поэтому значение SizeOf(Friend) будет равно 31.

```
type
  TName = string[30];
var
  Name: TName;
const
  FriendName: TName = 'Alexander';
```

Обратиться к отдельному символу можно так же, как и к символу обычной строки. Например, выражения FriendName[1] и FriendName[9] возвращают соответственно символы 'A' и 'r'. Значения FriendName[10] .. FriendName[30] будут случайными, так как при объявлении типизированной константы FriendName символы с номерами от 10 до 30 не были инициализированы. Символы, получаемые в результате индексирования короткой строки, принадлежат типу Char.

Поскольку существует два типа строк: обычные (длинные) строки и короткие строки, возникает закономерный вопрос, можно ли их совмещать. Да, можно! Короткие и длинные строки могут одновременно использоваться в одном выражении, поскольку компилятор языка Delphi автоматически генерирует код, преобразующий их тип. Более того, можно выполнять явные преобразования строк с помощью конструкций вида ShortString(S) и AnsiString(S).

2.10.5. Операции над строками

Выражения, в которых операндами служат строковые данные, называются *строковыми*. Они состоят из строковых констант, переменных, имен функций и строковых операций. Над строковыми данными допустимы операции сцепления и отношения.

Операция сцепления (+) применяется для сцепления нескольких строк в одну строку.

Выражение	*Результат*
'Object' + ' Pascal'	'Object Pascal'

Операции отношения (=, <>, >, <, >=, <=) проводят сравнение двух строковых операндов. Сравнение строк производится слева направо до первого несовпадающего символа, и та строка считается больше, в которой первый несовпадающий символ имеет больший номер в кодовой таблице. Строки считаются равными, если они полностью совпадают по длине и содержат одни и те же символы. Если строки имеют различную длину, но в общей части символы совпадают, считается, что более короткая строка меньше, чем более длинная.

Выражение	*Результат*
'USA' < 'USIS'	True { A < I }
'abcde' > 'ABCDE'	True
'Office' = 'Office'	True
'USIS' > 'US'	True

Если короткой строке присваивается значение, длина которого превышает максимально допустимую величину, то все лишние символы справа отбрасываются.

Объявление строки	*Выражение*	*Значение строки*
Name: string[6];	Name := 'Mark Twain';	'Mark T'

Допускается смешение в одном выражении операндов строкового и символьного типа, например при сцеплении строки и символа.

2.10.6. Строковые ресурсы

В языке Delphi существует специальный вид строковых данных — строковые ресурсы. *Строковые ресурсы* очень похожи на строковые константы, но отличаются от них тем, что размещаются не в области данных программы, а в специальной области выполняемого файла, называемой ресурсами. Если данные всегда загружаются вместе с кодом программы и остаются в оперативной памяти вплоть до завершения программы, то ресурсы подгружаются в оперативную память лишь по мере надобности.

В программе строковые ресурсы описываются как обычные строковые константы, с той лишь разницей что раздел их описания начинается не словом **const**, а словом **resourcestring**:

```
resourcestring
  SCreateFileError = 'Cannot create file: ';
  SOpenFileError = 'Cannot open file: ';
```

Использование строковых ресурсов ничем не отличается от использования строковых констант:

```
var
  S: string;
begin
  S := SCreateFileError + 'MyFile.txt';
  ...
end;
```

На роль строковых ресурсов отлично подходят сообщения об ошибках, которые занимают много места в памяти и остаются не нужны до тех пор, пока в программе не возникнет ошибка. Использование ресурсов упрощает перевод пользовательского интерфейса на другие языки, поскольку замена текстовых сообщений может производиться непосредственно в выполняемом файле, т.е. без перекомпиляции программы.

2.10.7. Форматы кодирования символов

Существуют различные форматы кодирования символов. Отдельный символ строки может быть представлен в памяти одним байтом (стандарт Ansi), двумя байтам (стандарт Unicode) и даже четырьмя байтами (стандарт UCS-4 — Unicode). Строка “Wirth” (фамилия автора языка Pascal — прародителя языка Delphi) будет представлена в указанных форматах следующим образом (рисунок 2.11):

Рисунок 2.11. Форматы кодирования символов

Существует также формат кодирования MBCS (Multibyte Character Set), согласно которому символы одной строки кодируются разным количеством байт (одним или двумя байтами в зависимости от алфавита). Например, буквы латинского алфавита кодируются одним байтом, а иероглифы японского алфавита — двумя. При этом латинские буквы и японские иероглифы могут встречаться в одной и той же строке.

2.10.8. Стандартные процедуры и функции для работы со строками

Так как обработка строк выполняется практически в каждой серьезной программе, стандартно подключаемый модуль System имеет набор процедур и функций, значительно

облегчающих этот процесс. Все следующие процедуры и функции применимы и к коротким, и к длинным строкам.

Concat(S1, S2, ... , Sn): string — возвращает строку, полученную в результате сцепления строк S1, S2, ..., Sn. По своей работе функция Concat аналогична операции сцепления (+).

Copy(S: string, Index, Count: Integer): string — выделяет из строки S подстроку длиной Count символов, начиная с позиции Index.

Delete(var S: string, Index, Count: Integer) — удаляет Count символов из строки S, начиная с позиции Index.

Insert(Source: string; var S: string, Index: Integer) — вставляет строку Source в строку S, начиная с позиции Index.

Length(S: string): Integer — возвращает реальную длину строки S в символах.

SetLength(var S: string; NewLength: Integer) — устанавливает для строки S новую длину NewLength.

Примеры:

Выражение	*Значение S*
S := Concat('Object ', 'Pascal');	'Object Pascal'
S:= Copy('Debugger', 3, 3);	'bug'
S := 'Compile'; Delete(S, 1, 3);	'pile'
S := 'Faction'; Insert('r', S, 2)	'Fraction'

Pos(Substr, S: string): Byte — обнаруживает первое появление подстроки Substr в строке S. Возвращает номер той позиции, где находится первый символ подстроки Substr. Если в S подстроки Substr не найдено, результат равен 0.

Выражение	*Результат*
Pos('rat', 'grated’)	2
Pos('sh', 'champagne')	0

Str(X [: Width [: Decimals]], var S: string) — преобразует числовое значение величины X в строку S. Необязательные параметры Width и Decimals являются целочисленными выражениями. Значение Width задает ширину поля результирующей строки. Значение Decimals используется с вещественными числами и задает количество символов в дробной части.

Выражение	*Значение S*
Str(–200, S);	'–200'
Str(200 : 4, S);	' 200'
Str(1.5E+02 : 4, S);	' 150'

Val(S: string, var V; var Code: Integer) — преобразует строку S в величину целого или вещественного типа и помещает результат в переменную V. Если во время операции

преобразования ошибки не обнаружено, значение переменной Code равно нулю; если ошибка обнаружена (строка содержит недопустимые символы), Code содержит номер позиции первого ошибочного символа, а значение V не определено.

Выражение	*Значение V*	*Значение Code*
Val('100', V, Code);	100	0
Val('2.5E+01', V, Code);	25.0	0
Val('2.5A+01', V, Code);	<не определено>	4

Описанные процедуры и функции являются базовыми для всех остальных подпрограмм обработки строк из модуля SysUtils.

AdjustLineBreaks(const S: string): string — возвращает копию строки S, в которой все мягкие переносы строк (одиночные символы #13 или #10) заменены жесткими переносами строк (последовательность символов #13#10).

AnsiCompareStr(const S1, S2: string): Integer — сравнивает две строки, делая различие между заглавными и строчными буквами; учитывает местный язык. Возвращаемое значение меньше нуля, если S1 < S2, равно нулю, если S1 = S2, и больше нуля, если S1 > S2.

AnsiCompareText(const S1, S2: string): Integer — сравнивает две строки, не делая различий между заглавными и строчными буквами; учитывает местный язык. Возвращаемое значение меньше нуля, если S1 < S2, равно нулю, если S1 = S2, и больше нуля, если S1 > S2.

AnsiDequotedStr(const S: string; Quote: Char): string — удаляет специальный символ, заданный параметром Quote, из начала и конца строки и заменяет парные спецсимволы на одиночные; если специальный символ отсутствует в начале или конце строки, то функция возвращает исходную строку без изменений.

AnsiExtractQuotedStr(var Src: PChar; Quote: Char): string — делает то же, что и функция AnsiDequotedStr, но результат возвращается вместо исходной строки, которая имеет тип PChar.

AnsiLowerCase(const S: string): string — преобразует заглавные буквы строки S к строчным буквам с учетом местного языка.

AnsiPos(const Substr, S: string): Integer — выполняет те же действия, что и функция Pos, но в отличие от нее поддерживает работу с многобайтовой MBCS-кодировкой.

AnsiQuotedStr(const S: string; Quote: Char): string — преобразует строку, заменяя все вхождения специального символа, заданного параметром Quote, на парные спецсимволы, а также помещает специальный символ в начало и конец строки. Поддерживает работу с MBCS-кодировкой.

AnsiSameCaption(const Text1, Text2: string): Boolean — сравнивает две строки, не делая различие между заглавными и строчными буквами, а также не учитывая символ ‘&’; учитывает местный язык.

AnsiSameStr(const S1, S2: string): Boolean — сравнивает строки, делая различие между строчными и заглавными буквами; учитывает местный язык.

AnsiSameText(const S1, S2: string): Boolean — сравнивает строки, не делая различие между строчными и заглавными буквами; учитывает местный язык.

AnsiUpperCase(const S: string): string — преобразует все строчные буквы в заглавные; учитывает местный язык.

CompareStr(const S1, S2: string): Integer — выполняет сравнение двух строк, делая различие между строчными и заглавными буквами; не учитывает местный язык. Возвращаемое значение меньше нуля, если S1 < S2, равно нулю, если S1 = S2, и больше нуля, если S1 > S2.

CompareText(const S1, S2: string): Integer — выполняет сравнение двух строк, не делая различий между строчными и заглавными буквами; не учитывает местный язык. Возвращаемое значение меньше нуля, если S1 < S2, равно нулю, если S1 = S2, и больше нуля, если S1 > S2.

DateTimeToStr(const DateTime: TDateTime): string — преобразует значение даты и времени в строку.

DateTimeToString(var Result: string; const Format: string; DateTime: TDateTime) — преобразует значение даты и времени в строку, выполняя при этом форматирование в соответствии со значением строки Format. Управляющие символы строки Format подробно описаны в справочнике по среде Delphi.

DateToStr(const DateTime: TDateTime): string — преобразует числовое значение даты в строку.

Format(const Format: string; const Args: array of const): string — форматирует строку в соответствии с шаблоном Format, заменяя управляющие символы шаблона на значения элементов открытого массива Args. Управляющие символы подробно описаны в справочнике по среде Delphi.

FormatDateTime(const Format: string; DateTime: TDateTime): string — преобразует значение даты и времени в строку, выполняя при этом форматирование в соответствии со значением строки Format. Управляющие символы строки Format подробно описаны в справочнике по среде Delphi.

BoolToStr(B: Boolean; UseBoolStrs: Boolean = False): string — преобразует булевское значение в строку. Если параметр UseBoolStrs имеет значение False, то результатом работы функции является одно из значений '0' или ‘–1’. Если же параметр UseBoolStrs имеет значение True, то результатом работы является одно из значений ‘FALSE’ или ‘TRUE’ (программист может задать другие значения; о том, как это сделать, читайте в справочнике по системе Delphi).

IntToHex(Value: Integer; Digits: Integer): string — возвращает шестнадцатиричное представление целого числа Value. Параметр Digits задает количество цифр результирующей строки.

IntToStr(Value: Integer): string — преобразует целое число Value в строку.

IsDelimiter(const Delimiters, S: string; Index: Integer): Boolean — проверяет, является ли символ S[Index] одним из символов строки Delimiters. Функция поддерживает работу с многобайтовой MBCS-кодировкой.

IsValidIdent(const Ident: string): Boolean — возвращает True, если строка Ident является правильным идентификатором языка Delphi.

LastDelimiter(const Delimiters, S: string): Integer — возвращает индекс последнего вхождения одного из символов строки Delimiters в строку S.

LowerCase(const S: string): string — преобразует все заглавные буквы строки S к строчным; не учитывает местный язык (в преобразовании участвуют лишь символы в диапазоне от 'A' до 'Z').

QuotedStr(const S: string): string — преобразует исходную строку в строку, взятую в одиночные кавычки; внутри строки символы кавычки дублируются.

SameText(const S1, S2: string): Boolean — сравнивает строки, не делая различие между строчными и заглавными буквами; учитывает местный язык.

SetString(var S: string; Buffer: PChar; Len: Integer) — копирует строку с типом PChar в строку с типом string. Длина копируемой строки задается параметром Len.

StringOfChar(Ch: Char; Count: Integer): string — возвращает строку, в которой повторяется один и тот же символ. Количество повторений задается параметром Count.

StringToGUID(const S: string): TGUID — преобразует строковое представление глобального уникального идентификатора в стандартный тип TGUID.

StrToBool(const S: string): Boolean — преобразует строку в булевское значение.

StrToBoolDef(const S: string; const Default: Boolean): Boolean — преобразует строку в булевское значение. В случае невозможности преобразования, функция возвращает значение, переданное через параметр Default.

StrToDate(const S: string): TDateTime — преобразует строку со значением даты в числовой формат даты и времени.

StrToDateDef(const S: string; const Default: TDateTime): TDateTime — преобразует строку со значением даты в числовой формат даты и времени. В случае невозможности преобразования, функция возвращает значение, переданное через параметр Default.

StrToDateTime(const S: string): TDateTime — преобразует строку в числовое значение даты и времени.

StrToDateTimeDef(const S: string; const Default: TDateTime): TDateTime — преобразует строку в числовое значение даты и времени. В случае невозможности преобразования, функция возвращает значение, переданное через параметр Default.

StrToInt(const S: string): Integer — преобразует строку в целое число. Если строка не может быть преобразована в целое число, функция генерирует исключительную ситуацию класса EConvertError (обработка исключительных ситуаций рассматривается в главе 4).

StrToIntDef(const S: string; Default: Integer): Integer — преобразует строку в целое число. Если строка не может быть преобразована в целое число, функция возвращает значение, заданное параметром Default.

StrToInt64(const S: string): Int64 — 64-битный аналог функции StrToInt — преобразует строку в 64-битное целое число. Если строка не может быть преобразована в 64-битное число, функция генерирует исключительную ситуацию класса EConvertError (обработка исключительных ситуаций рассматривается в главе 4).

StrToInt64Def(const S: string; const Default: Int64): Int64 — 64-битный аналог функции StrToIntDef — преобразует строку в 64-битное целое число. Если строка не может быть преобразована в 64-битное число, функция возвращает значение, заданное параметром Default.

StrToTime(const S: string): TDateTime — преобразует строку в числовой формат времени. Если строка не может быть преобразована в числовой формат времени, функция генерирует исключительную ситуацию класса EConvertError (обработка исключительных ситуаций рассматривается в главе 4).

StrToTimeDef(const S: string; const Default: TDateTime): TDateTime — преобразует строку в числовой формат времени. В случае ошибки преобразования, функция возвращает значение, заданное параметром Default.

TimeToStr(Time: TDateTime): string — преобразует числовое значение времени в строку.

Trim(const S: string): string — возвращает часть строки S без лидирующих и завершающих пробелов и управляющих символов.

Trim(const S: WideString): WideString — Unicode-аналог функции Trim — возвращает часть строки S без лидирующих и завершающих пробелов и управляющих символов.

TrimLeft(const S: string): string — возвращает часть строки S без лидирующих пробелов и управляющих символов.

TrimLeft(const S: WideString): WideString — Unicode-аналог функции TrimLeft — возвращает часть строки S без лидирующих пробелов и управляющих символов.

TrimRight(const S: string): string — возвращает часть строки S без завершающих пробелов и управляющих символов.

TrimRight(const S: WideString): WideString — Unicode-аналог функции TrimRight — возвращает часть строки S без завершающих пробелов и управляющих символов.

UpperCase(const S: string): string — преобразует все строчные буквы строки S в заглавные; не учитывает местный язык (в преобразовании участвуют лишь символы в диапазоне от 'a' до 'z').

WideFormat(const Format: WideString; const Args: array of const): WideString — Unicode-аналог функции Format, учитывающий символы местного языка, — форматирует строку в соответствии с шаблоном Format, заменяя управляющие символы в шаблоне на значения элементов открытого массива Args. Управляющие символы подробно описаны в справочнике по системе Delphi.

WideFmtStr(var Result: WideString; const Format: WideString; const Args: array of const) — аналог функции WideFormat. Отличие в том, что WideFmtStr возвращает результат через параметр Result, а не как значение функции.

WideLowerCase(const S: WideString): WideString — Unicode-аналог функции LowerCase (учитывает местный язык) — преобразует все заглавные буквы строки S к строчным буквам.

WideSameCaption(const Text1, Text2: WideString): Boolean — Unicode-аналог функции AnsiSameCaption — сравнивает две строки, не делая различие между строчными и заглавными буквами, а также не учитывая символ ‘&’; учитывает местный язык.

WideSameStr(const S1, S2: WideString): Boolean — Unicode-аналог стандартной операции сравнения строк — сравнивает две строки, делая различие между строчными и заглавными буквами.

WideSameText(const S1, S2: WideString): Boolean — Unicode-аналог функции SameText (учитывает местный язык) — сравнивает строки, не делая различие между строчными и заглавными буквами.

WideUpperCase(const S: WideString): WideString — Unicode-аналог функции UpperCase (учитывает местный язык) — преобразует все строчные буквы строки S в заглавные.

WrapText(const Line: string; MaxCol: Integer = 45): string — разбивает текст Line на строки, вставляя символы переноса строки. Максимальная длина отдельной строки задается параметром MaxCol.

WrapText(const Line, BreakStr: string; const BreakChars: TSysCharSet; MaxCol: Integer): string — более мощный аналог предыдущей функции — разбивает текст Line на строки, вставляя символы переноса строки.

AnsiToUtf8(const S: string): UTF8String — перекодирует строку в формат UTF8.

PUCS4Chars(const S: UCS4String): PUCS4Char — возвращает указатель на первый символ строки формата UCS-4 для работы со строкой, как с последовательностью символов, заканчивающейся символом с кодом нуль.

StringToWideChar(const Source: string; Dest: PWideChar; DestSize: Integer): PWideChar — преобразует стандартную строку к последовательности Unicode-символов, завершающейся символом с кодом нуль.

UCS4StringToWideString(const S: UCS4String): WideString — преобразует строку формата UCS-4 к строке формата Unicode.

Utf8Decode(const S: UTF8String): WideString — преобразует строку формата UTF-8 к строке формата Unicode.

Utf8Encode(const WS: WideString): UTF8String — преобразует строку формата Unicode к строке формата UTF-8.

Utf8ToAnsi(const S: UTF8String): string — преобразует строку формата UTF-8 к стандратной строке.

WideCharLenToString(Source: PWideChar; SourceLen: Integer): string — преобразует строку формата Unicode к стандартной строке. Длина исходной строки задается параметром SourceLen.

WideCharLenToStrVar(Source: PWideChar; SourceLen: Integer; var Dest: string) — аналог предыдущей функции — преобразует строку формата Unicode к стандартной строке. Длина исходной строки задается параметром SourceLen, а результат возвращается через параметр Dest.

WideCharToString(Source: PWideChar): string — преобразует последовательность Unicode-символов, завершающуюся символом с кодом нуль, к стандартной строке.

WideCharToStrVar(Source: PWideChar; var Dest: string) — аналог предыдущей функции — преобразует последовательность Unicode-символов, завершающуюся символом с кодом нуль, к стандартной строке. Результат возвращается через параметр Dest.

WideStringToUCS4String(const S: WideString): UCS4String — преобразует строку формата Unicode к строке формата UCS-4.

2.11. Массивы

2.11.1. Объявление массива

Массив — это составной тип данных, состоящий из фиксированного числа элементов одного и того же типа. Для описания массива предназначено словосочетание **array of**. После слова **array** в квадратных скобках записываются границы массива, а после слова **of** — тип элементов массива, например:

```
type
  TStates = array[1..50] of string;
  TCoordinates = array[1..3] of Integer;
```

После описания типа можно переходить к определению переменных и типизированных констант:

```
var
  States: TStates; { 50 strings }
const
  Coordinates: TCoordinates = (10, 20, 5); { 3 integers }
```

Обратите внимание, что инициализация элементов массива происходит в круглых скобках через запятую.

Массив может быть определен и без описания типа:

```
var
  Symbols: array[0..80] of Char; { 81 characters }
```

Чтобы получить доступ к отдельному элементу массива, нужно в квадратных скобках указать его индекс, например

```
Symbols[0]
```

Объявленные выше массивы являются *одномерными*, так как имеют только один индекс. Одномерные массивы обычно используются для представления линейной последовательности элементов. Если при описании массива задано два индекса, массив называется *двумерным*, если n индексов — *n-мерным*. Двумерные массивы используются для представления таблицы, а n-мерные — для представления пространств. Вот пример объявления таблицы, состоящей из 5 колонок и 20 строк:

```
var
  Table: array[1..5] of array[1..20] of Double;
```

То же самое можно записать в более компактном виде:

```
var
  Table: array[1..5, 1..20] of Double;
```

Чтобы получить доступ к отдельному элементу многомерного массива, нужно указать значение каждого индекса, например

```
Table[2][10]
```

или в более компактной записи

```
Table[2, 10]
```

Эти два способа индексации эквивалентны.

2.11.2. Работа с массивами

Массивы в целом участвуют только в операциях присваивания. При этом все элементы одного массива копируются в другой. Например, если объявлены два массива A и B,

```
var
  A, B: array[1..10] of Integer;
```

то допустим следующий оператор:

```
A := B;
```

Оба массива-операнда в левой и правой части оператора присваивания должны быть не просто идентичны по структуре, а описаны с одним и тем же типом, иначе компилятор сообщит об ошибке. Именно поэтому все массивы рекомендуется описывать в секции **type**.

С элементами массива можно работать, как с обычными переменными. В следующей программе элементы численного массива последовательно вводятся с клавиатуры, а затем суммируются. Результат выводится на экран.

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  A: array[1..5] of Double;
  Sum: Double;
  I: Integer;

begin
  for I := 1 to 5 do Readln(A[I]);
  Sum := 0;
  for I := 1 to 5 do Sum := Sum + A[I];
  Writeln(Sum);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Для массивов определены две встроенные функции — Low и High. Они получают в качестве своего аргумента имя массива. Функция Low возвращает нижнюю, а High — верхнюю границу этого массива. Например, Low(A) вернет значение 1, а High(A) — 5. Функции Low и High чаще всего используются для указания начального и конечного значений в операторе цикла **for**. Поэтому вычисление суммы элементов массива A лучше переписать так:

```
for I := Low(A) to High(A) do Sum := Sum + A[I];
```

В операциях с многомерными массивами циклы **for** вкладываются друг в друга. Например, для инициализации элементов таблицы, объявленной как

```
var
  Table: array[1..5, 1..20] of Double;
```

требуются два вложенных цикла **for** и две целые переменные Col и Row для параметров этих циклов:

```
for Col := 1 to 5 do
  for Row := 1 to 20 do
    Table[Col, Row] := 0;
```

2.11.3. Массивы в параметрах процедур и функций

Массивы, как и другие типы данных, могут выступать в качестве параметров процедур и функций. Вот как может выглядеть функция, вычисляющая среднее значение в массиве действительных чисел:

```
const
  Max = 63;
type
  TStatistics = array [0..Max] of Double;

function Average(const A: TStatistics): Double;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do Result := Result + A[I];
  Result := Result / (High(A) - Low(A) + 1);
end;
```

Функция Average принимает в качестве параметра массив известной размерности. Требование фиксированного размера для массива-параметра часто является чрезмерно сдерживающим фактором. Процедура для нахождения среднего значения должна быть способна работать с массивами произвольной длины. Для этой цели в язык Delphi введены открытые массивы-параметры. Такие массивы были заимствованы разработчиками языка

Delphi из языка Modula-2. Открытый массив-параметр описывается с помощью словосочетания **array of**, при этом границы массива опускаются:

```
function Average(const A: array of Double): Double;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do Result := Result + A[I];
  Result := Result / (High(A) - Low(A) + 1);
end;
```

Внутри подпрограммы Average нижняя граница открытого массива A равна нулю (Low(A) = 0), а вот значение верхней границы (High(A)) неизвестно и выясняется только на этапе выполнения программы.

Существует только два способа использования открытых массивов: обращение к элементам массива и передача массива другой подпрограмме, принимающей открытый массив. Нельзя присваивать один открытый массив другому, потому что их размеры заранее неизвестны.

Вот пример использования функции Average:

```
var
  Statistics: array[1..10] of Double;
  Mean: Double;
begin
  ...
  Mean := Average(Statistics);
  Mean := Average([0, Random, 1]);
  ...
end;
```

Заметьте, что во втором операторе открытый массив конструируется в момент вызова функции Average. *Конструктор открытого массива* представляет собой заключенный в квадратные скобки список выражений. В выражениях могут использоваться константы, переменные и функции. Тип выражений должен быть совместим с типом элементов массива. Конструирование открытого массива равносильно созданию и инициализации временной переменной.

И еще одно важное замечание по поводу открытых массивов. Некоторые библиотечные подпрограммы языка Delphi принимают параметры типа **array of const** — *открытые массивы констант*. Массив, передаваемый в качестве такого параметра, обязательно конструируется в момент вызова подпрограммы и может состоять из элементов различных типов (!). Физически он состоит из записей типа **TVarRec**, кодирующих тип и значение элементов массива (записи рассматриваются ниже). Открытый массив констант позволяет эмулировать подпрограммы с переменным количеством разнотипных параметров и используется, например, в функции Format для форматирования строки (см. выше).

2.11.4. Уплотнение структурных данных в памяти

С целью экономии памяти, занимаемой массивами и другими структурными данными, вы можете предварять описание типа зарезервированным словом **packed**, например:

```
var
  A: packed array[1..10] of Byte;
```

Ключевое слово **packed** указывает компилятору, что элементы структурного типа должны храниться плотно прижатыми друг к другу, даже если это замедляет к ним доступ. Если структурный тип данных описан без ключевого слова **packed**, компилятор выравнивает его элементы на 2- и 4-байтовых границах, чтобы ускорить к ним доступ.

Заметим, что ключевое слово **packed** применимо к любому структурному типу данных, т.е. массиву, множеству, записи, файлу, классу, ссылке на класс.

2.12. Множества

2.12.1. Объявление множества

Множество — это составной тип данных для представления набора некоторых элементов как единого целого. Область значений множества — набор всевозможных подмножеств, составленных из его элементов. Все элементы множества должны принадлежать однобайтовому порядковому типу. Этот тип называется *базовым типом множества*.

Для описания множественного типа используется словосочетание **set of**, после которого записывается базовый тип множества:

```
type
  TLetters = set of 'A'..'Z';
```

Теперь можно объявить переменную множественного типа:

```
var
  Letters: TLetters;
```

Можно объявить множество и без предварительного описания типа:

```
var
  Symbols: set of Char;
```

В выражениях значения элементов множества указываются в квадратных скобках: [2, 3, 5, 7], [1..9], ['A', 'B', 'C']. Если множество не имеет элементов, оно называется пустым и обозначается как [ ]. Пример инициализации множеств:

```
const
  Vowels: TLetters = ['A', 'E', 'I', 'O', 'U'];
begin
  Letters := ['A', 'B', 'C'];
  Symbols := [ ]; { пустое множество }
end;
```

Количество элементов множества называется *мощностью*. Мощность множества в языке Delphi не может превышать 256.

2.12.2. Операции над множествами

При работе с множествами допускается использование операций отношения (=, <>, >=, <=), объединения, пересечения, разности множеств и операции **in**.

Операции сравнения (=, <>). Два множества считаются равными, если они состоят из одних и тех же элементов. Порядок следования элементов в сравниваемых множествах значения не имеет. Два множества A и B считаются не равными, если они отличаются по мощности или по значению хотя бы одного элемента.

Выражение	*Результат*
[1, 2] <> [1, 2, 3]	True
[1, 2] = [1, 2, 2]	True
[1, 2, 3] = [3, 2, 1]	True
[1, 2, 3] = [1..3]	True

Операции принадлежности (>=, <=). Выражение A >= B равно True, если все элементы множества B содержатся в множестве A. Выражение A <= B равно True, если выполняется обратное условие, т.е. все элементы множества A содержатся в множестве B.

Выражение	*Результат*
[1, 2] <= [1, 2, 3]	True
[1, 2, 3] >= [1, 2]	True
[1, 2] <= [1, 3]	False

Операция **in**. Используется для проверки принадлежности элемента указанному множеству. Обычно применяется в условных операторах.

Выражение	*Результат*
5 in [1..9]	True
5 in [1..4, 6..9]	False

Операция **in** позволяет эффективно и наглядно выполнять сложные проверки условий, заменяя иногда десятки других операций. Например, оператор

```
if (X = 1) or (X = 2) or (X = 3) or (X = 5) or (X = 7) then
```

можно заменить более коротким:

```
if X in [1..3, 5, 7] then
```

Операцию **in** иногда пытаются записать с отрицанием: X **not in** S. Такая запись является ошибочной, так как две операции следуют подряд. Правильная запись имеет вид: **not** (X **in** S).

Объединение множеств (+). Объединением двух множеств является третье множество, содержащее элементы обоих множеств.

Выражение	*Результат*
[ ] + [1, 2]	[1, 2]
[1, 2] + [2, 3, 4]	[1, 2, 3, 4]

Пересечение множеств (*). Пересечение двух множеств — это третье множество, которое содержит элементы, входящие одновременно в оба множества.

Выражение	*Результат*
[ ] * [1, 2]	[ ]
[1, 2] * [2, 3, 4]	[2]

Разность множеств (–). Разностью двух множеств является третье множество, которое содержит элементы первого множества, не входящие во второе множество.

Выражение	*Результат*
[1, 2, 3] – [2, 3]	[1]
[1, 2, 3] – [ ]	[1, 2, 3]

В язык Delphi введены две стандартные процедуры Include и Exclude, которые предназначены для работы с множествами.

Процедура **Include**(S, I) включает в множество S элемент I. Она дублирует операцию + (плюс) с той лишь разницей, что при каждом обращении включает только один элемент и делает это более эффективно.

Процедура **Exclude**(S, I) исключает из множества S элемент I. Она дублирует операцию – (минус) с той лишь разницей, что при каждом обращении исключает только один элемент и делает это более эффективно.

Выражение	*Результат*
S := [1, 3];	[1, 3]
Include(S, 2);	[1, 2, 3]
Exclude(S, 3)	[1, 2]

Использование в программе множеств дает ряд преимуществ: значительно упрощаются сложные операторы **if**, улучшается наглядность программы и понимание алгоритма решения задачи, экономится время разработки программы. Поэтому множества широко используются в библиотеке компонентов среды Delphi.

2.13. Записи

2.13.1. Объявление записи

Запись — это составной тип данных, состоящий из фиксированного числа элементов одного или нескольких типов. Описание типа записи начинается словом **record** и заканчивается словом **end**. Между ними заключен список элементов, называемых *полями*, с указанием идентификаторов полей и типа каждого поля:

```
type
  TPerson = record
    FirstName: string[20]; // имя
    LastName: string[20];  // фамилия
    BirthYear: Integer;    // год рождения
  end;
```

Идентификаторы полей должны быть уникальными только в пределах записи. Допускается вложение записей друг в друга, т.е. поле записи может быть в свою очередь тоже записью.

Чтобы получить в программе реальную запись, нужно создать переменную соответствующего типа:

```
var
  Friend: TPerson;
```

Записи можно создавать и без предварительного описания типа, но это делается редко, так как мало отличается от описания полей в виде отдельных переменных.

Доступ к содержимому записи осуществляется посредством идентификаторов переменной и поля, разделенных точкой. Такая комбинация называется *составным именем*. Например, чтобы получить доступ к полям записи Friend, нужно записать:

```
Friend.FirstName := 'Alexander';
Friend.LastName := 'Ivanov';
Friend.BirthYear := 1991;
```

Обращение к полям записи имеет несколько громоздкий вид, что особенно неудобно при использовании мнемонических идентификаторов длиной более 5 символов. Для решения этой проблемы в языке Delphi предназначен оператор **with**, который имеет формат:

```
with <запись> do
  <оператор>;
```

Однажды указав имя записи в операторе **with**, можно работать с именами ее полей как с обычными переменными, т.е. без указания идентификатора записи перед идентификатором поля:

```
with Friend do
begin
  FirstName := 'Alexander';
  LastName := 'Ivanov';
  BirthYear := 1991;
end;
```

Допускается применение оператора присваивания и к записям в целом, если они имеют один и тот же тип. Например,

```
Friend := BestFriend;
```

После выполнения этого оператора значения полей записи Friend станут равными значениям соответствующих полей записи BestFriend.

2.13.2. Записи с вариантами

Строго фиксированная структура записи ограничивает возможность ее применения. Поэтому в языке Delphi имеется возможность задать для записи несколько вариантов структуры. Такие записи называются *записями с вариантами*. Они состоят из необязательной фиксированной и вариантной частей.

Вариантная часть напоминает условный оператор **case**. Между словами **case** и **of** записывается особое поле записи – *поле признака*. Оно определяет, какой из вариантов в данный момент будет активизирован. Поле признака должно быть равно одному из расположенных следом значений. Каждому значению сопоставляется вариант записи. Он заключается в круглые скобки и отделяется от своего значения двоеточием. Пример описания записи с вариантами:

```
type
  TFigure = record
    X, Y: Integer;
    case Kind: Integer of
      0: (Width, Height: Integer); // прямоугольник
      1: (Radius: Integer);        // окружность
end;
```

Обратите внимание, что у вариантной части нет отдельного **end**, как этого можно было бы ожидать по аналогии с оператором **case**. Одно слово **end** завершает и вариантную часть, и всю запись.

На этом мы пока закончим рассказ о записях, но хотим надеяться, что читатель уже догадался об их потенциальной пользе при организации данных с более сложной структурой.

2.14. Файлы

2.14.1. Понятие файла

С точки зрения пользователя файл — это именованная область данных на диске или любом другом внешнем носителе. В программе *файл* предстает как последовательность элементов некоторого типа. Так как размер одного файла может превышать объем всей оперативной памяти компьютера, доступ к его элементам выполняется *последовательно* с помощью процедур чтения и записи.

Для файла существует понятие *текущей позиции*. Она показывает номер элемента, который будет прочитан или записан при очередном обращении к файлу. Чтение-запись каждого

элемента продвигает текущую позицию на единицу вперед. Для большинства файлов можно менять текущую позицию чтения-записи, выполняя *прямой доступ* к его элементам.

В зависимости от типа элементов различают три вида файла:

- файл из элементов фиксированного размера; элементами такого файла чаще всего являются записи;
- файл из элементов переменного размера (*нетипизированный файл*); такой файл рассматривается просто как последовательность байтов;
- *текстовый файл*; элементами такого файла являются текстовые строки.

Для работы с файлом в программе объявляется *файловая переменная*. В файловой переменной запоминается имя файла, режим доступа (например, только чтение), другие атрибуты. В зависимости от вида файла файловая переменная описывается по-разному.

Для работы с файлом, состоящим из типовых элементов переменная объявляется с помощью словосочетания **file of**, после которого записывается тип элемента:

```
var
  F: file of TPerson;
```

К моменту такого объявления тип TPerson должен быть уже описан (см. выше).

Объявление переменной для работы с нетипизированным файлом выполняется с помощью отдельного слова **file**:

```
var
  F: file;
```

Для работы с текстовым файлом переменная описывается с типом TextFile:

```
var
  F: TextFile;
```

2.14.2. Работа с файлами

Наиболее часто приходится иметь дело с текстовым представлением информации, поэтому рассмотрим запись и чтение текстового файла.

Приступая к работе с файлом, нужно первым делом вызвать процедуру AssignFile, чтобы файловой переменной поставить в соответствие имя файла на диске:

```
AssignFile(F, 'MyFile.txt');
```

В результате этого действия поля файловой переменной F инициализируются начальными значениями. При этом в поле имени файла заносится строка 'MyFile.txt'.

Так как файла еще нет на диске, его нужно создать:

```
Rewrite(F);
```

Теперь запишем в файл несколько строк текста. Это делается с помощью хорошо вам знакомых процедур Write и Writeln:

```
Writeln(F, 'Pi = ', Pi);
Writeln(F, 'Exp = ', Exp(1));
```

При работе с файлами первый параметр этих процедур показывает, куда происходит вывод данных.

После работы файл должен быть закрыт:

```
CloseFile(F);
```

Рассмотрим теперь, как прочитать содержимое текстового файла. После инициализации файловой переменной (AssignFile) файл открывается с помощью процедуры Reset:

```
Reset(F);
```

Для чтения элементов используются процедуры Read и Readln, в которых первый параметр показывает, откуда происходит ввод данных. После работы файл закрывается. В качестве примера приведем программу, распечатывающую в своем окне содержимое текстового файла 'MyFile.txt':

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  F: TextFile;
  S: string;

begin
  AssignFile(F, 'MyFile.txt');
  Reset(F);
  while not Eof(F) do
  begin
    Readln(F, S);
    Writeln(S);
  end;
  CloseFile(F);
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Так как обычно размер файла заранее не известен, перед каждой операцией чтения вызывается функция Eof, которая возвращает True, если достигнут конец файла.

Внимание! Текстовые файлы можно открывать только для записи или только для чтения, но не для того и другого одновременно. Для того чтобы сначала записать текстовый файл, а потом прочитать, его нужно закрыть после записи и снова открыть, но уже только для чтения.

2.14.3. Стандартные подпрограммы управления файлами

Для обработки файлов в языке Delphi имеется специальный набор процедур и функций:

AssignFile(var F; FileName: string) — связывает файловую переменную F и файл, имя которого указано в FileName.

Reset(var F [: File; RecSize: Word]) — открывает существующий файл. Если открывается нетипизированный файл, то RecSize задает размер элемента файла.

Rewrite(var F [: File; RecSize: Word]) — создает и открывает новый файл.

Append(var F: TextFile) — открывает текстовый файл для добавления текста.

Read(F, V1 [, V2, ..., Vn]) — начиная с текущей позиции, читает из типизированного файла подряд расположенные элементы в переменные V1, V2, ..., Vn.

Read(var F: TextFile; V1 [, V2, ..., Vn]) — начиная с текущей позиции, читает из текстового файла символы или строки в переменные V1, V2, ..., Vn.

Write(F, V1 [, V2, ..., Vn]) — начиная с текущей позиции, записывает в типизированный файл значения V1, V2, ..., Vn.

Write(var F: TextFile; V1 [, V2, ..., Vn]) — начиная с текущей позиции указателя чтения-записи, записывает в текстовый файл значения V1, V2, ..., Vn.

CloseFile(var F) — закрывает ранее открытый файл.

Rename(var F; NewName: string) — переименовывает неоткрытый файл F любого типа. Новое имя задается в NewName.

Erase(var F) — удаляет неоткрытый внешний файл любого типа, заданный переменной F.

Seek(var F; NumRec: Longint) — устанавливает позицию чтения-записи на элемент с номером NumRec; F — типизированный или нетипизированный файл.

SetTextBuf(var F: TextFile; var Buf [; Size: Word]) — назначает текстовому файлу F новый буфер ввода-вывода Buf объема Size.

SetLineBreakStyle(var T: Text; Style: TTextLineBreakStyle) — устанавливает способ переноса строк в файле (одиночный символ #10 или пара символов #13#10).

Flush(var F: TextFile) — записывает во внешний файл все символы, переданные в буфер для записи.

Truncate(var F) — урезает файл, уничтожая все его элементы, начиная с текущей позиции.

IOResult: Integer — возвращает код, характеризующий результат (была ошибка или нет) последней операции ввода-вывода.

FilePos(var F): Longint — возвращает для файла F текущую файловую позицию (номер элемента, на которую она установлена, считая от нуля). Не используется с текстовыми файлами.

FileSize(var F): Longint — возвращает число компонент в файле F. Не используется с текстовыми файлами.

Eoln(var F: Text): Boolean — возвращает булевское значение True, если текущая позиция чтения-записи находится на маркере конца строки. Если параметр F не указан, функция применяется к стандартному устройству ввода с именем Input.

Eof(var F): Boolean — возвращает булевское значение True, если текущая позиция чтения-записи находится сразу за последним элементом, и False в противном случае.

SeekEoln(var F: Text): Boolean — возвращает True при достижении маркера конца строки. Все пробелы и знаки табуляции, предшествующие маркеру, пропускаются.

SeekEof(var F: Text): Boolean — возвращает значение True при достижении маркера конца файла. Все пробелы и знаки табуляции, предшествующие маркеру, пропускаются.

Для работы с нетипизированными файлами используются процедуры BlockRead и BlockWrite. Единица обмена для этих процедур 128 байт.

BlockRead(var F: File; var Buf; Count: Word [; Result: Word]) — считывает из файла F определенное число блоков в память, начиная с первого байта переменной Buf. Параметр Buf представляет любую переменную, используемую для накопления информации из файла F. Параметр Count задает число считываемых блоков. Параметр Result является необязательным и содержит после вызова процедуры число действительно считанных записей. Использование параметра Result подсказывает, что число считанных блоков может быть меньше, чем задано параметром Count.

BlockWrite(var F: File; var Buf; Count: Word [; Result: Word]) — предназначена для быстрой передачи в файл F определенного числа блоков из переменной Buf. Все параметры процедуры BlockWrite аналогичны параметрам процедуры BlockRead.

ChDir(const S: string) — устанавливает текущий каталог.

CreateDir(const Dir: string): Boolean — создает новый каталог на диске.

MkDir(const S: string) — аналог функции CreateDir. Отличие в том, что в случае ошибки при создании каталога функция MkDir создает исключительную ситуацию.

DeleteFile(const FileName: string): Boolean — удаляет файл с диска.

DirectoryExists(const Directory: string): Boolean — проверяет, существует ли заданный каталог на диске.

FileAge(const FileName: string): Integer — возвращает дату и время файла в числовом системно-зависимом формате.

FileExists(const FileName: string): Boolean — проверяет, существует ли на диске файл с заданным именем.

FileIsReadOnly(const FileName: string): Boolean — проверяет, что заданный файл можно только читать.

FileSearch(const Name, DirList: string): string — осуществляет поиск заданого файла в указанных каталогах. Список каталогов задается параметром DirList; каталоги разделяются точкой с запятой для операционной системы Windows и запятой для операционной системы Linux. Функция возвращает полный путь к файлу.

FileSetReadOnly(const FileName: string; ReadOnly: Boolean): Boolean — делает файл доступным только для чтения.

FindFirst/FindNext/FindClose

ForceDirectories(Dir: string): Boolean — создает новый каталог на диске. Позволяет одним вызовом создать все каталоги пути, заданного параметром Dir.

GetCurrentDir: string — возвращает текущий каталог.

SetCurrentDir(const Dir: string): Boolean — устанавливает текущий каталог. Если это сделать невозможно, функция возвращает значение False.

RemoveDir(const Dir: string): Boolean — удаляет каталог с диска; каталог должен быть пустым. Если удалить каталог невозможно, функция возвращает значение False.

RenameFile(const OldName, NewName: string): Boolean — изменяет имя файла. Если это сделать невозможно, функция возвращает значение False.

ChangeFileExt(const FileName, Extension: string): string — возвращает имя файла с измененным расширением.

ExcludeTrailingPathDelimiter(const S: string): string — отбрасывает символ-разделитель каталогов (символ ‘/’ — для Linux и ‘\’ — для Windows), если он присутствует в конце строки.

IncludeTrailingPathDelimiter(const S: string): string — добавляет символ-разделитель каталогов (символ ‘/’ — для Linux и ‘\’ — для Windows), если он отсутствует в конце строки.

ExpandFileName(const FileName: string): string — возвращает полное имя файла (с абсолютным путем) по неполному имени.

ExpandUNCFileName(const FileName: string): string — возвращает полное сетевое имя файла (с абсолютным сетевым путем) по неполному имени. Для операционной системы Linux эта функция эквивалентна функции ExpandFileName.

ExpandFileNameCase(const FileName: string; out MatchFound: TFilenameCaseMatch): string — возвращает полное имя файла (с абсолютным путем) по неполному имени, допуская несовпадения заглавных и строчных букв в имени файла для тех файловых систем, которые этого не допускают (например, файловая система ОС Linux).

ExtractFileDir(const FileName: string): string — выделяет путь из полного имени файла; путь не содержит в конце символ-разделитель каталогов.

ExtractFilePath(const FileName: string): string — выделяет путь из полного имени файла; путь содержит в конце символ-разделитель каталогов.

ExtractRelativePath(const BaseName, DestName: string): string — возвращает относительный путь к файлу DestName, отсчитанный от каталога BaseName. Путь BaseName должен заканчиваться символом-разделителем каталогов.

ExtractFileDrive(const FileName: string): string — выделяет имя диска (или сетевого каталога) из имени файла. Для операционной системы Linux функция возвращает пустую строку.

ExtractFileExt(const FileName: string): string — выделяет расширение файла из его имени.

ExtractFileName(const FileName: string): string — выделяет имя файла, отбрасывая путь к нему.

IsPathDelimiter(const S: string; Index: Integer): Boolean — проверяет, является ли символ S[Index] разделителем каталогов.

MatchesMask(const Filename, Mask: string): Boolean — проверяет, удовлетворяет ли имя файла заданной маске.

2.15. Указатели

2.15.1. Понятие указателя

Все переменные, с которыми мы имели дело, известны уже на этапе компиляции. Однако во многих задачах нужны переменные, которые по мере необходимости можно создавать и удалять во время выполнения программы. С этой целью в языке Delphi организована поддержка так называемых указателей, для которых введен специальный тип данных Pointer.

Не секрет, что любая переменная в памяти компьютера имеет адрес. Переменные, которые содержат адреса других переменных, принято называть *указателями*. Указатели объявляются точно так же, как и обычные переменные:

```
var
  P: Pointer; // переменная-указатель
  N: Integer; // целочисленная переменная
```

Переменная P занимает 4 байта и может содержать адрес любого участка памяти, указывая на байты со значениями любых типов данных: Integer, Real, **string**, **record**, **array** и других. Чтобы инициализировать переменную P, присвоим ей адрес переменной N. Это можно сделать двумя эквивалентными способами:

```
P := Addr(N); // с помощью вызова встроенной функции Addr
```

или

```
P := @N;      // с помощью оператора @
```

В дальнейшем мы будем использовать более краткий и удобный второй способ.

Если некоторая переменная P содержит адрес другой переменной N, то говорят, что P *указывает* на N. Графически это обозначается стрелкой, проведенной из P в N (рисунок 2.12 выполнен в предположении, что N имеет значение 10):

Рисунок 2.12. Графическое изображение указателя P на переменную N

Теперь мы можем изменить значение переменной N, не прибегая к идентификатору N. Для этого слева от оператора присваивания запишем не N, а P вместе с символом ^:

```
P^ := 10; // Здесь умышленно опущено приведение типа
```

Символ ^, записанный после имени указателя, называется *оператором доступа по адресу*. В данном примере переменной, расположенной по адресу, хранящемуся в P, присваивается значение 10. Так как в переменную P мы предварительно занесли адрес N, данное присваивание приводит к такому же результату, что и

```
N := 10;
```

Однако в примере с указателем мы умышленно допустили одну ошибку. Дело в том, что переменная типа Pointer может содержать адреса переменных любого типа, не только Integer. Из-за сильной типизации языка Delphi перед присваиванием мы должны были бы преобразовать выражение P^ к типу Integer:

```
Integer(P^) := 10;
```

Согласитесь, такая запись не совсем удобна. Для того, чтобы сохранить простоту и избежать постоянных преобразований к типу, указатель P следует объявить так:

```
var
  P: ^Integer;
```

При такой записи переменная P по прежнему является указателем, но теперь ей можно присваивать адреса только целых переменных. В данном случае указатель P называют *типизированным*, в отличие от переменных типа Pointer, которые называют *нетипизированными* указателями. При использовании типизированных указателей лучше предварительно вводить соответствующий указательный тип данных, а переменные-указатели просто объявлять с этим типом. Поэтому предыдущий пример можно модифицировать следующим образом:

```
type
  PInteger = ^Integer;
var
  P: PInteger;
```

PInteger — это *указательный тип данных*. Чтобы отличать указательные типы данных от других типов, будем назначать им идентификаторы, начинающиеся с буквы P (от слова Pointer). Объявление указательного типа данных является единственным способом введения указателей на составные переменные, такие как массивы, записи, множества и другие. Например, объявление типа данных для создания указателя на некоторую запись TPerson может выглядеть так:

```
type
  PPerson = ^TPerson;
  TPerson = record
    FirstName: string[20];
    LastName: string[20];
    BirthYear: Integer;
  end;

var
  P: PPerson;
```

Переменная P, описанная с типом данных PPerson, является указателем и может содержать адрес любой переменной типа TPerson. Впредь все указатели мы будем вводить через соответствующие указательные типы данных. Типом Pointer будем пользоваться лишь тогда, когда это действительно необходимо или оправдано.

2.15.2. Динамическое распределение памяти

После объявления в секции **var** указатель содержит неопределенное значение. Поэтому переменные-указатели, как и обычные переменные, перед использованием нужно инициализировать. Отсутствие инициализации указателей является наиболее распространенной ошибкой среди новичков. Причем если использование обычных

неинициализированных переменных приводит просто к неправильным результатам, то использование неинициализированных указателей обычно приводит к ошибке "Access violation" (доступ к неверному адресу памяти) и принудительному завершению приложения.

Один из способов инициализации указателя состоит в присваивании ему адреса некоторой переменной соответствующего типа. Этот способ мы уже рассмотрели. Второй способ состоит в динамическом выделении участка памяти под переменную соответствующего типа и присваивании указателю его адреса. Работа с *динамическими переменными* и есть основное назначение указателей. Размещение динамических переменных производится в специальной области памяти, которая называется Heap (куча). Ее размер равен размеру свободной памяти компьютера.

Для размещения динамической переменной вызывается стандартная процедура

New(var P: Pointer);

Она выделяет требуемый по размеру участок памяти и заносит его адрес в переменную-указатель P. В следующем примере создаются 4 динамических переменных, адреса которых присваиваются переменным-указателям P1, P2, P3 и P4:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type
  PInteger = ^Integer;
  PDouble = ^Double;
  PShortString = ^ShortString;

var
  P1, P2: PInteger;
  P3: PDouble;
  P4: PShortString;

begin
  New(P1);
  New(P2);
  New(P3);
  New(P4);
  ...
end.
```

Далее по адресам в указателях P1, P2, P3 и P4 можно записать значения:

```
P1^ := 10;
P2^ := 20;
P3^ := 0.5;
P4^ := 'Hello!';
```

В таком контексте динамические переменные P1^, P2^, P3^ и P4^ ничем не отличаются от обычных переменных соответствующих типов. Операции над динамическими переменными аналогичны подобным операциям над обычными переменными. Например, следующие операторы могут быть успешно откомпилированы и выполнены:

```
if P1^ < P2^ then
  P1^ := P1^ + P2^; // в P1^ заносится 30
P3^ := P1^;         // в P3^ заносится 30.0
```

После работы с динамическими переменными необходимо освободить занимаемую ими память. Для этого предназначена процедура:

Dispose(var P: Pointer);

Например, в приведенной выше программе явно не хватает следующих строк:

```
Dispose(P4);
Dispose(P3);
Dispose(P2);
Dispose(P1);
```

После выполнения данных утверждений указатели P1, P2, P3 и P4 опять перестанут быть связаны с конкретными адресами памяти. В них будут случайные значения, как и до обращения к процедуре New. Не стоит делать попытки присвоить значения переменным P1^, P2^, P3^ и P4^, ибо в противном случае это может привести к нарушению нормальной работы программы.

Важной особенностью динамической переменной является то, что она не прекращает свое существавание с выходом из области действия ссылающегося на нее указателя. Эта особенность может приводить к таким явлениям, как утечка памяти. Если Вы по каким-то причинам не освободили память, выделенную для динамической переменной, то при выходе из области действия указателя вы потеряете эту память, поскольку уже не будете знать ее адреса.

Поэтому следует четко придерживаться последовательности действий при работе с динамическими переменными:

- создать динамическую переменную;
- выполнить с ней необходимые действия;
- разрушить динамическую переменную.

2.15.3. Операции над указателями

С указателями можно работать как с обычными переменными, например присваивать значения других указателей:

```
P3^ := 20;
P1^ := 50;
P3 := P1;  // теперь P3^ = 50
```

После выполнения этого оператора оба указателя P1 и P3 будут указывать на один и тот же участок памяти. Однако будьте осторожны при операциях с указателями. Только что сделанное присваивание приведет к тому, что память, выделенная ранее для указателя P3, будет потеряна. Отдавая программе участок свободной памяти, система помечает его как занятый. После работы вся память, которая была выделена динамически, должна быть возвращена системе. Поэтому изменение значения указателя P3 без предварительного освобождения связанной с ним динамической переменной является ошибкой.

Использование одинаковых значений в разных указателях открывает некоторые интересные возможности. Так после оператора P3 := P1 изменение значения переменной P3^ будет равносильно изменению значения P1^.

```
P3^ := 70; // теперь P3^ = P1^ = 70
```

В этом нет ничего удивительного, так как указатели P1 и P3 указывают на одну и ту же физическую переменную. Просто для доступа к ней могут использоваться два имени: P1^ и P3^. Такая практика требует большой осмотрительности, поскольку всегда следует различать операции над адресами и операции над данными, хранящимися в памяти по этим адресам.

Указатели можно сравнивать. Так уж сложилось, что понятие больше-меньше в адресации памяти разных моделей компьютеров может иметь противоположный смысл. Из-за этого операции сравнения указателей ограничены двумя: сравнение на равенство или неравенство.

```
if P1 = P2 then ... // Указатели ссылаются на одни и те же данные

if P1 <> P2 then ... // Указатели ссылаются на разные данные
```

Чаще всего операции сравнения указателей используются для проверки того, связан ли указатель с динамической переменной. Если еще нет, то ему следует присвоить значение **nil** (зарезервированное слово):

```
P1 := nil;
```

Установка P1 в **nil** однозначно говорит о том, что указателю не выделена динамическая память. Если всем объявленным указателям присвоить значение **nil**, то внутри программы можно легко выполнить тестирование наподобие этого:

```
if P1 = nil then New(P1);
```

или

```
if P1 <> nil then Dispose(P1);
```

2.15.4. Процедуры GetMem и FreeMem

Для динамического распределения памяти служат еще две тесно взаимосвязанные процедуры: GetMem и FreeMem. Подобно New и Dispose, они во время вызова выделяют и освобождают память для одной динамической переменной:

GetMem(var P: Pointer; Size: Integer) — создает в динамической памяти новую динамическую переменную с заданным размером Size и присваивает ее адрес указателю P. Переменная-указатель P может указывать на данные любого типа.

FreeMem(var P: Pointer [; Size: Integer]) — освобождает динамическую переменную.

Если в программе используется этот способ распределения памяти, то вызовы GetMem и FreeMem должны соответствовать друг другу. Обращения к GetMem и FreeMem могут полностью соответствовать вызовам New и Dispose.

Пример:

```
New(P4);      // Выделить блок памяти для указателя P4
...
Dispose(P4); // Освободить блок памяти
```

Следующий отрывок программы даст тот же самый результат:

```
GetMem(P4, SizeOf(ShortString)); // Выделить блок памяти для P4
...
FreeMem(P4);                     // Освободить блок памяти
```

С помощью процедуры GetMem одной переменной-указателю можно выделить разное количество памяти в зависимости от потребностей. В этом состоит ее основное отличие от процедуры New.

```
GetMem(P4, 20); // Выделить блок в 20 байт для указателя P4
...
FreeMem(P4);    // Освободить блок памяти
```

В данном случае для указателя P4 выделяется меньше памяти, чем может уместиться в переменной типа ShortString, и программист сам должен обеспечить невыход строки за пределы выделенного участка.

В некоторых случаях бывает необходимо перевыделить динамическую память, например для того чтобы разместить в динамической области больше данных. Для этого предназначена процедура:

ReallocMem(var P: Pointer; Size: Integer) — освобождает блок памяти по значению указателя P и выделяет для указателя новый блок памяти заданного размера Size. Указатель P может иметь значение **nil**, а параметр Size — значение 0, что влияет на работу процедуры:

- если P = **nil** и Size = 0, процедура ничего не делает;
- если P = **nil** и Size <> 0, процедура выделяет новый блок памяти заданного размера, что соответствует вызову процедуры GetMem.
- если P <> **nil** и Size = 0, процедура освобождает блок памяти, адресуемый указателем P и устанавливает указатель в значение **nil**. Это соответствует вызову процедуры FreeMem, с той лишь разницей, что FreeMem не очищает указатель;
- если P <> **nil** и Size <> 0, процедура перевыделяет память для указателя P. Размер нового блока определяется значением Size. Данные из прежнего блока копируются в новый блок. Если новый блок больше прежнего, то приращенный участок остается неинициализированным и содержит случайные данные.

2.16. Представление строк в памяти

В некоторых случаях динамическая память неявно используется программой, например для хранения строк. Длина строки может варьироваться от нескольких символов до миллионов и даже миллиардов (теоретический предел равен 2 ГБ). Тем не менее, работа со строками в программе осуществляется так же просто, как работа с переменными простых типов данных. Это возможно потому, что компилятор автоматически генерирует код для выделения и освобождения динамической памяти, в которой хранятся символы строки. Но что стоит за такой простотой? Не идет ли она в ущерб эффективности? С полной уверенностью можем ответить, что эффективность программы не только не снижается, но даже повышается.

Физически переменная строкового типа представляет собой указатель на область динамической памяти, в которой размещаются символы. Например, переменная S на самом деле представляет собой указатель и занимает всего четыре байта памяти (SizeOf(S) = 4):

```
var
  S: string; // Эта переменная физически является указателем
```

При объявлении этот указатель автоматически инициализируется значением **nil**. Оно показывет, что строка является пустой. Функция SetLength, устанавливающая размер строки, на самом деле резервирует необходимый по размеру блок динамической памяти и записывает его адрес в строковую переменную:

```
SetLength(S, 100); // S получает адрес распределенного блока динамической памяти
```

За оператором присваивания строковых переменных на самом деле кроется копирование значения указателя, а не копирование блока памяти, в котором хранятся символы.

```
S2 := S1; // Копируются лишь адреса
```

Такой подход весьма эффективен как с точки зрения производительности, так и с точки зрения экономного использования оперативной памяти. Его главная проблема состоит в том, чтобы обеспечить удаление блока памяти, содержащего символы строки, когда все адресующие его строковые переменные прекращают свое существование. Эта проблема эффективно решается с помощью механизма подсчета количества ссылок (reference counting). Для понимания его работы рассмотрим формат хранения строк в памяти подробнее.

Пусть в программе объявлены две строковые переменные:

```
var
  S1, S2: string; // Физически эти переменные являются указателями
```

И пусть в программе существует оператор, присваивающий переменной S1 значение некоторой функции:

```
Readln(S1); // В S1 записывается адрес считанной строки
```

Для хранения символов строки S1 по окончании ввода будет выделен блок динамической памяти. Формат этого блока после ввода значения 'Hello' показан на рисунке 2.13:

Рисунок 2.13. Представление строковых переменных в памяти

Как можно заметить, блок динамической памяти, выделенный для хранения символов строки, дополнительно содержит два поля, расположенных перед первым символом (по отрицательным смещениям относительно строкового указателя). Первое поле хранит количество ссылок на данную строку, а второе — длину строки.

Если в программе встречается оператор присваивания значения одной строковой переменной другой строковой переменной,

```
S2 := S1; // Теперь S2 указывает на тот же блок памяти, что и S1
```

то, как мы уже сказали, копия строки в памяти не создается. Копируется только адрес, хранящийся в строковой переменной, и на единицу увеличивается количество ссылок на строку (рисунок 2.14).

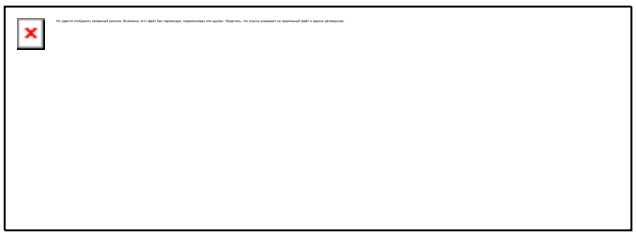


Рисунок 2.14. Результат копирования строковой переменной S1 в строковую переменную S2

При присваивании переменной S1 нового значения (например, пустой строки):

```
S1 := '';
```

количество ссылок на предыдущее значение уменьшается на единицу (рисунок 2.15).

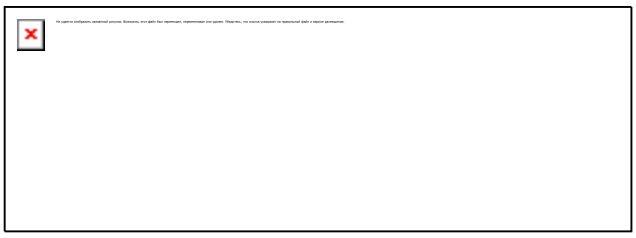


Рисунок 2.15. Результат присваивания строковой переменной S1 нового значения (пустой строки)

Блок динамической памяти освобождается, когда количество ссылок на строку становится равным нулю. Этим обеспечивается автоматическое освобождение неиспользуемой памяти.

Интересно, а что происходит при изменении символов строки, с которой связано несколько строковых переменных? Правила семантики языка требуют, чтобы две строковые

переменные были логически независимы, и изменение одной из них не влияло на другую. Это достигается с помощью механизма копирования при записи (copy-on-write).

Например, в результате выполнения операторов

```
  S1 := S2;                 // S1 указывает на ту же строку, что и S2
  S1[3] := '-';             // Автоматически создается копия строки
```

получим следующую картину в памяти (рисунок 2.16):

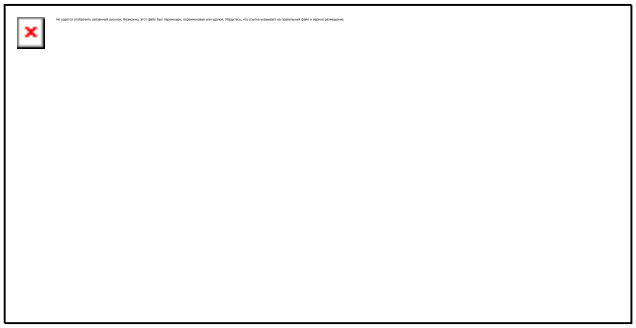


Рисунок 2.16. Результат изменения символа в строке S1

Работая сообща, механизмы подсчета количества ссылок и копирования при записи позволяют наиболее эффективно работать со строками. Это как раз тот случай, когда простота и удобство сочетается с мощью и эффективностью.

Все, что было сказано выше о представлении в памяти строк, относится только к строкам формата AnsiString. Строки формата WideString тоже хранятся в динамической памяти, но для них не поддерживаются механизм подсчета количества ссылок и механизм копирования по записи. Операция присваивания строковых переменных формата WideString означает выделение нового блока динамической памяти и полное копирование в него всех символов исходной строки. Что же касается коротких строк, то они целиком хранятся по месту объявления: или в области данных программы (если это глобальные переменные), или на стеке (если это локальные переменные). Динамическая память вообще не используется для хранения коротких строк.

2.17. Динамические массивы

Одним из мощнейших средств языка Delphi являются динамические массивы. Их основное отличие от обычных массивов заключается в том, что они хранятся в динамической памяти. Этим и обусловлено их название. Чтобы понять, зачем они нужны, рассмотрим пример:

```
var
  N: Integer;
  A: array[1..100] of Integer; // обычный массив
begin
  Write('Введите количество элементов: ');
  ReadLn(N);
  ...
end.
```

Задать размер массива A в зависимости от введенного пользователем значения невозможно, поскольку в качестве границ массива необходимо указать константные значения. А введенное пользователем значение никак не может претендовать на роль константы. Иными словами, следующее объявление будет ошибочным:

```
var
  N: Integer;
  A: array[1..N] of Integer; // Ошибка!
begin
  Write('Введите количество элементов: ');
  ReadLn(N);
  ...
end.
```

На этапе написания программы невозможно предугадать, какие именно объемы данных захочет обрабатывать пользователь. Тем не менее, Вам придется ответить на два важных вопроса:

- На какое количество элементов объявить массив?
- Что делать, если пользователю все-таки понадобится большее количество элементов?

Вы можете поступить следующим образом. В качестве верхней границы массива установить максимально возможное (с вашей точки зрения) количество элементов, а реально использовать только часть массива. Если пользователю потребуется большее количество элементов, чем зарезервировано Вами, то ему можно попросту вежливо отказать. Например:

```
const
  MaxNumberOfElements = 100;
var
  N: Integer;
  A: array[1.. MaxNumberOfElements] of Integer;
begin
  Write('Введите количество элементов (не более ', MaxNumberOfElements, '): ');
  ReadLn(N);
  if N > MaxNumberOfElements then
  begin
    Write('Извините, программа не может работать ');
    Writeln('с количеством элементов больше , ' MaxNumberOfElements, '.');
  end
  else
  begin
    ... // Инициализируем массив необходимыми значениями и обрабатываем его
  end;
end.
```

Такое решение проблемы является неоптимальным. Если пользователю необходимо всего 10 элементов, программа работает без проблем, но всегда использует объем памяти, необходимый для хранения 100 элементов. Память, отведенная под остальные 90 элементов, не будет использоваться ни Вашей программой, ни другими программами (по принципу «сам не гам и другому не дам»). А теперь представьте, что все программы поступают таким же образом. Эффективность использования оперативной памяти резко снижается.

Динамические массивы позволяют решить рассмотренную проблему наилучшим образом. Размер динамического массива можно изменять во время работы программы.

Динамический массив объявляется без указания границ:

```
var
  DynArray: array of Integer;
```

Переменная DynArray представляет собой ссылку на размещаемые в динамической памяти элементы массива. Изначально память под массив не резервируется, количество элементов в массиве равно нулю, а значение переменной DynArray равно **nil**.

Работа с динамическими массивами напоминает работу с длинными строками. В частности, создание динамического массива (выделение памяти для его элементов) осуществляется той же процедурой, которой устанавливается длина строк — SetLength.

```
SetLength(DynArray, 50);       // Выделить память для 50 элементов
```

Изменение размера динамического массива производится этой же процедурой:

```
SetLength(DynArray, 100);       // Теперь размер массива 100 элементов
```

При изменении размера массива значения всех его элементов сохраняются. При этом последовательность действий такова: выделяется новый блок памяти, значения элементов из старого блока копируются в новый, старый блок памяти освобождается.

При уменьшении размера динамического массива лишние элементы теряются.

При увеличении размера динамического массива добавленные элементы не инициализируются никаким значением и в общем случае их значения случайны. Однако если динамический массив состоит из элементов, тип которых предполагает автоматическую инициализацию пустым значением (string, Variant, динамический массив, др.), то добавленная память инициализируется нулями.

Определение количества элементов производится с помощью функции Length:

```
N := Length(DynArray);          // N получит значение 100
```

Элементы динамического массива всегда индексируются от нуля. Доступ к ним ничем не отличается от доступа к элементам обычных статических массивов:

```
DynArray[0] := 5;               // Присвоить начальному элементу значение 5
DynArray[High(DynArray)] := 10; // присвоить конечному элементу значение 10
```

К динамическим массивам, как и к обычным массивам, применимы функции Low и High, возвращающие минимальный и максимальный индексы массива соответственно. Для динамических массивов функция Low всегда возвращает 0.

Освобождение памяти, выделенной для элементов динамического массива, осуществляется установкой длины в значение 0 или присваиванием переменной-массиву значения **nil** (оба варианта эквивалентны):

```
SetLength(DynArray, 0); // Эквивалентно: DynArray := nil;
```

Однако Вам вовсе необязательно по окончании использования динамического массива освобождать выделенную память, поскольку она освобождается автоматически при выходе из области действия переменной-массива (удобно, не правда ли!). Данная возможность обеспечивается уже известным Вам механизмом подсчета количества ссылок.

Также, как и при работе со строками, при присваивании одного динамического массива другому, копия уже существующего массива не создается.

```
var
  A, B: array of Integer;
begin
  SetLength(A, 100);  // Выделить память для 100 элементов
  A[0] := 5;
  B := A;             // A и B указывают на одну и ту же область памяти!
  B[1] := 7;          // Теперь A[1] тоже равно 7!
  B[0] := 3;          // Теперь A[0] равно 3, а не 5!
end.
```

В приведенном примере, в переменную B заносится адрес динамической области памяти, в которой хранятся элементы массива A (другими словами, ссылочной переменной B присваивается значение ссылочной переменной A).

Как и в случае со строками, память освобождается, когда количество ссылок становится равным нулю.

```
var
  A, B: array of Integer;
begin
  SetLength(A, 100); // Выделить память для 100 элементов
  A[0] := 10;
  B := A;            // B указывает на те же элементы, что и A
  A := nil;          // Память еще не освобождается, поскольку на нее указывает B
  B[1] := 5;         // Продолжаем работать с B, B[0] = 10, а B[1] = 5
  B := nil;          // Теперь ссылок на блок памяти нет. Память освобождается
end;
```

Для работы с динамическими массивами вы можете использовать знакомую по строкам функцию Copy. Она возвращает часть массива в виде нового динамического массива.

Не смотря на сильное сходство динамических массивов со строками, у них имеется одно существенное отличие: отсутствие механизма копирования при записи (copy-on-write).

2.18. Нуль-терминированные строки

Кроме стандартных строк ShortString и AnsiString, в языке Delphi поддерживаются нуль-терминированные строки языка C, используемые процедурами и функциями Windows. *Нуль-терминированная строка* представляет собой индексированный от нуля массив ASCII-символов, заканчивающийся нулевым символом #0. Для поддержки нуль-терминированных строк в языке Delphi введены три указательных типа данных:

```
type
  PAnsiChar = ^AnsiChar;
  PWideChar = ^WideChar;
  PChar = PAnsiChar;
```

Типы PAnsiChar и PWideChar являются фундаментальными и на самом деле используются редко. PChar — это обобщенный тип данных, в основном именно он используется для описания нуль-терминированных строк.

Ниже приведены примеры объявления нуль-терминированных строк в виде типизированных констант и переменных:

```
const
  S1: PChar = 'Object Pascal';               // #0 дописывается автоматически
  S2: array[0..12] of Char = 'Delphi/Kylix'; // #0 дописывается автоматически
var
  S3: PChar;
```

Переменные типа PChar являются указателями, а не настоящими строками. Поэтому, если переменной типа PChar присвоить значение другой переменной такого же типа, то в результате получится два указателя на одну и ту же строку, а не две копии исходной строки. Например, в результате оператора

```
S3 := S1;
```

переменная S3 получит адрес уже существующей строки 'Object Pascal'.

Для удобной работы с нуль-терминированными строками в языке Delphi предусмотрена директива **$EXTENDEDSYNTAX**. Если она включена (**ON**), то появляются следующие дополнительные возможности:

- массив символов, в котором нижний индекс равен 0, совместим с типом PChar;
- строковые константы совместимы с типом PChar.
- указатели типа PChar могут участвовать в операциях сложения и вычитания с целыми числами; допустимо также вычитание (но не сложение!) указателей.

В режиме расширенного синтаксиса допустимы, например, следующие операторы:

```
S3 := S2;      // S3 указывает на строку 'Delphi/Kylix'
S3 := S1 + 7; // S3 указывает на подстроку 'Pascal'
```

В языке Delphi существует богатый набор процедур и функций для работы с нуль-терминированными строками (см. справочник по среде Delphi).

2.19. Переменные с непостоянным типом значений

2.19.1. Тип данных Variant

В среде Delphi определен стандартный тип данных Variant, с помощью которого объявляются переменные с непостоянным типом значений. Такие переменные могут принимать значения разных типов данных в зависимости от типа выражения, в котором используются. Следующий пример хорошо демонстрирует мощь переменных с непостоянным типом значений:

```
program Console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  V1, V2, V3, V4: Variant;

begin
  V1 := 5;              // целое число
  V2 := 0.8;            // вещественное число
  V3 := '10';           // строка
  V4 := V1 + V2 + V3;   // вещественное число 15.8
  Writeln(V4);          // 15.8
  Writeln('Press Enter to exit...');
  Readln;
end.
```

2.19.2. Значения переменных с типом Variant

Переменные с непостоянным типом содержат целые, вещественные, строковые, булевские значения, дату и время, массивы и др. Кроме того, переменные с типом Variant принимают два специальных значения: Unassigned и Null.

Значение **Unassigned** показывает, что переменная является нетронутой, т.е. переменной еще не присвоено значение. Оно автоматически устанавливается в качестве начального значения любой переменной с типом Variant.

Значение **Null** показывает, что переменная имеет неопределенное значение. Если в выражении участвует переменная со значением Null, то результат всего выражения тоже равен Null.

Переменная с типом Variant занимает в памяти 16 байт. В них хранятся текущее значение переменной (или адрес значения в динамической памяти) и тип этого значения.

Тип значения выясняется с помощью функции

VarType(const V: Variant): Integer;

Возвращаемый результат формируется из констант, перечисленных в таблице 2.10. Например, следующий условный оператор проверяет, содержит ли переменная строку (массив строк):

```
if VarType(V) and varTypeMask = varString then ...
```

Код типа	Значение	Описание
varEmpty	$0000	Переменная содержит значение Unassigned.
varNull	$0001	Переменная содержит значение Null.
varSmallint	$0002	Переменная содержит значение типа Smallint.
varInteger	$0003	Переменная содержит значение типа Integer.
varSingle	$0004	Переменная содержит значение типа Single.
varDouble	$0005	Переменная содержит значение типа Double.
varCurrency	$0006	Переменная содержит значение типа Currency.
varDate	$0007	Переменная содержит значение типа TDateTime.
varOleStr	$0008	Переменная содержит ссылку на строку формата Unicode в динамической памяти.
varDispatch	$0009	Переменная содержит ссылку на интерфейс IDispatch (интерфейсы рассмотрены в главе 6).
varError	$000A	Переменная содержит системный код ошибки.
varBoolean	$000B	Переменная содержит значение типа WordBool.
varVariant	$000C	Элемент варьируемого массива содержит значение типа Variant (код varVariant используется только в сочетании с флагом varArray).
varUnknown	$000D	Переменная содержит ссылку на интерфейс IUnknown (интерфейсы рассмотрены в главе 6).
varShortint	$0010	Переменная содержит значение типа Shortint
varByte	$0011	Переменная содержит значение типа Byte.
varWord	$0012	Переменная содержит значение типа Word
varLongword	$0013	Переменная содрежит значение типа Longword
varInt64	$0014	Переменная содержит значение типа Int64
varStrArg	$0048	Переменная содержит строку, совместимую

		со стандартом COM, принятым в операционной системе Windows.
varString	$0100	Переменная содержит ссылку на длинную строку.
varAny	$0101	Переменная содержит значение любого типа данных технологии CORBA
Флаги		
varTypeMask	$0FFF	Маска для выяснения типа значения.
varArray	$2000	Переменная содержит массив значений.
varByRef	$4000	Переменная содержит ссылку на значение.

Таблица 2.10. Коды и флаги варьируемых переменных

Функция

VarAsType(const V: Variant; VarType: Integer): Variant;

позволяет вам преобразовать значение варьируемой переменной к нужному типу, например:

```
V1 := '100';
V2 := VarAsType(V1, varInteger);
```

Пока это все, что нужно знать о типе Variant, но мы к нему еще вернемся при обсуждении технологии COM Automation.

2.20. Delphi + ассемблер

В процессе разработки программы вы можете неожиданно обнаружить, что описанных выше средств языка Delphi для решения некоторых насущных проблем явно недостаточно. Например, организация критичных по времени вычислений требует использования ассемблера. Кроме того, часто возникает необходимость включить в программу на языке Delphi откомпилированные ранее процедуры и функции, написанные на ассемблере. Разработчики языка учли эти проблемы и дали программисту необходимые средства их решения.

2.20.1. Встроенный ассемблер

Пользователю предоставляется возможность делать вставки на встроенном ассемблере в исходный текст на языке Delphi.

К встроенному ассемблеру можно обратиться с помощью зарезервированного слова **asm**, за которым следуют команды ассемблера и слово **end**:

```
asm
  <оператор ассемблера>
  ...
  <оператор ассемблера>
end;
```

На одной строке можно поместить несколько операторов ассемблера, разделенных двоеточием. Если каждый оператор размещен на отдельной строке, двоеточие не ставится.

В языке Delphi имеется возможность не только делать ассемблерные вставки, но писать процедуры и функции полностью на ассемблере. В этом случае тело подпрограммы ограничивается словами **asm** и **end** (а не **begin** и **end**), между которыми помещаются

инструкции ассемблера. Перед словом **asm** могут располагаться объявления локальных констант, типов, и переменных. Например, вот как могут быть реализованы функции вычисления минимального и максимального значения из двух целых чисел:

```
function Min(A, B: Integer): Integer; register;
asm
  CMP    EDX, EAX
  JGE    @@1
  MOV    EAX, EDX
  @@1:
end;

function Max(A, B: Integer): Integer; register;
asm
  CMP    EDX, EAX
  JLE    @@1
  MOV    EAX, EDX
  @@1:
end;
```

Обращение к этим функциям имеет привычный вид:

```
Writeln(Min(10, 20));
Writeln(Max(10, 20));
```

2.20.2. Подключение внешних подпрограмм

Программисту предоставляется возможность подключать к программе или модулю отдельно скомпилированные процедуры и функции, написанные на языке ассемблера или C. Для этого используется директива компилятора **$LINK** и зарезервированное слово **external**. Директива **{$LINK <имя файла>}** указывает подключаемый объектный модуль, а **external** сообщает компилятору, что подпрограмма внешняя.

Предположим, что на ассемблере написаны и скомпилированы функции Min и Max, их объектный код находится в файле MINMAX.OBJ. Подключение функций Min и Max к программе на языке Delphi будет выглядеть так:

```
function Min(X, Y: Integer): Integer; external;
function Max(X, Y: Integer): Integer; external;
{$LINK MINMAX.OBJ}
```

В модулях внешние подпрограммы подключаются в разделе **implementation**.

2.21. Итоги

Все, что вы изучили, называется языком Delphi. Мы надеемся, что вам понравились стройность и выразительная сила языка. Но это всего лишь основа. Теперь пора подняться на следующую ступень и изучить технику объектно-ориентированного программирования, без которого немыслимо стать профессиональным программистом. Именно этим вопросом в рамках применения объектов в среде Delphi мы и займемся в следующей главе.

Глава 3. Объектно-ориентированное программирование (ООП)

Объекты — это крупнейшее достижение в современной технологии программирования. Они позволили строить программу не из чудовищных по сложности процедур и функций, а из кирпичиков-объектов, заранее наделенных нужными свойствами. Самое приятное в объктах то, что их внутренняя сложность скрыта от программиста, который просто пользуется готовым строительным материалом.

Сейчас преимущества использования объектов очевидны для всех. Однако так было не всегда. Сначала старая гвардия не поняла и не приняла объекты, поэтому они почти 20 лет потихоньку развивались в различных языках, первым из которых была Simula 67.

Постепенно объектно-ориентированный подход нашел себе место и в более мощных языках, таких как C++, Delphi и множестве других языков. Блестящим примером реализации объектов была библиотека Turbo Vision, предназначенная для построения пользовательского интерфейса программ в операционной системе MS-DOS.

Полную победу объекты одержали с приходом эпохи многофункциональных графических пользовательских интерфейсов. Теперь без объектов в программировании просто не обойтись. Чтобы вы не рылись в других книгах, собирая информацию по крохам, мы не поленились и объединили в этой главе все, что нужно знать об объектах. Для новичка важнейшее здесь: инкапсуляция, наследование, полиморфизм, остальное можно просто просмотреть и возвращаться к материалу по мере накопления опыта. Профессионалу полезно прочитать внимательно все от начала до конца. Поэтому давайте засучим рукава и приступим к делу.

3.1. Краеугольные камни ООП

3.1.1. Формула объекта

Авторы надеются, что читатель помнит кое-что из главы 2 и такие понятия как тип данных, процедура, функция, запись для него не в новинку. Это прекрасно. Так вот, в конце 60-х годов кому-то пришло в голову объединить эти понятия, и то, что получилось, назвать объектом. Рассмотрение данных в неразрывной связи с методами их обработки позволило вывести *формулу объекта*:

Объект = Данные + Операции

На основании этой формулы была разработана методология *объектно-ориентированного программирования* (ООП).

3.1.2. Природа объекта

Об объектах можно думать как о полезных существах, которые "живут" в вашей программе и коллективно решают некоторую прикладную задачу. Вы, как Демиург, лепите этих существ, распределяете между ними обязанности и устанавливаете правила их взаимодействия.

В общем случае каждый объект "помнит" необходимую информацию, "умеет" выполнять некоторый набор действий и характеризуется набором свойств. То, что объект "помнит", хранится в его *полях*. То, что объект "умеет делать", реализуется в виде его внутренних процедур и функций, называемых *методами*. *Свойства* объектов аналогичны свойствам, которые мы наблюдаем у обычных предметов. Значения свойств можно устанавливать и читать. Программно свойства реализуются через поля и методы.

Например, объект "кнопка" имеет свойство "цвет". Значение цвета кнопка запоминает в одном из своих полей. При изменении значения свойства "цвет" вызывается метод, который перерисовывает кнопку.

Кстати, этот пример позволяет сделать важный вывод: свойства имеют первостепенное значение для программиста, использующего объект. Чтобы понять суть и назначение объекта вы обязательно должны знать его свойства, иногда — методы, очень редко — поля (объект и сам знает, что с ними делать).

3.1.3. Объекты и компоненты

Когда прикладные программы были консольно-ориентированными, а пользовательский интерфейс был простым, объекты казались пределом развития программирования, поскольку были идеальным средством разбиения сложных задач на простые подзадачи. Однако с появлением графических систем программирование пользовательского интерфейса резко усложнилось. Программист в какой-то мере стал дизайнером, а визуальная компоновка и увязка элементов пользовательского интерфейса (кнопок, меток, строк редактора) начали

отнимать основную часть времени. И тогда программистам пришла в голову идея визуализировать объекты, объединив программную часть объекта с его видимым представлением на экране дисплея в одно целое. То, что получилось в результате, было названо компонентом.

Компоненты в среде Delphi — это особые объекты, которые являются строительными кирпичиками визуальной среды разработки и приспособлены к визуальной установке свойств. Чтобы превратить объект в компонент, первый разрабатывается по определенным правилам, а затем помещается в палитру компонентов. Конструируя приложение, вы берете компоненты из Палитры Компонентов, располагаете на форме и устанавливаете их свойства в окне Инспектора Объектов. Внешне все выглядит просто, но чтобы достичь такой простоты, потребовалось создать механизмы, обеспечивающие функционирование объектов-компонентов уже на этапе проектирования приложения! Все это было придумано и блестяще реализовано в среде Delphi. Таким образом, компонентный подход значительно упростил создание приложений с графическим пользовательским интерфейсом и дал толчок развитию новой индустрии компонентов.

В данной главе мы рассмотрим лишь вопросы создания и использования объектов. Чуть позже мы научим вас превращать объекты в компоненты (см. главу 13).

3.1.4. Классы объектов

Каждый объект всегда принадлежит некоторому классу объектов. *Класс объектов* — это обобщенное (абстрактное) описание множества однотипных объектов. Объекты являются конкретными представителями своего класса, их принято называть *экземплярами класса*. Например, класс СОБАКИ — понятие абстрактное, а экземпляр этого класса МОЙ ПЕС БОБИК — понятие конкретное.

3.1.5. Три кита ООП

Весь мир ООП держится на трех китах: инкапсуляции, наследовании и полиморфизме. Для начала о них надо иметь только самое общее представление.

Объединение данных и операций в одну сущность — объект — тесно связано с понятием *инкапсуляции,* которое означает сокрытие внутреннего устройства. Инкапсуляция делает объекты похожими на маленькие программные модули, в которых скрыты внутренние данные и у которых имеется интерфейс использования в виде подпрограмм. Переход от понятий «структура данных» и «алгоритм» к понятию «объект» значительно повысил ясность и надежность программ.

Второй кит ООП — *наследование*. Этот простой принцип означает, что если вы хотите создать новый класс объектов, который расширяет возможности уже существующего класса, то нет необходимости в переписывании заново всех полей, методов и свойств. Вы объявляете, что новый класс является *потомком* (или *дочерним классом*) имеющегося класса объектов, называемого *предком* (или *родительским классом*), и добавляете к нему новые поля, методы и свойства. Процесс порождения новых классов на основе других классов называется *наследованием*. Новые классы объектов имеют как унаследованные признаки, так и, возможно, новые. Например, класс СОБАКИ унаследовал многие свойства своих предков — ВОЛКОВ.

Третий кит — это *полиморфизм*. Он означает, что в производных классах вы можете изменять работу уже существующих в базовом классе методов. При этом весь программный код, управляющий объектами родительского класса, пригоден для управления объектами дочернего класса без всякой модификации. Например, вы можете породить новый класс кнопок с рельефной надписью, переопределив метод рисования кнопки. Новую кнопку можно "подсунуть" вместо стандартной в какую-нибудь подпрограмму, вызывающую рисование кнопки. При этом подпрограмма "думает", что работает со стандартной кнопкой,

но на самом деле кнопка принадлежит производному классу кнопок и отображается в новом стиле. Пока достаточно самого поверхностного понимания всех приведенных выше понятий, ниже мы рассмотрим их подробнее и покажем, как они реализованы в среде Delphi.

3.2. Классы

Для поддержки ООП в язык Delphi введены *объектные типы данных*, с помощью которых одновременно описываются данные и операции над ними. Объектные типы данных называют *классами*, а их экземпляры — *объектами*.

Классы объектов определяются в секции **type** глобального блока. Описание класса начинается с ключевого слова **class** и заканчивается ключевым словом **end**. По форме объявления классы похожи на обычные записи, но помимо полей данных могут содержать объявления пользовательских процедур и функций. Такие процедуры и функции обобщенно называют *методами*, они предназначены для выполнения над объектами различных операций. Приведем пример объявления класса, который предназначен для чтения текстового файла в формате "delimited text" (файл в таком формате представляет собой последовательность строк; каждая строка состоит из значений, которые отделены друг от друга символом-разделителем):

```
type
  TDelimitedReader = class
    // Поля
    FileVar: TextFile;
    Items: array of string;
    Delimiter: Char;
    // Методы
    procedure PutItem(Index: Integer; const Item: string);
    procedure SetActive(const AActive: Boolean);
    function ParseLine(const Line: string): Integer;
    function NextLine: Boolean;
    function GetEndOfFile: Boolean;
  end;
```

Класс содержит поля (FileVar, Items, Delimiter) и методы (PutItem, SetActive, ParseLine, NextLine, GetEndOfFile). Заголовки методов, (всегда) следующие за списком полей, играют роль упреждающих (forward) описаний. Программный код методов пишется отдельно от определения класса и будет приведен позже.

Класс обычно описывает сущность, моделируемую в программе. Например, класс TDelimitedReader представляет собой "читатель" текстового файла с разбором считываемых строк на элементы (подстроки), которые отделены друг от друга некоторым символом, называемым разделителем.

Класс содержит несколько полей:

- FileVar — файловая переменная, необходимая для доступа к файлу;
- Delimiter — символ, который служит разделителем элементов;
- Items — массив элементов, полученных разбором последней считанной строки;

Класс также содержит ряд методов (процедур и функций):

- PutItem — помещает элемент в массив Items по индексу Index; если индекс превышает верхнюю границу массива, то размер массива автоматически увеличивается;
- SetActive — открывает или закрывает файл, из которого производится чтение строк;
- ParseLine — осуществляет разбор строки: выделяет элементы из строки и помещает их в массив Items; возвращает количество выделенных элементов;

- NextLine — считывает очередную строку из файла и с помощью метода ParseLine осуществляет ее разбор; в случае успешного чтения очередной строки функция возвращает значение True, а иначе — значение False (достигнут конец файла);
- GetEndOfFile — возвращает булевское значение, показывающее, достигнут ли конец файла.

Обратите внимание, что приведенное выше описание является ничем иным, как декларацией интерфейса для работы с объектами класса TDelimitedReader. Реализация методов PutItem, SetActive, ParseLine, NextLine и GetEndOfFile на данный момент отсутствует, однако для создания и использования экземпляров класса она пока и не нужна.

В некотором смысле объекты похожи на программные модули, для использования которых необходимо изучить лишь интерфейсную часть, раздел реализации для этого изучать не требуется. Поэтому дальше от описания класса мы перейдем не к реализации методов, а к созданию на их основе объектов.

3.3. Объекты

Чтобы от описания класса перейти к объекту, следует выполнить соответствующее объявление в секции **var**:

```
var
  Reader: TDelimitedReader;
```

При работе с обычными типами данных этого объявления было бы достаточно для получения экземпляра типа. Однако объекты в среде Delphi являются динамическими данными, т.е. распределяются в динамической памяти. Поэтому переменная Reader — это просто *ссылка на экземпляр* (*объект* в памяти), которого физически еще не существует. Чтобы *сконструировать* объект (выделить память для экземпляра) класса TDelimitedReader и *связать* с ним переменную Reader, нужно в тексте программы поместить следующий оператор:

```
Reader := TDelimitedReader.Create;
```

Create — это так называемый *конструктор* объекта; он всегда присутствует в классе и служит для создания и инициализации экземпляров. При создании объекта в памяти выделяется место только для его полей. Методы, как и обычные процедуры и функции, помещаются в область кода программы; они умеют работать с любыми экземплярами своего класса и не дублируются в памяти.

После создания объект можно использовать в программе: получать и устанавливать значения его полей, вызывать его методы. Доступ к полям и методам объекта происходит с помощью уточненных имен, например:

```
Reader.NextLine;
```

Кроме того, как и при работе с записями, допустимо использование оператора **with**, например:

```
with Reader do
  NextLine;
```

Если объект становится ненужным, он должен быть удален вызовом специального метода Destroy, например:

```
Reader.Destroy; // Освобождение памяти, занимаемой объектом
```

Destroy — это так называемый *деструктор* объекта; он присутствует в классе наряду с конструктором и служит для удаления объекта из динамической памяти. После вызова деструктора переменная Reader становится *несвязанной* и не должна использоваться для доступа к полям и методам уже несуществующего объекта. Чтобы отличать в программе связанные объектные переменные от несвязанных, последние следует инициализировать

значением **nil**. Например, в следующем фрагменте обращение к деструктору Destroy выполняется только в том случае, если объект реально существует:

```
Reader := nil;
...
if Reader <> nil then Reader.Destroy;
```

Вызов деструктора для несуществующих объектов недопустим и при выполнении программы приведет к ошибке. Чтобы избавить программистов от лишних ошибок, в объекты ввели предопределенный метод Free, который следует вызывать вместо деструктора. Метод Free сам вызывает деструктор Destroy, но только в том случае, если значение объектной переменной не равно **nil**. Поэтому последнюю строчку в приведенном выше примере можно переписать следующим образом.

```
Reader.Free;
```

После уничтожения объекта переменная Reader сохраняет свое значение, продолжая ссылаться на место в памяти, где объекта уже нет. Если эту переменную предполагается еще использовать, то желательно присвоить ей значение **nil**, чтобы программа могла проверить, существует объект или нет. Таким образом, наиболее правильная последовательность действий при уничтожении объекта должна быть следующая:

```
Reader.Free;
Reader := nil;
```

С помощью стандартной процедуры FreeAndNil это можно сделать проще и элегантнее:

```
FreeAndNil(Reader); // Эквивалентно: Reader.Free; Reader := nil;
```

Значение одной объектной переменной можно присвоить другой. При этом объект не копируется в памяти, а вторая переменная просто связывается с тем же объектом, что и первая:

```
var
  R1, R2: TDelimitedReader; // Переменные R1 и R2 не связаны с объектом
begin
  R1 := TDelimitedReader.Create; // Связывание переменной R1 с новым объектом
            // Переменная R2 пока еще не связана ни с каким объектом
  R2 := R1; // Связывание переменной R2 с тем же объектом, что и R1
            // Теперь обе переменные связаны с одним объектом
  R2.Free;  // Уничтожение объекта
            // Теперь R1 и R2 не связаны ни с каким объектом
end;
```

Объекты могут выступать в программе не только в качестве переменных, но также элементов массивов, полей записей, параметров процедур и функций. Кроме того, они могут служить полями других объектов. Во всех этих случаях программист фактически оперирует указателями на экземпляры объектов в динамической памяти. Следовательно, объекты изначально приспособлены для создания сложных динамических структур данных, таких как списки и деревья. Указатели на объекты для этого не нужны.

В некоторых случаях требуется, чтобы объекты разных классов содержали ссылки друг на друга. Возникает проблема: объявление первого класса будет содержать ссылку на еще не определенный класс. Она решается с помощью упреждающего объявления:

```
type
  TReadersList = class;  // упреждающее объявление класса TReadersList

  TDelimitedReader = class
    Owner: TReadersList;
    ...
  end;

  TReadersList = class
    Readers: array of TDelimitedReader;
    ...
  end;
```

Первое объявление класса TReadersList называется *упреждающим* (от англ. forward). Оно необходимо для того, чтобы компилятор нормально воспринял объявление поля Owner в классе TDelimitedReader.

Итак, вы уже имеете некоторое представление об объектах, перейдем теперь к вопросу реализации их методов.

3.4. Конструкторы и деструкторы

Особой разновидностью методов являются *конструкторы* и *деструкторы*. Напомним, что конструкторы создают, а деструкторы разрушают объекты. Создание объекта включает выделение памяти под экземпляр и инициализацию его полей, а разрушение — очистку полей и освобождение памяти. Действия по инициализации и очистке полей специфичны для каждого конкретного класса объектов. По этой причине язык Delphi позволяет *переопределить* стандартный конструктор Create и стандартный деструктор Destroy для выполнения любых полезных действий. Можно даже определить несколько конструкторов и деструкторов (имена им назначает сам программист), чтобы обеспечить различные процедуры создания и разрушения объектов.

Объявление конструкторов и деструкторов похоже на объявление обычных методов с той лишь разницей, что вместо зарезервированных слов **function** и **procedure** используются слова **constructor** и **destructor**. Для нашего класса TDelimitedReader потребуется конструктор, которому в качестве параметра будет передаваться имя обрабатываемого файла и разделитель элементов:

```
type
  TDelimitedReader = class
    ...
    // Конструкторы и деструкторы
    constructor Create(const FileName: string; const ADelimiter: Char = ';');
    destructor Destroy; override;
    ...
  end;
```

Приведем их возможную реализацию:

```
constructor TDelimitedReader.Create(const FileName: string;
  const ADelimiter: Char = ';');
begin
  AssignFile(FileVar, FileName);
  Delimiter := ADelimiter;
end;

destructor TDelimitedReader.Destroy;
begin
  // Пока ничего не делаем
end;
```

Если объект содержит встроенные объекты или другие динамические данные, то конструктор — это как раз то место, где их нужно создавать.

Конструктор применяется к классу или к объекту. Если он применяется к классу,

```
Reader := TDelimitedReader.Create('MyData.del', ';');
```

то выполняется следующая последовательность действий:

- в динамической памяти выделяется место для нового объекта;
- выделенная память заполняется нулями. В результате все числовые поля и поля порядкового типа приобретают нулевые значения, строковые поля становятся пустыми, а поля, содержащие указатели и объекты получают значение **nil**;
- затем выполняются заданные программистом действия конструктора;
- ссылка на созданный объект возвращается в качестве значения конструктора. Тип возвращаемого значения совпадает с типом класса, использованного при вызове (в нашем примере это тип TDelimitedReader).

Если конструктор применяется к объекту,

```
Reader.Create('MyData.del', ';');
```

то конструктор выполняется как обычный метод. Другими словами, новый объект не создается, а происходит повторная инициализация полей существующего объекта. В этом случае конструктор не возвращает никакого значения. Далеко не все объекты корректно себя ведут при повторной инициализации, поскольку программисты редко закладывают такую возможность в свои классы. Поэтому на практике повторная инициализация применяется крайне редко.

Деструктор уничтожает объект, к которому применяется:

```
Reader.Destroy;
```

В результате:

- выполняется заданный программистом код завершения;
- освобождается занимаемая объектом динамическая память.

В теле деструктора обычно должны уничтожаться встроенные объекты и динамические данные, как правило, созданные конструктором.

Как и обычные методы, деструктор может иметь параметры, но эта возможность используется редко.

3.5. Методы

Процедуры и функции, предназначенные для выполнения над объектами действий, называются *методами*. Предварительное объявление методов выполняется при описании класса в секции **interface** модуля, а их программный код записывается в секции **implementation**. Однако в отличие от обычных процедур и функций заголовки методов должны иметь уточненные имена, т.е. содержать наименование класса. Приведем возможную реализацию одного из методов в классе TDelimitedReader:

```
procedure TDelimitedReader.SetActive(const AActive: Boolean);
begin
  if AActive then
    Reset(FileVar)         // Открытие файла
  else
    CloseFile(FileVar);    // Закрытие файла
end;
```

Обратите внимание, что внутри методов обращения к полям и другим методам выполняются как к обычным переменным и подпрограммам без уточнения экземпляра объекта. Такое упрощение достигается путем использования в пределах метода псевдопеременной Self (стандартный идентификатор). Физически Self представляет собой дополнительный неявный параметр, передаваемый в метод при вызове. Этот параметр и указывает экземпляр объекта,

к которому данный метод применяется. Чтобы пояснить сказанное, перепишем метод SetActive, представив его в виде обычной процедуры:

```
procedure TDelimitedReader_SetActive(Self: TDelimitedReader;
  const AActive: Boolean);
begin
  if AActive then
    Reset(Self.FileVar)          // Открытие файла
  else
    CloseFile(Self.FileVar);     // Закрытие файла
end;
```

Согласитесь, что метод SetActive выглядит лаконичнее процедуры TDelimitedReader_SetActive.

Практика показывает, что псевдопеременная Self редко используется в явном виде. Ее необходимо применять только тогда, когда при написании метода может возникнуть какая-либо двусмысленность для компилятора, например при использовании одинаковых имен и для локальных переменных, и для полей объекта.

Если выполнить метод SetActive,

```
Reader.SetActive(True);
```

то обрабатываемый файл будет открыт. При этом неявный параметр Self будет содержать значение переменной Reader. Такой вызов реализуется обычными средствами процедурного программирования приблизительно так:

```
TDelimitedReader_SetActive(Reader, True);
```

3.6. Свойства

3.6.1. Понятие свойства

Помимо полей и методов в объектах существуют *свойства*. При работе с объектом свойства выглядят как поля: они принимают значения и участвуют в выражениях. Но в отличие от полей свойства не занимают места в памяти, а операции их чтения и записи ассоциируются с обычными полями или методами. Это позволяет создавать необходимые сопутствующие эффекты при обращении к свойствам. Например, в объекте Reader присваивание свойству Active значения True вызовет открытие файла, а присваивание значения False — закрытие файла. Создание сопутствующего эффекта (открытие или закрытие файла) достигается тем, что за присваиванием свойству значения стоит вызов метода.

Объявление свойства выполняется с помощью зарезервированного слова **property**, например:

```
type
  TDelimitedReader = class
    ...
    FActive: Boolean;
    ...
    // Метод записи (установки значения) свойства
    procedure SetActive(const AActive: Boolean);
    property Active: Boolean read FActive write SetActive; // Свойство
  end;
```

Ключевые слова **read** и **write** называются спецификаторами доступа. После слова **read** указывается поле или метод, к которому происходит обращение при чтении (получении) значения свойства, а после слова **write** — поле или метод, к которому происходит обращение при записи (установке) значения свойства. Например, чтение свойства Active означает чтение поля FActive, а установка свойства — вызов метода SetActive. Чтобы имена свойств не совпадали с именами полей, последние принято писать с буквы **F** (от англ. field). Мы в дальнейшем также будем пользоваться этим соглашением. Начнем с того, что переименуем

поля класса TDelimitedReader: поле FileVar переименуем в FFile, Items — в FItems, а поле Delimiter — в FDelimiter.

```
type
  TDelimitedReader = class
    // Поля
    FFile: TextFile;          // FileVar    -> FFile
    FItems: array of string;  // Items      -> FItems
    FActive: Boolean;
    FDelimiter: Char;         // Delimiter  -> FDelimiter
    ...
  end;
```

Обращение к свойствам выглядит в программе как обращение к полям:

```
var
  Reader: TDelimitedReader;
  IsOpen: Boolean;
...
  Reader.Active := True;   // Эквивалентно Reader.SetActive(True);
  IsOpen := Reader.Active; // Эквивалентно IsOpen := Reader.FActive
```

Если один из спецификаторов доступа опущен, то значение свойства можно либо только читать (задан спецификатор **read**), либо только записывать (задан спецификатор **write**). В следующем примере объявлено свойство, значение которого можно только читать.

```
type
  TDelimitedReader = class
    ...
    FItems: array of string;
    ...
    function GetItemCount: Integer;
    ...
    property ItemCount: Integer read GetItemCount; // Только для чтения!
  end;

function TDelimitedReader.GetItemCount: Integer;
begin
  Result := Length(FItems);
end;
```

Здесь свойство ItemCount показывает количество элементов в массиве FItems. Поскольку оно определяется в результате чтения и разбора очередной строки файла, пользователю объекта разрешено лишь узнавать количество элементов.

В отличие от полей свойства не имеют адреса в памяти, поэтому к ним запрещено применять операцию @. Как следствие, их нельзя передавать в **var**- и **out**-параметрах процедур и функций.

Технология объектно-ориентированного программирования в среде Delphi предписывает избегать прямого обращения к полям, создавая вместо этого соответствующие свойства. Это упорядочивает работу с объектами, изолируя их данные от непосредственной модификации. В будущем внутренняя структура класса, которая иногда является достаточно сложной, может быть изменена с целью повышения эффективности работы программы. При этом потребуется переработать только методы чтения и записи значений свойств; внешний интерфейс класса не изменится.

3.6.2. Методы получения и установки значений свойств

Методы получения (чтения) и установки (записи) значений свойств подчиняются определенным правилам. Метод чтения свойства — это всегда функция, возвращающая значение того же типа, что и тип свойства. Метод записи свойства — это обязательно процедура, принимающая параметр того же типа, что и тип свойства. В остальных отношениях это обычные методы объекта. Примерами методов чтения и записи свойств являются методы GetItemCount и SetActive в классе TDelimitedReader:

```
type
  TDelimitedReader = class
    FActive: Boolean;
    ...
    procedure SetActive(const AActive: Boolean);
    function GetItemCount: Integer;
    ...
    property Active: Boolean read FActive write SetActive;
    property ItemCount: Integer read GetItemCount;
  end;
```

Использование методов для получения и установки свойств позволяет проверить корректность значения свойства, сделать дополнительные вычисления, установить значения зависимых полей и т.д. Например, в методе SetActive вполне целесообразно осуществить проверку состояния файла (открыт или закрыт), чтобы избежать его повторного открытия или закрытия:

```
procedure TDelimitedReader.SetActive(const AActive: Boolean);
begin
  if Active <> AActive then // Если состояние изменяется
  begin
    if AActive then
      Reset(FFile)          // Открытие файла
    else
      CloseFile(FFile);     // Закрытие файла
    FActive := AActive;     // Сохранение состояния в поле
  end;
end;
```

Наличие свойства Active позволяет нам отказаться от использования методов Open и Close, традиционных при работе с файлами. Согласитесь, что открывать и закрывать файл с помощью свойства Active гораздо удобнее и естественнее. Одновременно с этим свойство Active можно использовать и для проверки состояния файла (открыт или нет). Таким образом, для осуществления трех действий требуется всего лишь одно свойство! Это делает использование Ваших классов другими программистами более простым, поскольку им легче запомнить одно понятие Active, чем, например, три метода: Open, Close и IsOpen.

Значение свойства может не храниться, а вычисляться при каждом обращении к свойству. Примером является свойство ItemCount, значение которого вычисляется как Length(FItems).

3.6.3. Свойства-массивы

Кроме обычных свойств в объектах существуют свойства-массивы (array properties). Свойство-массив — это индексированное множество значений. Например, в классе TDelimitedReader множество элементов, выделенных из считанной строки, удобно представить в виде свойства-массива:

```
type
  TDelimitedReader = class
    ...
    FItems: array of string;
    ...
    function GetItem(Index: Integer): string;
    ...
    property Items[Index: Integer]: string read GetItem;
  end;

function TDelimitedReader.GetItem(Index: Integer): string;
begin
  Result := FItems[Index];
end;
```

Элементы массива Items можно только читать, поскольку класс TDelimitedReader предназначен только для чтения данных из файла.

В описании свойства-массива разрешено использовать только методы, но не поля. В этом состоит отличие свойства-массива от обычного свойства.

Основная выгода от применения свойства-массива — возможность выполнения итераций с помощью цикла **for**, например:

```
var
  Reader: TDelimitedReader;
  I: Integer;
...
  for I := 0 to Reader.ItemCount - 1 do
    Writeln(Reader.Items[I]);
...
```

Свойство-массив может быть многомерным. В этом случае методы чтения и записи элементов должны иметь столько же индексных параметров соответствующих типов, что и свойство-массив.

Свойства-массивы имеют два важных отличия от обычных массивов:

- их индексы не ограничиваются диапазоном и могут иметь любой тип данных, а не только Integer. Например, можно создать свойство-массив, в котором индексами будут строки. Обращение к такому свойству могло бы выглядеть примерно так:

  ```
  Reader.Items['FirstName'] := 'Alexander';
  ```

- операции целиком со всем свойством-массивом запрещены; разрешены операции только с его элементами.

3.6.4. Свойство-массив как основное свойство объекта

Свойство-массив можно сделать основным свойством объектов данного класса. Для этого в описание свойства добавляется слово **default**:

```
type
  TDelimitedReader = class
    ...
    property Items[Index: Integer]: string read GetItem; default;
    ...
  end;
```

Такое объявление свойства Items позволяет рассматривать сам объект класса TDelimitedReader как массив и опускать имя свойства-массива при обращении к нему из программы, например:

```
var
  R: TDelimitedReader;
  I: Integer;
...
  for I := 0 to R.ItemCount - 1 do
    Writeln(R[I]);
...
```

Следует помнить, что только свойства-массивы могут быть основными свойствами объектов; для обычных свойств это недопустимо.

3.6.5. Методы, обслуживающие несколько свойств

Один и тот же метод может использоваться для получения (установки) значений нескольких свойств одного типа. В этом случае каждому свойству назначается целочисленный индекс, который передается в метод чтения (записи) первым параметром.

В следующем примере уже известный Вам метод GetItem обслуживает три свойства: FirstName, LastName и Phone:

```
type
  TDelimitedReader = class
    ...
    property FirstName: string index 0 read GetItem;
    property LastName: string index 1 read GetItem;
    property Phone: string index 2 read GetItem;
  end;
```

Обращения к свойствам FirstName, LastName и Phone заменяются компилятором на вызовы одного и того же метода GetItem, но с разными значениями параметра Index:

```
var
  Reader: TDelimitedReader;
...
  Writeln(Reader.FirstName); // Эквивалентно: Writeln(Reader.GetItem(0));
  Writeln(Reader.LastName);  // Эквивалентно: Writeln(Reader.GetItem(1));
  Writeln(Reader.Phone);     // Эквивалентно: Writeln(Reader.GetItem(2));
...
```

Обратите внимание, что метод GetItem обслуживает как свойство-массив Items, так и свойства FirstName, LastName и Phone. Удобно, не правда ли!

Перед тем, как перейти к более сложным понятиям ООП, приведем полную реализацию класса TDelimitedReader. Настоятельно рекомендуем Вам внимательно ознакомиться с этой реализацией, поскольку в ней сведено воедино все то, о чем говорилось в предыдущих разделах.

```
type
  TDelimitedReader = class
    // Поля
    FFile: TextFile;
    FItems: array of string;
    FActive: Boolean;
    FDelimiter: Char;
    // Методы чтения и записи свойств
    procedure SetActive(const AActive: Boolean);
    function GetItemCount: Integer;
    function GetEndOfFile: Boolean;
    function GetItem(Index: Integer): string;
    // Методы
    procedure PutItem(Index: Integer; const Item: string);
    function ParseLine(const Line: string): Integer;
    function NextLine: Boolean;
    // Конструкторы и деструкторы
    constructor Create(const FileName: string; const ADelimiter: Char = ';');
    destructor Destroy; override;
    // Свойства
    property Active: Boolean read FActive write SetActive;
    property Items[Index: Integer]: string read GetItem; default;
    property ItemCount: Integer read GetItemCount;
    property EndOfFile: Boolean read GetEndOfFile;
    property Delimiter: Char read FDelimiter;
  end;

{ TDelimitedReader }

constructor TDelimitedReader.Create(const FileName: string;
  const ADelimiter: Char = ';');
begin
  AssignFile(FFile, FileName);
  FActive := False;
  FDelimiter := ADelimiter;
end;

destructor TDelimitedReader.Destroy;
begin
  Active := False;
end;

function TDelimitedReader.GetEndOfFile: Boolean;
begin
  Result := Eof(FFile);
end;

function TDelimitedReader.GetItem(Index: Integer): string;
begin
  Result := FItems[Index];
end;

function TDelimitedReader.GetItemCount: Integer;
begin
  Result := Length(FItems);
end;

function TDelimitedReader.NextLine: Boolean;
var
  S: string;
  N: Integer;
begin
  Result := not EndOfFile;
  if Result then              // Если не достигнут конец файла
  begin
    Readln(FFile, S);         // Чтение очередной строки из файла
    N := ParseLine(S);        // Разбор считанной строки
    if N <> ItemCount then
      SetLength(FItems, N);   // Отсечение массива (если необходимо)
  end;
end;
```

```
function TDelimitedReader.ParseLine(const Line: string): Integer;
var
  S: string;
  P: Integer;
begin
  S := Line;
  Result := 0;
  repeat
    P := Pos(Delimiter, S);  // Поиск разделителя
    if P = 0 then            // Если разделитель не найден, то считается, что
      P := Length(S) + 1;    // разделитель находится за последним символом
    PutItem(Result, Copy(S, 1, P - 1)); // Установка элемента
    Delete(S, 1, P);                    // Удаление элемента из строки
    Result := Result + 1;               // Переход к следующему элементу
  until S = '';                         // Пока в строке есть символы
end;

procedure TDelimitedReader.PutItem(Index: Integer; const Item: string);
begin
  if Index > High(FItems) then    // Если индекс выходит за границы массива,
    SetLength(FItems, Index + 1); // то увеличение размера массива
  FItems[Index] := Item;          // Установка соответствующего элемента
end;

procedure TDelimitedReader.SetActive(const AActive: Boolean);
begin
  if Active <> AActive then // Если состояние изменяется
  begin
    if AActive then
      Reset(FFile)          // Открытие файла
    else
      CloseFile(FFile);     // Закрытие файла
    FActive := AActive;     // Сохранение состояния в поле
  end;
end;
```

3.7. Наследование

3.7.1. Понятие наследования

Классы инкапсулируют (т.е. включают в себя) поля, методы и свойства; это их первая черта. Следующая не менее важная черта классов — способность *наследовать* поля, методы и свойства других классов. Чтобы пояснить сущность наследования обратимся к примеру с читателем текстовых файлов в формате "delimited text".

Класс TDelimitedReader описывает объекты для чтения из текстового файла элементов, разделенных некоторым символом. Он не пригоден для чтения элементов, хранящихся в другом формате, например в формате с фиксированным количеством символов для каждого элемента. Для этого необходим другой класс:

```
type
  TFixedReader = class
  private
    // Поля
    FFile: TextFile;
    FItems: array of string;
    FActive: Boolean;
    FItemWidths: array of Integer;
    // Методы чтения и записи свойств
    procedure SetActive(const AActive: Boolean);
    function GetItemCount: Integer;
    function GetEndOfFile: Boolean;
    function GetItem(Index: Integer): string;
    // Методы
    procedure PutItem(Index: Integer; const Item: string);
    function ParseLine(const Line: string): Integer;
    function NextLine: Boolean;
    // Конструкторы и деструкторы
    constructor Create(const FileName: string;
      const AItemWidths: array of Integer);
    destructor Destroy; override;
    // Свойства
    property Active: Boolean read FActive write SetActive;
    property Items[Index: Integer]: string read GetItem; default;
    property ItemCount: Integer read GetItemCount;
    property EndOfFile: Boolean read GetEndOfFile;
  end;

{ TFixedReader }

constructor TFixedReader.Create(const FileName: string;
  const AItemWidths: array of Integer);
var
  I: Integer;
begin
  AssignFile(FFile, FileName);
  FActive := False;
  // Копирование AItemWidths в FItemWidths
  SetLength(FItemWidths, Length(AItemWidths));
  for I := 0 to High(AItemWidths) do
    FItemWidths[I] := AItemWidths[I];
end;

destructor TFixedReader.Destroy;
begin
  Active := False;
end;

function TFixedReader.GetEndOfFile: Boolean;
begin
  Result := Eof(FFile);
end;

function TFixedReader.GetItem(Index: Integer): string;
begin
  Result := FItems[Index];
end;

function TFixedReader.GetItemCount: Integer;
begin
  Result := Length(FItems);
end;

function TFixedReader.NextLine: Boolean;
var
  S: string;
  N: Integer;
begin
  Result := not EndOfFile;
  if Result then             // Если не достигнут конец файла
  begin
```

```
    Readln(FFile, S);         // Чтение очередной строки из файла
    N := ParseLine(S);        // Разбор считанной строки
    if N <> ItemCount then
      SetLength(FItems, N);   // Отсечение массива (если необходимо)
  end;
end;

function TFixedReader.ParseLine(const Line: string): Integer;
var
  I, P: Integer;
begin
  P := 1;
  for I := 0 to High(FItemWidths) do
  begin
    PutItem(I, Copy(Line, P, FItemWidths[I])); // Установка элемента
    P := P + FItemWidths[I];                   // Переход к следующему элементу
  end;
  Result := Length(FItemWidths); // Количество элементов постоянно
end;

procedure TFixedReader.PutItem(Index: Integer; const Item: string);
begin
  if Index > High(FItems) then    // Если индекс выходит за границы массива,
    SetLength(FItems, Index + 1); // то увеличение размера массива
  FItems[Index] := Item;          // Установка соответствующего элемента
end;

procedure TFixedReader.SetActive(const AActive: Boolean);
begin
  if Active <> AActive then // Если состояние изменяется
  begin
    if AActive then
      Reset(FFile)          // Открытие файла
    else
      CloseFile(FFile);     // Закрытие файла
    FActive := AActive;     // Сохранение состояния в поле
  end;
end;
```

Поля, свойства и методы класса TFixedReader практически полностью аналогичны тем, что определены в классе TDelimitedReader. Отличие состоит в отсутствии свойства Delimiter, наличии поля FItemWidths (для хранения размеров элементов), другой реализации метода ParseLine и немного отличающемся конструкторе. Если в будущем появится класс для чтения элементов из файла еще одного формата (например, зашифрованного текста), то придется снова определять общие для всех классов поля, методы и свойства. Чтобы избавиться от дублирования общих атрибутов (полей, свойств и методов) при определении новых классов, воспользуемся механизмом наследования. Прежде всего, выделим в отдельный класс TTextReader общие атрибуты всех классов, предназначенных для чтения элементов из текстовых файлов. Реализация методов TTextReader, кроме метода ParseLine, полностью идентична реализации TDelimitedReader, приведенной в предыдущем разделе.

```
type
  TTextReader = class
  private
    // Поля
    FFile: TextFile;
    FItems: array of string;
    FActive: Boolean;
    // Методы получения и установки значений свойств
    procedure SetActive(const AActive: Boolean);
    function GetItemCount: Integer;
    function GetItem(Index: Integer): string;
    function GetEndOfFile: Boolean;
    // Методы
    procedure PutItem(Index: Integer; const Item: string);
    function ParseLine(const Line: string): Integer;
    function NextLine: Boolean;
    // Конструкторы и деструкторы
    constructor Create(const FileName: string);
    destructor Destroy; override;
    // Свойства
    property Active: Boolean read FActive write SetActive;
    property Items[Index: Integer]: string read GetItem; default;
    property ItemCount: Integer read GetItemCount;
    property EndOfFile: Boolean read GetEndOfFile;
  end;
...
constructor TTextReader.Create(const FileName: string);
begin
  AssignFile(FFile, FileName);
  FActive := False;
end;

function TTextReader.ParseLine(const Line: string): Integer;
begin
  // Функция просто возвращает 0, поскольку не известно,
  // в каком именно формате хранятся элементы
  Result := 0;
end;
...
```

При реализации класса TTextReader ничего не известно о том, как хранятся элементы в считываемых строках, поэтому метод ParseLine ничего не делает. Очевидно, что создавать объекты класса TTextReader не имеет смысла. Для чего тогда нужен класс TTextReader? Ответ: чтобы на его основе определить (*породить*) два других класса — TDelimitedReader и TFixedReader, предназначенных для чтения данных в конкретных форматах:

```
type
  TDelimitedReader = class(TTextReader)
    FDelimiter: Char;
    function ParseLine(const Line: string): Integer; override;
    constructor Create(const FileName: string; const ADelimiter: Char = ';');
    property Delimiter: Char read FDelimiter;
  end;

  TFixedReader = class(TTextReader)
    FItemWidths: array of Integer;
    function ParseLine(const Line: string): Integer; override;
    constructor Create(const FileName: string;
      const AItemWidths: array of Integer);
  end;
...
```

Классы TDelimitedReader и TFixedReader определены как *наследники* TTextReader (об этом говорит имя в скобках после слова **class**). Они автоматически включают в себя все описания, сделанные в классе TTextReader и добавляют к ним некоторые новые. В результате формируется *дерево классов*, показанное на рисунке 3.1 (оно всегда рисуется перевернутым).

Рисунок 3.1. Дерево классов

Класс, который наследует атрибуты другого класса, называется порожденным классом или *потомком*. Соответственно класс, от которого происходит наследование, выступает в роли базового, или *предка*. В нашем примере класс TDelimitedReader является прямым потомком класса TTextReader. Если от TDelimitedReader породить новый класс, то он тоже будет потомком класса TTextReader, но уже не прямым.

Очень важно, что в отношениях наследования любой класс может иметь только одного непосредственного предка и сколь угодно много потомков. Поэтому все связанные отношением наследования классы образуют *иерархию*. Примером иерархии классов является библиотека VCL; с ее помощью в среде Delphi обеспечивается разработка GUI-приложений.

3.7.2. Прародитель всех классов

В языке Delphi существует предопределенный класс TObject, который служит неявным предком тех классов, для которых предок не указан. Это означает, что объявление

```
type
  TTextReader = class
    ...
  end;
```

эквивалентно следующему:

```
type
  TTextReader = class(TObject)
    ...
  end;
```

Класс TObject выступает корнем любой иерархии классов. Он содержит ряд методов, которые по наследству передаются всем остальным классам. Среди них конструктор Create, деструктор Destroy, метод Free и некоторые другие методы.

Таким образом, полное дерево классов для чтения элементов из текстового файла в различных форматах выглядит так, как показано на рисунке 3.2.

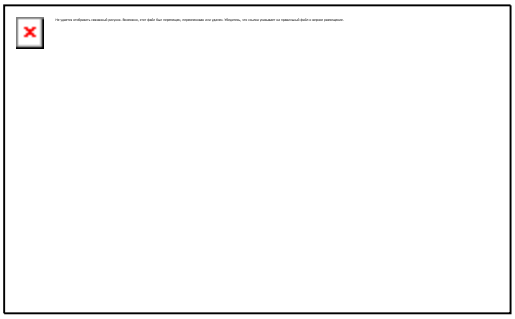


Рисунок 3.2. Полное дерево классов

Поскольку класс TObject является предком для всех других классов (в том числе и для ваших собственных), то не лишним будет кратко ознакомиться с его методами:

```
type
  TObject = class
    constructor Create;
    procedure Free;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass;
    class function ClassName: ShortString;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer;
    class function InstanceSize: Longint;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: ShortString): Pointer;
    class function MethodName(Address: Pointer): ShortString;
    function FieldAddress(const Name: ShortString): Pointer;
    function GetInterface(const IID: TGUID; out Obj): Boolean;
    class function GetInterfaceEntry(const IID: TGUID): PInterfaceEntry;
    class function GetInterfaceTable: PInterfaceTable;
    function SafeCallException(ExceptObject: TObject;
      ExceptAddr: Pointer): HResult; virtual;
    procedure AfterConstruction; virtual;
    procedure BeforeDestruction; virtual;
    procedure Dispatch(var Message); virtual;
    procedure DefaultHandler(var Message); virtual;
    class function NewInstance: TObject; virtual;
    procedure FreeInstance; virtual;
    destructor Destroy; virtual;
  end;
```

Некоторые конструкции этого описания будут вам непонятны, поскольку мы их еще не изучали. Сейчас это не важно. Снова вернитесь к этому описанию после прочтения всей главы.

Краткое описание методов в классе TObject:

Create — стандартный конструктор.

Free — уничтожает объект: вызывает стандартный деструктор Destroy, если значение псевдопеременной Self не равно **nil**.

InitInstance(Instance: Pointer): TObject — при создании объекта инициализирует нулями выделенную память. На практике нет необходимости вызывать этот метод явно.

CleanupInstance — освобождает память, занимаемую полями с типом string, Variant, динамический массив и интерфейс. На практике нет необходимости вызывать этот метод явно.

ClassType: TClass — возвращает описатель класса (метакласс).

ClassName: ShortString — возвращает имя класса.

ClassNameIs(const Name: string): Boolean — проверяет, является ли заданная строка именем класса.

ClassParent: TClass — возвращает описатель базового класса.

ClassInfo: Pointer — возвращает указатель на соответствующую классу таблицу RTTI (от англ. Runtime Type Information). Таблица RTTI используется для проверки типов данных на этапе выполнения программы.

InstanceSize: Longint — возвращает количество байт, необходимых для хранения в памяти одного объекта соответствующего класса. Заметим, что значение, возвращаемое этим методом и значение, возвращаемое функцией SizeOf при передаче ей в качестве аргумента объектной переменной — это разные значения. Функция SizeOf всегда возвращает значение 4 (SizeOf(Pointer)), поскольку объектная переменная — это ни что иное, как ссылка на

данные объекта в памяти. Значение InstanceSize — это размер этих данных, а не размер объектной переменной.

InheritsFrom(AClass: TClass): Boolean — проверяет, является ли класс AClass базовым классом.

MethodAddress(const Name: ShortString): Pointer — возвращает адрес published-метода, имя которого задается параметром Name.

MethodName(Address: Pointer): ShortString — возвращает имя published-метода по заданному адресу.

FieldAddress(const Name: ShortString): Pointer — возвращает адрес published-поля, имя которого задается параметром Name.

GetInterface(const IID: TGUID; out Obj): Boolean — возвращает ссылку на интерфейс через параметр Obj; идентификатор интерфейса задается параметром IID. (Интерфейсы рассмотрены в главе 6)

GetInterfaceEntry(const IID: TGUID): PInterfaceEntry — возвращает информацию об интерфейсе, который реализуется классом. Идентификатор интерфейса задается параметром IID.

GetInterfaceTable: PInterfaceTable — возвращает указатель на таблицу с информацией обо всех интерфейсах, реализуемых классом.

AfterConstruction — автоматически вызывается после создания объекта. Метод не предназначен для явного вызова из программы. Используется для того, чтобы выполнить определенные действия уже после создания объекта (для этого его необходимо переопределить в производных классах).

BeforeDestruction — автоматически вызывается перед уничтожением объекта. Метод не предназначен для явного вызова из программы. Используется для того, чтобы выполнить определенные действия непосредственно перед уничтожением объекта (для этого его необходимо переопределить в производных классах).

Dispatch(var Message) — служит для вызова методов, объявленных с ключевым словом **message**.

DefaultHandler(var Message) — вызывается методом **Dispatch** в том случае, если метод, соответствующий сообщению Message, не был найден.

NewInstance: TObject — вызывается при создании объекта для выделения динамической памяти, чтобы разместить в ней данные объекта. Метод вызывается автоматически, поэтому нет необходимости вызывать его явно.

FreeInstance — вызывается при уничтожении объекта для освобождения занятой объектом динамической памяти. Метод вызывается автоматически, поэтому нет необходимости вызывать его явно.

Destroy — стандартный деструктор.

3.7.3. Перекрытие атрибутов в наследниках

В механизме наследования можно условно выделить три основных момента:

- наследование полей;
- наследование свойств;
- наследование методов.

Любой порожденный класс наследует от родительского все поля данных, поэтому классы TDelimitedReader и TFixedReader автоматически содержат поля FFile, FActive и FItems, объявленные в классе TTextReader. Доступ к полям предка осуществляется по имени, как если бы они были определены в потомке. В потомках можно определять новые поля, но их имена должны отличаться от имен полей предка.

Наследование свойств и методов имеет свои особенности.

Свойство базового класса можно *перекрыть* (от англ. override) в производном классе, например чтобы добавить ему новый атрибут доступа или связать с другим полем или методом.

Метод базового класса тоже можно перекрыть в производном классе, например чтобы изменить логику его работы. Обратимся к классам TDelimitedReader и TFixedReader. В них методы PutItem, GetItem, SetActive и GetEndOfFile унаследованы от TTextReader, поскольку логика их работы не зависит от того, в каком формате хранятся данные в файле. А вот метод ParseLine перекрыт, так как способ разбора строк зависит от формата данных:

```
function TDelimitedReader.ParseLine(const Line: string): Integer;
var
  S: string;
  P: Integer;
begin
  S := Line;
  Result := 0;
  repeat
    P := Pos(Delimiter, S);  // Поиск разделителя
    if P = 0 then            // Если разделитель не найден, то считается, что
      P := Length(S) + 1;    // разделитель находится за последним символом
    PutItem(Result, Copy(S, 1, P - 1)); // Установка элемента
    Delete(S, 1, P);                    // Удаление элемента из строки
    Result := Result + 1;               // Переход к следующему элементу
  until S = '';                         // Пока в строке есть символы
end;

function TFixedReader.ParseLine(const Line: string): Integer;
var
  I, P: Integer;
begin
  P := 1;
  for I := 0 to High(FItemWidths) do
  begin
    PutItem(I, Copy(Line, P, FItemWidths[I])); // Установка элемента
    P := P + FItemWidths[I];                   // Переход к следующему элементу
  end;
  Result := Length(FItemWidths); // Количество элементов постоянно
end;
```

В классах TDelimitedReader и TFixedReader перекрыт еще и конструктор Create. Это необходимо для инициализации специфических полей этих классов (поля FDelimiter в классе TDelimitedReader и поля FItemWidths в классе TFixedReader):

```
constructor TDelimitedReader.Create(const FileName: string;
  const ADelimiter: Char = ';');
begin
  inherited Create(FileName);
  FDelimiter := ADelimiter;
end;

constructor TFixedReader.Create(const FileName: string;
  const AItemWidths: array of Integer);
var
  I: Integer;
begin
  inherited Create(FileName);
  // Копирование AItemWidths в FItemWidths
  SetLength(FItemWidths, Length(AItemWidths));
  for I := 0 to High(AItemWidths) do
    FItemWidths[I] := AItemWidths[I];
end;
```

Как видно из примера, в наследнике можно вызвать перекрытый метод предка, указав перед именем метода зарезервированное слово **inherited**. Когда метод предка полностью совпадает с методом потомка по формату заголовка, то можно использовать более короткую запись. Воспользуемся ей и перепишем деструктор в классе TTextReader правильно:

```
destructor TTextReader.Destroy;
begin
  Active := False;
  inherited; // Эквивалентно: inherited Destroy;
end;
```

Два последних примера демонстрируют важный принцип реализации конструкторов и деструкторов. В конструкторах сначала вызывается конструктор предка, а затем инициализируются дополнительные поля данных. В деструкторах применяется обратная последовательность действий: сначала разрушаются данные, недоступные предку, а затем вызывается унаследованный деструктор. Всегда пользуйтесь этими правилами в своих программах, чтобы избежать ошибок.

3.7.4. Совместимость объектов различных классов

Для классов, связанных отношением наследования, вводится новое правило совместимости типов. Вместо объекта базового класса можно подставить объект любого производного класса. Обратное неверно. Например, переменной типа TTextReader можно присвоить значение переменной типа TDelimitedReader:

```
var
  Reader: TTextReader;
...
  Reader := TDelimitedReader.Create('MyData.del', ';');
```

Объектная переменная Reader *формально* имеет тип TTextReader, а *фактически* связана с экземпляром класса TDelimitedReader.

Правило совместимости классов чаще всего применяется при передаче объектов в параметрах процедур и функций. Например, если процедура работает с объектом класса TTextReader, то вместо него можно передать объект класса TDelimitedReader или TFixedReader.

Заметим, что все объекты являются представителями известного вам класса TObject. Поэтому любой объект любого класса можно использовать как объект класса TObject.

3.7.5. Контроль и преобразование типов

Поскольку реальный экземпляр объекта может оказаться наследником класса, указанного при описании объектной переменной или параметра, бывает необходимо проверить, к какому классу принадлежит объект на самом деле. Чтобы программист мог выполнять

такого рода проверки, каждый объект хранит информацию о своем классе. В языке Delphi существуют операторы **is** и **as**, с помощью которых выполняется соответственно *проверка на тип* (type checking) и *преобразование к типу* (type casting).

Например, чтобы выяснить, принадлежит ли некоторый объект Obj к классу TTextReader или его наследнику, следует использовать оператор **is**:

```
var
  Obj: TObject;
...
  if Obj is TTextReader then ...
```

Для преобразования объекта к нужному типу используется оператор **as**, например

```
  with Obj as TTextReader do
    Active := False;
```

Стоит отметить, что для объектов применим и обычный способ приведения типа:

```
  with TTextReader(Obj) do
    Active := False;
```

Вариант с оператором **as** лучше, поскольку безопасен. Он генерирует ошибку (точнее исключительную ситуацию; об исключительных ситуациях мы расскажем в главе 4) при выполнении программы (run-time error), если реальный экземпляр объекта Obj не совместим с классом TTextReader. Забегая вперед, скажем, что ошибку приведения типа можно обработать и таким образом избежать досрочного завершения программы.

3.8. Виртуальные методы

3.8.1. Понятие виртуального метода

Все методы, которые до сих пор рассматривались, имеют одну общую черту — все они *статические*. При обращении к статическому методу компилятор точно знает класс, которому данный метод принадлежит. Поэтому, например, обращение к статическому методу ParseLine в методе NextLine (принадлежащем классу TTextReader) компилируется в вызов TTextReader.ParseLine:

```
function TTextReader.NextLine: Boolean;
var
  S: string;
  N: Integer;
begin
  Result := not EndOfFile;
  if Result then
  begin
    Readln(FFile, S);
    N := ParseLine(S); // Компилируется в вызов TTextReader.ParseLine(S);
    if N <> ItemCount then
      SetLength(FItems, N);
  end;
end;
```

В результате метод NextLine работает неправильно в наследниках класса TTextReader, так как внутри него вызов перекрытого метода ParseLine не происходит. Конечно, в классах TDelimitedReader и TFixedReader можно продублировать все методы и свойства, которые прямо или косвенно вызывают ParseLine, но при этом теряются преимущества наследования, и мы возвращаемся к тому, что необходимо описать два класса, в которых большая часть кода идентична. ООП предлагает изящное решение этой проблемы — метод ParseLine всего-навсего объявляется *виртуальным*:

```
type
  TTextReader = class
    ...
    function ParseLine(const Line: string): Integer; virtual; //Виртуальный метод
    ...
  end;
```

Объявление виртуального метода в базовом классе выполняется с помощью ключевого слова **virtual**, а его перекрытие в производных классах — с помощью ключевого слова **override**. Перекрытый метод должен иметь точно такой же формат (список параметров, а для функций еще и тип возвращаемого значения), что и перекрываемый:

```
type
  TDelimitedReader = class(TTextReader)
    ...
    function ParseLine(const Line: string): Integer; override;
    ...
  end;

  TFixedReader = class(TTextReader)
    ...
    function ParseLine(const Line: string): Integer; override;
    ...
  end;
```

Суть виртуальных методов в том, что они вызываются по фактическому типу экземпляра, а не по формальному типу, записанному в программе. Поэтому после сделанных изменений метод NextLine будет работать так, как ожидает программист:

```
function TTextReader.NextLine: Boolean;
var
  S: string;
  N: Integer;
begin
  Result := not EndOfFile;
  if Result then
  begin
    Readln(FFile, S);
    N := ParseLine(S); // Работает как <фактический класс>.ParseLine(S)
    if N <> ItemCount then
      SetLength(FItems, N);
  end;
end;
```

Работа виртуальных методов основана на механизме позднего связывания (late binding). В отличие от *раннего связывания* (early binding), характерного для статических методов, *позднее связывание* основано на вычислении адреса вызываемого метода при выполнении программы. Адрес метода вычисляется по хранящемуся в каждом объекте описателю класса.

Благодаря механизму наследования и виртуальных методов в среде Delphi реализуется такая концепция ООП как полиморфизм. Полиморфизм существенно облегчает труд программиста, поскольку обеспечивает повторное использование кода уже написанных и отлаженных методов.

3.8.2. Механизм вызова виртуальных методов

Работа виртуальных методов основана на косвенном вызове подпрограмм. При косвенном вызове команда вызова подпрограммы оперирует не адресом подпрограммы, а адресом места в памяти, где хранится адрес подпрограммы. Вы уже сталкивались с косвенным вызовом при использовании процедурных переменных. Процедурная переменная и была тем местом в памяти, где хранился адрес вызываемой подпрограммы. Для каждого виртуального метода тоже создается процедурная переменная, но ее наличие и использование скрыто от программиста.

Все процедурные переменные с адресами виртуальных методов пронумерованы и хранятся в таблице, называемой таблицей виртуальных методов (VMT — от англ. Virtual Method Table).

Такая таблица создается одна для каждого класса объектов, и все объекты этого класса хранят на нее ссылку.

Структуру объекта в оперативной памяти поясняет рисунок 3.3:

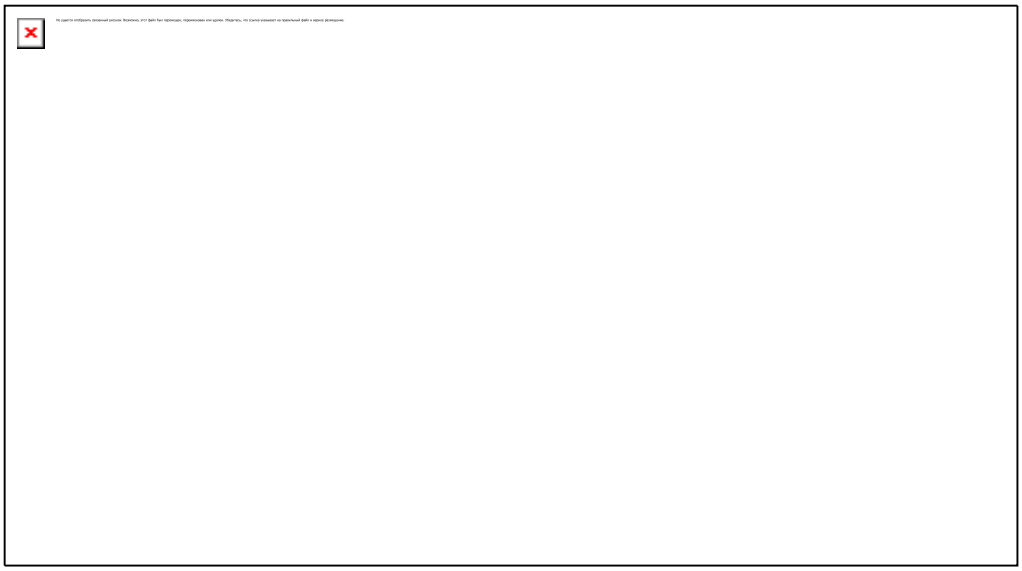


Рисунок 3.3. Структура объекта TTextReader в оперативной памяти

Вызов виртуального метода осуществляется следующим образом:

1. Через объектную переменную выполняется обращение к занятому объектом блоку памяти;
2. Далее из этого блока извлекается адрес таблицы виртуальных методов (он записан в четырех первых байтах);
3. На основании порядкового номера виртуального метода извлекается адрес соответствующей подпрограммы;
4. Вызывается код, находящийся по этому адресу.

Покажем, как можно реализовать косвенный вызов виртуального метода ParseLine (он имеет нулевой номер в таблице виртуальных методов) обычными средствами процедурного программирования:

```
type
  TVMT = array[0..9999] of Pointer;
  TParseLineFunc = function (Self: TTextReader; const Line: string): Integer;
var
  Reader: TTextReader;     // объектная переменна
  ObjectDataPtr: Pointer; // указатель на занимаемый объектом блок памяти
  VMTPtr: ^TVMT;          // указатель на таблицу виртуальных методов
  MethodPtr: Pointer;     // указатель на метод
begin
  ...
  ObjectDataPtr := Pointer(Reader);      // 1) обращение к данным объекта
  VMTPtr := Pointer(ObjectDataPtr^);     // 2) извлечение адреса VMT
  MethodPtr := VMTPtr^[0];               // 3) извлечение адреса метода из VMT
  TParseLineFunc(MethodPtr)(Reader, S);  // 4) вызов метода
  ...
end.
```

Поддержка механизма вызова виртуальных методов на уровне языка Delphi избавляет программиста от всей этой сложности.

3.8.3. Абстрактные виртуальные методы

При построении иерархии классов часто возникает ситуация, когда работа виртуального метода в базовом классе не известна и наполняется содержанием только в наследниках. Так случилось, например, с методом ParseLine, тело которого в классе TTextReader объявлено пустым. Конечно, тело метода всегда можно сделать пустым или почти пустым (так мы и поступили), но лучше воспользоваться директивой **abstract**:

```
type
  TTextReader = class
    ...
    function ParseLine(const Line: string): Integer; virtual; abstract;
    ...
  end;
```

Директива **abstract** записывается после слова **virtual** и исключает необходимость написания кода виртуального метода для данного класса. Такой метод называется *абстрактным*, т.е. подразумевает логическое действие, а не конкретный способ его реализации. Абстрактные виртуальные методы часто используются при создании классов-полуфабрикатов. Свою реализацию такие методы получают в законченных наследниках.

3.8.4. Динамические методы

Разновидностью виртуальных методов являются так называемые *динамические методы*. При их объявлении вместо ключевого слова **virtual** записывается ключевое слово **dynamic**, например:

```
type
  TTextReader = class
    ...
    function ParseLine(const Line: string): Integer; dynamic; abstract;
    ...
  end;
```

В наследниках динамические методы перекрываются так же, как и виртуальные — с помощью зарезервированного слова **override**.

По смыслу динамические и виртуальные методы идентичны. Различие состоит только в механизме их вызова. Методы, объявленные с директивой **virtual**, вызываются максимально быстро, но платой за это является большой размер системных таблиц, с помощью которых определяются их адреса. Размер этих таблиц начинает сказываться с увеличением числа классов в иерархии. Методы, объявленные с директивой **dynamic** вызываются несколько дольше, но при этом таблицы с адресами методов имеют более компактный вид, что способствует экономии памяти. Таким образом, программисту предоставляются два способа оптимизации объектов: по скорости работы (**virtual**) или по объему памяти (**dynamic**).

3.8.5. Методы обработки сообщений

Специализированной формой динамических методов являются *методы обработки сообщений*. Они объявляются с помощью ключевого слова **message**, за которым следует целочисленная константа — *номер сообщения*. Следующий пример взят из исходных текстов библиотеки VCL:

```
type
  TWidgetControl = class(TControl)
    ...
    procedure CMKeyDown(var Msg: TCMKeyDown); message CM_KEYDOWN;
    ...
  end;
```

Метод обработки сообщений имеет формат процедуры и содержит единственный **var**-параметр. При перекрытии такого метода название метода и имя параметра могут быть любыми, важно лишь, чтобы неизменным остался номер сообщения, используемый для вызова метода. Вызов метода выполняется не по имени, как обычно, а с помощью обращения

к специальному методу Dispatch, который имеется в каждом классе (метод Dispatch определен в классе TObject).

Методы обработки сообщений применяются внутри библиотеки VCL для обработки команд пользовательского интерфейса и редко нужны при написании прикладных программ.

3.9. Классы в программных модулях

Классы очень удобно собирать в модули. При этом их описание помещается в секцию **interface**, а код методов — в секцию **implementation**. Создавая модули классов, нужно придерживаться следующих правил:

- все классы, предназначенные для использования за пределами модуля, следует определять в секции **interface**;
- описание классов, предназначенных для употребления внутри модуля, следует располагать в секции **implementation**;
- если модуль B использует модуль A, то в модуле B можно определять классы, порожденные от классов модуля A.

Соберем рассмотренные ранее классы TTextReader, TDelimitedReader и TFixedReader в отдельный модуль ReadersUnit:

```
unit ReadersUnit;

interface

type
  TTextReader = class
  private
    // Поля
    FFile: TextFile;
    FItems: array of string;
    FActive: Boolean;
    // Методы
    procedure PutItem(Index: Integer; const Item: string);
    // Методы чтения и записи свойств
    procedure SetActive(const AActive: Boolean);
    function GetItemCount: Integer;
    function GetEndOfFile: Boolean;
  protected
    // Методы чтения и записи свойств
    function GetItem(Index: Integer): string;
    // Абстрактные методы
    function ParseLine(const Line: string): Integer; virtual; abstract;
  public
    // Конструкторы и деструкторы
    constructor Create(const FileName: string);
    destructor Destroy; override;
    // Методы
    function NextLine: Boolean;
    // Свойства
    property Active: Boolean read FActive write SetActive;
    property Items[Index: Integer]: string read GetItem; default;
    property ItemCount: Integer read GetItemCount;
    property EndOfFile: Boolean read GetEndOfFile;
  end;

  TDelimitedReader = class(TTextReader)
  private
    // Поля
    FDelimiter: Char;
  protected
    // Методы
    function ParseLine(const Line: string): Integer; override;
  public
    // Конструкторы и деструкторы
    constructor Create(const FileName: string; const ADelimiter: Char = ';');
    // Свойства
    property Delimiter: Char read FDelimiter;
  end;

  TFixedReader = class(TTextReader)
  private
    // Поля
    FItemWidths: array of Integer;
  protected
    // Методы
    function ParseLine(const Line: string): Integer; override;
  public
    // Конструкторы и деструкторы
    constructor Create(const FileName: string;
      const AItemWidths: array of Integer);
  end;

  TMyReader = class(TDelimitedReader)
    property FirstName: string index 0 read GetItem;
    property LastName: string index 1 read GetItem;
    property Phone: string index 2 read GetItem;
  end;

implementation

{ TTextReader }
```

```
constructor TTextReader.Create(const FileName: string);
begin
  inherited Create;
  AssignFile(FFile, FileName);
  FActive := False;
end;

destructor TTextReader.Destroy;
begin
  Active := False;
  inherited;
end;

function TTextReader.GetEndOfFile: Boolean;
begin
  Result := Eof(FFile);
end;

function TTextReader.GetItem(Index: Integer): string;
begin
  Result := FItems[Index];
end;

function TTextReader.GetItemCount: Integer;
begin
  Result := Length(FItems);
end;

function TTextReader.NextLine: Boolean;
var
  S: string;
  N: Integer;
begin
  Result := not EndOfFile;
  if Result then              // Если не достигнут конец файла
  begin
    Readln(FFile, S);         // Чтение очередной строки из файла
    N := ParseLine(S);        // Разбор считанной строки
    if N <> ItemCount then
      SetLength(FItems, N);   // Отсечение массива (если необходимо)
  end;
end;

procedure TTextReader.PutItem(Index: Integer; const Item: string);
begin
  if Index > High(FItems) then    // Если индекс выходит за границы массива,
    SetLength(FItems, Index + 1); // то увеличение размера массива
  FItems[Index] := Item;          // Установка соответствующего элемента
end;

procedure TTextReader.SetActive(const AActive: Boolean);
begin
  if Active <> AActive then // Если состояние изменяется
  begin
    if AActive then
      Reset(FFile)          // Открытие файла
    else
      CloseFile(FFile);     // Закрытие файла
    FActive := AActive;     // Сохранение состояния в поле
  end;
end;

{ TDelimitedReader }

constructor TDelimitedReader.Create(const FileName: string;
  const ADelimiter: Char = ';');
begin
  inherited Create(FileName);
  FDelimiter := ADelimiter;
end;
```

```
function TDelimitedReader.ParseLine(const Line: string): Integer;
var
  S: string;
  P: Integer;
begin
  S := Line;
  Result := 0;
  repeat
    P := Pos(Delimiter, S);  // Поиск разделителя
    if P = 0 then            // Если разделитель не найден, то считается, что
      P := Length(S) + 1;    // разделитель находится за последним символом
    PutItem(Result, Copy(S, 1, P - 1)); // Установка элемента
    Delete(S, 1, P);                    // Удаление элемента из строки
    Result := Result + 1;               // Переход к следующему элементу
  until S = '';                         // Пока в строке есть символы
end;

{ TFixedReader }

constructor TFixedReader.Create(const FileName: string;
  const AItemWidths: array of Integer);
var
  I: Integer;
begin
  inherited Create(FileName);
  // Копирование AItemWidths в FItemWidths
  SetLength(FItemWidths, Length(AItemWidths));
  for I := 0 to High(AItemWidths) do
    FItemWidths[I] := AItemWidths[I];
end;

function TFixedReader.ParseLine(const Line: string): Integer;
var
  I, P: Integer;
begin
  P := 1;
  for I := 0 to High(FItemWidths) do
  begin
    PutItem(I, Copy(Line, P, FItemWidths[I])); // Установка элемента
    P := P + FItemWidths[I];                   // Переход к следующему элементу
  end;
  Result := Length(FItemWidths); // Количество элементов постоянно
end;

end.
```

Как можно заметить, в описании классов присутствуют новые ключевые слова **private**, **protected** и **public**. С их помощью регулируется видимость частей класса для других модулей и основной программы. Назначение каждого ключевого слова поясняется ниже.

3.10. Разграничение доступа к атрибутам объектов

Программист может разграничить доступ к атрибутам своих объектов для других программистов (и себя самого) с помощью специальных ключевых слов: **private**, **protected**, **public**, **published** (последнее не используется в модуле ReadersUnit).

- **Private**. Все, что объявлено в секции **private** недоступно за пределами модуля. Секция **private** позволяет скрыть те поля и методы, которые относятся к так называемым особеностям реализации. Например, в этой секции класса TTextReader объявлены поля FFile, FActive и FItems, а также методы PutItem, SetActive, GetItemCount и GetEndOfFile.
- **Public**. Поля, методы и свойства, объявленные в секции **public** не имеют никаких ограничений на использование, т.е. всегда видны за пределами модуля. Все, что помещается в секцию **public**, служит для манипуляций с объектами и составляет

программный интерфейс класса. Например, в классе TTextReader в эту секцию помещены конструктор Create, метод NextLine, свойства Active, Items, ItemCount.

- **Protected**. Поля, методы и свойства, объявленные в секции **protected**, видны за пределами модуля только потомкам данного класса; остальным частям программы они не видны. Так же как и **private**, директива **protected** позволяет скрыть особенности реализации класса, но в отличие от нее разрешает другим программистам порождать новые классы и обращаться к полям, методам и свойствам, которые составляют так называемый интерфейс разработчика. В эту секцию обычно помещаются виртуальные методы. Примером такого метода является ParseLine.
- **Published**. Устанавливает правила видимости те же, что и директива **public**. Особенность состоит в том, что для элементов, помещенных в секцию **published**, компилятор генерирует информацию о типах этих элементов. Эта информация доступна во время выполнения программы, что позволяет превращать объекты в компоненты визуальной среды разработки. Секцию **published** разрешено использовать только тогда, когда для самого класса или его предка включена директива компилятора **$TYPEINFO**.

Перечисленные секции могут чередоваться в объявлении класса в произвольном порядке, однако в пределах секции сначала следует описание полей, а потом методов и свойств. Если в определении класса нет ключевых слов **private**, **protected**, **public** и **published**, то для обычных классов всем полям, методам и свойствам приписывается атрибут видимости **public**, а для тех классов, которые порождены от классов библиотеки VCL, — атрибут видимости **published**.

Внутри модуля никакие ограничения на доступ к атрибутам классов, реализованных в этом же модуле, не действуют. Кстати, это отличается от соглашений, принятых в некоторых других языках программирования, в частности в языке C++.

3.11. Указатели на методы объектов

В языке Delphi существуют процедурные типы данных для методов объектов. Внешне объявление процедурного типа для метода отличается от обычного словосочетанием **of object**, записанным после прототипа процедуры или функции:

```
type
  TReadLineEvent = procedure (Reader: TTextReader; const Line: string) of object;
```

Переменная такого типа называется *указателем на метод* (method pointer). Она занимает в памяти 8 байт и хранит одновременно ссылку на объект и адрес его метода.

```
type
  TTextReader = class
  private
    FOnReadLine: TReadLineEvent;
    ...
  public
    property OnReadLine: TReadLineEvent read FOnReadLine write FOnReadLine;
  end;
```

Методы объектов, объявленные по приведенному выше шаблону, становятся совместимы по типу со свойством OnReadLine.

```
type
  TForm1 = class(TForm)
    procedure HandleLine(Reader: TTextReader; const Line: string);
  end;

var
  Form1: TForm1;
  Reader: TTextReader;
```

Если установить значение свойства OnReadLine:

```
Reader.OnReadLine := Form1.HandleLine;
```

и переписать метод NextLine,

```
function TTextReader.NextLine: Boolean;
var
  S: string;
  N: Integer;
begin
  Result := not EndOfFile;
  if Result then            // Если строки для считывания еще есть, то
  begin
    Readln(FFile, S);       // Считывание очередной строки
    N := ParseLine(S);      // Выделение элементов строки (разбор строки)
    if N <> ItemCount then
      SetLength(FItems, N);
    if Assigned(FOnReadLine) then
      FOnReadLine(Self, S); // уведомление о чтении очередной строки
  end;
end;
```

то объект Form1 через метод HandleLine получит уведомление об очередной считанной строке. Обратите внимание, что вызов метода через указатель происходит лишь в том случае, если указатель не равен **nil**. Эта проверка выполняется с помощью стандартной функции Assigned, которая возвращает True, если ее аргумент является связанным указателем.

Описанный выше механизм называется *делегированием*, поскольку он позволяет передать часть работы другому объекту, например, сосредоточить в одном объекте обработку событий, возникающих в других объектах. Это избавляет программиста от необходимости порождать многочисленные классы-наследники и перекрывать в них виртуальные методы. Делегирование широко применяется в среде Delphi. Например, все компоненты делегируют обработку своих событий той форме, в которую они помещены.

3.12. Метаклассы

3.12.1. Ссылки на классы

Язык Delphi позволяет рассматривать классы объектов как своего рода объекты, которыми можно манипулировать в программе. Такая возможность рождает новое понятие — *класс класса*; его принято обозначать термином *метакласс*.

Для поддержки метаклассов введен специальный тип данных — ссылка на класс (class reference). Он описывается с помощью словосочетания **class of**, например:

```
type
  TTextReaderClass = class of TTextReader;
```

Переменная типа TTextReaderClass объявляется в программе обычным образом:

```
var
  ClassRef: TTextReaderClass;
```

Значениями переменной ClassRef могут быть класс TTextReader и все порожденные от него классы. Допустимы следующие операторы:

```
ClassRef := TTextReader;
ClassRef := TDelimitedReader;
ClassRef := TFixedReader;
```

По аналогии с тем, как для всех классов существует общий предок TObject, у ссылок на классы существует базовый тип TClass, определенный, как:

```
type
  TClass = class of TObject;
```

Переменная типа TClass может ссылаться на любой класс.

Практическая ценность ссылок на классы состоит в возможности создавать программные модули, работающие с любыми классами объектов, даже теми, которые еще не разработаны.

Физический смысл и взаимосвязь таких понятий, как переменная-объект, экземпляр объекта в памяти, переменная-класс и экземпляр класса в памяти поясняет рисунок 3.4.

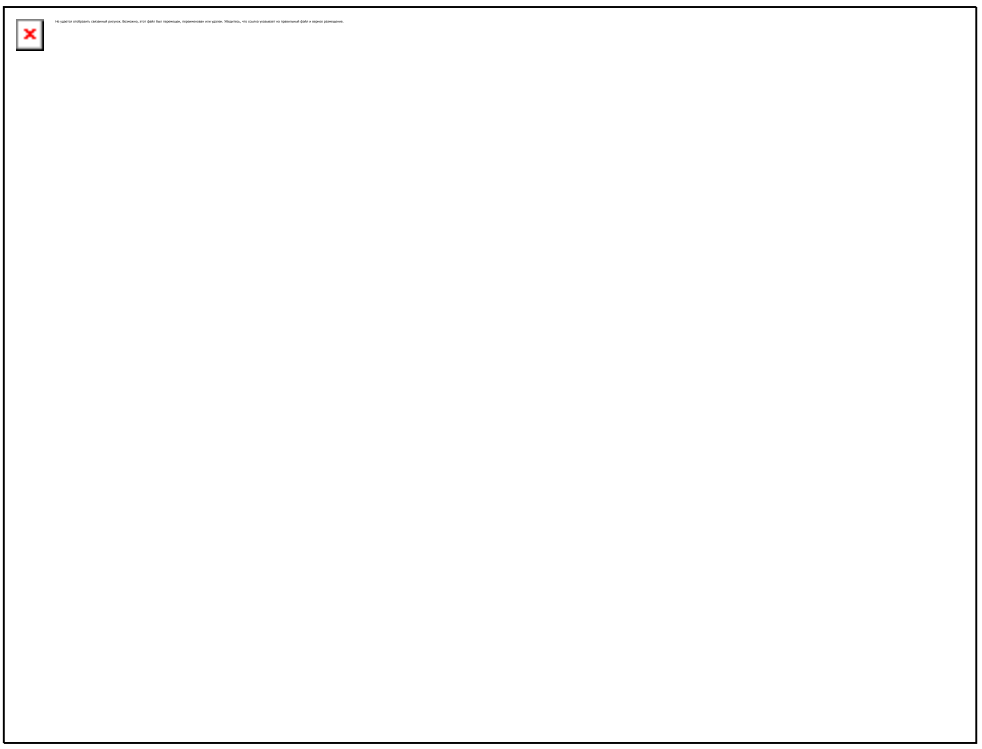


Рисунок 3.4. Переменная-объект, экземпляр объекта в памяти, переменная-класс и экземпляр класса в памяти

3.12.2. Методы классов

Метаклассы привели к возникновению нового типа методов — методов класса. *Метод класса* оперирует не экземпляром объекта, а непосредственно классом. Он объявляется как обычный метод, но перед словом **procedure** или **function** записывается зарезервированное слово **class**, например:

```
type
  TTextReader = class
    ...
    class function GetClassName: string;
  end;
```

Передаваемый в метод класса неявный параметр Self содержит не ссылку на объект, а ссылку на класс, поэтому в теле метода нельзя обращаться к полям, методам и свойствам объекта. Зато можно вызывать другие методы класса, например:

```
class function TTextReader.GetClassName: string;
begin
  Result := ClassName;
end;
```

Метод ClassName объявлен в классе TObject и возвращает имя класса, к которому применяется. Очевидно, что надуманный метод GetClassName просто дублирует эту функциональность для класса TTextReader и всех его наследников.

Методы класса применимы и к классам, и к объектам. В обоих случаях в параметре Self передается ссылка на класс объекта. Пример:

```
var
  Reader: TTextReader;
  S: string;
begin
  // Вызов метода с помощью ссылки на класс
  S := TTextReader.GetClassName;  // S получит значение 'TTextReader'

  // Создание объекта класса TDelimitedReader
  Reader := TDelimitedReader.Create('MyData.del');

  // Вызов метода с помощью ссылки на объект
  S := Reader.GetClassName;       // S получит значение 'TDelimitedReader'
end.
```

Методы классов могут быть виртуальными. Например, в классе TObject определен виртуальный метод класса NewInstance. Он служит для распределения памяти под объект и автоматически вызывается конструктором. Его можно перекрыть в своем классе, чтобы обеспечить нестандартный способ выделения памяти для экземпляров. Метод NewInstance должен перекрываться вместе с другим методом FreeInstance, который автоматически вызывается из деструктора и служит для освобождения памяти. Добавим, что размер памяти, требуемый для экземпляра, можно узнать вызовом предопределенного метода класса InstanceSize.

3.12.3. Виртуальные конструкторы

Особыевозможности ссылок на классы проявляется в сочетании с виртуальными конструкторами. Виртуальный конструктор объявляется с ключевым словом **virtual**. Вызов виртуального конструктора происходит по фактическому значению ссылки на класс, а не по ее формальному типу. Это позволяет создавать объекты, классы которых неизвестны на этапе компиляции. Механизм виртуальных конструкторов применяется в среде Delphi при восстановлении компонентов формы из файла. Восстановление компонента происходит следующим образом. Из файла считывается имя класса. По этому имени отыскивается ссылка на класс (метакласс). У метакласса вызывается виртуальный конструктор, который создает объект нужного класса.

```
var
  P: TComponent;
  T: TComponentClass;  // TComponentClass = class of TComponent;
...
  T := FindClass(ReadStr);
  P := T.Create(nil);
...
```

На этом закончим изучение теории объектно-ориентированного программирования и в качестве практики рассмотрим несколько широко используемых инструментальных классов среды Delphi. Разберитесь с их назначением и работой. Это поможет глубже понять ООП и пригодится на будущее.

3.13. Классы общего назначения

Как показывает практика, в большинстве задач приходится использовать однотипные структуры данных: списки, массивы, множества и т.д. От задачи к задаче изменяются только

их элементы, а методы работы сохраняются. Например, для любого списка нужны процедуры вставки и удаления элементов. В связи с этим возникает естественное желание решить задачу "в общем виде", т.е. создать универсальные средства для управления основными структурами данных. Эта идея не нова. Она давно пришла в голову разработчикам инструментальных пакетов, которые быстро наплодили множество вспомогательных библиотек. Эти библиотеки содержали классы объектов для работы со списками, коллекциями (динамические массивы с переменным количеством элементов), словарями (коллекции, индексированные строками) и другими "абстрактными" структурами. Для среды Delphi тоже разработаны аналогичные классы объектов. Их большая часть сосредоточена в модуле Classes. Наиболее нужными для вас являются списки строк (TStrings, TStringList) и потоки (TStream, THandleStream, TFileStream, TMemoryStream и TBlobStream). Рассмотрим кратко их назначение и применение.

3.13.1. Классы для представления списка строк

Для работы со списками строк служат классы TStrings и TStringList. Они используются в библиотеке VCL повсеместно и имеют гораздо большую универсальность, чем та, что можно почерпнуть из их названия. Классы TStrings и TStringList служат для представления не просто списка строк, а списка элементов, каждый из которых представляет собой пару строка-объект. Если со строками не ассоциированы объекты, получается обычный список строк.

Класс TStrings используется визуальными компонентами и является абстрактным. Он не имеет собственных средств хранения строк и определяет лишь интерфейс для работы с элементами. Класс TStringList является наследником TStrings и служит для организации списков строк, которые используются отдельно от управляющих элементов. Объекты TStringList хранят строки и объекты в динамической памяти.

Свойства класса TStrings описаны ниже.

Count: Integer — число элементов в списке.

Strings[Index: Integer]: string — обеспечивает доступ к массиву строк по индексу. Первая строка имеет индекс, равный 0. Свойство Strings является основным свойством объекта.

Objects[Index: Integer]: TObject — обеспечивает доступ к массиву объектов. Свойства Strings и Objects позволяют использовать объект TStrings как хранилище строк и ассоциированных с ними объектов произвольных классов.

Text: string — позволяет интерпретировать список строк, как одну большую строку, в которой элементы разделены символами #13#10 (возврат каретки и перевод строки).

Наследники класса TStrings иногда используются для хранения строк вида Имя=Значение, в частности, строк INI-файлов (см. гл. 6). Для удобной работы с такими строками в классе TStrings дополнительно имеются следующие свойства.

Names[Index: Integer]: string — обеспечивает доступ к той части строки, в которой содержится имя.

Values[const Name: string]: string — обеспечивает доступ к той части строки, в которой содержится значение. Указывая вместо Name ту часть строки, которая находится слева от знака равенства, вы получаете ту часть, что находится справа.

Управление элементами списка осуществляется с помощью следующих методов:

Add(const S: string): Integer — добавляет новую строку S в список и возвращает ее позицию. Новая строка добавляется в конец списка.

AddObject(const S: string; AObject: TObject): Integer — добавляет в список строку S и ассоциированный с ней объект AObject. Возвращает индекс пары строка-объект.

AddStrings(Strings: TStrings) — добавляет группу строк в существующий список.

Append(const S: string) — делает то же, что и Add, но не возвращает значения.

Clear — удаляет из списка все элементы.

Delete(Index: Integer) — удаляет строку и ассоциированный с ней объект. Метод Delete, также как метод Clear не разрушают объектов, т.е. не вызывают у них деструктор. Об этом вы должны позаботиться сами.

Equals(Strings: TStrings): Boolean — Возвращает True, если список строк в точности равен тому, что передан в параметре Strings.

Exchange(Index1, Index2: Integer) — меняет два элемента местами.

GetText: PChar — возвращает все строки списка в виде одной большой нуль-терминированной строки.

IndexOf(const S: string): Integer — возвращает позицию строки S в списке. Если заданная строка в списке отсутствует, функция возвращает значение -1.

IndexOfName(const Name: string): Integer — возвращает позицию строки, которая имеет вид Имя=Значение и содержит в себе Имя, равное Name.

IndexOfObject(AObject: TObject): Integer — возвращает позицию объекта AObject в массиве Objects. Если заданный объект в списке отсутствует, функция возвращает значение -1.

Insert(Index: Integer; const S: string) — вставляет в список строку S в позицию Index.

InsertObject(Index: Integer; const S: string; AObject: TObject) — вставляет в список строку S и ассоциированный с ней объект AObject в позицию Index.

LoadFromFile(const FileName: string) — загружает строки списка из текстового файла.

LoadFromStream(Stream: TStream) — загружает строки списка из потока данных (см. ниже).

Move(CurIndex, NewIndex: Integer) — изменяет позицию элемента (пары строка-объект) в списке.

SaveToFile(const FileName: string) — сохраняет строки списка в текстовом файле.

SaveToStream(Stream: TStream) — сохраняет строки списка в потоке данных.

SetText(Text: PChar) — загружает строки списка из одной большой нуль-терминированной строки.

Класс TStringList добавляет к TStrings несколько дополнительных свойств и методов, а также два свойства-события для уведомления об изменениях в списке. Они описаны ниже.

Свойства:

Duplicates: TDuplicates — определяет, разрешено ли использовать дублированные строки в списке. Свойство может принимать следующие значения: dupIgnore (дубликаты игнорируются), dupAccept (дубликаты разрешены), dupError (дубликаты запрещены, попытка добавить в список дубликат вызывает ошибку).

Sorted: Boolean — если имеет значение True, то строки автоматически сортируются в алфавитном порядке.

Методы:

Find(const S: string; var Index: Integer): Boolean — выполняет поиск строки S в списке строк. Если строка найдена, Find помещает ее позицию в переменную, переданную в параметре Index, и возвращает True.

Sort — сортирует строки в алфавитном порядке.

События:

OnChange: TNotifyEvent — указывает на обработчик события, который выполнится при изменении содержимого списка. Событие OnChange генерируется после того, как были сделаны изменения.

OnChanging: TNotifyEvent — указывает на обработчик события, который выполнится при изменении содержимого списка. Событие OnChanging генерируется перед тем, как будут сделаны изменения.

Ниже приводится фрагмент программы, демонстрирующий создание списка строк и манипулирование его элементами:

```
var
  Items: TStrings;
  I: Integer;
begin
  // Создание списка
  Items := TStringList.Create;
  Items.Add('Туризм');
  Items.Add('Наука');
  Items.Insert(1, 'Бизнес');
  ...
  // Работа со списком
  for I := 0 to Items.Count - 1 do
    Items[I] := UpperCase(Items[I]);
  ...
  // Удаление списка
  Items.Free;
end;
```

3.13.2. Классы для представления потока данных

В среде Delphi существует иерархия классов для хранения и последовательного ввода-вывода данных. Классы этой иерархии называются *потоками*. Потоки лучше всего представлять как файлы. Классы потоков обеспечивают различное физическое представление данных: файл на диске, раздел оперативной памяти, поле в таблице базы данных (таблица 3.1).

Класс	Описание
TStream	Абстрактный поток, от которого наследуются все остальные. Свойства и методы класса TStream образуют базовый интерфейс потоковых объектов.
THandleStream	Поток, который хранит свои данные в файле. Для чтения-записи файла используется дескриптор (handle), поэтому поток называется дескрипторным. Дескриптор — это номер открытого файла в операционной системе. Его возвращают низкоуровневые функции создания и открытия файла.
TFileStream	Поток, который хранит свои данные в файле. Отличается от ThandleStream тем, что сам открывает (создает) файл по имени, переданному в конструктор.
TMemoryStream	Поток, который хранит свои данные в оперативной памяти. Моделирует работу с файлом. Используется для хранения промежуточных результатов, когда файловый поток не подходит из-за низкой скорости

	передачи данных.
TResourceStream	Поток, обеспечивающий доступ к ресурсам в Windows-приложении.
TBlobStream	Обеспечивает последовательный доступ к большим полям таблиц в базах данных.

Таблица 3.1. Классы потоков

Потоки широко применяются в библиотеке VCL и наверняка вам понадобятся. Поэтому ниже кратко перечислены их основные общие свойства и методы.

Общие свойства:

Position: Longint — текущая позиция чтения-записи.

Size: Longint — текущий размер потока в байтах.

Общие методы:

CopyFrom(Source: TStream; Count: Longint): Longint — копирует Count байт из потока Source в свой поток.

Read(var Buffer; Count: Longint): Longint — читает Count байт из потока в буфер Buffer, продвигает текущую позицию на Count байт вперед и возвращает число прочитанных байт. Если значение функции меньше значения Count, то в результате чтения был достигнут конец потока.

ReadBuffer(var Buffer; Count: Longint) — читает из потока Count байт в буфер Buffer и продвигает текущую позицию на Count байт вперед. Если выполняется попытка чтения за концом потока, то генерируется ошибка.

Seek(Offset: Longint; Origin: Word): Longint — продвигает текущую позицию в потоке на Offset байт относительно позиции, заданной параметром Origin. Параметр Origin может иметь одно из следующих значений: 0 — смещение задается относительно начала потока; 1 — смещение задается относительно текущей позиции в потоке; 2 — смещение задается относительно конца потока.

Write(const Buffer; Count: Longint): Longint — записывает в поток Count байт из буфера Buffer, продвигает текущую позицию на Count байт вперед и возвращает реально записанное количество байт. Если значение функции отличается от значения Count, то при записи была ошибка.

WriteBuffer(const Buffer; Count: Longint) — записывает в поток Count байт из буфера Buffer и продвигает текущую позицию на Count байт вперед. Если по какой-либо причине невозможно записать все байты буфера, то генерируется ошибка.

Ниже приводится фрагмент программы, демонстрирующий создание файлового потока и запись в него строки:

```
var
  Stream: TStream;
  S: AnsiString;
  StrLen: Integer;

begin
  // Создание файлового потока
  Stream := TFileStream.Create('Sample.Dat', fmCreate);
  ...
  // Запись в поток некоторой строки
  StrLen := Length(S) * SizeOf(Char);
  Stream.Write(StrLen, SizeOf(Integer)); // запись длины строки
  Stream.Write(S, StrLen);               // запись символов строки
  ...
  // Закрытие потока
  Stream.Free;
end;
```

3.14. Итоги

Теперь для вас нет секретов в мире ООП. Вы на достаточно серьезном уровне познакомились с объектами и их свойствами; узнали, как объекты создаются, используются и уничтожаются. Если не все удалось запомнить сразу — не беда. Возвращайтесь к материалам главы по мере решения стоящих перед вами задач, и работа с объектами станет простой, естественной и даже приятной. Когда вы достигните понимания того, как работает один объект, то автоматически поймете, как работают все остальные. По мере накопления опыта вырастет и сложность ваших программ, поэтому в следующей главе мы рассмотрим то, с чем вы встретитесь очень скоро — ошибки программирования.

Глава 4. Исключительные ситуации и надежное программирование

Когда программист после компиляции получает готовый к исполнению файл, он искренне верит, что программа будет работать именно так, как он хочет. Пока она в его заботливых руках, так оно обычно и бывает. Когда же программа попадает в более суровые условия — к новому пользователю и на другой компьютер — с ней может произойти все, что угодно. “Новый хозяин“ может вместо ожидаемых цифр ввести буквы, извлечь корень из отрицательного числа, делить на ноль и выполнять множество других необдуманных, часто случайных действий. Особенно это касается интерактивных (диалоговых) приложений, а таких — громадное большинство. Из этого следует, что программист должен организовать мощную оборону от всех посягательств на жизнедеятельность своей программы в процессе ее выполнения. О том, как это сделать, рассказывается в этой главе.

4.1. Ошибки и исключительные ситуации

Вы должны отдавать себе отчет в том, что в любом работающем приложении могут происходить ошибки. Причины этих ошибок бывают разными. Некоторые из них носят субъективный характер и вызваны неграмотными действиями программиста. Но существуют и объективные ошибки, их нельзя избежать при проектировании программы, но можно обнаружить во время ее работы. Примеров таких ошибок сколько угодно: недостаточный объем свободной памяти, отсутствие файла на диске, выход значений исходных данных из допустимого диапазона и т.д.

Хорошая программа должна справляться со своими ошибками и работать дальше, не зацикливаясь и не зависая ни при каких обстоятельствах. Для обработки ошибок можно, конечно, пытаться использовать структуры вида **if** <error> **then** Exit. Однако в этом случае ваш стройный и красивый алгоритм решения основной задачи обрастет уродливыми проверками так, что через неделю вы сами в нем не разберетесь. Из этой почти тупиковой

ситуации среда Delphi предлагает простой и элегантный выход — механизм обработки исключительных ситуаций.

Исключительная ситуация (exception) — это прерывание нормального хода работы программы из-за невозможности правильно выполнить последующие действия.

Представим, что подпрограмма выделяет область динамической памяти и загружает в нее содержимое некоторого файла. Если в системе окажется недостаточно памяти, то данные будет негде разместить и попытка загрузить файл приведет к ошибке. Скорее всего, вся программа будет аварийно завершена из-за того, что оператор загрузки данных обратится по недоступному для программы адресу. Как этого избежать? При обнаружении проблемы подпрограмма должна создать исключительную ситуацию — прервать нормальный ход своей работы и передать управление тем операторам, которые смогут обработать ошибку. Как правило, операторы обработки исключительных ситуаций находятся в одной из вызывающих подпрограмм.

Механизм обработки исключительных ситуаций лучше всего подходит для взаимодействия программы с библиотекой подпрограмм. Подпрограммы библиотеки обнаруживают ошибки, но в большинстве случаев не знают, как на них реагировать. Вызывающая программа, наоборот, знает, что делать при возникновении ошибок, но, как правило, не умеет их своевременно обнаруживать. Благодаря механизму обработки исключительных ситуаций обеспечивается связь между библиотекой и использующей ее программой при обработке ошибок.

Механизм обработки исключительных ситуаций довольно сложен в своей реализации, но для программиста он прост и прозрачен. Для его использования в язык Delphi введены специальные конструкции **try**...**except**...**end**, **try**...**finally**...**end** и оператор **raise**, рассмотренные в этой главе.

4.2. Классы исключительных ситуаций

Исключительные ситуации в языке Delphi описываются классами. Каждый класс соответствует определенному типу исключительных ситуаций. Когда в программе возникает исключительная ситуация, создается объект соответствующего класса, который переносит информацию об этой ситуации из места возникновения в место обработки.

Классы исключительных ситуаций образуют иерархию, корнем которой является класс **Exception**. Класс **Exception** описывает самый общий тип исключительных ситуаций, а его наследники — конкретные виды таких ситуаций (таблица 4.1). Например, класс **EOutOfMemory** порожден от **Exception** и описывает ситуацию, когда свободная оперативная память исчерпана.

В следующей таблице приведены стандартные классы исключительных ситуаций, объявленные в модуле SysUtils. Они покрывают практически весь спектр возможных ошибок. Если их все-таки окажется недостаточно, вы можете объявить новые классы исключительных ситуаций, порожденные от класса **Exception** или его наследников.

Класс исключительных ситуаций	Описание
EAbort	«Безмолвная» исключительная ситуация, используемая для выхода из нескольких уровней вложенных блоков или подпрограмм. При этом на экран не выдается никаких сообщений об ошибке. Для генерации исключительной ситуации класса EAbort нужно вызвать стандартную процедуру

	Abort.
EInOutError	Ошибка доступа к файлу или устройству ввода-вывода. Код ошибки содержится в поле ErrorCode.
EExternal	Исключительная ситуация, возникшая вне программы, например, в операционной системе.
EExternalException	Исключительная ситуация, возникшая за пределами программы, например в DLL-библиотеке, разработанной на языке C++.
EHeapException	Общий класс исключительных ситуаций, возникающих при работе с динамической памятью. Является базовым для классов EOutOfMemory и EInvalidPointer. Внимание! Создание исключительных ситуаций этого класса (и всех его потомков) полностью берет на себя среда Delphi, поэтому никогда не создавайте такие исключительные ситуации с помощью оператора **raise**.
EOutOfMemory	Свободная оперативная память исчерпана (см. EHeadException).
EInvalidPointer	Попытка освободить недействительный указатель (см. EHeadException). Обычно это означает, что указатель уже освобожден.
EIntError	Общий класс исключительных ситуаций целочисленной арифметики, от которого порождены классы EDivByZero, ERangeError и EIntOverflow.
EDivByZero	Попытка деления целого числа на нуль.
ERangeError	Выход за границы диапазона целого числа или результата целочисленного выражения.
EIntOverflow	Переполнение в результате целочисленной операции.
EMathError	Общий класс исключительных ситуаций вещественной математики, от которого порождены классы EInvalidOp, EZeroDivide, EOverflow и EUnderflow.
EInvalidOp	Неверный код операции вещественной математики.
EZeroDivide	Попытка деления вещественного числа на нуль.
EOverflow	Потеря старших разрядов вещественного числа в

	результате переполнения разрядной сетки.
EUnderflow	Потеря младших разрядов вещественного числа в результате переполнения разрядной сетки.
EInvalidCast	Неудачная попытка приведения объекта к другому классу с помощью оператора **as**.
EConvertError	Ошибка преобразования данных с помощью функций IntToStr, StrToInt, StrToFloat, StrToDateTime.
EVariantError	Невозможность преобразования варьируемой переменной из одного формата в другой.
EAccessViolation	Приложение осуществило доступ к неверному адресу в памяти. Обычно это означает, что программа обратилась за данными по неинициализированному указателю.
EPrivilege	Попытка выполнить привилегированную инструкцию процессора, на которую программа не имеет права.
EStackOverflow	Стек приложения не может быть больше увеличен.
EControlC	Во время работы консольного приложения пользователь нажал комбинацию клавиш Ctrl+C.
EAssertionFailed	Возникает при вызове процедуры Assert, когда первый параметр равен значению False.
EPackageError	Проблема во время загрузки и инициализации библиотеки компонентов.
EOSError	Исключительная ситуация, возникшая в операционной системе.

Таблица 4.1. Классы исключительных ситуаций

Наследование классов позволяет создавать семейства родственных исключительных ситуаций. Примером такого семейства являются классы исключительных ситуаций вещественной математики, которые объявлены в модуле SysUtils следующим образом.

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

Класс исключительных ситуаций EMathError является базовым для классов EInvalidOp, EZeroDivide, EOverflow и EUnderflow, поэтому, обрабатывая исключительные ситуации класса EMathError, вы будете обрабатывать все ошибки вещественной математики, включая EInvalidOp, EZeroDivide, EOverflow и EUnderflow.

Нетрудно заметить, что имена классов исключений начинаются с буквы **E** (от слова Exception). Этого правила полезно придерживаться при объявлении собственных классов исключений, например:

```
type
  EMyException = class(Exception)
    MyErrorCode: Integer;
  end;
```

Как описываются классы исключительных ситуаций понятно, рассмотрим теперь, как такие ситуации обрабатываются.

4.3. Обработка исключительных ситуаций

4.3.1. Создание исключительной ситуации

Идея обработки исключительных ситуаций состоит в следующем. Когда подпрограмма сталкивается с невозможностью выполнения последующих действий, она создает объект с описанием ошибки и прерывает нормальный ход своей работы с помощью оператора **raise**. Так возникает исключительная ситуация.

```
raise EOutOfMemory.Create('Маловато памяти');
```

Данный оператор создает объект класса **EOutOfMemory** (класс ошибок исчерпания памяти) и прерывает нормальное выполнение программы. Вызывающие подпрограммы могут эту исключительную ситуацию перехватить и обработать. Для этого в них организуется так называемый *защищенный блок*:

```
try
  // защищаемые от ошибок операторы
except
  // операторы обработки исключительной ситуации
end;
```

Между словами **try** и **except** помещаются защищаемые от ошибок операторы. Если при выполнении любого из этих операторов возникает исключительная ситуация, то управление передается операторам между словами **except** и **end**, образующим блок обработки исключительных ситуаций. При нормальном (безошибочном) выполнении программы блок **except**...**end** пропускается (рисунок 4.1).

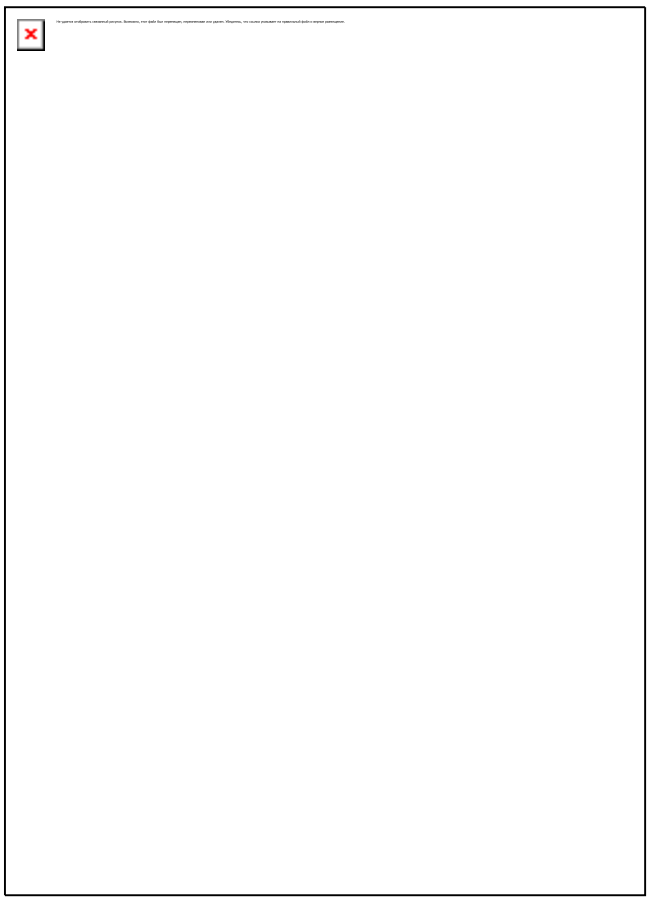


Рисунок 4.1. Логика работы оператора try…except…end

При написании программы вы можете использовать вложенные защищенные блоки, чтобы организовать локальную и глобальную обработку исключительных ситуаций. Концептуально это выглядит следующим образом:

```
try
  // защищаемые операторы
  try
    // защищаемые операторы
  except
    // локальная обработка исключительных ситуаций
  end;
  // защищаемые операторы
except
  // глобальная обработка исключительных ситуаций
end;
```

Исключительные ситуации внешнего защищенного блока, возникающие за пределами вложенного блока, обрабатываются внешней секцией **except**...**end**. Исключительные ситуации вложенного защищенного блока обрабатываются вложенной секцией **except**...**end**.

4.3.2. Распознавание класса исключительной ситуации

Распознавание класса исключительной ситуации выполняется с помощью конструкций

```
on <класс исключительной ситуации> do <оператор>;
```

которые записываются в секции обработки исключительной ситуации, например:

```
try
  // вычисления с вещественными числами
except
  on EZeroDivide do ... ; // обработка ошибки деления на нуль
  on EMathError do ... ;  // обработка других ошибок вещественной математики
end;
```

Поиск соответствующего обработчика выполняется последовательно до тех пор, пока класс исключительной ситуации не окажется совместимым с классом, указанным в операторе **on**. Как только обработчик найден, выпоняется оператор, стоящий за словом **do** и управление передается за секцию **except**...**end**. Если исключительная ситуация не относится ни к одному из указанных классов, то управление передается во внешний блок **try**...**except**...**end** и обработчик ищется в нем.

Обратите внимание, что порядок операторов **on** имеет значение, поскольку распознавание исключительных ситуаций должно происходить от частных классов к общим классам, иначе говоря, от потомков к предкам. С чем это связано? Сейчас поймете. Представьте, к чему приведет изменение порядка операторов **on** в примере выше, если принять во внимание, что класс **EMathError** является базовым для **EZeroDivide**. Ответ простой: обработчик **EMathError** будет поглощать все ошибки вещественной математики, в том числе **EZeroDivide**, в результате обработчик **EZeroDivide** никогда не выполнится.

На самом высоком уровне программы бывает необходимо перехватывать все исключительные ситуации, чтобы в случае какой-нибудь неучтенной ошибки корректно завершить приложение. Для этого применяется так называемый *обработчик по умолчанию* (default exception handler). Он записывается в секции **except** после всех операторов **on** и начинается ключевым словом **else**:

```
try
  { вычисления с вещественными числами }
except
  on EZeroDivide do { обработка ошибки деления на нуль };
  on EMathError do  { обработка других ошибок вещественной математики };
  else { обработка всех остальных ошибок (обработчик по умолчанию) };
end;
```

Учтите, что отсутствие части **else** соответствует записи **else raise**, которое нет смысла использовать явно. Мы со своей стороны вообще не советуем вам пользоваться обработкой исключительных ситуаций по умолчанию, поскольку все ваши приложения будут строиться, как правило, на основе библиотеки VCL, в которой обработка по умолчанию уже предусмотрена.

4.3.3. Пример обработки исключительной ситуации

В качестве примера обработки исключительной ситуации рассмотрим две функции: StringToCardinal и StringToCardinalDef.

Функция StringToCardinal выполняет преобразование строки в число с типом Cardinal. Если преобразование невозможно, функция создает исключительную ситуацию класса EConvertError.

```
function StringToCardinal(const S: string): Cardinal;
var
  I: Integer;
  B: Cardinal;
begin
  Result := 0;
  B := 1;
  for I := Length(S) downto 1 do
  begin
    if not (S[I] in ['0'..'9']) then
      raise EConvertError.Create(S + ' is not a valid cardinal value');
    Result := Result + B * (Ord(S[I]) - Ord('0'));
    B := B * 10;
  end;
end;
```

Функция StringToCardinalDef также выполняет преобразование строки в число с типом Cardinal, но в отличие от функции StringToCardinal она не создает исключительную ситуацию. Вместо этого она позволяет задать значение, которое возвращается в случае неудачной попытки преобразования:

```
function StringToCardinalDef(const S: string; Default: Cardinal = 0): Cardinal;
begin
  try
    Result := StringToCardinal(S);
  except
    on EConvertError do
      Result := Default;
  end;
end;
```

Для преобразования исходной строки в число используется определенная выше функция StringToCardinal. Если при преобразовании возникает исключительная ситуация, то она «поглощается» функцией StringToCardinalDef, которая в этом случае возвращает значение параметра Default. Если происходит какая-нибудь другая ошибка (не EConvertError), то управление передается внешнему блоку обработки исключительных ситуаций, из которого была вызвана функция StringToCardinalDef.

Пример очень прост, но хорошо демонстрирует преимущества исключительных ситуаций перед традиционной обработкой ошибок. Представьте более сложные вычисления, состоящие из множества операторов, в каждом из которых может произойти ошибка. Насколько сложной окажется обработка ошибок многочисленными операторами **if** и насколько простой оператором **try**.

4.3.4. Возобновление исключительной ситуации

В тех случаях, когда защищенный блок не может обработать исключительную ситуацию полностью, он выполняет только свою часть работы и возобновляет исключительную ситуацию с тем, чтобы ее обработку продолжил внешний защищенный блок:

```
try
  // вычисления с вещественными числами
except
  on EZeroDivide do
  begin
    // частичная обработка ошибки
    raise; // возобновление исключительной ситуации
  end;
end;
```

Если ни один из внешних защищенных блоков не обработал исключительную ситуацию, то управление передается стандартному обработчику исключительной ситуации, завершающему приложение.

4.3.5. Доступ к объекту, описывающему исключительную ситуацию

При обработке исключительной ситуации может потребоваться доступ к объекту, описывающему эту ситуацию и содержащему код ошибки, текстовое описание ошибки и т.д. В этом случае используется расширенная запись оператора **on**:

```
on <идентификатор объекта> : <класс исключительной ситуации> do <оператор>;
```

Например, объект исключительной ситуации нужен для того, чтобы выдать пользователю сообщение об ошибке:

```
try
  // защищаемые операторы
except
  on E: EOutOfMemory do
    ShowMessage(E.Message);
end;
```

Переменная **E** — это объект исключительной ситуации, **ShowMessage** — процедура модуля DIALOGS, отображающая на экране небольшое окно с текстом и кнопкой OK. Свойство **Message** типа **string** определено в классе **Exception**, оно содержит текстовое описание ошибки. Исходное значение для текста сообщения указывается при конструировании объекта исключительной ситуации.

Обратите внимание, что после обработки исключительной ситуации освобождение соответствующего объекта выполняется автоматически, вам этого делать не надо.

4.4. Защита выделенных ресурсов от пропадания

4.4.1. Утечка ресурсов и защита от нее

Программы, построенные с использованием механизма исключительных ситуаций, обязаны придерживаться строгих правил распределения и освобождения таких ресурсов, как память, файлы, ресурсы операционной системы.

Представьте ситуацию: подпрограмма распределяет некоторый ресурс, но исключительная ситуация прерывает ее выполнение, и ресурс остается не освобожденным. Даже подумать страшно, к чему может привести такая ошибка: утечка памяти, файловых дескрипторов, других ресурсов операционной системы. Следовательно, ресурсы нуждаются в защите от исключительных ситуаций. Для этого в среде Delphi предусмотрен еще один вариант защищенного блока:

```
// запрос ресурса
try
  // защищаемые операторы, которые используют ресурс
finally
  // освобождение ресурса
end;
```

Особенность этого блока состоит в том, что секция **finally**...**end** выполняется всегда независимо от того, происходит исключительная ситуация или нет. Если какой-либо оператор секции **try**...**finally** генерирует исключительную ситуацию, то сначала выполняется секция **finally**...**end**, называемая секцией завершения (освобождения ресурсов), а затем управление передается внешнему защищенному блоку. Если все защищаемые операторы выполняются без ошибок, то секция завершения тоже работает, но управление передается следующему за ней оператору. Обратите внимание, что секция **finally**...**end** не обрабатывает исключительную ситуацию, в ней нет ни средств ее обнаружения, ни средств доступа к объекту исключительной ситуации.

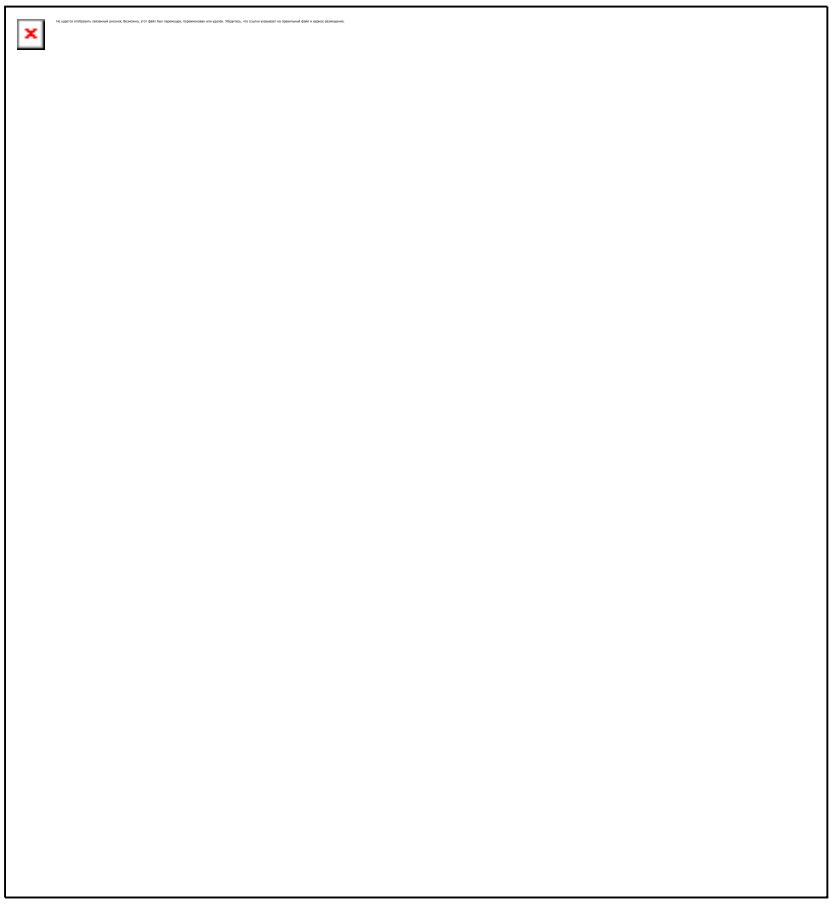


Рисунок 4.1. Логика работы оператора try…finally…end

Блок **try**...**finally**...**end** обладает еще одной важной особенностью. Если он помещен в цикл, то вызов из защищенного блока процедуры Break с целью преждевременного выхода из цикла или процедуры Continue с целью перехода на следующую итерацию цикла сначала обеспечивает выполнение секции **finally**...**end**, а затем уже выполняется соответствующий переход. Это утверждение справедливо также и для процедуры Exit (выход из подпрограммы).

Как показывает практика, подпрограммы часто распределяют сразу несколько ресурсов и используют их вместе. В таких случаях применяются вложенные блоки **try**...**finally**...**end**:

```
// распределение первого ресурса
try
  ...
  // распределение второго ресурса
  try
    // использование обоих ресурсов
  finally
    // освобождение второго ресурса
  end;
  ...
finally
  // освобождение первого ресурса
end;
```

Кроме того, вы успешно можете комбинировать блоки **try**...**finally**...**end** и **try**...**except**...**end** для защиты ресурсов и обработки исключительных ситуаций.

4.5. Итоги

В этой главе вы узнали многое об исключительных ситуациях и способах борьбы с ними. Теперь ваши программы наверняка дадут достойный отпор не только самому неотесанному пользователю, но и его деревянному компьютеру. Это, кстати говоря, одно из необходимых

качеств, которые позволят отнести вашу программу к классу хороших. Позволим себе также напомнить, что здесь были рассмотрены только ошибки времени выполнения, поэтому не забудьте прочитать гл.10, где рассказано о борьбе с логическими ошибками.

Глава 5. Динамически загружаемые библиотеки

До сих пор создаваемые нами программы были монолитными и фактически состояли из одного выполняемого файла. Это, конечно, очень удобно, но не всегда эффективно. Если вы создаете не одну программу, а несколько, и в каждой из них пользуетесь общим набором подпрограмм, то код этих подпрограмм включается в каждую вашу программу. В результате достаточно большие общие части кода начинают дублироваться во всех ваших программах, неоправданно «раздувая» их размеры. Поддержка программ затрудняется, ведь если вы исправили ошибку в некоторой подпрограмме, то вам придется перекомпилировать и переслать потребителю целиком все программы, которые ее используют. Решение проблемы напрашивается само собой — перейти к модульной организации выполняемых файлов. В среде Delphi эта идея реализуется с помощью динамически загружаемых библиотек. Техника работы с ними рассмотрена в данной главе.

5.1. Динамически загружаемые библиотеки

Динамически загружаемая библиотека (от англ. dynamically loadable library) — это библиотека подпрограмм, которая загружается в оперативную память и подключается к использующей программе во время ее работы (а не во время компиляции и сборки). Файлы динамически загружаемых библиотек в среде Windows обычно имеют расширение **.dll** (от англ. Dynamic-Link Library). Для краткости в этой главе мы будем использовать термин динамическая библиотека, или даже просто библиотека, подразумевая DLL-библиотеку.

Несколько разных программ могут использовать в работе общую динамически загружаемую библиотеку. При этом операционная система в действительности загружает в оперативную память лишь одну копию библиотеки и обеспечивает совместный доступ к ней со стороны всех программ. Кроме того, такие библиотеки могут динамически загружаться и выгружаться из оперативной памяти по ходу работы программы, освобождая ресурсы системы для других задач.

Одно из важнейших назначений динамически загружаемых библиотек — это взаимодействие подпрограмм, написанных на разных языках программирования. Например, вы можете свободно использовать в среде Delphi динамически загружаемые библиотеки, разработанные в других системах программирования с помощью языков C и C++. Справедливо и обратное утверждение — динамически загружаемые библиотеки, созданные в среде Delphi, можно подключать к программам на других языках программирования.

5.2. Разработка библиотеки

5.2.1. Структура библиотеки

По структуре исходный текст библиотеки похож на исходный текст программы, за исключением того, что текст библиотека начинается с ключевого слова **library**, а не слова **program**. Например:

```
library SortLib;
```

После заголовка следуют секции подключения модулей, описания констант, типов данных, переменных, а также описания процедур и функций. Процедуры и функции — это главное, что должно быть в динамически загружаемой библиотеке, поскольку лишь они могут быть экспортированы.

Если в теле библиотеки объявлены некоторые процедуры,

```
procedure BubleSort(var Arr: array of Integer);
procedure QuickSort(var Arr: array of Integer);
```

то это еще не значит, что они автоматически станут доступны для вызова извне. Для того чтобы это разрешить, нужно поместить имена процедур в специальную секцию **exports**, например:

```
exports
  BubleSort,
  QuickSort;
```

Перечисленные в секции **exports** процедуры и функции отделяются запятой, а в конце всей секции ставится точка с запятой. Секций **exports** может быть несколько, и они могут располагаться в программе произвольным образом.

Ниже приведен пример исходного текста простейшей динамически загружаемой библиотеки SortLib. Она содержит единственную процедуру BubleSort, сортирующую массив целых чисел методом «пузырька»:

```
library SortLib;

procedure BubleSort(var Arr: array of Integer);
var
  I, J, T: Integer;
begin
  for I := Low(Arr) to High(Arr) - 1 do
    for J := I + 1 to High(Arr) do
      if Arr[I] > Arr[J] then
      begin
        T := Arr[I];
        Arr[I] := Arr[J];
        Arr[J] := T;
      end;
end;

exports
  BubleSort;

begin
end.
```

Исходный текст динамически загружаемой библиотеки заканчивается операторным блоком **begin...end**, в который можно вставить любые операторы для подготовки библиотеки к работе. Эти операторы выполняются во время загрузки библиотеки основной программой. Наша простейшая библиотека SortLib не требует никакой подготовки к работе, поэтому ее операторный блок пустой.

5.2.2. Экспорт подпрограмм

Если бы мы смогли заглянуть внутрь компилированного файла библиотеки, то обнаружили бы, что каждая экспортируемая подпрограмма представлена там уникальным символьным именем. Эти имена собраны в таблицу и используются при поиске подпрограмм — с их помощью выполняется динамическая привязка записанных в программе команд вызова к адресам соответствующих процедур и функций в библиотеке. В качестве экспортного имени может выступать любая последовательность символов, причем между заглавными и строчными буквами делается различие.

В стандартном случае экспортное имя подпрограммы считается в точности таким, как ее идентификатор в исходном тексте библиотеки (с учетом заглавных и строчных букв). Например, если секция **exports** имеет следующий вид,

```
exports
  BubleSort;
```

то это означает, что экспортное имя процедуры будет ’BubleSort’. При желании это имя можно сделать отличным от программного имени, дополнив описание директивой **name**, например:

```
exports
  BubleSort name 'BubleSortIntegers';
```

В итоге, экспортное имя процедуры BubleSort будет ’BubleSortIntegers’.

Экспортные имена подпрограмм должны быть уникальны в пределах библиотеки, поэтому их нужно всегда указывать явно для перегруженных (**overload**) процедур и функций. Например, если имеются две перегруженные процедуры с общим именем QuickSort,

```
procedure QuickSort(var Arr: array of Integer); overload; // для целых чисел
procedure QuickSort(var Arr: array of Real); overload;    // для вещественных
```

то при экспорте этим двум процедурам необходимо явно указать отличные друг от друга экспортные имена:

```
exports
  QuickSort(var Arr: array of Integer) name 'QuickSortIntegers';
  QuickSort(var Arr: array of Real) name 'QuickSortReals';
```

Полные списки параметров нужны для того, чтобы компилятор мог разобраться, о какой процедуре идет речь в каждом случае.

5.2.3. Соглашения о вызове подпрограмм

В главе 2 мы уже кратко рассказывали о том, что в различных языках программирования используются различные правила вызова подпрограмм, и что для совместимости с ними в языке Delphi существуют директивы **register**, **stdcall**, **pascal** и **cdecl**. Применение этих директив становится особенно актуальным при разработке динамически загружаемых библиотек, которые используются в программах, написанных на других языках программирования.

Чтобы разобраться с применением директив, обратимся к механизму вызова подпрограмм. Он основан на использовании стека.

Стек — это область памяти, в которую данные помещаются в прямом порядке, а и извлекаются в обратном, по аналогии с наполнением и опустошением магазина патронов у стрелкового оружия. Очередность работы с элементами в стеке обозначается термином LIFO (от англ. Last In, First Out — последним вошел, первым вышел).

> **ПРИМЕЧАНИЕ**
>
> Существует еще обычная очередность работы с элементами, обозначаемая термином FIFO (от англ. First In, First Out — первым вошел, первым вышел).

Для каждой программы на время работы создается свой стек. Через него передаются параметры подпрограмм и в нем же сохраняются адреса возврата из этих подпрограмм. Именно благодаря стеку подпрограммы могут вызывать друг друга, или даже рекурсивно сами себя.

Вызов подпрограммы состоит из «заталкивания» в стек всех аргументов и адреса следующей команды (для возврата к ней), а затем передачи управления на начало подпрограммы. По окончании работы подпрограммы из стека извлекается адрес возврата с передачей управления на этот адрес; одновременно с этим из стека выталкиваются аргументы. Происходит так называемая очистка стека. Это общая схема работы и у нее бывают разные реализации. В частности, аргументы могут помещаться в стек либо в прямом порядке (слева направо, как они перечислены в описании подпрограммы), либо в обратном порядке (справа налево), либо вообще, не через стек, а через свободные регистры процессора для повышения скорости

работы. Кроме того, очистку стека может выполнять либо вызываемая подпрограмма, либо вызывающая программа. Выбор конкретного соглашения о вызове обеспечивают директивы **register**, **pascal**, **cdecl** и **stdcall**. Их смысл поясняет таблица 5.1.

Директива	Порядок занесения аргументов в стек	Кто отвечает за очистку стека	Передача аргументов через регистры
register	Слева направо	Подпрограмма	Да
pascal	Слева направо	Подпрограмма	Нет
cdecl	Справа налево	Вызывающая программа	Нет
stdcall	Справа налево	Подпрограмма	Нет

Таблица 5.1. Соглашения о вызове подпрограмм

> **ПРИМЕЧАНИЕ**
>
> Директива register не означает, что все аргументы обязательно передаются через регистры процессора. Если число аргументов больше числа свободных регистров, то часть аргументов передается через стек.

Возникает резонный вопрос: какое соглашение о вызове следует выбирать для процедур и функций динамически загружаемых библиотек. Ответ — соглашение **stdcall**:

```
procedure BubleSort(var Arr: array of Integer); stdcall;
procedure QuickSort(var Arr: array of Integer); stdcall;
```

Именно соглашение **stdcall**, изначально предназначенное для вызова подпрограмм операционной системы, лучше всего подходит для взаимодействия программ и библиотек, написанных на разных языках программирования. Все программы так или иначе используют функции операционной системы, следовательно они обязательно поддерживают соглашение **stdcall**.

5.2.4. Пример библиотеки

Вооруженные теорией, приступим к практике — разработаем какую-нибудь полезную библиотеку, а затем подключим ее к своей программе. На этом примере мы покажем вам, как оформляется динамически загружаемая библиотека, составленная из нескольких программных модулей.

Шаг 1. Запустите систему Delphi и выберите в меню команду **File | New | Other...** . В диалоговом окне, которое откроется на экране, выберите значок с подписью **DLL Wizard** и нажмите кнопку **OK** (рисунок 5.1):

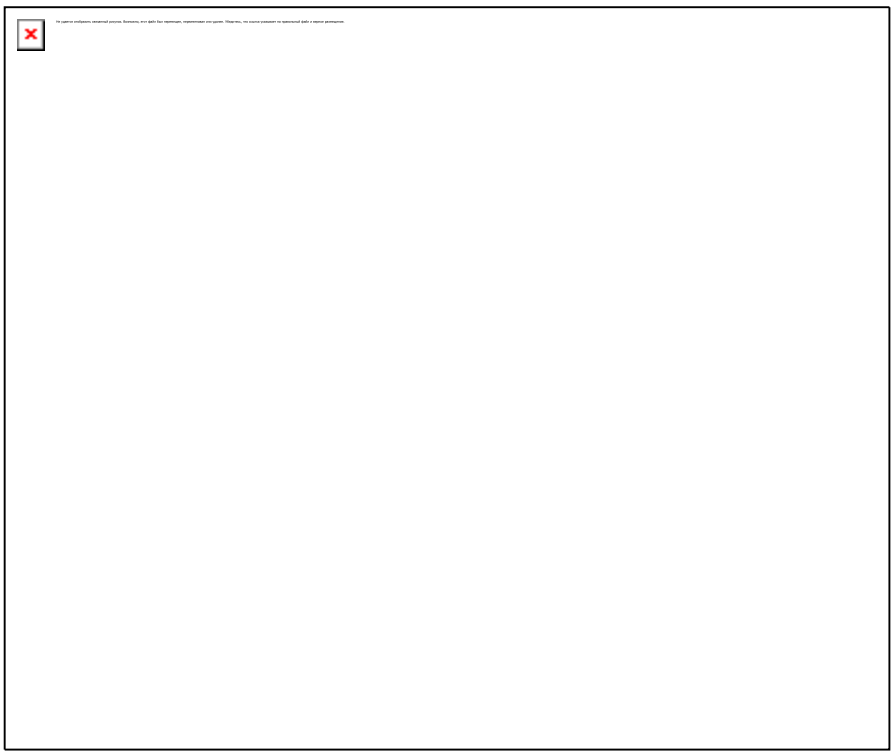


Рисунок 5.1. Окно выбора нового проекта, в котором выделен пункт DLL Wizard

Среда Delphi создаст новый проект со следующей заготовкой библиотеки:

```
library Project1;

{ Important note about DLL memory management ... }

uses
  SysUtils,
  Classes;

begin
end.
```

Шаг 2. С помощью команды **File | New | Unit** создайте в проекте новый программный модуль. Его заготовка будет выглядеть следующим образом:

```
unit Unit1;

interface

implementation

end.
```

Шаг 3. Сохраните модуль под именем SortUtils.pas, а проект — под именем SortLib.dpr. Прейдите к главному файлу проекта и удалите из секции **uses** модули SysUtils и Classes (они сейчас не нужны). Главный программный модуль должен стать следующим:

```
library SortLib;

{ Important note about DLL memory management ... }

uses
  SortUtils in 'SortUtils.pas';

begin
end.
```

Шаг 4. Наберите исходный текст модуля SortUtils:

```
unit SortUtils;

interface

procedure BubleSort(var Arr: array of Integer); stdcall;
procedure QuickSort(var Arr: array of Integer); stdcall;

exports
  BubleSort name 'BubleSortIntegers',
  QuickSort name 'QuickSortIntegers';

implementation

procedure BubleSort(var Arr: array of Integer);
var
  I, J, T: Integer;
begin
  for I := Low(Arr) to High(Arr) - 1 do
    for J := I + 1 to High(Arr) do
      if Arr[I] > Arr[J] then
      begin
        T := Arr[I];
        Arr[I] := Arr[J];
        Arr[J] := T;
      end;
end;

procedure QuickSortRange(var Arr: array of Integer; Low, High: Integer);
var
  L, H, M: Integer;
  T: Integer;
begin
  L := Low;
  H := High;
  M := (L + H) div 2;
  repeat
    while Arr[L] < Arr[M] do
      L := L + 1;
    while Arr[H] > Arr[M] do
      H := H - 1;
    if L <= H then
    begin
      T := Arr[L];
      Arr[L] := Arr[H];
      Arr[H] := T;
      if M = L then
        M := H
      else if M = H then
        M := L;
      L := L + 1;
      H := H - 1;
    end;
  until L > H;
  if H > Low then QuickSortRange(Arr, Low, H);
  if L < High then QuickSortRange(Arr, L, High);
end;

procedure QuickSort(var Arr: array of Integer);
begin
  if Length(Arr) > 1 then
    QuickSortRange(Arr, Low(Arr), High(Arr));
end;

end.
```

В этом модуле процедуры BubleSort и QuickSort сортируют массив чисел двумя способами: методом «пузырька» и методом «быстрой» сортировки соответственно. С их реализацией мы предоставляем вам разобраться самостоятельно, а нас сейчас интересует правильное оформление процедур для их экспорта из библиотеки.

Директива **stdcall**, использованная при объявлении процедур BubleSort и QuickSort,

```
procedure BubleSort(var Arr: array of Integer); stdcall;
procedure QuickSort(var Arr: array of Integer); stdcall;
```

позволяет вызывать процедуры не только из программ на языке Delphi, но и из программ на языках C/C++ (далее мы покажем, как это сделать).

Благодаря присутствию в модуле секции **exports**,

```
exports
  BubleSort name 'BubleSortIntegers',
  QuickSort name 'QuickSortIntegers';
```

подключение модуля в главном файле библиотеки автоматически приводит к экспорту процедур.

Шаг 5. Сохраните все файлы проекта и выполните компиляцию. В результате вы получите на диске в своем рабочем каталоге двоичный файл библиотеки SortLib.dll. Соответствующее расширение назначается файлу автоматически, но если вы желаете, чтобы компилятор назначал другое расширение, воспользуйтесь командой меню **Project | Options…** и в появившемся окне **Project Options** на вкладке **Application** впишите расширение файла в поле **Target file extension** (рисунок 5.2).

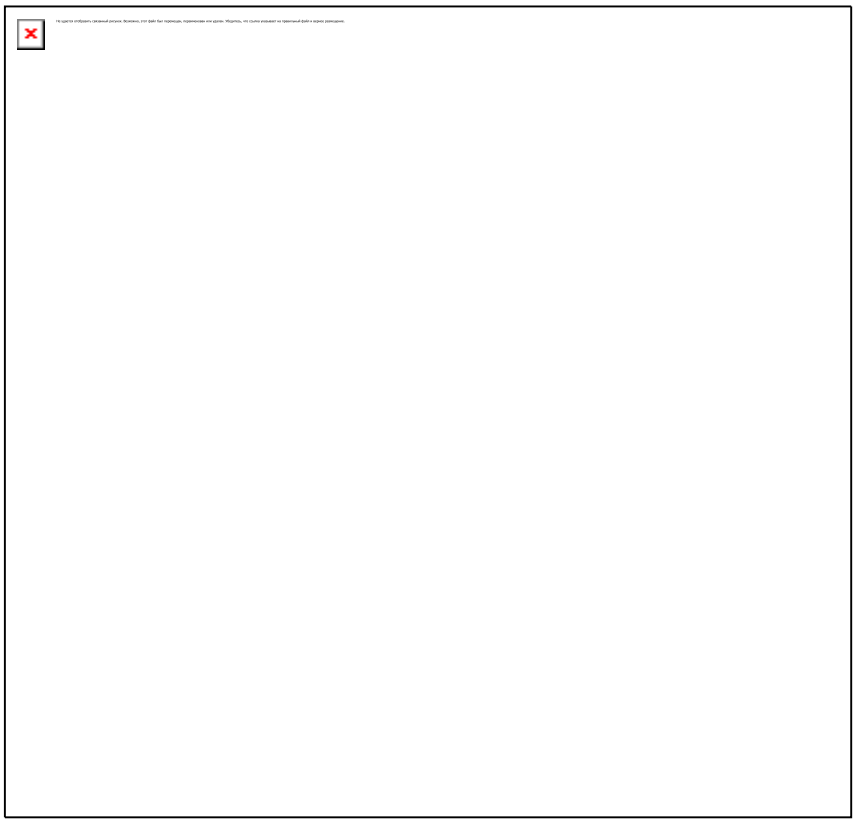


Рисунок 5.2. Окно настройки параметров проекта

Кстати, с помощью полей **LIB Prefix**, **LIB Suffix** и **LIB Version** этого окна вы можете задать правило формирования имени файла, который получается при сборке библиотеки. Имя файла составляется по формуле:

<LIB Prefix> + <имя проекта> + <LIB Suffix> + ’.’ + <Target file extention> + [’.’ + <LIB Version>]

5.3. Использование библиотеки в программе

Для того чтобы в прикладной программе воспользоваться процедурами и функциями библиотеки, необходимо выполнить так называемый импорт. *Импорт* обеспечивает загрузку

библиотеки в оперативную память и привязку записанных в программе команд вызова к адресам соответствующих процедур и функций библиотеки. Существуют два способа импорта, отличающихся по удобству и гибкости программирования:

- *статический импорт* (обеспечивается директивой компилятора **external**);
- *динамический импорт* (обеспечивается функциями **LoadLibrary** и **GetProcAddress**).

Статический импорт является более удобным, а динамический — более гибким.

5.3.1. Статический импорт

При статическом импорте все действия по загрузке и подключению библиотеки выполняются автоматически операционной системой во время запуска главной программы. Чтобы задействовать статический импорт, достаточно просто объявить в программе процедуры и функции библиотеки как внешние. Это делается с помощью директивы **external**, например:

```
procedure BubleSortIntegers(var Arr: array of Integer); stdcall;
  external 'SortLib.dll';

procedure QuickSortIntegers(var Arr: array of Integer); stdcall;
  external 'SortLib.dll';
```

После ключевого слова **external** записывается имя двоичного файла библиотеки в виде константной строки или константного строкового выражения. Вместе с директивой **external** может использоваться уже известная вам директива **name**, которая служит для явного указания экспортного имени процедуры в библиотеке. С ее помощью объявления процедур можно переписать по-другому:

```
procedure BubleSort(var Arr: array of Integer); stdcall;
  external 'SortLib.dll' name 'BubleSortIntegers';

procedure QuickSort(var Arr: array of Integer); stdcall;
  external 'SortLib.dll' name 'QuickSortIntegers';
```

Поместив в программу приведенные выше объявления, можно вызывать процедуры BubleSort и QuickSort, как будто они являются частью самой программы. Давайте это проверим.

Шаг 6. Создайте новую консольную программу. Для этого выберите в меню команду **File | New | Other...** и в открывшемся диалоговом окне выделите значок **Console Application**. Затем нажмите кнопку **OK**.

Шаг 7. Добавьте в программу **external**-объявления процедур BubleSort и QuickSort, а также наберите приведенный ниже текст программы. Сохраните проект под именем TestStaticImport.dpr.

```
program TestStaticImport;

{$APPTYPE CONSOLE}

procedure BubleSort(var Arr: array of Integer); stdcall;
  external 'SortLib.dll' name 'BubleSortIntegers';
procedure QuickSort(var Arr: array of Integer); stdcall;
  external 'SortLib.dll' name 'QuickSortIntegers';

var
  Arr: array [0..9] of Integer;
  I: Integer;

begin
  // Метод «пузырька»
  Randomize;
  for I := Low(Arr) to High(Arr) do
    Arr[I] := Random(100); // Заполнение массива случайными числами
  BubleSort(Arr);
  for I := Low(Arr) to High(Arr) do
    Write(Arr[I], ' ');
  Writeln;
  // Метод быстрой сортировки
  for I := Low(Arr) to High(Arr) do
    Arr[I] := Random(100); // Заполнение массива случайными числами
  QuickSort(Arr);
  for I := Low(Arr) to High(Arr) do
    Write(Arr[I], ' ');
  Writeln;
  Writeln('Press Enter to exit...');
  Readln;
end.
```

Шаг 8. Выполните компиляцию и запустите программу. Если числа печатаются на экране по возрастанию, то сортировка работает правильно.

В результате проделанных действий можно уже сделать первый важный вывод: компиляция программы не требует наличия компилированной библиотеки, а это значит, что их разработка может осуществляться совершенно независимо, причем разными людьми. Нужно лишь договориться о типах и списках параметров, передаваемых в процедуры и функции, а также выбрать единое соглашение о вызове.

5.3.2. Модуль импорта

При разработке динамически загружаемых библиотек нужно всегда думать об их удобном использовании. Давайте, например, обратимся к последнему примеру и представим, что в библиотеке не две процедуры, а сотня, и нужны они не в одной программе, а в нескольких. В этом случае намного удобнее вынести **external**-объявления процедур в отдельный модуль, подключаемый ко всем программам в секции **uses**. Такой модуль условно называют *модулем импорта*. Кроме объявлений внешних подпрограмм он обычно содержит определения типов данных и констант, которыми эти подпрограммы оперируют.

Модуль импорта для библиотеки SortLib будет выглядеть так:

```
unit SortLib;

interface

procedure BubleSort(var Arr: array of Integer); stdcall;
procedure QuickSort(var Arr: array of Integer); stdcall;

implementation

const
  DllName = 'SortLib.dll';

procedure BubleSort(var Arr: array of Integer); external
  DllName name 'BubleSortIntegers';
procedure QuickSort(var Arr: array of Integer); external
  DllName name 'QuickSortIntegers';

end.
```

Выполняемый файл библиотеки должен всегда сопровождаться модулем импорта, чтобы потребитель мог разобраться с параметрами подпрограмм и правильно воспользоваться библиотекой.

5.3.3. Динамический импорт

Действия по загрузке и подключению библиотеки (выполняемые при статическом импорте автоматически) можно проделать самостоятельно, обратившись к стандартным функциям операционной системы. Таким образом, импорт можно произвести динамически во время работы программы (а не во время ее запуска).

Для динамического импорта необходимо загрузить библиотеку в оперативную память вызовом функции **LoadLibrary**, а затем извлечь из нее адреса подпрограмм с помощью функции **GetProcAddress**. Полученные адреса нужно сохранить в процедурных переменных соответствующего типа. После этого вызов подпрограмм библиотеки может выполняться путем обращения к процедурным переменным. Для завершения работы с библиотекой необходимо вызвать функцию **FreeLibrary**.

Ниже приведено краткое описание функций **LoadLibrary**, **FreeLibrary** и **GetProcAddress**.

LoadLibrary(LibFileName: PChar): HModule — загружает в оперативную память библиотеку, которая хранится на диске в файле с именем **LibFileName**. При успешном выполнении функция возвращает числовой описатель библиотеки, который должен использоваться в дальнейшем для управления библиотекой. Если при загрузке библиотеки призошла какая-нибудь ошибка, то возвращается нулевое значение. Если аргумент **LibFileName** содержит имя файла без маршрута, то этот файл ищется в следущих каталогах: в каталоге, из которого была запущена главная программа, в текущем каталоге, в системном каталоге операционной системы Windows (его точный маршрут можно узнать вызовом функции **GetSystemDirectory**), в каталоге, по которому установлена операционная система (его точный маршрут можно узнать вызовом функции **GetWindowsDirectory**), а также в каталогах, перечисленных в переменной окружения **PATH**.

FreeLibrary(LibModule: HModule): Bool — выгружает библиотеку, заданную описателем **LibModule**, из оперативной памяти и освобождает занимаемые библиотекой ресурсы системы.

GetProcAddress(Module: HModule; ProcName: PChar): Pointer — возвращает адрес подпрограммы с именем **ProcName** в библиотеке с описателем **Module**. Если подпрограмма с именем **ProcName** в библиотеке не существует, то функция возвращает значение **nil** (пустой указатель).

Приведенная ниже программа **TestDynamicImport** аналогична по функциональности программе **TestStaticImport**, но вместо статического импорта использует технику динамического импорта:

```
program TestDynamicImport;

{$APPTYPE CONSOLE}

uses
  Windows;

type
  TBubleSortProc = procedure (var Arr: array of Integer); stdcall;
  TQuickSortProc = procedure (var Arr: array of Integer); stdcall;

var
  BubleSort: TBubleSortProc; // указатель на функцию BubleSort
  QuickSort: TQuickSortProc; // указатель на функцию QuickSort
  LibHandle: HModule;        // описатель библиотеки

  Arr: array [0..9] of Integer;
  I: Integer;

begin
  LibHandle := LoadLibrary('SortLib.dll');
  if LibHandle <> 0 then
  begin
    @BubleSort := GetProcAddress(LibHandle, 'BubleSortIntegers');
    @QuickSort := GetProcAddress(LibHandle, 'QuickSortIntegers');
    if (@BubleSort <> nil) and (@QuickSort <> nil) then
    begin
      Randomize;
      for I := Low(Arr) to High(Arr) do
        Arr[I] := Random(100);
      BubleSort(Arr);
      for I := Low(Arr) to High(Arr) do
        Write(Arr[I], ' ');
      Writeln;
      for I := Low(Arr) to High(Arr) do
        Arr[I] := Random(100);
      QuickSort(Arr);
      for I := Low(Arr) to High(Arr) do
        Write(Arr[I], ' ');
      Writeln;
    end
    else
      Writeln('Ошибка отсутствия процедуры в библиотеке.');
    FreeLibrary(LibHandle);
  end
  else
    Writeln('Ошибка загрузки библиотеки.');
  Writeln('Press Enter to exit...');
  Readln;
end.
```

В программе определены два процедурных типа данных, которые по списку параметров и правилу вызова (**stdcall**) соответствуют подпрограммам сортировки **BubleSort** и **QuickSort** в библиотеке:

```
type
  TBubleSortProc = procedure (var Arr: array of Integer); stdcall;
  TQuickSortProc = procedure (var Arr: array of Integer); stdcall;
```

Эти типы данных нужны для объявления процедурных переменных, в которых сохраняются адреса подпрограмм:

```
var
  BubleSort: TBubleSortProc;
  QuickSort: TQuickSortProc;
```

В секции **var** объявлена также переменная для хранения целочисленного описателя библиотеки, возвращаемого функцией **LoadLibrary**:

```
var
  ...
  LibHandle: HModule;
```

Программа начинает свою работу с того, что вызывает функцию **LoadLibrary**, в которую передает имя файла DLL-библиотеки. Функция возвращает описатель библиотеки, который сохраняется в переменной **LibHandle**.

```
  LibHandle := LoadLibrary('SortLib.dll');
  if LibHandle <> 0 then
  begin
    ...
  end
```

Если значение описателя отлично от нуля, значит библиотека была найдена на диске и успешно загружена в оперативную память. Убедившись в этом, программа обращается к функции **GetProcAddress** за адресами подпрограмм. Полученные адреса сохраняются в соответствующих процедурных переменных:

```
    @BubleSort := GetProcAddress(LibHandle, 'BubleSortIntegers');
    @QuickSort := GetProcAddress(LibHandle, 'QuickSortIntegers');
```

Обратите внимание на использование символа **@** перед именем каждой переменной. Он говорит о том, что выполняется не вызов подпрограммы, а работа с ее адресом.

Если этот адрес отличен от значения **nil**, значит подпрограмма с указанным именем была найдена в библиотеке и ее можно вызвать путем обращения к процедурной переменной:

```
    if (@BubleSort <> nil) and (@QuickSort <> nil) then
    begin
      ...
      BubleSort(Arr);
      ...
      QuickSort(Arr);
      ...
    end
```

По окончании сортировки программа выгружает библиотеку вызовом функции **FreeLibrary**.

Как вы убедились, динамический импорт в сравнении со статическим требует значительно больше усилий на программирование, но он имеет ряд преимуществ:

- Более эффективное использование ресурсов оперативной памяти по той причине, что библиотеку можно загружать и выгружать по мере надобности;
- Динамический импорт помогает в тех случаях, когда некоторые процедуры и функции могут отсутствовать в библиотеке. При статическом импорте такие ситуации обрабатывает операционная система, которая выдает сообщение об ошибке и прекращает работу программы. Однако при динамическом импорте программа сама решает, что ей делать, поэтому она может отключить часть своих возможностей и работать дальше.

Динамический импорт отлично подходит для работы с библиотеками драйверов устройств. Он, например, используется самой средой Delphi для работы с драйверами баз данных.

5.4. Использование библиотеки из программы на языке C++

Созданные в среде Delphi библиотеки можно использовать в других языках программирования, например в языке C++. Язык C++ получил широкое распространение как

язык системного программирования, и в ряде случаев программистам приходится прибегать к нему.

Ниже показано, как выполнить импорт подпрограмм BubleSort и QuickSort в языке C++.

```
extern "C" __declspec(dllimport)
void __stdcall BubleSort(int* Array, int HighIndex);

extern "C" __declspec(dllimport)
void __stdcall QuickSort(int* Array, int HighIndex);
```

Не углубляясь в детали синтаксиса, заметим, что в языке C++ отсутствуют открытые массивы в параметрах подпрограмм. Тем не менее, программист может вызывать такие подпрограммы, основываясь на том, что открытый массив неявно состоит из двух параметров: указателя на начало массива и номера последнего элемента.

5.5. Глобальные переменные и константы

Глобальные переменные и константы, объявленные в библиотеке, не могут быть экспортированы, поэтому если необходимо обеспечить к ним доступ из использующей программы, это нужно делать с помощью функций, возвращающих значение.

Несмотря на то, что библиотека может одновременно подключаться к нескольким программам, ее глобальные переменные не являются общими и не могут быть использованы для обмена данными между программами. На каждое подключение библиотеки к программе, операционная система создает новое множество глобальных переменных, поэтому библиотеке кажется, что она работает лишь с одной программой. В результате программисты избавлены от необходимости согласовывать работу нескольких программ с одной библиотекой.

5.6. Инициализация и завершение работы библиотеки

Инициализация библиотеки происходит при ее подключении к программе и состоит в выполнении секций **initialization** во всех составляющих библиотеку модулях, а также в ее главном программном блоке. Завершение работы библиотеки происходит при отключении библиотеки от программы; в этот момент в каждом модуле выполняется секция **finalization**. Используйте эту возможность тогда, когда библиотека запрашивает и освобождает какие-то системные ресурсы, например файлы или соединения с базой данных. Запрос ресурса выполняется в секции **initialization**, а его освобождение — в секции **finalization**.

Существует еще один способ инициализации и завершения библиотеки, основанный на использовании предопределенной переменной **DllProc**. Переменная **DllProc** хранит адрес процедуры, которая автоматически вызывается при отключении библиотеки от программы, а также при создании и уничтожении параллельных потоков в программах, использующих DLL-библиотеку (потоки обсуждаются в главе 14). Ниже приведен пример использования переменной **DllProc**:

```
library MyLib;

var
  SaveDllProc: TDLLProc;

procedure LibExit(Reason: Integer);
begin
  if Reason = DLL_PROCESS_DETACH then
  begin
    ...                    // завершение библиотеки
  end;
  SaveDllProc(Reason);     // вызов предыдущей процедуры
end;

begin
  ...                      // инициализация библиотеки
  SaveDllProc := DllProc;  // сохранение предыдущей процедуры
  DllProc := @LibExit;     // установка процедуры LibExit
end.
```

Процедура **LibExit** получает один целочисленный аргумент, который уточняет причину вызова. Возможные значения аргумента:

- DLL_PROCESS_DETACH — отключение программы;
- DLL_PROCESS_ATTACH — подключение программы;
- DLL_THREAD_ATTACH — создание параллельного потока;
- DLL_THREAD_DETACH — завершение параллельного потока.

Обратите внимание, что установка значения переменной **DllProc** выполняется в главном программном блоке, причем предыдущее значение сохраняется для вызова "по цепочке".

Мы рекомендуем вам прибегать к переменной **DllProc** лишь в том случае, если библиотека должна реагировать на создание и уничтожение параллельных потоков. Во всех остальных случаях лучше выполнять инициализацию и завершение с помощью секций **initialization** и **finalization**.

5.7. Исключительные ситуации и ошибки выполнения подпрограмм

Для поддержки исключительных ситуаций среда Delphi использует средства операционной системы Window. Поэтому, если в библиотеке возникает исключительная ситуация, которая никак не обрабатывается, то она передается вызывающей программе. Программа может обработать эту исключительную ситуацию самым обычным способом — с помощью операторов **try … except ... end**. Такие правила действуют для программ и DLL-библиотек, созданных в среде Delphi. Если же программа написана на другом языке программирования, то она должна обрабатывать исключение в библиотеке, написанной на языке Delphi как исключение операционной системы с кодом $0EEDFACE. Адрес инструкции, вызвавшей исключение, содержится в первом элементе, а объект, описывающий исключение, — во втором элементе массива **ExceptionInformation**, который является частью системной записи об исключительной ситуации.

Если библиотека не подключает модуль SysUtils, то обработка исключительных ситуаций недоступна. В этом случае при возникновении в библиотеке любой ошибки происходит завершение вызывающей программы, причем программа просто удаляется из памяти и код ее завершения не выполняется. Это может стать причиной побочных ошибок, поэтому если вы решите не подключать к библиотеке модуль SysUtils, позаботьтесь о том, чтобы исключения "не выскальзывали" из подпрограмм библиотеки.

5.8. Общий менеджер памяти

Если выделение и освобождение динамической памяти явно или неявно поделены между библиотекой и программой, то и в библиотеке, и в программе следует обязательно подключить модуль ShareMem. Его нужно указать в секции **uses** первым, причем как в библиотеке, так и в использующей ее программе.

Модуль ShareMem является модулем импорта динамически загружаемой библиотеки Borlndmm.dll, которая должна распространяться вместе с вашей программой. В момент инициализации модуль ShareMem выполняет подмену стандартного менеджера памяти на менеджер памяти из библиотеки Borlndmm.dll. Благодаря этому библиотека и программа могут выделять и освобождать память совместно.

Модуль ShareMem следует подключать еще и в том случае, если между библиотекой и программой происходит передача длинных строк или динамических массивов. Поскольку длинные строки и динамические массивы размещаются в динамической памяти и управляются автоматически (путем подсчета количества ссылок), то блоки памяти для них, выделяемые программой, могут освобождаться библиотекой (а также наоборот). Использование единого менеджера памяти из библиотеки Borlndmm.dll избавляет программу и библиотеку от скрытых разрушений памяти.

ПРИМЕЧАНИЕ

Последнее правило не относится к отрытым массивам-параметрам, которые мы использовали в подпрограммах BubleSort и QuickSort при создании библиотеки SortLib.dll.

5.9. Стандартные системные переменные

Как вы уже знаете, в языке Delphi существует стандартный модуль **System**, неявно подключаемый к каждой программе или библиотеке. В этом модуле содержатся предопределенные системные подпрограммы и переменные. Среди них имеется переменная **IsLibrary** с типом Boolean, значение которой равно True для библиотеки и False для обычной программы. Проверив значение переменной **IsLibrary**, подпрограмма может определить, является ли она частью библиотеки.

В модуле System объявлена также переменная **CmdLine: PChar**, содержащая командную строку, которой была запущена программа. Библиотеки не могут запускаться самостоятельно, поэтому для них переменная **CmdLine** всегда содержит значение **nil**.

5.10. Итоги

Прочитав главу, вы наверняка вздохнули с облегчением. Жизнь стала легче: сделал одну уникальную по возможностям библиотеку и вставляй ее во все программы! Нужно подключить к Delphi-программе модуль из другой среды программирования — пожалуйста! И все это делается с помощью динамически загружаемых библиотек. Надеемся, вы освоили технику работы с ними и осилите подключение к своей программме библиотек, написанных не только на языке Delphi, но и на языках C и C++. В следующей главе мы рассмотрим некоторые другие взаимоотношения между программами, включая управление объектами одной программы из другой.

Глава 6. Интерфейсы

При программировании нередко возникает необходимость выполнить обращение к объекту, находящемуся в другом загрузочном модуле, например EXE или DLL. Для решения поставленной задачи компания Microsoft разработала технологию COM (Component Object Model) — компонентную модель объектов. Технология получила такое название благодаря тому, что обеспечивает создание программных компонентов — независимо разрабатываемых и поставляемых двоичных модулей. Поскольку объекты различных

программ разрабатываются на различных языках программирования, например Delphi, C++, Visual Basic и др., технология COM стандартизирует формат взаимодействия между объектами на уровне двоичного представления в оперативной памяти. Согласно технологии COM взаимодействие между объектами осуществляется посредством так называемых интерфейсов. Рассмотрим, что же они собой представляют и как с ними работают.

6.1. Понятие интерфейса

Из предыдущих глав вы уже знаете, что собой представляет объект. Представьте, что получится, если из объекта убрать поля и код всех методов. Останется лишь *интерфейс* — заголовки методов и описания свойств. Схематично понятие интерфейса можно представить в виде формулы:

Интерфейс = Объект – Реализация

В отличие от объекта интерфейс сам ничего “не помнит” и ничего “не умеет делать”; он является всего лишь "разъемом" для работы с объектом. Объект может поддерживать много интерфейсов и выступать в разных ролях в зависимости от того, через какой интерфейс вы его используете. Совершенно различные по структуре объекты, поддерживающие один и тот же интерфейс, являются взаимозаменяемыми. Не важно, есть у объектов общий предок или нет. В данном случае интерфейс служит их дополнительным общим предком.

6.2. Описание интерфейса

В языке Delphi интерфейсы описываются в секции **type** глобального блока. Описание начинается с ключевого слова **interface** и заканчивается ключевым словом **end**. По форме объявления интерфейсы похожи на обычные классы, но в отличие от классов:

- интерфейсы не могут содержать поля;
- интерфейсы не могут содержать конструкторы и деструкторы;
- все атрибуты интерфейсов являются общедоступными (public);
- все методы интерфейсов являются абстрактными (virtual, abstract).

Приведем пример интерфейса и сразу заметим, что интерфейсам принято давать имена, начинающиеся с буквы I (от англ. Interface):

```
type
  ITextReader = interface
    // Методы
    function NextLine: Boolean;
    // Свойства
    property Active: Boolean;
    property ItemCount: Integer;
    property Items[Index: Integer]: string;
    property EndOfFile: Boolean;
  end;
```

Интерфейс **ITextReader** предназначен для считывания табличных данных из текстовых источников. В главе 3 мы уже создавали объекты, которые умеют это делать, поэтому назначение методов и свойств должно быть вам понятно. Непонятно пока другое — зачем вообще нужен интерфейс для доступа к табличным данным, если уже есть готовый класс **TTextReader** с требуемой функциональностью.

Объяснение состоит в следующем. Не определив интерфейс **ITextReader**, невозможно разместить класс **TTextReader** в DLL-библиотеке и обеспечить доступ к нему из EXE-программы. Создавая DLL-библиотеку, мы с помощью оператора **uses** должны включить модуль **ReadersUnit** в проект библиотеки. Создавая EXE-программу, мы должны включить модуль **ReadersUnit** и в нее, чтобы воспользоваться описанием класса **TTextReader**. Но

тогда весь программный код класса попадет внутрь EXE-файла, а это именно то, от чего мы хотим избавиться. Решение проблемы обеспечивается введением понятия интерфейса.

Чтобы вам было легче разобраться с интерфейсом **ITextReader**, мы привели его незаконченный вариант. Компиляция интерфейса в таком виде приведет к ошибкам: для свойств не указаны методы чтения и записи. Полное описание интерфейса выглядит так:

```
type
  ITextReader = interface
    // Методы
    function NextLine: Boolean;
    procedure SetActive(const Active: Boolean);
    function GetActive: Boolean;
    function GetItemCount: Integer;
    function GetItem(Index: Integer): string;
    function GetEndOfFile: Boolean;
    // Свойства
    property Active: Boolean read GetActive write SetActive;
    property Items[Index: Integer]: string read GetItem; default;
    property ItemCount: Integer read GetItemCount;
    property EndOfFile: Boolean read GetEndOfFile;
  end;
```

Поскольку интерфейс не может содержать поля, все его свойства отображены на его методы.

6.3. Расширение интерфейса

Новый интерфейс можно создать с нуля, а можно создать путем расширения уже существующего интерфейса. Во втором случае в описании интерфейса после слова **interface** указывается имя базового интерфейса:

```
type
  IExtendedTextReader = interface(ITextReader)
    procedure SkipLines(Count: Integer);
  end;
```

Определенный таким образом интерфейс включает все методы и свойства своего предшественника и добавляет к ним свои собственные. Несмотря на синтаксическое сходство с наследованием классов, расширение интерфейсов имеет другой смысл. В классах наследуется реализация, а в интерфейсах просто расширяется набор методов и свойств.

В языке Delphi существует предопределенный интерфейс **IInterface**, который служит неявным базовым интерфейсом для всех остальных интерфейсов. Это означает, что объявление

```
type
  ITextReader = interface
    ...
  end;
```

эквивалентно следующему:

```
type
  ITextReader = interface(IInterface)
    ...
  end;
```

Мы рекомендуем использовать вторую, более полную форму записи.

Описание интерфейса **IInterface** находится в стандартном модуле **System**:

```
type
  IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```

Непонятная последовательность нулей и других цифр в квадратных скобках — это так называемый глобально-уникальный идентификатор интерфейса. Мы к нему еще вернемся, а сейчас рассмотрим методы.

Методы интерфейса **IInterface** явно или неявно попадают во все интерфейсы и имеют особое назначение. Метод **QueryInterface** нужен для того, чтобы, имея некоторый интерфейс, запросить у объекта другой интерфейс. Этот метод автоматически вызывается при преобразовании одних интерфейсов в другие. Метод **_AddRef** автоматически вызывается при присваивании значения интерфейсной переменной. Метод **_Release** автоматически вызывается при уничтожении интерфейсной переменной. Последние два метода позволяют организовать подсчет ссылок на объект и автоматическое уничтожение объекта, когда количество ссылок на него становится равным нулю. Вызовы всех трех методов генерируются компилятором автоматически, и вызывать их явно нет необходимости, однако программист должен позаботиться об их реализации.

6.4. Глобально-уникальный идентификатор интерфейса

Интерфейс является особым типом данных: он может быть реализован в одной программе, а использоваться из другой. Для этого нужно обеспечить идентификацию интерфейса при межпрограммном взаимодействии. Понятно, что программный идентификатор интерфейса для этого не подходит — разные программы пишутся разными людьми, а разные люди подчас дают одинаковые имена своим творениям. Поэтому каждому интерфейсу выдается своеобразный «паспорт» — глобально-уникальный идентификатор (Globally Unique Identifier — GUID).

Глобально-уникальный идентификатор — это 16-ти байтовое число, представленное в виде заключенной в фигурные скобки последовательности шестнадцатеричных цифр:

```
{DC601962-28E5-4BF7-9583-0CE22B605045}
```

В среде Delphi глобально-уникальный идентификатор описывается типом данных **TGUID**:

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

Константы с типом **TGUID** разрешено инициализировать строковым представлением глобально-уникального идентификатора. Компилятор сам преобразует строку в запись с типом **TGUID**. Пример:

```
const
  InterfaceID: TGUID = '{DC601962-28E5-4BF7-9583-0CE22B605045}';
```

Если глобально-уникальный идентификатор назначается интерфейсу, то он записывается после ключевого слова **interface** и заключается в квадратные скобки, например:

```
type
  IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    ...
  end;
```

В будущем нашему интерфейсу **ITextReader** понадобится глобально-уникальный идентификатор. Но как его выбрать так, чтобы он оказался уникальным? Очень просто — нажмите в редакторе кода комбинацию клавиш **Ctrl+Shift+G**.

```
type
  ITextReader = interface
    ['{DC601962-28E5-4BF7-9583-0CE22B605045}'] // Результат нажатия Ctrl+Shift+G
    ...
  end;
```

Генерация глобально-уникальных идентификаторов осуществляется системой Windows по специальному алгоритму, в котором задействуется адрес сетевого адаптера, текущее время и генератор случайных чисел. Можете смело полагаться на уникальность всех получаемых идентификаторов.

Наличие глобально-уникального идентификатора в описании интерфейса не является обязательным, однако использование интерфейса без такого идентификатора ограничено, например, запрещено использовать оператор **as** для преобразования одних интерфейсов в другие.

Если у интерфейса есть глобально-уникальный идентификатор, то программный идентификатор интерфейса можно использовать там, где ожидается тип данных **TGUID**, например:

```
const
  IID_ITextReader: TGUID = '{DC601962-28E5-4BF7-9583-0CE22B605045}';

function TestInterface(const IID: TGUID): Boolean;

begin
  ...
  TestInterface(ITextReader);
  // эквивалентно
  TestInterface(IID_ITextReader);
  ...
end;
```

6.5. Реализация интерфейса

Интерфейс бесполезен до тех пор, пока он не реализован. Реализацией интерфейса занимается класс. Если класс реализует интерфейс, то интерфейс может использоваться для доступа к объектам этого класса. При объявлении класса имя реализуемого интерфейса записывается через запятую после имени базового класса:

```
type
  TTextReader = class(TObject, ITextReader)
    ...
  end;
```

Такая запись означает, что класс **TTextReader** унаследован от класса **TObject** и реализует интерфейс **ITextReader** (см. рисунок 6.1).

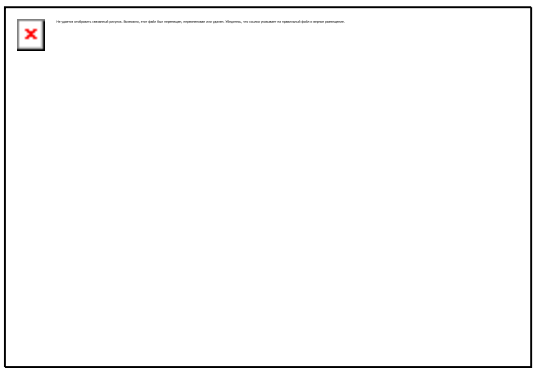


Рисунок 6.1. Класс TTextReader унаследован от класса TObject и реализует интерфейс ITextReader. Сплошными линиями отмечено наследование классов, а пунктирной линией — реализация интерфейса классом.

Класс, реализующий интерфейс, должен содержать код для всех методов интерфейса. Класс **TTextReader** в модуле **ReadersUnit** (см. главу 3) вроде бы содержит код для всех методов интерфейса **ITextReader**, и все, что нужно сделать, — это добавить имя интерфейса в заголовок класса. Сделайте это в модуле **ReadersUnit**:

```
unit ReadersUnit;

interface

type
  ITextReader = interface
    ...
  end;

  TTextReader = class(TObject, ITextReader)
    ...
  end;
```

Если класс содержит только часть методов интерфейса, то недостающие методы придется добавить. Так в интерфейсе **ITextReader** описан метод **GetActive**, а в классе **TTextReader** такого метода нет. Добавьте метод **GetActive** в класс **TTextReader**:

```
type
  TTextReader = class(TObject, ITextReader)
    ...
    function GetActive: Boolean;
    ...
  end;

function TTextReader.GetActive: Boolean;
begin
  Result := FActive;
end;
```

Но это еще не все. Мы совсем забыли о методах **QueryInterface**, **_AddRef** и **_Release**, которые тоже должны быть реализованы. К счастью, вам нет необходимости ломать голову над реализацией этих методов, поскольку разработчики системы Delphi уже позаботились об этом. Стандартная реализация методов интерфейса **IInterface** находится в классе **TInterfacedObject**. Мы его рассмотрим ниже, а сейчас просто унаследуем класс **TTextReader** от класса **TInterfacedObject** — и он получит готовую реализацию методов **QueryInterface**, **_AddRef** и **_Release**.

```
type
  TTextReader = class(TInterfacedObject, ITextReader)
    ...
  end;
```

Теперь реализация интерфейса **ITextReader** полностью завершена и можно переходить к использованию объектов класса **TTextReader** через этот интерфейс.

6.6. Использование интерфейса

Для доступа к объекту через интерфейс нужна интерфейсная переменная:

```
var
  Intf: ITextReader;
```

Интерфейсная переменная занимает в оперативной памяти четыре байта, хранит ссылку на интерфейс объекта и автоматически инициализируется значением **nil**.

Перед использованием интерфейсную переменную инициализируют значением объектной переменной:

```
var
  Obj: TTextReader;  // объектная переменная
  Intf: ITextReader; // интерфейсная переменная
begin
  ...
  Intf := Obj;
  ...
end;
```

После инициализации интерфейсную переменную **Intf** можно использовать для вызова методов объекта **Obj**:

```
Intf.Active := True; // -> Obj.SetActive(True);
Intf.NextLine;       // -> Obj.NextLine;
```

Через интерфейсную переменную доступны только те методы и свойства объекта, которые есть в интерфейсе:

```
Intf.Free; // Ошибка! У интерфейса ITextReadaer нет метода Free.
Obj.Free;  // Метод Free можно вызвать только так.
```

6.7. Реализация нескольких интерфейсов

Один класс может содержать реализацию нескольких интерфейсов. Такая возможность позволяет воплотить в классе несколько понятий. Например, класс **TTextReader** — "считыватель табличных данных" — может выступить еще в одной роли — "считыватель строк". Для этого он должен реализовать интерфейс **IStringIterator**:

```
type
  IStringIterator = interface
    function Next: string;
    function Finished: Boolean;
  end;
```

Интерфейс **IStringIterator** предназначен для последовательного доступа к списку строк. Метод **Next** возвращает очередную строку из списка, метод **Finished** проверяет, достигнут ли конец списка.

Реализуем интерфейс **IStringIterator** в классе **TTextReader** таким образом, чтобы последовательно считывались значения из ячеек таблицы. Например, представьте, что в некотором файле дана таблица:

```
Aaa Bbb Ccc
Ddd Eee Fff
Ggg Hhh Iii
```

Чтение этой таблицы через интерфейс **IStringIterator** вернет следующую последовательность строк:

```
Aaa
Bbb
Ccc
Ddd
Eee
Fff
Ggg
Hhh
Iii
```

Ниже приведен программный код, обеспечивающий поддержку интерфейса **IStringIterator** в классе **TTextReader**:

```
type
  TTextReader = class(TInterfacedObject, ITextReader, IStringIterator)
    FColumnIndex: Integer;
    function Next: string;
    function Finished: Boolean;
    ...
  end;
...

function TTextReader.Next: string;
begin
  if FColumnIndex = ItemCount then // Если пройден последний элемент текущей
строки,
  begin                            // то переходим к следующей строке таблицы
    NextLine;
    FColumnIndex := 0;
  end;
  Result := Items[FColumnIndex];
  FColumnIndex := FColumnIndex + 1;
end;

function TTextReader.Finished: string;
begin
  Result := EndOfFile and (FColumnIndex = ItemCount);
end;
```

Теперь объекты класса **TTextReader** совместимы сразу с тремя типами данных: **TInterfacedObject**, **ITextReader**, **IStringIterator**.

```
var
  Obj: TTextReader;
  Reader: ITextReader;
  Iterator: IStringIterator;
begin
  ...
  Reader := Obj;   // Правильно
  Iterator := Obj; // Правильно
  ...
end;
```

В одном случае объект класса **TTextReader** рассматривается как считыватель табличных данных, а в другом случае — как обычный список строк с последовательным доступом. Например, если есть две процедуры:

```
procedure LoadTable(Reader: ITextReader);
procedure LoadStrings(Iterator: IStringIterator);
```

то объект класса **TTextReader** можно передать в обе процедуры:

```
LoadTable(Obj);   // Obj воспринимается как ITextReader
LoadStrings(Obj); // Obj воспринимается как IStringIterator
```

6.8. Реализация интерфейса несколькими классами

Несколько совершенно разных классов могут содержать реализацию одного и того же интерфейса. С объектами таких классов можно работать так, будто у них есть общий базовый класс. Интерфейс выступает аналогом общего базового класса.

Рассмотрим пример. Представьте, что есть два класса: **TTextReader** и **TIteratableStringList**:

```
type
  TTextReader = class(TInterfacedObject, ITextReader, IStringIterator)
    ...
  end;

  TIteratableStringList = class(TStringList, IStringIterator)
    ...
  end;
```

Схематично полученную иерархию классов можно представить так (рисунок 6.2):

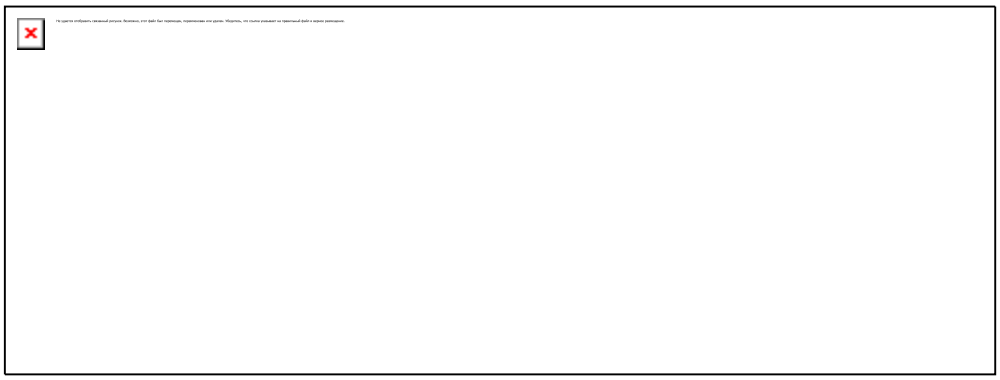


Рисунок 6.2. Иерархия классов, реализующих интерфейсы. Сплошными линиями отмечено наследование классов, а пунктирными линиями — реализация интерфейсов классами.

Объекты классов **TTextReader** и **TIteratableStringList** несовместимы между собой. Тем не менее, они совместимы с переменными типа **IStringIterator**. Это значит, что если есть процедура:

```
procedure LoadStrings(Iterator: IStringIterator);
```

то вы можете передавать ей объекты обоих упомянутых классов в качестве аргумента:

```
var
  ReaderObj: TTextReader;
  StringsObj: TIteratableStringList;
begin
  ...
  LoadStrings(ReaderObj);  // Все правильно
  LoadStrings(StringsObj); // Все правильно
  ...
end;
```

6.9. Связывание методов интерфейса с методами класса

Метод интерфейса связывается с методом класса по имени. Если имена по каким-то причинам не совпадают, то можно связать методы явно с помощью специальной конструкции языка Delphi.

Например, в классе **TTextReader** добавлены методы **Next** и **Finished** для поддержки интерфейса **IStringIterator**. Согласитесь, что существование в одном классе методов **Next** и **NextLine** вносит путаницу. По названию метода **Next** не понятно, что для этого метода является следующим элементом. Поэтому уточним название метода в классе **TTextReader** и воспользуемся явным связыванием методов, чтобы сохранить имя **Next** в интерфейсе **IStringIterator**:

```
type
  TTextReader = class(TInterfacedObject, ITextReader, IStringIterator)
    ...
    function NextItem: string;
    function IStringIterator.Next := NextItem; // Явное связывание
  end;
```

При работе с объектами класса **TTextReader** через интерфейс **IStringIterator** вызов метода **Next** приводит к вызову метода **NextItem**:

```
var
  Obj: TTextReader;
  Intf: IStringIterator;
begin
  ...
  Intf := Obj;
  Intf.Next; // -> Obj.NextItem;
  ...
end;
```

Очевидно, что связываемые методы должны совпадать по сигнатуре (списку параметров и типу возвращаемого значения).

6.10. Реализация интерфейса вложенным объектом

Случается, что реализация интерфейса содержится во вложенном объекте класса. Тогда не требуется программировать реализацию интерфейса путем замыкания каждого метода интерфеса на соответствующий метод вложенного объекта. Достаточно делегировать реализацию интерфейса вложенному объекту с помощью директивы **implements**:

```
type
  TTextParser = class(TInterfacedObject, ITextReader)
    ...
    FTextReader: ITextReader;
    property TextReader: ITextReader read FTextReader implements ITextReader;
    ...
  end;
```

В этом примере интерфейс **ITextReader** в классе **TTextParser** реализуется не самим классом, а его внутренней переменной **FTextReader**.

Очевидно, что внутренний объект должен быть совместим с реализуемым интерфейсом.

6.11. Совместимость интерфейсов

Совместимость интерфейсов подчиняется определенным правилам. Если интерфейс создан расширением уже существующего интерфейса:

```
type
  IExtendedTextReader = interface(ITextReader)
    ...
  end;
```

то интерфейсной переменной базового типа может быть присвоено значение интерфейсной переменной производного типа:

```
var
  Reader: ITextReader;
  ExtReader: IExtendedTextReader;
begin
  ...
  Reader := ExtReader; // Правильно
  ...
end;
```

Но не наоборот:

```
ExtReader := Reader; // Ошибка!
```

Правило совместимости интерфейсов чаще всего применяется при передаче параметров в процедуры и функции. Например, если процедура работает с переменными типа **ITextReader**,

```
procedure LoadFrom(const R: ITextReader);
```

то ей можно передать переменную типа **IExtendedTextReader**:

```
LoadFrom(ExtReader);
```

Заметим, что любая интерфейсная переменная совместима с типом данных **IInterface** — прародителем всех интерфейсов.

6.12. Совместимость класса и интерфейса

Интерфейсной переменной можно присвоить значение объектной переменной при условии, что объект (точнее его класс) реализует упомянутый интерфейс:

```
var
  Intf: ITextReader; // интерфейсная переменная
  Obj: TTextReader;  // объектная переменная
begin
  ...
  Intf := Obj; // В переменную Intf копируется ссылка на объект Obj
  ...
end;
```

Такая совместимость сохраняется в производных классах. Если класс реализует некоторый интерфейс, то и все его производные классы совместимы с этим интерфейсом (см. рисунок 6.3):

```
type
  TTextReader = class(TInterfacedObject, ITextReader)
    ...
  end;

  TDelimitedReader = class(TTextReader)
    ...
  end;

var
  Intf: ITextReader;         // интерфейсная переменная
  Obj: TDelimitedReader;     // объектная переменная
begin
  ...
  Intf := Obj;
  ...
end;
```

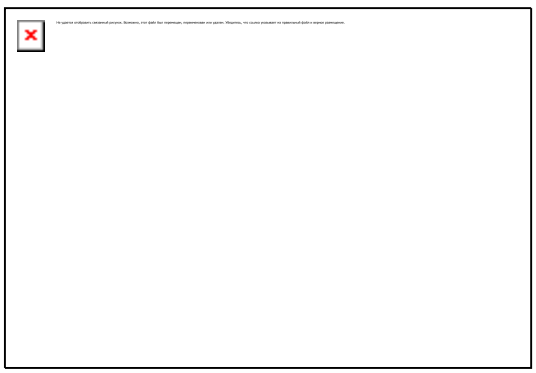


Рисунок 6.3. Классы TTextReader, TDelimitedReader и TFixedReader совместимы с интерфейсом ITextReader

Однако, если класс реализует производный интерфейс, то это совсем не означает, что он совместим с базовым интерфейсом (см. рисунок 6.4):

```
type
  ITextReader = interface(IInterface)
    ...
  end;

  IExtendedTextReader = interface(ITextReader)
    ...
  end;

  TExtendedTextReader = class(TInterfacedObject, IExtendedTextReader)
    ...
  end;

var
  Obj: TExtendedTextReader;
  Intf: ITextReader;
begin
  ...
  Intf := Obj;  // Ошибка! Класс TExtendedTextReader не реализует
                // интерфейс ITextReader.
  ...
end;
```

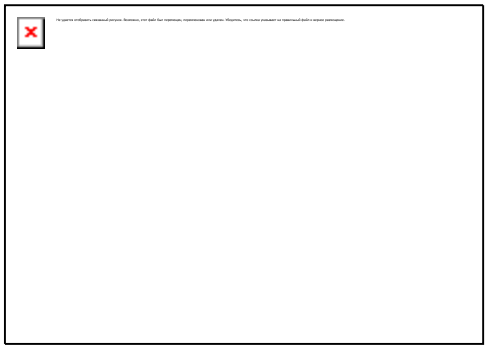


Рисунок 6.4. Класс TExtendedTextReader совместим лишь с интерфейсом IExtendedTextReader

Для совместимости с базовым интерфейсом нужно реализовать этот интерфейс явно:

```
type
  TExtendedTextReader = class(TInterfacedObject, ITextReader, IExtendedTextReader)
    ...
  end;
```

Теперь класс **TExtendedTextReader** совместим и с интерфейсом **ITextReader**, поэтому следующее присваивание корректно:

```
  Intf := Obj;
```

Исключением из только что описанного правила является совместимость всех снабженных интерфейсами объектов с интерфейсом **IInterface**:

```
var
  Obj: TExtendedTextReader;
  Intf: IInterface;
begin
  ...
  Intf := Obj;  // Правильно, IInterface - особый интерфейс.
  ...
end;
```

6.13. Получение интерфейса через другой интерфейс

Через интерфейсную переменную у объекта всегда можно запросить интерфейс другого типа. Для этого используется оператор **as**, например:

```
var
  Intf: IInterface;
begin
  ...
  with Intf as ITextReader do
    Active := True;
  ...
end;
```

Если объект действительно поддерживает запрашиваемый интерфейс, то результатом является ссылка соответствующего типа. Если же объект не поддерживает интерфейс, то возникает исключительная ситуация **EIntfCastError**.

В действительности оператор **as** преобразуется компилятором в вызов метода **QueryInterface**:

```
var
  Intf: IInterface;
  IntfReader: ITextReader;
...
  IntfReader := Intf as ITextReader; // Intf.QueryInterface(ITextReader,
IntfReader);
```

Напомним, что метод **QueryInterface** описан в интерфейсе **IInterface** и попадает автоматически во все интерфейсы. Стандартная реализация этого метода находится в классе **TInterfacedObject**.

6.14. Механизм подсчета ссылок

Механизм подсчета ссылок на объект предназначен для автоматического уничтожения неиспользуемых объектов. Неиспользуемым считается объект, на который не ссылается ни одна интерфейсная переменная.

Подсчет ссылок на объект обеспечивают методы **_AddRef** и **_Release** интерфейса **IInterface**. При копировании значения интерфейсной переменной вызывается метод **_AddRef**, а при уничтожении интерфейсной переменной — метод **_Release**. Вызовы этих методов генерируются компилятором автоматически:

```
var
  Intf, Copy: IInterface;
begin
  ...
  Copy := Intf; // Copy._Release; Intf._AddRef;
  Intf := nil;  // Intf._Release;
end;            // Copy._Release
```

Стандартная реализация методов **_AddRef** и **_Release** находится в классе **TInterfacedObject**. Она достаточно проста и вы легко разберетесь с ней, читая комментарии в исходном тексте.

```
type
  TInterfacedObject = class(TObject, IInterface)
    ...
    FRefCount: Integer;                        // Счетчик ссылок
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
    ...
  end;

function TInterfacedObject._AddRef: Integer;
begin
  Result := InterlockedIncrement(FRefCount);  // Увеличение счетчика ссылок
end;

function TInterfacedObject._Release: Integer;
begin
  Result := InterlockedDecrement(FRefCount);  // Уменьшение счетчика ссылок
  if Result = 0 then                          // Если ссылок больше нет, то
    Destroy;                                  // уничтожение объекта
end;
```

Заметим, что функции **InterlockedIncrement** и **InterlockedDecrement** просто увеличивают значение целочисленной переменной на единицу. В отличие от обычного оператора сложения, они обеспечивают атомарное изменение значения переменной, что очень важно для правильной работы распараллеленных (многопоточных) программ.

Приведенную выше реализацию методов **_AddRef** и **_Release** автоматически получают все наследники класса **TInterfacedObject**, в том числе и классы **TTextReader**, **TDelimitedReader** и **TFixedReader**. Поэтому неиспользуемые объекты классов **TDelimitedReader** и **TFixedReader** тоже автоматически уничтожаются при работе с ними через интерфейсные переменные:

```
var
  Obj: TDelimitedReader;
  Intf, Copy: ITextReader;
begin
  Obj := TDelimitedReader.Create('MyData.del', ';');
  Intf := Obj;   // Obj._AddRef -> Obj.FRefCount = 1
  Copy := Intf;  // Obj._AddRef -> Obj.FRefCount = 2
  ...
  Intf := nil;   // Obj._Release -> Obj.FRefCount = 1
  Copy := nil;   // Obj._Release -> Obj.FRefCount = 0 -> Obj.Destroy
  Obj.Free;      // Ошибка! Объект уже уничтожен и переменная Obj указывает в
никуда
end;
```

Обратите внимание, что объектные переменные не учитываются при подсчете ссылок. Поэтому мы настоятельно рекомендуем избегать смешивания интерфейсных и объектных переменных. Если вы планируете использовать объект через интерфейс, то лучше всего результат работы конструктора сразу присвоить интерфейсной переменной:

```
var
  Intf: ITextReader;
begin
  Intf := TDelimitedReader.Create('MyData.del', ';'); // FRefCount = 1
  ...
  Intf := nil;                                        // FRefCount = 0 -> Destroy
end;
```

Если интерфейс является входным параметром подпрограммы, то при вызове подпрограммы создается копия интерфейсной переменной с вызовом метода **_AddRef**:

```
procedure LoadItems(R: ITextReader);
begin
...
end;

var
  Reader: ITextReader;
begin
  ...
  LoadItems(Reader); // Создается копия переменной Reader и вызывается
Reader._AddRef
end;
```

Копия не создается, если входной параметр описан с ключевым словом **const**:

```
procedure LoadItems(const R: ITextReader);
begin
...
end;

var
  Reader: ITextRedaer;
begin
  ...
  LoadItems(Reader); // Копия не создается, метод _AddRef не вызывается
end;
```

Интерфейсная переменная уничтожается при выходе из области действия переменной, а это значит, что у нее автоматически вызывается метод **_Release**:

```
var
  Intf: ITextRedaer;
begin
  Intf := TDelimitedReader.Create('MyData.del', ';');
  ...
end; // Intf._Release
```

6.15. Представление интерфейса в памяти

Глубокое понимание работы интерфейсов требует знания их технической реализации. Поэтому вам необходимо разобраться в том, как представляется интерфейс в оперативной памяти компьютера, и что стоит за операторами Intf := Obj и Intf.NextLine.

Интерфейс по сути выступает дополнительной таблицей виртуальных методов, ссылка на которую укладывается среди полей объекта (рисунок 6.5). Эта таблица называется *таблицей методов интерфейса*. В ней хранятся указатели на методы класса, реализующие методы интерфейса.

Интерфейсная переменная хранит ссылку на скрытое поле объекта, которое содержит указатель на таблицу методов интерфейса. Когда интерфейсной переменной присваивается значение объектой переменной,

```
Intf := Obj; // где Intf: ITextReader и Obj: TTextReader
```

к адресу объекта добавляется смещение до скрытого поля внутри объекта и этот результат заносится в интерфейсную переменную. Чтобы убедиться в сказанном, посмотрите в отладчике значения Pointer(Obj) и Pointer(Intf) сразу после выполнения оператора Intf := Obj. Эти значения будут разными! Причина в том, что объектная ссылка указывает на начало объекта, а интерфейсная ссылка — на скрытое поле внутри объекта.

Рисунок 6.5. Представление интерфейса в памяти

Алгоритм вызова метода интерфейса такой же, как алгоритм вызова метода класса. Когда через интерфейсную переменную выполняется вызов метода,

```
Intf.NextLine;
```

реализуется следующий алгоритм:

5. Из интерфейсной переменной извлекается адрес (по нему хранится адрес таблицы методов интерфейса);
6. По полученному адресу извлекается адрес таблицы методов интерфейса;
7. На основании порядкового номера метода в интерфейсе из таблицы извлекается адрес соответствующей подпрограммы;
8. Вызывается код, находящийся по этому адресу. Этот код является переходником от метода интерфейса к методу объекта. Его задача — восстановить из ссылки на интерфейс значение указателя Self (путем вычитания заранее известного значения) и выполнить прямой переход на код метода класса.

Обычными средствами процедурного программирования этот алгоритм реализуется так:

```
type
  TMethodTable = array[0..9999] of Pointer;
  TNextLineFunc = function (Self: ITextReader): Boolean;
var
  Intf: ITextReader;              // интерфейсная переменна
  IntfPtr: Pointer;               // адрес внутри интерфейсной переменной
  TablePtr: ^TMethodTable;        // указатель на таблицу методов интерфейса
  MethodPtr: Pointer;             // указатель на метод
begin
  ...
  IntfPtr := Pointer(Intf);       // 1) извлечение адреса из интерфейсной
переменной
  TablePtr := Pointer(IntfPtr^);  // 2) извлечение адреса таблицы методов
интерфейса
  MethodPtr := TablePtr^[3];      // 3) извлечение адреса нужного метода из таблицы
  TNextLineFunc(MethodPtr)(Intf); // 4) вызов метода через переходник
  ...
end.
```

Вся эта сложность скрыта в языке Delphi за понятием интерфейса. Причем несмотря на такое количество операторов в примере, вызов метода через интерфейс в машинном коде выполняется весьма эффективно (всего несколько инструкций процессора), поэтому в подавляющем большинстве случаев потерями на вызов можно пренебречь.

6.16. Применение интерфейса для доступа к объекту DLL-

библиотеки

Если вы поместите свой класс в DLL-библиотеку, то при необходимости использовать его в главной программе столкнетесь с проблемой. Подключение модуля с классом к главной программе приведет к включению в нее кода всех методов класса, т.е. задача выделения класса в DLL-библиотеку не будет решена. Если же не подключить модуль с описанием класса, главная программа вообще не будет знать о существовании класса, и воспользоваться классом будет невозможно. Эта проблема решается с помощью интерфейсов. Покажем это на примере модуля **ReadersUnit**.

Сначала вынесем описание интерфейса **ITextReader** в отдельный модуль (например, **ReaderIntf**), чтобы этот модуль в дальнейшем можно было подключить к главной программе:

```
unit ReadersIntf;

interface
type
  ITextReader = interface(IInterface)
    ...
  end;

implementation

end.
```

Затем удалим описание интерфейса из модуля **ReadersUnit**, а вместо него подключим модуль **ReaderIntf**:

```
unit ReadersUnit;

interface

uses
  ReaderIntf;
...
```

Наконец включим скорректированный модуль **ReadersUnit** в DLL-библиотеку, которую назовем **ReadersLib**:

```
library ReadersLib;

uses
  SysUtils, Classes, ReadersUnit;

{$R *.res}

begin
end.
```

Вроде бы все готово, и теперь в главной программе достаточно подключить модуль **ReaderIntf** и работать с объектами через интерфейс **ITextReader** (рисунок 6.6).

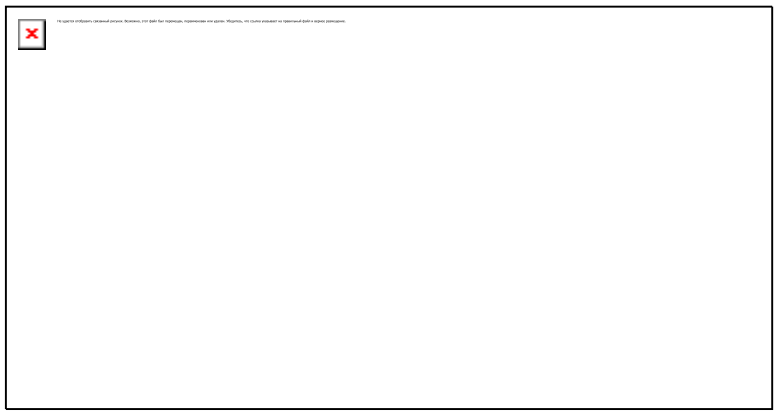


Рисунок 6.6. Схема получения программы и DLL-библиотеки

Но постойте! А как в программе создавать объекты классов, находящихся в DLL-библиотеке? Ведь в интерфейсе нет методов для создания объектов! Для этого определим в DLL-библиотеке специальную функцию и экспортируем ее:

```
library ReadersLib;
...

function GetDelimitedReader(const FileName: string;
  const Delimiter: Char = ';'): ITextReader;
begin
  Result := TDelimitedReader.Create(FileName, Delimiter);
end;

exports
  GetDelimitedReader;

begin
end.
```

В главной программе импортируйте функцию **GetDelimitedReader**, чтобы с ее помощью создавать объекты класса **TDelimitedReader**:

```
program Example;

uses
  ReadersIntf;

function GetDelimitedReader(const FileName: string;
  const Delimiter: Char = ';'): ITextReader;
  external 'ReadersLib.dll' name 'GetDelimitedReader';

var
  Intf: ITextReader;
begin
  Intf := GetDelimitedReader;
  ...
end.
```

Теперь вы знаете, как разместить объекты в DLL-библиотеке. Смело пользуйтесь динамически загружаемыми библиотеками, не теряя преимуществ ООП.

6.17. Итоги

Вы прочитали и усвоили весь материал всех предыдущих глав? Тогда спешим вас поздравить! Можете смело утверждать, что знаете язык программирования Delphi. Что же дальше? Вас ждет новая высота — среда программирования Delphi. Сейчас вы имеете лишь поверхностное представление о ее возможностях. Настало время подготовить себя к профессиональной работе в среде Delphi.

Глава 7. Проект приложения

Решаемая на компьютере задача реализуется в виде прикладной программы, которую для краткости называют приложением. В основе разработки приложения в среде Delphi лежит проект. Центральной частью проекта является форма, на которую помещаются необходимые для решения конкретной задачи компоненты. В такой последовательности — проект - формы - компоненты — мы и рассмотрим процесс создания приложения в среде Delphi. По ходу изложения материала мы будем часто обращаться к примеру с вычислением идеального веса, который был рассмотрен в первой главе. Если вы его забыли, перечитайте первую главу еще раз.

7.1. Проект

7.1.1. Понятие проекта

Приложение собирается из многих элементов: форм, программных модулей, внешних библиотек, картинок, пиктограмм и др. Каждый элемент размещается в отдельном файле и имеет строго определенное назначение. Набор всех файлов, необходимых для создания приложения, называется *проектом*. Компилятор последовательно обрабатывает файлы проекта и строит из них выполняемый файл. Основные файлы проекта можно разделить на несколько типов:

- *Файлы описания форм* — текстовые файлы с расширением DFM, описывающие формы с компонентами. В этих файлах запоминаются начальные значения свойств, установленные вами в окне свойств.
- *Файлы программных модулей* — текстовые файлы с расширением PAS, содержащие исходные программные коды на языке Delphi. В этих файлах вы пишите методы обработки событий, генерируемых формами и компонентами.
- *Главный файл проекта* — текстовый файл с расширением DPR, содержащий главный программный блок. Файл проекта подключает все используемые программные модули и содержит операторы для запуска приложения. Этот файл среда Delphi создает и контролирует сама.

На основании сказанного можно изобразить процесс создания приложения в среде Delphi от постановки задачи до получения готового выполняемого файла (рисунок 7.1):

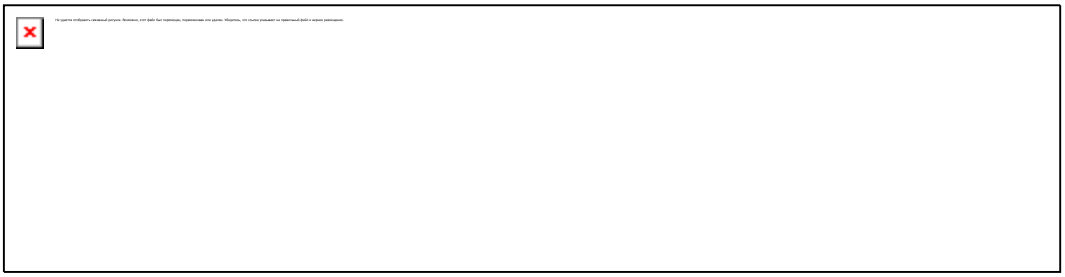


Рисунок 7.1. Процесс создания приложения в среде Delphi

Давайте рассмотрим назначение и внутреннее устройство файлов проекта. Это поможет вам легче ориентироваться в проекте.

7.1.2. Файлы описания форм

Помните, с чего вы начинали знакомство со средой Delphi? Конечно, с формы. Итак, первая составная часть проекта — это текстовый файл с расширением DFM, описывающий форму. В DFM-файле сохраняются значения свойств формы и ее компонентов, установленные вами в окне свойств во время проектирования приложения. Количество DFM-файлов равно количеству используемых в приложении форм. Например, в нашем примере об идеальном весе используется только одна форма, поэтому и DFM-файл только один — Unit1.DFM.

Если вы желаете взглянуть на содержимое DFM-файла, вызовите у формы контекстное меню щелчком правой кнопки мыши и выберите команду **View as Text** (рисунок 7.2).

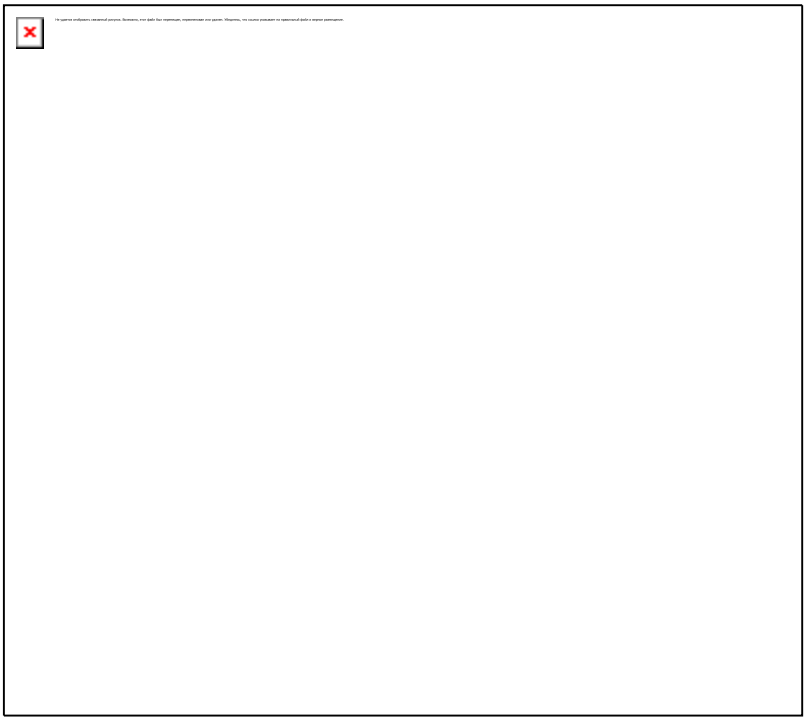


Рисунок 7.2. Переход к текстовому представлению формы с помощью команды View as Text контекстного меню

В ответ среда Delphi вместо графического изображения формы покажет следующий текст в редакторе кода**:**

```
object Form1: TForm1
  Left = 250
  Top = 150
  Width = 400
  Height = 303
  Caption = 'Weight Calculator'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Label1: TLabel
    Left = 64
    Top = 48
    Width = 93
    Height = 13
    Caption = 'Specify your height:'
  end
  object Label2: TLabel
    Left = 64
    Top = 144
    Width = 84
    Height = 13
    Caption = 'Your ideal weight:'
  end
  object Button1: TButton
    Left = 248
    Top = 64
    Width = 75
    Height = 25
    Caption = 'Compute'
    TabOrder = 0
    OnClick = Button1Click
  end
  object Button2: TButton
    Left = 248
    Top = 160
    Width = 75
    Height = 25
    Caption = 'Close'
    TabOrder = 1
  end
  object Edit1: TEdit
    Left = 64
    Top = 64
    Width = 121
    Height = 21
    TabOrder = 2
  end
  object Edit2: TEdit
    Left = 64
    Top = 160
    Width = 121
    Height = 21
    TabOrder = 3
  end
end
```

Несмотря на столь длинный текст описания, разобраться в нем совсем не сложно. Здесь на специальном языке задаются исходные значения для свойств формы Form1 и ее компонентов Button1, Button2, Edit1, Edit2, Label1, Label2. Большего знать не требуется, поскольку вы всегда будете использовать визуальные средства проектирования и работать с графическим представлением формы, а не с текстовым описанием. Раз так, давайте поспешим вернуться к графическому представлению, не внося в текст никаких изменений. Для этого вызовите контекстное меню редактора кода и выберите команду **View as Form** (рисунок 7.3).

Рисунок 7.3. Переход к графическому представлению формы с помощью команды View as Form контекстного меню

На экране снова появится графический образ формы. Если вы все-таки внесли корректировки в текст, то они отразятся на внешнем виде формы.

Файл описания формы (DFM-файл) нужен только на этапе проектирования. При сборке приложения описание формы из DFM-файла помещается в специальную область данных выполняемого файла (область ресурсов). Когда во время работы приложения происходит создание формы, ее описание извлекается из области ресурсов и используется для инициализации формы и ее компонентов. В результате форма отображается на экране так, как вы задали при проектировании.

7.1.3. Файлы программных модулей

Каждой форме в проекте соответствует свой *программный модуль* (unit), содержащий все относящиеся к форме объявления и методы обработки событий, написанные на языке Delphi. Программные модули размещаются в отдельных файлах с расширением PAS. Их количество может превышать количество форм. Почему? Потому, что в ряде случаев программные модули могут и не относиться к формам, а содержать вспомогательные процедуры, функции, классы и проч. Наша задача об идеальном весе очень простая, поэтому в ней имеется только один программный модуль, связанный с формой. Не поленитесь изучить его внимательно:

```
unit Unit1;

interface

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

type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  // Алгоритм вычисление идеального веса
  Edit2.Text := IntToStr(StrToInt(Edit1.Text) - 100 - 10);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;

end.
```

Дадим необходимые комментарии к тексту программного модуля. В самом начале после ключевого слова **unit** записывается имя модуля

```
unit Unit1;
```

Ни в коем случае не изменяйте это имя вручную. Среда Delphi требует, чтобы имя модуля совпадало с именем файла, поэтому если вы хотите переименовать модуль, сохраните его в файле с новым именем, воспользовавшись командой меню **File | Save As...**. Среда Delphi сама подставит после слова **unit** новое имя. После этого удалите старый модуль.

Содержание интерфейсной секции модуля (**interface**) начинается с подключения стандартных модулей библиотеки VCL, в которых определены часто вызываемые подпрограммы и классы помещенных на форму компонентов.

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

Среда Delphi формирует список модулей без вашего участия и автоматически пополняет его, когда вы добавляете на форму новые компоненты. Тем не менее, список подключенных модулей можно изменять прямо в редакторе кода (вручную).

Смотрим дальше. В разделе описания типов (**type**) объявлен класс формы. По умолчанию он называется **TForm1** и порожден от стандартного класса **TForm**.

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Edit1: TEdit;
    Edit2: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Помещенные на форму компоненты представлены полями формы. У нас на форме шесть компонентов, поэтому и полей в описании класса тоже шесть. Имена полей совпадают с именами компонентов, заданными в окне свойств.

После полей следуют заголовки методов обработки событий. Название каждого такого метода среда Delphi формирует автоматически на основании имени компонента и имени генерируемого им события. Например, для кнопки **Button1** метод обработки события **OnClick** называется **Button1Click**.

Обратите внимание, что поля, представляющие компоненты формы, а также методы обработки событий получают атрибут видимости **published** (он принимается по умолчанию для всех наследников **TForm**). Благодаря этому вы можете работать с ними на визуальном уровне, например, видеть их имена в окне свойств. Поскольку среда Delphi сама управляет содержимым секции **published**, никогда не модифицируйте эту секцию вручную (в редакторе кода), пользуйтесь визуальными инструментами: палитрой компонентов и окном свойств. Запомните:

- когда вы помещаете на форму компоненты, среда Delphi сама добавляет соответствующие поля в описание класса формы, а когда вы удаляете компоненты с формы, среда удаляет поля из описания класса;
- когда вы определяете в форме или компонентах обработчики событий, среда Delphi сама определяет в классе соответствующие методы, а когда вы удаляете весь код из методов обработки событий, среда Delphi удаляет и сами методы.

Для вашего удобства в классе формы заранее объявлены пустые секции **private** и **public**, в которых вы можете размещать любые вспомогательные поля, методы и свойства. Среда Delphi их "в упор не видит", поэтому с ними можно работать только на уровне программного кода. Вы можете помещать в секцию **private** атрибуты, которые нужны только самой форме, а в секцию **public** — атрибуты, которые нужны еще и другим формам и модулям.

После описания класса идет объявление собственно объекта формы:

```
var
  Form1: TForm1;
```

Переменная **Form1** — это ссылка на объект класса **TForm1**, конструирование которого выполняется в главном файле проекта – DPR-файле (см. далее).

На этом содержание интерфейсной секции модуля заканчивается и начинается раздел реализации (**implementation**). Сначала в нем подключается файл описания формы:

```
{$R *.dfm}
```

Пожалуйста, не подумайте, что эта директива подключает все файлы с расширением DFM. Подключается лишь один DFM-файл, в котором описана форма данного модуля. Имя DFM-файла получается заменой звездочки на имя модуля, в котором записана директива.

Далее следует реализация методов обработки событий. Пустые заготовки для них среда Delphi создает сама одновременно с добавлением заголовков в класс формы. Вы же наполняете их содержанием.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  // Алгоритм вычисление идеального веса
  Edit2.Text := IntToStr(StrToInt(Edit1.Text) - 100 - 10);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

Внимание! Если вы хотите удалить метод обработки события и убрать ссылки на него, просто сделайте метод пустым — удалите весь написанный вами код, включая комментарии, и объявления локальных переменных. При сохранении или компиляции проекта среда Delphi сама выбросит из текста пустые методы.

При внимательном изучении исходного текста модуля остается невыясненным один вопрос: как обеспечивается вызов методов **Button1Click** и **Button2Click** при нажатии на форме кнопок, ведь в тексте модуля отсутствует даже намек на это. Все очень просто. Загляните в DFM-файл. Кроме установки значений свойств вы найдете установку и обработчиков событий.

```
object Button1: TButton
  ...
  OnClick = Button1Click
end
object Button2: TButton
  ...
  OnClick = Button2Click
end
```

Благодаря этому описанию обеспечивается привязка методов формы к соответствующим событиям. Кстати, смысл приведенного фрагмента описания становится более прозрачным, если вспомнить, что события в языке Delphi — это на самом деле свойства, но их значениями являются указатели на методы. Таким образом, установка событий мало чем отличается от установки свойств формы, ведь по природе это одно и то же.

Мы достаточно глубоко погрузились во внутреннее устройство файлов описания форм и файлов программных модулей и, признаемся, сделали это намеренно, чтобы дать вам полное понимание вопроса, не заставляя принимать на веру далеко неочевидные вещи. А сейчас пора подняться на уровень проекта и посмотреть, что же объединяет все эти файлы.

7.1.4. Главный файл проекта

Для того чтобы компилятор знал, какие конкретно файлы входят в проект, необходимо какое-то организующее начало. И оно действительно есть. Это так называемый *файл проекта*, имеющий расширение DPR (сокр. от Delphi Project). Он представляет собой главный программный файл на языке Delphi, который подключает с помощью оператора **uses** все файлы модулей, входящих в проект. Для каждого проекта существует только один DPR-файл.

Когда вы по команде **File | New | Application** начинаете разработку нового приложения, среда Delphi автоматически создает файл проекта. По мере создания новых форм содержимое этого файла видоизменяется автоматически. Когда вы закончите работу и будете готовы компилировать проект, в DPR-файле будет находиться перечень программных модулей, которые будут поданы на вход компилятору. Чтобы увидеть содержимое DPR-файла нашего приложения, вычисляющего идеальный вес, выберите в меню среды Delphi

команду **Project | View Source**. В редакторе кода появится новая страница со следующим текстом:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Немного прокомментируем этот текст. Подключение модуля **Forms** обязательно для всех программ, так как в нем содержится определение объекта **Application**. Этот объект лежит в основе любого графического приложения и доступен на протяжении всей его работы.

Подключаемый следом модуль Unit1 содержит определение формы. Название формы приводится в фигурных скобках. Директива **in** указывает на то, что модуль является необходимой частью проекта и существует в виде исходного текста на языке Delphi.

Директива {$R *.res} подключает к результирующему выполняемому файлу так называемые ресурсы, в данном случае значок приложения. Этот значок будет виден на Панели Задач.

Дальше следует главный программный блок, содержащий вызовы трех методов объекта **Application**. Вызов метода **Initialize** подготавливает приложение к работе, метод **CreateForm** загружает и инициализирует форму **Form1**, а метод **Run** активизирует форму и начинает выполнение приложения. Фактически время работы метода **Run** — это время работы приложения. Выход из метода **Run** происходит тогда, когда пользователь закрывает главную форму приложения; в результате приложение завершается.

Внимание! Никогда не изменяйте DPR-файл вручную. Оставьте эту работу для среды Delphi. Добавление и удаление модулей, а также управление созданием форм осуществляется с помощью команд и диалоговых окон среды.

7.1.5. Другие файлы проекта

Выше мы рассмотрели основные файлы проекта. Кроме них существует ряд дополнительных файлов:

- Файл с расширением DOF (сокр. от Delphi Options File), где хранятся заданные программистом параметры компиляции и сборки проекта;
- Файл с расширением DSK (сокр. от англ. Desktop), где хранятся настройки среды Delphi для данного проекта. Чтобы среда Delphi сохраняла свои настройки в DSK-файле, выберите в меню команду **Tools | Environment Options...** и в диалоговом окне **Environment Options** на вкладке **Preferences** в группе **Autosave options** отметьте пункт **Project Desktop**.
- Файл с расширением CFG (сокр. от англ. Configuration), где хранятся настройки для консольного варианта компилятора.
- Файл с расширением DCI (сокр. от англ. Delphi CodeInsight), где среда Delphi хранит сделанные вами настройки для программного "суфлера" (CodeInsight).
- Файл с расширением DCT (сокр. от англ. Delphi Component Templates), где хранятся ваши домашние заготовки компонентов.

- Файл с расширением DMT (сокр. от англ. Delphi Menu Templates), где хранятся ваши домашние заготовки меню.
- Файл с расширением DRO, где хранятся настройки и ваши добавки к хранилищу компонентов.
- Файл с расширением TODO — записная книжка для хранения заданий на программирование и коротких примечаний.
- Файл с расширением DDP (сокр. от англ. Delphi Diagram Portfolio), где хранятся графические схемы, наглядно поясняющие взаимосвязи между компонентами.
- Файл ресурсов с расширением RES (сокр. от RESource). В нем, например, хранится значок приложения, который отображается на Панели Задач. О том, как задать значок приложения, мы расскажем при обсуждении вопросов управления проектом.

В проект могут входить также логически автономные элементы: точечные рисунки (BMP-файлы), значки (ICO-файлы), файлы справки (HLP-файлы) и т.п., однако ими управляет сам программист.

Теперь можно уточнить рисунок, отражающий состав проекта (рисунок 7.4):

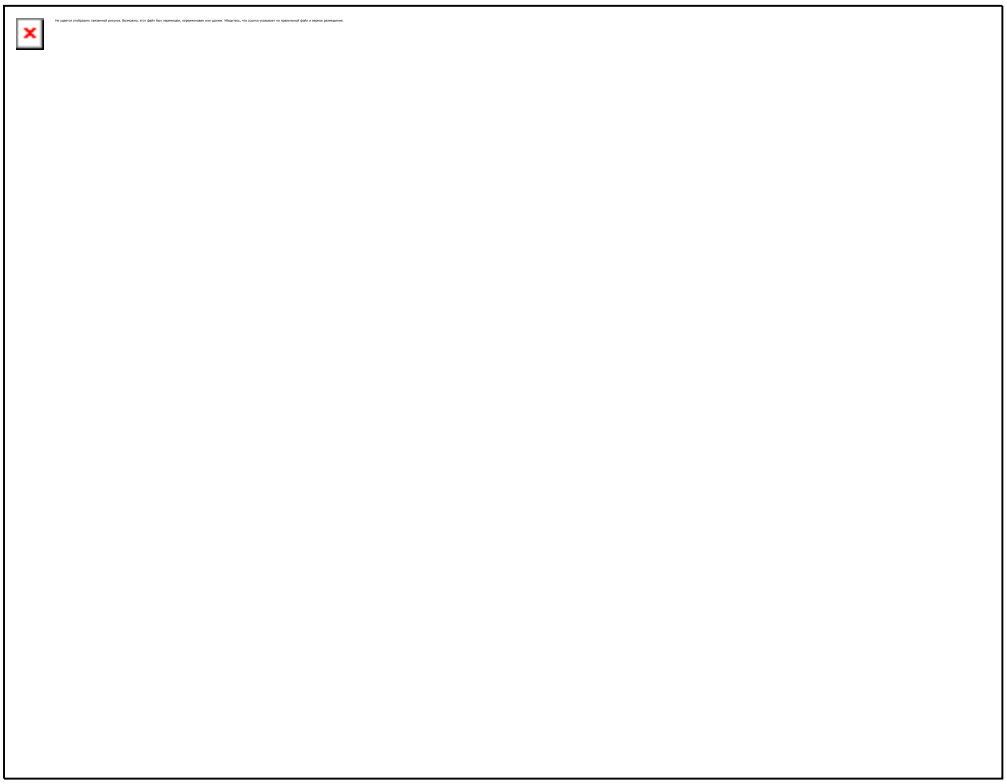


Рисунок 7.4.Состав проекта в среде Delphi

Итак, состав проекта понятен. Нужно теперь выяснить, как им управлять — создавать и сохранять проект, добавлять и удалять модули, устанавливать параметры компиляции, собирать и запускать приложение. Этим сейчас и займемся.

7.2. Управление проектом

7.2.1. Создание, сохранение и открытие проекта

При запуске среды Delphi автоматически создается новый проект. Это сделано для вашего удобства. Если вам потребуется создать новый проект, не перегружая среду Delphi, просто выполните команду меню **File | New | Application**. В результате старый проект будет закрыт, а вместо него создан новый. В новый проект среда Delphi всегда помещает чистую форму.

В процессе разработки приложения вы добавляете на форму компоненты, пишите обработчики событий, добавляете в проект дочерние формы, в общем, проектируете приложение. Когда что-то уже сделано, имеет смысл сохранить проект. Для этого выполните команду главного меню **File | Save All**. Среда запросит сначала имя для программного модуля с формой, а затем имя для проекта (кстати, вы уже сохраняли файл в первой главе). Если файл с введенным именем уже есть на диске, среда Delphi сообщит вам об этом и запросит подтверждение на перезапись существующего файла или запись под другим именем.

Если вдруг потребуется заменить имя проекта другим именем, воспользуйтесь командой меню **File | Save Project As...** . Если же нужно заменить имя модуля, воспользуйтесь командой **File | Save As...** . Операции эти элементарны и не требуют дальнейших пояснений.

Для открытия в среде Delphi ранее сохраненного на диске проекта достаточно выполнить команду главного меню **File | Open...** . На экране появится окно диалога (рисунок 7.5), где вы должны указать или выбрать из представленного списка каталог и имя загружаемого проекта.

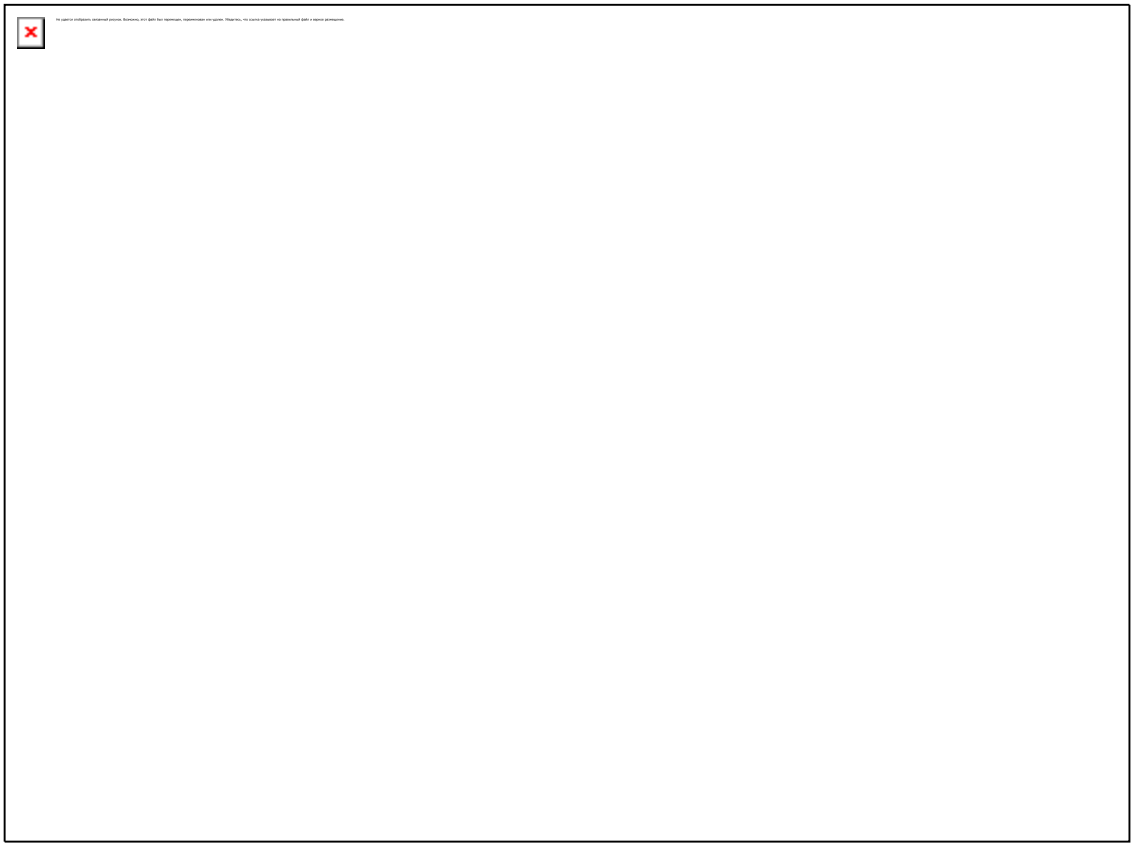


Рисунок 7.5. Окно выбора проекта

С открытым проектом можно продолжить работу: исправить, компилировать, выполнить, и не забыть сохранить.

7.2.2. Окно управления проектом

При создании более или менее сложного приложения программист всегда должен знать, на какой стадии разработки он находится, иметь представление о составе проекта, уметь быстро активизировать нужный файл, добавить в проект какой-либо новый файл или удалить ненужный, установить параметры компиляции, и т.д. Для этого в среде Delphi имеется *окно управления проектом* — окно **Project Manager**. Фактически это визуальный инструмент для редактирования главного файла проекта. Окно управления проектом вызывается из главного

меню командой **View | Project Manager**. После выбора этой команды на экране появится окно, в котором проект представлен в виде дерева (рисунок 7.6).

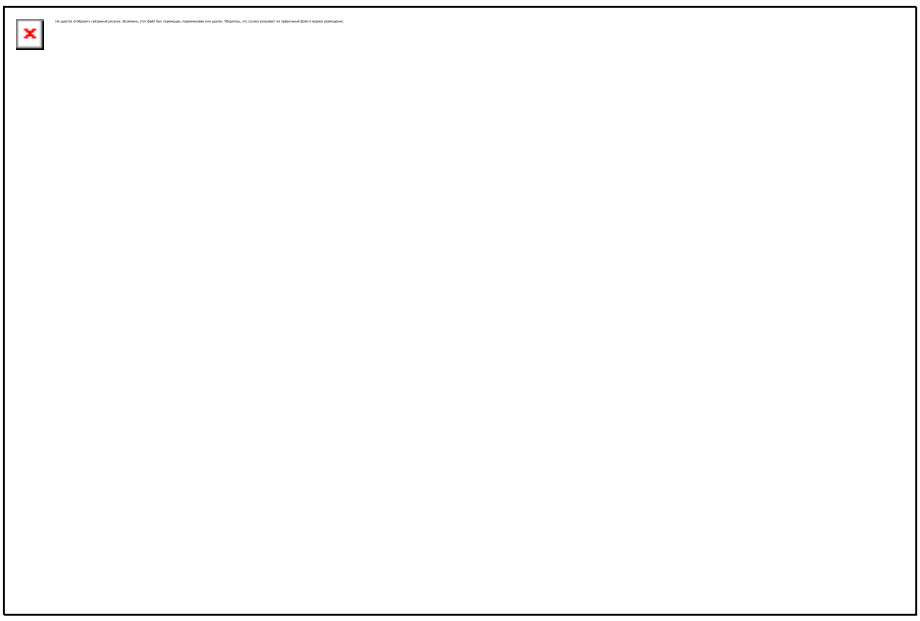


Рисунок 7.6. Окно управления проектом

Выделенный жирным шрифтом элемент Project1 — это наш проект. Его имя совпадает с именем выполняемого файла, который получается в результате компиляции и сборки всех модулей проекта. Список модулей отображается в виде подчиненных элементов. Если элемент является формой, то он в свою очередь сам состоит из двух подчиненных элементов: программного модуля формы (PAS-файл) и описания формы (DFM-файл).

Управление проектом выполняется с помощью контекстного меню, которое вызывается щелчком правой кнопки мыши по элементу Project1 (рисунок 7.7).

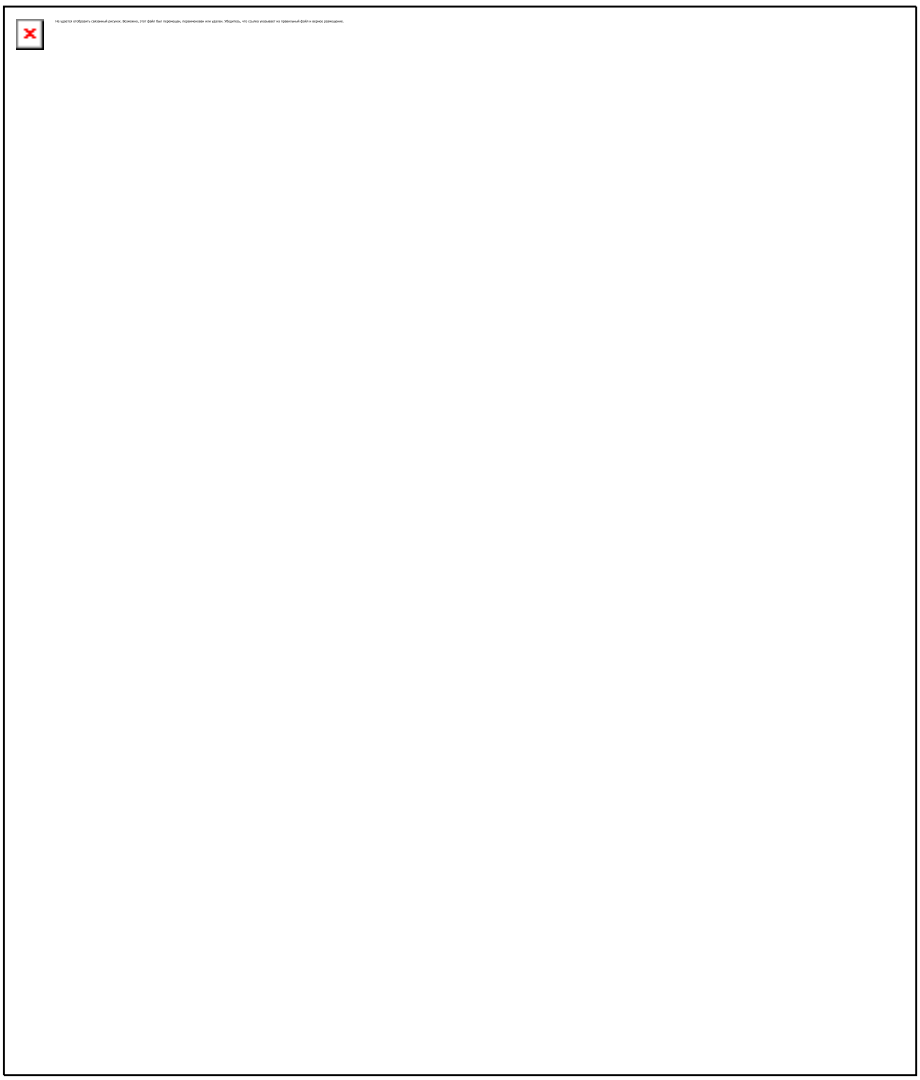


Рисунок 7.7. Контекстное меню проекта

Назначение команд контекстного меню кратко описано в следующей таблице:

Команда	Описание
Add…	Добавляет существующий файл (модуль) в проект.
Remove File…	Удаляет файл (модуль) из проекта.
Save	Сохраняет проект на диск.
Options…	Вызывает диалоговое окно настройки проекта (**Project Options**).
Activate	Делает проект активным (при работе с группой проектов, см. параграф 7.2.3).
Close	Закрывает проект.
Remove Project	Удаляет проект из группы (см. параграф 7.2.3).
Build Sooner	Перемещает проект вверх по списку, определяющему очередность сборки проектов. Используется при

	работе с группой проектов (см. параграф 7.2.3).
Build Later	Перемещает проект вниз по списку, определяющему очередность сборки проектов. Используется при работе с группой проектов (см. параграф 7.2.3).
Compile All From Here	Компилирует измененные проекты по порядку, начиная с выделенного проекта. Используется при работе с группой проектов (см. параграф 7.2.3).
Build All From Here	Компилирует все проекты по порядку, начиная с выделенного проекта. Используется при работе с группой проектов (см. параграф 7.2.3).

Управление отдельным модулем выполняется с помощью контекстного меню, которое вызывается щелчком правой кнопки мыши по соответствующему элементу, например Unit1 (рисунок 7.8).

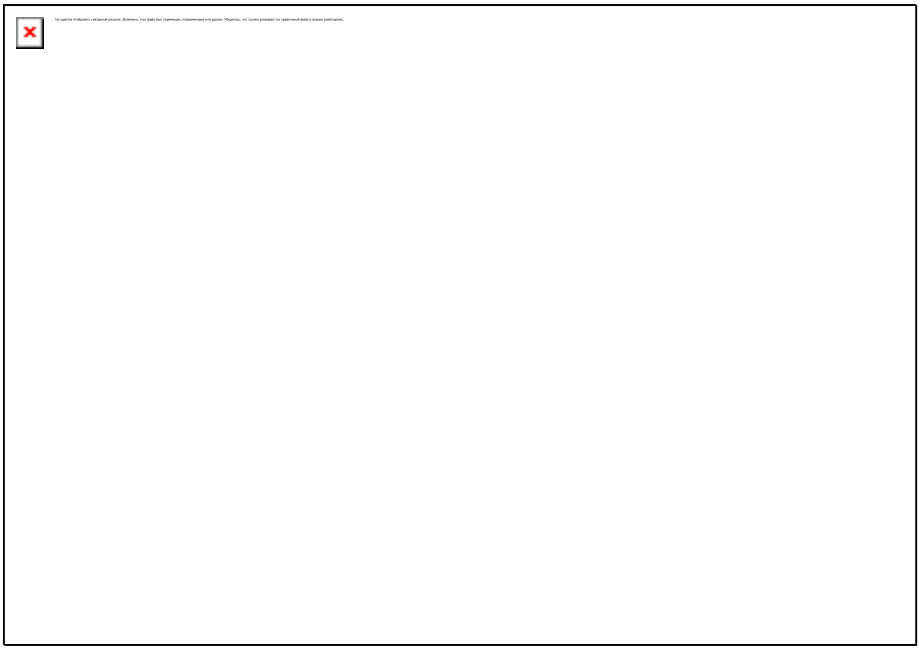


Рисунок 7.8. Контекстное меню модуля в окне управления проектом

Назначение основных команд контекстного меню кратко описано в следующей таблице:

Команда	Описание
Open	Открывает модуль. Если модуль содержит форму, то на экране появляется ее графическое представление. Иначе, на экране появляется редактор кода с исходным текстом программного модуля.
Remove From Project	Удаляет модуль из проекта.
Save	Сохраняет модуль на диск.
Save As…	Сохраняет модуль с новым именем.

Теперь вы всегда сможете узнать, из каких файлов состоит тот или иной проект, а управление им не составит для вас никакого труда.

7.2.3. Группы проектов

На практике несколько проектов могут быть логически связаны между собой, например проект динамически подключаемой библиотеки связан с проектом приложения, в котором используется эта библиотека. Среда Delphi позволяет объединить такие проекты в группу. Именно для этого в окне управления проектом имеется корневой элемент ProjectGroup1, подчиненными элементами которого и являются логически связанные проекты. Порядок элементов определяет очередность сборки проектов. Изменить порядок можно с помощью команд **Build Sooner** и **Build Later**, которые находятся в контекстном меню, вызываемом щелчком правой кнопки мыши по соответствующему проекту (рисунок 7.7).

На данный момент в группе существует только один проект — Project1. Для добавления других проектов в группу воспользуйтесь контекстным меню, которое вызывается щелчком правой кнопки мыши по элементу ProjectGroup1 (рисунок 7.9).

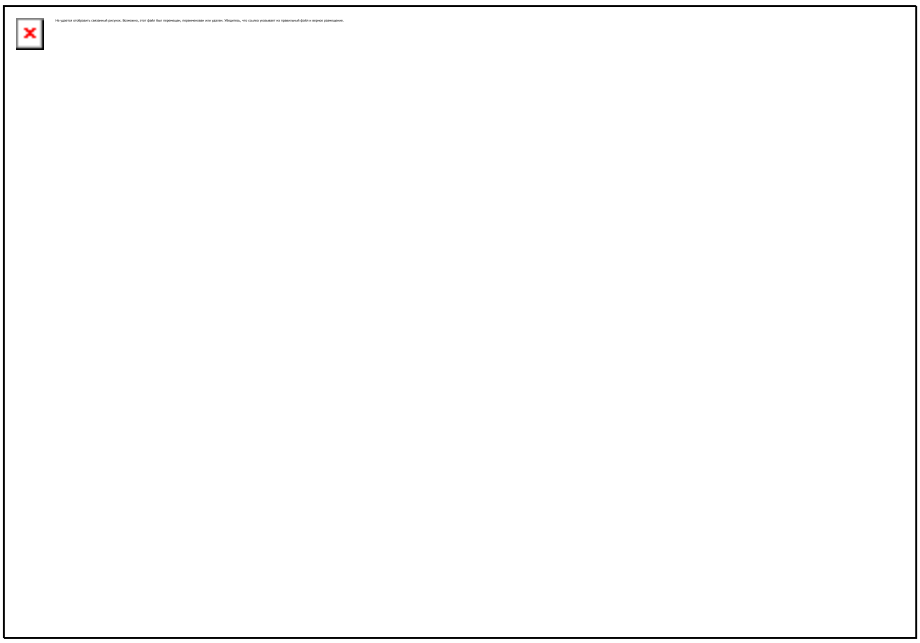


Рисунок 7.9. Контекстное меню группы проектов

Назначение команд контекстного меню кратко описано в следующей таблице:

Команда	Описание
Add New Project…	Создает новый проект и добавляет его в группу.
Add Existing Project…	Добавляет существующий проект в группу.
Save Project Group	Сохраняет файл, описывающий группу проектов.
Save Project Group As…	Сохраняет описание группы проектов в файле с другим именем.
View Project Group source	Показывает текстовый файл, описывающий группу проектов.

Когда в группу объединены несколько проектов, среда Delphi создает специальный текстовый файл с описанием этой группы. Файл имеет расширение BPG (от англ. Borland

Project Group), а его имя запрашивается у пользователя. Для групп, состоящих из одного единственного проекта BPG-файл не создается.

7.2.4. Настройка параметров проекта

Проект имеет много различных параметров, с помощью которых вы управляете процессом компиляции и сборки приложения. Установить параметры проекта можно в окне **Project Options**. Для этого выберите в главном меню команду **Project | Options...** или в окне управления проектом вызовите контекстное меню элемента Project1 и выберите команду **Options...**. На экране появится диалоговое окно; вам останется лишь установить в нем нужные значения параметров.

Диалоговое окно параметров проекта состоит из нескольких вкладок. Параметров очень много, поэтому мы рассмотрим только те, которые используются наиболее часто.

На вкладке **Forms** (рисунок 7.10) можно задать главную форму приложения (**Main form**) и в списке **Auto-create forms** указать формы, которые будут создаваться одновременно с главной формой.

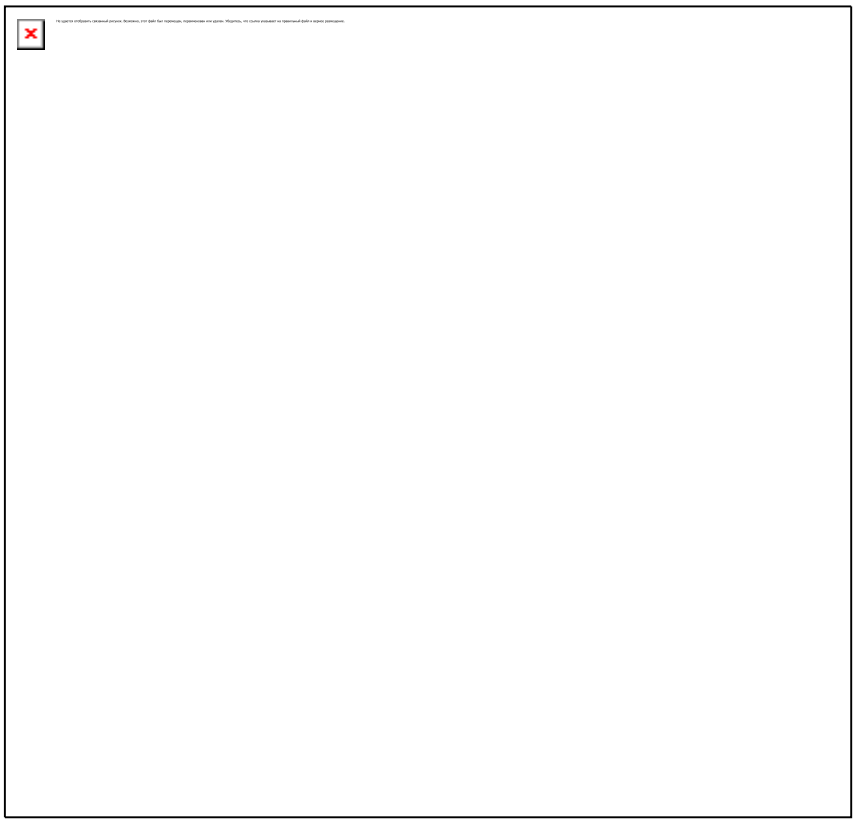


Рисунок 7.10. Окно параметров проекта. Вкладка Forms

На вкладке **Application** (рисунок 7.11) можно задать название (**Title**) вашей программы. В среде Delphi дополнительно можно задать файл справки (**Help file**) и значок (**Icon**).

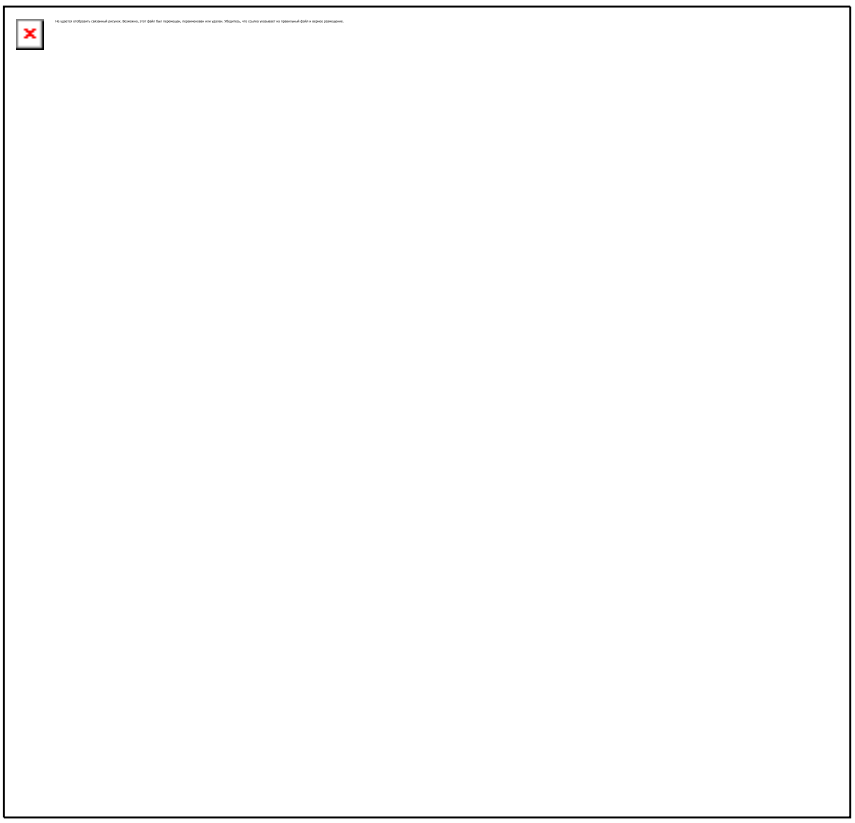


Рисунок 7.11. Вкладка Application в окне параметров проекта

На вкладке **Compiler** (рисунок 7.12) настраиваются параметры компилятора. Наиболее интересными из них являются переключатели **Optimization** (включает оптимизацию генерируемого кода) и **Use Debug DCUs** (позволяет отлаживать исходный код системных библиотек). Оба этих переключателя полезны при отладке программы: первый следует выключить, а второй — включить.

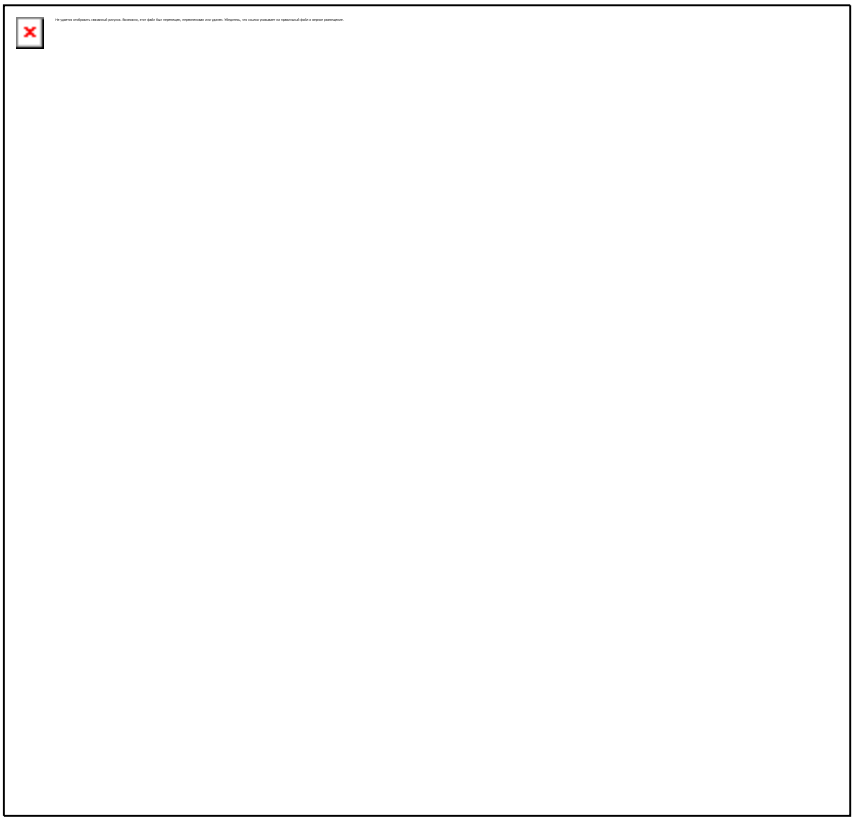


Рисунок 7.12. Вкладка Compiler в окне параметров проекта

На вкладке **Compiler Messages** (рисунок 7.13) настраивается чувствительность компилятора к подозрительному коду. Включив переключатели **Show hints** и **Show warnings**, вы будете получать от компилятора весьма полезные подсказки (hints) и предупреждения (warnings), а не только сообщения об ошибках.

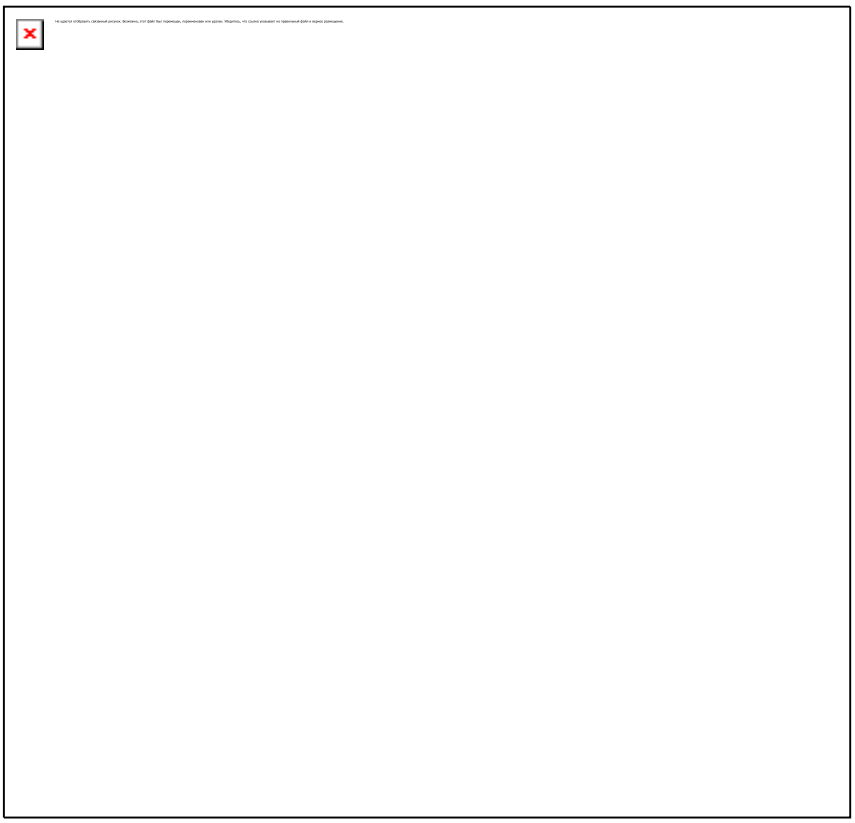


Рисунок 7.13. Вкладка Compiler Messages в окне параметров проекта

На вкладке **Linker** (рисунок 7.14) настраиваются параметры сборки проекта. Обладателям среды Delphi следует обратить внимание на группу **Memory sizes**, особенно на два параметра: **Min stack size** и **Max stack size**. Они задают соответственно минимальный и максимальный размеры стека прикладной программы. Вам может потребоваться увеличить значения этих параметров при написании приложения, активно использующего рекурсивные подпрограммы.

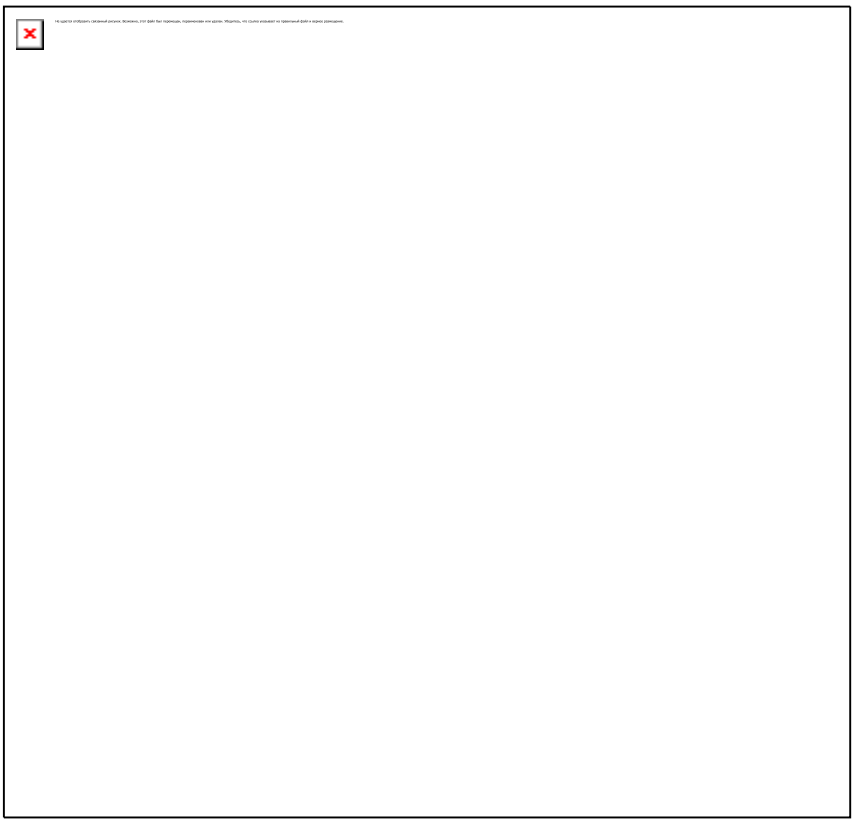


Рисунок 7.14. Вкладка Linker в окне параметров проекта

На вкладке **Directories/Conditionals** (рисунок 7.15) можно задать каталоги для различных файлов. Наиболее важные из них: **Output directory** — каталог, в который помещается выполняемый файл; **Unit output directory** — каталог, в который помещаются промежуточные объектные модули (DCU-файлы); **Search path** — список каталогов, в которых осуществляется поиск программных модулей.

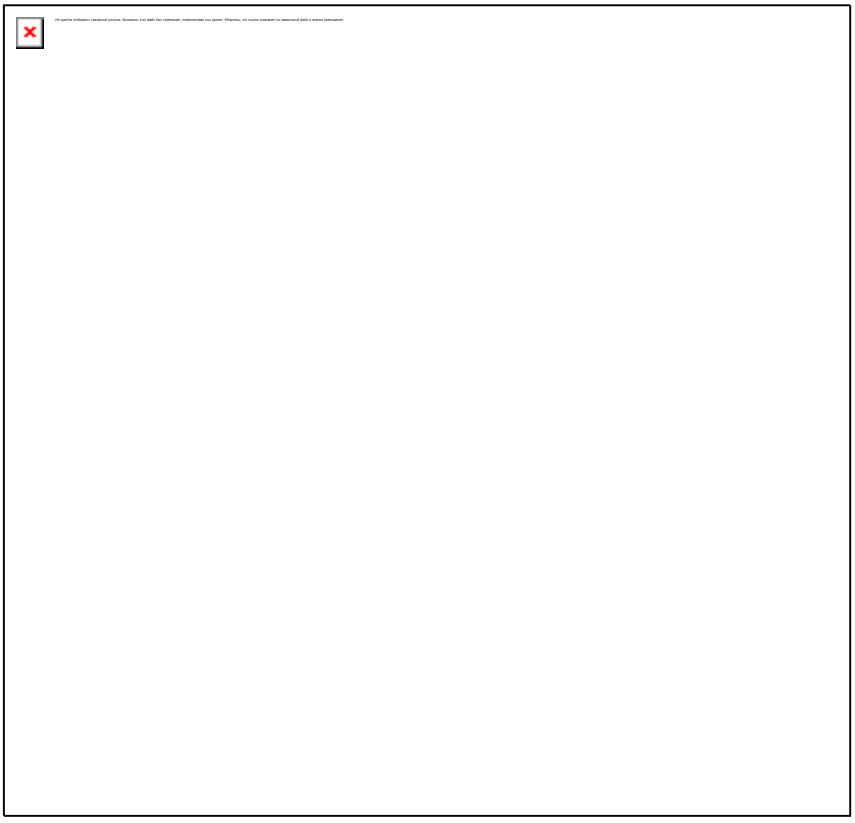


Рисунок 7.15. Вкладка Directories/Conditionals в окне параметров проекта

На вкладке **Version Info** (рисунок 7.16) выводится информация о версии приложения. Для того чтобы эта информация помещалась в выполняемый файл, нужно включить переключатель **Include version information in project**. Номер версии задается в виде четырех чисел: **Major version** — старший номер версии программы (его обычно увеличивают при внесении в программу концептуально новых возможностей); **Minor version** — младший номер версии программы (его обычно увеличивают при незначительном расширении функциональных возможностей программы); **Release** — номер выпуска программы, которая отлажна и пригодна к использованию заказчиком; **Build** — порядковый номер сборки проекта (он автоматически увеличивается на единицу при компиляции проекта, если включен переключатель **Auto-increment build number**). Все эти параметры несут лишь информативный характер и не влияют на работу самой программы. Однако, информация о версии может использоваться специальными программами установки пользовательских программ для контроля за тем, чтобы более новые версии библиотек не заменялись более старыми.

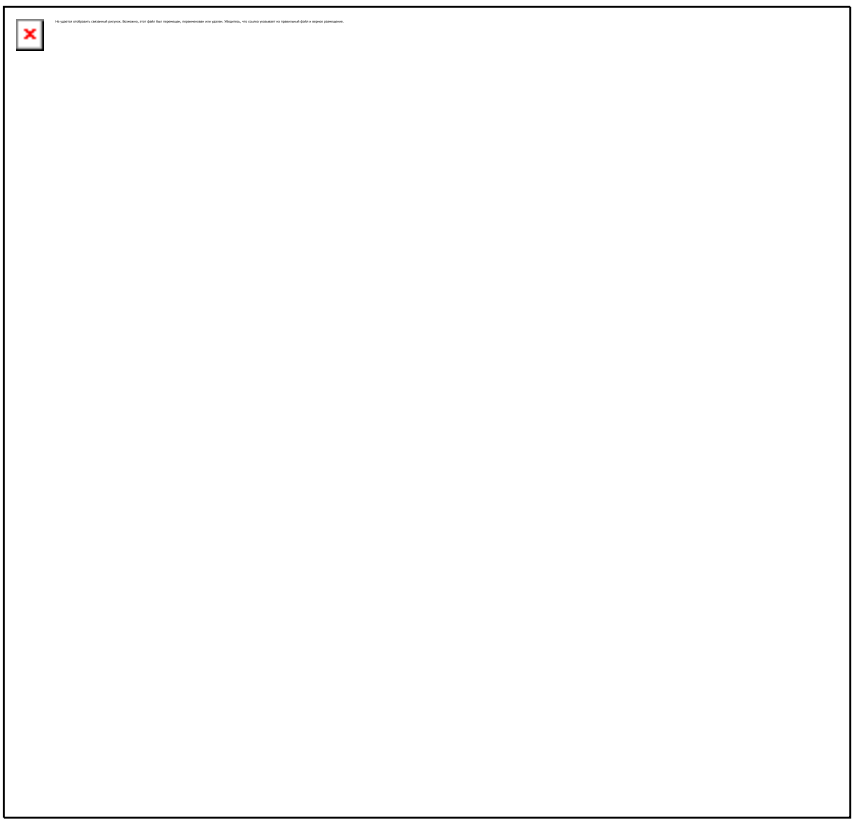


Рисунок 7.16. Вкладка Version Info в окне параметров проекта

На вкладке **Packages** (рисунок 7.17) вы можете управлять списком пакетов, используемых в вашем проекте. *Пакеты* — это внешние библиотеки компонентов, они рассмотрены в главе 13. Обратите внимание на переключатель **Build with runtime packages**, который позволяет существенно уменьшить размер выполняемого файла за счет использования внешних библиотек компонентов вместо встраивания их кода непосредственно в выполняемый файл. Этот режим выгоден при создании нескольких программ, построенных на базе большого количества общих компонентов.

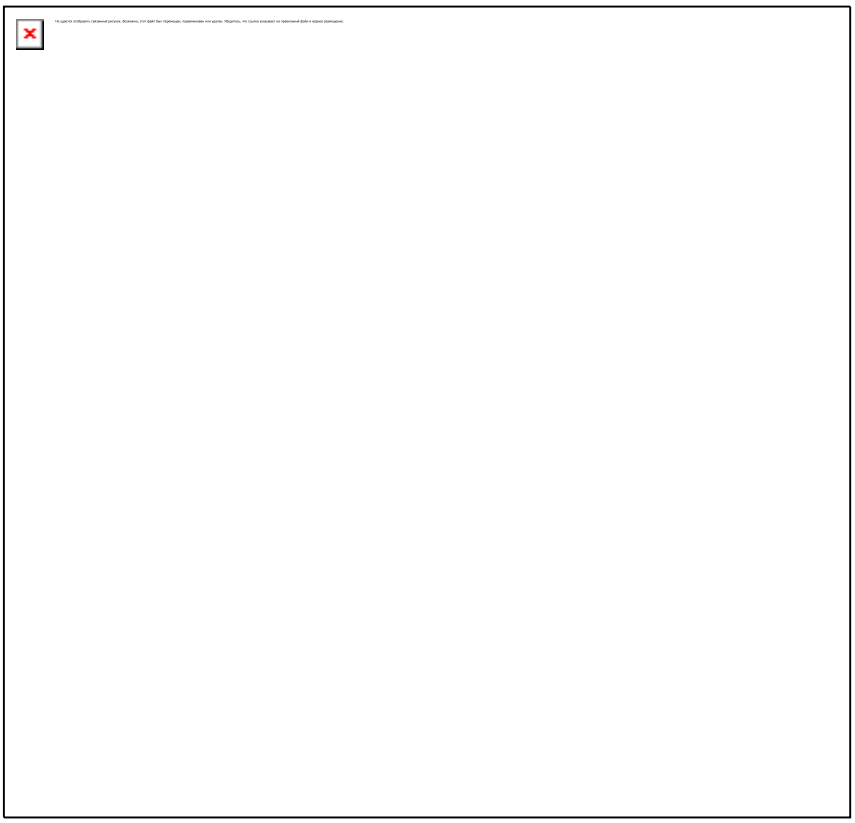


Рисунок 7.17. Вкладка Packages в окне параметров проекта

Когда все параметры проекта установлены, можно приступать к его компиляции.

7.2.5. Компиляция и сборка проекта

Компиляция и сборка проекта могут выполняться на любой стадии разработки проекта. Под *компиляцией* понимается получение объектных модулей (DCU-файлов) из исходных текстов программных модулей (PAS-файлов). Под *сборкой* понимается получение выполняемого файла из объектных модулей. В среде Delphi компиляция и сборка проекта совмещены.

Для выполнения компиляции достаточно выполнить команду меню **Project | Compile <Имя проекта>** или нажать комбинацию клавиш Ctrl+F9. При этом компилируются все исходные модули, содержимое которых изменялось после последней компиляции. В результате для каждого программного модуля создается файл с расширением DCU (сокр. от Delphi Compiled Unit). Затем среда Delphi компилирует главный файл проекта и собирает (иногда говорят компонует) из DCU-модулей выполняемый файл, имя которого совпадает с именем проекта. К сведению профессионалов заметим, что смышленый компилятор среды Delphi выбрасывает из выполняемого файла весь неиспользуемый программный код, поэтому не стоит волноваться по поводу лишних объектов и подпрограмм, которые могут присутствовать в подключенных модулях.

Существует особый вид компиляции и сборки — полная принудительная компиляция всех программных модулей проекта, для которых доступны исходные тексты, с последующей сборкой выполняемого файла. При этом не важно, вносились в них изменения после предыдущей компиляции или нет. Полная компиляция проекта выполняется с помощью команды главного меню **Project | Build <Имя проекта>**. В результате тоже создается выполняемый файл, но на это тратиться немного больше времени.

7.2.6. Запуск готового приложения

Когда после многочисленных компиляций вы исправите все ошибки и получите-таки выполняемый файл, можно будет посмотреть на результат вашего самоотверженного труда.

Для этого надо выполнить созданное приложение с помощью команды меню **Run | Run** или клавиши F9. Перед выполнением будет автоматически повторен процесс компиляции (если в проект вносились изменения) и после его успешного завершения приложение запустится на выполнение. В результате вы увидите на экране его главную форму.

Вот собственно и все, что мы хотели поведать вам о проекте. В целом вы представляете, что такое проект, и знаете, как им управлять. Пора заняться составными частями проекта и одновременно основными элементами любого приложения в среде Delphi — формами.

7.3. Форма

7.3.1. Понятие формы

Из первой главы вы уже имеете общее представление о форме, теперь настало время изучить ее более пристально. Фактически форма — это главный компонент приложения, который, как и менее значительные компоненты, имеет свойства. Важнейшие свойства формы: заголовок, высота, ширина, местоположение, цвет фона и др. При создании новой формы среда Delphi сама задает начальные значения свойствам формы, но вы можете изменить их так, как считаете нужным. Это можно сделать во время проектирования формы (в окне свойств) или во время выполнения приложения (с помощью операторов языка Delphi).

Форма имеет очень много свойств, и поначалу в них легко запутаться. Практика показывает, что путаница возникает из-за алфавитного порядка свойств в окне **Object Inspector**: близкие по смыслу свойства оказываются разбросанными по ячейкам списка. Чтобы у вас сложилось представление о возможностях формы, рассмотрим основные свойства формы в порядке их важности. Для этого нам понадобится новое приложение.

Выберите в меню команду **File | New | Application**. Среда Delphi автоматически создаст в новом проекте чистую форму и поместит ее исходный текст в редактор кода (рисунок 7.18).

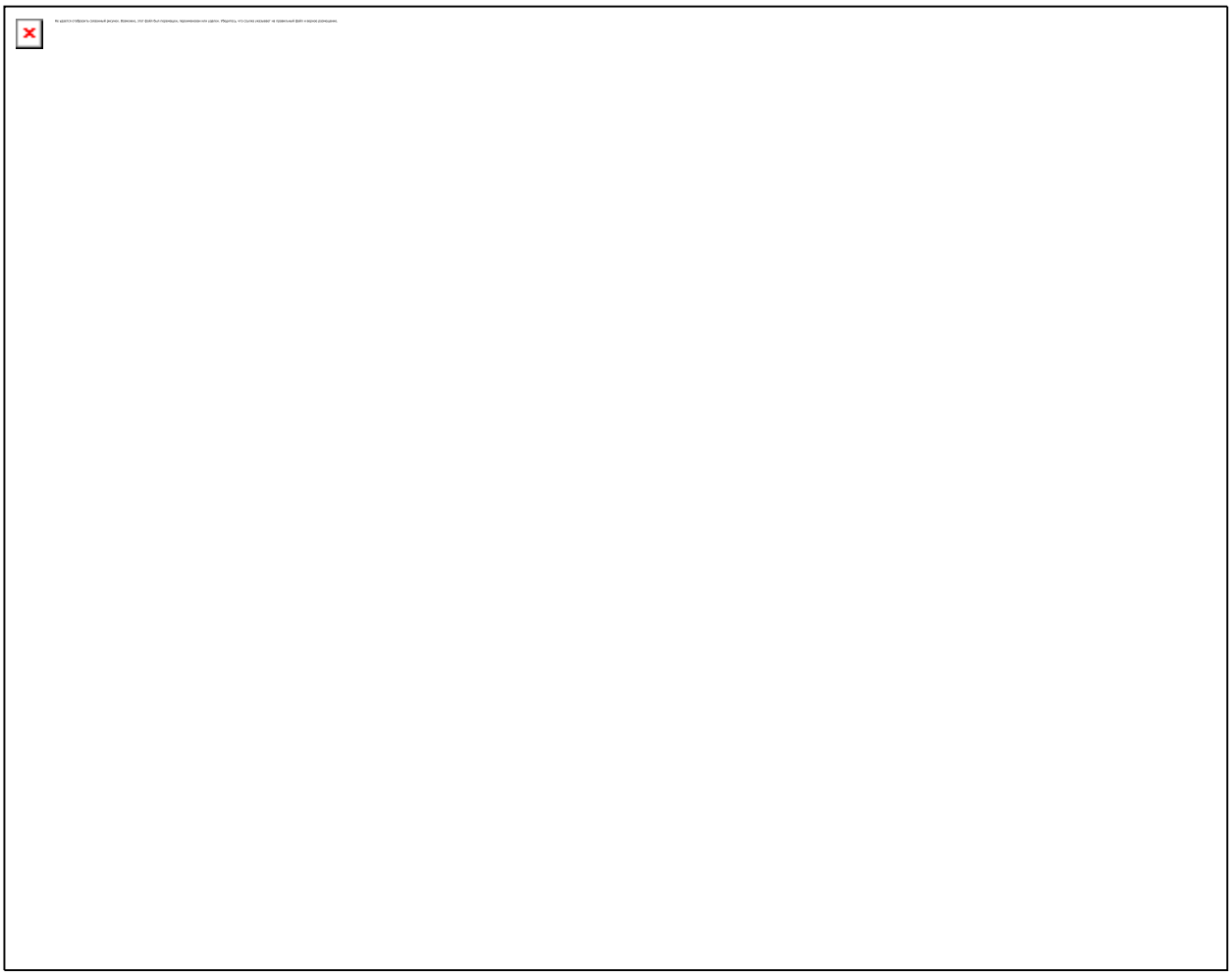


Рисунок 7.18. Форма на экране и ее описание в редакторе кода

Сразу сохраните проект и его форму, чтобы дать им осмысленные имена. Выберите в меню команду **File | Save All** и дайте модулю имя **Main.pas**, а проекту — имя **FormTest.dpr**. Полигон для изучения формы подготовлен, можно заняться ее свойствами.

7.3.2. Имя и заголовок формы

Главное свойство, с которого вы начинаете настройку формы, – это свойство **Name** — имя. Оно содержит идентификатор, используемый для обращения к форме из программы (рисунок 7.19). Первой же форме нового проекта автоматически назначается имя **Form1**. Мы советуем всегда его изменять, чтобы имя формы отражало ее роль в приложении. Например, главную форму приложения можно назвать **MainForm** (если ничего лучше в голову не приходит).

Рисунок 7.19. Программный идентификатор формы

На будущее заметим, что свойство **Name** есть в любом компоненте, и оно редактируется в окне свойств.

Каждая форма приложения должна иметь понятный заголовок, говорящий пользователю о ее назначении. Заголовок задается в свойстве **Caption**. Наша форма — учебная, поэтому мы дадим ей заголовок **Main**, говорящий о том, что это просто главная форма (рисунок 7.20).

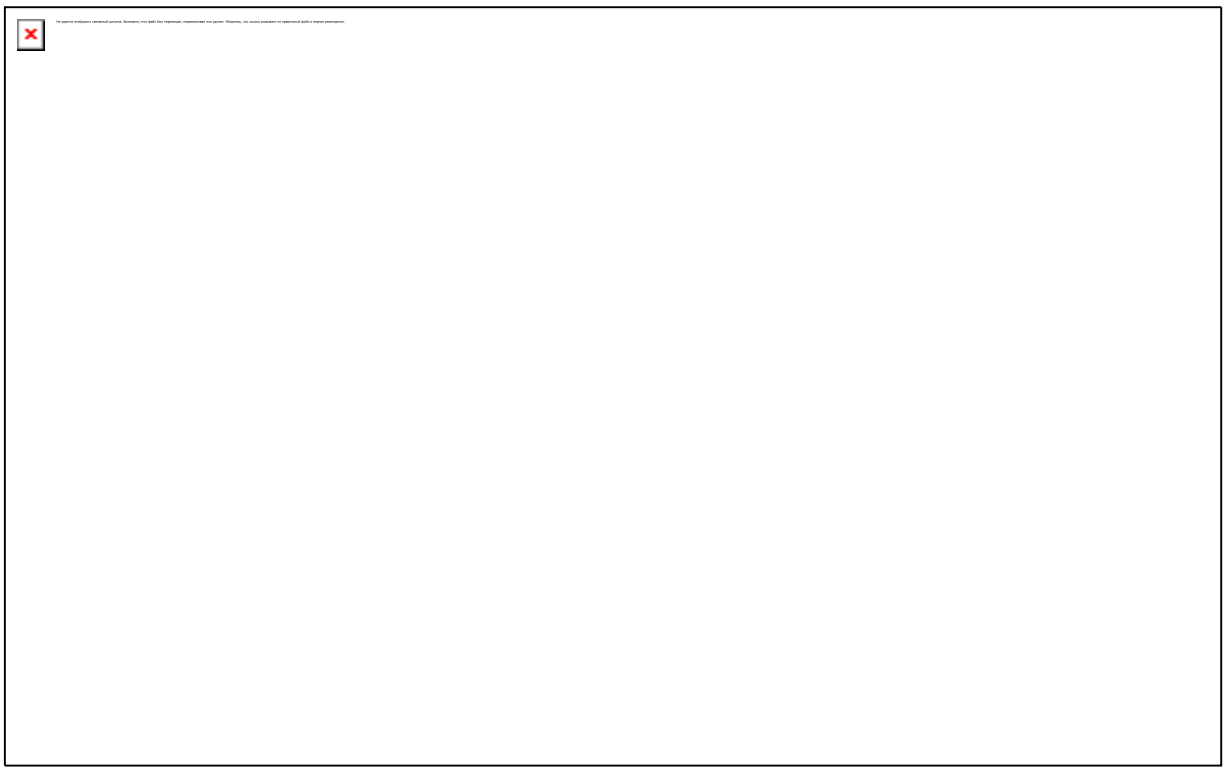


Рисунок 7.20. Заголовок формы

Внимание! Если вам вдруг взбрела в голову идея изменить шрифт, цвет или высоту заголовка, то не ищите для этого подходящих свойств. Все это — параметры графической оболочки операционной системы, и задаются они в настройках этой оболочки.

7.3.3. Стиль формы

Настраивая форму, нужно принимать во внимание, какой пользовательский интерфейс будет иметь ваше приложение: многодокументный интерфейс MDI (от англ. Multiple Document Interface) или обычный одно-документный интерфейс SDI (от англ. Single Document Interface). За это отвечает свойство формы **FormStyle**, которое может принимать следующие значения:

- fsMDIChild – дочернее окно MDI-приложения;
- fsMDIForm – главное окно MDI-приложения;
- fsNormal – обычное окно (значение по умолчанию);
- fsStayOnTop – окно, всегда расположенное поверх других окон на экране.

Многие приложения, с которыми вы работаете, имеют пользовательский интерфейс MDI. Они состоят из основного окна, которое включает одно или несколько внутренних окон. Внутренние окна ограничены областью основного окна и не могут выходить за его границы. Для главной формы, соответствующей основному окну MDI-приложения, значение свойства **FormStyle** должно быть равно fsMDIForm. Для всех второстепенных форм, соответствующих внутренним окнам, значение свойства **FormStyle** равно fsMDIChild. Для окон диалога, выполняющихся в монопольном режиме, свойство **FormStyle** равно значению fsNormal, что дает возможность выносить их за пределы основной формы.

Если программа имеет пользовательский интерфейс SDI, то каждая форма существует в виде отдельного независимого окна. Одно из окон является главным, однако оно не содержит другие окна. В SDI-приложении значение свойства **FormStyle** равно fsNormal и для главной

формы, и для второстепенных форм. В некоторых случаях допускается установка значения fsStayOnTop для того, чтобы форма всегда отображалось поверх других форм.

Очевидно, что наш простой вычислитель идеального веса является SDI-приложением и поэтому свойство **FormStyle** имеет значение по умолчанию — fsNormal.

7.3.4. Размеры и местоположение формы на экране

Теперь определимся с размерами формы и ее местоположением на экране. Установить размеры и положение формы проще всего во время проектирования с помощью мыши. Другой способ — обратиться к окну свойств и задать размеры формы с помощью свойств **Width** и **Height**, а местоположение — с помощью свойств **Left** и **Top** (значения задаются в пикселах). Смысл свойств поясняет рисунок 7.21.

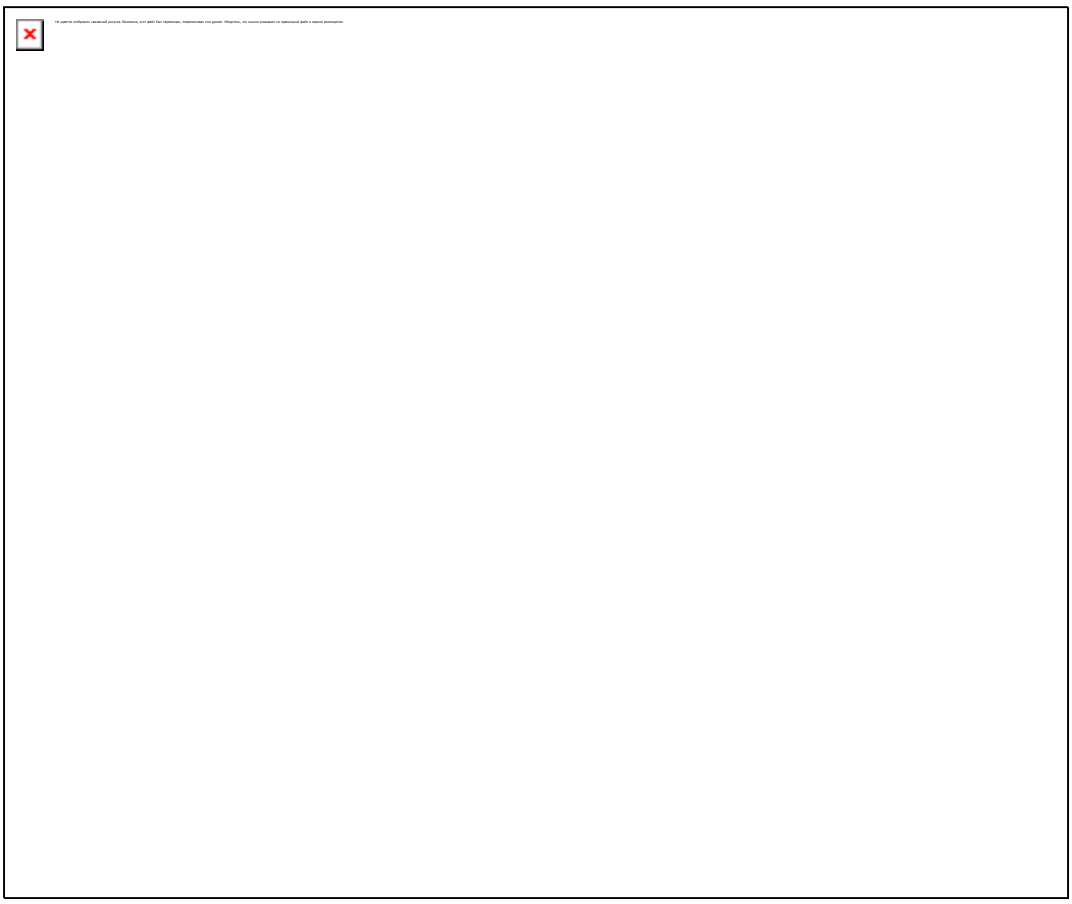


Рисунок 7.21. Размеры и местоположение формы на экране

Кроме того, с помощью свойства **Position** можно организовать автоматическое размещение формы на экране, выбрав одно из следующих возможных значений:

- poDefault – размеры и положение формы подбираются автоматически исходя из размеров экрана.
- poDefaultPosOnly – положение формы подбирается автоматически, а ширина и высота определяются значениями свойств **Width** и **Height** соответственно.
- poDefaultSizeOnly – размеры формы устанавливаются автоматически, а местоположение определяется значениями свойств **Left** и **Top**.
- poDesigned – размеры и положение формы определяются значениями свойств **Left**, **Top**, **Width**, **Height**.

- poDesktopCenter – форма размещается в центре рабочего стола (т.е. экрана, из которого исключена панель задач). Размеры формы определяются значениями свойств **Width** и **Height**.
- poMainFormCenter – форма центрируется относительно главной формы. Размеры формы определяются значениями свойств **Width** и **Height**.
- poOwnerFormCenter – форма центрируется относительно формы-владельца. Размеры формы определяются значениями свойств **Width** и **Height**.
- poScreenCenter – форма размещается в центре экрана. Размеры формы определяются значениями свойств **Width** и **Height**.

Иногда размеры формы рассчитываются исходя из размеров ее внутренней *рабочей области* (client area), на которой размещаются компоненты. Как известно, в рабочую область не входят рамка и заголовок. Размеры рабочей области хранятся в свойствах **ClientWidth** и **ClientHeight**. При их установке значения свойств **Width** и **Height** автоматически пересчитываются (и наоборот).

Бывает, что при выборе размеров формы учитываются размеры экрана. Поскольку современные видео-адаптеры поддерживают множество режимов с различными разрешениями, встает вопрос: как обеспечить одинаковую пропорцию между формой и экраном независимо от разрешающей способности дисплея. На этот случай в форме предусмотрено свойство **Scaled**. Если оно установлено в значение True, то форма будет автоматически масштабироваться в зависимости от разрешающей способности дисплея.

При перемещении по экрану, форма может слегка прилипать к краям экрана, если края формы находятся в непосредственной близости от них. Это происходит в том случае, если свойство **ScreenSnap** содержит значение True. Расстояние формы до краев экрана, при котором форма прилипает, задается в свойстве **SnapBuffer** и измеряется в пикселях.

Работая с приложением, пользователь может свернуть форму или развернуть ее на всю рабочую область экрана с помощью соответствующих кнопок рамки. Состояние формы (свернута или развернута) определяется свойством **WindowState**, которое принимает следующие значения:

- wsNormal – форма находится в нормальном состоянии (ни свернута, ни развернута на весь экран);
- wsMinimized – форма свернута;
- wsMaximized – форма развернута на весь экран.

Если при проектировании вы измените значение свойства **WindowState** на wsMinimized или wsMaximized, то получите форму, которая при первом появлении будет автоматически либо свернута в панель задач, либо развернута на весь экран.

На компьютере с двумя и более мониторами существует возможность выбрать для формы монитор, на котором она отображается. Для этого следует установить свойство **DefaultMonitor** в одно из следующих значений:

- dmDesktop – форма отображается на текущем мониторе; никаких попыток разместить форму на каком-то конкретном мониторе не делается;
- dmPrimary – форма отображается на первом мониторе в списке **Monitors** объекта **Screen** (см. параграф 7.7.2);
- dmMainForm – форма отображается на том мониторе, на котором находится главная форма;

- dmActiveForm – форма отображается на том мониторе, на котором находится активная в данный момент форма.

Свойство **DefaultMonitor** учитывается лишь в том случае, если в программе существует главная форма.

7.3.5. Цвет рабочей области формы

С размерами формы все ясно и теперь желающие могут изменить стандартный цвет ее фона с помощью свойства **Color**. Для этого следует обратиться к окну свойств. Перейдите к свойству **Color**, откройте выпадающий список, и выберите любой цвет из набора базовых цветов. Базовые цвета представлены в списке именованными константами. Вы можете также выбрать цвет из всей цветовой палитры, выполнив двойной щелчок мыши в поле значения свойства. На экране появится стандартное диалоговое окно выбора цвета (рисунок 7.22).

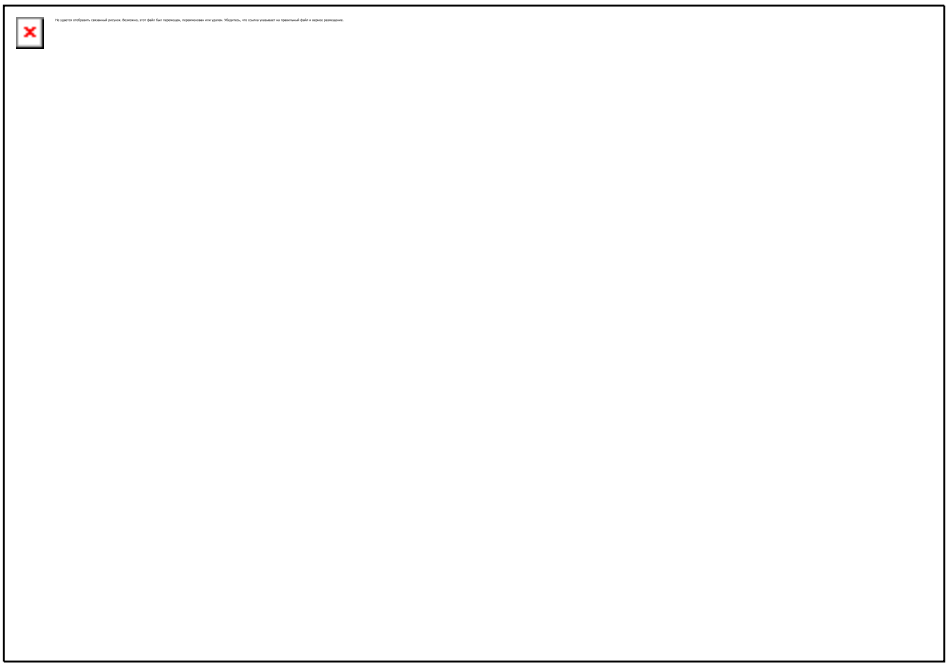


Рисунок 7.22. Стандартоне диалоговое окно выбора цвета

Когда вы установите цвет в окне свойств, изменение немедленно отразится на форме. Можно работать с самыми разными цветами, но хорошим тоном считается использовать стандартную цветовую гамму. Поэтому лучшее значение для свойства **Color** — clBtnFace (цвет такой, как у кнопок).

7.3.6. Рамка формы

Во внешнем виде формы очень важную роль играет рамка и расположенные на ней кнопки "Свернуть", "Развернуть", "Закрыть" (рисунок 7.23). Стиль рамки задается с помощью свойства **BorderStyle**, которое может принимать следующие значения:

- bsNone — у окна вообще нет ни рамки, ни заголовка;
- bsDialog — неизменяемая в размерах рамка, свойственная диалоговым окнам;
- bsSingle — неизменяемая в размерах рамка для обычного окна;
- bsSizeable — изменяемая в размерах рамка для обычного окна;
- bsToolWindow — аналогично значению bsSingle, но окно имеет слегка уменьшенный заголовок, что свидетельствует о его служебном назначении;
- bsSizeToolWin — аналогично значению bsSizeable, но окно имеет слегка уменьшенный заголовок, что свидетельствует о его служебном назначении.

Обычно свойство **BorderStyle** имеет значение bsSizeable. В этом случае форма имеет стандартную изменяемую в размерах рамку (как при проектировании), заголовок, меню управления, кнопки "Свернуть", "Развернуть", "Закрыть" и, иногда, "Справка". Для указания того, какие именно из этих элементов отображать, используется свойство **BorderIcons**. Список его возможных значений следующий:

- biSystemMenu — рамка формы содержит меню управления, которое вызывается щелчком правой кнопки мыши по заголовку формы;
- biMinimize – рамка формы имеет кнопку "Свернуть";
- biMaximize – рамка формы имеет кнопку "Развернуть";
- biHelp – рамка формы имеет кнопку "Справка". При нажатии кнопки "Справка", курсор мыши превращается в стрелку со знаком вопроса. Выбирая таким курсором нужный элемент формы, пользователь получает по нему справку во всплывающем окне.

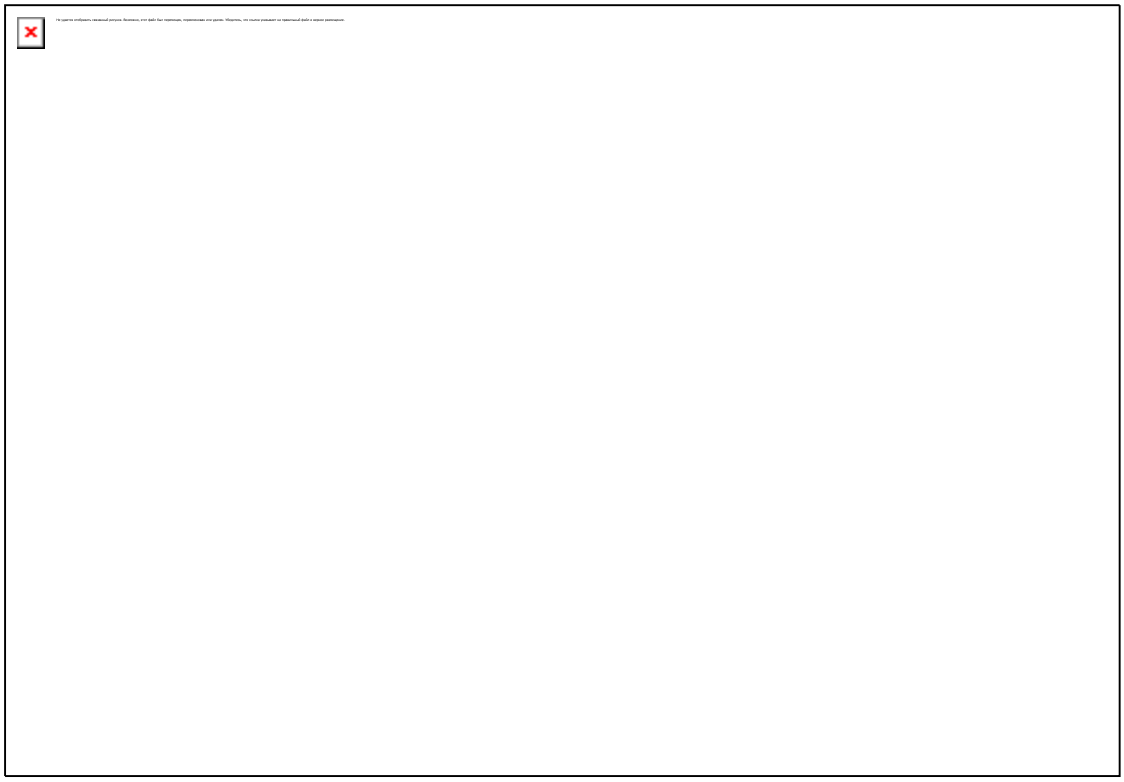


Рисунок 7.23. Рамка формы и ее контекстное меню

Команды меню управления не нуждаются в комментариях.

7.3.7. Значок формы

Если вы разрабатываете коммерческое приложение, а не тестовый пример, следует позаботиться о том, чтобы форма имела в своем левом верхнем углу выразительный значок. Для разработки значков существует множество средств, на которых мы не будем останавливаться. Когда значок готов и сохранен в файле, его нужно просто установить в качестве значения свойства **Icon** (рисунок 7.24).

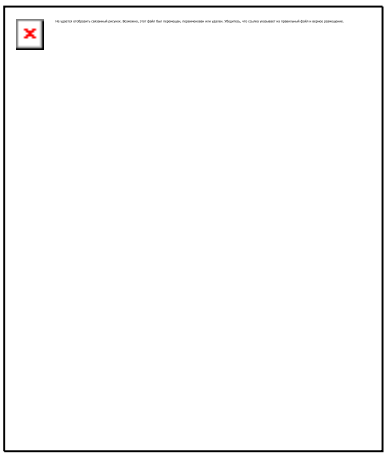


Рисунок 7.24. Установка значка формы

Для этого откройте окно **Picture Editor** нажатием кнопки с многоточием (рисунок 7.25). Нажмите кнопку **Load...** и выберите какой-нибудь файл из стандартной коллекции значков (как правило, каталог C:\Program Files\Common Files\Borland Shared\Images). После этого закройте диалоговое окно с помощью кнопки **OK**.

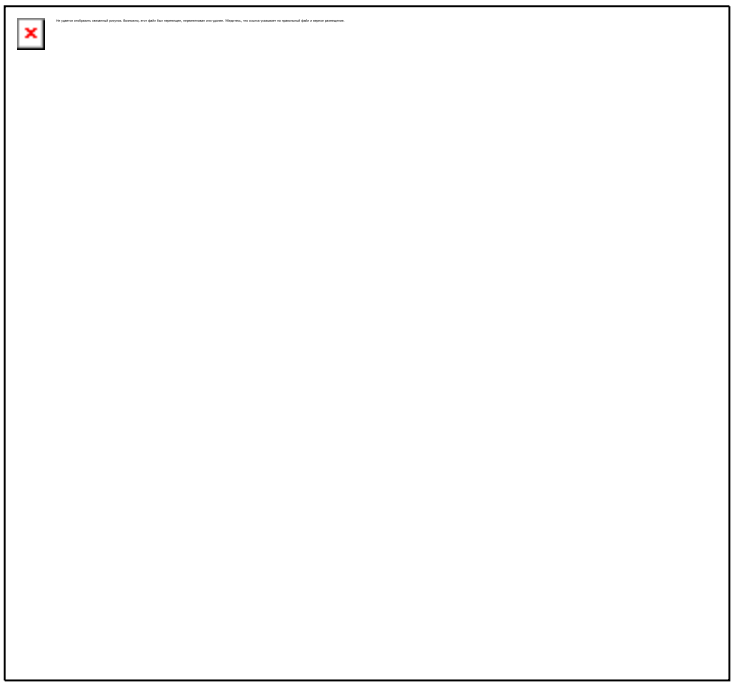


Рисунок 7.25. Окно выбора значка для формы

Среда Delphi сразу же подставит выбранный значок в левый верхний угол формы (рисунок 7.26).

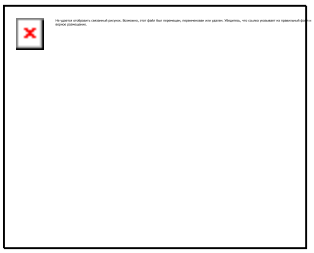


Рисунок 7.26. Новый значок формы

7.3.8. Невидимая форма

Сценарий решения задачи может потребовать, чтобы в некоторый момент форма стала невидимой, т.е. исчезла с экрана. За “видимость” формы отвечает булевское свойство **Visible**. Установка ему значения False скроет форму, а установка значения True покажет ее.

7.3.9. Прозрачная форма

Некоторые части формы можно сделать прозрачными (рисунок 7.27). Причем щелчок мыши по прозрачной области формы будет приводить к активизации окон (форм), находящихся за формой. Это достигается установкой свойства **TransparentColor** в значение True и выбором цвета прозрачности в свойстве **TransparentColorValue**. Все пиксели формы с цветом **TransparentColorValue** будут прозрачными.

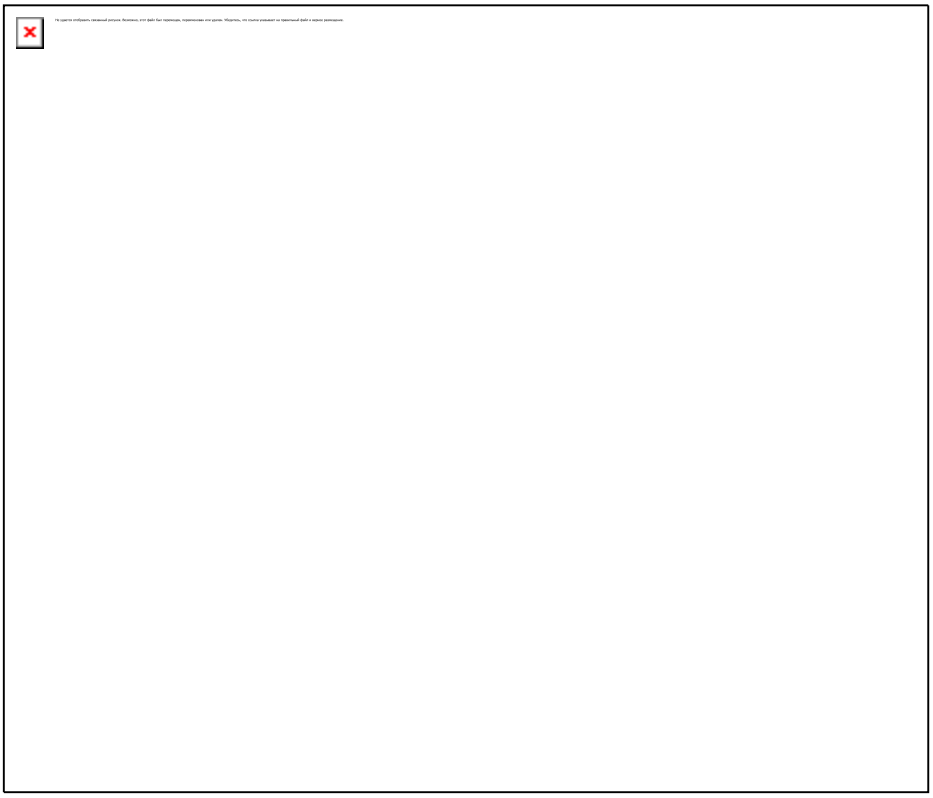


Рисунок 7.27. Прозрачная форма

Например, рисунок 7.27 был получен следующим образом. Мы положили на форму компонент **Shape**, превратили его в эллипс (**Shape** = stEllipse), растянули до размеров формы (**Align** = alClient), в форме установили свойство **TransparentColor** в значение True и уравняли в форме значение свойства **TransparentColorValue** со свойством **Brush.Color** компонента **Shape**. После сборки и запуска программы получили «дырявую» форму.

7.3.10. Полупрозрачная форма

Форма может быть полупрозрачной (рисунок 7.28). За полупрозрачность формы отвечают свойства **AlphaBlend** и **AlphaBlendValue**. Первое свойство включает и выключает эффект полупрозрачности, а второе определяет силу прозрачности.

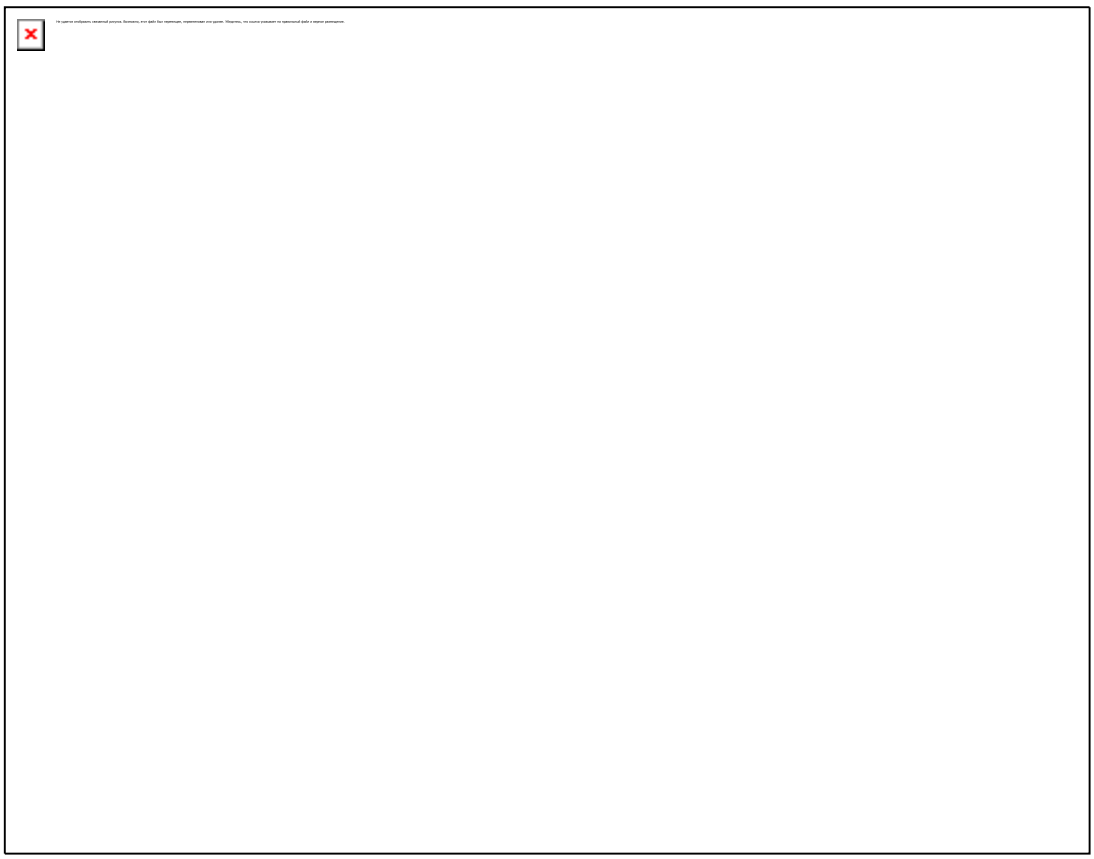


Рисунок 7.28. Полупрозрачная форма

Например, рисунок 7.28 был получен следующим образом. Мы положили на форму компонент **Image**, загрузили в него картинку (свойство **Picture**), затем в форме установили свойство **AlphaBlend** в значение True и свойство **AlphaBlendValue** — в значение 150. После сборки и запуска программы получили эффект полупрозрачности.

7.3.11. Недоступная форма

Иногда бывает нужно просто запретить доступ к форме, не убирая ее с экрана. Для этого служит другое булевское свойство **Enabled**. Обычно оно равно значению True, но стоит ему присвоить противоположное значение, и после запуска приложения вы не сможете сделать форму активной.

Как вы понимаете, все описанные выше свойства доступны не только в окне свойств, но и в редакторе кода, т.е. в тексте программы. При работе с формой на уровне исходного кода вы также получаете доступ к некоторым дополнительным свойствам, которые не видны в окне свойств. Они будут рассматриваться по мере надобности.

7.3.12. События формы

Итак, со свойствами мы разобрались и пора сказать пару слов о возникающих при работе с формой событиях. С вашего позволения мы опустим те события формы, которые происходят во всех видимых на экране компонентах (мы о них расскажем позже, когда будем рассматривать компоненты). Перечислим лишь характерные события форм:

OnCreate — происходит сразу после создания формы. Обработчик этого события может установить начальные значения для свойств формы и ее компонентов, запросить у операционной системы необходимые ресурсы, создать служебные объекты, а также выполнить другие действия прежде, чем пользователь начнет работу с формой. Парным для события OnCreate является событие OnDestroy.

OnDestroy — происходит непосредственно перед уничтожением формы. Обработчик этого события может освободить ресурсы, разрушить служебные объекты, а также выполнить другие действия прежде, чем объект формы будет разрушен.

OnShow — происходит непосредственно перед отображением формы на экране. Парным для события OnShow является событие OnHide.

OnHide — происходит непосредственно перед исчезновением формы с экрана. Парным для события OnHide является событие OnShow.

OnActivate — происходит, когда пользователь переключается на форму, т.е. форма становится активной. Парным для события OnActivate является событие OnDeactivate.

OnDeactivate — происходит, когда пользователь переключается на другую форму, т.е. текущая форма становится неактивной. Парным для события OnDeactivate является OnActivate.

OnCloseQuery — происходит при попытке закрыть форму. Запрос на закрытие формы может исходить от пользователя, который нажал на рамке формы кнопку "Закрыть", или от программы, которая вызвала у формы метод Close. Обработчику события OnCloseQuery передается по ссылке булевский параметр CanClose, разрешающий или запрещающий действительное закрытие формы.

OnClose — происходит после события OnCloseQuery, непосредственно перед закрытием формы.

OnContextPopup — происходит при вызове контекстного меню формы.

OnMouseDown — происходит при нажатии пользователем кнопки мыши, когда указатель мыши наведен на форму. После отпускания кнопки мыши в компоненте происходит событие **OnMouseUp**. При перемещении указателя мыши над формой периодически возникает событие **OnMouseMove**, что позволяет отслеживать позицию указателя.

OnMouseWheelUp — происходит, когда колесико мыши проворачивается вперед (от себя).

OnMouseWheelDown — происходит, когда колесико мыши проворачивается назад (на себя).

OnMouseWheel — происходит, когда колесико мыши проворачивается в любую из сторон.

OnStartDock — происходит, когда пользователь начинает буксировать стыкуемый компонент.

OnGetSiteInfo — происходит, когда стыкуемый компонент запрашивает место для стыковки.

OnDockOver — периодически происходит при буксировке стыкуемого компонента над формой.

OnDockDrop — происходит при стыковке компонента (см. главу 10).

OnEndDock — происходит по окончании стыковки компонента.

OnUnDock — происходит, когда пользователь пытается отстыковать компонент.

OnDragDrop — происходит, когда пользователь опускает в форму буксируемый объект.

OnDragOver — периодически происходит при буксировке объекта над формой.

OnCanResize — происходит при попытке изменить размеры формы. Запрос на изменение размеров может исходить от пользователя. Обработчику события OnCanResize передается по ссылке булевский параметр **Resize**, разрешающий или запрещающий действительное изменение размеров формы.

OnResize — происходит при изменении размеров формы.

OnConstrainedResize — происходит при изменении размеров формы и позволяет на лету изменять минимальные и максимальные размеры формы.

OnShortCut — происходит, когда пользователь нажимает клавишу на клавиатуре (до события **OnKeyDown**, см. параграф 7.5.5). Позволяет перехватывать нажатия клавиш еще до того, как они дойдут до стандартного обработчика формы.

7.4. Несколько форм в приложении

Часто одной формы для решения задачи бывает мало. Поэтому сейчас мы рассмотрим, как добавить в проект новую форму, выбрать главную форму приложения, переключаться между формами. Затем мы расскажем, как на этапе работы приложения решается вопрос показа форм на экране.

7.4.1. Добавление новой формы в проект

Добавить в проект новую форму крайне просто: выберите команду меню **File | New | Form** и на экране сразу появиться вторая форма. При этом в окне редактора кода автоматически появится соответствующий новой форме программный модуль. Только что созданной форме дайте имя **SecondaryForm** (свойство **Name**) и заголовок **Secondary** (свойство **Caption**) — рисунок 7.29.

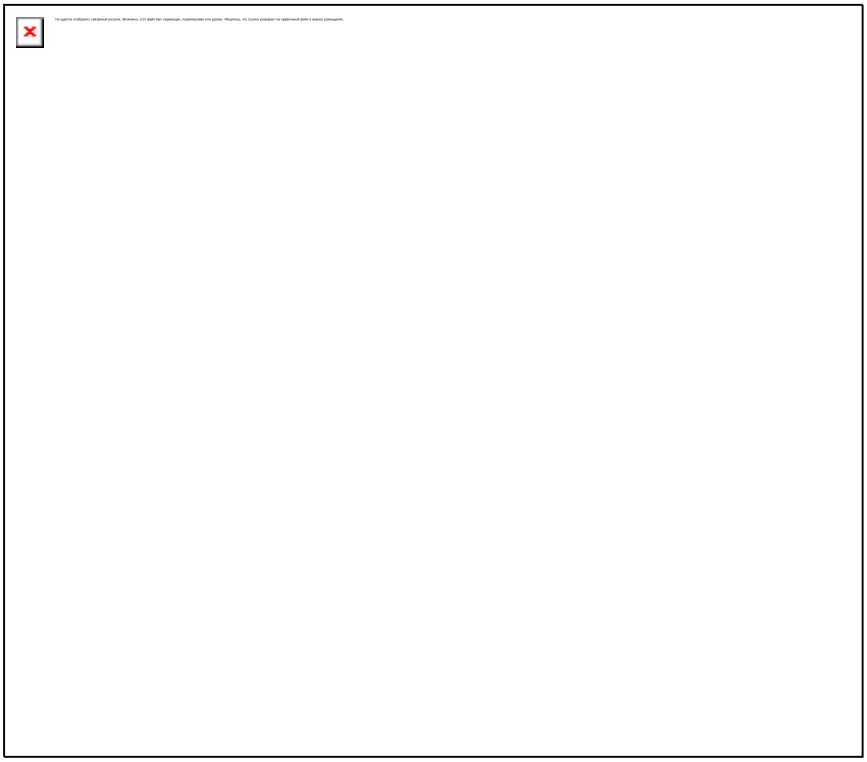


Рисунок 7.29. Две формы в проекте

Сохраните модуль с новой формой под именем **Second.pas** — форма нам еще понадобится.

7.4.2. Добавление новой формы из Хранилища Объектов

Существует и второй, более продуктивный, способ создания форм. Он основан на использовании готовых форм, существующих в Хранилище Объектов среды Delphi. *Хранилище Объектов* (**Object Repository**) содержит заготовки форм, программных модулей и целых проектов, которые вы можете либо просто скопировать в свой проект, либо унаследовать, либо вообще использовать напрямую. Чтобы взять новую форму из Хранилища объектов, выберите в меню команду **File | New | Other...**. Среда Delphi откроет окно, показанное на рисунке 7.30:

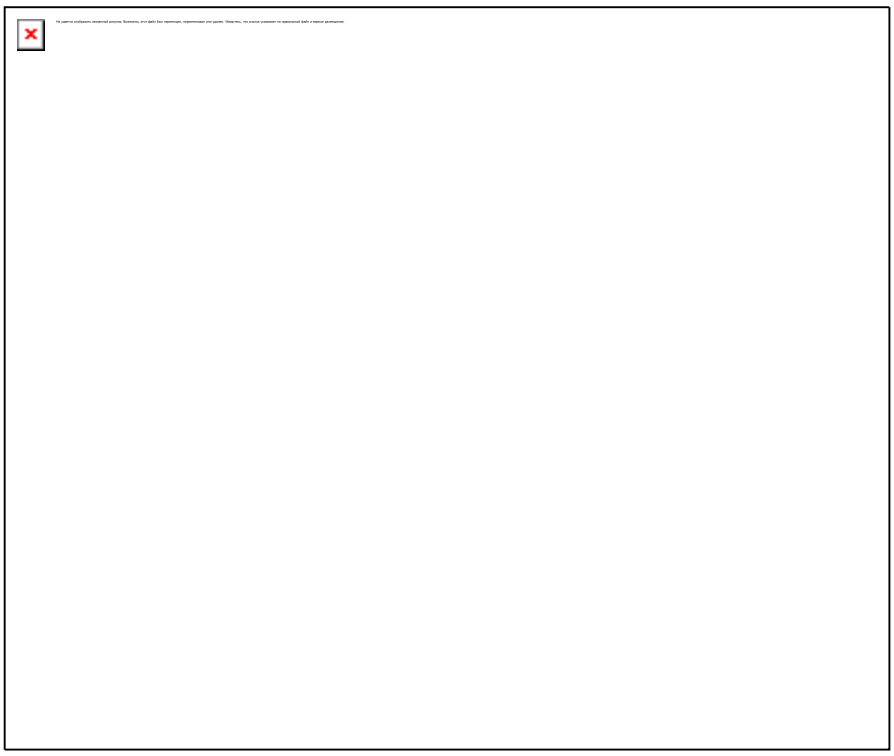


Рисунок 7.30. Окно создания новой формы или другого элемента проекта

Если на вкладке **New** диалогового окна выбрать значок с подписью **Form**, то в проект добавится обычная пустая форма, как по команде меню **File | New Form**. Если вас интересуют формы с «начинкой», обратитесь к вкладкам **Forms** и **Dialogs** (рисунок 7.31).

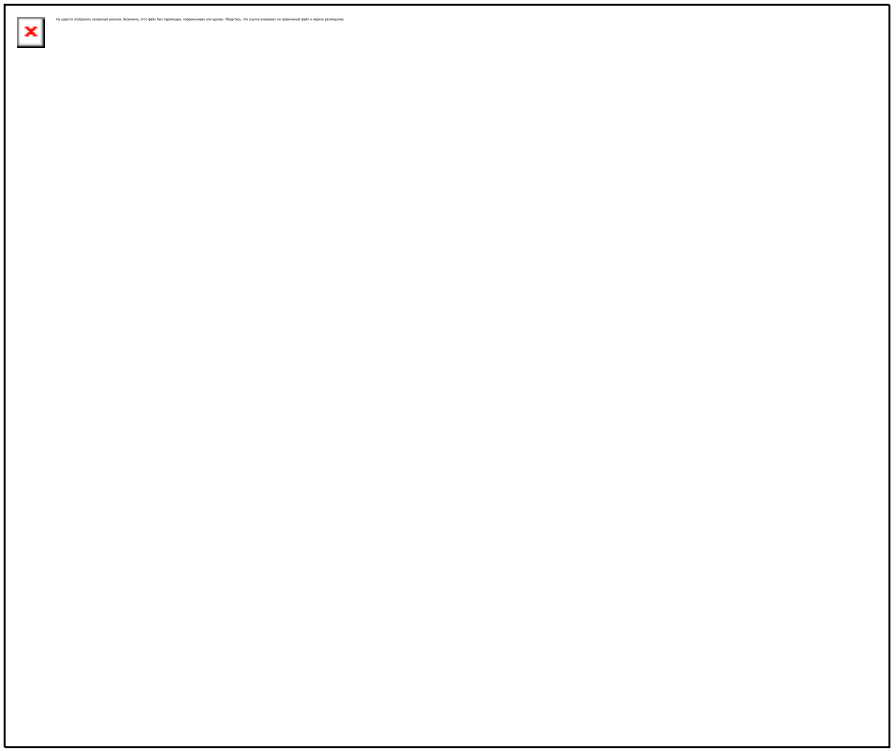


Рисунок 7.31. Быстрое создание формы с «начинкой»

На вкладках **Forms** и **Dialogs** существует переключатель, указывающий, что нужно сделать с формой-заготовкой: копировать (**Copy**), наследовать (**Inherit**) или использовать (**Use**). Отличие между ними состоит в следующем:

- **Copy** — означает, что в проект помещается полная копия формы-заготовки.

- **Inherit** — означает, что добавляемая в проект форма создается методом наследования от формы-заготовки, находящейся в Хранилище Объектов;
- **Use** — означает, что в проект добавляется сама форма-заготовка; изменение формы в проекте означает изменение формы-заготовки в Хранилище Объектов.

Какой из режимов использовать — зависит от условия задачи. Режим **Copy** хорош просто тем, что не с нуля начинает разработку новой формы. Режим **Inherit** полезен, когда в проекте существует несколько форм, у которых совпадают некоторые части. В этом случае все похожие между собой формы порождаются от какой-то одной формы, реализующей общую для всех наследников часть. Режим **Use** позволяет подкорректировать форму-заготовку прямо в Хранилище Объектов.

Для нашего учебного примера двух форм достаточно, поэтому вернемся к уже созданным формам, нажав кнопку Cancel.

7.4.3. Переключение между формами во время проектирования

Иногда за формами становиться трудно уследить. Навести порядок помогает окно **View Form**, для вызова которого служит команда меню **View | Forms...** (рисунок 7.32).

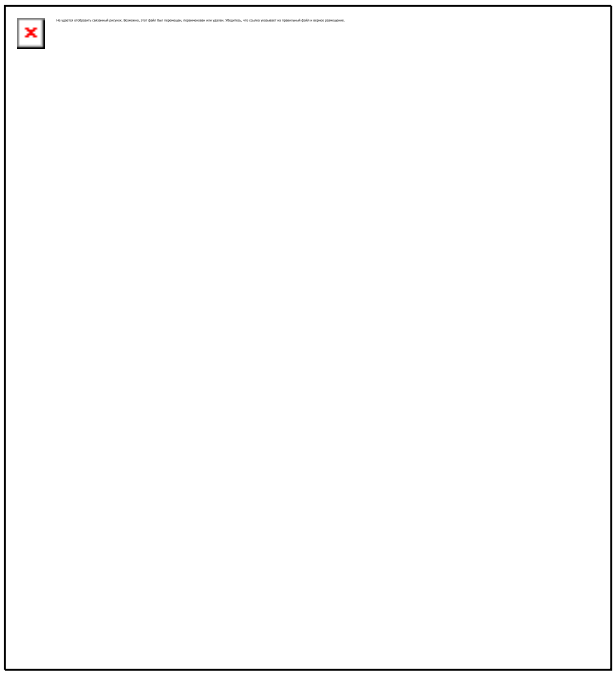


Рисунок 7.32. Окно для переключения на другую форму

Выберите в этом окне форму, с которой собираетесь работать, и щелкните по кнопке **OK**. Выбранная форма сразу же станет активной.

7.4.4. Выбор главной формы приложения

Когда в проекте несколько форм, возникает вопрос: какая из них главная. Давайте не будем ломать голову, а обратимся к известному вам диалоговому окну **Project Options** и посмотрим на вкладке **Forms**, какая форма выбрана в выпадающем списке **Main form** (рисунок 7.33).

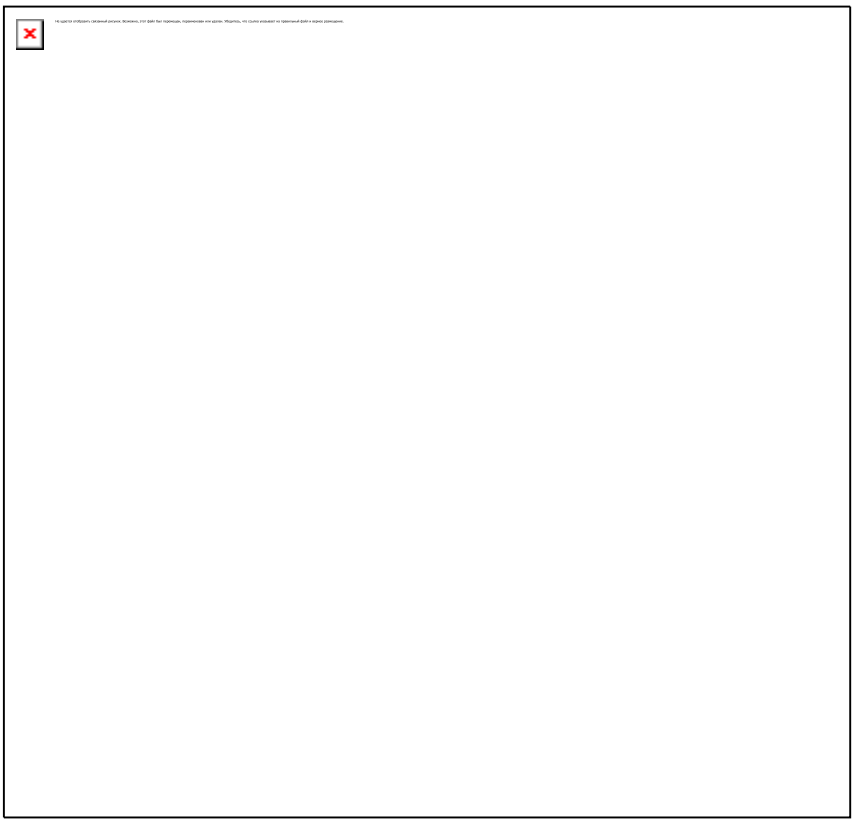


Рисунок 7.33. Главная форма в проекте записана в поле Main form

Вы обнаружите, что выбрана форма **MainForm**, которая была добавлена в проект первой (среда Delphi создает ее автоматически при создании нового проекта). Вы можете выбрать другую форму — и тогда она будет отображаться при запуске приложения. В данном случае этого делать не надо, поскольку главная форма уже установлена правильно.

7.4.5. Вызов формы из программы

Работая с несколькими формами, вы должны принимать во внимание, что после загрузки приложения отображается только главная форма. Остальные формы хотя и создаются вслед за ней автоматически, на экране сразу не показываются, а ждут пока их вызовут. Форму можно вызвать для работы двумя разными способами:

- вызвать для работы в обычном режиме с помощью метода **Show**. В этом режиме пользователь может работать одновременно с несколькими формами, переключаясь между ними;
- вызвать для работы в монопольном режиме с помощью метода **ShowModal**. В этом режиме пользователь не может переключиться на другую форму, пока не завершит работу с данной формой;

Покажем, как реализуются эти способы на примере вызова формы **SecondaryForm** из формы **MainForm**.

Чтобы форма **SecondaryForm** была доступна для использования формой **MainForm**, необходимо подключить модуль формы **SecondaryForm** к модулю формы **MainForm**. Это делается очень просто.

1. Активизируйте форму **MainForm** и выберите в главном меню команду **File | Use Unit...** . В появившемся диалоговом окне выберите модуль **Second** (так называется модуль формы **SecondaryForm**) и нажмите кнопку OK (рисунок 7.34).

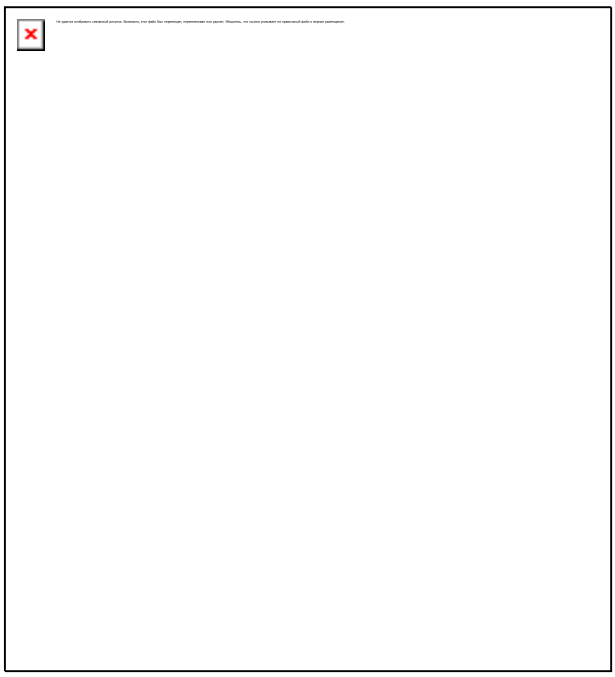


Рисунок 7.34. Окно для выбора подключаемого модуля

На экране не произойдет видимых изменений, но в секции **implementation** программного модуля Main добавится строка

```
uses Second;
```

Теперь обеспечим вызов формы **SecondaryForm** из формы **MainForm**. В большинстве случаев формы вызываются при нажатии некоторой кнопки. Добавим такую кнопку на форму **MainForm**.

2. Отыщите в палитре компонентов на вкладке **Standard** значок с подсказкой **Button**, щелкните по нему, а затем щелкните по форме **MainForm**. На форме будет создана кнопка **Button1** и в окне свойств отобразится список ее свойств. Перейдите к свойству **Caption** и замените текст **Button1** на текст **Secondary** (рисунок 7.35).

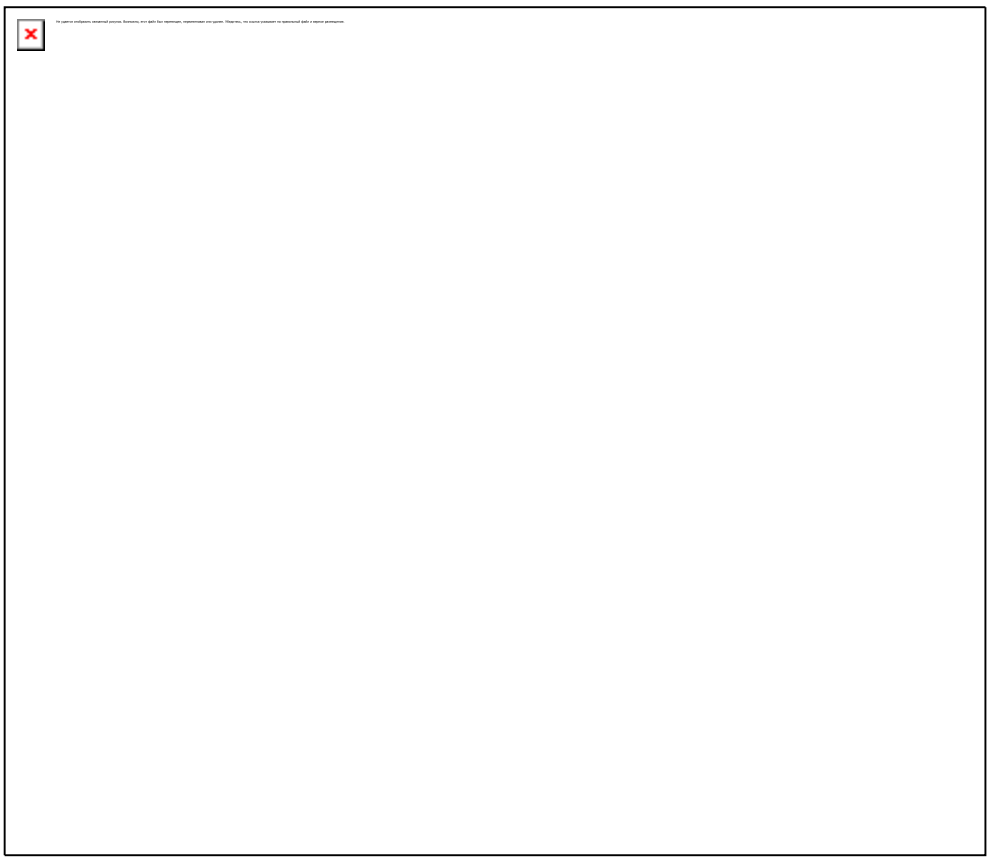


Рисунок 7.35. Текст на кнопке записывается в свойстве Caption

Чтобы при нажатии кнопки отображалась форма **SecondaryForm**, необходимо для этой кнопки определить обработчик события **OnClick**. Это делается очень просто.

3. Активизируйте в окне свойств вкладку **Events** и сделайте двойной щелчок в поле события **OnClick**. Среда Delphi определит для кнопки обработчик события, поместив программную заготовку в исходный текст модуля **Main**. Вставьте в тело обработчика оператор **SecondaryForm.Show**, в результате метод обработки события примет следующий вид:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  SecondaryForm.Show;
end;
```

4. Выполните компиляцию и запустите приложение. Когда на экране покажется форма **MainForm**, нажмите кнопку **Secondary**. На экране покажется еще одна форма — **SecondaryForm**. Вы можете произвольно активизировать любую из двух форм (рисунок 7.36).

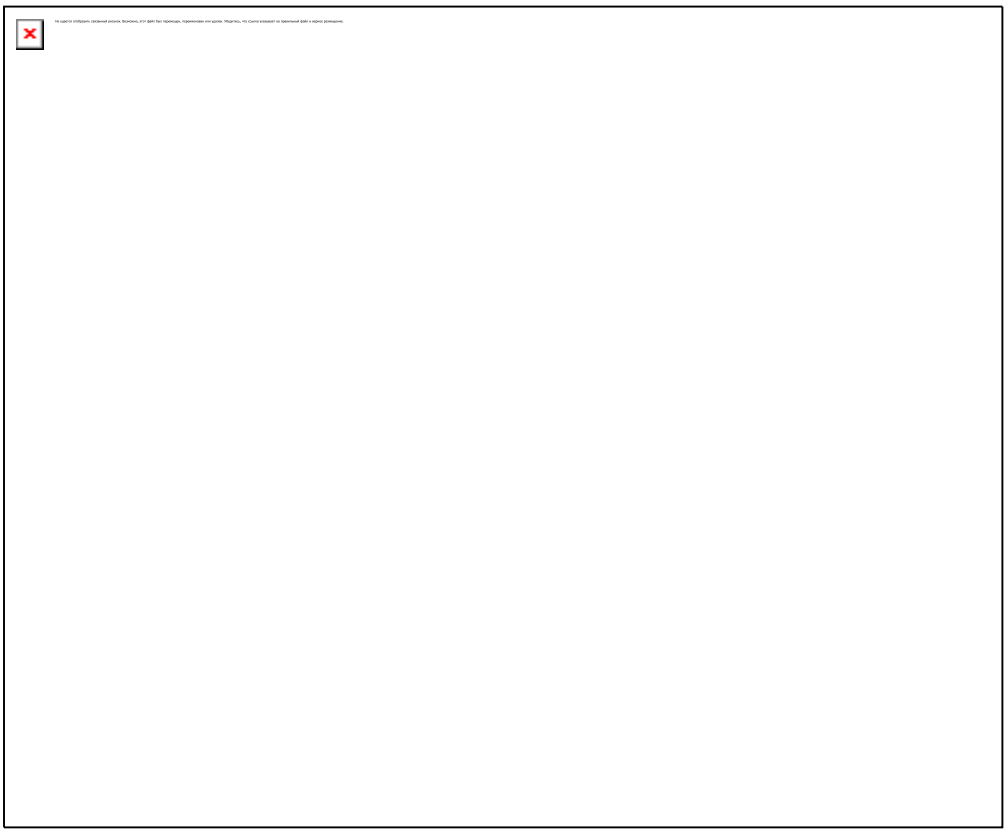


Рисунок 7.36. Приложение с двумя формами на экране

Таким образом, при использовании метода **Show** пользователь может работать одновременно с несколькими формами, переключаясь между ними.

Внимание! Переключаясь между формами **MainForm** и **SecondaryForm**, вы можете предположить, что они равноправны. Однако на самом деле это не так. Форма **MainForm** является главной, а форма **SecondaryForm** — второстепенной. В чем это проявляется? Да хотя бы в том, что если вы закроете форму **MainForm**, то форма **SecondaryForm** тоже закроется, и приложение завершится. Если же вы закроете форму **SecondaryForm**, то форма **MainForm** на экране останется.

Ситуация, когда пользователю предлагается для работы сразу несколько доступных форм, встречается редко. Поэтому для показа формы в основном применяется метод **ShowModal**. Он обеспечивает работу формы в монопольном режиме, не возвращая управление до тех пор, пока пользователь не закроет форму.

5. Посмотрим, что произойдет, если в предыдущем примере заменить вызов метода **Show** на **ShowModal**.

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  SecondaryForm.ShowModal;
end;
```

6. После компиляции и запуска приложения нажмите на форме **MainForm** кнопку **Secondary**. После появления формы **SecondaryForm** попробуйте активизировать форму **MainForm**. У вас ничего не выйдет, поскольку на этот раз форма **SecondaryForm** отображается в монопольном режиме (рисунок 7.37).

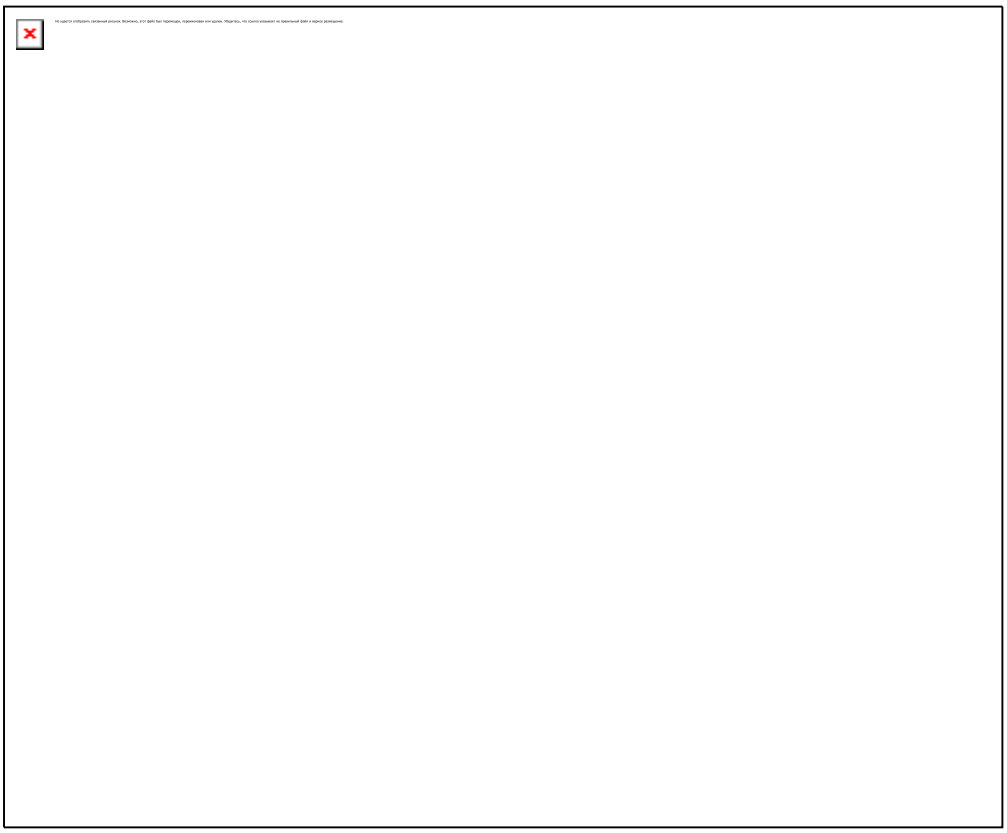


Рисунок 7.37. Вторая форма работает в монопольном режиме

Только закрыв форму **SecondaryForm**, вы вернетесь к форме **MainForm**. Теперь понятно и назначение метода **ShowModal**. С его помощью организуется пошаговый (диалоговый) режим взаимодействия с пользователем, который подробно рассмотрен в главе 9.

На этом мы закончим разговор о формах и перейдем к их содержимому — компонентам.

7.5. Компоненты

7.5.1. Понятие компонента

Понятие компонента является фундаментальным для среды Delphi. Без компонентов все преимущества визуальной разработки приложений исчезают и говорить становится не о чем. Поэтому соберите все силы и внимательно прочитайте этот параграф, пытаясь усвоить не только технику использования компонентов, но и саму их суть.

Существует два взгляда на компоненты.

- Взгляд снаружи, точнее из среды визуальной разработки приложений. С этой точки зрения компоненты — это самодостаточные строительные блоки, которые вы берете из палитры компонентов и переносите на форму для создания собственно приложения. Примеры компонентов вам известны: это кнопки, списки, надписи и др.
- Существует еще и взгляд изнутри, т.е. взгляд из программы на языке Delphi. С этой точки зрения компоненты — это классы, порожденные прямо или косвенно от класса **TComponent** и имеющие **published**-свойства. Экземпляры компонентов — это объекты этих классов, существующие в качестве полей формы. Среди опубликованных свойств компонентов обязательно присутствует имя (**Name**), под которым экземпляр компонента представляется в окне свойств.

Объединение этих двух точек зрения дает цельное представление о том, что такое компоненты. При работе с компонентами из среды визуальной разработки приложений вы всегда видите их лицевую сторону. Однако как только вы начинаете писать обработчики

событий, и управлять компонентами программно, вы соприкасаетесь с программной стороной компонентов, суть которой — объекты. Таким образом, среда Delphi обеспечивает симбиоз визуального и объектно-ориентированного программирования.

При анализе структуры компонента обнаруживается, что его природа троична и лучше всего описывается формулой:

Компонент = состояние (свойства) + поведение (методы) + обратная реакция (события).

- *Состояние компонента* определяется его свойствами. Свойства бывают изменяемые (для чтения и записи) и неизменяемые (только для чтения). Помимо этого, свойства могут получать значения либо на этапе проектирования (design-time), либо только во время выполнения программы (run-time). Первые устанавливаются для каждого компонента в окне свойств и определяют начальное состояние компонента. Во время выполнения приложения эти свойства могут быть изменены программно, соответственно изменится внешний вид и поведение компонента. Вторая группа — это свойства, которые не видны в окне свойств, и управлять которыми можно только программно. С точки зрения языка Delphi различие между этими группами свойств состоит в том, что первые объявлены в секции **published**, а вторые — в секции **public**.
- *Поведение компонента* описывается с помощью его процедур и функций (*методов*). Вызовы методов компонента помещаются в исходный код программы и происходят только во время выполнения приложения. Методы не имеют под собой визуальной основы.
- *Обратная реакция компонента* — это его *события*. События позволяют, например, связать нажатие кнопки с вызовом метода формы. События реализуются с помощью свойств, содержащих указатели на методы (см. гл. 3).

7.5.2. Визуальные и невизуальные компоненты

Все компоненты делятся на две группы: визуальные и невизуальные компоненты (рисунок 7.38).

- *Визуальные компоненты* (visual components) — это видимые элементы пользовательского интерфейса: кнопки, метки, блоки списков и др. Они выглядят одинаково и на стадии проектирования, и во время работы приложения.
- *Невизуальные компоненты* (non-visual components) — это, так сказать, бойцы невидимого фронта; они работают, но сами на экране не видны. К невизуальным компонентам относятся таймер, компоненты доступа к базам данным и др. В процессе проектирования такие компоненты представляются на форме небольшим значком. Их свойства устанавливаются в уже известном вам окне свойств. Некоторые компоненты хоть и являются невизуальными, могут что-нибудь отображать на экране. Например, невизуальный компонент **MainMenu** отображает на форме полосу главного меню, а компонент **OpenDialog** — стандартное диалоговое окно выбора файла.

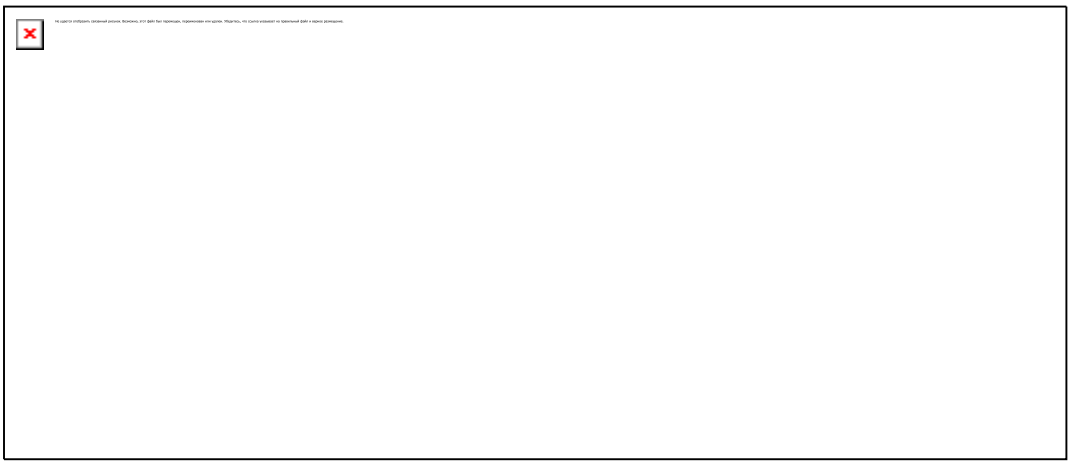


Рисунок 7.38. Визуальные и невизуальные компоненты

Невизуальные компоненты могут иметь подписи (рисунок 7.38). Отображение подписей обеспечивается установкой переключателя **Show component captions** в окне **Environment Options** на вкладке **Designer**. Окно вызывается по команде меню **Tools | Environment Options…**

7.5.3. «Оконные» и «графические» компоненты

Визуальные компоненты подразделяются на компоненты, рисуемые оконной системой Windows, и компоненты, рисуемые графической библиотекой VCL (рисунок 7.39). На программистском жаргоне первые называют «оконными» компонентами, а вторые — «графическими» компонентами.

- *«Оконные» компоненты* (windowed controls) являются специализированными окнами внутри окна формы. Их самое главное качество — способность получать фокус ввода. К числу оконных компонентов относятся, например, компоненты **Button**, **RadioButton**, **CheckBox**, **GroupBox**, и т.д. Некоторые оконные компоненты (**GroupBox**, **TabControl**, **PageControl**) способны содержать другие визуальные компоненты и называются *контейнерами* (container controls). Отображение оконных компонентов обеспечивается операционной системой Windows. Для профессионалов, имевших дело Windows API, заметим, что оконные компоненты имеют свойство **Handle**. Оно связывает компонент среды Delphi с соответствующим объектом операционной системы.
- *«Графические» компоненты* (graphical controls) не являются окнами, поэтому не могут получать фокус ввода и содержать другие визуальные компоненты. Графические компоненты не основаны на объектах операционной системы Windows, их отображение полностью выполняет библиотека VCL. К числу графических компонентов относятся, например, компоненты **SpeedButton**, **Image**, **Bevel** и т.д.

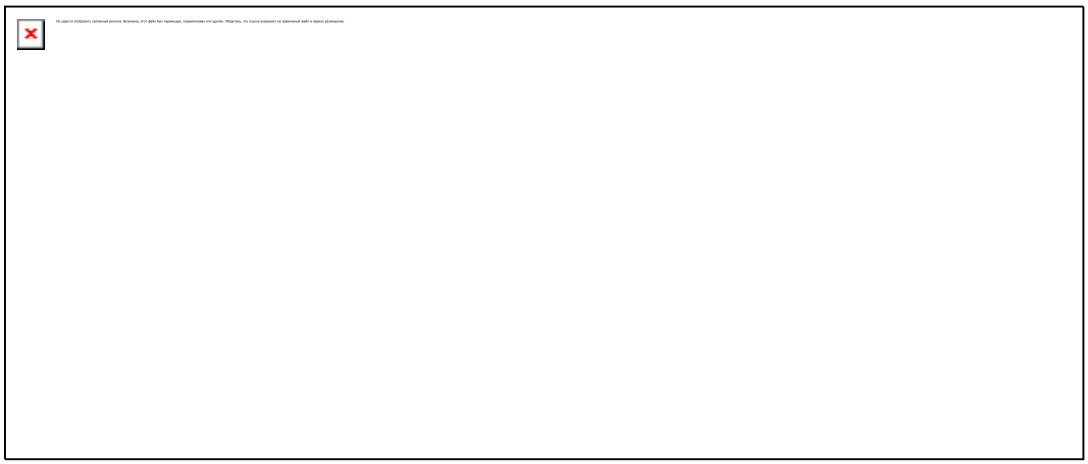


Рисунок 7.39. Компоненты, рисуемые оконной системой Windows и графической библиотекой Delphi

Общая классификация компонентов составлена, поэтому перейдем к обсуждению их свойств и событий. Очевидно, каждый компонент обладает специфичным набором свойств и событий и, казалось бы, изучать их следует в контексте изучения компонента. Так мы и будем поступать в будущем при рассмотрении отличительных свойств компонентов. Однако сейчас имеет смысл рассмотреть общие для большинства компонентов свойства и события.

Невизуальные компоненты практически не имеют общих свойств и событий, единственные общие для них свойства — это **Name** (комментариев не требует) и **Tag** (целочисленное значение, не несущее смысловой нагрузки — вы можете использовать его по своему усмотрению). А вот визуальные компоненты имеют много общих свойств и событий, которые мы сейчас и рассмотрим.

7.5.4. Общие свойства визуальных компонентов

Визуальные компоненты имеют ряд общих свойств:

Left и **Top** — местоположение визуального компонента внутри формы (или внутри компонента-владельца).

Width и **Height** — горизонтальный и вертикальный размеры компонента соответственно.

Anchors — позволяет привязать границы компонента к границам формы. Привязанная граница компонента будет следовать за соответствующей границей формы при изменении размеров формы. Поэкспериментируйте со значениями этого свойства и вы быстро уловите логику его работы.

BiDiMode — позволяет сделать так, чтобы текст читался справа налево (используется при работе с восточными языками). Компонент либо использует свое собственное значение свойства, либо копирует его из компонента-владельца, если вспомогательное свойство **ParentBiDiMode** равно значению True.

Caption — надпись компонента. Установленная в свойстве текстовая строка может содержать специальный символ ‘&’ (амперсант). Если в строке встречается амперсант, то следующий за ним символ отображается подчеркнутым (амперсант при этом не отображается). Нажатие соответствующей символьной клавиши на клавиатуре в сочетании с клавишей Alt активизирует компонент.

Constraints — ограничения на размеры компонента. Вложенные свойства **MinWidth** и **MinHeight** определяют минимальные ширину и высоту, а вложенные свойства **MaxWidth** и **MaxHeight** — максимальные ширину и высоту соответственно.

Color — цвет компонента. Компонент либо использует свой собственный цвет, либо копирует цвет содержащего компонента. Это определяется значением свойства **ParentColor**.

Если свойство ParentColor имеет значение True, то изменение цвета у содержащего компонента (например, формы) автоматически приводит к изменению цвета вложенного компонента (например, кнопки). Однако, если вы измените значение свойства Color, то свойство ParentColor автоматически примет значение False, и компонент получит свой собственный цвет.

Cursor — определяет, какой вид принимает указатель мыши, когда пользователь наводит его на компонент. Каждому варианту указателя соответствует своя целочисленная константа (например, константа **crArrow** соответствует обычному указателю в виде стрелки). Полный список значений с описанием вы сможете найти в справочной системе среды Delphi.

DragCursor — вид указателя мыши, когда пользователь буксирует объект над компонентом. Этот вид курсора устанавливается лишь в том случае, если объект может быть принят (см. главу 10).

DragKind — определяет поведение компонента при буксировке: просто буксировка (**dkDrag**) или стыковка (**dkDock**). В зависимости от значения этого свойства возникает та или иная цепочка событий: цепочка событий буксировки или цепочка событий стыковки.

DragMode — определяет режим буксировки компонента по экрану. Если в свойстве установлено значение dmManual (принято по умолчанию), то буксировка должна инициироваться программно. Если же в свойстве установлено значение dmAutomatic, то компонент уже готов к буксировке, пользователю достаточно навести указатель мыши на компонент, нажать кнопку мыши и, удерживая ее, отбуксировать компонент в нужное место.

Enabled — определяет, доступен ли компонент для пользователя. Если свойство имеет значение True, то компонент доступен, а если значение False, то недоступен. Недоступный компонент обычно имеет блеклый вид.

Font — шрифт надписи на компоненте. Параметры шрифта задаются с помощью вложенных свойств **CharSet**, **Color**, **Name**, **Size**, **Style**, **Height**, **Pitch**, **Weight**. Компонент либо использует свой собственный шрифт, либо копирует шрифт содержащего компонента. Это определяется значением свойства **ParentFont**. Если свойство ParentFont имеет значение True, то изменение шрифта у содержащего компонента (например, формы) автоматически приводит к изменению шрифта у вложенного компонента (например, кнопки). Однако, если вы измените значение свойства Font, то свойство ParentFont автоматически примет значение False, и компонент получит свой собственный шрифт для надписи.

HelpType — определяет, каким образом в файле справки будет осуществляться поиск темы, соответствующей компоненту. Когда компонент обладает фокусом ввода, пользователь может нажать клавишу F1, чтобы получить оперативную справку. Поиск соответствующей темы осуществляется либо по номеру, заданному в свойстве **HelpContext**, либо по ключевому слову, заданному в свойстве **HelpKeyword**. В первом случае свойство HelpType должно иметь значение **htContext**, а во втором — **htKeyword**.

HelpContext — содержит номер соответствующей темы в файле справки. Используется, когда свойство **HelpType** имеет значение **htContext**. Если свойство HelpContext имеет значение 0, то номер темы берется из аналогичного свойства компонента-владельца (как правило, формы).

HelpKeyword — содержит ключевое слово для поиска темы в файле справки. Используется, когда свойство **HelpType** имеет значение **htKeyword**. Если свойство HelpKeyword имеет пустое значение, то поиск осуществляется по ключевому слову, которое берется из аналогичного свойства компонента-владельца (как правило, формы).

Hint — подсказка, появляющаяся над компонентом, когда пользователь временно задерживает над ним указатель мыши. Появление подсказки может быть разрешено или запрещено с помощью свойства **ShowHint**. Значение свойства ShowHint может копироваться

из содержащего компонента в зависимости от значения свойства **ParentShowHint**. Если свойство ParentShowHint имеет значение True, то запрет подсказки для содержащего компонента (например, для формы), автоматически приводит к запрету подсказки для вложенного компонента (например, для кнопки). Однако, если вы измените значение свойства ShowHint, то свойство ParentShowHint автоматически примет значение False, и управление запретом подсказки перейдет к компоненту.

PopupMenu — используется для привязки контекстного меню к компоненту. Это меню вызывается щелчком правой кнопки мыши по компоненту. Меню подробно рассмотрены в главе 8.

TabOrder — содержит порядковый номер компонента в пределах своего компонента-владельца. Это номер очереди, в которой компонент получает фокус ввода при нажатии клавиши Tab на клавиатуре. Свойство TabOrder присутствует только в оконных компонентах.

TabStop — определяет, может ли компонент получать фокус ввода. Если свойство имеет значение True, то компонент находится в очереди на фокус ввода, а если значение False, то — нет. Свойство TabStop присутствует только в оконных компонентах.

Visible — определяет видимость компонента на экране. Если свойство имеет значение True, то компонент виден, а если значение False, то — не виден.

7.5.5. Общие события визуальных компонентов

Визуальные компоненты имеют ряд общих событий:

OnClick — происходит в результате щелчка мыши по компоненту.

OnContextPopup — происходит при вызове контекстного меню компонента.

OnDblClick — происходит в результате двойного щелчка мыши по компоненту.

OnEnter — происходит при получении компонентом фокуса ввода. Когда компонент теряет фокус ввода, происходит событие **OnExit**. События OnEnter и OnExit не происходят при переключении между формами и приложениями.

OnKeyDown — происходит при нажатии пользователем любой клавиши (если компонент обладает фокусом ввода). При отпускании нажатой клавиши происходит событие **OnKeyUp**. Если пользователь нажал символьную клавишу, то вслед за событием OnKeyDown и до события OnKeyUp происходит событие **OnKeyPress**. События о нажатии клавиш обычно приходят активному компоненту, обладающему фокусом ввода. Однако с помощью свойства формы **KeyPreview** можно сделать так, чтобы форма перехватывала клавиатурные события до того, как их получит активный компонент. Для этого свойство KeyPreview устананавливается в значение True.

OnMouseDown — происходит при нажатии пользователем кнопки мыши, когда указатель мыши наведен на компонент. После отпускания кнопки мыши в компоненте происходит событие **OnMouseUp**. При перемещении указателя мыши над компонентом, в последнем периодически возникает событие **OnMouseMove**, что позволяет отслеживать позицию указателя.

Для организации буксировки и стыковки, в визуальных компонентах существует еще несколько событий:

OnStartDrag — происходит, когда пользователь начинает что-нибудь буксировать.

OnDragOver — периодически происходит, когда пользователь буксирует что-нибудь над компонентом.

OnDragDrop — происходит, когда пользователь опускает буксируемый объект на компонент.

OnEndDrag — происходит по окончании буксировки объекта.

OnStartDock — происходит, когда пользователь начинает буксировать стыкуемый компонент.

OnEndDock — происходит по окончании стыковки компонента.

Подробно события буксировки и стыковки рассмотрены в главе 10.

7.6. Управление компонентами при проектировании

7.6.1. Помещение компонентов на форму и их удаление

Чтобы поместить на форму нужный компонент из палитры компонентов, выполните следующие действия:

9. Наведите указатель мыши на значок нужного компонента в палитре и щелкните левой кнопкой мыши.
10. Наведите указатель мыши на нужное место формы и еще раз щелкните левой кнопкой мыши.

Выбранный компонент окажется на форме и будет готов к настройке в окне свойств.

Часто требуется разместить на форме несколько компонентов одного и того же типа, например, кнопок. В этом случае действуйте следующим образом:

11. Наведите указатель мыши на значок нужного компонента в палитре. Нажмите клавишу Shift и, удерживая ее, щелкните левой кнопкой мыши. Отпустите клавишу Shift.
12. Наведите указатель мыши на то место формы, где будет находиться первый компонент, и щелкните левой кнопкой мыши.
13. Наведите указатель мыши на то место формы, где будет размещен следующий компонент, и снова щелкните левой кнопкой мыши. Повторите это действие столько раз, сколько вам нужно компонентов;
14. Разместив последний компонент, наведите указатель мыши на кнопку с изображением стрелки (она расположена в левой части палитры компонентов), и щелкните левой кнопкой мыши. Это будет сигналом, что размещение однотипных компонентов закончено.

Если вы по каким-либо причинам решили убрать лишний компонент с формы, просто выберите его с помощью мыши и нажмите клавишу Del.

7.6.2. Выделение компонентов на форме

На стадии проектирования любой компонент может быть выделен на форме. Свойства выделенного компонента видны в окне свойств и доступны для редактирования. Чтобы выделить компонент, достаточно навести на него указатель и нажать кнопку мыши. Вокруг компонента тут же появятся так называемые "точки растяжки" (sizing handles) для изменения размеров компонента по ширине и высоте (рисунок 7.40).

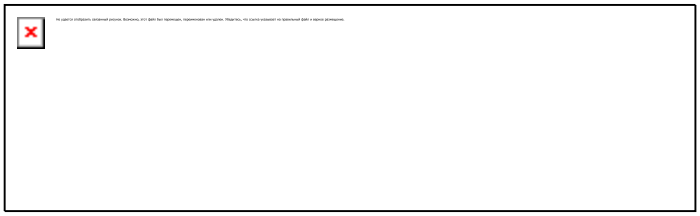


Рисунок 7.40. Точки растяжки компонента

При проектировании сложных форм вы столкнетесь с ситуацией, когда сразу в нескольких компонентах нужно установить некоторое свойство в одно и то же значение. Например, в нескольких кнопках установить свойство **Enabled** в значение False. Быстрее всего это можно сделать, если выделить несколько компонентов, после чего перейти к окну свойств и изменить нужное свойство. Когда на форме выделено несколько компонентов, в окне свойств видны только их общие свойства.

Выделить несколько компонентов можно двумя способами:

- Нажать клавишу Shift и, удерживая ее, отметить щелчками мыши все требуемые компоненты, после чего клавишу Shift отпустить. В углах каждого выделенного компонента появятся небольшие квадратики-маркеры.
- Нажать кнопку мыши, когда курсор находится вне компонентов. Затем, удерживая кнопку в нажатом состоянии, протянуть курсор над выделяемыми компонентами, включив их в пунктирный прямоугольник. Когда в пунктирный прямоугольник попадут все требуемые компоненты, кнопку мыши следует отпустить. (Если выделяемые компоненты находятся внутри компонента Panel или GroupBox, то эту операцию нужно выполнять с нажатой клавишей Ctrl.) В результате перечисленных действий в углах всех компонентов, хотя бы частично попавших в пунктирный прямоугольник, появятся небольшие квадратики-маркеры, свидетельствующие о том, что компоненты выделены.

Вы можете комбинировать оба способа для выделения лишь тех компонентов, которые вам нужны.

Когда на форме выделено несколько компонентов, в окне свойств отображаются лишь их общие свойства. Активизируйте нужное свойство и установите в нем нужное значение. Вы увидите, что эта установка отразится на всех выделенных компонентах (рисунок 7.41).

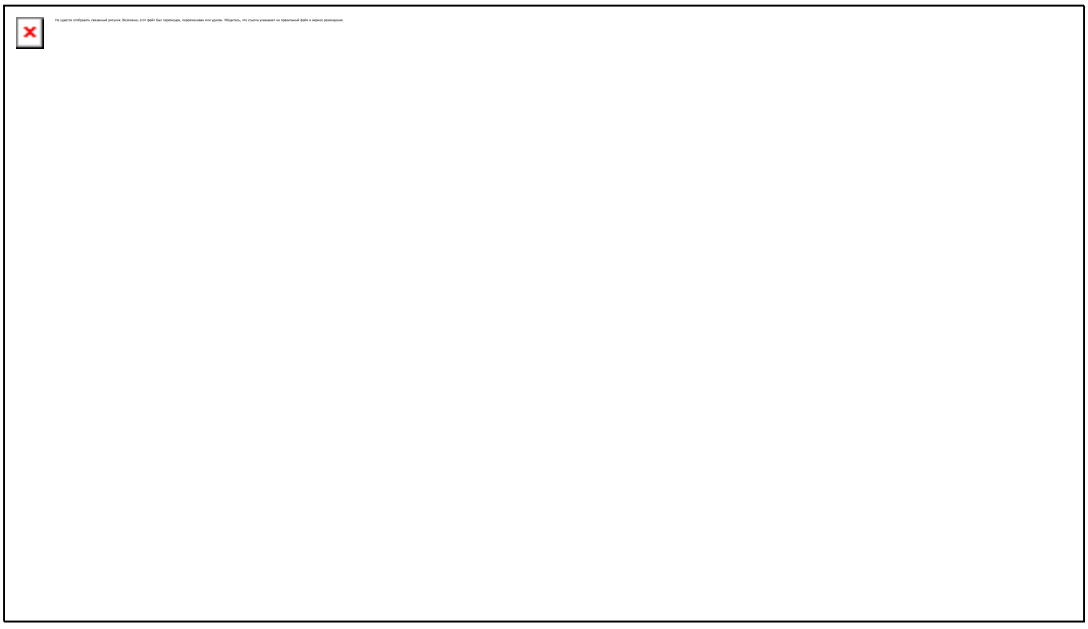


Рисунок 7.41. Установка свойства для группы компонентов

Когда на форме выделено несколько компонентов, некоторые свойства могут как бы не иметь значения в окне **Object Inspector** (в поле значения — пусто). Это говорит о том, что свойства имеют различные значения в выделенных компонентах.

7.6.3. Перемещение и изменение размеров компонента

Когда все компоненты помещены на форму, нужно оценить ее с точки зрения эстетики. Отойдите от компьютера и посмотрите на форму со стороны. Как правило, в пейзаже обнаружатся некоторые изъяны. Наверняка что-то захочется уменьшить, что-то увеличить, что-то переместить на другое место. Сделать все это — проще простого.

Для перемещения компонента на другое место:

15. Поместите курсор над компонентом, который хотите переместить, и щелкните мышью (компонент тут же окажется в фокусе). Не отпуская кнопки мыши, отбуксируйте компонент на новое место.
16. Когда компонент будет там, где надо, отпустите кнопку мыши.

Изменить размер компонента тоже просто:

17. Щелчком мыши активизируйте компонент, размер которого хотите изменить (он тут же окажется в фокусе);
18. Наведите указатель мыши на точку вертикальной или горизонтальной “растяжки”, при этом вид указателя изменится на двунаправленную стрелку. Нажмите кнопку мыши и, удерживая ее, перемещайте указатель в сторону уменьшения или увеличения размера компонента;
19. Добившись желаемого размера, отпустите кнопку мыши и отведите указатель от точки растяжки (при этом указатель примет обычный вид). Компонент с новыми размерами готов к работе.

Чтобы упростить вам позиционирование и изменение размеров компонентов, форма отображает на этапе разработки *сетку* (grid). Компоненты автоматически выравниваются на пересечении воображаемых линий сетки, когда вы переносите их из палитры компонентов на форму. Изначально шаг между горизонтальными и вертикальными линиями сетки равен 8, но его легко изменить. Для этого выполните команду меню **Tools | Environment Options...** . На экране появится диалоговый блок **Environment Options**. Отыщите поля **Grid size X** и **Grid size Y** на вкладке **Designer** и установите те параметры сетки, которые вам нужны (рисунок 7.42). Кстати, с помощью соседних переключателей можно указать среде Delphi, следует ли вообще показывать сетку (**Display grid**) и следует ли выравнивать по ней компоненты (**Snap to grid**).

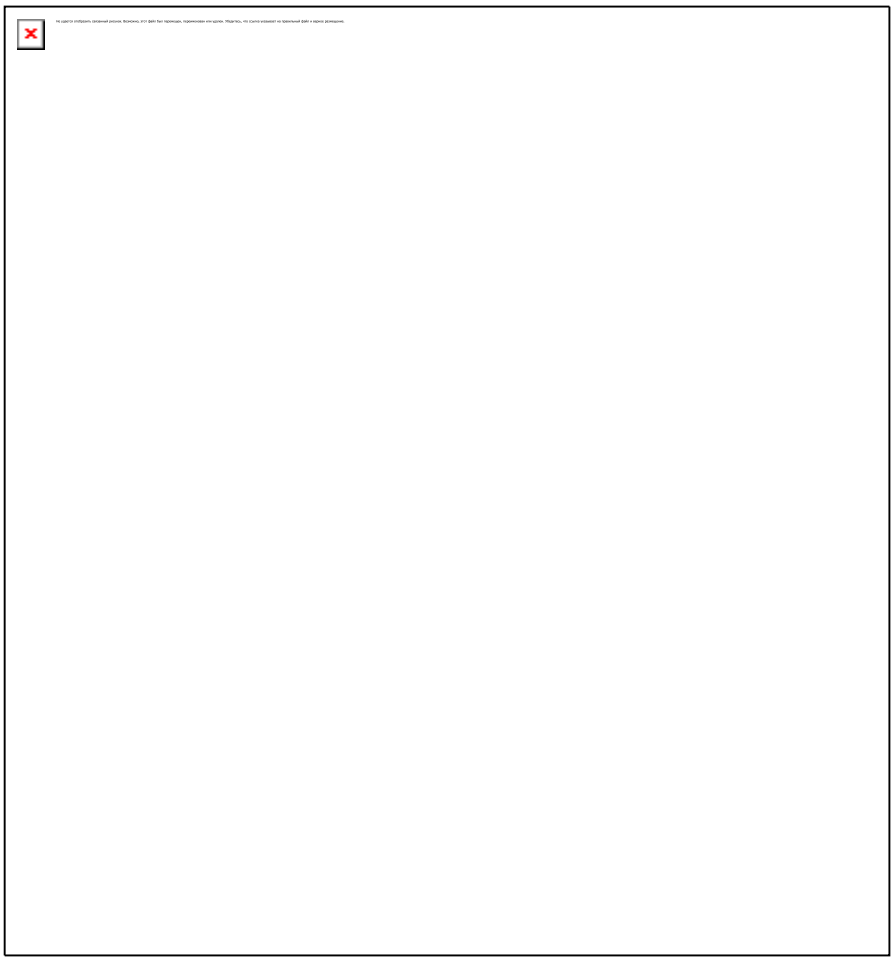


Рисунок 7.42. Расстояние между узлами сетки по горизонтали и вертикали

Иногда после грубого размещения компонента на сетке необходимо подогнать его положение и размеры с точностью до точки экрана. Для этого не требуется отключать сетку или изменять ее шаг. Просто выберите компонент на форме и действуйте следующим образом:

- Нажмите клавишу Ctrl и, удерживая ее нажатой, с помощью клавиш со стрелками подвиньте компонент на нужное количество точек экрана. Отпустите клавишу Ctrl.
- Нажмите клавишу Shift и, удерживая ее нажатой, с помощью клавиш со стрелками растяните или сожмите компонент на нужное количество точек экрана. Отпустите клавишу Shift.

7.6.4. Выравнивание компонентов на форме

Когда на форме много компонентов, ручное выравнивание становится весьма утомительным занятием. На этот случай в среде Delphi предусмотрены средства автоматизированного выравнивания компонентов. Алгоритм выравнивания следующий:

20. Выделите компоненты, которые собираетесь выравнять. Во всех четырех углах каждого выделенного компонента появятся небольшие квадратики-маркеры;
21. Обратитесь к главному меню и вызовите окно **Alignment** (рисунок 7.43) с помощью команды меню **Edit | Align...** .

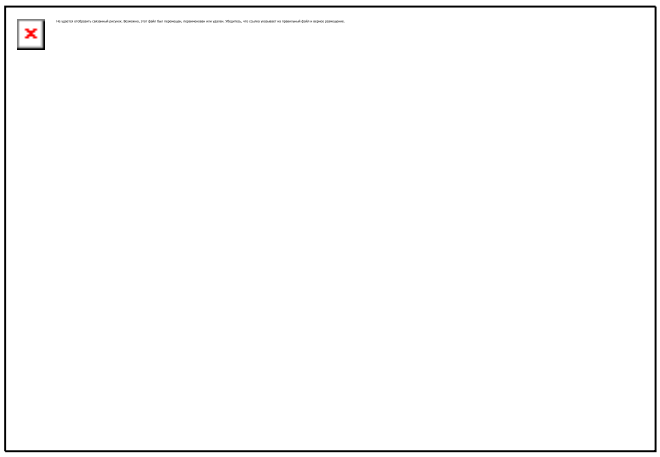


Рисунок 7.43. Диалоговое окно для выбора способа выравнивания группы компонентов на форме

22. Выберите в списке то, что вам надо, и нажмите кнопку **OK**. Окно закроется и все компоненты будут выравнены согласно вашим указаниям.

Если компонентов на форме много и вам предстоит большая работа по их выравниванию, откройте окно **Align** (с помощью команды меню **View | Alignment Palette**) и используйте его на втором шаге приведенного выше алгоритма (рисунок 7.44).

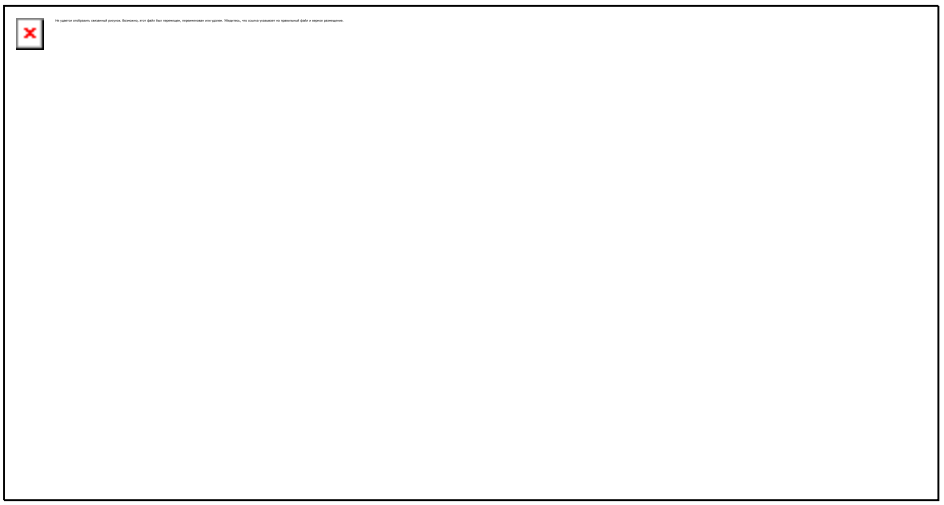


Рисунок 7.44. Вспомогательная панель кнопок для выравнивания группы компонентов на форме

7.6.5. Использование Буфера обмена

При работе с несколькими формами иногда встает задача копирования и перемещения компонентов с одной формы на другую. Обычное перетягивание здесь не помогает. Проблема легко решается с помощью Буфера обмена:

23. С помощью известных приемов выберите на первой форме компоненты, которые вы желаете скопировать или переместить на вторую форму.
24. Если вы собираетесь скопировать компоненты, выберите в меню команду **Edit | Copy**. Если вы собираетесь переместить компоненты, выберите в меню команду **Edit | Cut**. Компоненты окажутся в Буфере обмена.
25. Активизируйте вторую форму и выберите в меню команду **Edit | Paste**. По этой команде среда Delphi извлечет компоненты из Буфера обмена и поместит их на активную форму.

Добавим, что команды работы с Буфером обмена применяются не только для копирования и перемещения компонентов с одной формы на другую, но также для копирования и перемещения компонентов в пределах одной формы между разными компонентами-владельцами, например для перемещения кнопок с одной панели на другую. Необходимость использования Буфера обмена в этом случае вызвана тем, что компоненты твердо знают своего владельца (например, кнопки знают панель, на которой они расположены), поэтому обычная операция буксировки ни к чему не приводит.

Итак, вы уже много знаете о компонентах, и дальше углубляться в них не имеет смысла. Начиная со следующей главы, мы начнем знакомить вас с элементами пользовательского интерфейса: меню, панелью инструментов, строкой состояния, диалоговыми окнами и др. — вот там и поговорим о деталях. А сейчас скажем несколько слов о тех объектах, которые усердно работают "за кулисами" приложения и обеспечивают ему доступ к различным ресурсам компьютера, например экрану, принтеру, Буферу обмена и др.

7.7. Закулисные объекты приложения

7.7.1. Application — главный объект, управляющий приложением

То, о чем мы рассказали выше — это внешняя сторона приложения. А что же происходит внутри? Дело обстоит так. Над всеми формами и компонентами стоит объект **Application** (класса TApplication), олицетворяющий собой приложение в целом. Это главное “действующее лицо”, которое создается в начале выполнения любого приложения. Объект **Application** держит в руках все нити управления: создает главную и второстепенные формы, уничтожает их, обслуживает исключительные ситуации. Вы, кстати, уже встречались с ним в файле проекта:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Объект **Application** отсутствует в палитре компонентов, поэтому его свойства можно изменять только из программы. Кратко рассмотрим наиболее важные свойства этого объекта:

Active — равно значению True, если приложение активно.

AutoDragDocking — режим автоматической или ручной стыковки форм и компонентов. В автоматическом режиме (значение True) стыковка происходит по окончании буксировки при отпускании кнопки мыши. В ручном режиме (значение False) для стыковки необходимо удерживать клавишу Ctrl при отпускании кнопки мыши.

BiDiKeyboard — раскладка клавиатуры при работе с восточными языками.

BiDiMode — позволяет сделать так, чтобы надписи читались справа налево (используется при работе с восточными языками).

CurrentHelpFile — имя файла справки активной формы программы (каждая форма может иметь свой собственный файл справки). Если у активной формы нет своего файла справки, то в свойстве **CurrentHelpFile** просто дублируется значение свойства **HelpFile**.

HintColor — цвет фона всплывающей подсказки.

HintHidePause — время, в течение которого всплывающая подсказка задерживается на экране.

HintPause — задержка перед появлением всплывающей подсказки.

HintShortCuts — определяет, включается ли в текст подсказки название комбинации клавиш.

HintShortPause — время, через которое появляется всплывающая подсказка, если в данный момент на экране уже отображена другая подсказка.

MainForm — указывает главную форму приложения. По умолчанию главной считается первая создаваемая форма.

NonBiDiKeyboard — раскладка клавиатуры.

ExeName — содержит полное имя (включая маршрут) выполняемого файла программы. Имя выполняемого файла совпадает с именем главного файла проекта. Если имя проекта не было указано, то по умолчанию выполняемому файлу назначается имя Project1.

Title — содержит название приложения, которое отображается на Панели Задач во время работы. По умолчанию значением свойства является имя главного файла проекта.

HelpFile — содержит имя файла справочника, который используется программой для отображения оперативной справочной информации по формам и компонентам.

HelpSystem — интерфейс к справочной системе.

Icon — содержит значок, отображаемый на Панели Задач во время работы программы. Значок отображается слева от названия (см. **Title**).

UpdateFormatSettings — включает автоматическое обновление форматных строк в программе вслед за изменением этих параметров в операционной системе. Форматные строки управляют показом даты, времени, денежных единиц и др.

UpdateMetricSettings — включает автоматическое обновление шрифта и фона системных надписей (всплывающих подсказок и подписей значков) при изменении настроек экрана в операционной системе.

Terminated — значение True говорит о том, что программа находится в процессе завершения.

Если вы желаете задать заголовок (свойство **Title**), значок (свойство **Icon**) и имя файла справочника (свойство **HelpFile**) для приложения, не корректируйте главный программный файл, а обратитесь лучше к диалоговому окну **Project Options** (рисунок 7.45), которое появляется по команде меню **Project | Options...** .

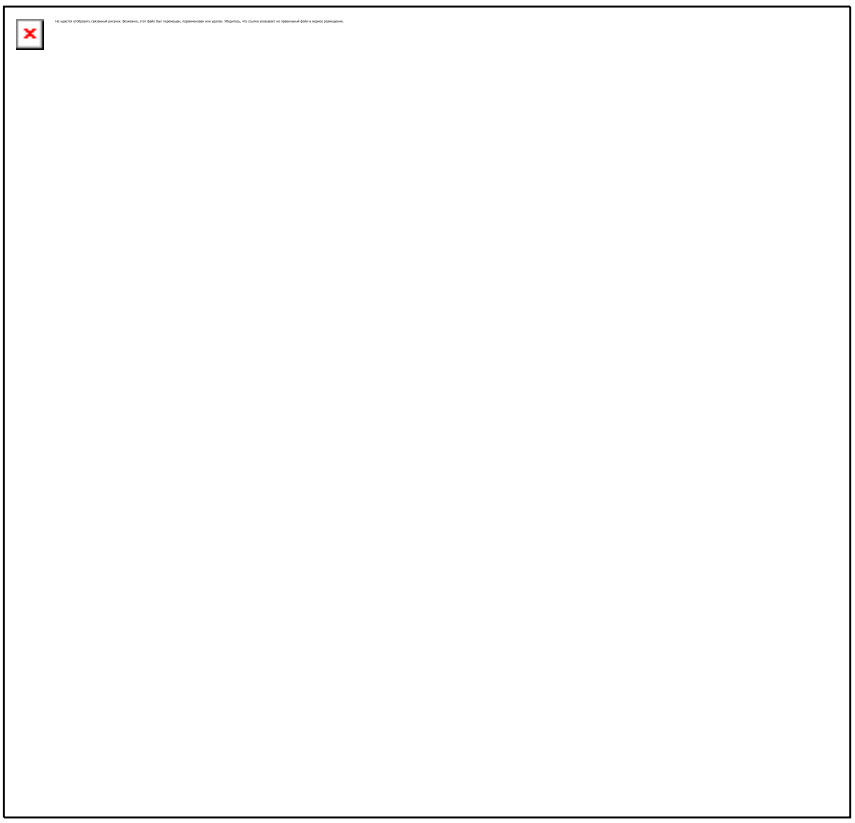


Рисунок 7.45. Окно параметров проекта

Объект **Application** имеет несколько полезных событий. Самые важные из них: **OnActivate**, **OnDeactivate**, **OnException**.

OnActionExecute — происходит при выполнении любой команды в компоненте **ActionList** (см. главу 10).

OnActionUpdate — происходит во время простоя программы для обновления состояния команд в компоненте **ActionList** (см. главу 10).

OnActivate — происходит, когда программа получает фокус ввода, т.е. когда пользователь переключается на нее с другой программы.

OnDeactivate — происходит, когда программа теряет фокус ввода, т.е. когда пользователь переключается с нее на другую программу.

OnException — происходит, когда в программе возникает необработанная исключительная ситуация. Стандартный обработчик этого события вызывает метод **ShowException** для отображения окна сообщений с пояснением причины ошибки. Вы можете изменить реакцию на событие OnException, переписав его обработчик.

OnHelp — происходит, когда пользователь вызывает справку.

OnHint — происходит, когда курсор мыши наводится на компонент, содержащий всплывающую подсказку.

OnIdle — периодически происходит во время простоя программы.

OnMessage — происходит при получении программой сообщения операционной системы Windows.

OnMinimize — происходит, когда пользователь сворачивает программу.

OnModalBegin — происходит при отображении монопольной формы на экране.

OnModalEnd — происходит при закрытии монопольной формы.

OnRestore — происходит, когда пользователь восстанавливает свернутую программу.

OnSettingChange — происходит при изменении настроек операционной системы, например, настроек экрана или региональных настроек.

OnShortCut — происходит при нажатии клавиш на клавиатуре (еще до того, как в форме происходит событие **OnKeyDown**).

OnShowHint — происходит непосредственно перед появлением любой всплывающей подсказки.

Из всех методов объекта **Application** мы упомянем лишь один — **Terminate**. Он выполняет штатное завершение приложения. Запомните, метод **Terminate** не вызывает немедленного завершения приложения, давая возможность всем формам корректно себя закрыть. Во время закрытия форм свойство **Terminated** имеет значение True.

При необходимости на помощь объекту **Application** спешат менее значительные “персоны”: объекты **Screen**, **Printer** и **Clipboard**. Они также являются глобальными и создаются автоматически при запуске приложения (если, конечно, подключены стандартные модули, где они расположены).

7.7.2. Screen — объект, управляющий экраном

Каждая программа что-то выводит на экран, иначе она просто бесполезна. В среде Delphi экран трактуется как глобальный объект **Screen** класса TScreen, имеющий набор свойств. Многие из них жестко связаны с физическими характеристиками экрана (с “железом“), поэтому в большинстве случаев не доступны для записи. Обозначим самые важные свойства:

Width и **Height** — ширина и высота экрана в пикселях.

ActiveForm — активная форма (та, которая в текущий момент находится в фокусе ввода).

ActiveControl — указывает компонент, который обладает фокусом ввода в активной форме.

Cursor — управляет внешним видом указателя мыши для всех форм приложения.

Cursors — список доступных указателей мыши.

DataModuleCount — количество модулей данных, созданных приложением. Модуль данных — это нечто вроде невидимой формы, в которой можно размещать исключительно невизуальные компоненты. Перемещение невизуальных компонентов из формы в модуль данных может в ряде случаев улучшить структуризацию программы за счет отделения предметной программной логики от программной логики пользовательского интерфейса.

DataModules — список всех модулей данных, созданных приложением.

DesktopWidth и **DesktopHeight** — ширина и высота виртуального экрана (используется, когда к компьютеру подключено несколько мониторов).

DesktopLeft и **DesktopTop** — позиция виртуального экрана на экране монитора.

DesktopRect — координаты виртуального экрана.

Fonts — список всех поддерживаемых шрифтов.

FormCount — количество форм, созданных приложением.

Forms — список всех форм, созданных приложением.

HintFont — шрифт всплывающих подсказок.

IconFont — шрифт подписей к значкам.

MenuFont — шрифт текста в меню.

MonitorCount — количество мониторов, подключенных к компьютеру.

Monitors — список всех мониторов, подключенных к компьютеру.

PixelsPerInch — количество пикселей в одном дюйме экрана монитора.

WorkAreaWidth и **WorkAreaHeight** — ширина и высота рабочей области экрана (не включает панель задач). Если к компьютеру подключено несколько мониторов, то рассчитывается ширина и высота рабочей области на основном мониторе.

WorkAreaLeft и **WorkAreaTop** — позиция рабочей области на экране монитора.

WorkAreaRect — размеры и позиция рабочей области на экране монитора.

В качестве примера использования объекта **Screen** приведем фрагмент, устанавливающий указателю мыши вид песочных часов на время выполнения какой-либо длительной операции:

```
Screen.Cursor := crHourGlass;
try
  { Длительная операция }
finally
  Screen.Cursor := crDefault;
end;
```

7.7.3. Mouse — объект, представляющий мышь

Вряд ли сейчас можно встретить компьютеры без миниатюрного “хвостатого” устройства, называемого мышью. Для работы с ним в среде Delphi есть специальный объект **Mouse**: **TMouse**, автоматически добавляемый в программу при подключении модуля **Controls**. Перечислим наиболее важные свойства этого объекта:

Capture — содержит описатель окна, захватившего мышь для монопольного использования (это объект операционной системы Windows).

CursorPos — позиция указателя мыши.

DragImmediate — определяет, когда начинается буксировка: значение True — немедленно, значение False — после того, как указатель мыши переместиться на **DragThreshold** позиций при удерживаемой кнопке мыши.

DragThreshold — количество пикселей, на которые необходимо переместить указатель при нажатой кнопке мыши, чтобы началась буксировка.

IsDragging — проверяет, идет ли в данный момент процесс буксировки.

MousePresent — проверяет, подключена ли мышь к компьютеру.

WheelPresent — проверяет, есть ли у мыши колесико.

WheelScrollLines — количество логических строк, на которые смещается страница при прокрутке колесика мыши на один шаг.

7.7.4. Printer — объект, управляющий принтером

Большинство программ выводят некоторый текст или рисунки на печатающее устройство. Для этого полезного дела в среде Delphi имеется специальный объект **Printer**. Он становится доступен после подключения модуля Printers. Если вы включите этот модуль в проект, сразу после старта будет порожден объект **Printer** класса TPrinter. Его свойства и методы дают вам весьма неплохие возможности для печати из приложения на все виды принтеров. Однако, тема эта заслуживает отдельной главы (см. гл. 10).

7.7.5. Clipboard — объект, управляющий Буфером обмена

Каждый, кто работал с текстом, знает, какая это великолепная штука — *Буфер обмена* (Clipboard). Напомним, что это буфер, куда можно что-то положить (например, текст или рисунок), а потом взять это оттуда. За операции с Буфером обмена в среде Delphi отвечает

глобальный объект **Clipboard** класса TClipboard. Он расположен в модуле Clipbrd. О том, как объект **Clipboard** используется практически, подробно рассказано в гл. 8.

7.8. Итоги

Время потрачено не зря! Вы узнали о проекте все:

- что он собой представляет и из каких частей состоит (файлы описания форм, файлы программных модулей, главный файл проекта и др.);
- как открывать, сохранять, выполнять проект и управлять им с помощью окна **Project Manager**;
- что есть форма, как изменять ее стиль, размер, местоположение, цвет, как переключаться с главной формы на второстепенную и наоборот и т.д.
- что есть компонент, откуда его взять, куда поместить, как навести порядок в группе компонентов;
- кто управляет приложением изнутри (объект Application) и кто ему в этом помогает (объекты Screen, Printer, Clipboard).

Да, трудновато все это было усвоить, но надо. Тяжело в учении — легко в бою. Утешив себя этой истиной, перейдем к изучению важнейших элементов пользовательского интерфейса — меню, панели инструментов и строки состояния.

Глава 8. Меню, строка состояния и панель инструментов

Практически любому графическому приложению необходимо иметь меню, строку состояния и панель инструментов. Эти стандартные элементы пользовательского интерфейса приходится создавать каждому, кто решает с помощью компьютера любую более или менее серьезную проблему. Мы решили совместить приятное с полезным и рассмотреть технологию создания меню и других жизненно важных элементов программы в процессе создания простого, но очень наглядного приложения. Оно предназначено для просмотра стандартных графических файлов (например, точечных и векторных рисунков, значков).

8.1. Меню

8.1.1. Идея меню

Важнейшим элементом пользовательского интерфейса является меню. Оно очень похоже на список блюд, который вы не раз видели в ресторане. Отличие только одно — там его подает официант, а здесь оно само появляется на экране вашего компьютера после старта практически любого приложения. Короче говоря, *меню* — это список возможностей, которые программа предоставляет в распоряжение пользователя и которые он может выбирать по своему желанию. Выбор пунктов меню осуществляется с помощью мыши или клавиатуры.

Различают два типа меню:

- главное меню формы;
- контекстное меню формы или компонента.

Главное меню всегда одно и располагается под заголовком формы. Выбор одного из пунктов главного меню вызывает появление на экране подчиненного меню со списком вложенных пунктов. Любой пункт подчиненного меню может быть либо командой, либо содержать другое подчиненное меню, о чем свидетельствует стрелка справа от пункта. Уровень

вложенности подчиненных меню практически не ограничен, но современное представление о хорошем пользовательском интерфейсе требует, чтобы вложенность была минимальной.

Контекстных меню может быть много и они не имеют постоянного места внутри формы. Такие меню не связаны с главным меню и появляются лишь по специальному требованию со стороны пользователя, как правило, по щелчку правой кнопкой мыши, когда указатель мыши наведен на нужный элемент. Пункты контекстного меню могут содержать подчиненные меню. Контекстное меню привязывается к конкретному элементу формы и идеально подходит для размещения команд, специфичных только этому элементу. Поскольку доступ к командам контекстного меню можно получить быстрее, чем к командам главного меню, использование контекстных меню делает пользовательский интерфейс более удобным.

Для создания главного и контекстного меню среда Delphi имеет два разных компонента: **MainMenu** и **PopupMenu**. Заполнение этих компонентов пунктами меню происходит одинаково, но результат будет разным. В первом случае мы получим стандартную строку главного меню, а во втором — окно контекстного меню.

8.1.2. Главное меню

Шаг 1. Приступая к практической работе, создайте новое приложение. Как это сделать, вы уже знаете. Поскольку в качестве примера мы решили разработать приложение для просмотра графических файлов, давайте назовем форму **PictureForm** (значение свойства **Name**), и дадим ей заголовок **Picture Viewer** (значение свойства **Caption**).

Шаг 2. Теперь сохраните модуль формы и весь проект, выполнив команду меню **File | Save All**. Модуль назовите **MainUnit.pas**, а файл проекта — **PictureViewer.dpr**. Вот теперь можно приступать к изучению и меню и всего остального.

Отображение в форме *главного меню (main menu)* обеспечивает компонент **MainMenu**, расположенный в палитре компонентов на вкладке **Standard** (рисунок 8.1). Поместите этот компонент на форму и дайте ему имя **MainMenu** (значение свойства **Name**).

Рисунок 8.1. Компонент MainMenu

Компонент **MainMenu** имеет небогатый набор свойств, подробно мы на них останавливаться не будем, а обозначим лишь самые важные (таблица 8.1):

Свойство	Описание
AutoHotKeys	Значение **maAutomatic** избавляет программиста от необходимости назначать пунктам меню "горячие" клавиши (с помощью специального символа **&** в тексте пунктов); компонент автоматически подбирает "горячие" клавиши. Значение **maManual** требует, чтобы "горячие" клавиши назначил программист (см. параграф 8.1.3).
AutoLineReduction	Если равно значению **maAutomatic**, то при отображении меню идущие подряд пункты-разделители рисуются как один разделитель, а пункты-разделители, находящиеся в начале или конце меню вообще не показываются. Свойство AutoLineReduction

	применяется при программном добавлении и удалении пунктов меню, чтобы избежать нежелательных явлений вроде повторяющихся и повисших разделительных линий. Если свойство AutoLineReduction равно значению **maManual**, то все пункты меню рисуются как есть.
AutoMerge	Определяет, сливается ли главное меню вторичной формы с главным меню главной формы. Способ слияния определяется значением свойства GroupIndex каждого пункта меню верхнего уровня.
Images	Список значков, отображаемых рядом с пунктами меню. Свойство **Images** используется совместно со свойством **ImageIndex** компонентов **MenuItem** (см. параграф 8.1.12).
Items	Массив пунктов меню.
OwnerDraw	Если равно значению True, то каждый пункт меню получает возможность участвовать в процессе своего отображения при помощи специальных событий **OnMeasureItem** и **OnDrawItem**. Событие **OnMeasureItem** происходит в пункте меню, когда рассчитываются размеры пункта. Событие **OnDrawItem** происходит в пункте меню, когда пункт рисуется на экране. Если свойство **OwnerDraw** равно значению False, то пункты меню имеют стандартный вид и события **OnMeasureItem** и **OnDrawItem** не происходят.
OnChange	Происходит при изменении структуры меню.

*Таблица 8.1. Важнейшие свойства и события компонента **MainMenu***

Значок компонента **MainMenu**, который вы видите на форме, отображается лишь на этапе разработки. Он нужен для того, чтобы вы могли быстро активизировать компонент и перейти к установке его свойств. Однако компонент **MainMenu** является невизуальным и на этапе выполнения приложения его значок не отображается. Пользователь видит результат работы компонента — строку меню.

Пока в меню нет пунктов, нет и самого меню. Добавление новых пунктов выполняется в специальном окне — *дизайнере меню* (**Menu Designer**).

8.1.3. Дизайнер меню

Вызов дизайнера меню осуществляется с помощью команды **Menu Designer…**, которая находится в контекстном меню компонента **MainMenu** (рисунок 8.2).

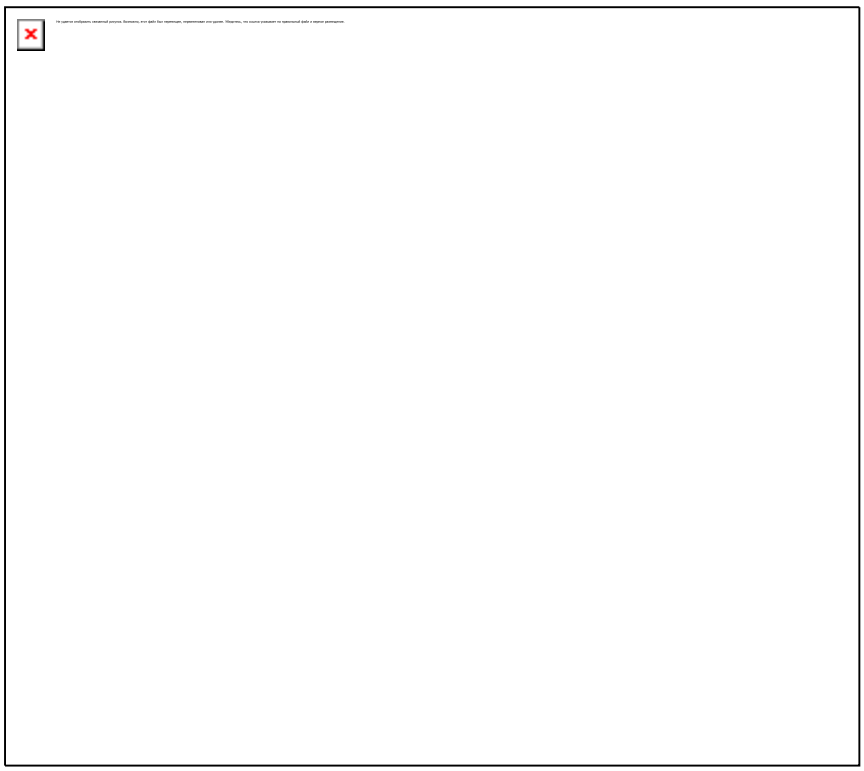


Рисунок 8.2. Вызов дизайнера меню (Menu Designer)

Шаг 3. Выберите показанную на рисунке команду **Menu Designer…** и на экране появится окно с заголовком **PictureForm.MainMenu**. Это и есть дизайнер меню.

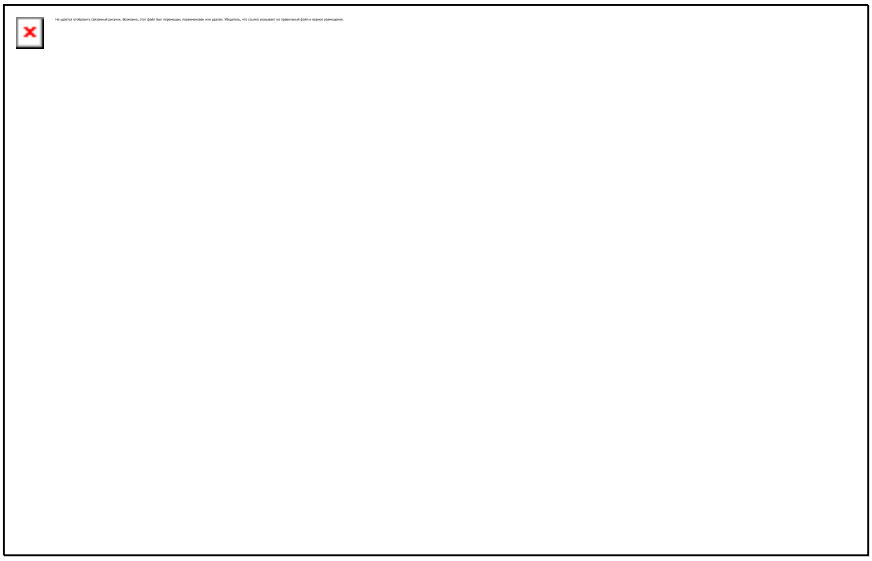


Рисунок 8.3. Дизайнер меню (Menu Designer)

Дизайнер меню работает в паре с окном свойств. Создание и удаление пунктов осуществляется в дизайнере меню, а свойства отдельно взятого пункта устанавливаются в окне свойств.

Шаг 4. Сейчас строка главного меню состоит из одного безымянного пункта. Дайте этому пункту программный идентификатор **FileMenuItem** (значение свойства **Name**) и заголовок **&File** (значение свойства **Caption**). Символ **&** обеспечивает подчеркивание следующего за ним символа при отображении текста, поэтому пункт меню будет виден как **File** (рисунок 8.4). Подчеркнутая буква используется в комбинации с клавишей **Alt** для быстрого выбора пункта меню и называется *горячей клавишей*. В данном случае активизация пункта **File** будет

происходить по комбинации клавиш **Alt**+**F**. Заметим, что в некоторых версиях операционной системы Windows «горячие» клавиши подчеркиваются только после нажатия клавиши Alt.

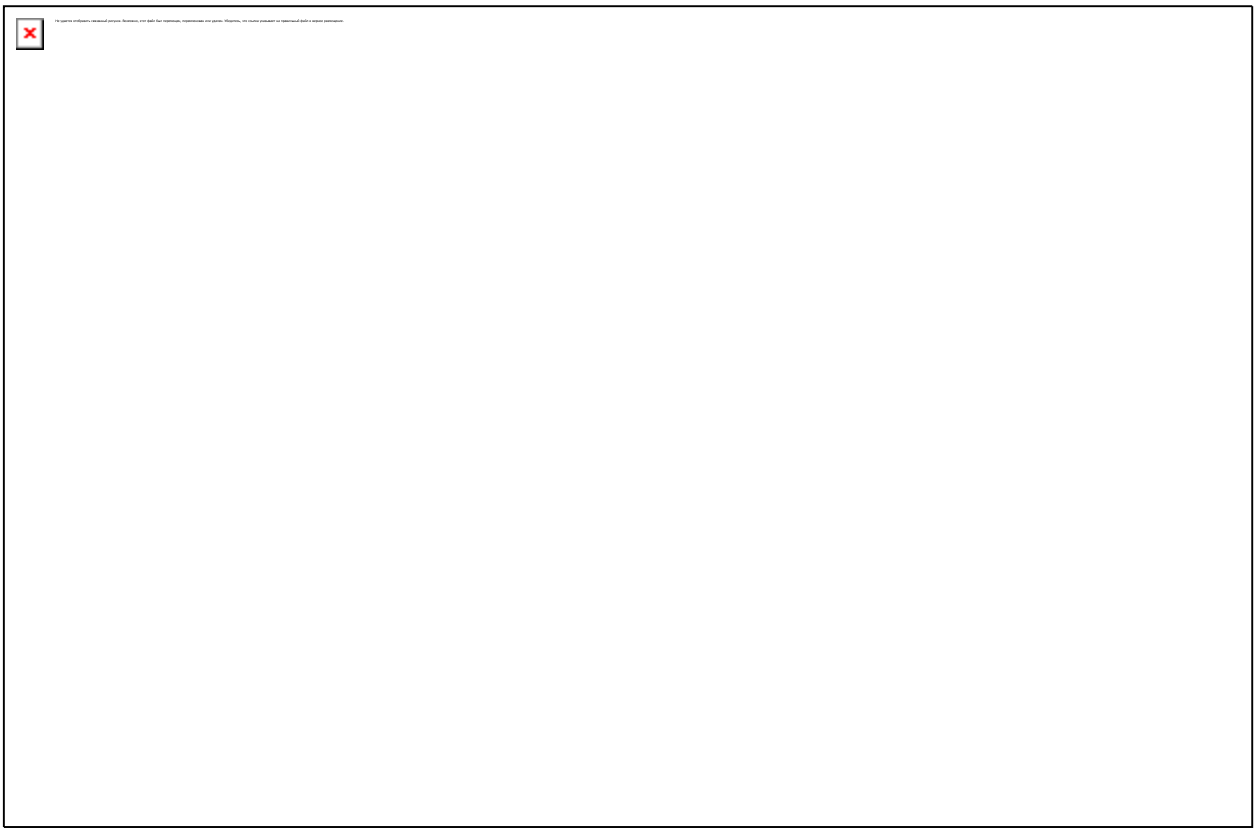


Рисунок 8.4. Текст пункта меню

Бывает очень утомительно назначать пунктам меню горячие клавиши. К тому же приходится заботиться о том, чтобы горячие клавиши не дублировались в нескольких пунктах. К счастью в компоненте **MainMenu** существует свойство **AutoHotKeys**. Если оно установлено в значение **maAutomatic**, то подбор горячих клавиш выполняется автоматически. С этого момента мы будем пользоваться этой возможностью, поэтому удалите символ амперсанта (&) из свойства **Caption** компонента **FileMenuItem** и убедитесь, что в компоненте **MainMenu** свойство **AutoHotKeys** установлено в значение **maAutomatic**.

Шаг 5. Сейчас под пунктом **File** нужно создать подчиненное меню со списком команд. Для этого просто щелкните в дизайнере меню на пункте **File**, среда Delphi все сделает за вас. Под пунктом **File** появится пустая ячейка — заготовка первого пункта выпадающего списка. Выберите этот пункт с помощью мыши и дайте ему программный идентификатор **OpenMenuItem** (свойство **Name**), а в свойстве **Caption** впишите текст **Open...** . Вместо пустой ячейки появится текст **Open...** и пустая ячейка переместится ниже.

Шаг 6. Действуя по аналогии, добавьте еще три пункта: **Save As...** , **Close** и **Exit**. В программе они должны называться **SaveAsMenuItem**, **CloseMenuItem** и **ExitMenuItem** соответственно.

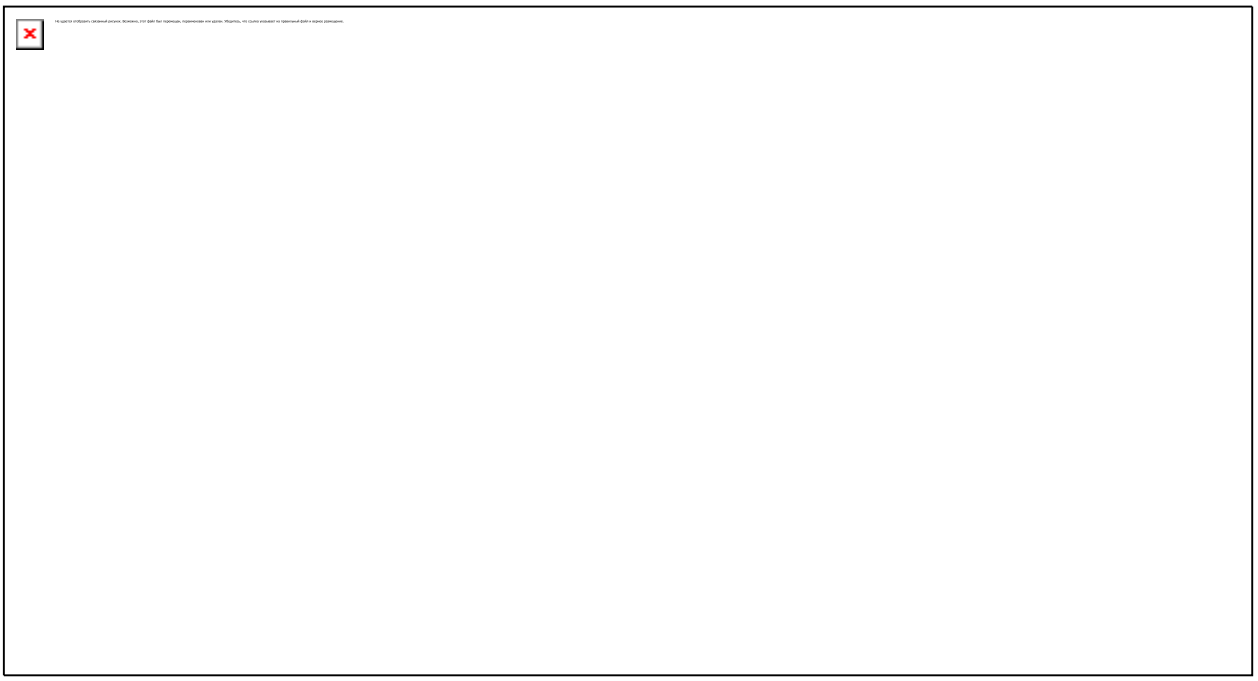


Рисунок 8.5. Пункты в меню File

Внимание! Не пытайтесь удалить пустой пункт, завершающий список команд — у вас ничего не выйдет. Да это и не требуется, поскольку пустые висячие пункты не отображаются в меню во время работы программы.

Согласитесь, что добавление новых пунктов сделано в среде Delphi очень удобно. Но для создания полноценного меню, одной этой возможности явно недостаточно — нужны средства вставки и удаления пунктов, создания вложенных меню и прочие. Поэтому в дизайнере меню для каждого отдельно взятого пункта предусмотрено контекстное меню с необходимым набором команд (рисунок 8.6 и таблица 8.2).

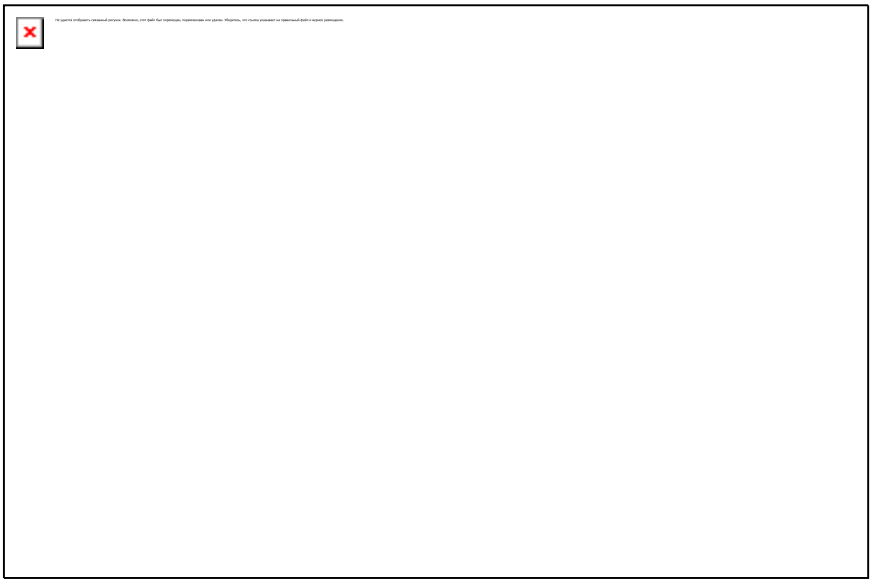


Рисунок 8.6. Контекстные команды в дизайнере меню

Команда	Описание
Insert	Вставляет новый пункт.
Delete	Удаляет выбранный пункт.

Create Submenu	Создает в позиции пункта подчиненное меню.
Select Menu	Предлагает выбрать для работы другой компонент меню.
Save As Template	Сохраняет текущую структуру меню в списке шаблонов.
Insert From Template	Вставляет меню из списка шаблонов.
Delete Templates	Удаляет шаблон(ы) меню.
Insert From Resource	Вставляет меню из файла с описанием меню (расширение MNU) или из стандартного файла ресурсов (расширение RC).

Таблица 8.2. Контекстные команды в дизайнере меню

Бывает, создав меню, вы вдруг обнаруживаете, что какой-то незадачливый пункт находится не на своем месте. Проблема решается просто: захватите пункт меню щелчком левой кнопки мыши и, удерживая нажатой кнопку мыши, отбуксируйте его к новой позиции. Таким образом, можно переместить не только отдельный пункт, но и целое подчиненное меню со всеми его пунктами и вложенными меню.

8.1.4. Пункты меню

Нетрудно догадаться, что пункты меню, как и все элементы интерфейса, являются компонентами. Класс пункта меню называется **TMenuItem**, самые характерные его свойства обозначены в таблице 8.3.

Свойство	Описание
Action	Задает так называемую команду, которая будет выполняться при выборе данного пунтка меню. Весь список команд содержится в компоненте **ActionList** (см. параграф 8.6).
AutoCheck	Если равно значению True, то выбор пункта меню автоматически приводит к изменению значения свойства Checked на противоположное. Если равно значению False, то изменением свойства Checked управляет программист.
AutoHotkeys	Значение **maAutomatic** избавляет программиста от необходимости назначать пункту меню "горячую" клавишу (с помощью специального символа **&** в тексте пункта); компонент автоматически подбирает "горячую" клавишу. Значение **maManual** требует, чтобы "горячую" клавишу назначил программист (см. параграф 8.1.3). Значение **maParent** показывает, что способ назначения горячей клавиши определяется "родительским" компонентом **MainMenu**.

AutoLineReduction	Если равно значению **maAutomatic**, то при отображении меню подряд идущие пункты-разделители рисуются как один разделитель, а пункты-разделители, находящиеся в начале или конце меню вообще не показываются. Свойство AutoLineReduction применяется при программном добавлении и удалении пунктов меню, чтобы избежать нежелательных явлений вроде повторяющихся и повисших разделительных линий. Если свойство AutoLineReduction равно значению **maManual**, то все пункты меню отображаются как есть. Если свойство равно значению **maParent**, то способ определяется “родительским” компонентом (например, **MainMenu**).
Bitmap	Значок, который отображается рядом с текстом пункта меню. Если для данного пункта меню указан индекс значка с помощью свойство **ImageIndex**, то значение свойства **Bitmap** игнорируется. Значки в пунктах меню более подробно расмотрены в параграфе 8.1.12.
Break	Если равно **mbBreak** или **mbBarBreak**, то пункт меню начинает новый столбец. Значение mbBarBreak обеспечивает отделение нового столбца от предыдущего вертикальной чертой.
Caption	Текст пункта меню.
Checked	Если равно значению True, то пункт меню содержит метку в виде "птички".
Default	Значение True говорит о том, что выбор пункта меню можно осуществить двойным щелчком "родительского" пункта меню.
Enabled	Определяет, доступен ли пользователю данный пункт меню.
GroupIndex	Работает по-разному в зависимости от того, находится пункт в подчиненном меню или в строке главного меню. Пункты подчиненного меню с одинаковым положительным значением GroupIndex согласовано переключают между собой метку — установка у одного пункта свойства **Checked** в значение **True** снимает метку с другого пункта. В MDI-формах свойство **GroupIndex** работает по-другому. Пункты главного меню, находящиеся в дочерней форме MDI, сливаются с пунктами главного меню обрамляющей формы MDI при активизации дочерней формы. При этом если в строке главного меню обрамляющей формы

	существуют пункты с таким же значением свойства **GroupIndex**, то новый пункт со своим списком пунктов полностью заменяет старый; в противном случае новый пункт со своим списком пунктов вставляется в строку главного меню. Более подробно слияние меню изложено в справочнике среды Delphi.
Hint	Краткая подсказка для пользователя, отображаемая в строке состояния.
ImageIndex	Номер значка в списке **Images** компонента **MainMenu**. Значок отображается рядом с текстом пункта меню (см. параграф 8.1.12). Отрицательное значение свойства **ImageIndex** говорит о том, что для пункта меню значок не задан. Свойство **ImageIndex** имеет приоритет над свойством **Bitmap**.
RadioItem	Если равно значению True, то метка имеет вид жирной точки.
ShortCut	Комбинация клавиш для выполнения команды, не открывая меню.
SubMenuImages	Список значков, отображаемых рядом с пунктами подчиненного меню. Свойство **SubMenuImages** используется совместно со свойством **ImageIndex** компонентов **MenuItem** (см. параграф 8.1.12).
Visible	Определяет, виден ли пользователю пункт меню.
OnAdvancedDrawItem	Происходит при рисовании отдельно взятого пункта меню на экране. Событие происходит только в том случае, если соответствующий компонент меню (**MainMenu** или **PopupMenu**) содержит значение True в свойстве **OwnerDraw**. Предоставляет более широкие возможности по сравнению с событием **OnDrawItem**.
OnClick	Происходит при выборе пункта меню пользователем.
OnDrawItem	Происходит при рисовании отдельно взятого пункта меню на экране. Событие происходит только в том случае, если соответствующий компонент меню (**MainMenu** или **PopupMenu**) содержит значение True в свойстве **OwnerDraw**.
OnMeasureItem	Происходит при расчете размеров отдельно взятого пункта меню перед его рисованием на экране. Событие происходит только в том случае, если соответствующий компонент меню (**MainMenu** или **PopupMenu**) содержит значение True в свойстве

OwnerDraw.

*Таблица 8.3. Важнейшие свойства и события компонента **MenuItem***

По аналогии с остальными классами компонентов можно было бы предположить, что в палитре компонентов существует компонент **MenuItem**. Однако его там нет, поскольку пункты меню не существуют сами по себе, а работают только в составе строки главного меню или окна контекстного меню. Тем не менее, они во многом ведут себя как настоящие компоненты, например, настраиваются в окне свойств и наряду с остальными компонентами помещаются в исходный текст формы в виде отдельных полей. Чтобы в этом убедиться, активизируйте редактор кода и найдите определение класса формы. Оно будет таким, как на рисунке 8.7.

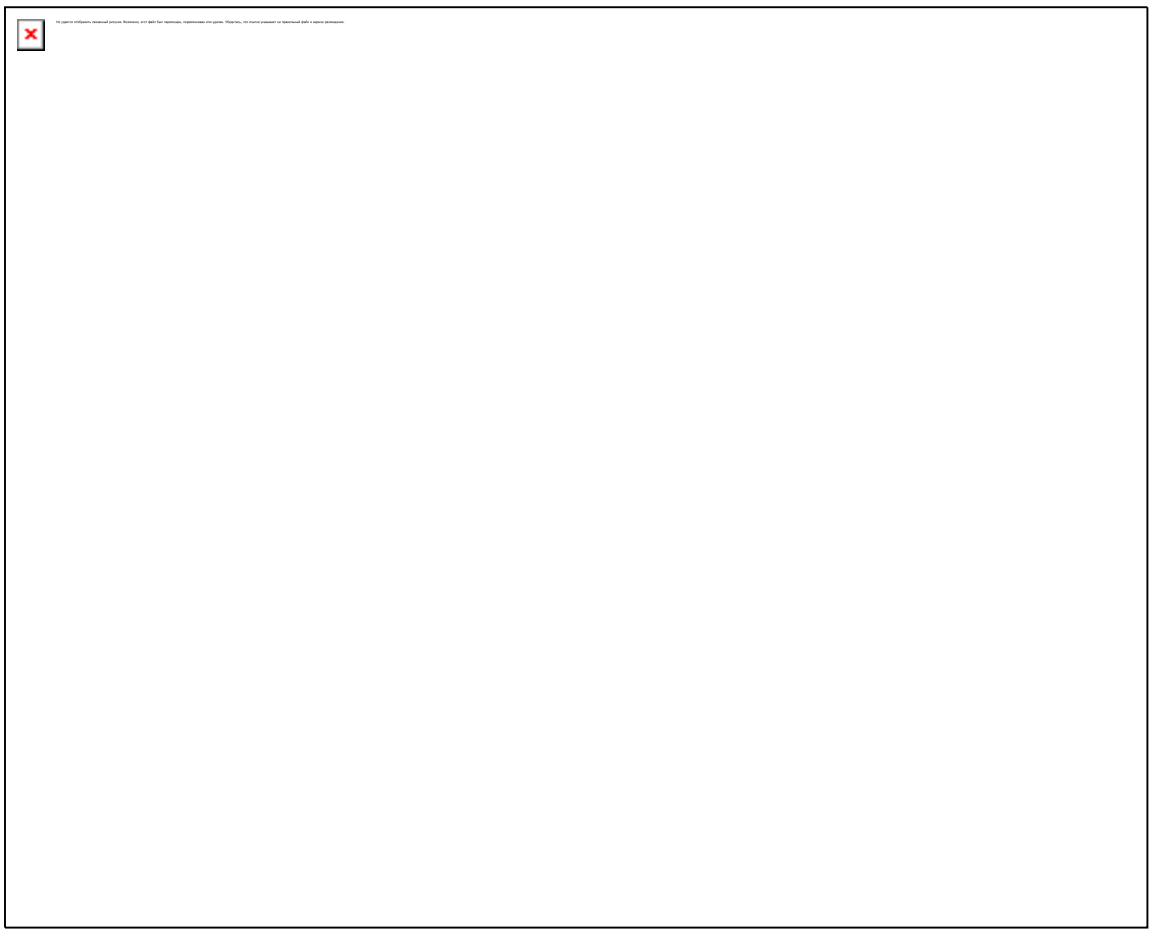


Рисунок 8.7. Пункты меню в программном коде

8.1.5. Разделительные линии

Шаг 7. Логически связанные между собой команды принято отделять от других команд горизонтальной линией. Например, пункт **Exit** хорошо бы отделить от остальных (рисунок 8.8). Для этого вставьте новый пункт и запишите в значении свойства **Caption** символ минуса (–).

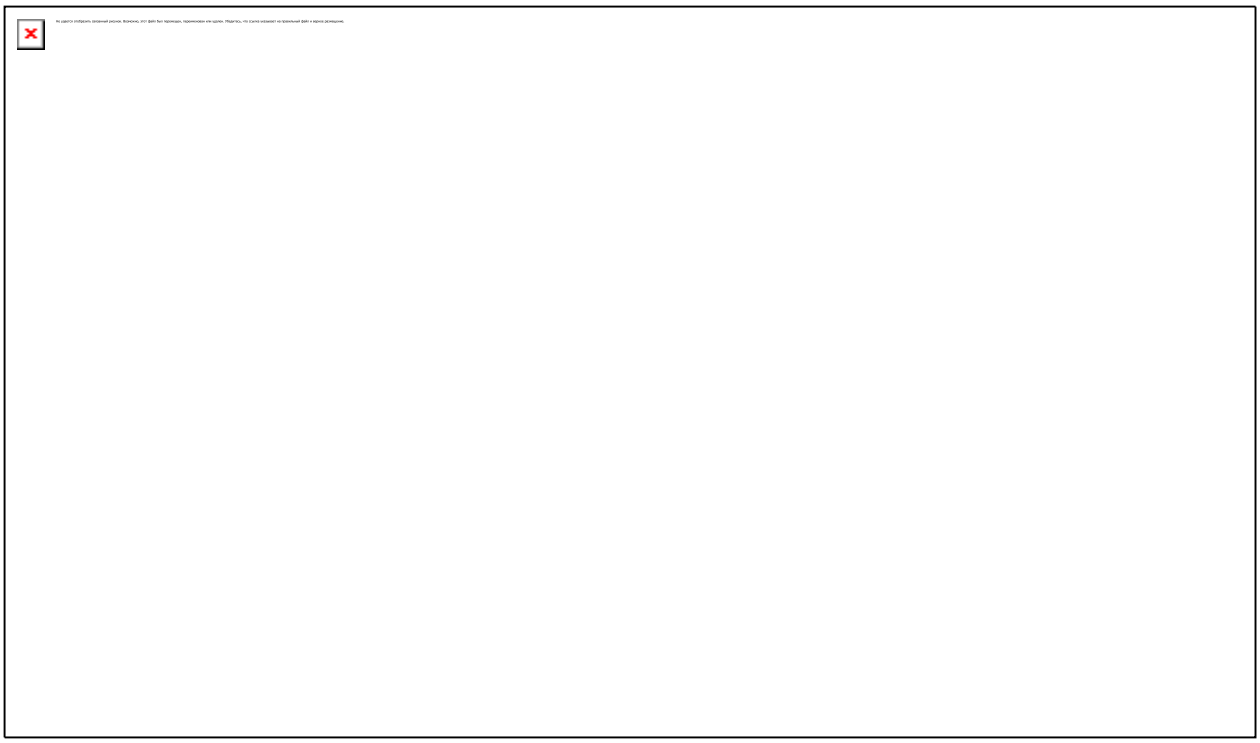


Рисунок 8.8. Разделительная линия в меню

Среда Delphi знает, что одиночный символ минуса в имени пункта меню означает разделитель и нарисует для пункта горизонтальную линию. Кстати, это не запрещает вам создавать пункты, имена которых начинаются со знака минус. Если вы запишите что-нибудь после знака минуса, то в имени пункта отобразится весь введенный текст.

8.1.6. Комбинации клавиш

Некоторым пунктам меню назначают *комбинации клавиш* (shortcut), чтобы выполнять команды, не открывая меню. Они ускоряют работу с приложением и популярны среди опытных пользователей. Названия комбинаций клавиш отображаются справа от текста соответствующих пунктов. Например, во многих программах команде меню **File | Open...** назначается комбинация клавиш **Ctrl+O**.

Шаг 8. Чтобы назначить пункту комбинацию клавиш, активизируйте пункт в дизайнере меню, перейдите к окну свойств и выберите в списке значений свойства **ShortCut** требуемую комбинацию клавиш (рисунок 8.9). Если ее там нет, то введите название комбинации клавиш вручную.

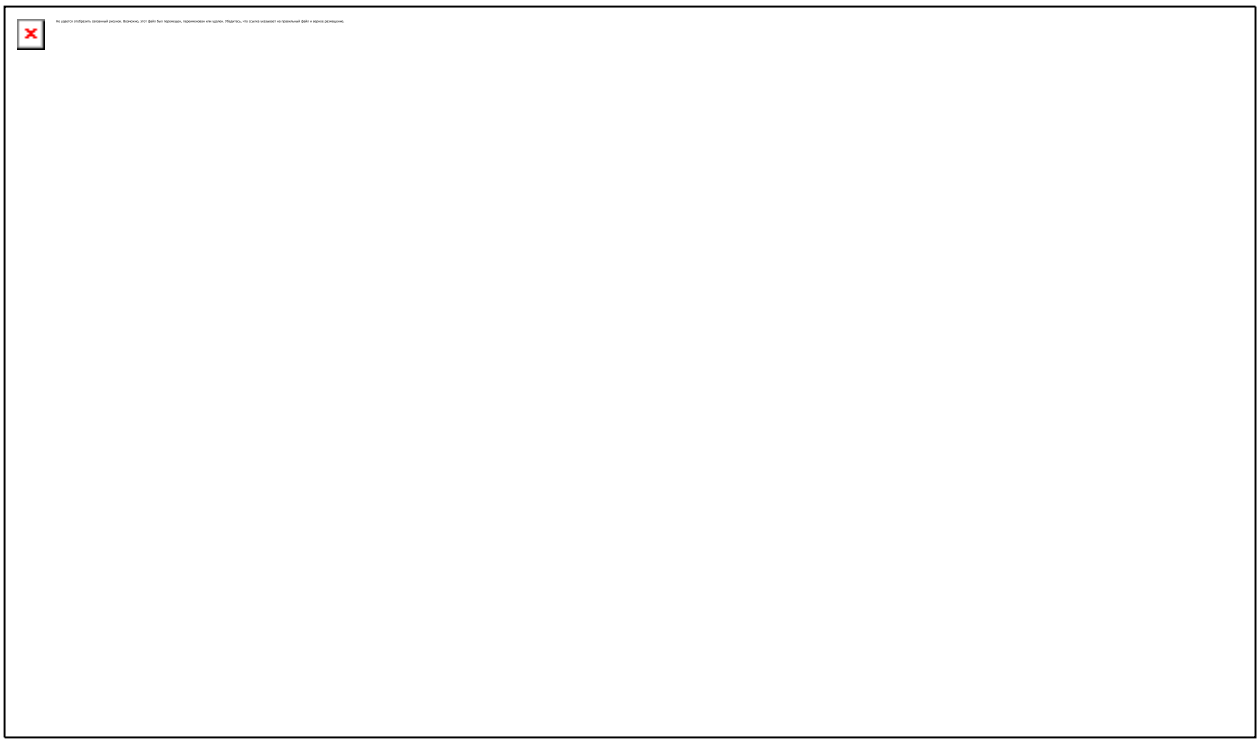


Рисунок 8.9. Комбинация клавиш для активизации пункта меню

Внимание! Имейте в виду, что среда Delphi не отслеживает дублирование одной и той же комбинации клавиш для нескольких пунктов меню, за это отвечает программист.

8.1.7. Обработка команд меню

В первом приближении меню готово и вам наверняка не терпится его опробовать. Давайте реализуем закрытие формы по команде **Exit**. Решение этой задачи сводится к обработке события **OnClick** компонента **ExitMenuItem**. Это событие возникает при выборе пользователем в меню пункта **Exit**.

Шаг 9. Итак, активизируйте в дизайнере меню пункт **Exit** и выберите в окне свойств вкладку **Events**. Теперь сделайте двойной щелчок мышью на значении события **OnClick** (рисунок 8.10).

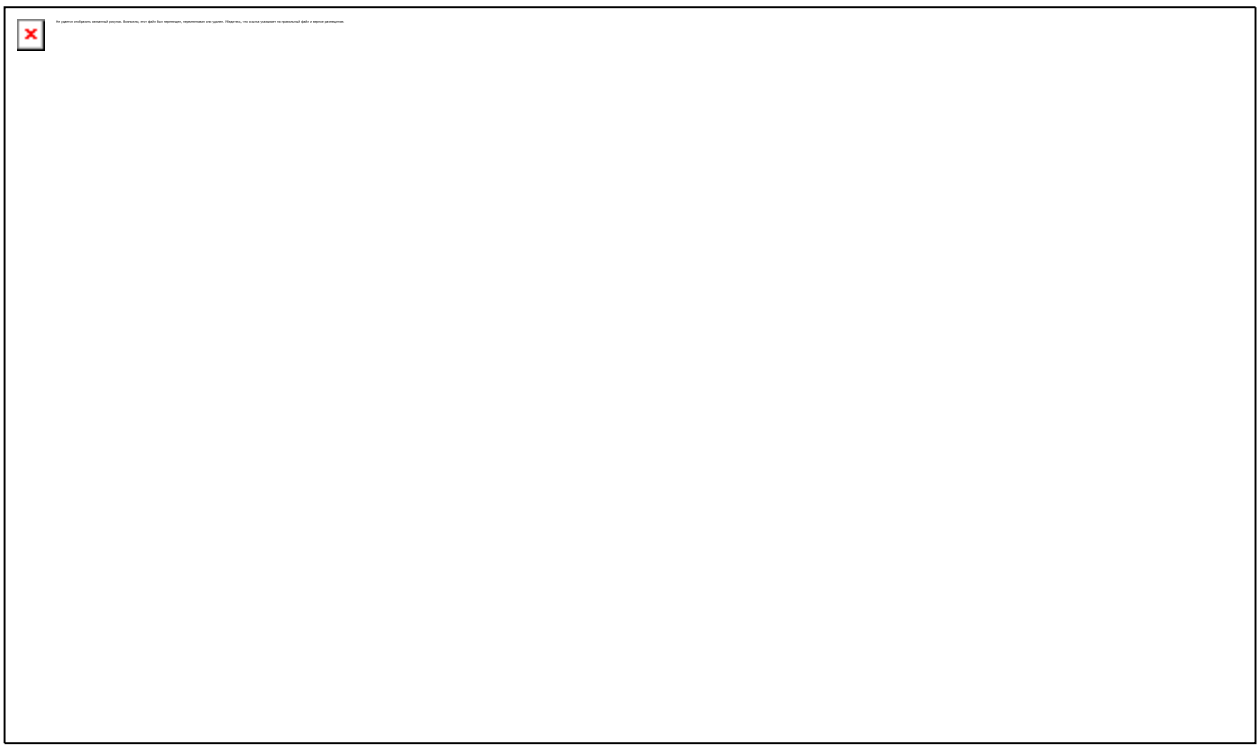


Рисунок 8.10. Создание обработчика команды меню

В результате откроется редактор кода, в котором появится заготовка обработчика события. Обработка команды **Exit** сводится к вызову метода **Close**, закрывающего форму (а заодно и приложение, поскольку это единственная форма):

```
procedure TPictureForm.ExitMenuItemClick(Sender: TObject);
begin
  Close;
end;
```

Подключение меню к форме выполняется с помощью свойства формы **Menu**. Отыскав его в окне свойств, вы обнаружите, что оно уже содержит идентификатор разработанного меню **MainMenu**, поэтому в данном случае для работы меню больше ничего не нужно.

Проверим, работает ли меню. Выполните компиляцию и запустите проект. На экране появится форма со строкой меню под заголовком. Выбор в меню любой команды кроме **Exit** безрезультатен. По команде **Exit** окно закроется и приложение завершится (рисунок 8.11).

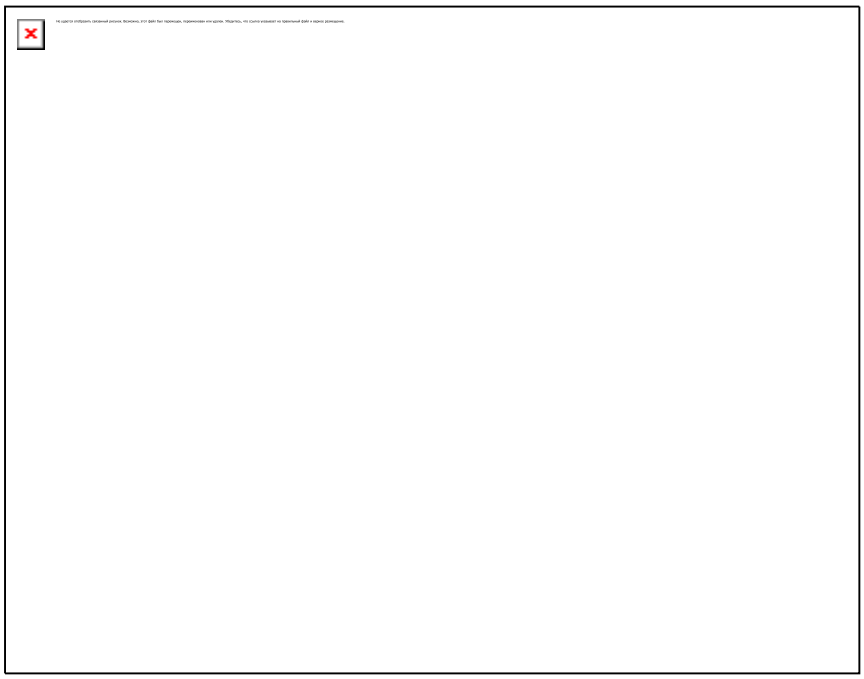


Рисунок 8.11. Проверка работы команды Exit

8.1.8. Пункты-переключатели

Во многих программах существуют пункты меню, которые работают как переключатели. Если вы еще не сообразили, о чем идет речь, посмотрите на рисунок 8.12. В нашей программе переключателями удобно сделать пункты меню, отвечающие за отображение панели инструментов и строки состояния. Установка флажка щелчком пункта-переключателя показывает панель инструментов (или строку состояния), а снятие флажка — прячет. Рассмотрим, как программируется такое поведение.

Шаг 10. В строке главного меню создайте выпадающее меню **View** с пунктами **Toolbar** (программное имя **ToolBarMenuItem**) и **Status bar** (программное имя **StatusBarMenuItem**). Установите в последних двух пунктах свойство **Checked** в значение **True**. В пунктах меню появятся метки (рисунок 8.12).

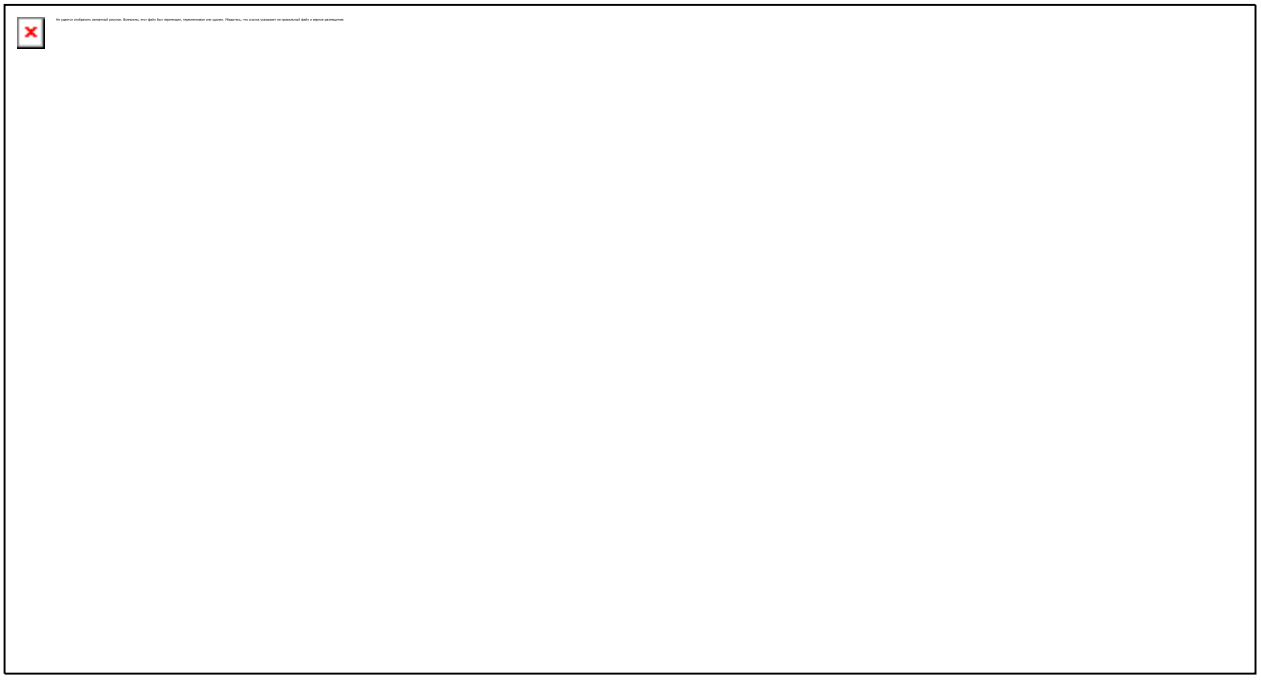


Рисунок 8.12. Пункты-переключатели в меню

Шаг 11. В ответ на выбор пользователем пунктов **Toolbar** и **Status bar** будем переключать флажок. Вы уже знаете, как определить обработчик события **OnClick** для пункта меню, поэтому сразу приведем то, что вы должны получить:

```
procedure TPictureForm.ToolBarMenuItemClick(Sender: TObject);
begin
  // Спрятать или показать панель инструментов
  ToolBarMenuItem.Checked := not ToolBarMenuItem.Checked;
end;

procedure TPictureForm.StatusBarMenuItemClick(Sender: TObject);
begin
  // Спрятать или показать строку состояния
  StatusBarMenuItem.Checked := not StatusBarMenuItem.Checked;
end;
```

Готово, соберите проект и проверьте, что пункты **Toolbar** и **Status bar** стали работать как переключатели. Позже, когда вы создадите в своем приложении строку состояния и панель инструментов, мы допишем эти обработчики событий. А сейчас рассмотрим еще один тип пунктов меню — взаимоисключающие переключатели.

8.1.9. Взаимоисключающие переключатели

Кроме обычных переключателей в меню широко применяются взаимоисключающие переключатели. Такие пункты работают согласовано — включение одного из них означает выключение остальных. В нашем примере с помощью взаимоисключающих переключателей удобно реализовать выбор масштаба для рисунка. Рассмотрим, как это делается.

Шаг 12. Добавьте в меню **View** три пункта: **Half Size** (программное имя **HalfSizeMenuItem**), **Normal Size** (программное имя **NormalSizeMenuItem**) и **Double Size** (программное имя **DoubleSizeMenuItem**), отделив их от остальных пунктов меню с помощью разделительной линии.

Шаг 13. Объедините только что созданные пункты меню в одну согласовано работающую группу. Для этого у всех пунктов установите одинаковое ненулевое значение свойства **GroupIndex** (например, 1). Кроме того, установите для всех этих пунктов свойство **RadioItem** в значение True, чтобы метка имела вид жирной точки. Один из пунктов (например, **Normal Size**) переведите во включенное состояние, установив его свойство **Checked** в значение True (рисунок 8.13).

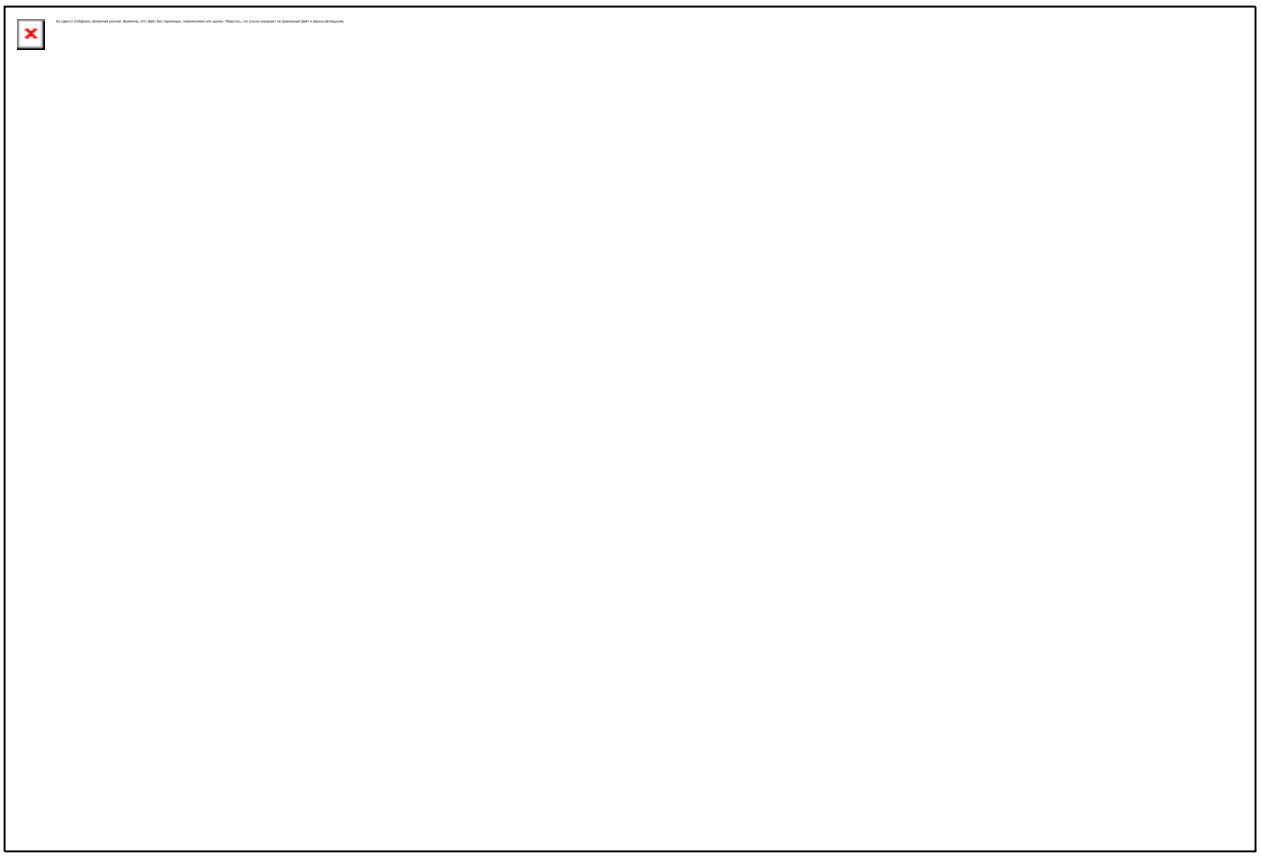


Рисунок 8.13. Взаимоисключающие переключатели в меню

Шаг 14. Чтобы привести в действие механизм переключения пунктов, определите в них следующие обработчики события **OnClick**:

```
procedure TPictureForm.HalfSizeMenuItemClick(Sender: TObject);
begin
  // Показать рисунок половинного размера
  HalfSizeMenuItem.Checked := True;
end;

procedure TPictureForm.NormalSizeMenuItemClick(Sender: TObject);
begin
  // Показать рисунок нормального размера
  NormalSizeMenuItem.Checked := True;
end;

procedure TPictureForm.DoubleSizeMenuItemClick(Sender: TObject);
begin
  // Показать рисунок двойного размера
  DoubleSizeMenuItem.Checked := True;
end;
```

Выполнив компиляцию, запустите программу и проверьте, что новые пункты меню работают как взаимоисключающие переключатели.

8.1.10. Недоступные пункты меню

Некоторые пункты меню могут быть недоступны пользователю в тот или иной момент времени. Такие пункты выглядят блеклыми, а попытки их выбрать ни к чему не приводят. Согласитесь, что легче запретить выбор отдельных пунктов меню, чем программировать логику поведения на случай, когда пользователь выбрал неправильную команду.

Шаг 15. В нашем примере логично было бы сделать недоступными пункты **Save As...**, **Close**, а также **Half Size**, **Normal Size** и **Double Size**, когда нет открытого графического файла. Для этого в каждом из указанных пунктов меню установите свойство **Enabled** в значение **False** (рисунок 8.14).

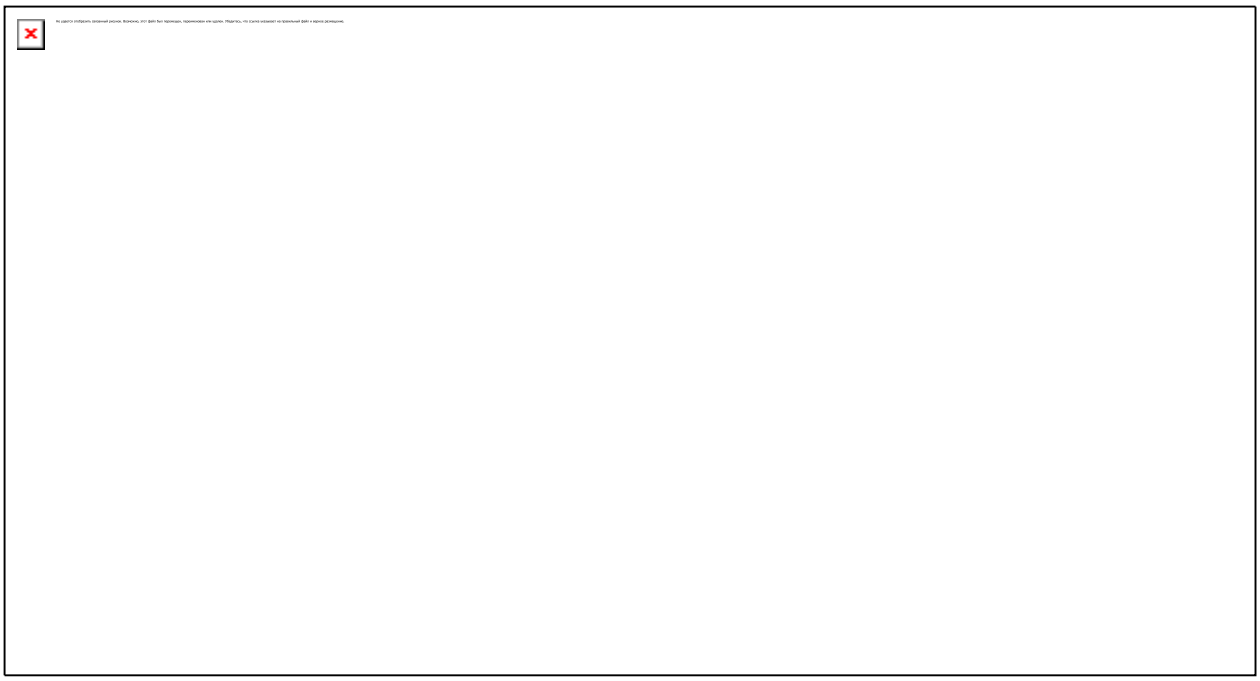


Рисунок 8.14. Недоступные пункты меню

Шаг 16. Во время работы приложения нужно еще динамически делать пункты меню доступными или недоступными в зависимости от того, открыт графический файл или нет. Так как эти действия достаточно универсальны, оформим их в виде отдельного метода **EnableCommands**:

```
type
  TPictureForm = class(TForm)
    ...
  private
    { Private declarations }
    procedure EnableCommands(Enable: Boolean);
  end;

...

procedure TPictureForm.EnableCommands(Enable: Boolean);
begin
  SaveAsMenuItem.Enabled := Enable;
  CloseMenuItem.Enabled := Enable;
  HalfSizeMenuItem.Enabled := Enable;
  NormalSizeMenuItem.Enabled := Enable;
  DoubleSizeMenuItem.Enabled := Enable;
end;
```

Параметр **Enable** данного метода определяет, в какое состояние перевести пункты меню: доступны — True или недоступны — False.

Шаг 17. Создайте обработчики команд **Open...** и **Close**. Как вы понимаете, в обработчик команды меню **Open...** следует поместить вызов метода **EnableCommands** с параметром True, а в обработчик команды **Close** — вызов метода **EnableCommands** с параметром False:

```
procedure TPictureForm.OpenMenuItemClick(Sender: TObject);
begin
  // Открыть рисунок и разрешить команды
  EnableCommands(True);
end;

procedure TPictureForm.CloseMenuItemClick(Sender: TObject);
begin
  // Закрыть рисунок и запретить команды
  EnableCommands(False);
end;
```

Выполните компиляцию и запустите программу. Посмотрите, как изменилось ее меню (рисунок 8.15).

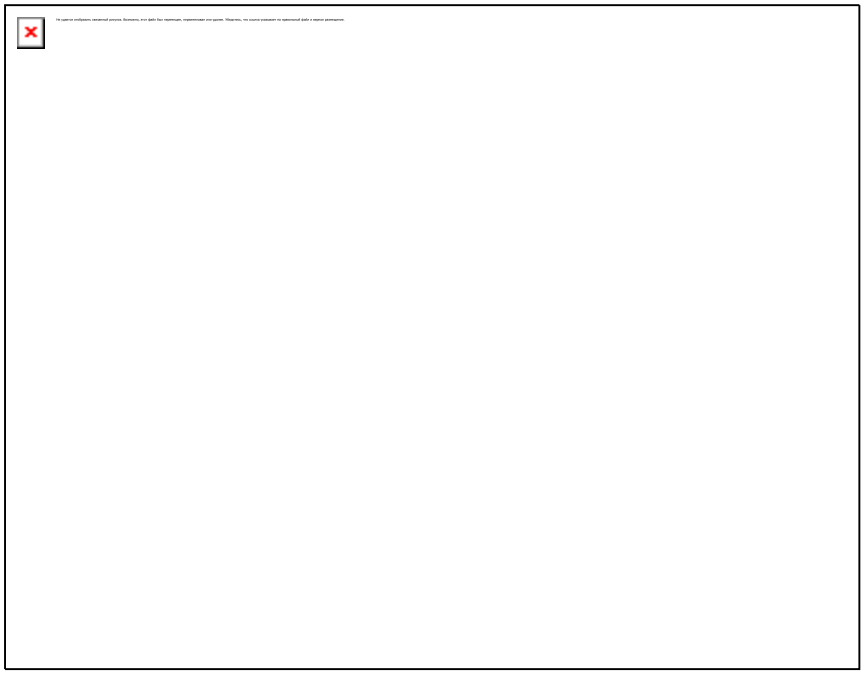


Рисунок 8.15. Меню работающей программы. Некоторые пункты недоступны.

В самом начале работы отдельные пункты выпадающих меню **File** и **View** недоступны. Они становятся доступными после выбора команды **File | Open...**, а после выбора команды **File | Close** — снова недоступными.

Итак, вы изучили все основные аспекты программирования главного меню, поэтому перейдем к вопросу разработки контекстных меню.

8.1.11. Контекстное меню

Контекстное (вспомогательное) меню представлено в среде Delphi компонентом **PopupMenu** (рисунок 8.16). Отыщите его в палитре компонентов на вкладке **Standard** и поместите на форму. Дайте новому компоненту имя **PopupMenu**.

Рисунок 8.16. Компонент PopupMenu

Прежде, чем перейти к практическому использованию контекстных меню, кратко опишем отличительные свойства компонента **PopupMenu** (таблица 8.4).

Свойство	Описание

Alignment	Определяет место появления меню относительно указателя мыши: **paLeft** — левый верхний угол меню совпадает с позицией курсора мыши; **paCenter** — середина верхнего края меню совпадает с позицией курсора мыши; **paRight** — правый верхний угол меню совпадает с позицией курсора мыши.
AutoHotkeys	Значение **maAutomatic** избавляет программиста от необходимости назначать пунктам меню "горячие" клавиши (с помощью специального символа **&** в тексте пунктов); компонент автоматически подбирает "горячие" клавиши. Значение **maManual** требует, чтобы "горячие" клавиши назначил программист (см. параграф 8.1.3).
AutoLineReduction	Если равно значению **maAutomatic**, то при отображении меню подряд идущие пункты-разделители рисуются как один разделитель, а пункты-разделители, находящиеся в начале или конце меню вообще не показываются. Свойство AutoLineReduction применяется при программном добавлении и удалении пунктов меню, чтобы избежать нежелательных явлений вроде повторяющихся и повисших разделительных линий. Если свойство AutoLineReduction равно значению **maManual**, то все пункты меню отображаются как есть.
AutoPopup	Если равно значению **True**, то меню появляется автоматически по нажатию правой кнопки мыши. Если равно значению **False**, то меню необходимо отображать программно.
Images	Список значков, отображаемых рядом с пунктами меню. Свойство **Images** используется совместно со свойством **ImageIndex** компонентов **MenuItem** (см. параграф 8.1.12).
Items	Обеспечивает нумерованный доступ к пунктам меню.
MenuAnimation	Набор флажков, определяющих способ появления меню на экране: **maLeftToRight** — слева направо, **maRightToLeft** — справа налево, **maTopToBottom** — сверху вниз, **maBottomToTop** — снизу вверх, **maNone** — мгновенное отображение. Чтобы флажки начали работать, запустите программу настройки экрана (Start–>Settings–>Control Panel–>Display) и на вкладке Effects выберите способ появления меню и подсказок — Scroll Effect.
OwnerDraw	Если равно значению True, то каждый пункт меню получает возможность участвовать в процессе своего отображения при помощи специальных событий **OnMeasureItem** и **OnDrawItem**. Событие

	OnMeasureItem происходит в пункте меню, когда расчитываются размеры пункта. Событие **OnDrawItem** происходит в пункте меню, когда пункт рисуется на экране. Если свойство **OwnerDraw** равно значению False, то пукнты меню имеют стандартный вид и события **OnMeasureItem** и **OnDrawItem** не происходят.
TrackButton	Кнопка мыши для выбора пункта меню: **tbLeftButton** — левая кнопка, **tbRightButton** — еще и правая кнопка.
OnChange	Происходит при изменении структуры меню.
OnPopup	Происходит при вызове меню пользователем.

*Таблица 8.4. Важнейшие свойства и события компонента **PopupMenu***

Шаг 18. Контекстное меню наполняется пунктами, как и главное меню, в дизайнере меню. Двойным щелчком мыши на компоненте **PopupMenu** откройте окно конструктора меню и, используя уже известные вам приемы, добавьте в меню пункты **Half Size** (с идентификатором **HalfSizePopupItem**), **Normal Size** (с идентификатором **NormalSizePopupItem**) and **Double Size** (с идентификатором **DoubleSizePopupItem**). Во всех пунктах контекстного меню установите следующие свойства:

Enabled = False

GroupIndex = 1

RadioItem = True

Кроме этого пометьте пункт **Normal Size**, установив в нем свойство **Checked** в значение **True**. Таким образом, команды всплывающего меню дублируют некоторые команды главного меню, обеспечивая пользователю дополнительные удобства (рисунок 8.17).

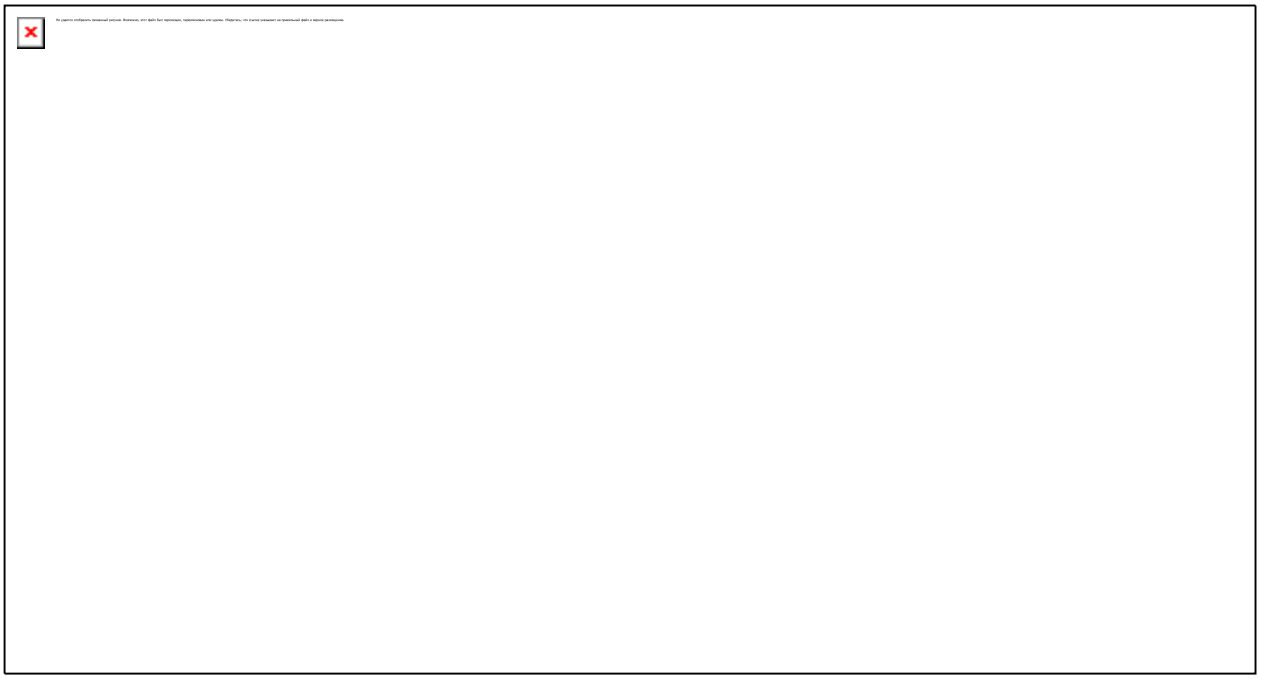


Рисунок 8.17. Команды контекстного меню

Проектирование меню завершено и сейчас перейдем к программированию обработчиков событий. В данном примере команды контекстного меню обрабатываются так же, как и команды одноименных пунктов главного меню. Поскольку для пунктов главного меню обработчики уже написаны, то их просто нужно связать с пунктами контекстного меню. Это делается очень просто.

Шаг 19. Активизируйте в дизайнере меню пункт **Half Size** и выберите в окне свойств вкладку **Events**. Выберите обработчик **HalfSizeMenuItemClick** из раскрывающегося списка события **OnClick**. То же самое проделайте с пунктами **Normal Size** и **Double Size**, установив для них обработчики **NormalSizeMenuItemClick** и **DoubleSizeMenuItemClick** соответственно (рисунок 8.18).

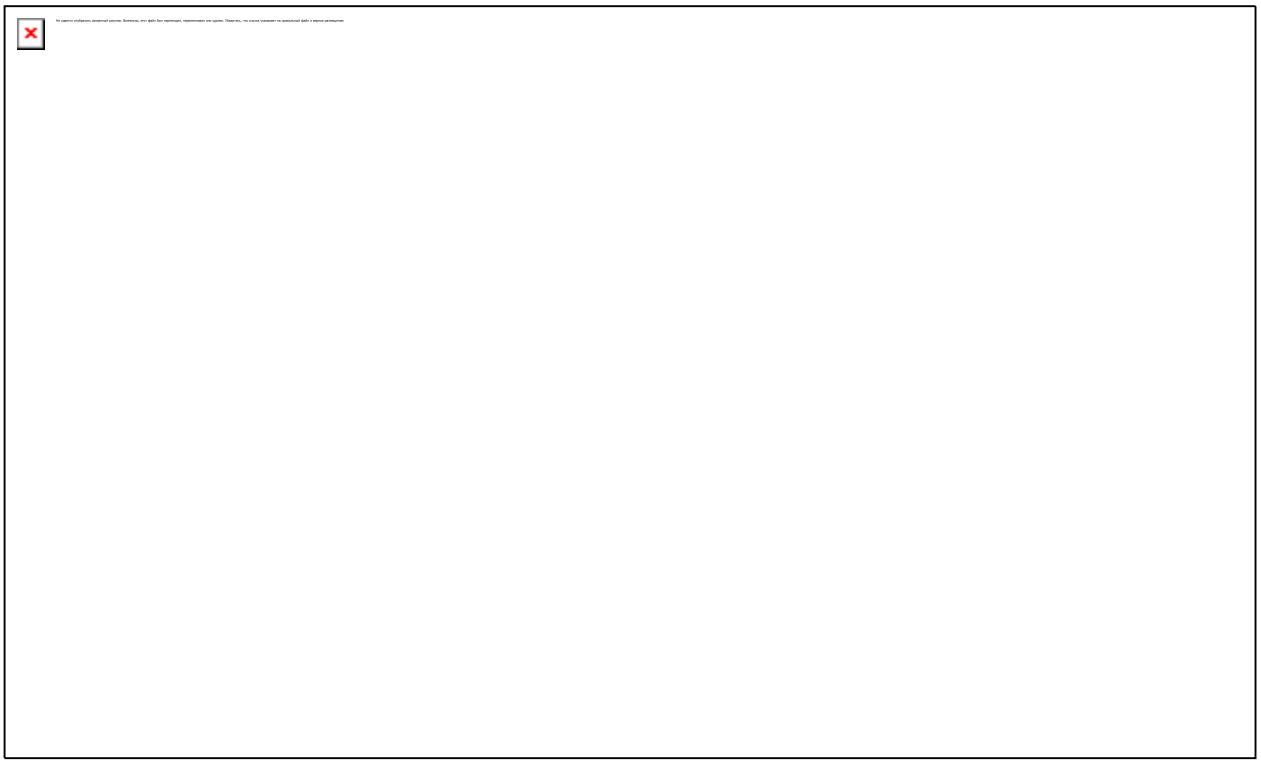


Рисунок 8.18. Установка обработчиков команд контекстного меню

Шаг 20. Для синхронной работы главного и контекстного меню нужно еще подправить некоторые обработчики:

```
procedure TPictureForm.HalfSizeMenuItemClick(Sender: TObject);
begin
  // Показать рисунок половинного размера
  HalfSizeMenuItem.Checked := True;
  HalfSizePopupItem.Checked := True;
end;

procedure TPictureForm.NormalSizeMenuItemClick(Sender: TObject);
begin
  // Показать рисунок нормального размера
  NormalSizeMenuItem.Checked := True;
  NormalSizePopupItem.Checked := True;
end;

procedure TPictureForm.DoubleSizeMenuItemClick(Sender: TObject);
begin
  // Показать рисунок двойного размера
  DoubleSizeMenuItem.Checked := True;
  DoubleSizePopupItem.Checked := True;
end;

procedure TPictureForm.EnableCommands(Enable: Boolean);
begin
  SaveAsMenuItem.Enabled := Enable;
  CloseMenuItem.Enabled := Enable;
  HalfSizeMenuItem.Enabled := Enable;
  HalfSizePopupItem.Enabled := Enable;
  NormalSizeMenuItem.Enabled := Enable;
  NormalSizePopupItem.Enabled := Enable;
  DoubleSizeMenuItem.Enabled := Enable;
  DoubleSizePopupItem.Enabled := Enable;
end;
```

Шаг 21. Контекстное меню готово, осталось сделать так, чтобы оно вызывалось по щелчку правой кнопки мыши на форме. Нет ничего проще — активизируйте форму и запишите в значении свойства **PopupMenu** имя разработанного ранее контекстного меню — **PopupMenu**. Вы можете ввести это значение с клавиатуры или выбрать из раскрывающегося списка (рисунок 8.19).

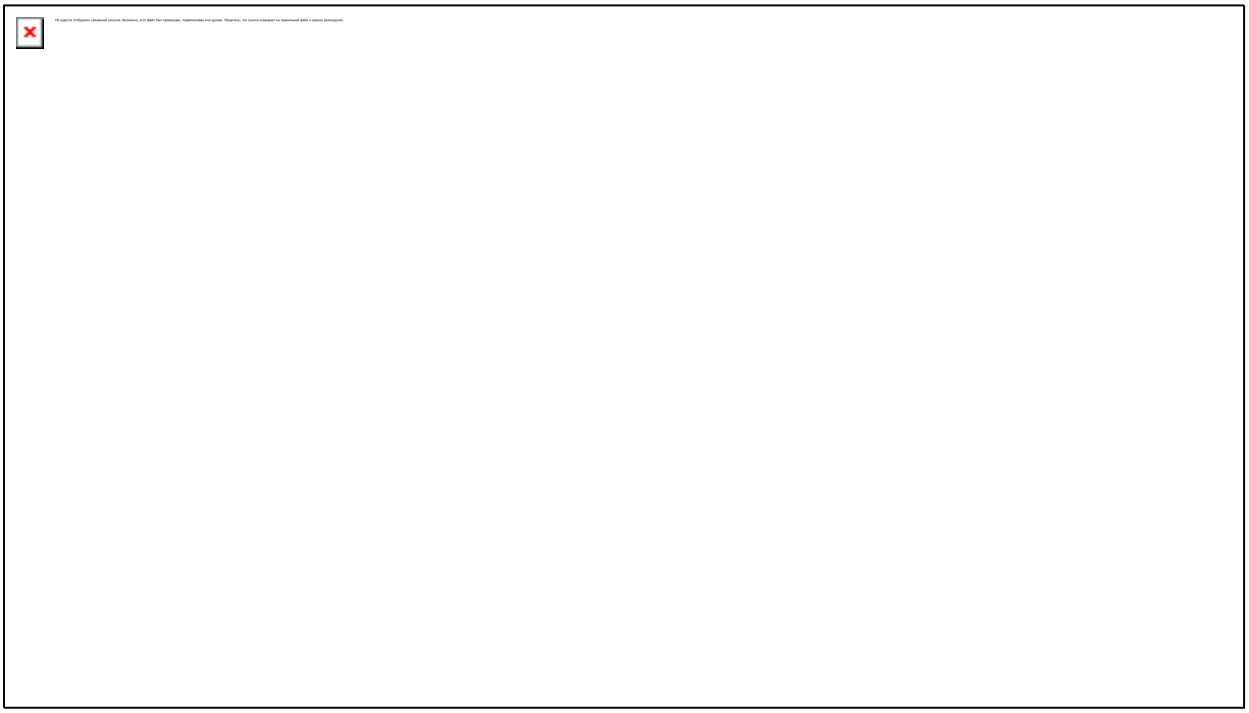


Рисунок 8.19. Привязка контекстного меню к форме

Готово, выполните компиляцию и запустите программу. Нажатие правой кнопки мыши в окне приложения вызовет появление контекстного меню. Все его пункты окажутся

недоступными. Чтобы пункты контекстного меню заработали, выполните команду главного меню **File | Open**. После этого проверьте, что контекстное меню работает синхронно с главным меню.

8.1.12. Значки в пунктах меню

Каждый пункт меню помимо текста может содержать красочный значок, наглядно поясняющий назначение пункта. Самый простой способ создания значка в пункте меню — установить свойство **Bitmap**.

Шаг 22. Вызовите дизайнер меню для компонента **MainMenu** формы **PictureForm**. Выберите пункт **File | Open** и перейдите к свойству **Bitmap** в окне свойств (рисунок 8.20).

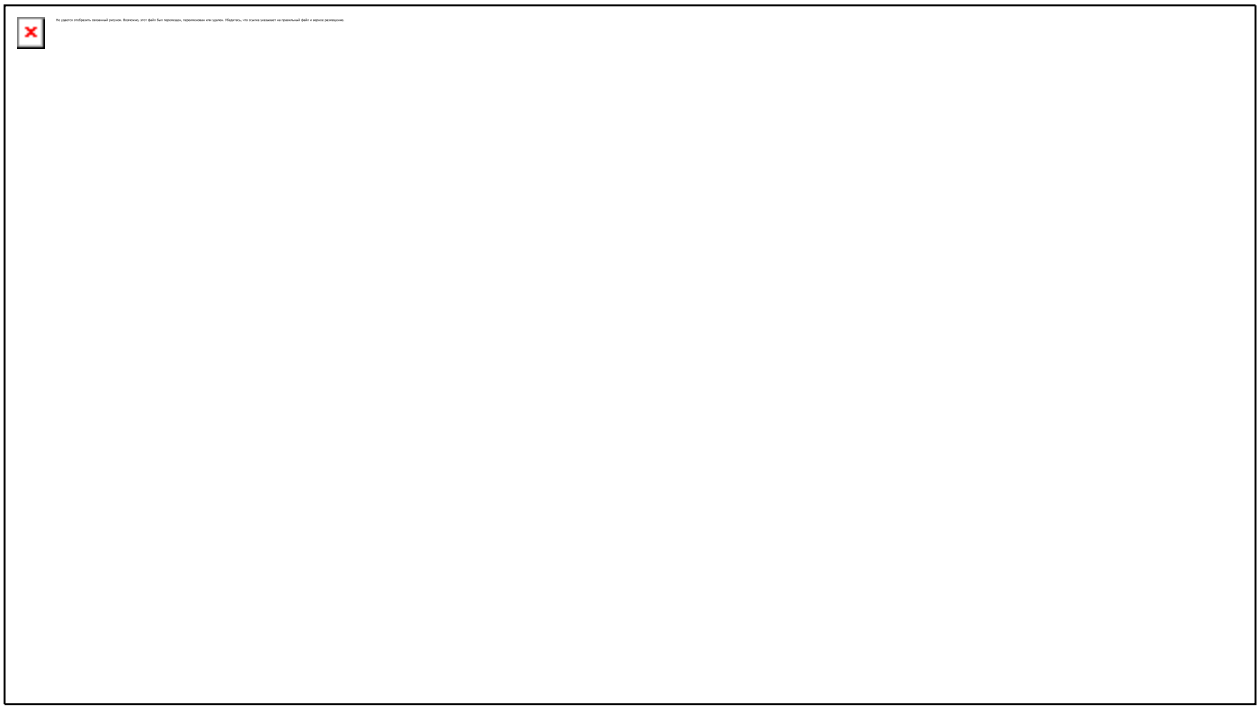


Рисунок 8.20. Свойство Bitmap пункта меню

Шаг 23. Установка значения свойства **Bitmap** осуществляется с помощью уже знакомого вам окна **Picture Editor** (рисунок 8.21), вызываемого нажатием кнопки с многоточием в поле свойства. В этом окне нажмите кнопку **Load...** и выберите файл Open.bmp из коллекции рисунков на компакт-диске (каталог \Images).

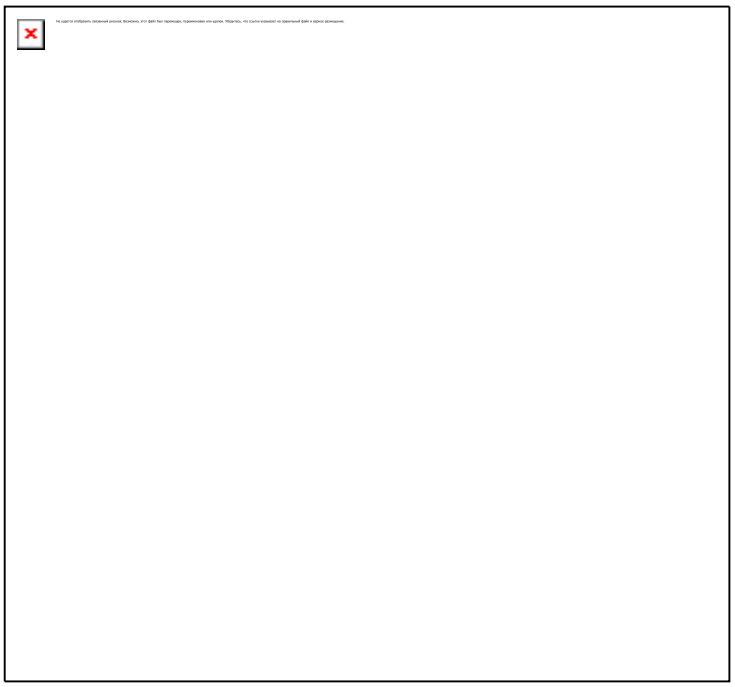


Рисунок 8.21. Окно Picture Editor

Наконец, закройте диалоговое окно с помощью кнопки **OK**. Результат показан на рисунке 8.22.

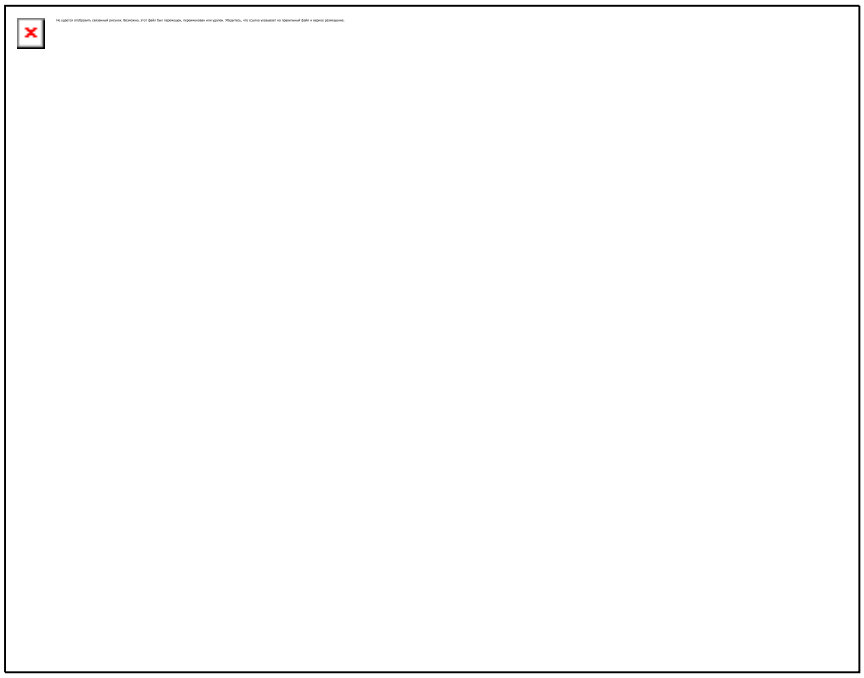


Рисунок 8.22. Пункт Open имеет значок

Такой способ создания значков в меню очень прост, но не всегда удобен. В тех случаях, когда количество пунктов меню исчисляется десятками и многие пункты главного меню дублируются в контекстных меню и панели кнопок, использовать для каждого пункта отдельную копию одного и того же значка не эффективно, да и неудобно. В таких случаях на помощь приходит компонент **ImageList**. Его основные свойства приведены в таблице 8.5.

Как вы уже могли догадаться из названия компонента **ImageList**, он предназначен для хранения заранее подготовленного списка значков (в общем случае, любых рисунков). Другие компоненты берут значки из этого списка по номеру. Поскольку для хранения номера требуется гораздо меньше памяти, чем для хранения самого значка, то при

использовании одного и того же значка в нескольких компонентах, достигается значительная экономия памяти. Да и управлять значками с помощью компонента **ImageList** тоже очень удобно. Если изменить значок в списке, то он автоматически изменится во всех компонентах, которые его используют. Существует лишь одно ограничение при использовании компонента **ImageList** — все хранимые значки должны иметь одинаковые размеры.

Свойство	Описание
Width, Height	Ширина и высота рисунков в списке.
AllocBy	Приращение массива. Когда массив полностью заполнен и делается попытка добавить новый рисунок, размер массива увеличивается на **AllocBy** элементов. Используется для оптимизации скорости добавления элементов и занимаемой ими памяти.
BkColor	Цвет фона. Этим цветом запоняются те части рисунков, которые должны быть прозрачными. Значение clNone оставляет фон прозрачным.
BlendColor	Цвет, которым подсвечиваются рисунки. Наличие подсветки и ее яркость зависит от значения свойства DrawingStyle.
DrawingStyle	Способ отображения рисунков: **dsFocus** — легкая (25%) подсветка цветом **BlendColor**; **dsSelected** — сильная (50%) подсветка цветом **BlendColor**; **dsNormal** — подсветка отсутствует, цвет фона берется из свойства **BkColor**. Если свойство BkColor содержит значение **clNone**, то фон прозрачный. **dsTransparent** — подсветка отсутствует, цвет фона прозрачный независимо от значения свойства **BkColor**. Значение этого свойства может игнорироваться стандартными компонентами. Оно полезно, если рисунки из списка отображаются программно с помощью метода **Draw**.
ImageType	Выбирает между отображением рисунков (значение **tiImage**) и их масок (значение **tiMask**). Значение этого свойства может игнорироваться стандартными компонентами. Оно полезно, если рисунки из списка отображаются программно с помощью метода **Draw**.
Masked	Если равно значению True, то при добавлении рисунка в список для него создается специальная маска. Маска описывает фоновые пиксели рисунка и используется при выводе рисунка на экран. Наличие маски позволяет манипулировать фоном рисунка с помощью свойств **BkColor** и **DrawingStyle**. Если свойство Masked равно значению False, то рисунок помещается в список без маски и всегда рисуется как есть. В этом случае свойство **BkColor**

	игнорируется, а значение **dsTrasnparent** в свойстве **DrawingStyle** не производит эффекта.
ShareImages	Если равно значению False, то при уничтожении компонента уничтожается также соответствующий объект операционной системы Windows, который скрыт внутри компонента **ImageList**. Если равно значению True, то при уничтожении компонента связанный с ним объект операционной системы не уничтожается, что позволяет использовать этот объект за пределами библиотеки VCL.
OnChange	Происходит при любом изменении списка.

*Таблица 8.5. Основные свойства и события компонента **ImageList***

Воспользуемся компонентом **ImageList** для хранения значков в нашей программе. Найдите его в палитре компонентов на вкладке **Win32** (рисунок 8.23).

Рисунок 8.23. Компонент ImageList

Поместите компонент **ImageList** на форму и дайте ему имя **ImageList**. Обратите внимание, что стандартные значения свойств **Width** и **Height** равны 16, поэтому не забудьте их изменить, если ваши значки имеют другие размеры.

Шаг 24. Редактирование списка рисунков осуществляется в специальном окне. Вызовите его с помощью команды **ImageList Editor…**, находящейся в контекстном меню компонента **ImageList** (рисунок 8.24).

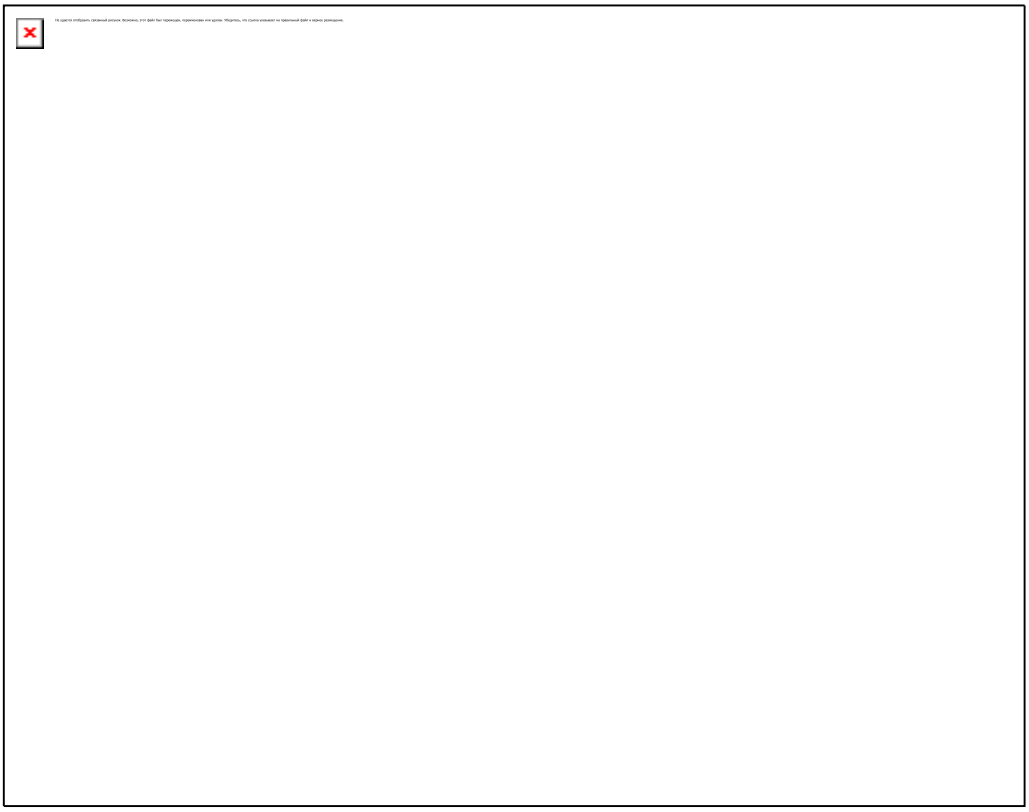


Рисунок 8.24. Вызов окна для редактирования списка значков

Шаг 25. В открывшемся окне (рисунок 8.25) нажмите кнопку **Add...** и выберите несколько файлов из коллекции рисунков на компакт-диске (каталог \Images).

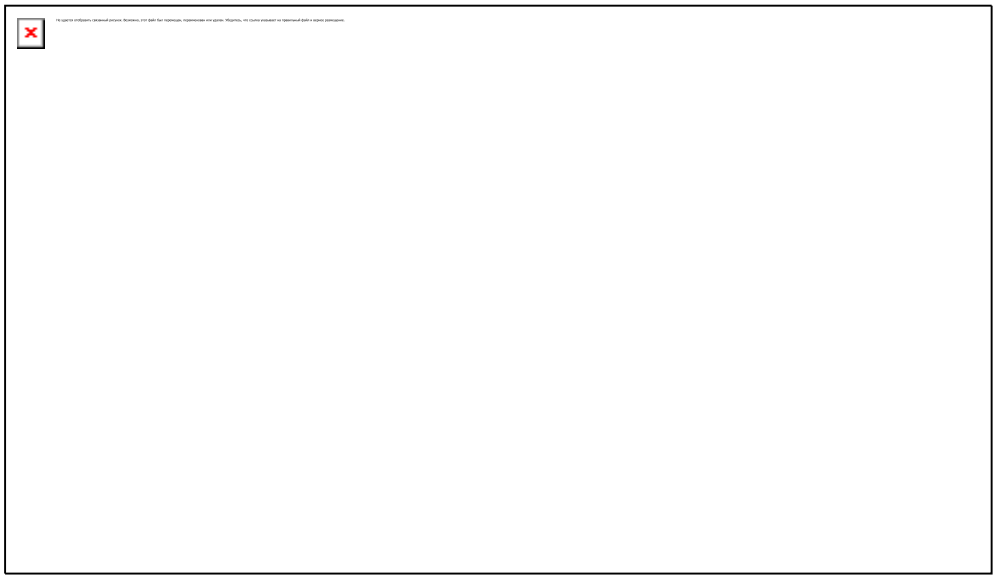


Рисунок 8.25. Окно, в котором редактируется список значков

В этом окне параметры **Transparent Color**, **Fill Color** и **Options** нуждаются в пояснении. Параметр **Transparent Color** — это цвет пикселей, которые становятся прозрачными. Параметры **Options** и **Fill Color** используются, если размеры рисунка не совпадают с размерами, указанными в свойствах **Width** и **Height** компонента **ImageList**. Параметр **Options** принимает следующие значения: **Crop** — заполнить лишние пиксели цветом **Fill Color** (либо отсечь правую нижнюю часть рисунка, если рисунок превышает размеры); **Stretch** — растянуть/сжать рисунок до принятых размеров; **Center** — центрировать рисунок, заполнив оставшуюся область цветом **Fill Color**.

Закройте диалоговое окно с помощью кнопки **OK**.

Шаг 26. Список значков мы подготовили. Теперь нужно указать нашему компоненту **MainMenu**, откуда брать значки для своих пунктов. Выделите компонент **MainMenu** на форме и перейдите к свойству **Images** в окне свойств. Из списка значений свойства **Images** выберите элемент **ImageList**.

Шаг 27. Теперь осталось указать пунктам меню номера соответствующих им значков. Для этого вызовите дизайнер меню для компонента **MainMenu**. Как это сделать, вы уже знаете. Далее выберите пункт **Open...** (программный идентификатор **OpenMenuItem**), перейдите к свойству **ImageIndex** и выберите из раскрывающегося списка подходящий значок (рисунок 8.26).

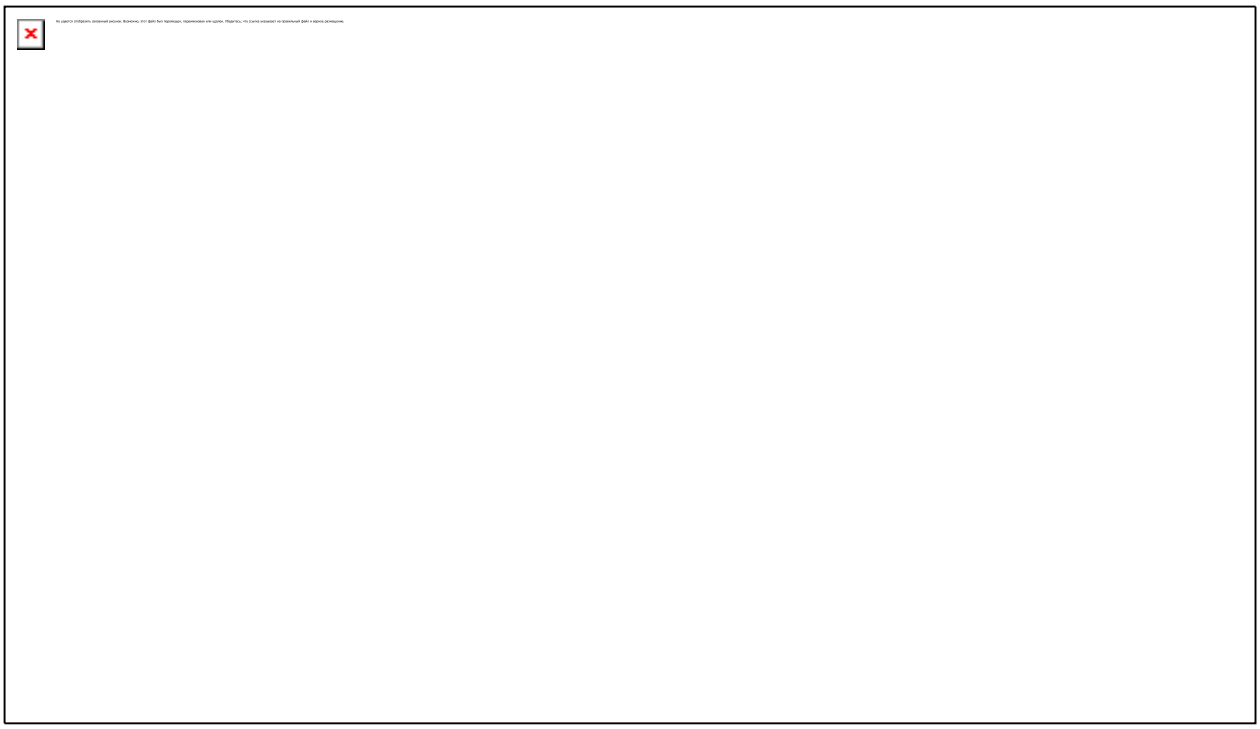


Рисунок 8.26. Установка значка для пункта меню

Аналогично установите номера соответствующих значков для пунктов **Save As…**, **Half Size**, **Normal Size** и **Double Size**. Не пугайтесь внешнего вида значков в недоступных пунктах меню. Они, как и текст, отображаются блеклыми.

На рисунке 8.27 показана форма после добавления значков.

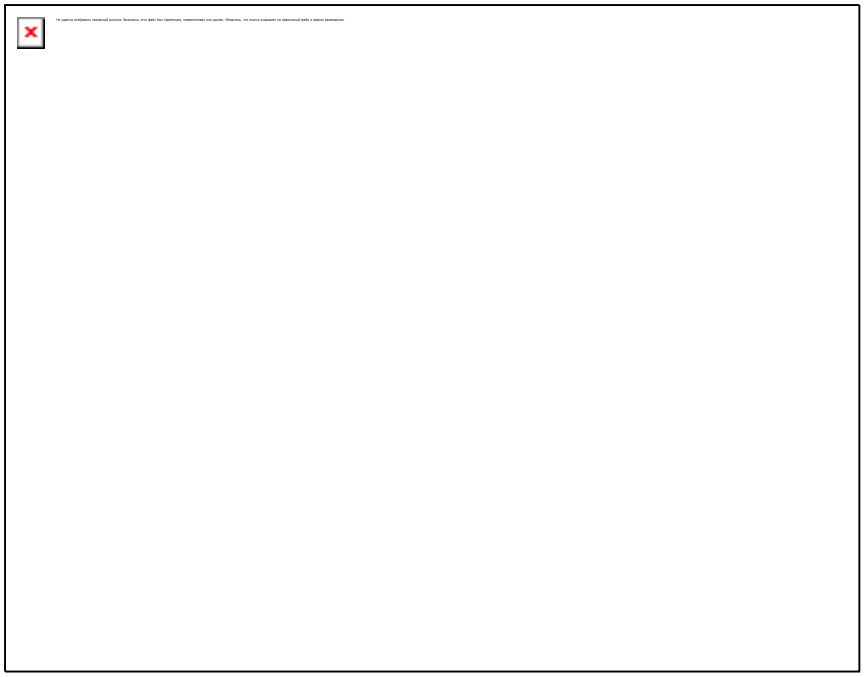


Рисунок 8.27. В меню добавлены значки

Шаг 28. Теперь установим значки для пунктов контекстного меню нашей формы. Активизируйте форму **PictureForm** и выберите на ней компонент **PopupMenu**. Затем в окне свойств перейдите к свойству **Images** и из списка значений этого свойства выберите элемент **ImageList**. После этого вызовите дизайнер меню у компонента **PopupMenu**, и аналогично тому, как вы это делали в главном меню, назначьте номера подходящих значков для пунктов **Half Size**, **Normal Size** и **Double Size** контекстного меню.

Внимание! Значок, заданный с помощью свойства **Bitmap**, используется только в случае, когда свойство **ImageIndex** содержит отрицательное число. Поэтому если вы обнаружите, что установка значка с помощью свойства **Bitmap** не приносит желаемого результата, не паникуйте, а просто проверьте свойство **ImageIndex**.

Шаг 29. Кстати, мы совсем забыли про значок, все еще хранящийся в свойстве **Bitmap** компонента **OpenMenuItem**. Сейчас в нем нет необходимости, поскольку реально используется значок, заданный с помощью свойства **ImageIndex**. Чтобы не держать в программе ненужные данные, удалите значок из свойства **Bitmap**. Для этого вызовите дизайнер меню для компонента **MainMenu** и выберите пункт **Open...**. Далее в окне свойств перейдите к свойству **Bitmap** и нажмите клавишу **Del**, после чего нажмите клавишу **Enter**.

Теперь мы вплотную приблизились к полноценному приложению для просмотра графических файлов.

8.2. Полноценное приложение для просмотра графических файлов

Сейчас вы достаточно много знаете о меню, и вас наверняка одолевает желание сделать из вышеприведенной заготовки полноценное приложение для просмотра графических файлов. Для этого необходимо решить две задачи:

- организовать выбор файла по командам меню **Open...** и **Save As...** ;
- реализовать загрузку и отображение рисунка.

Первая задача решается с помощью стандартных диалоговых компонентов **OpenDialog** и **SaveDialog**, вторая — с помощью специального компонента **Image**.

8.2.1. Диалоговые окна открытия и сохранения файла

Шаг 30. Диалоговые окна для выбора открываемого или сохраняемого файла организуются с помощью компонентов **OpenDialog** и **SaveDialog** (рисунок 8.29). Найдите их в палитре компонентов на вкладке **Dialogs** и поместите на форму. Первый компонент назовите **OpenDialog**, а второй — **SaveDialog**.

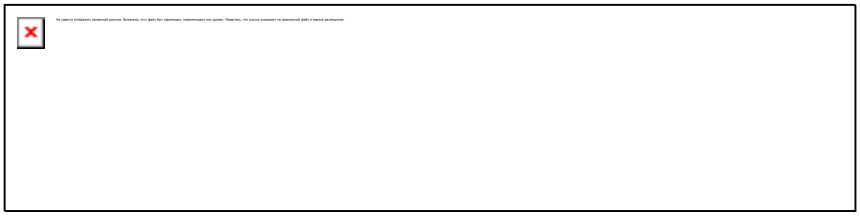


Рисунок 8.28. Компоненты OpenDialog и SaveDialog

Характерные свойства этих компонентов кратко описаны в таблице 8.6.

Свойство	Описание
DefaultExt	Расширение, которое добавляется к имени файла, если пользователь его не указал.
FileName	Имя выбранного файла.
Filter	Фильтры имени файла.
FilterIndex	Номер активного фильтра.
InitialDir	Начальный каталог, открываемый при первом

	появлении окна диалога.
Options	Параметры, определяющие внешний вид и поведение окна диалога. (см. таблицу 8.7).
OptionsEx	Дополнительные параметры, определяющие внешний вид и поведение окна диалога (см. таблицу 8.7).
Title	Заголовок окна диалога. Если значение свойства не указано, то заголовок будет стандартным — Open (Открыть) или Save (Сохранить) в зависимости от типа компонента.
OnCanClose	Происходит, когда пользователь пытается закрыть окно диалога. Позволяет выполнить дополнительные проверки и отменить закрытие окна при необходимости.
OnClose	Происходит непосредственно перед закрытием формы после события OnCanClose.
OnFolderChange	Происходит, если пользователь переходит в другой каталог.
OnIncludeItem	Происходит при добавлении каждого файла в список отображаемых в окне файлов. Позволяет выполнять дополнительную фильтрацию файлов.
OnSelectionChange	Происходит при смене выделенного файла или списка файлов.
OnShow	Происходит непосредственно перед отображением окна диалога на экране.
OnTypeChange	Происходит, когда пользователь выбирает фильтр.

*Таблица 8.6. Важнейшие свойства и события компонентов **OpenDialog** и **SaveDialog**.*

Компоненты **OpenDialog** и **SaveDialog** очень схожи между собой, оба являются объектно-ориентированными оболочками стандартных диалоговых окон Windows: Open и Save. На следующем рисунке показано окно Open (рисунок 8.29).

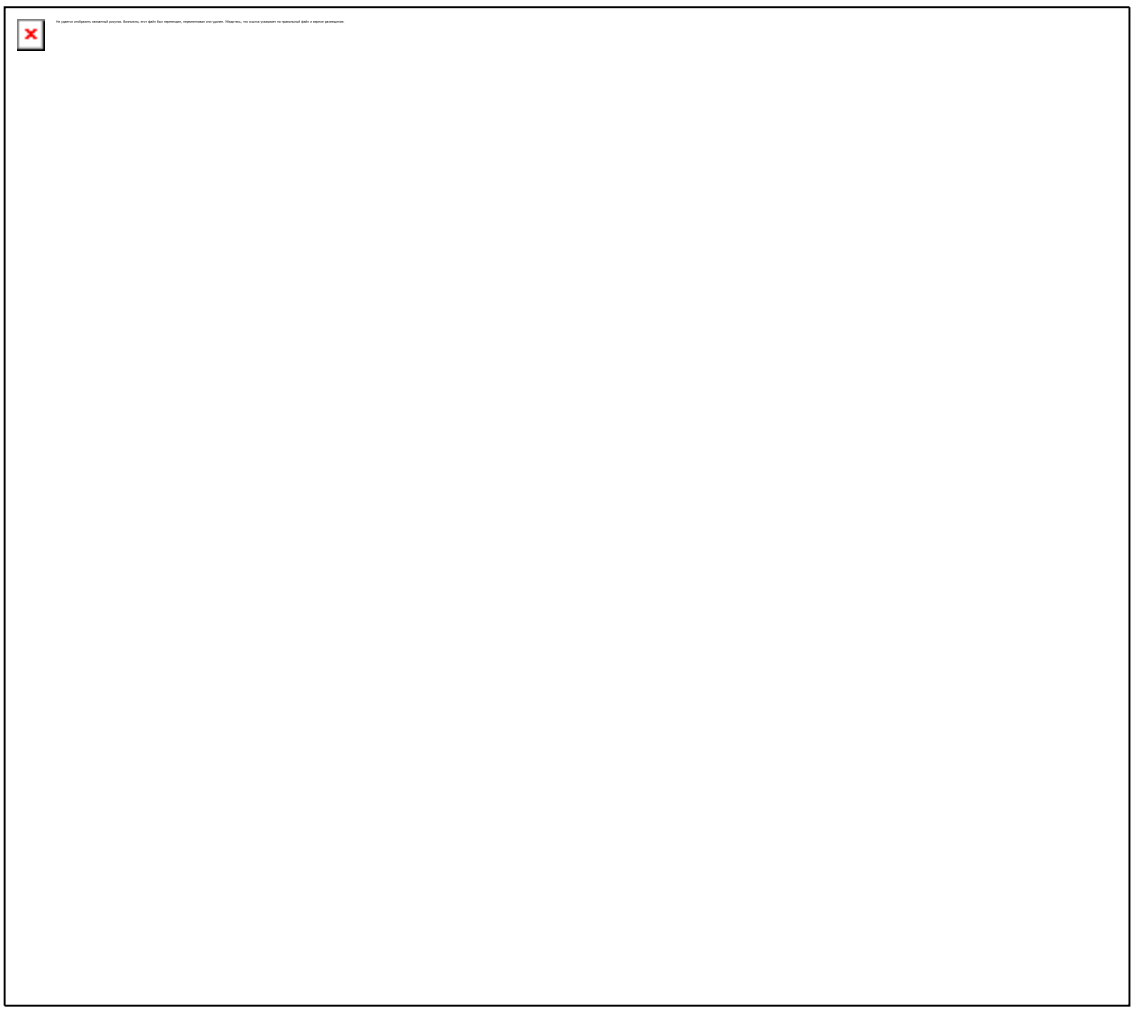


Рисунок 8.29. Стандартное окно Open для выбора открываемого файла

Приблизительный сценарий работы с каждым из компонентов **OpenDialog** и **SaveDialog** таков. Компонент помещается на форму и конфигурируется для выбора тех или иных файлов. По команде меню **Open...** или **Save As...** у соответствующего компонента вызывается метод **Execute**. Он вызывает диалог и возвращает значение **True**, если пользователь выбрал файл. Полный маршрут к файлу запоминается в свойстве **FileName**. Ход дальнейших действий зависит от прикладной задачи и, как правило, включает или чтение, или запись файла, в зависимости от обрабатываемой команды меню.

Придерживаясь написанного сценария, приспособим компоненты **OpenDialog** и **SaveDialog** для выбора графических файлов, поддерживаемых нашей программой. Чтобы пользователь мог просматривать файлы выборочно (какого-то одного типа) в диалоговых блоках имеется набор фильтров, оформленный в виде раскрывающегося списка с подписью **Files of type** (см. рис. выше). Исходные данные для этого списка устанавливаются в свойстве **Filter**. Номер активного в данный момент фильтра записывается в свойстве **FilterIndex**.

Шаг 31. Приступим к формированию списка фильтров. Активизируйте на форме компонент **OpenDialog** и в окне свойств выберите свойство **Filter** (рисунок 8.30). Щелчком кнопки с многоточием откройте редактор фильтров — окно **Filter Editor** (рисунок 8.31).

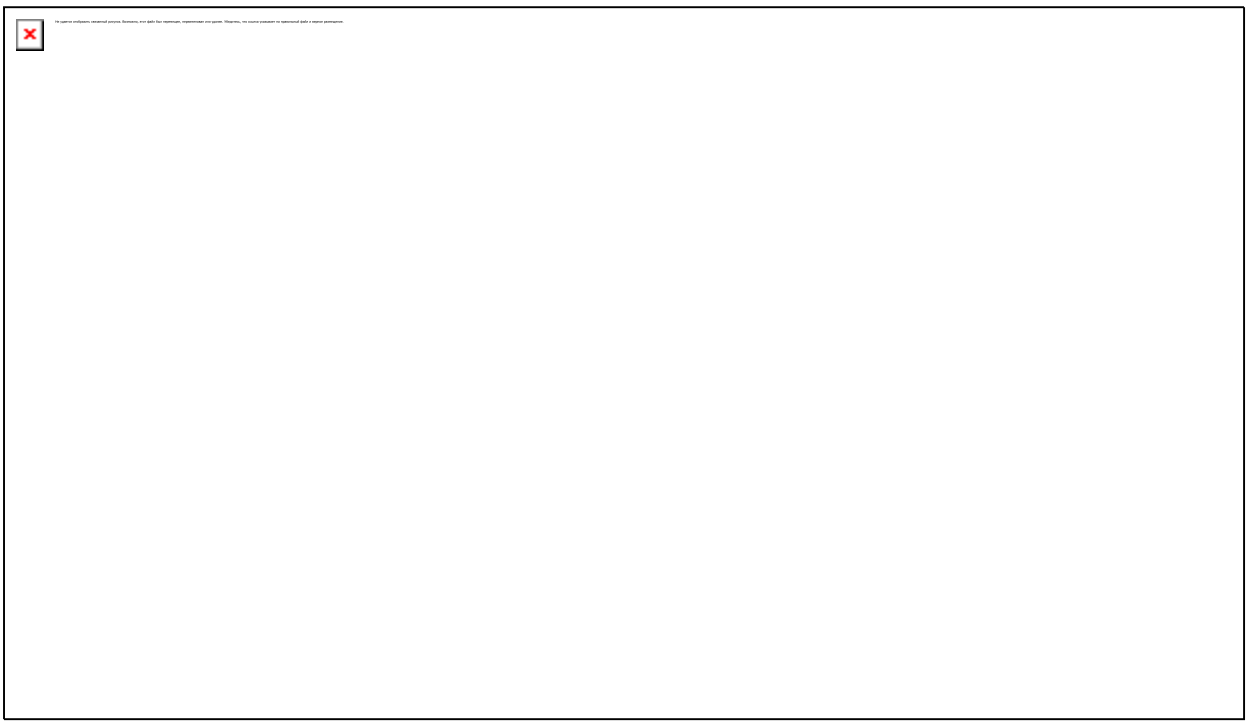


Рисунок 8.30. Нажатие кнопки с многоточием вызывает редактор фильтров

Окно **Filter Editor** представляет собой список с двумя столбцами. В левой колонке вводится текст, отображаемый в раскрывающемся списке **Files of type** окна диалога. В правом столбце через точку с запятой записываются маски, на основании которых выполняется фильтрация файлов.

Шаг 32. Установите в компоненте **OpenDialog** фильтры, как показано на рисунке 8.31.

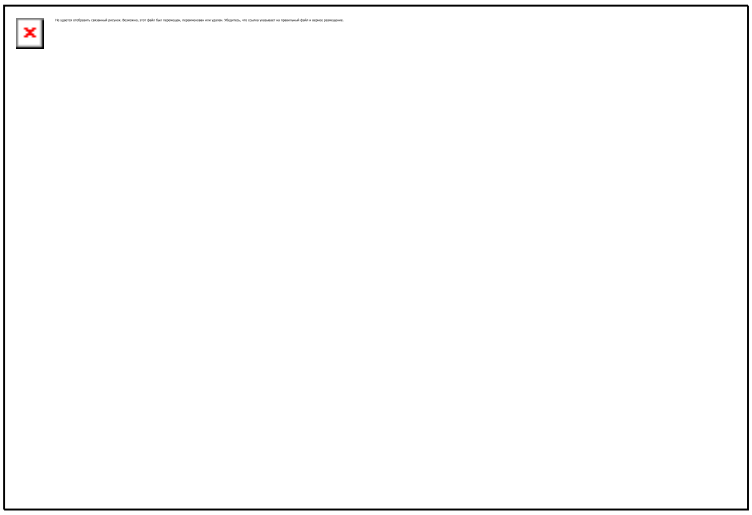


Рисунок 8.31. Окно для редактирования фильтров — Filter Editor

Шаг 33. Аналогичным образом установите фильтры в компоненте **SaveDialog**. Самый простой и быстрый способ в данном случае — скопировать текст свойства Filter из компонента OpenDialog в компонент SaveDialog через буфер обмена (результат показан на рисунке 8.32):

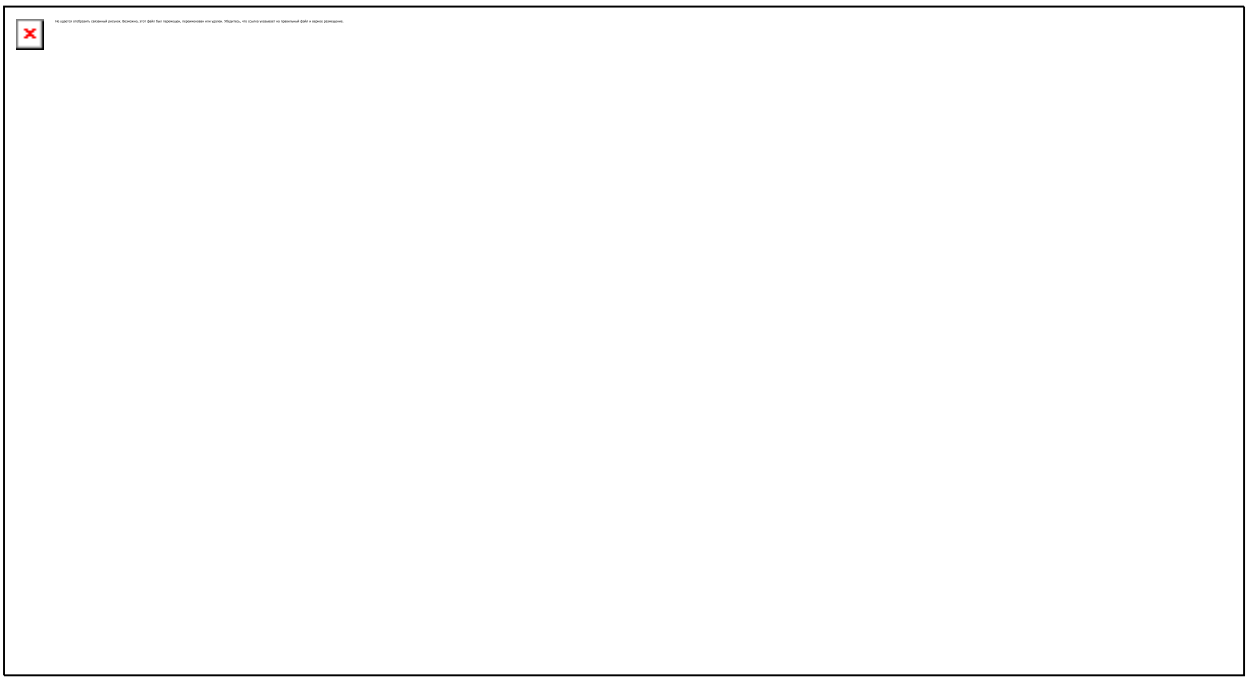


Рисунок 8.32. Фильтры для окна Save скопированы из окна Open

Компоненты **OpenDialog** и **SaveDialog** имеют большое количество булевских параметров, организованных в виде составных свойств **Options** и **OptionsEx**. Эти параметры влияют на то, как окно диалога выглядит и работает. Их смысл поясняет таблица 8.7.

Параметр	Описание
ofReadOnly	Если равно True, то переключатель Read-only в окне диалога включен.
ofOverwritePrompt	Если равно True, то пользователю выдается предупреждение при попытке сохранить файл с именем, которое уже существует.
ofHideReadOnly	Если равно True, то переключатель Read-only отсутствует в окне диалога.
ofNoChangeDir	Если равно True, то пользователь не сможет сменить каталог в окне диалога.
ofShowHelp	Если равно True, то в окне диалога присутствует кнопка Help.
ofNoValidate	Если равно True, то пользователь может вводить в имени файла любые символы, даже недопустимые.
ofAllowMultiSelect	Если равно True, то пользователь может выделить сразу несколько файлов.
ofExtensionDifferent	Этот параметр устанавливается после завершения диалога, если расширение в имени файла отличается от начального расширения.
ofPathMustExist	Если равно True, то пользователь не сможет ввести

	для файла несуществующий маршрут.
ofFileMustExist	Если равно True, то пользователь не сможет ввести имя несуществующего файла.
ofCreatePrompt	Если равно True и пользователь вводит имя несуществующего файла, то пользователю задается вопрос, желает ли он создать новый файл с таким именем.
ofShareAware	Если равно True, то ошибки одновременного доступа к файлу со стороны нескольких приложений игнорируются.
ofNoReadOnlyReturn	Если равно True, то пользователь не сможет ввести файл с атрибутом read-only (только для чтения).
ofNoTestFileCreate	Если равно True, то проверка на возможность записи в каталог не выполняется.
ofNoNetworkButton	Если равно True, то кнопка Network отсутствует в окне диалога. Этот параметр работает только в паре с параметром ofOldStyleDialog.
ofNoLongNames	Если равно True, то длинные имена файлов запрещены.
ofOldStyleDialog	Если равно True, то окно диалога отображается в старом стиле Windows 3.1.
ofNoDereferenceLinks	Если равно True, то ярлыки к каталогам трактуются как обычные файлы. В противном случае они трактуются как каталоги.
ofEnableIncludeNotify	Если равно True, то при формировании списка отображаемых файлов происходит событие OnIncludeItem (для каждого файла). В обработчике этого события обычно выполняется дополнительная фильтрация файлов.
ofEnableSizing	Если равно значению True, то пользователь имеет возможность изменять размеры окна диалога.
ofDontAddToRecent	Если равно значению True, то файл не помещается в список последних открытых файлов.
ofShowHidden	Если равно True, то в окне показываются скрытые файлы (файлы с атрибутом Hidden).
ofExNoPlaceBar	Если равно True, то боковая панель не показывается в окне диалога. Флажок ofExNoPlaceBar относится к свойству OptionsEx.

*Таблица 8.7. Параметры компонентов **OpenDialog** и **SaveDialog***

Шаг 34. В нашем простом примере ограничимся тем, что установим в компоненте **SaveDialog** параметр **ofOverwritePrompt** в значение **True** (см. табл. 6.6).

Заметим, что проверить работу компонентов **OpenDialog** и **SaveDialog** можно с помощью команды **Test Dialog**. Она находится в контекстном меню значка компонента в форме.

8.2.2. Отображение рисунков

Шаг 35. Ну вот, диалоговые компоненты настроены. Теперь нужен компонент, обеспечивающий отображение рисунков различных форматов. Такой компонент в среде Delphi есть, он называется **Image** и находится в палитре компонентов на вкладке **Additional** (рисунок 8.33). Выберите его из палитры и поместите на форму. Назовите новый компонент **Image**, а свойствам **Left** и **Top** установите значение 0.

Рисунок 8.33. Компонент Image

Характерные свойства компонента **Image** кратко описаны в таблице 8.8.

Свойство	Описание
AutoSize	Если равно значению **True**, то размеры компонента автоматически подгоняются под размеры рисунка.
Center	Центрирует рисунок в пределах компонента.
IncrementalDisplay	Обеспечивает постепенное (по мере загрузки) отображение больших рисунков. Используется для устранения эффекта блокировки пользовательского ввода во время отображения рисунка.
Picture	Содержит рисунок, отображаемый в области компонента. Свойство **Picture** является объектом класса **TPicture** и может хранить точечный рисунок (bitmap), метафайл (metafile), значок (icon).
Proportional	Если равно значению True, то при масштабировании сохраняется пропорция между вертикальным и горизонтальным размерами рисунка.
Stretch	Если равно значению True, то рисунок масштабируется так, чтобы его размеры совпадали с размерами компонента. Масштабирование выполняется только для точечных рисунков и метафайлов.
Transparent	Если равно значению True, то фон рисунков становится прозрачным. Эффект появляется только после установки свойства Picture.
OnProgress	Происходит по мере выполнения длительных

	операций, например во время загрузки больших рисунков.

*Таблица 8.8. Основные свойства компонента **Image***

Компонент **Image** позволяет отображать рисунки разных форматов: точечные рисунки (BMP), значки (ICO), метафайлы (WMF, EMF). Сам рисунок хранится в свойстве **Picture**.

Шаг 36. Размеры установленного рисунка могут не совпадать с текущими размерами компонента. В этом случае лишняя часть изображения отсекается. Чтобы подогнать размеры компонента под размеры рисунка установите свойство **AutoSize** в значение True (рисунок 8.34). После этого при каждой установке свойства **Picture** размеры компонента (но не рисунка) будут изменяться автоматически.

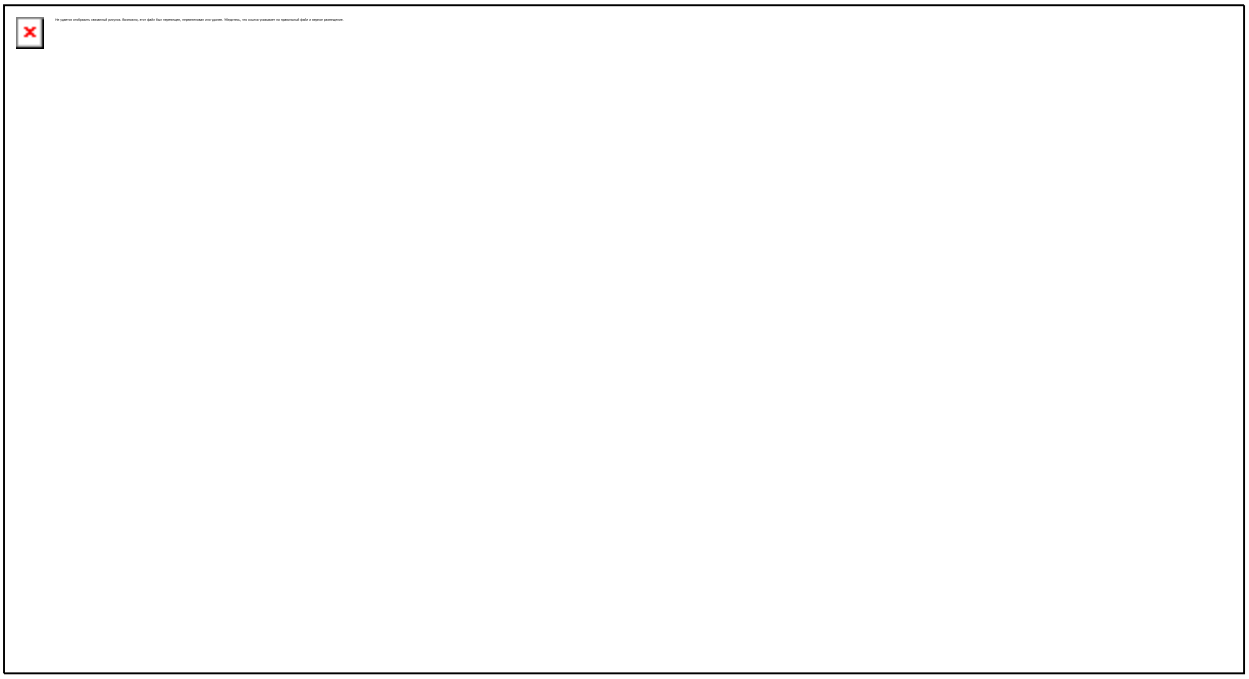


Рисунок 8.34. Свойство AutoSize в компоненте Image установлено в значение True

Бывает и обратная ситуация, когда нужно подогнать размеры рисунка под заданные размеры компонента. Для этого свойство **Stretch** устанавливается в значение True, а **AutoSize** — в значение False. Масштабирование целесообразно применять только для векторных изображений; для точечных рисунков оно не всегда дает приятный результат — начинает сказываться точечная природа изображения.

Сейчас компонент **Image** находится на своем месте и подготовлен к работе (свойство **AutoSize** имеет значение True). Рассмотрим, как осуществляется загрузка и сохранение рисунка по командам меню **Open...** и **Save As...** .

Шаг 37. В исходном тексте уже имеется недописанный обработчик команды **Open...** . В нем нужно вызвать стандартное диалоговое окно открытия файла и загрузить рисунок в том случае, если пользователь ввел в этом окне имя файла:

```
procedure TPictureForm.OpenMenuItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    Image.Picture.LoadFromFile(OpenDialog.FileName);
    EnableCommands(True);
    NormalSizeMenuItem.Click;
  end;
end;
```

В данном обработчике обратите внимание на вызов метода **Click** у компонента **NormalSizeItem**. Он имитирует выбор пункта меню **Normal Size**, чтобы сразу после загрузки рисунок имел нормальный размер.

Шаг 38. Пункт меню **Save As...** еще не имеет обработчика события **OnClick**, поэтому вам придется его создать (напомним, что это делается в окне свойств на вкладке **Events**). Обработка команды **Save As...** состоит в вызове стандартного диалогового окна Save с последующем сохранением рисунка в файле:

```
procedure TPictureForm.SaveAsMenuItemClick(Sender: TObject);
begin
  if SaveDialog.Execute then
    Image.Picture.SaveToFile(SaveDialog.FileName);
end;
```

Шаг 39. Чтобы наш пример, наконец, заработал, осталось дописать несколько обработчиков событий. В обработчике команды меню **Close** добавим операторы удаления рисунка из компонента **Image** и уменьшения размеров компонента до нуля, чтобы в отсутствие рисунка компонент не занимал места на форме:

```
procedure TPictureForm.CloseMenuItemClick(Sender: TObject);
begin
  with Image do
  begin
    Picture := nil;
    Width := 0;
    Height := 0;
  end;
  NormalSizeMenuItem.Click;
  EnableCommands(False);
end;
```

Шаг 40. Еще остались незавершенными обработчики команд меню **Half Size**, **Normal Size** и **Double Size**, которые тоже нужно доработать. С ними вы легко разберетесь:

```
procedure TPictureForm.HalfSizeMenuItemClick(Sender: TObject);
begin
  HalfSizeMenuItem.Checked := True;
  HalfSizePopupItem.Checked := True;
  with Image do
  begin
    AutoSize := False;
    Width := Picture.Width div 2;
    Height := Picture.Height div 2;
    Stretch := True;
  end;
end;

procedure TPictureForm.NormalSizeMenuItemClick(Sender: TObject);
begin
  NormalSizeMenuItem.Checked := True;
  NormalSizePopupItem.Checked := True;
  Image.AutoSize := True; // восстановить нормальные размеры компонента
end;

procedure TPictureForm.DoubleSizeMenuItemClick(Sender: TObject);
begin
  DoubleSizeMenuItem.Checked := True;
  DoubleSizePopupItem.Checked := True;
  with Image do
  begin
    AutoSize := False;
    Width := Picture.Width * 2;
    Height := Picture.Height * 2;
    Stretch := True;
  end;
end;
```

В первом приближении программа для просмотра графических файлов готова. Выполните компиляцию программы и проверьте ее работоспособность. Например, откройте файл Chemical.bmp из стандартной коллекции изображений среды Delphi (C:\Program Files\Common Files\Borland Shared\Images\Splash\256Color). Вашему взору предстанет следующая картина (рисунок 8.35):

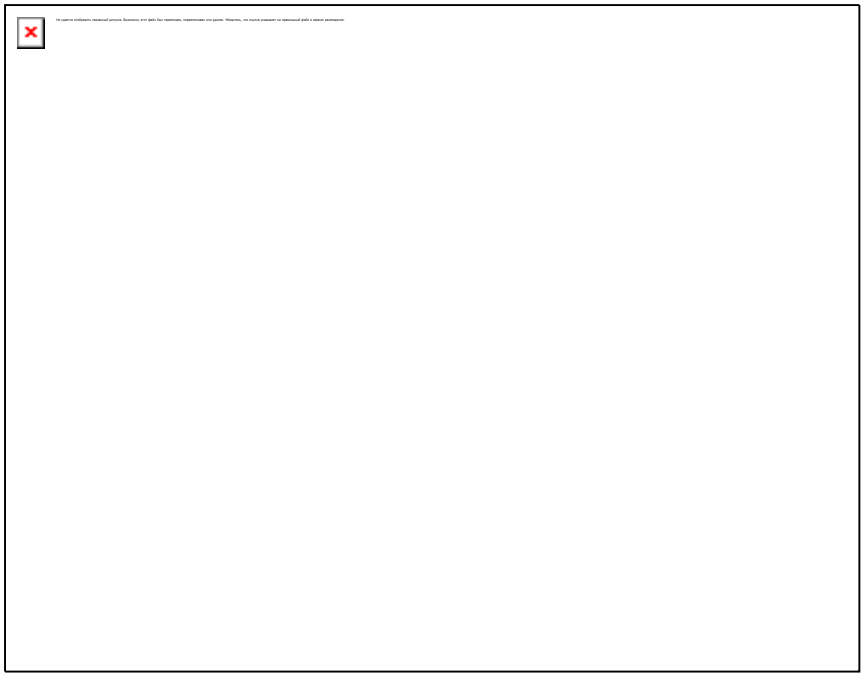


Рисунок 8.35. Программа для просмотра графических файлов в работе

Внимание! В каталоге C:\Program Files\Common Files\Borland Shared\Images вы найдете для своих приложений много полезных и красивых точечных рисунков, значков, курсоров. Если вы еще не исследовали этот каталог, то сделайте это с помощью своей программы.

Экспериментируя с приложением, обратите внимание на способность формы прокручивать рисунки, которые в ней не умещаются. Это явление называется *автоматической прокруткой*. Автоматическая прокрутка не требует никаких забот со стороны программиста и очень хорошо выручает в тех случаях, когда изображение превышает размеры рабочей области формы.

8.3. Строка состояния

8.3.1. Создание строки состояния

Строка состояния (status bar) — это панель в нижней части окна, предназначенная для вывода вспомогательной информации: параметров документа, с которым работает пользователь, подсказок к пунктам меню и др. В среде Delphi она организуется с помощью компонента **StatusBar**, расположенного в палитре компонентов на вкладке **Win32** (рисунок 8.36).

Шаг 41. Поместите компонент на форму и дайте ему имя **StatusBar**.

Рисунок 8.36. Компонент StatusBar

Таблица 8.9 знакомит вас с основными свойствами компонента **StatusBar**. Когда вы изучите компонент, она пригодится вам в качестве справочника, а сейчас просто окиньте ее взглядом и двигайтесь дальше.

Свойство	Описание
Action	Задает так называемую команду, которая будет выполняться по щелчку на строке состояния. Весь список команд содержится в компоненте **ActionList** (см. параграф 8.6).
Align	Способ выравнивания строки состояния в пределах содержащего компонента (например, формы или панели).
AutoHint	Если равно значению True, то текст строки состояния автоматически устанавливается равным текущей всплывающей подсказке.
BorderWidth	Величина отступа от границ компонента до границ информационных панелей.
Panels	Информационные панели, отображаемые на строке состояния.
SimplePanel	Если равно значению True, то вместо информационных панелей на строке состояния отображается одна простая строка текста, хранящаяся в свойстве SimpleText.
SimpleText	Строка текста, отображаемая в строке состояния,

	когда свойство SimplePanel равно значению True.
SizeGrip	Если равно значению True, то в правом нижнем углу строки состояния рисуется "гармошка", работающая как размерная рамка.
UseSystemFont	Если равно значению True, то используется стандартный шрифт системы.
OnCreatePanelClass	Позволяет создавать свои собственные панели на базе класса TStatusPanel.
OnDrawPanel	Происходит при рисовании панели, если свойство **Style** содержит значение **psOwnerDraw**.

*Таблица 8.9. Важнейшие свойства и события компонента **StatusBar***

Как только вы добавили на форму строку состояния, она тут же прижалась к нижнему краю формы и растянулась по всей ее ширине (см. рис. ниже). Какая сообразительная! А ну-ка изменим ширину формы. Ба! Строка состояния тоже корректирует свою ширину и всегда занимает всю нижнюю часть формы (рисунок 8.37).

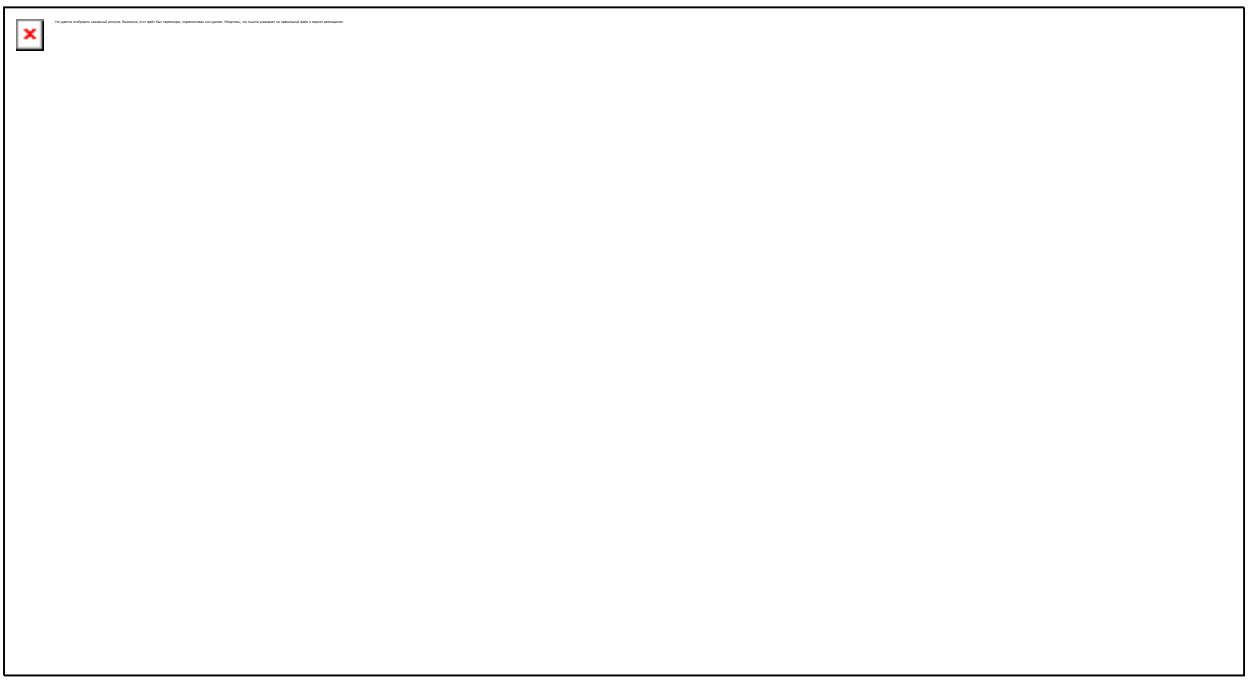


Рисунок 8.37. Строка состояния автоматически прижимается к нижнему краю формы

Такое поведение обеспечивает свойство **Align**, которое в компоненте **StatusBar** изначально содержит значение **alBottom**. Свойство **Align** есть во многих визуальных компонентах. С его помощью вы можете заставить компонент подгонять свои размеры и положение при изменении размеров своего владельца (формы или компонента, на котором он находится). Возможные значения свойства **Align** описаны в таблице 8.10.

Значение	Описание
alNone	Позиция и размеры компонента остаются неизменными в пределах владельца.

alTop	Компонент прижимается к верхнему краю владельца и растягивается по всей его ширине.
alBottom	Компонент прижимается к нижнему краю владельца и растягивается по всей его ширине.
alLeft	Компонент прижимается к левому краю владельца и растягивается по всей его высоте.
alRight	Компонент прижимается к правому краю владельца и растягивается по всей его высоте.
alClient	Компонент подгоняется под размеры владельца.

*Таблица 8.10. Значения свойства **Align***

Принимая во внимание, что некоторые компоненты могут содержать другие компоненты, становится ясно, какую мощь таит в себе свойство **Align**, избавляя программистов от огромной работы по перерасчету координат компонентов при изменении размеров формы. Всегда помните об этой чудесной возможности и старайтесь использовать ее в полной мере.

Шаг 42. Вернемся к примеру и приспособим строку состояния для отображения размеров рисунка и имени файла, в котором рисунок хранится на диске. С этой целью разделим строку состояния на две информационные панели. Перейдите к окну свойств и в поле **Panels** щелкните кнопку с многоточием (либо в контекстном меню строки состояния выберите пункт **Panels Editor…**). Откроется специальное окно с заголовком **Editing StatusBar.Panels** для создания панелей в строке состояния (рисунок 8.38).

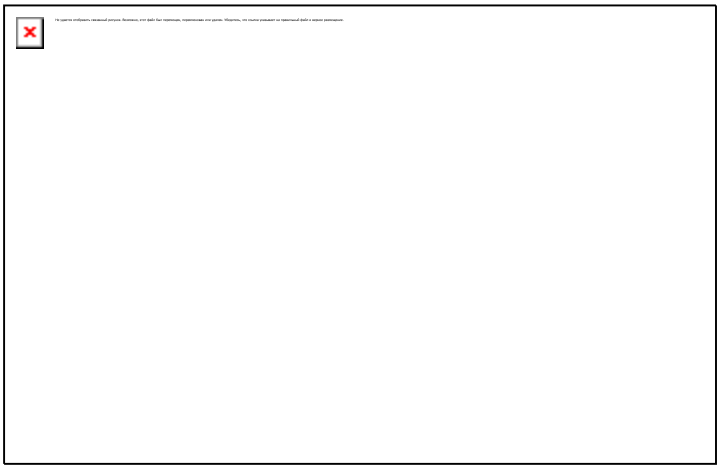


Рисунок 8.38. Окно для создания панелей в строке состояния

Шаг 43. В этом окне создаются, редактируются и удаляются панели строки состояния. Оно работает в паре с окном свойств, в котором настраиваются свойства отдельно взятой панели строки состояния. Нажатием кнопки **Add New** создайте первую панель и установите ее свойства так, чтобы она получилась шириной 70 пикселей (**Width** = 70), продавленной (**Bevel** = **pbLowered**) и с центрированным текстом (**Alignment** = **taCenter**). См. рисунок 8.39.

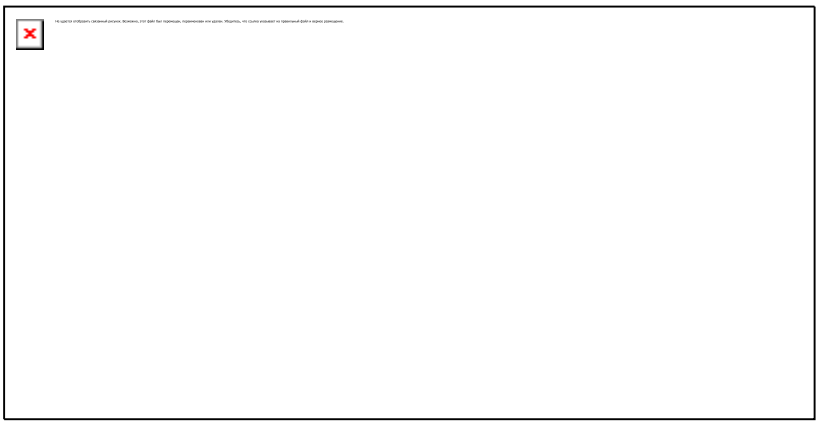


Рисунок 8.39. В строке состояния создана панель

В этой панели будут отображаться размеры рисунка. Аналогично создайте вторую панель (рисунок 8.40) неограниченной ширины (**Width** = -1), продавленной (**Bevel** = **pbLowered**) и с прижатым влево текстом (**Alignment** = **taLeftJustify**). В ней будет отображаться имя файла.

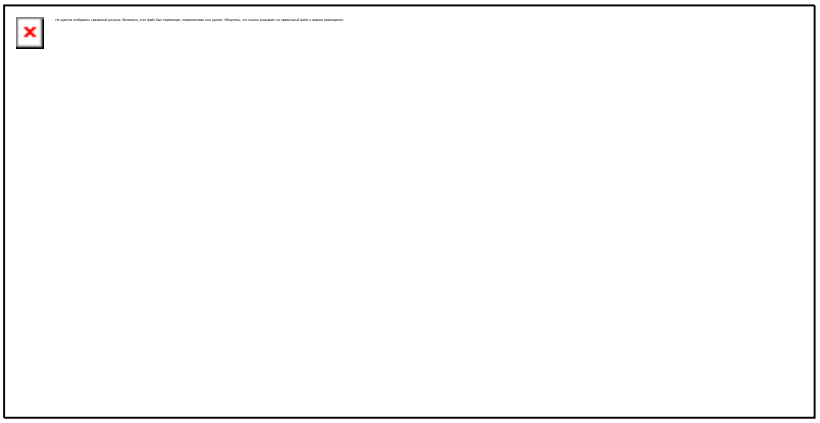


Рисунок 8.40. В строке состояния создана еще одна панель

После этого закройте окно **Editing StatusBar.Panels**.

Строка состояния создана и сейчас рассмотрим, как вывести в ней текст. Доступ к панелям обеспечивает свойство **Panels**. Оно содержит массив **Items**, элементами которого являются объекты-панели. Каждая панель имеет свойство **Text**, в котором хранится отображаемый на панели текст. Итак, установка содержимого строки состояния в нашем примере будет выглядеть так:

```
StatusBar.Panels.Items[0].Text :=
  Format('%d x %d', [Image.Picture.Width, Image.Picture.Height]);
StatusBar.Panels.Items[1].Text := OpenDialog.FileName;
```

Учитывая, что массив **Items** выступает главным свойством объекта **Panels**, эти операторы можно записать короче:

```
StatusBar.Panels[0].Text :=
  Format('%d x %d', [Image.Picture.Width, Image.Picture.Height]);
StatusBar.Panels[1].Text := OpenDialog.FileName;
```

Для вывода информации на первую панель (с индексом 0) мы воспользовались функцией **Format**, форматирующей строку. Первый параметр функции — это строка-шаблон, а второй — открытый массив с аргументами, подставляемыми вместо управляющих символов строки-шаблона.

Шаг 44. Обновление строки состояния удобно оформить в виде метода формы:

```
procedure TPictureForm.UpdateStatusBar;
begin
  if Image.Width <> 0 then
  begin
    StatusBar.Panels[0].Text := Format('%d x %d',
      [Image.Picture.Width, Image.Picture.Height]);
    StatusBar.Panels[1].Text := OpenDialog.FileName;
  end
  else // в компоненте Image нет рисунка
  begin
    StatusBar.Panels[0].Text := '';
    StatusBar.Panels[1].Text := '';
  end;
end;
```

Шаг 45. Вызовы метода **UpdateStatusBar** поместите в обработчики команд меню **Open...** и **Close**.

```
procedure TPictureForm.OpenMenuItemClick(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    Image.Picture.LoadFromFile(OpenDialog.FileName);
    EnableCommands(True);
    NormalSizeMenuItem.Click;
  end;
  UpdateStatusBar;
end;

procedure TPictureForm.CloseMenuItemClick(Sender: TObject);
begin
  with Image do
  begin
    Picture := nil;
    Width := 0;
    Height := 0;
  end;
  NormalSizeMenuItem.Click;
  EnableCommands(False);
  UpdateStatusBar;
end;
```

Наконец выполните компиляцию приложения и проверьте, что строка состояния работает. Например, откройте файл Chemical.bmp, расположенный по маршруту C:\Program Files\Common Files\Borland Shared\Images\Splash\256Color. В строке состояния отобразятся размеры рисунка и путь к файлу.

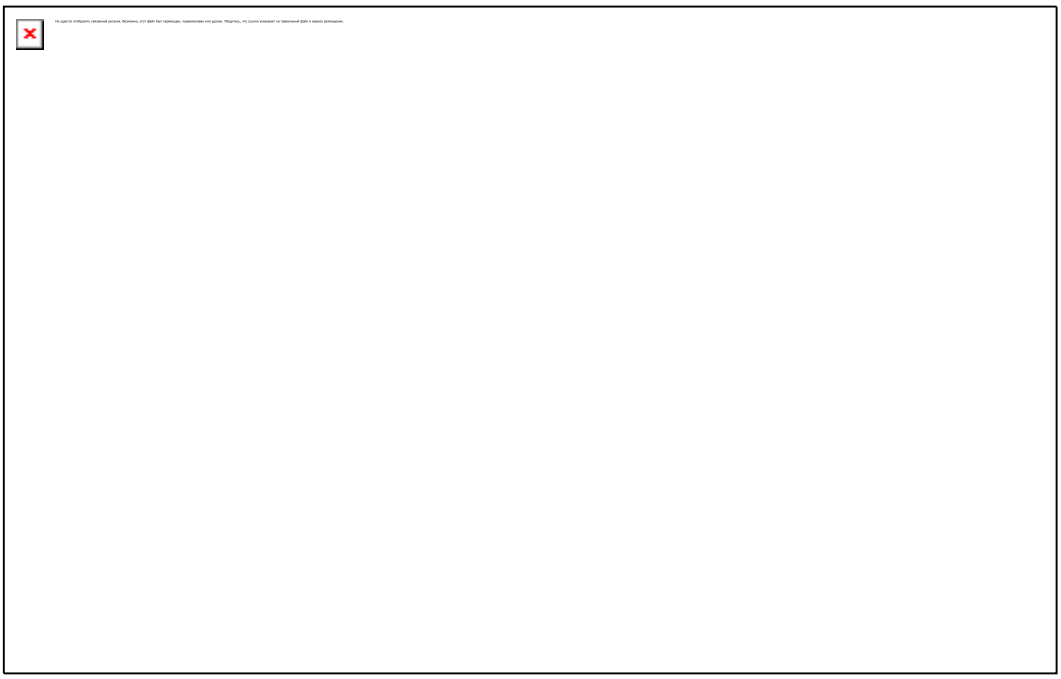


Рисунок 8.41. Программа для просмотра графических файлов теперь имеет строку состояния

8.3.2. Подсказки в строке состояния

Как вы хорошо знаете, строка состояния — это еще стандартное место отображения подсказок к пунктам меню. Сейчас самое время заняться этим вопросом. Вспомните, как работает строка состояния вашего любимого текстового процессора. Когда вы активизируете меню, строка состояния, состоящая из нескольких панелей, превращается в простую длинную панель и на ней отображается подсказка текущего пункта меню. Когда вы завершаете работу с меню (например, выбираете команду), строка состояния восстанавливает свой первоначальный вид.

Для того чтобы вы могли получить строку состояния с описанной выше логикой работы, в компоненте **StatusBar** предусмотрен режим отображения простого текста. Его обеспечивает булевское свойство **SimplePanel**. По умолчанию оно равно значению False и в строке состояния отображаются панели объекта **Panels**. Если установить свойство **SimplePanel** в значение True, то в строке состояния будет отображаться текст, хранящийся в свойстве **SimpleText**. Итак, задача состоит в том, чтобы при активизации меню записывать подсказку выбранного пункта в свойстве **SimpleText** и, в том случае если подсказка содержит текст, устанавливать свойство **SimplePanel** в значение **True**. Для решения этой задачи вы должны представлять механизм работы подсказок. Его суть состоит в следующем.

Каждый пункт меню имеет свойство **Hint** для хранения подсказки. Когда вы выделяете пункт меню с помощью мыши или клавиатуры, текст подсказки переписывается из пункта меню в объект **Application**, у которого тоже есть свойство **Hint**. При этом в объекте **Application** возникает событие **OnHint**. Все, что нам нужно — написать обработчик этого события, который отобразит значение свойства **Hint** объекта **Application** в строке состояния.

Объект **Application** не виден в окне свойств, но вы можете получить доступ к его событиям на этапе проектирования. Для этого в форму помещается специальный компонент **ApplicationEvents**, который вы найдете в палитре компонентов на вкладке **Additional** (рисунок 8.42).

Рисунок 8.42. Компонент ApplicationEvents

Шаг 46. Поместите на форму компонент **ApplicationEvents**. Дайте ему имя **ApplicationEvents**. Обратите внимание, что у этого компонента всего два свойства: **Name** и **Tag**. Это не удивительно, так как основное назначение компонента — представить события объекта **Application** (таблица 8.11).

Событие	Описание
OnActionExecute	Происходит при выполнении любой команды в компоненте **ActionList** (см. параграф 8.6).
OnActionUpdate	Происходит во время простоя программы для обновления состояния команд в компоненте **ActionList** (см. параграф 8.6).
OnActivate	Происходит, когда приложение получает активность, т.е. когда пользователь переключается на него с другого приложения.
OnDeactivate	Происходит, когда приложение теряет активность, т.е. когда пользователь переключается с него на другое приложения.
OnException	Происходит, когда в приложении возникает необработанная исключительная ситуация. По умолчанию обработчик этого события вызывает метод **ShowException** для отображения окна сообщений с пояснением причины ошибки. Вы можете изменить реакцию на событие OnException, переписав его обработчик.
OnHelp	Происходит, когда пользователь вызывает справку.
OnHint	Происходит, когда курсор мыши наводится на компонент, содержащий всплывающую подсказку.
OnIdle	Периодически происходит во время простоя программы.
OnMessage	Происходит при получении программой сообщения операционной системы Windows.
OnMinimize	Происходит, когда пользователь сворачивает приложение.
OnRestore	Происходит, когда пользователь восстанавливает свернутое приложение.
OnSettingChange	Происходит при изменении настроек операционной

	системы, например, настроек экрана или региональных настроек.
OnShortCut	Происходит при нажатии клавиш на клавиатуре (еще до того, как в форме происходит событие **OnKeyDown**).
OnShowHint	Происходит непосредственно перед появлением любой всплывающей подсказки.

Таблица 8.11. События компонента ApplicationEvents

Шаг 47. В окне свойств переключитесь на вкладку **Events**, найдите событие **OnHint** и создайте следующий обработчик:

```
procedure TPictureForm.ApplicationEventsHint(Sender: TObject);
begin
  with StatusBar do
  begin
    SimpleText := Application.Hint;
    SimplePanel := SimpleText <> '';
  end;
end;
```

Шаг 48. Теперь в свойстве **Hint** каждого пункта меню впишите угодную вам строку-подсказку (рисунок 8.43).

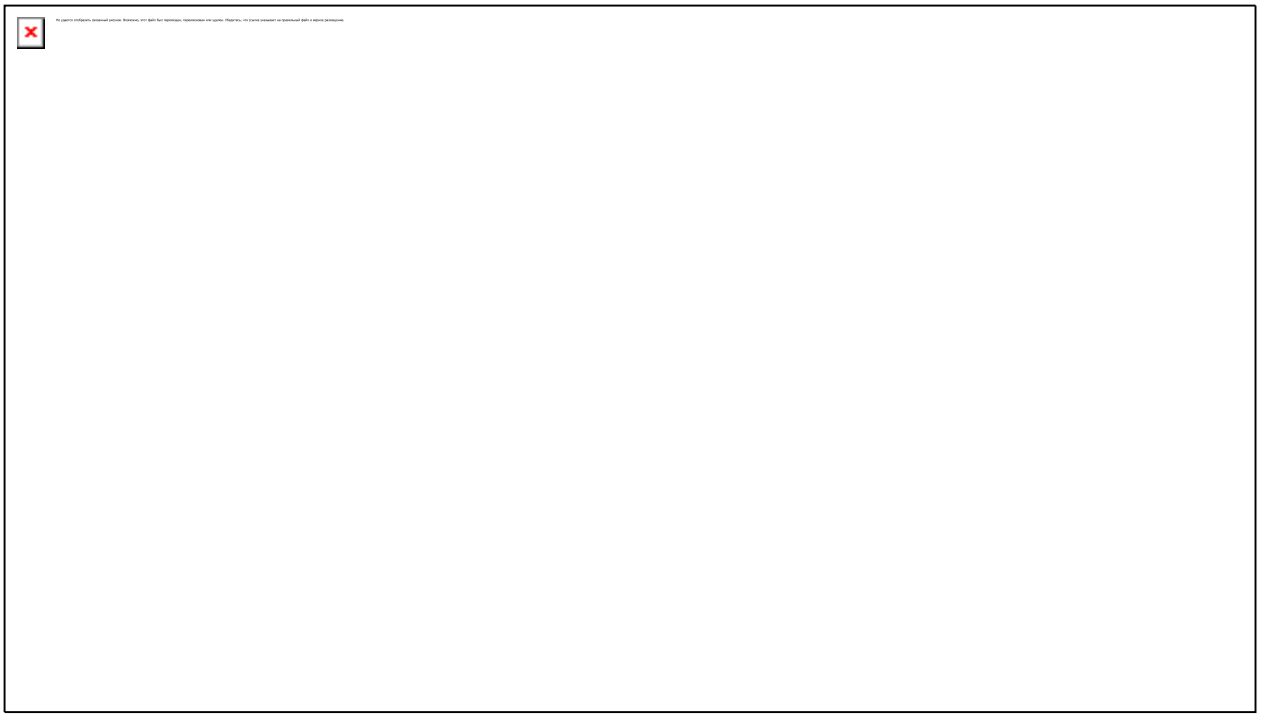


Рисунок 8.43. Подсказка для пункта меню

Шаг 49. Выполните компиляцию и запустите программу. Проверьте работу механизма подсказок в строке состояния (рисунок 8.44).

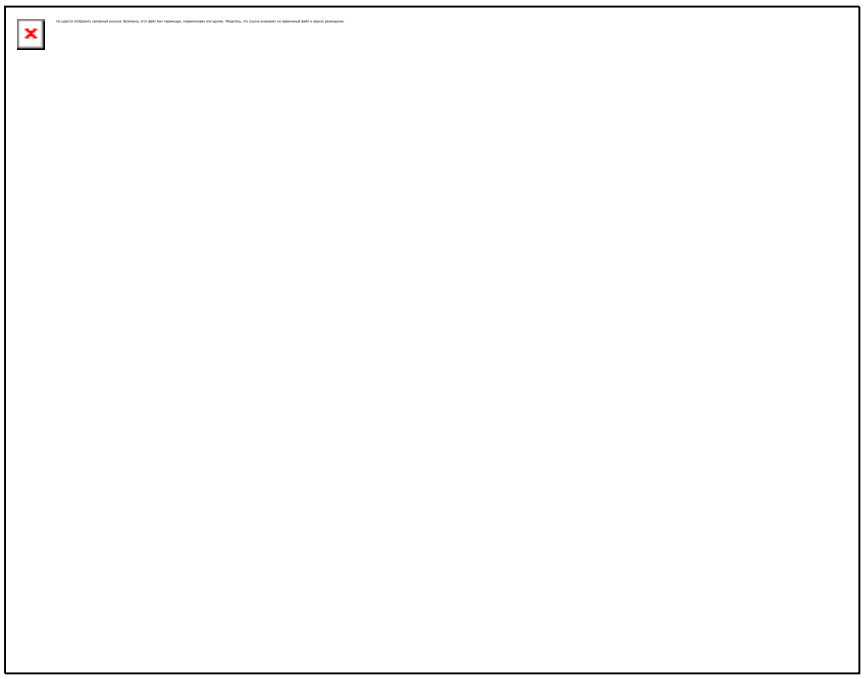


Рисунок 8.44. Программа для просмотра графических файлов теперь показывает подсказки для пунктов меню в строке состояния

Если критически взглянуть на нынешний вариант программы, то среди прочих мелких замечаний выделяется существенный недостаток: форма неправильно прокручивает свое содержимое, когда размеры рисунка превышают размеры формы. Дело в том, что в прокрутке участвует и строка состояния, а этого быть не должно. Строка состояния должна оставаться на своем месте, прижимаясь к нижнему краю формы. Чтобы разобраться с этой проблемой, читайте следующий параграф.

8.4. Прокрутка

8.4.1. Прокрутка рабочей области формы

На практике часто бывает, что отображаемая информация не умещается на форме целиком (даже если форма раскрыта на весь экран). Например, в нашем примере можно загрузить рисунок, размеры которого превосходят размеры формы (и даже всего экрана) в несколько раз. Лучшее, что можно предпринять в таком случае, — это организовать *прокрутку* (scrolling) рисунка внутри формы.

В *области прокрутки* видна только часть всей картины. Доступ к скрытым частям происходит с помощью *полос прокрутки*. Щелчок мыши на стрелке полосы прокрутки сдвигает изображение на одну "информативную строку", а щелчок мыши на самой линейке прокрутки (но не на бегунке) сдвигает изображение на одну "информативную страницу" (понятия строки и страницы существуют для прокрутки и по вертикали, и по горизонтали). Перемещая бегунок, можно быстро прокрутить изображение на любое число информативных строк или страниц.

Форма имеет встроенную поддержку прокрутки, благодаря чему реализуется просмотр содержимого формы при любом изменении ее размеров. Когда размеры или координаты компонентов превышают размеры формы, форма создает полосы прокрутки и пользователь получает возможность прокручивать изображение. Встроенные в форму полосы прокрутки представлены составными свойствами **HorzScrollBar** (горизонтальная полоса прокрутки) и **VertScrollBar** (вертикальная полоса прокрутки). Они кратко описаны в таблице 8.12.

Свойство	Описание

ButtonSize	Размер кнопок со стрелками.
Increment	Величина “информативной строки” в пикселах.
Margin	Минимальный отступ прокручиваемых элементов от края области прокрутки.
Position	Позиция бегунка на полосе прокрутки.
Range	Виртуальный размер области прокрутки.
Size	Ширина полосы прокрутки.
Smooth	Значение True указывает, что значение свойства **Increment** должно автоматически рассчитываться компонентом.
Style	Стиль полосы прокрутки: **ssRegular** — обычный рельефный, **ssFlat** — плоский, **ssHotTrack** — плоский с подсветкой при наведении указателя мыши.
ThumbSize	Размер бегунка.
Tracking	Если равно значению True, то прокрутка изображения происходит по мере передвижения бегунка.
Visible	Определяет, видна ли полоса прокрутки пользователю.

*Таблица 8.12. Составные свойства **HorzScrollBar** и **VertScrollBar***

Наибольший интерес представляют вложенные свойства **Tracking** и **Increment**. Установка булевского свойства **Tracking** в значение True обеспечивает прокрутку изображения по мере передвижения бегунка с помощью мыши. Свойство **Increment** задает величину "информативной строки" в пикселях. Уменьшив это значение до 1, вы получите более плавную прокрутку.

8.4.2. Отдельная область прокрутки

Как ни крути, а форма не позволяет организовать прокрутку в отдельной своей части. Например, в приложении для просмотра графических файлов хотелось бы организовать прокрутку рисунка, но так, чтобы строка состояния в прокрутке не участвовала. Форма этого сделать не позволяет. Здесь на помощь приходит компонент **ScrollBox**, представляющий собой *отдельную область прокрутки*. Он расположен в палитре компонентов на вкладке **Additional** (рисунок 8.45).

Рисунок 8.45. Компонент ScrollBox

Таблица 8.13 содержит краткую характеристику его отличительных свойств.

Свойство	Описание
Align	Способ выравнивания области прокрутки в пределах владельца.

AutoScroll	Если равно значению True, полосы прокрутки появляются и скрываются автоматически по мере необходимости.
AutoSize	Режим автоматического изменения размеров области прокрутки в зависимости от размеров и положения внутренних компонентов.
BevelEdges	Вложенные свойства **beLeft**, **beTop**, **beRight** и **beBottom** определяют видимость соответственно левой, верхней, правой и нижней сторон рельефной рамки.
BevelInner	Внутренний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelKind	Вид рельефной рамки: **bkNone** — рамки нет, **bkTile** — рамка с четкими скосами, **bkSoft** — рамка со сглаженными скосами, **bkFlat** — плоская рамка (без скосов).
BevelOuter	Внешний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelWidth	Ширина скосов рельефной рамки.
BorderStyle	Определяет, имеет ли область прокрутки рамку.
DockSite	Определяет, используется ли область прокрутки для стыковки других компонентов.
HorzScrollBar	Определяет параметры и поведение горизонтальной полосы прокрутки (см. табл. 6.10).
VertScrollBar	Определяет параметры и поведение вертикальной полосы прокрутки (см. табл. 6.10).
OnGetSiteInfo	Происходит, когда у компонента запрашивается место для стыковки.

*Таблица 8.13. Важнейшие свойства компонента **ScrollBox***

Компонент **ScrollBox** служит контейнером для других компонентов и обеспечивает их прокрутку внутри себя. Давайте поместим на него рисунок (компонент **Image**), а область прокрутки расположим между меню и строкой состояния. В результате большие рисунки будут прокручиваться уже не формой, а компонентом **ScrollBox** и строка состояния останется на своем месте, прижатой к нижнему краю формы.

Шаг 50. Выделите на форме компонент **Image** и временно удалите его в буфер (команда меню **Edit | Cut**). Теперь опустите на форму компонент **ScrollBox**, выбрав его из палитры компонентов. Назовите новый компонент **ScrollBox** и подгоните его под всю незанятую область формы, установив свойство **Align** в значение **alClient** (рисунок 8.46).

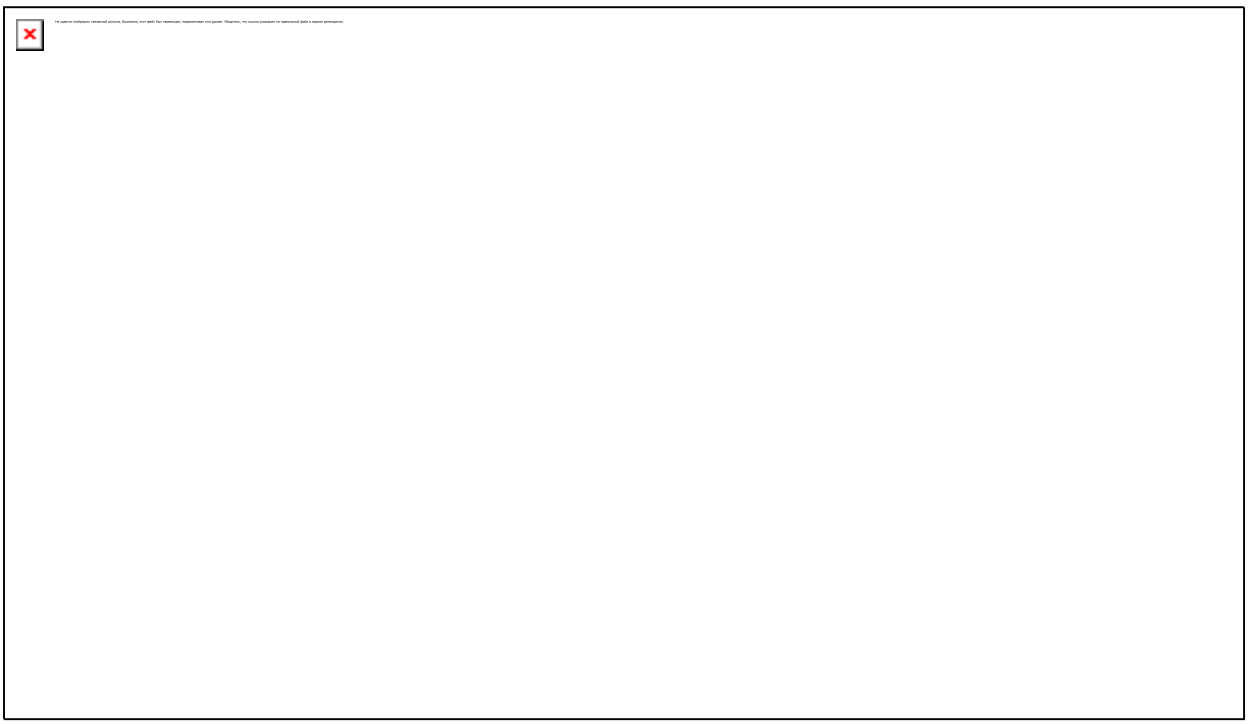


Рисунок 8.46. Свойство Align обеспечивает подгонку компонента под размеры контейнера

Шаг 51. А сейчас переключитесь на форму (так, чтобы компонент **ScrollBox** остался выделенным) и вставьте из буфера обмена компонент **Image** (команда меню **Edit | Paste**). Убедитесь, что он находится в левом верхнем углу области прокрутки.

Готово. Выполните компиляцию и запустите приложение, загрузите в него какой-нибудь рисунок из каталога C:\Program Files\Common Files\Borland Shared\Images\Splash\256Color. Увеличивая и уменьшая окно, понаблюдайте за тем, как появляются и исчезают полосы прокрутки между меню и строкой состояния (рисунок 8.47). Обратите внимание, что величина бегунков на полосах прокрутки зависит от соотношения видимой части и всего изображения. Это работает компонент **ScrollBox**. Правда, здорово! А самое главное — быстро и без единой строчки кода.

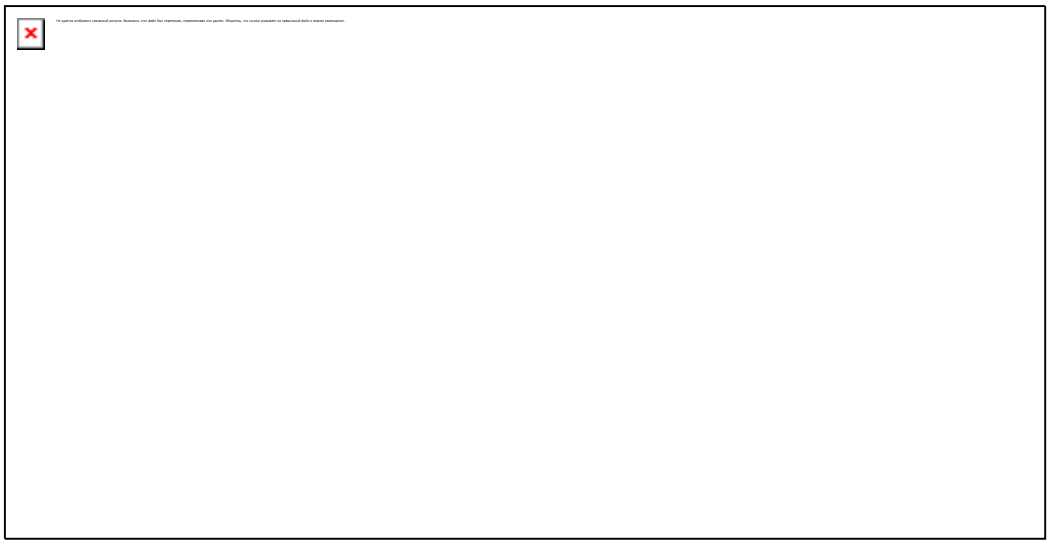


Рисунок 8.47. Программа для просмотра графических файлов теперь умеет прокручивать не уместившееся внутри окна изображение

8.4.3. Полосы прокрутки

Коль уж речь зашла о прокрутке, сделаем небольшое отступление и скажем пару слов о компоненте **ScrollBar**. Вы, наверное, еще раньше заметили его в палитре компонентов на вкладке **Standard** и сейчас не совсем понимаете, для чего он нужен (рисунок 8.48).

Рисунок 8.48. Компонент ScrollBar

ScrollBar — это отдельная полоса прокрутки без области прокрутки. Ее согласованная работа с другими компонентами обеспечивается программистом. Для этого в компоненте **ScrollBar** предусмотрено событие **OnScroll**, в ответ на которое и нужно выполнять необходимые действия. Должны вам сообщить, что компонент **ScrollBar** не имеет никакого отношения ни к форме, ни к компоненту **ScrollBox**. И вообще, он используется редко. Авторы этой книги будут вам признательны, если вы сообщите им о применении компонента **ScrollBar** в реальной задаче.

Следуя традиции данной книги, мы приводим табличное описание свойств компонента (таблица 8.14).

Свойство	Описание
Kind	Вид полосы прокрутки: горизонтальная или вертикальная.
LargeChange	Величина "информативной страницы".
Min, Max	Начальная и конечная виртуальные позиции на полосе прокрутки.
Position	Позиция бегунка на полосе прокрутки.
SmallChange	Величина "информативной строки".
OnChange	Происходит при изменении значения свойства **Position**. Если значение свойства Position изменяется при перемещении пользователем бегунка, то событие OnChange происходит сразу после события **OnScroll**.
OnScroll	Происходит при перемещении бегунка.

*Таблица 8.14. Важнейшие свойства и события компонента **ScrollBar***

Рисунок 8.49 наглядно поясняет смысл свойств **LargeChange** и **SmallChange**.

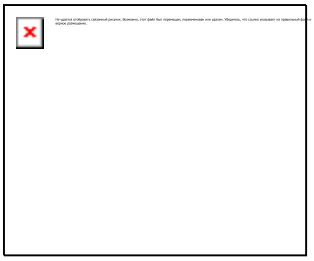


Рисунок 8.49. Свойства LargeChange и SmallChange применяются при расчете величины прокрутки

Ну вот вы и разобрались с прокруткой. Уверены, что вам понравилось, как она реализована в среде Delphi. Действительно, компонентное программирование. Взял компонент **ScrollBox**, поместил на форму, набросал в него других компонентов — и готово. А теперь пора засучить рукава, ибо вас ждет самая увлекательная часть этой главы — проектирование панели инструментов.

8.5. Панель инструментов

Панель инструментов (toolbar) — это расположенная под главным меню панель с кнопками, раскрывающимися списками, переключателями и другими компонентами. Компоненты панели инструментов, как правило, дублируют часто используемые команды меню.

8.5.1. Панель

Для создания панели инструментов в среде Delphi существует компонент **ToolBar**, расположенный в палитре компонентов на вкладке **Win32**.

Рисунок 8.50. Компонент ToolBar

Шаг 52. Активизируйте форму и поместите на нее компонент **ToolBar**. Дайте новому компоненту имя **ToolBar** (рисунок 8.51).

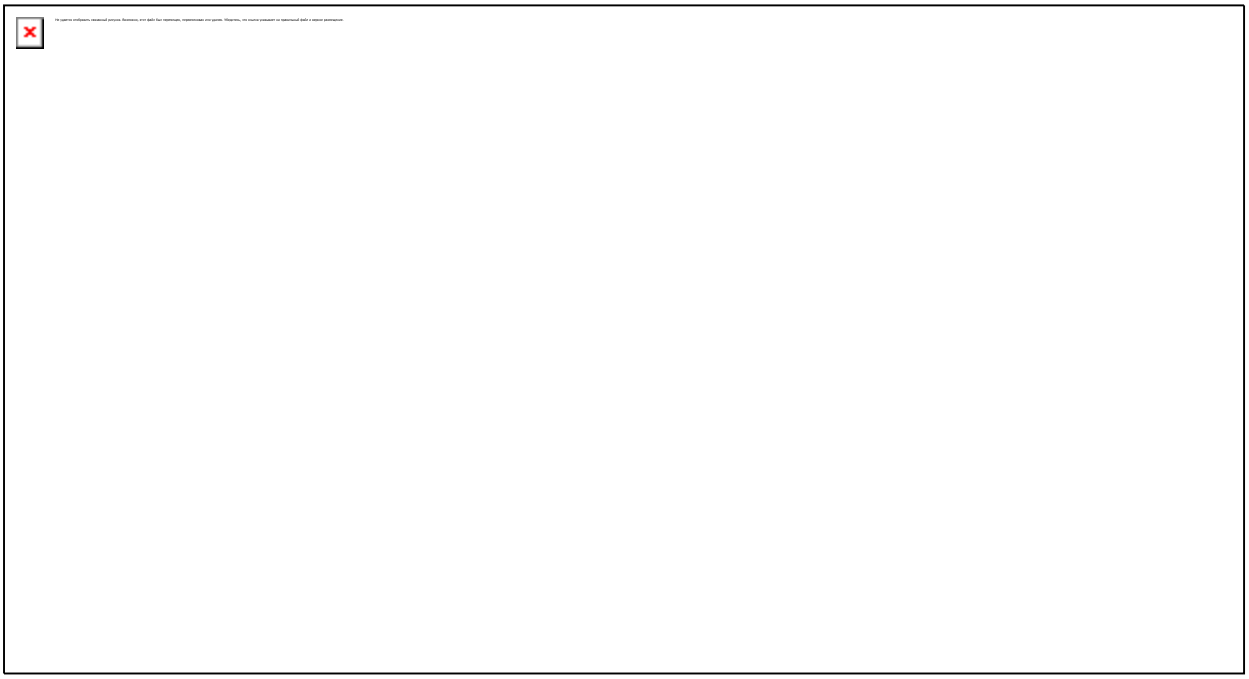


Рисунок 8.51. Панель инструментов оказалась в области прокрутки

Внимательный читатель, наверное, уже обратил внимание, что компонент **ToolBar** попал в область прокрутки (внутрь компонента **ScrollBox)**, и поэтому будет прокручиваться вместе с рисунком. Нам нужно вынести компонент **ToolBar** из области прокрутки и поместить его прямо в форму. Для этого воспользуемся окном **Object TreeView**.

Шаг 53. Перейдите к окну **Object TreeView** и найдите в нем компонент **ToolBar**. Захватите его с помощью мыши и перетащите к элементу **PictureForm** в этом же окне (рисунок 8.52).

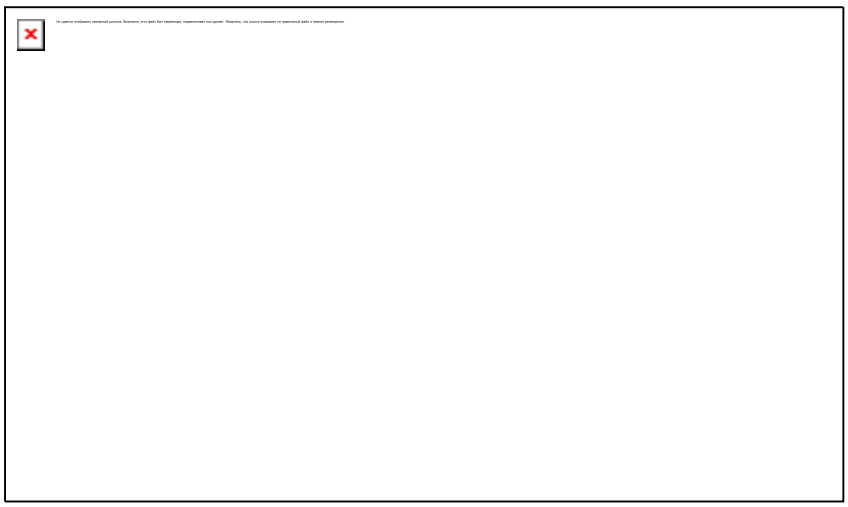


Рисунок 8.52. Буксировка в окне Object TreeView позволяет быстро перенести компонент с одной панели на другую

Теперь компонент **ToolBar** находится именно там, где нужно (рисунок 8.53):

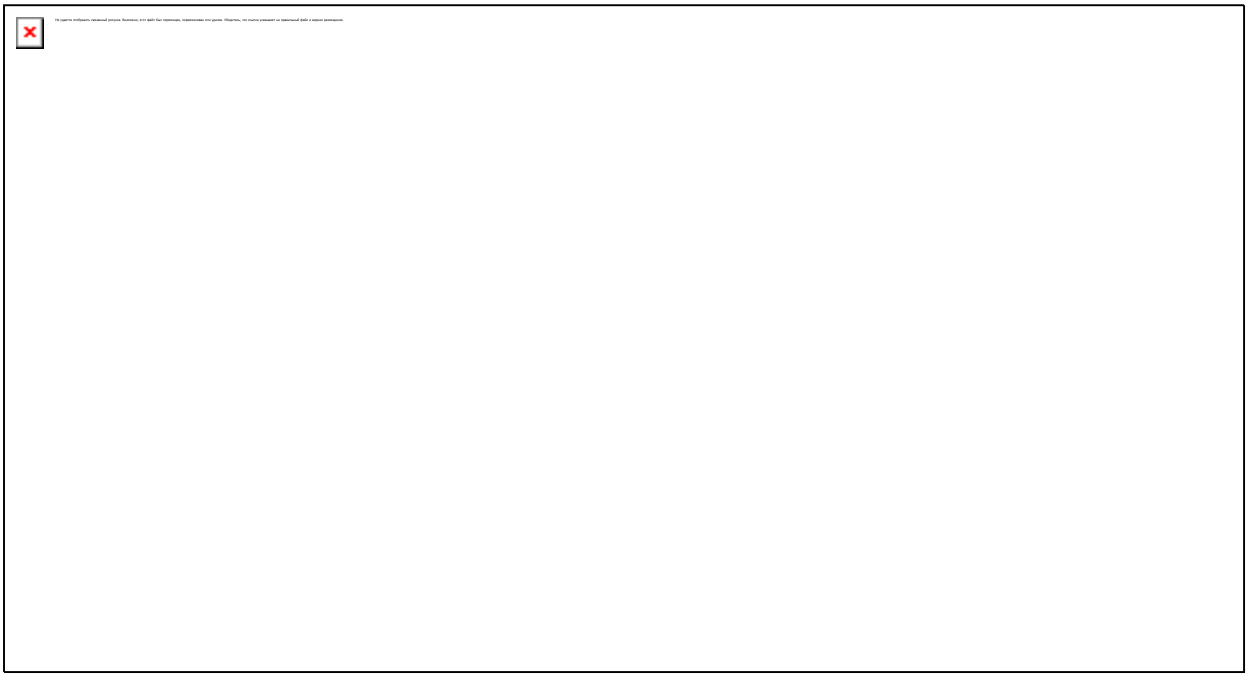


Рисунок 8.53. Панель инструментов вынесена за пределы области прокрутки

Между прочим, если вы сразу хотите поместить компонент на другой компонент, закрытый от вашего взора, выберите первый компонент в палитре компоненте и щелкните второй компонент в окне **Object TreeView** (рисунок 8.54):

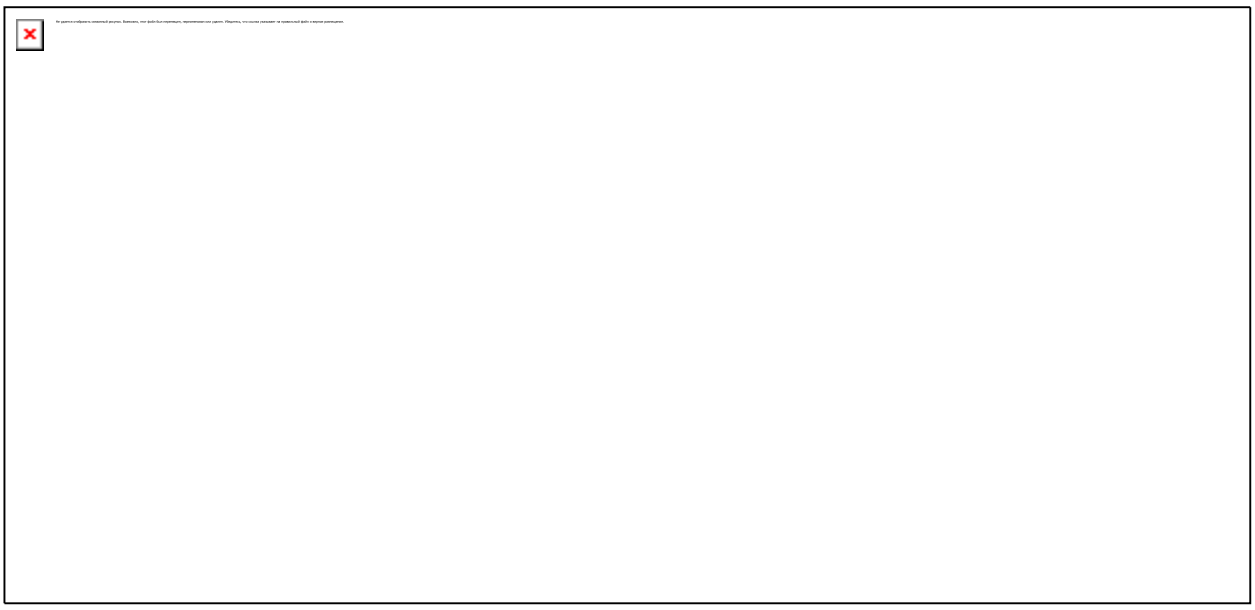


Рисунок 8.54. Размещение компонента сразу внутри нужного компонента с помощью окна Object TreeView

Шаг 54. В окне свойств установите свойство **AutoSize** в значение True. После этого панель инструментов будет автоматически подгонять свои размеры в зависимости от размеров и количества размещенных на ней компонентов.

Основу для размещения кнопок вы создали и в качестве передышки мы предлагаем вам пробежаться по наиболее важным свойствам компонента **ToolBar** и поэкспериментировать с их значениями (таблица 8.15).

Свойство	Описание
AutoSize	Если равно значению True, то панель автоматически изменяет свою высоту в зависимости от размеров размещенных на ней компонентов.
BorderWidth	Величина отступа от границ компонента до кнопок.
ButtonWidth, ButtonHeight	Ширина и высота кнопок на панели инструментов.
Customizable	Если равно значению True, то пользователь во время работы программы имеет возможность управлять расположением кнопок на панели инструментов. Удерживая клавишу Shift пользователь может захватить кнопку и перенести ее на нужное место, а двойным щелчком панели инструментов (но не ее кнопок!), пользователь может вызвать специальное окно настройки.
DisabledImages	Список значков, отображаемых на недоступных кнопках. Свойство **DisabledImages** используется совместно со

	свойством **ImageIndex** компонента **ToolButton**.
DockSite	Определяет, используется ли панель инструментов для стыковки других компонентов.
EdgeBorders	Вложенные свойства **ebLeft**, **ebTop**, **ebRight** и **ebBottom** определяют видимость соответственно левой, верхней, правой и нижней сторон рельефной рамки.
EdgeInnter	Внутренний скос рельефной рамки: **esNone** — скос отсутствует, **esLowered** — скос внутрь, **esRaised** — скос наружу.
EdgeOuter	Внешний скос рельефной рамки: **esNone** — скос отсутствует, **esLowered** — скос внутрь, **esRaised** — скос наружу.
Flat	Если равно значению True, то все кнопки, находящиеся на панели инструментов, не имеют рельефных границ. Рельефные границы появляются при наведении указателя мыши на кнопку.
HideClippedButtons	Если равно значению True, то кнопки, не уместившиеся на панели целиком, не показываются вообще.
HotImages	Список значков, которые отображаются на кнопках при наведении на них указателя мыши. Свойство **HotImages** используется совместно со свойством **ImageIndex** компонента **ToolButton**.
Images	Список значков, которые отображаются на кнопках. Свойство **Images** используется совместно со свойством **ImageIndex** компонента **ToolButton**.
Indent	Отступ от края панели до первой кнопки.
List	Если равно значению True, то надписи на кнопках отображаются справа от значков. Иначе надписи отображаются под значками.
Menu	Ссылка на компонент MainMenu. Установка значения этого свойства приводит к тому, что панель инстурментов выглядит как строка главного меню.
ShowCaptions	Определяет, отображаются ли надписи на кнопках. Если установлено значение False,

	то на кнопках отображаются только значки.
ShowHint	Разрешает (значение True) или запрещает (значение False) показ высплывающих подсказок для кнопок панели инструментов.
Transparent	Если равно значению True, то фон панели инструментов является прозрачным.
Wrapable	Включает автоматический перенос невместившихся кнопок панели инструментов на новую строку. Если равно значению False, то перенос кнопок регулируется с помощью свойства Wrap компонента **ToolButton**.
OnAdvancedCustomDraw	Происходит до и после рисования панели инструментов на экране.
OnAdvancedCustomDrawButton	Происходит до и после рисования каждой кнопки панели инструментов на экране.
OnCustomDraw	Происходит при рисовании панели инструментов на экране.
OnCustomDrawButton	Происходит при рисовании каждой кнопки панели инструментов на экране.
OnCustomizeAdded	Происходит, когда пользователь добавляет кнопку на панель с помощью окна настройки панели инструментов.
OnCustomizeCanDelete	Происходит, когда пользователь пытается убрать кнопку с помощи окна настройки панели инструментов.
OnCustomizeCanInsert	Происходит, когда пользователь пытается добавить кнопку с помощи окна настройки панели инструментов.
OnCustomized	Происходит по окончании любого изменения на панели инструментов.
OnCustomizeDelete	Происходит, когда пользователь убирает кнопку с панели при помощи окна настройки панели инструментов.
OnCustomizeNewButton	Используется для формирования списка кнопок, доступных для добавления на панель инструментов. В этом обработчике, как правило, динамически создаются и возвращаются через параметр **Button** объекты класса **TToolButton** (или производных классов).

OnCustomizeReset	Происходит при нажатии пользователем кнопки Reset в окне настройки панели инструментов.
OnCustomizing	Происходит при вызове пользователем окна настройки панели инструментов и при нажатии кнопки Reset в этом окне.
OnGetSiteInfo	Происходит, когда у компонента запрашивается место для стыковки.

*Таблица 8.15. Важнейшие свойства и события компонента **ToolBar***

8.5.2. Кнопки

Кнопки панели инструментов представлены компонентами **ToolButton**. Не пытайтесь найти компонент **ToolButton** в палитре компонентов. Его там нет, поскольку он не является самостоятельным компонентом, а создается и управляется из компонента **ToolBar**.

Шаг 55. Для добавления кнопки вызовите контекстное меню компонента **ToolBar** и выберите команду **New Button** (рисунок 8.55).

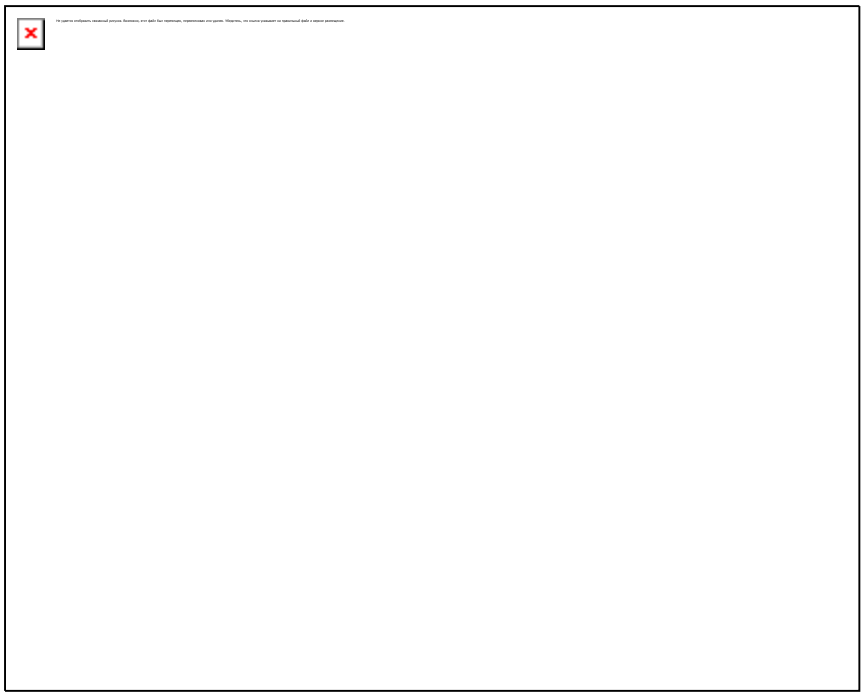


Рисунок 8.55. Создание кнопки на панели инструментов с помощью команды New Button контекстного меню

На панели инструментов появится кнопка, свойства которой будут тут же показаны в окне свойств. Дайте компоненту имя **OpenToolButton**.

Шаг 56. Аналогичным образом создайте еще четыре кнопки с программными идентификаторами **SaveAsToolButton**, **HalfSizeToolButton**, **NormalSizeToolButton** и **DoubleSizeToolButton** (рисунок 8.56).

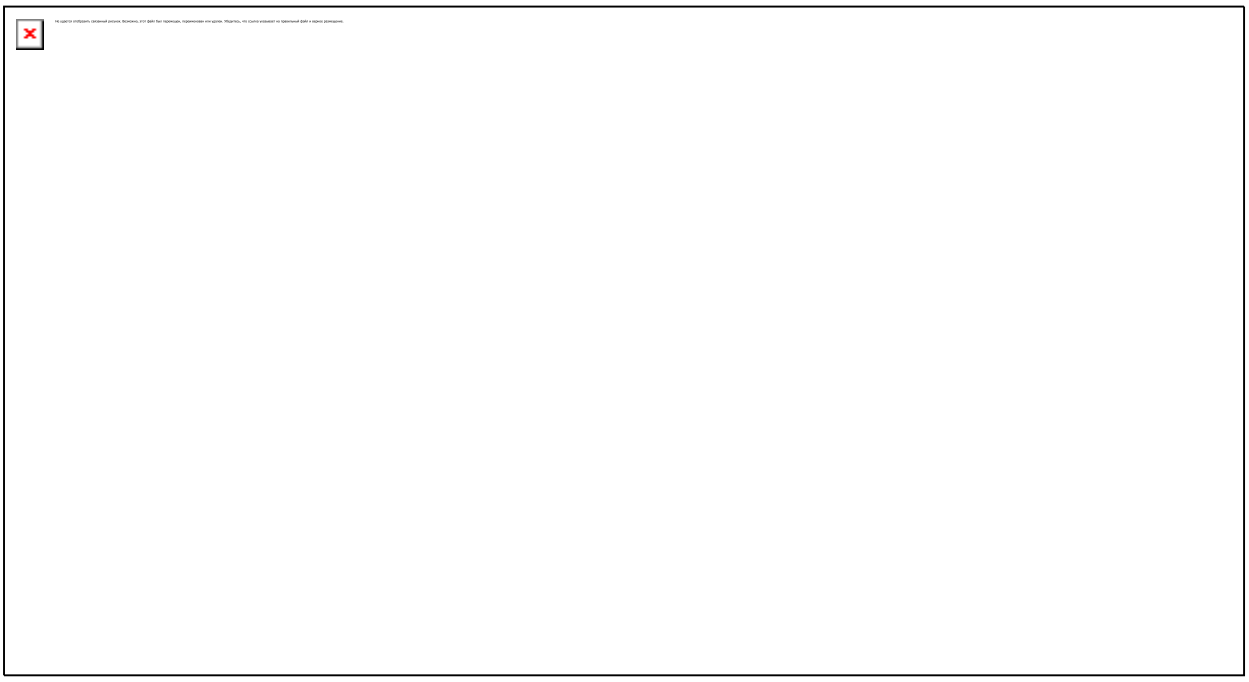


Рисунок 8.56. Все необходимые кнопки созданы, но для них еще не заданы значки

Подготовительная работа завершена, список кнопок готов. На следующем шаге мы назначим кнопкам значки, но прежде перечислим важнейшие свойства компонента **ToolButton**, с которыми нам придется дальше работать (таблица 8.16).

Свойство	Описание
Action	Команда, хранящаяся в компоненте **ActionList** и выполняемая при нажатии кнопки (см. параграф 8.6).
AllowAllUp	Разрешает всем кнопкам одной группы находиться в отжатом состоянии.
AutoSize	Включает режим автоматического подбора размеров кнопки в зависимости от размеров значка и надписи.
Caption	Надпись на кнопке.
Down	Если равно True, то кнопка рисуется нажатой.
DropDownMenu	Выпадающее меню, которое появляется при нажатии кнопки. Это свойство используется, если свойство Style содержит значение tbdDropDown.
Grouped	Определяет, принадлежит ли кнопка группе взаимоисключающих переключателей. Сгруппированными считаются расположенные рядом кнопки со значением True в свойстве **Grouped** и значением **tbdCheck** в свойстве **Style**.
ImageIndex	Номер значка в списке **Images** компонента **ToolBar**.
Indeterminate	Если равно True, то кнопка имеет неопределенное состояние и рисуется поблекшей.

Marked	Если равно значению True, то кнопка подсвечивается цветом выделенных элементов (стандартно — синим цветом).
MenuItem	Пункт меню, с которым ассоциирована кнопка. При установке этого свойства из соответствующего пункта меню копируются значения наиболее важных свойств и событий, например Caption, ImageIndex, Enabled, Hint, OnClick.
Style	Тип кнопки: **tbsButton** — обычная кнопка, **tbsCheck** — кнопка-переключатель, **tbsDivider** — разделитель в виде вертикальной черты, **tbsDropDown** — выпадающий список, **tbsSeparator** — разделитель в виде вертикальной черты или пробела в зависимости от значения свойства **Flat** компонента **ToolBar**.
Wrap	Обеспечивает перенос последующих кнопок на новую строку.

*Таблица 8.16. Важнейшие свойства компонента **ToolButton***

8.5.3. Значки на кнопках

Главным атрибутом кнопки является значок. Он определяется значением свойства **ImageIndex** (номер значка в списке **Images** компонента **ToolBar**). Установим на кнопках значки, воспользовавшись ранее созданным списком **ImageList**.

Шаг 57. Выделите на форме компонент **ToolBar**, перейдите к окну свойств и установите свойству **Images** значение **ImageList** (рисунок 8.57).

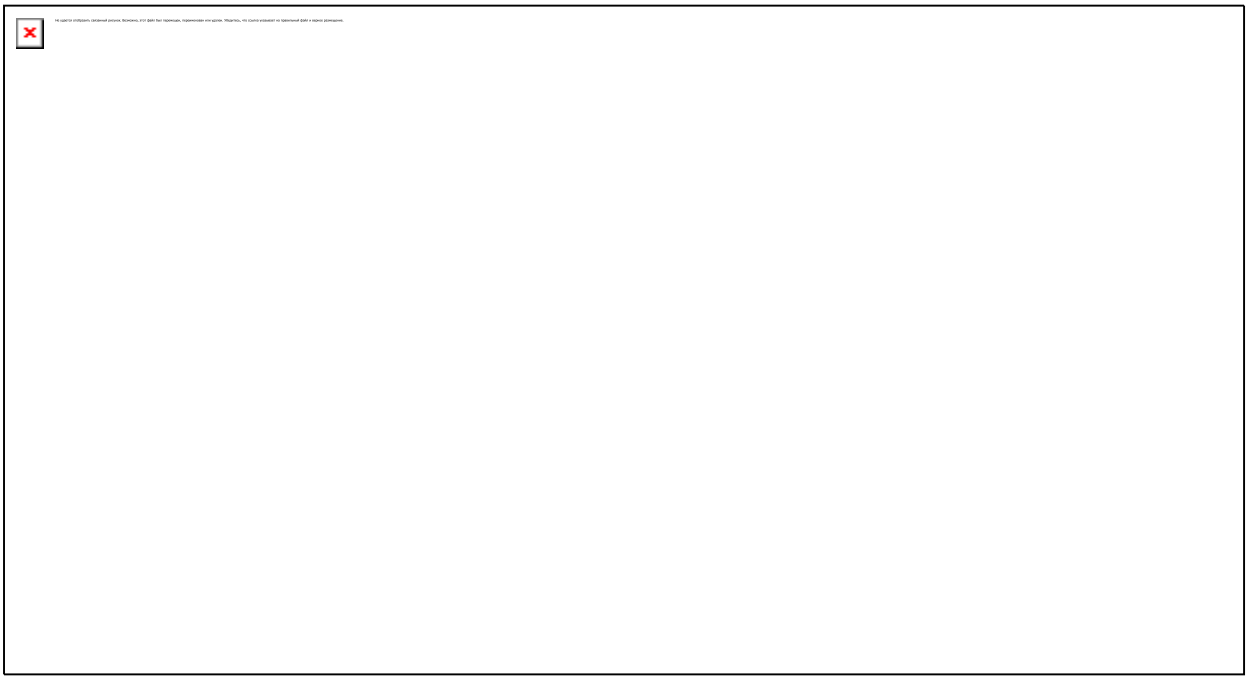


Рисунок 8.57. На кнопках панели инструментов появились значки

Вот здорово! На всех кнопках появились разные значки, хотя мы не устанавливали свойство **ImageIndex** ни в одной из кнопок. Это произошло потому, что компонент **ToolBar** сделал

это за нас, назначив каждой кнопке номер значка в соответствии с очередностью добавления кнопок на панель.

Признаемся, что мы немного схитрили, заранее расположив значки в компоненте **ImageList** в том порядке, в котором они расположены на панели инструментов. В реальных задачах вам, вероятно, потребуется вручную указывать номера значков для каждой кнопки с помощью свойства **ImageIndex**.

8.5.4. Надписи на кнопках

Шаг 58. Кнопка может содержать надпись рядом со значком. Текст надписи устанавливается в свойстве **Caption**. Сначала он не виден и, чтобы его показать, задайте в компоненте **ToolBar** свойству **ShowCaptions** значение **True** (рисунок 8.58).

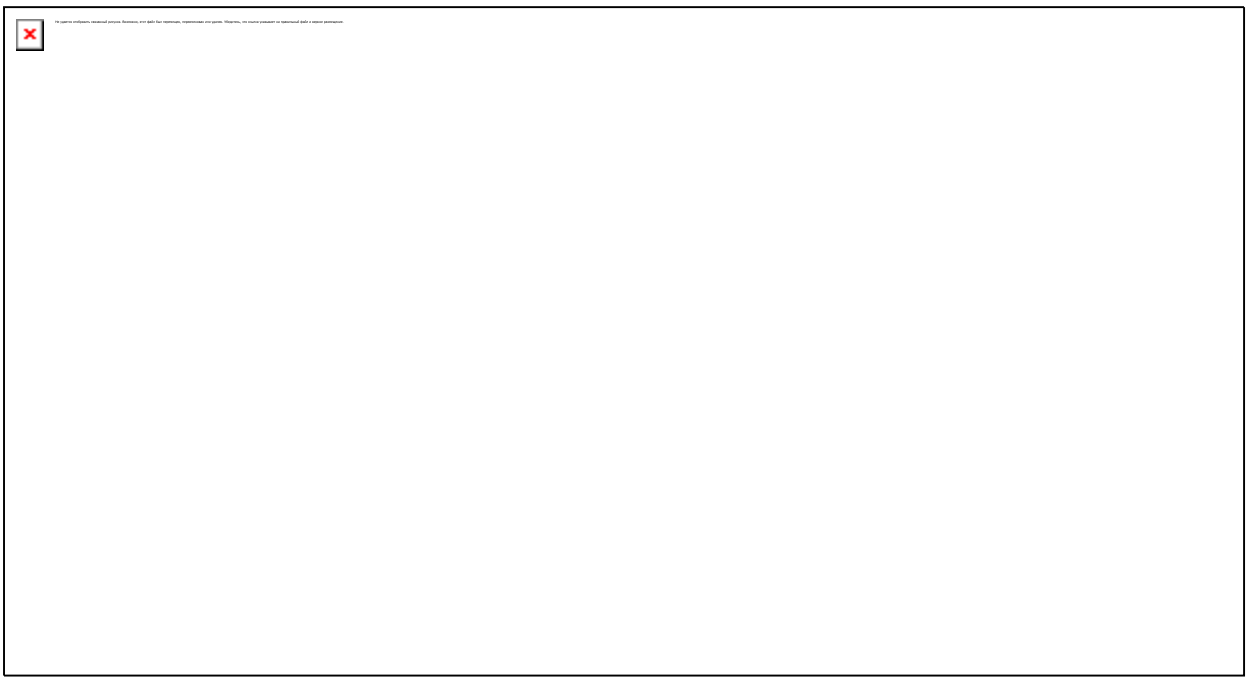


Рисунок 8.58. Кнопки панели инструментов могут содержать надписи (свойство ShowCaptions равно True)

Результат получился немного неожиданный: стандартные надписи на кнопках эквивалентны их программным идентификаторам и расположены под значками вместо того, чтобы находиться справа.

Шаг 59. Подправим надписи. В окне свойств переключите свойство **List** в значение **True** (рисунок 8.59).

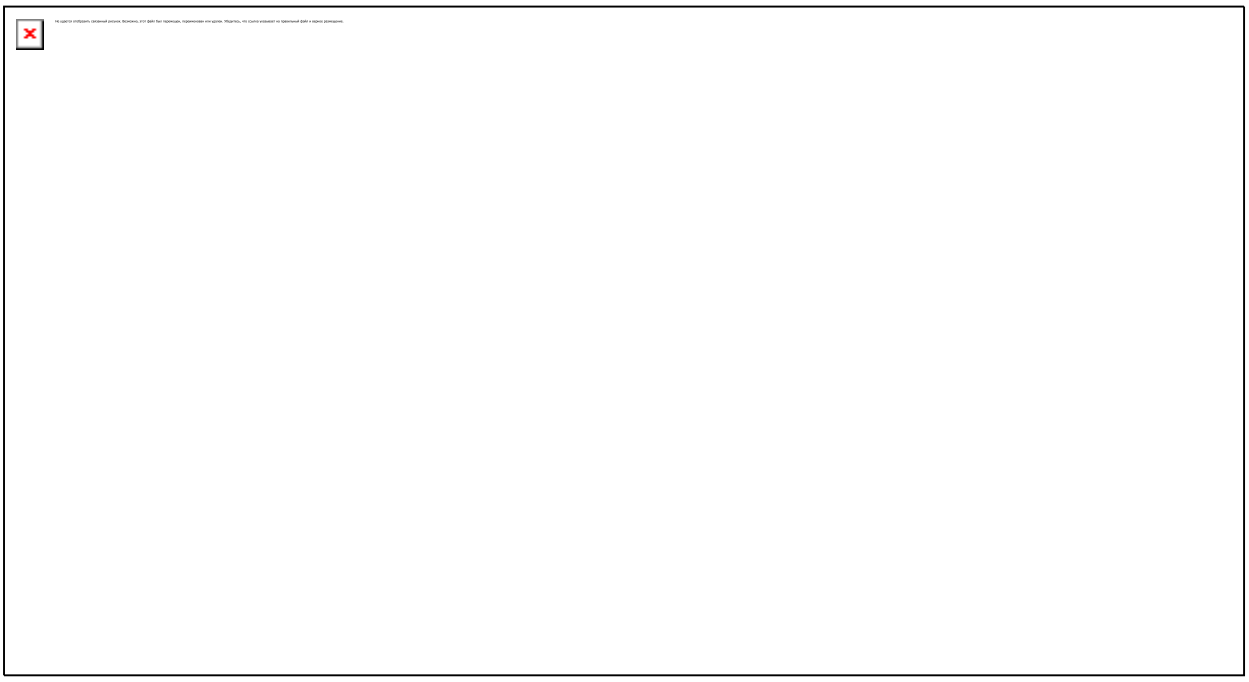


Рисунок 8.59. Надписи на кнопках располагаются справа от значков (свойство List равно True)

Как вы уже догадались, свойство **List** управляет расположением текста и значков относительно друг друга.

Шаг 60. Теперь измените надписи на кнопках (свойство **Caption**), чтобы получить панель инструментов, показанную на рисунке 8.60.

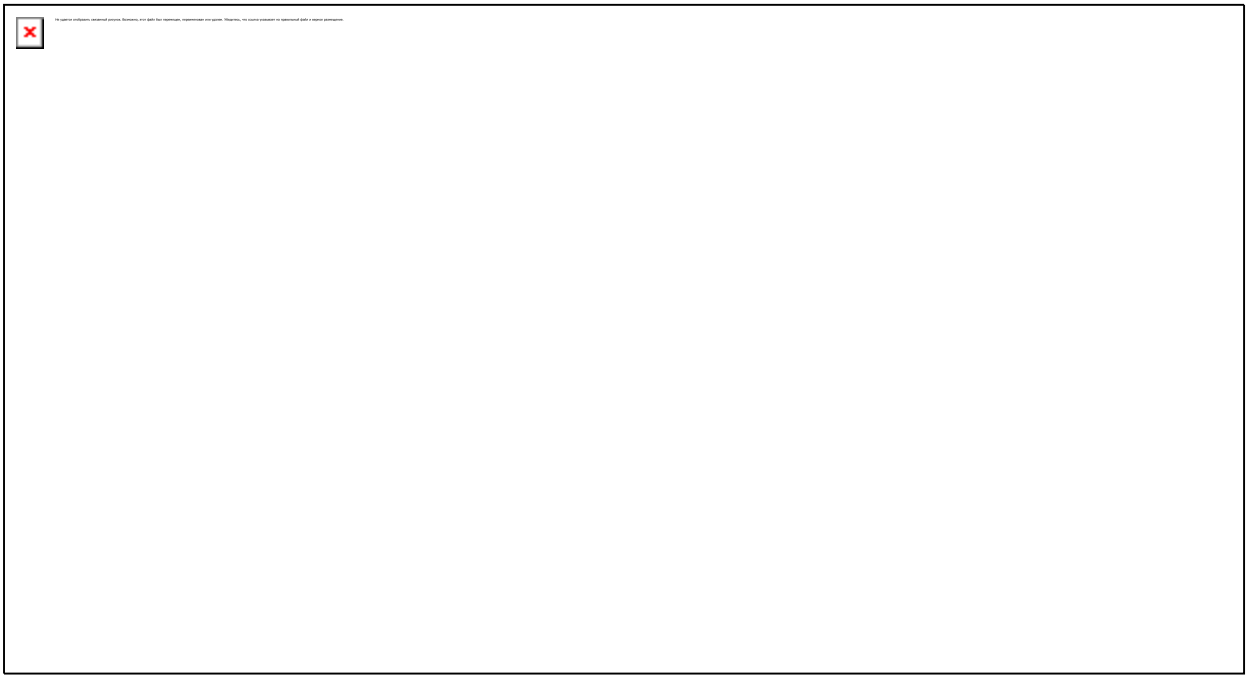


Рисунок 8.60. Кнопкам заданы правильные надписи

Шаг 61. Чтобы панель инструментов выглядела лучше, подгоним размеры кнопок под размеры надписей. Для этого воспользуемся свойством **AutoSize**, которое есть у каждой кнопки. При установке этого свойства применим технику группового редактирования компонентов.

Шаг 62. Выделите щелчком мыши первую кнопку, а затем, удерживая клавишу Shift, выделите щелчками мыши все остальные кнопки. В окне свойств произойдут следующие изменения:

- вместо имени активного компонента будет показано общее количество выделенных компонентов:
- в списке свойств останутся только общие для выделенных компонентов свойства и события;
- свойства и события, которые у выделенных компонентов имеют разные значения, окажутся пустыми.

Перейдите к окну свойств и установите свойство **AutoSize** в значение True (рисунок 8.61).

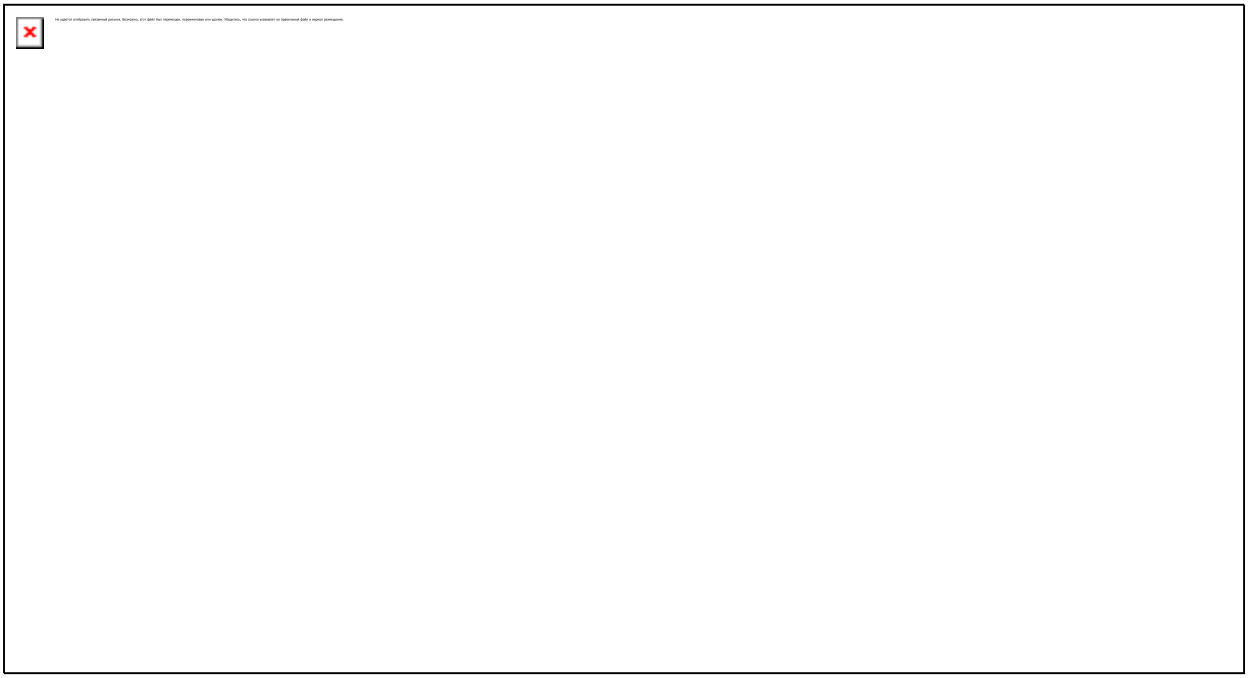


Рисунок 8.61. Применение техники группового редактирования при установке во всех кнопках свойства AutoSize в значение True

Теперь выполните компиляцию и запустите программу. Результат представлен на рисунке 8.62.

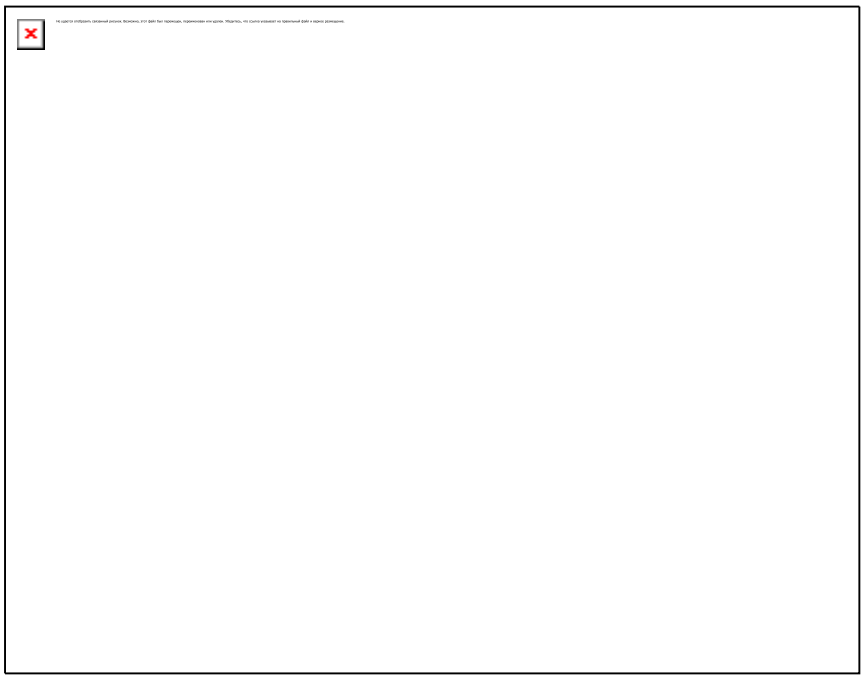


Рисунок 8.62. Программа для просмотра графических файлов имеет панель инструментов, однако кнопки пока еще не работают

Кнопки нажимаются, но реакции на них пока нет. Мы этим займемся потом, а сейчас придадим панели инструментов более современный вид. Избавимся от чрезмерного количества 3D-эффектов на кнопках.

Шаг 63. Закройте приложение и вы вернетесь в среду Delphi. Выделите на форме компонент **ToolBar** и переключите его свойство **Flat** в значение True. Теперь снова запустите программу и полюбуйтесь на результат (рисунок 8.63).

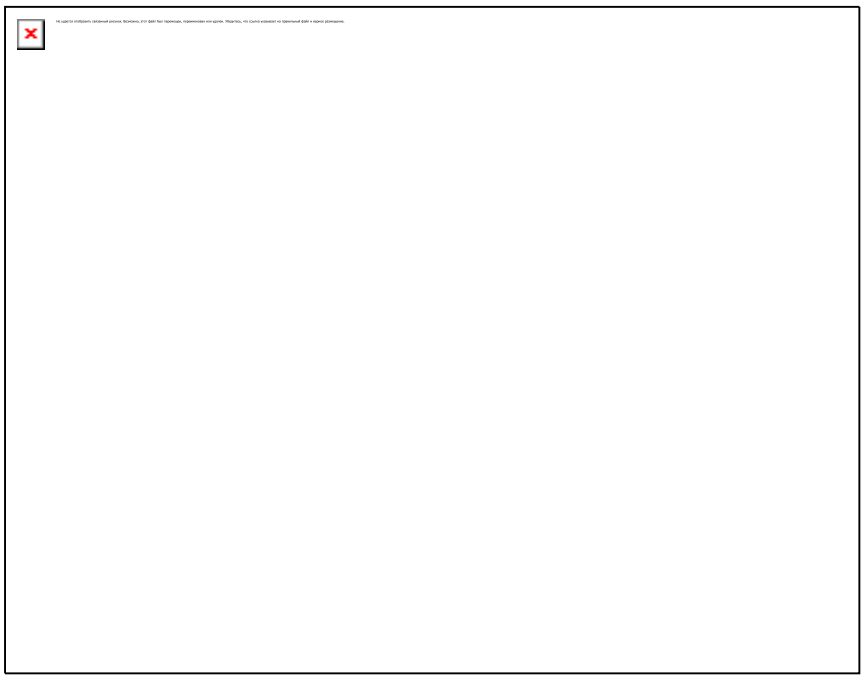


Рисунок 8.63. Кнопки панели инструментов получили современный «плоский» вид

Кнопки приобретают рельефный вид только при наведении на них указателя мыши.

8.5.5. Разделительные линии

Важными элементами панели инструментов являются разделительные линии, которые используются для группировки кнопок.

Шаг 64. В нашем примере логично отделить группу кнопок, отвечающих за размеры рисунка (**Half Size**, **Normal Size** и **Double Size**), от кнопок **Open** и **Save As**. Для этого вызовите контекстное меню панели инструментов и выберите команду **New Separator** (рисунок 8.64).

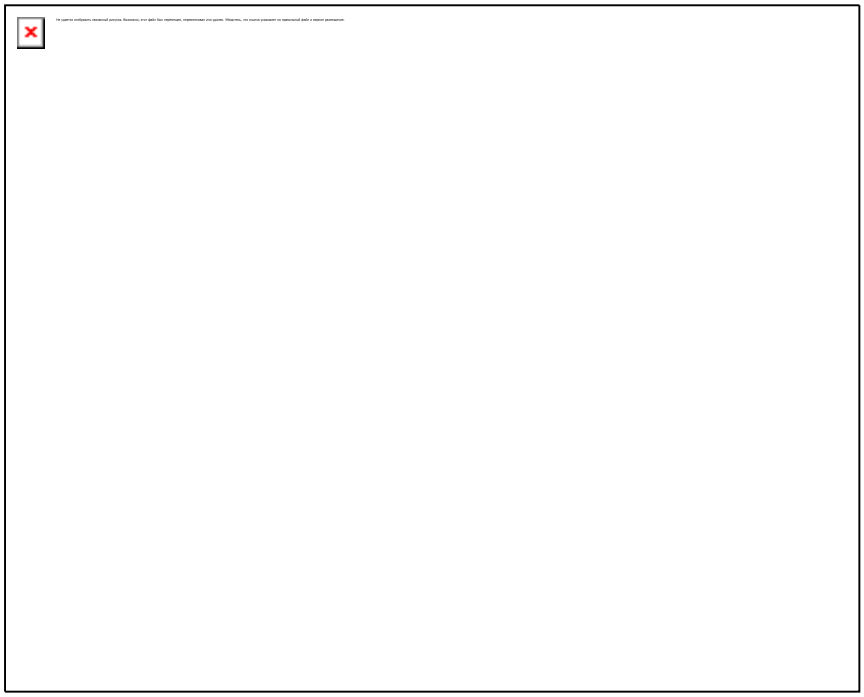


Рисунок 8.64. Создание разделительной линии на панели инструментов с помощью команды New Separator контекстного меню

На панель инструментов будет добавлен новый компонент, имеющий вид вертикальной черты. С помощью мыши отбуксируйте его на место между кнопками **Save As** и **Half Size** (рисунок 8.65):

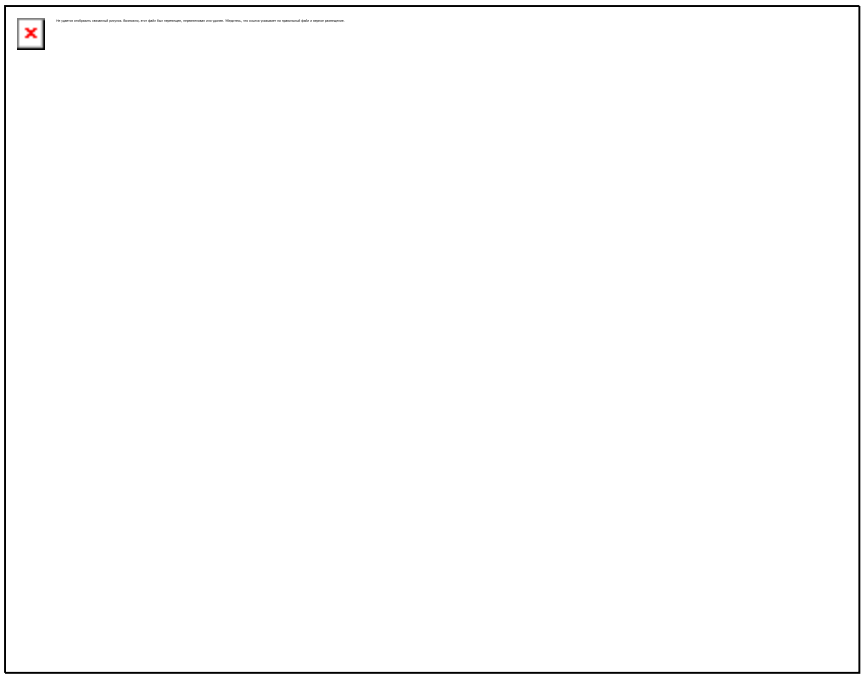


Рисунок 8.65. Буксировка разделительной линии на место между кнопками Save As и Half Size

Выполните компиляцию и запустите программу. Результат представлен на рисунке 8.66.

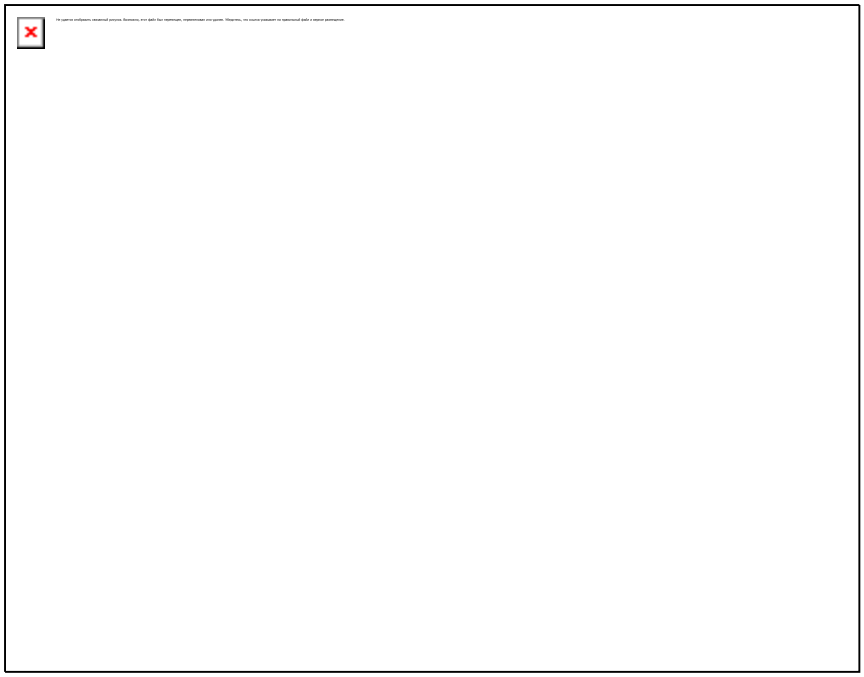


Рисунок 8.66. Кнопки на панели инструментов внешне сгруппированы по назначению

Напоследок заметим, что разделительная линия представлена обычным компонентом **ToolButton**. То, какой вид имеет этот компонент (кнопка или разделительная линия), определяется свойством **Style**. Это свойство имеет много значений, которые перечислены в таблице 8.16.

8.5.6. Кнопки-переключатели

Кнопки панели инструментов могут работать как переключатели, «залипая» при нажатии. Для того, чтобы кнопка была переключателем, ее свойство **Style** должно содержать значение **tbsCheck**. Состояние кнопки (нажата она или нет) определяется значением свойства **Down**.

Кнопки-переключатели могут работать согласовано, т.е. включение одной из них означает выключение остальных. Именно так должны работать кнопки выбора масштаба отображения рисунка. Согласованная работа кнопок обеспечивается не так, как согласованная работа пунктов меню. Кнопки панели инструментов не имеют свойства **GroupIndex**, они группируются по другому принципу. Сгруппированными считаются расположенные рядом кнопки, у которых свойство **Grouped** равно значению **True** и свойство **Style** равно значению **tbsCheck**.

Шаг 65. Сгруппируйте кнопки **Half Size**, **Normal Size** и **Double Size**. Они уже находятся рядом друг с другом, поэтому просто установите их свойства **Grouped** и **Style** как показано на рисунке 8.67.

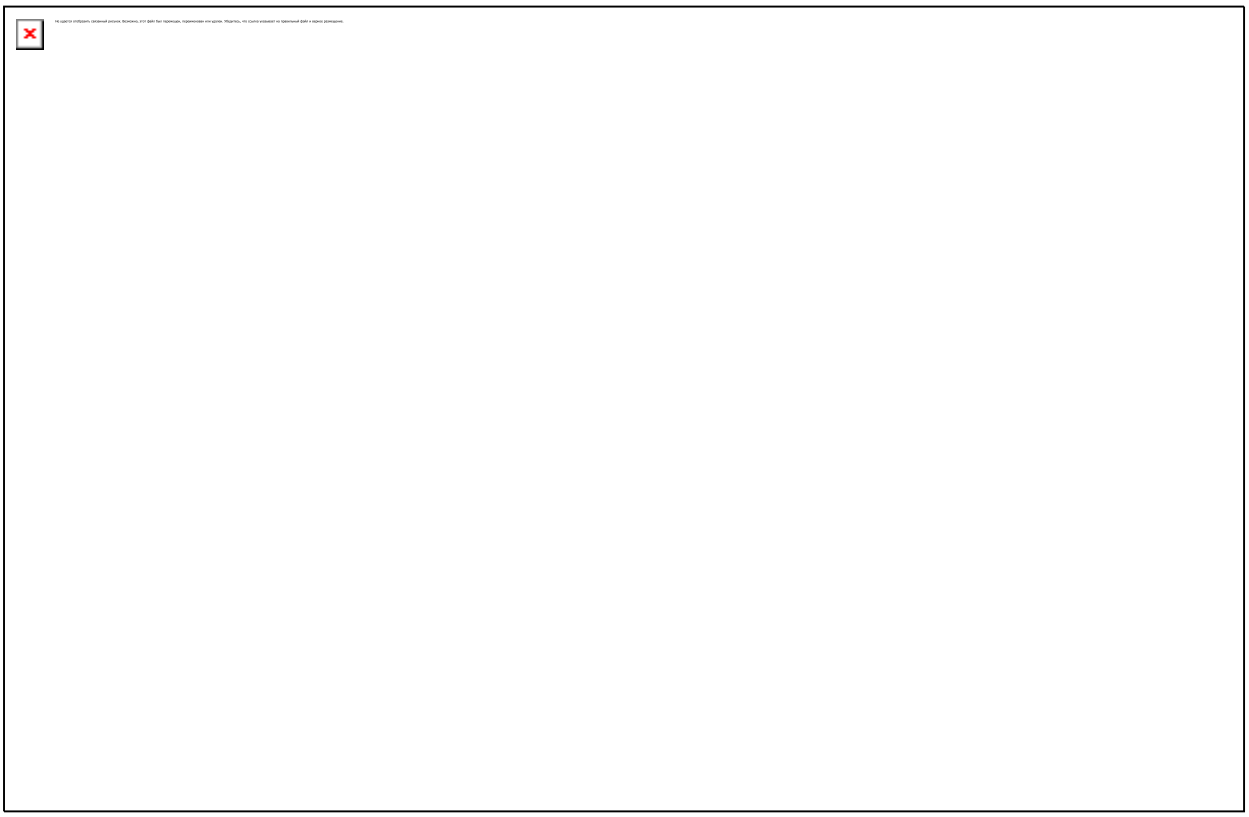


Рисунок 8.67. Кнопки Half Size, Normal Size и Double Size сгруппированы в трехпозиционный переключатель

У кнопки **Normal Size** установите свойство **Down** в значение **True**, предварительно убрав выделение кнопок **Half Size** и **Normal Size** (рисунок 8.68).

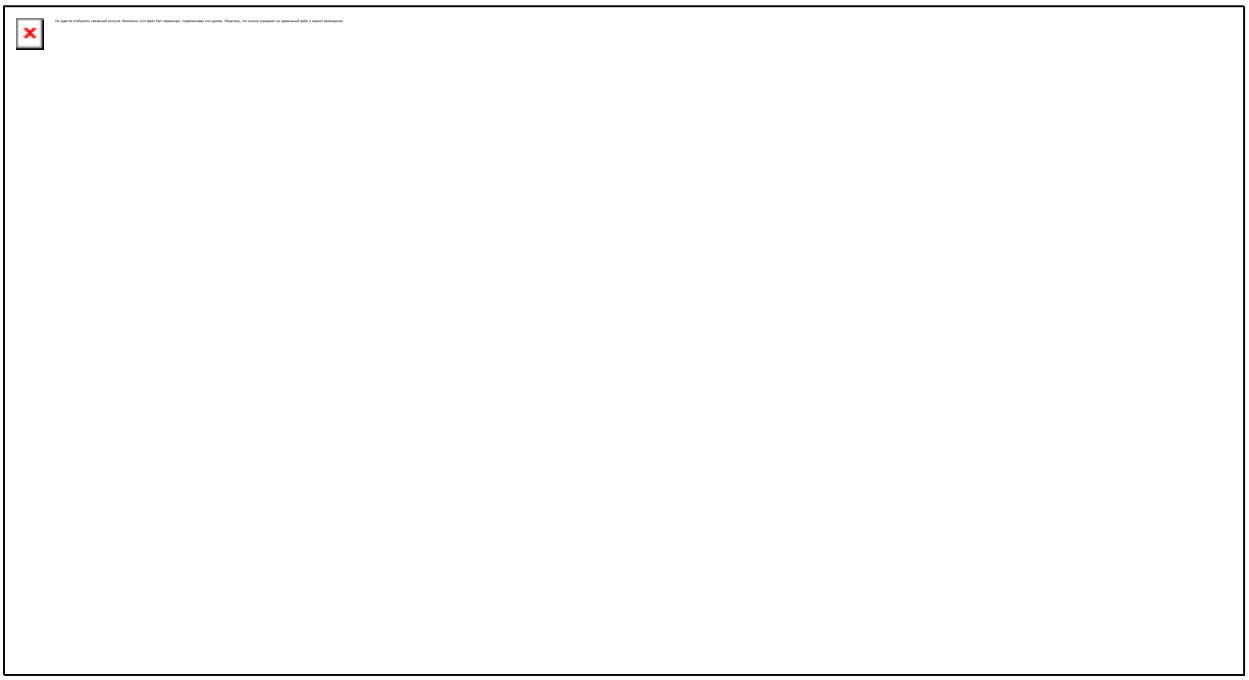


Рисунок 8.68. Начальное положение трехпозиционного переключателя — Normal Size

Выполните компиляцию и запустите программу. Проверьте, что кнопки **Half Size**, **Normal Size** и **Double Size** работают как трехпозиционный переключатель (рисунок 8.69).

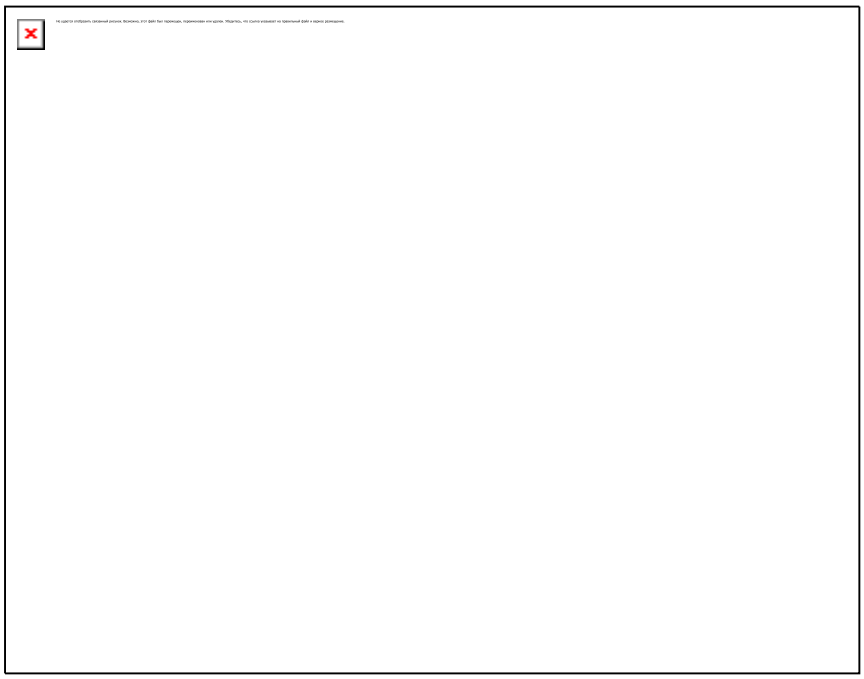


Рисунок 8.69. Кнопки-переключатели в работе

Шаг 66. Завершая дизайн панели инструментов, сделайте кнопки **Save As**, **Half Size**, **Normal Size** и **Double Size** недоступными, установив у них свойство **Enabled** в значение **False** (рисунок 8.70). Эти кнопки будут оставаться недоступными, пока пользователь не откроет какой-нибудь рисунок.

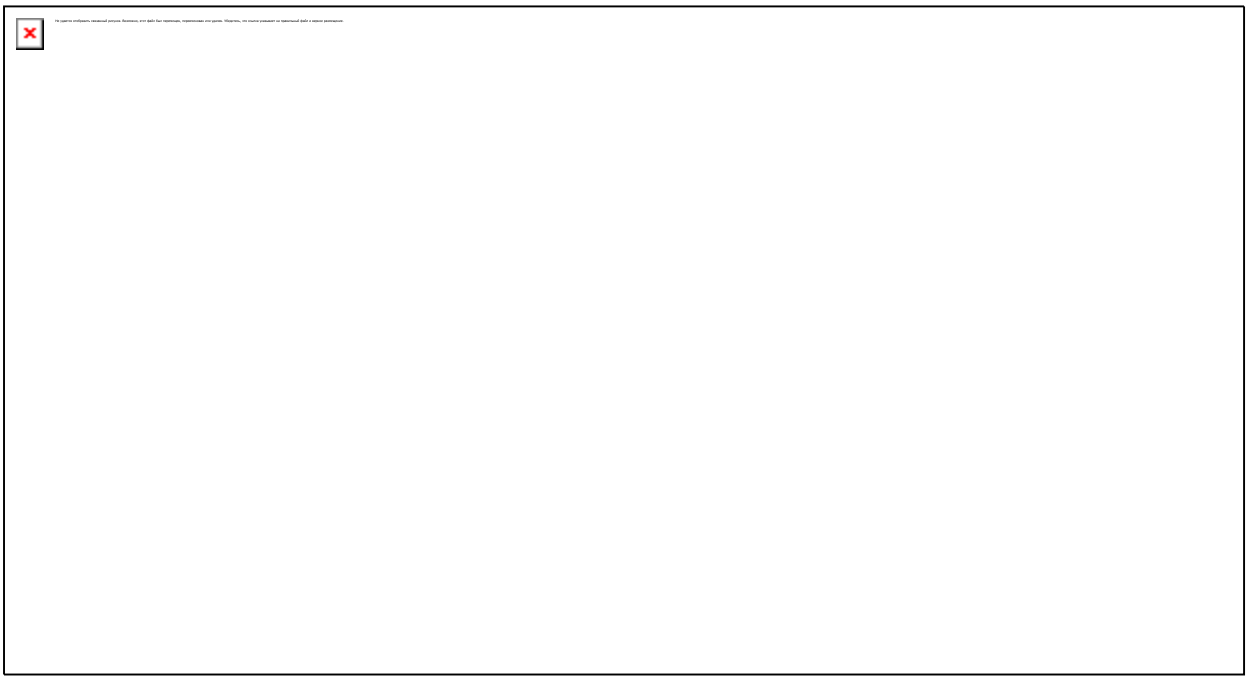


Рисунок 8.70. Некоторые кнопки на панели инструментов отключены до тех пор, пока пользователь не откроет какой-нибудь графический файл

Теперь все готово к тому, чтобы запрограммировать обработчики для кнопок панели инструментов.

8.5.7. Обработка нажатий кнопок

При нажатии кнопок возникают события **OnClick**, которые должны обрабатываться точно так же, как и команды меню. Поскольку все события **OnClick** имеют одинаковый формат для всех компонентов, просто подключите к кнопкам уже существующие обработчики событий.

Шаг 67. Группа кнопок, управляющих масштабом рисунка, должна правильно переключаться даже тогда, когда пользователь устанавливает масштаб с помощью команд меню. Поэтому дополните обработчики событий так, как показано ниже:

```
procedure TPictureForm.HalfSizeMenuItemClick(Sender: TObject);
begin
  HalfSizeToolButton.Down := True; // кнопка согласуется с пунктом меню
  ...
end;

procedure TPictureForm.NormalSizeMenuItemClick(Sender: TObject);
begin
  NormalSizeToolButton.Down := True; // кнопка согласуется с пунктом меню
  ...
end;

procedure TPictureForm.DoubleSizeMenuItemClick(Sender: TObject);
begin
  DoubleSizeToolButton.Down := True; // кнопка согласуется с пунктом меню
  ...
end;
```

Шаг 68. Чтобы кнопки становились доступными или недоступными в зависимости от того, открыт рисунок или нет, подправьте метод **EnableCommands**:

```
procedure TPictureForm.EnableCommands(Enable: Boolean);
begin
  ...
  SaveAsToolButton.Enabled := Enable;
  HalfSizeToolButton.Enabled := Enable;
  NormalSizeToolButton.Enabled := Enable;
  DoubleSizeToolButton.Enabled := Enable;
end;
```

Вроде бы все. После компиляции и запуска программы вы получите работающую панель инструментов (рисунок 8.71). Нажмите кнопку **Open** и выберите рисунок. Когда рисунок откроется, все остальные кнопки станут доступными. Понажимайте кнопки, отвечающие за масштаб, и убедитесь, что они работают согласовано с командами меню.

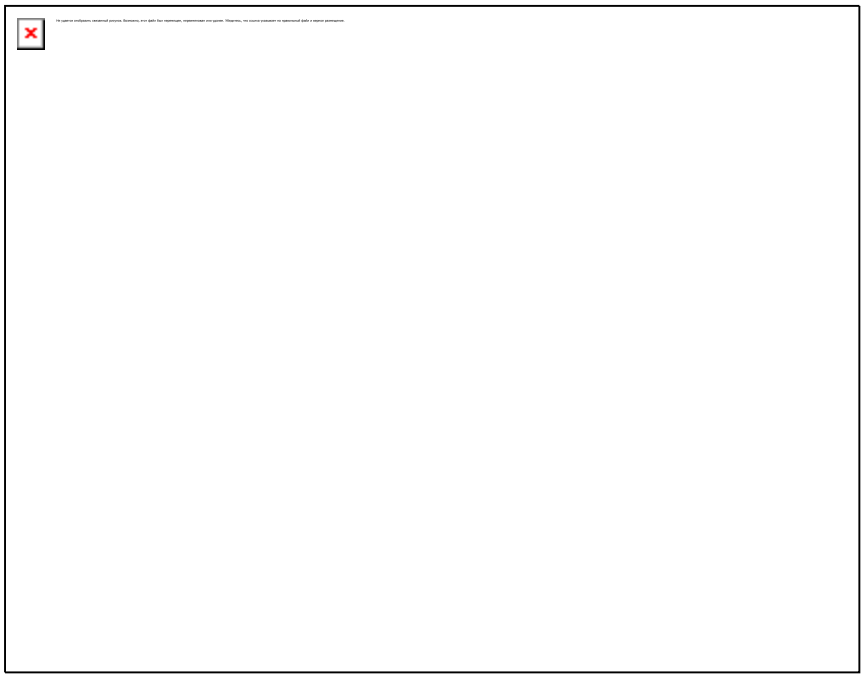


Рисунок 8.71. Программа для просмотра графических файлов получила работоспособную панель инструментов

Панель инструментов работоспособна, но в ней не хватает одной мелочи — подсказок к кнопкам.

8.5.8. Подсказки к кнопкам

Все визуальные компоненты в среде Delphi, в том числе и кнопки панели инструментов, могут иметь подсказки. Подсказки бывают двух видов: *всплывающие подсказки* и *подсказки в строке состояния* (рисунок 8.72).

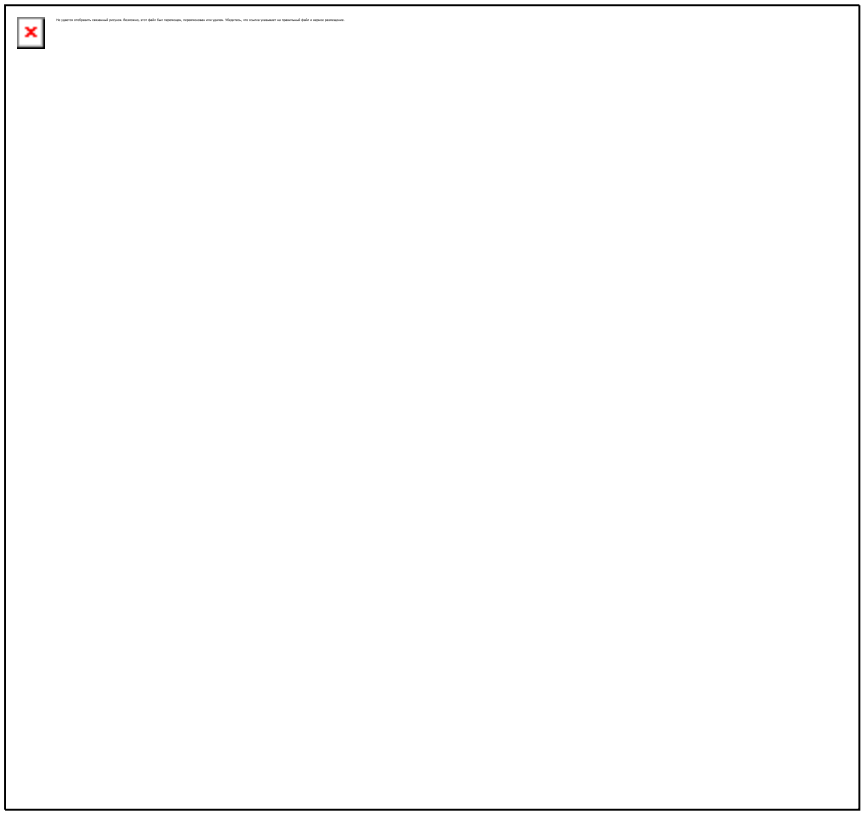


Рисунок 8.72. Всплывающая подсказка и подсказка в строке состояния

Обе подсказки хранятся в свойстве **Hint**, которое имеет следующий формат:

<всплывающая подсказка>|<подсказка в строке состояния>

Всплывающая подсказка отделяется вертикальной чертой от подсказки для строки состояния.

Если в программе есть строка состояния, то при попадании указателя мыши на визуальный компонент в ней автоматически отображается поясняющий текст, записанный справа от символа вертикальной черты. Это легко проверить. Впишите в свойстве **Hint** кнопки **Open** текст "Open an existing file...|Open an existing file...". После запуска программы вы обнаружите, что подсказка для кнопки работает точно так же, как и подсказка к пункту меню.

Наиболее удобный вид подсказок — это всплывающие подсказки. *Всплывающая подсказка* появляется спустя секунду после того, как пользователь задерживает указатель мыши над компонентом. Приятная особенность всплывающих подсказок состоит в том, что они вообще не требуют программирования — достаточно просто разрешить компоненту отображать всплывающую подсказку, и подсказка начнет работать.

Шаг 69. В каждом визуальном компоненте существует булевское свойство **ShowHint**, определяющее, появляется подсказка, или нет (рисунок 8.73). Его значение может устанавливаться напрямую, а может копироваться из содержащего компонента (например, из формы). Копирование значения происходит тогда, когда вспомогательное свойство **ParentShowHint** установлено в True. Таким образом, появлением подсказок можно управлять из содержащего компонента. Этой возможностью мы воспользуемся в нашей

задаче. Принимая во внимание, что во всех компонентах свойство **ParentShowHint** изначально равно True, просто установите в компоненте **ToolBar** (владелец кнопок) свойство **ShowHint** в значение True. В результате во всех кнопках свойство **ShowHint** тоже получит значение True и это заставит их отображать свои подсказки.

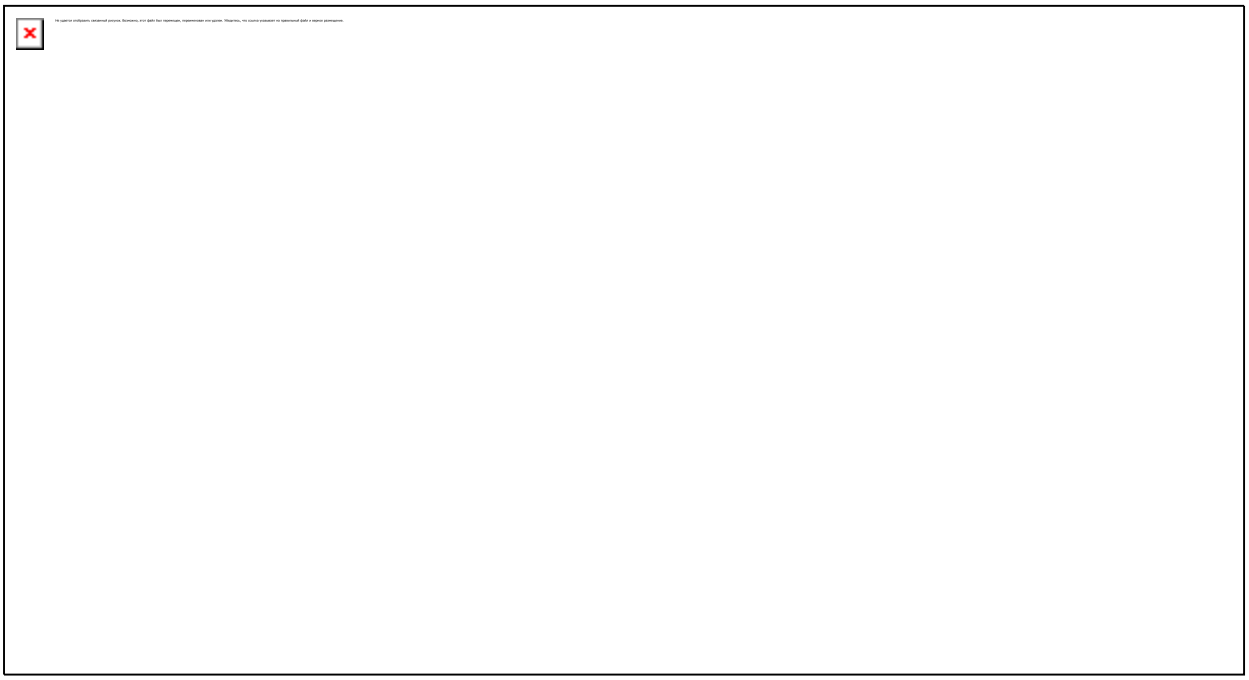


Рисунок 8.73. Свойство ShowHint управляет отображением всплывающих подсказок

Шаг 70. Впишите для каждой кнопки текст подсказки (свойство **Hint**). Как вы знаете, свойство **Hint** может содержать сразу две подсказки: всплывающую подсказку и подсказку в строке состояния (они разделяются символом вертикальной черты). Если вторая кажется вам лишней, просто не набирайте ее, но символ вертикальной черты поставьте (рисунок 8.74).

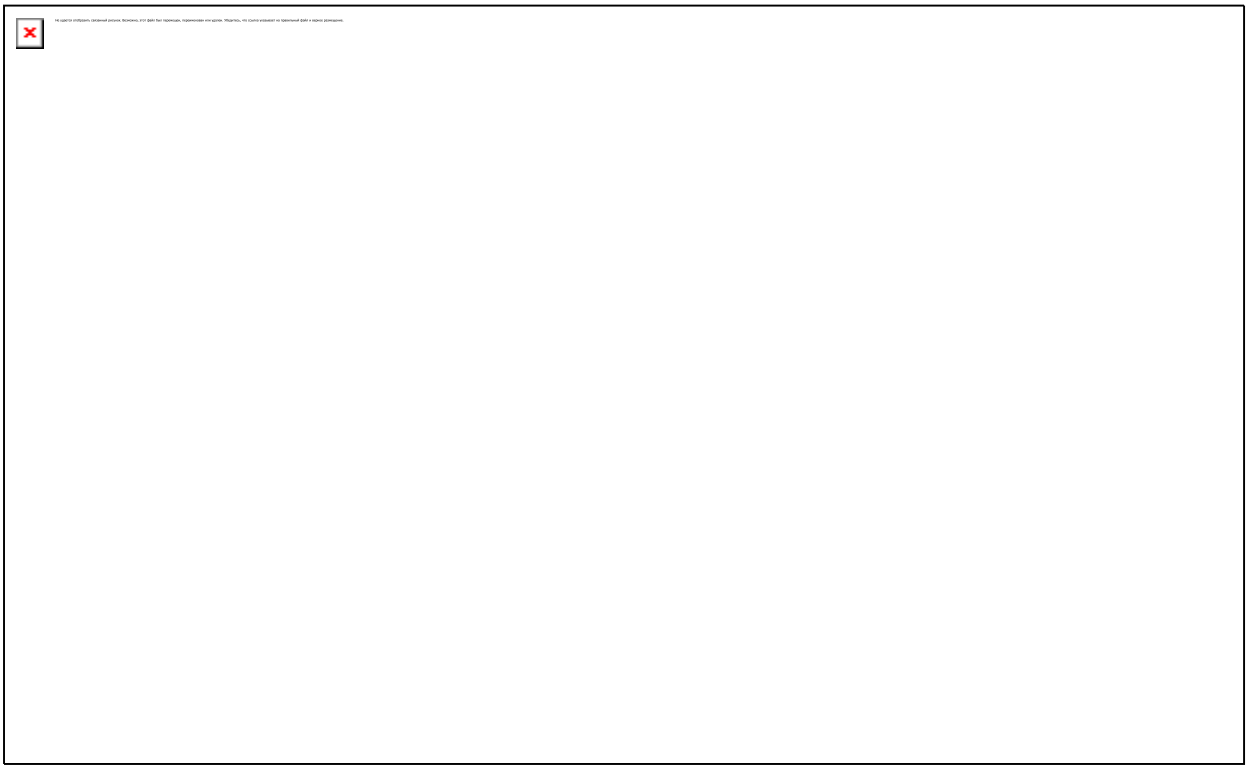


Рисунок 8.74. Символ вертикальной черты в тексте подсказки оставлен, чтобы подсказка не отображалась в строке состояния

Выполните компиляцию и запустите программу. Убедитесь, что панель инструментов работает правильно.

Давайте придадим приложению завершенность и доработаем обработчики команд меню **View | Toolbar** и **View | Status bar**.

8.5.9. Управление видимостью панели кнопок

Шаг 71. Обработка команд **View | Toolbar** и **View | Status bar** выполняется на удивление просто — у соответствующей панели изменяется значение булевского свойства **Visible** на противоположное. При этом панель исчезает или появляется в окне вместе с изменением значения свойства. Проще не придумаешь! Окончательный вариант обработчиков будет следующим:

```
procedure TPictureForm.ToolBarMenuItemClick(Sender: TObject);
begin
  ToolBar.Visible := not ToolBar.Visible;
  ToolBarMenuItem.Checked := not ToolBarMenuItem.Checked;
end;

procedure TPictureForm.StatusBarMenuItemClick(Sender: TObject);
begin
  StatusBar.Visible := not StatusBar.Visible;
  StatusBarMenuItem.Checked := not StatusBarMenuItem.Checked;
end;
```

Итак, приложение для просмотра графических файлов обладает полным набором функциональных возможностей. Выполните его компиляцию и посмотрите, как исчезают и появляются панель инструментов и строка состояния при выборе в меню **View** соответствующих команд.

8.6. Список команд

Часто одна и та же команда дублируется в разных местах пользовательского интерфейса: и в главном меню, и в контекстном меню, и на панели инструментов. Например, команды управления масштабом рисунка присутствуют во всех перечисленных местах программы Picture Viewer. Это очень удобно для пользователя, но добавляет работы программисту, поскольку изменение команды в одном месте требует таких же изменений во всех других местах. На помощь приходит компонент **ActionList**, который централизованно управляет всеми командами пользовательского интерфейса. Рассмотрим его использование.

8.6.1. Создание списка команд

Отыщите в палитре компонентов на вкладке **Standart** компонент **ActionList** и добавьте его в форму (рисунок 8.75).

Рисунок 8.75. Компонент ActionList

Дайте компоненту имя **ActionList** (рисунок 8.76).

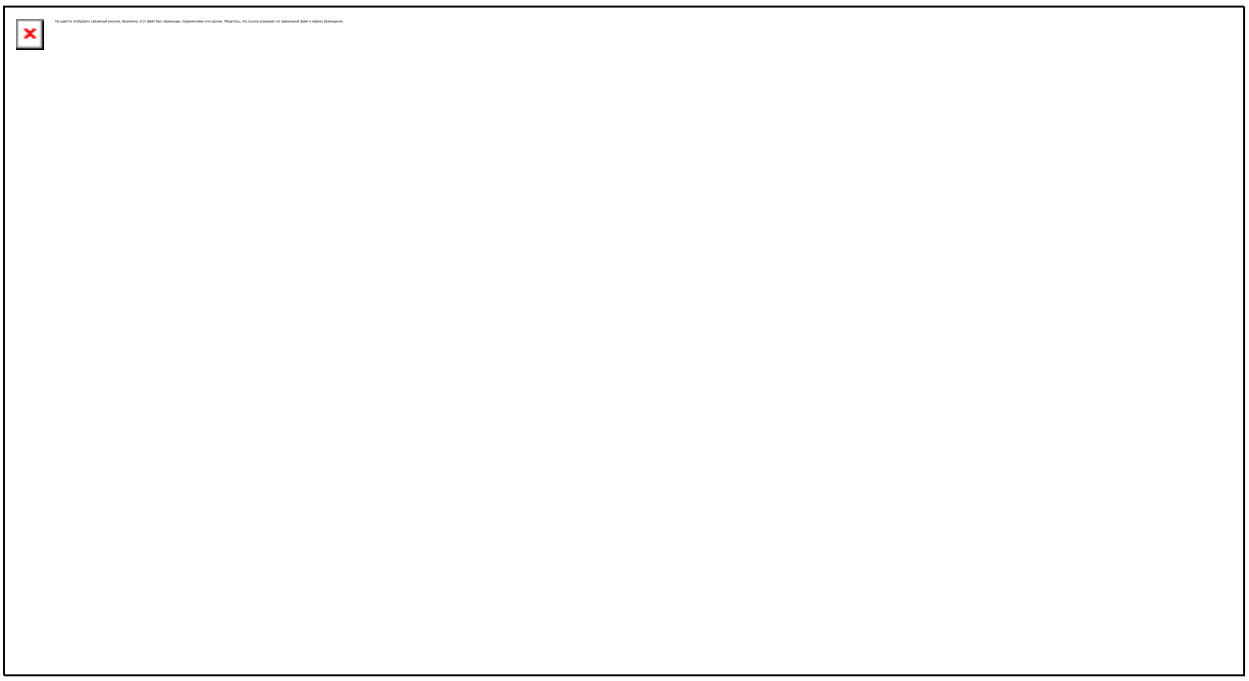


Рисунок 8.76. Компонент ActionList на форме

Ознакомьтесь со свойствами компонента **ActionList** в таблице 8.17.

Свойство	Описание
Images	Список значков, отображаемых в компонентах, использующих команды. Свойство **Images** используется совместно со свойством **ImageIndex** компонентов **Action**.
State	Позволяет временно запретить выполнение всех команд списка. Возможные значения: **asNormal** — команды работают в обычном режиме и доступность каждой команды определяется значением свойства **Enabled** в соответствующем компоненте **Action**; **asSuspended** — все команды недоступны, но при этом не затрагиваются свойства **Enabled** в компонентах **Action**; **asSuspendedEnabled** — все команды недоступны, но имеют обычный вид благодаря тому, что свойство **Enabled** каждого компонента **Action** устанавливается в значение **True**.
OnChange	Происходит при модификации команд в списке. Однако не происходит при создании и уничтожении команд.
OnExecute	Происходит при выполнении команды.
OnStateChange	Происходит при изменении свойства State. Следует учитывать, что из-за ошибки в библиотеке VCL событие не происходит при переводе свойства State в значение asSuspended.
OnUpdate	Происходит при выполнении команды и периодически во время простоя программы. Позволяет отслеживать и

изменять состояния команд.

*Таблица 8.17. Важнейшие свойства и события компонента **ActionList***

Шаг 72. Создание списка команд начнем с команды **Open**. В контекстном меню компонента **ActionList** выберите команду **Action List Editor…** (рисунок 8.77).

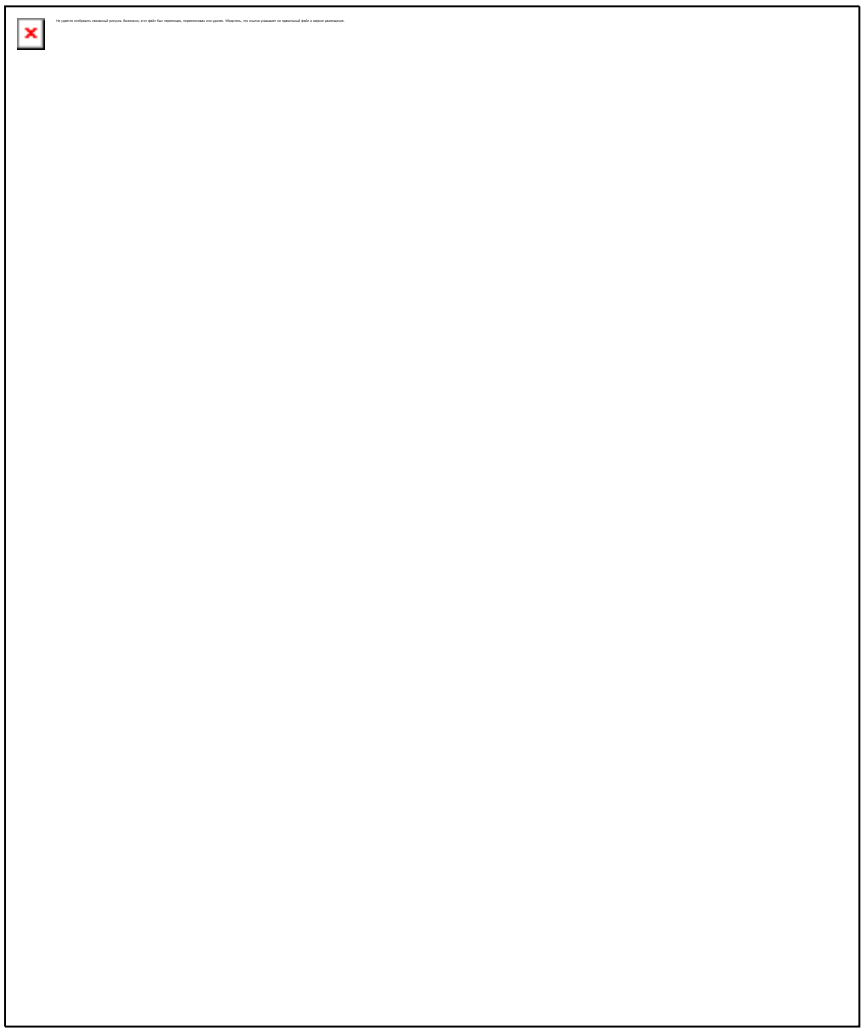


Рисунок 8.77. Вызов списка команд из контекстного меню компонента ActionList

Перед вами откроется окно команд (рисунок 8.78).

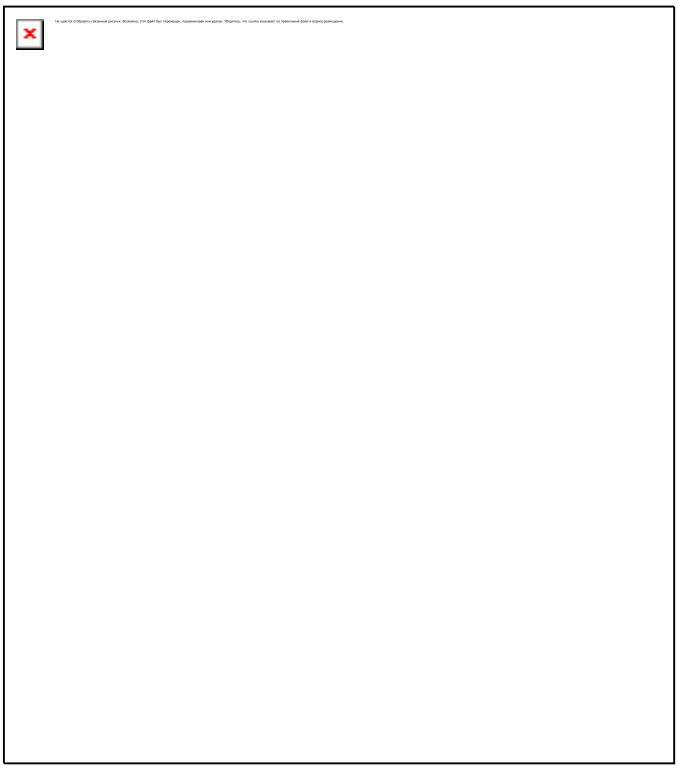


Рисунок 8.78. Окно команд компонента ActionList

Окно команд работает в паре с окном свойств. Создание и удаление команд осуществляется в окне команд, а свойства отдельно взятой команды устанавливаются в окне свойств (рисунок 8.78).

Шаг 73. Щелчком на первой кнопке добавьте в список новую команду. Свойства команды немедленно появятся в окне свойств.

8.6.2. Команды

Когда вы создаете очередную команду в компоненте **ActionList**, среда Delphi добавляет в описание формы компонент **Action**. Компонент **Action** не существует отдельно от компонента **ActionList** и поэтому отсутствует в палитре компонентов. В остальном это обычный компонент, его важнейшие свойства приведены в таблице 8.18.

Свойство	Описание
AutoCheck	Если равно значению True, то выполнение команды (вызов метода Execute) автоматически приводит к изменению значения свойства Checked на противоположное. Если равно значению False, то изменением состояния свойства Checked управляет сам программист.
Caption	Заголовок команды.
Category	Категория команды.
Checked	Если равно значению True, то команда считается выбранной. В этом случае связанные с командой пунткы меню содержат метку, а кнопки имеют вдавленный вид.

Enabled	Определяет, доступна ли команда пользователю.
GroupIndex	Команды с одинаковым положительным значением GroupIndex согласовано переключают свойство Checked — установка у одной команды свойства **Checked** в значение **True** приводит к устновке его в значение False в других командах.
ImageIndex	Номер значка в списке **Images** компонента **ActionList**. Значок отображается рядом с текстом пункта меню (см. параграф 8.1.12). Отрицательное значение свойства **ImageIndex** говорит о том, что для пункта меню значок не задан. Свойство **ImageIndex** имеет приоритет над свойством **Bitmap**.
SecondaryShortCuts	Дополнительные комбинации клавиш.
ShortCut	Комбинация клавиш для выполнения команды.
Visible	Определяет, видна ли пользователю команда.
OnExecute	Происходит при выполнении команды.
OnHint	Происходит в момент появления всплывающей подсказки.
OnUpdate	Происходит при выполнении команды и периодически во время простоя программы. Позволяет отслеживать и изменять состояние команды.

*Таблица 8.18. Важнейшие свойства и события компонента **Action***

Шаг 74. Перейдем к настройке команды, созданной на предыдущем шаге. Дайте команде имя **OpenAction**, в свойстве **Caption** впишите текст **Open...** и в свойстве **ShortCut** выберите значение **Ctrl+O** (рисунок 8.79).

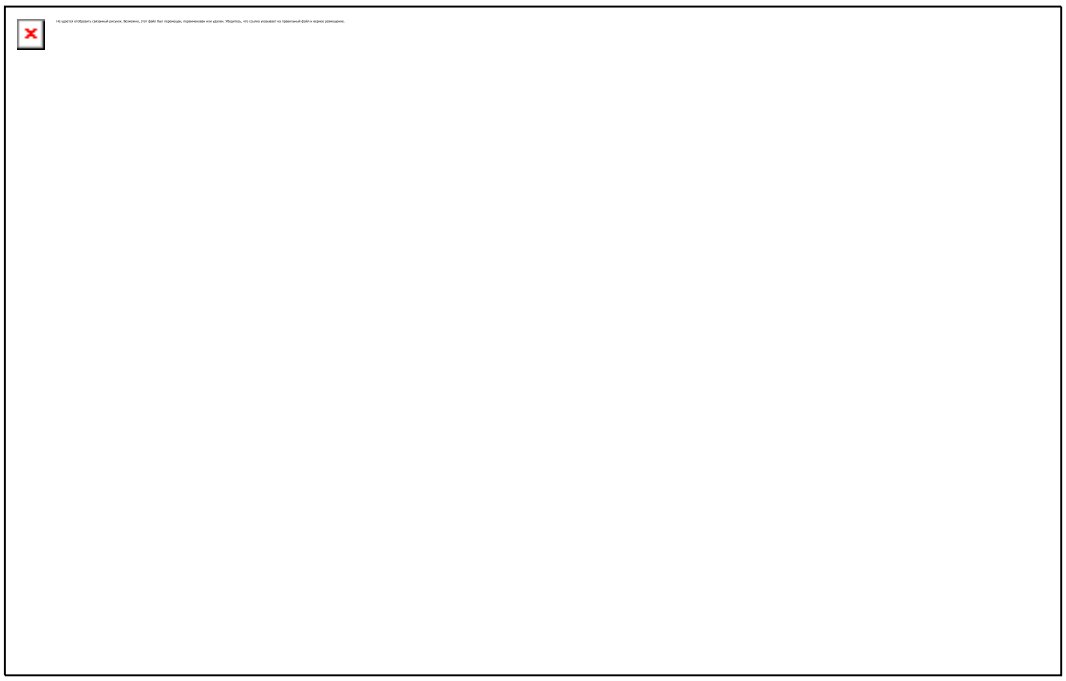


Рисунок 8.79. Для команды Open задана комбинация клавиш Ctrl+O

Команда может иметь значок. Он определяется значением свойства **ImageIndex** (номер значка в списке **Images** компонента **ActionList**). Прежде чем выбрать значение для свойства **ImageIndex**, нужно указать список значков компоненту **ActionList**.

Шаг 75. Выделите на форме компонент **ActionList** и перейдите к окну свойств. Выберите в свойстве **Images** значение **ImageList** (рисунок 8.80).

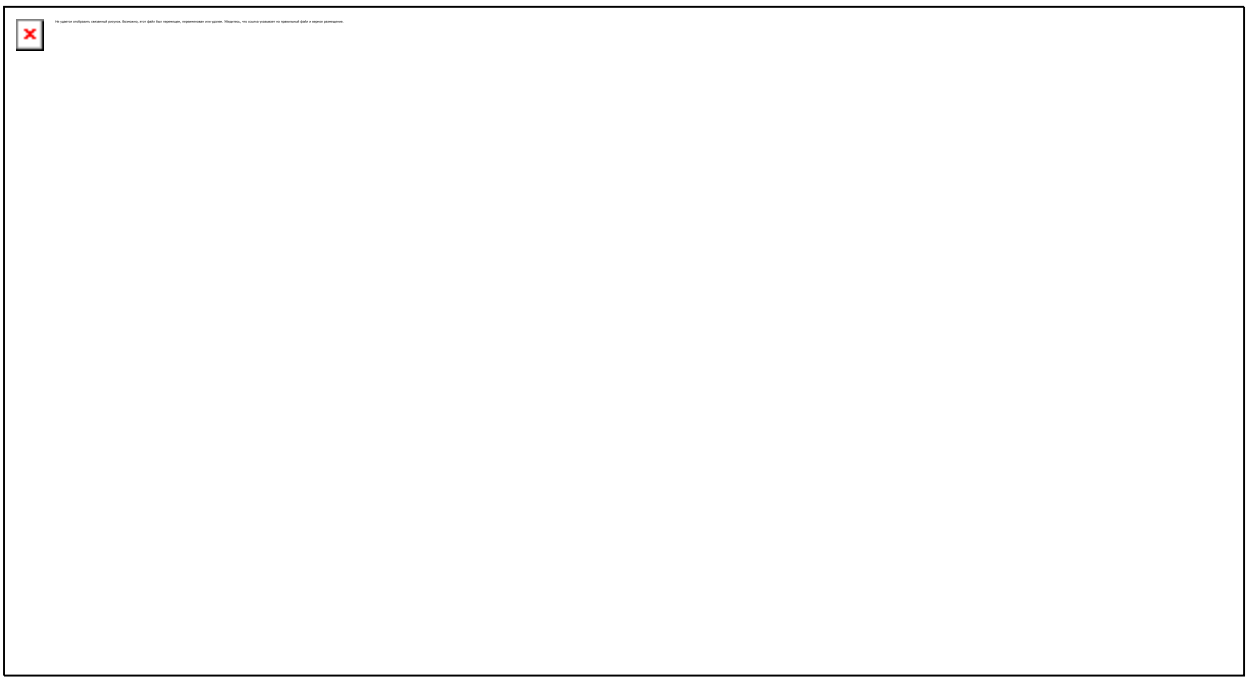


Рисунок 8.80. Для компонента ActionList задается компонент ImageList со списком значков

Шаг 76. А теперь установим компоненту **OpenAction** соответствующий значок. Перейдите к окну команд и выделите команду **OpenAction**. Затем в окне свойств отыщите свойство **ImageIndex** и выберите значок с номером 0 (рисунок 8.81).

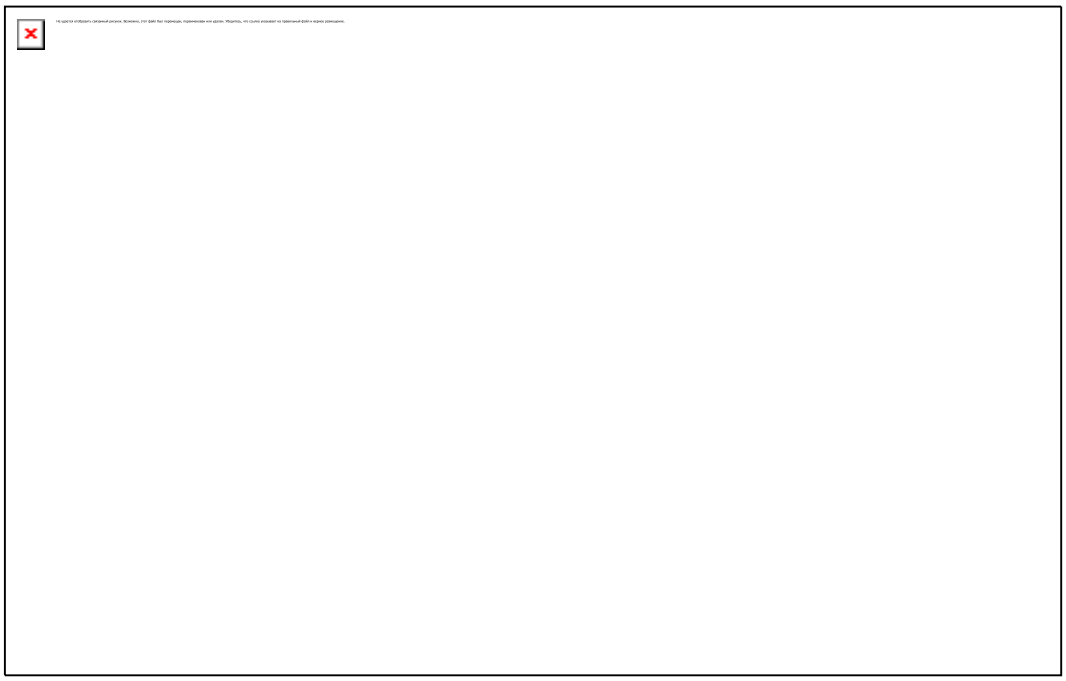


Рисунок 8.81. Для команды Open задан значок с индексом 0

Шаг 77. С командой **OpenAction** мы разобрались, теперь самостоятельно создайте команды **SaveAsAction**, **CloseAction**, **ExitAction**, **ToolBarAction**, **StatusBarAction**, **HalfSizeAction**, **NormalSizeAction**, **DoubleSizeAction** (рисунок 8.82) с соответствующими заголовками (свойство Caption).

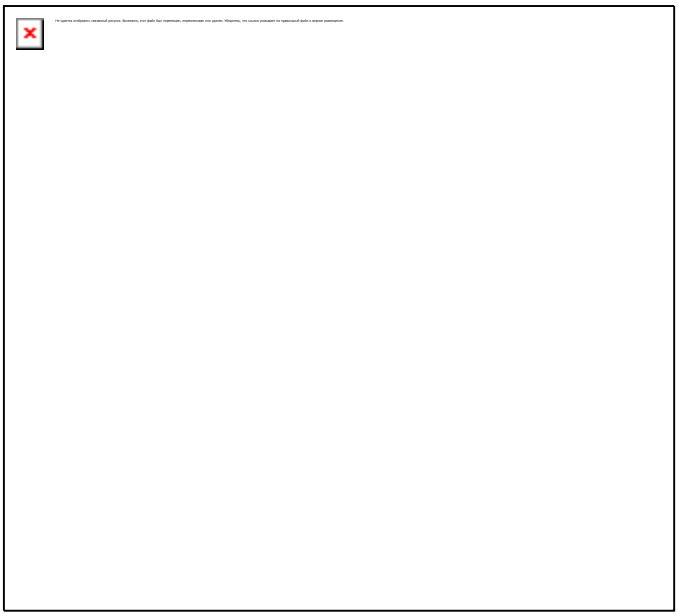


Рисунок 8.82. Полный список команд для программы

Самая ответственная часть работы завершена, список команд сформирован. Теперь привяжем команды к визуальным компонентам: кнопкам и пунктам меню.

8.6.3. Привязка команд

Кнопки, пункты меню и некоторые другие визуальные компоненты имеют свойство **Action**, с помощью которого к ним привязываются команды. В результате установки свойства **Action** визуальный компонент копирует к себе значения свойств команды (надпись, значок, подсказку и др.). Кроме того, команда запоминает, к каким компонентам она привязана с

тем, чтобы изменение свойств команды вызывало изменение соответствующих свойств во всех связанных с ней компонентах.

Шаг 78. Привязку команд начнем с кнопки **Open** на панели инструментов. Выделите ее и в раскрывающемся списке свойства **Action** выберите значение **OpenAction** (рисунок 8.83).

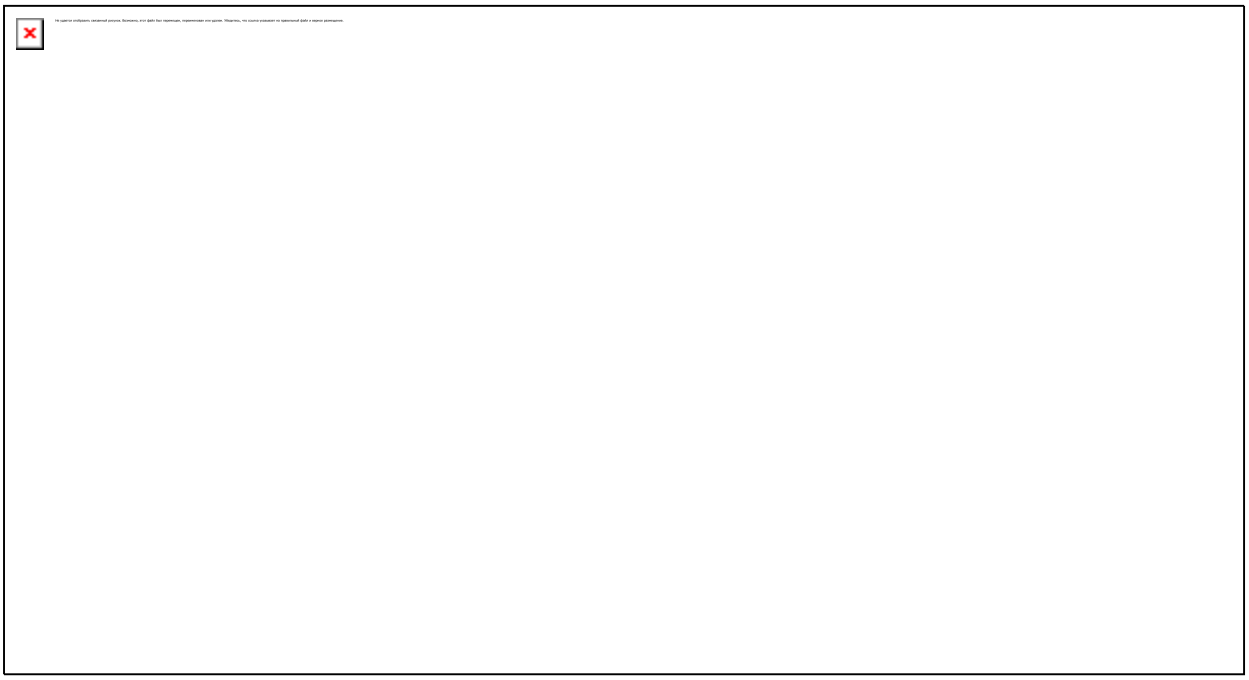


Рисунок 8.83. К кнопке OpenToolButton привязывается команда OpenAction

Обратите внимание, что надпись на кнопке изменилась. Это результат копирования значения свойства **Caption** из компонента **OpenAction** в компонент **OpenToolButton**.

Шаг 79. Аналогичным образом привяжите команду **OpenAction** к пункту **Open…** главного меню (рисунок 8.84).

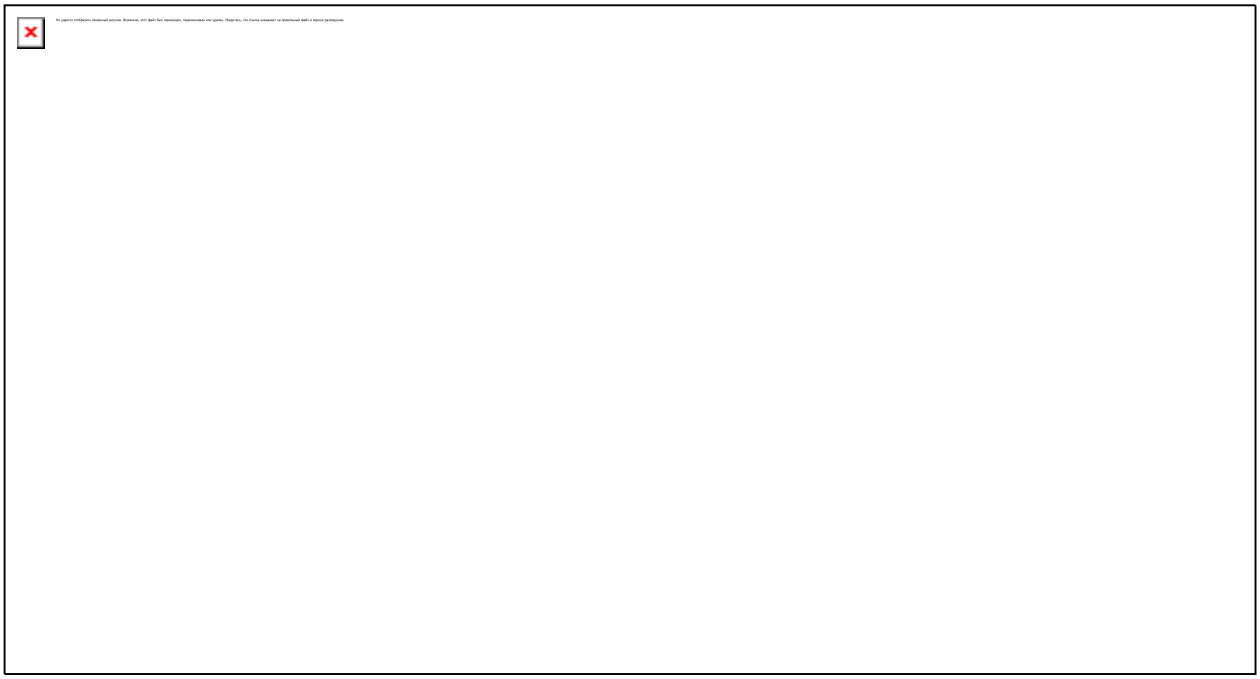


Рисунок 8.84. К пункту меню OpenMenuItem привязывается команда OpenAction

Проверим, что у нас получилось. Выполните компиляцию и запустите программу (рисунок 8.85).

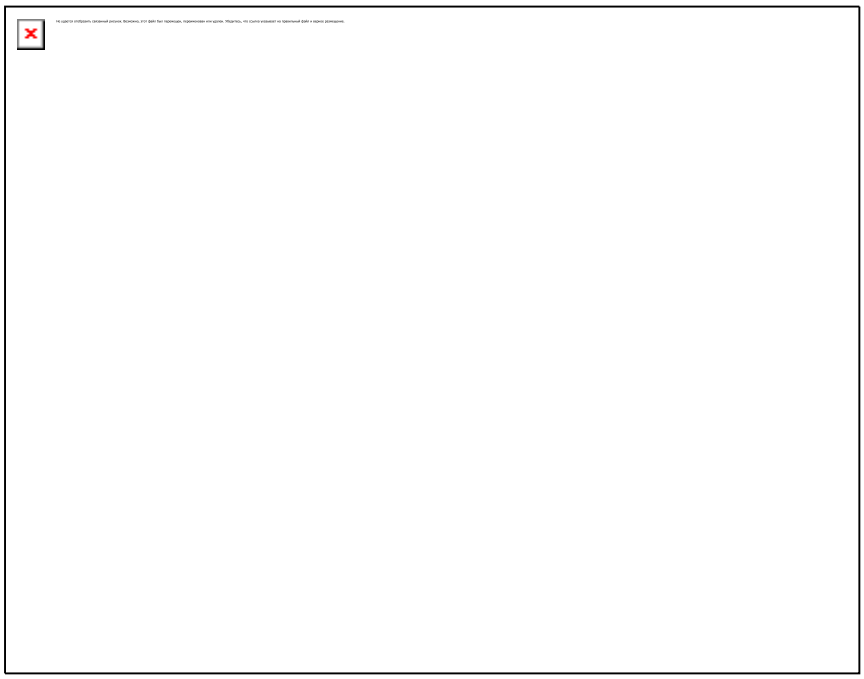


Рисунок 8.85. При запуске программы команда Open оказалась недоступна

Странно: и кнопка **Open…** на панели инструментов, и пункт **Open…** в главном меню недоступны. Это объясняется отсутствием у компонента **OpenAction** обработчика события **OnExecute**. Им сейчас и займемся.

8.6.4. Реакция на команды

Когда пользователь нажимает кнопку или выбирает пункт меню, происходит событие **OnExecute**. Если для команды не определен обработчик события **OnExecute**, то все компоненты, использующие эту команду, становятся недоступными (свойство **Enabled** устанавливается в значение **False**).

Шаг 80. Определим в компоненте **OpenAction** обработчик события **OnExecute**. Обратитесь к контекстному меню компонента **ActionList** и вызовите окно команд. В этом окне выберите команду **OpenAction**, после чего в окне свойств выберите вкладку **Events**. Теперь сделайте двойной щелчок мыши на значении события **OnExecute**. Среда Delphi создаст заготовку для будущего обработчика:

```
procedure TPictureForm.OpenActionExecute(Sender: TObject);
begin

end;
```

Обработчик у нас уже есть в виде метода **OpenMenuItemClick**, поэтому мы просто перенесем код этого метода (слегка подправив его) в только что созданный метод, удалив код метода **OpenMenuItemClick**.

```
procedure TPictureForm.OpenMenuItemClick(Sender: TObject);
begin
end;
...
procedure TPictureForm.OpenActionExecute(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    Image.Picture.LoadFromFile(OpenDialog.FileName);
    EnableCommands(True);
    NormalSizeAction.Execute; // Вместо NormalSizeMenuItem.Click;
  end;
  UpdateStatusBar;
end;
```

Сохраните проект; пустой метод **OpenMenuItemClick** будет автоматически удален из исходного текста.

Обратите внимание, что компонент **Action** автоматически подменяет обработчики **OnClick** в связанных с ним компонентах. Поэтому если вы перейдете к окну свойств и посмотрите на событие **OnClick** в компоненте **OpenMenuItem**, то обнаружите там метод **OpenActionExecute** (обработчик события **OnExecute** компонента **OpenAction**).

Выполните компиляцию и запустите программу. Команда **Open** снова доступна пользователю (рисунок 8.86).

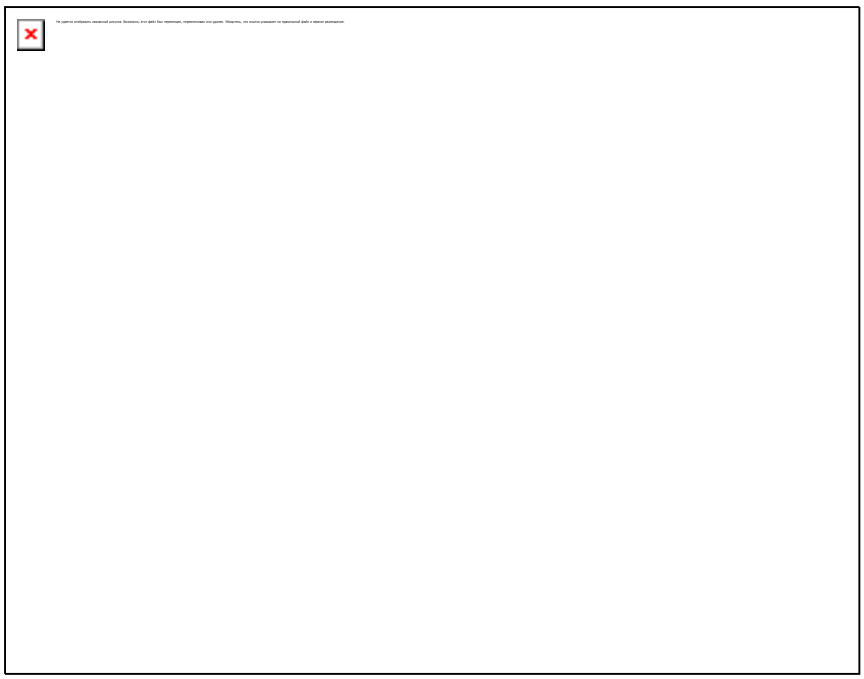


Рисунок 8.86. Команда Open опять доступна пользователю (компонент OpenAction обрабатывает событие OnExecute)

Закрыв программу, вернитесь к проекту в среде Delphi, чтобы продолжить настройку оставшихся команд.

Шаг 81. Обойдите все пункты меню (не забудьте про контекстное меню) и кнопки панели инструментов и установите в каждом из них свойство **Action** в соответствующее значение. Попутно значения некоторых других свойств тоже изменятся, например свойство **Enabled** получит значение **True**. Пусть вас это не беспокоит, так и должно быть (рисунок 8.87).

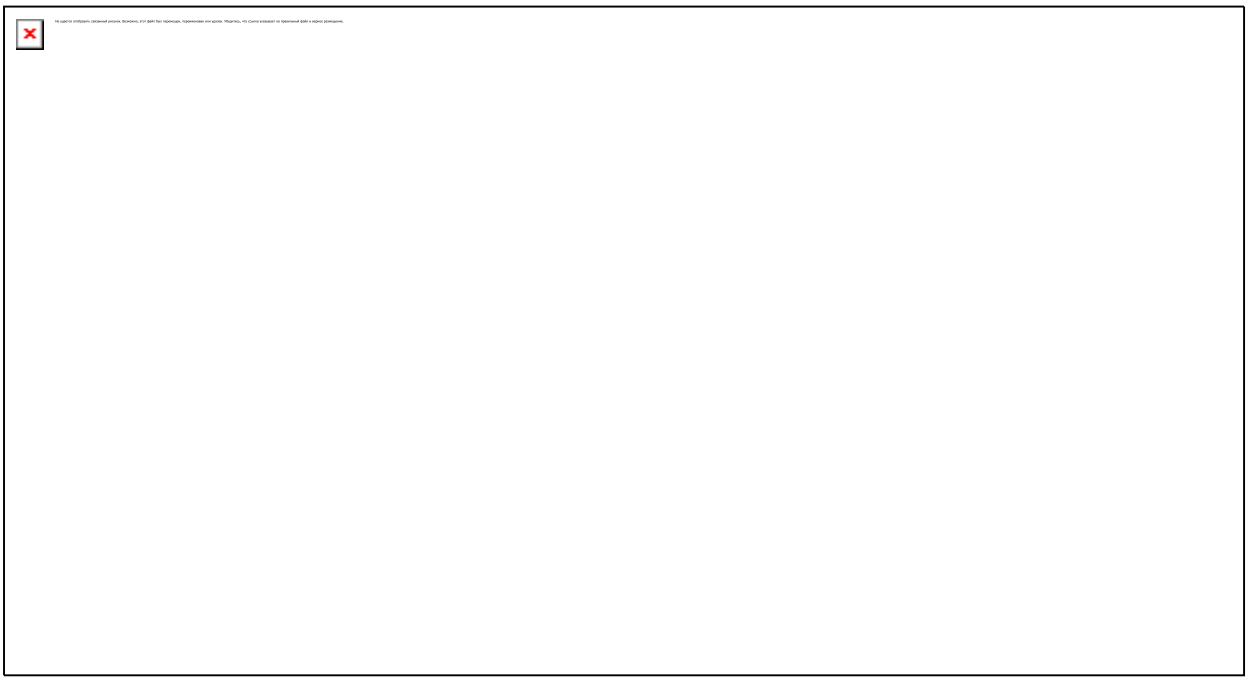


Рисунок 8.87. Все пункты меню и кнопки панели инструментов привязаны к командам

Восстановим правильную логику работы кнопок и пунктов меню.

Шаг 82. Сделайте недоступной команду **SaveAsAction**, установив ее свойство **Enabled** в значение **False**. Одновременно кнопка и пункт меню **Save As...** станут недоступными (рисунок 8.88).

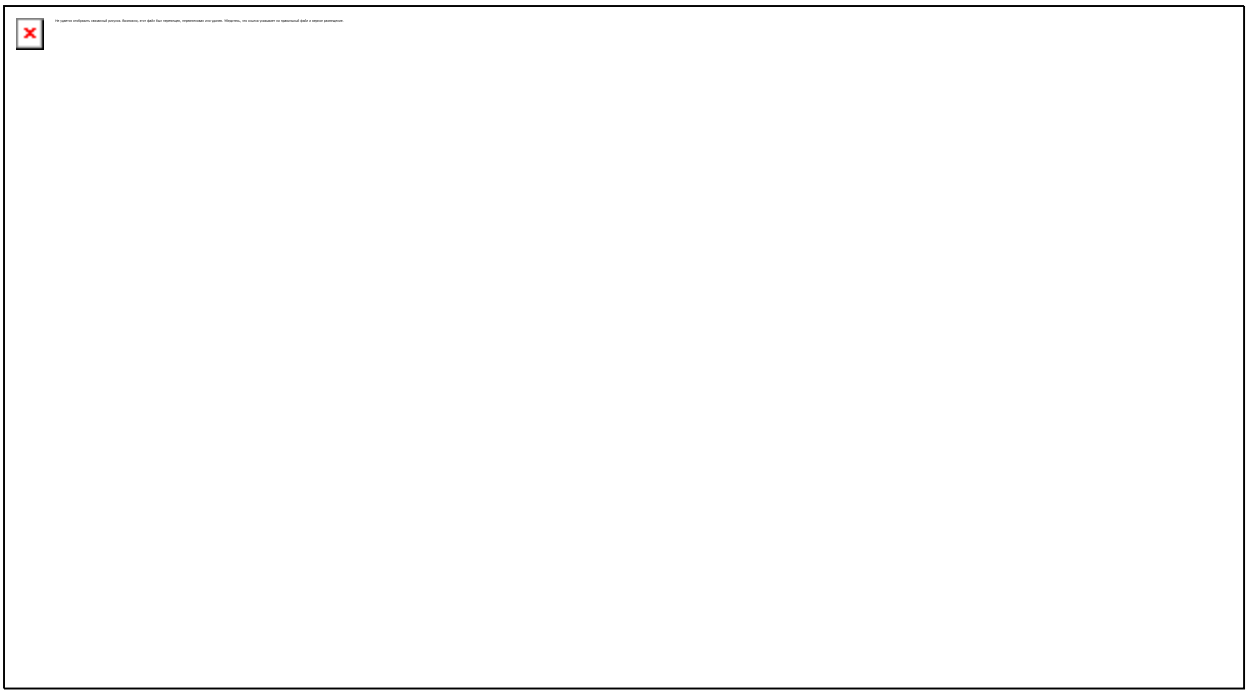


Рисунок 8.88. Команда Save As отключена до тех пор, пока пользователь не откроет какой-нибудь графический файл

Шаг 83. Создайте для компонента **SaveAsAction** обработчик события **OnExecute** и перенесите код метода **SaveAsMenuItemClick** в только что созданный метод **SaveAsActionExecute**:

```
procedure TPictureForm.SaveAsMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.SaveAsActionExecute(Sender: TObject);
begin
  if SaveDialog.Execute then
    Image.Picture.SaveToFile(SaveDialog.FileName);
end;
```

Шаг 84. Доработку команды **SaveAsAction** мы закончили и теперь по аналогии доработаем команды **ExitAction** и **CloseAction**:

```
procedure TPictureForm.ExitMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.CloseMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.ExitActionExecute(Sender: TObject);
begin
  Close;
end;

procedure TPictureForm.CloseActionExecute(Sender: TObject);
begin
  with Image do
  begin
    Picture := nil;
    Width := 0;
    Height := 0;
  end;
  NormalSizeAction.Execute; // Вместо NormalSizeMenuItem.Click;
  EnableCommands(False);
  UpdateStatusBar;
end;
```

Шаг 85. Теперь настало время команд **ToolBarAction** и **StatusBarAction**:

```
procedure TPictureForm.ToolBarMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.StatusBarMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.ToolBarActionExecute(Sender: TObject);
begin
  ToolBarAction.Checked := not ToolBarAction.Checked;
  ToolBar.Visible := not ToolBar.Visible;
end;

procedure TPictureForm.StatusBarActionExecute(Sender: TObject);
begin
  StatusBarAction.Checked := not StatusBarAction.Checked;
  StatusBar.Visible := not StatusBar.Visible;
end;
```

Теперь восстановим логику работы команд, отвечающих за масштаб рисунка.

Шаг 86. Вернитесь к окну редактирования списка команд и выделите команду **HalfSizeAction**. После этого нажмите клавишу **Ctrl** и, удерживая ее, выделите команды **NormalSizeAction** и **DoubleSizeAction**. Перейдите к окну свойств и установите свойство **GroupIndex** в значение 1 (рисунок 8.89).

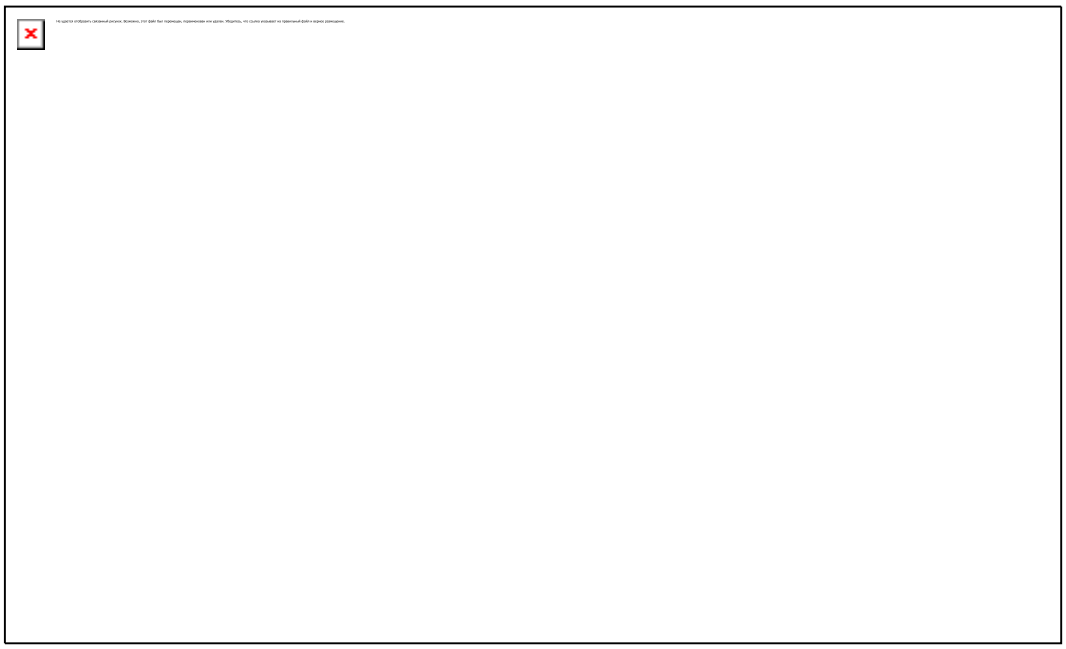


Рисунок 8.89. Группировка команд с помощью свойства GroupIndex

Шаг 87. Свойство **Checked** компонента **NormalSizeAction** установите в значение **True** — при запуске программы рисунок не масштабируется (рисунок 8.90).

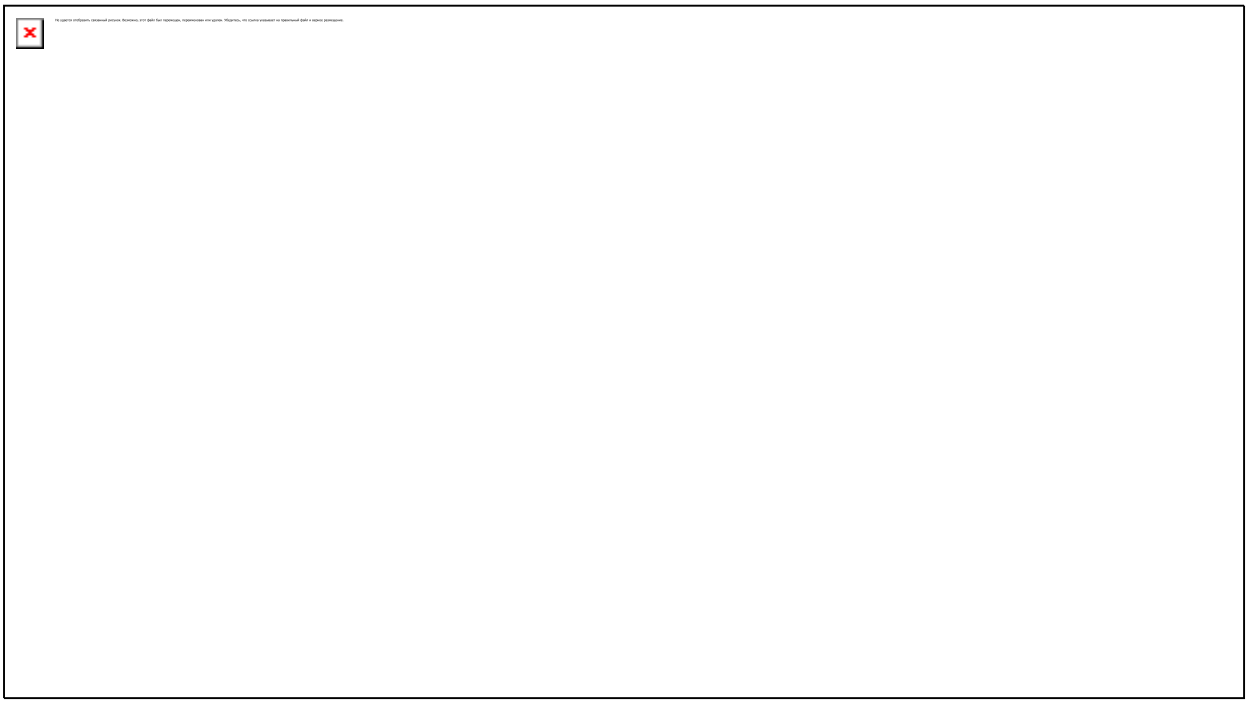


Рисунок 8.90. Начальное значение для переключателя масштаба — Normal Size

Шаг 88. Установите свойство **Enabled** команд **HalfSizeAction**, **NormalSizeAction** и **DoubleSizeAction** в значение **False** — при запуске программы рисунок еще не загружен, поэтому команды переключения масштаба должны быть недоступны (рисунок 8.91).

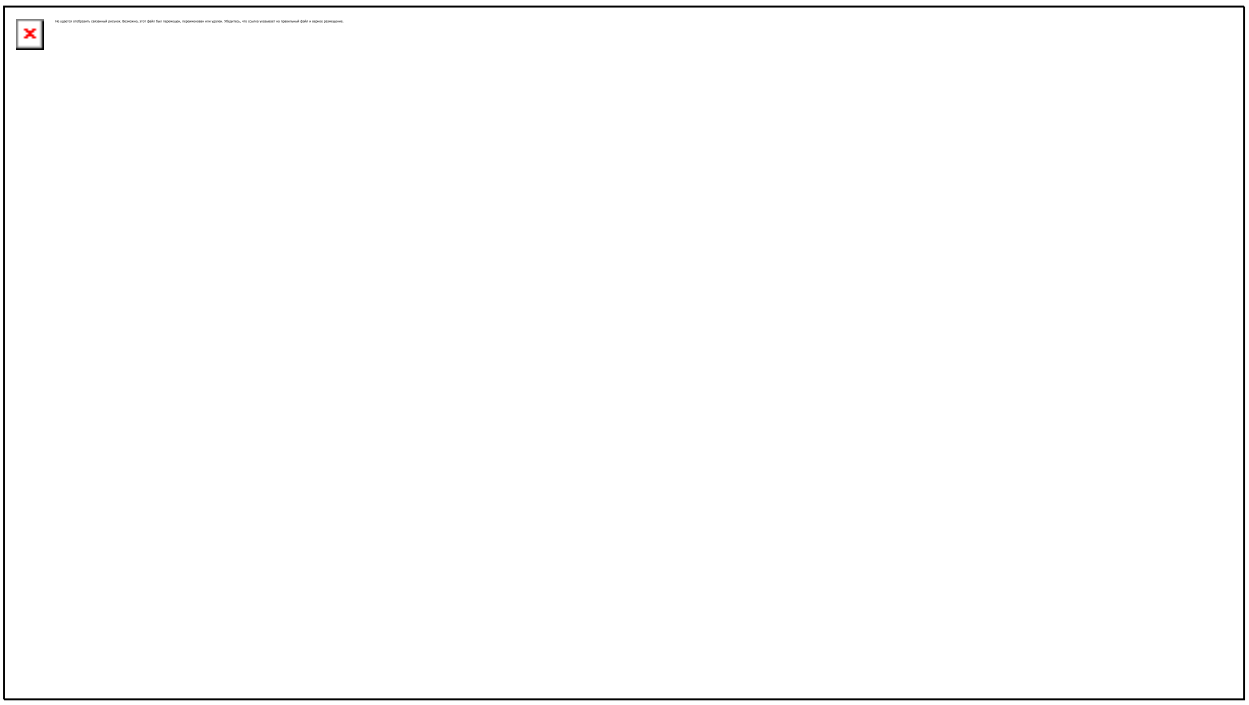


Рисунок 8.91. Команды переключения масштаба отключены до тех пор, пока пользователь не откроет какой-нибудь графический файл

Шаг 89. Теперь создадим обработчики команд **HalfSizeAction**, **NormalSizeAction** и **DoubleSizeAction**. Для каждой команды определите обработчик события **OnExecute** и перенесите код из уже имеющихся методов:

```
procedure TPictureForm.HalfSizeMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.NormalSizeMenuItemClick(Sender: TObject);
begin
end;

procedure TPictureForm.DoubleSizeMenuItemClick(Sender: TObject);
begin
end;

...

procedure TPictureForm.HalfSizeActionExecute(Sender: TObject);
begin
  HalfSizeToolButton.Down := True;
  HalfSizeMenuItem.Checked := True;
  HalfSizePopupItem.Checked := True;
  with Image do
  begin
    AutoSize := False;
    Width := Picture.Width div 2;
    Height := Picture.Height div 2;
    Stretch := True;
  end;
end;

procedure TPictureForm.NormalSizeActionExecute(Sender: TObject);
begin
  NormalSizeToolButton.Down := True;
  NormalSizeMenuItem.Checked := True;
  NormalSizePopupItem.Checked := True;
  Image.AutoSize := True;
end;

procedure TPictureForm.DoubleSizeActionExecute(Sender: TObject);
begin
  DoubleSizeToolButton.Down := True;
  DoubleSizeMenuItem.Checked := True;
  DoubleSizePopupItem.Checked := True;
  with Image do
  begin
    AutoSize := False;
    Width := Picture.Width * 2;
    Height := Picture.Height * 2;
    Stretch := True;
  end;
end;
```

Шаг 90. Обработчики можно упростить за счет управления состоянием пунктов меню и кнопок через компоненты **Action**, т.е. первые три оператора каждого обработчика заменяются на один оператор:

```
procedure TPictureForm.HalfSizeActionExecute(Sender: TObject);
begin
  HalfSizeAction.Checked := True;
  with Image do
  begin
    AutoSize := False;
    Width := Picture.Width div 2;
    Height := Picture.Height div 2;
    Stretch := True;
  end;
end;

procedure TPictureForm.NormalSizeActionExecute(Sender: TObject);
begin
  NormalSizeAction.Checked := True;
  Image.AutoSize := True;
end;

procedure TPictureForm.DoubleSizeActionExecute(Sender: TObject);
begin
  DoubleSizeAction.Checked := True;
  with Image do
  begin
    AutoSize := False;
    Width := Picture.Width * 2;
    Height := Picture.Height * 2;
    Stretch := True;
  end;
end;
```

А теперь воспользуемся свойством **AutoCheck** компонентов **Action**, чтобы избавиться от необходимости программно переключать метку в пунктах **Toolbar** и **Status bar** главного меню. Когда свойство **AutoCheck** равно **True**, то при выполнении команды свойство **Checked** автоматически меняет свое значение на противоположное. Это отражается на связанных с командой пунктах меню и кнопках-переключателях.

Шаг 91. У команд **ToolBarAction** и **StatusBarAction** установите свойства **AutoCheck** и **Checked** в значение **True**.

Шаг 92. Подправьте методы **ToolBarActionExecute** и **StatusBarActionExecute**:

```
procedure TPictureForm.ToolBarActionExecute(Sender: TObject);
begin
  ToolBar.Visible := ToolBarAction.Checked;
end;

procedure TPictureForm.StatusBarActionExecute(Sender: TObject);
begin
  StatusBar.Visible := StatusBarAction.Checked;
end;
```

8.6.5. Управление состоянием команд

Компонент **ActionList** имеет удобный механизм управления состоянием команд (например, доступна/недоступна). После выполнения очередной команды и во время простоя программы в компоненте возникает событие **OnUpdate**. Реакцией на это событие может быть изменение состояния отдельных команд, например переключение в них свойства **Enabled**. Напомним, что сейчас для этих целей используется метод **EnableCommand**, вызываемый при открытии и закрытии файла. Избавимся от него.

Шаг 93. Выделите на форме компонент **ActionList**, и в окне свойств на вкладке **Events** отыщите событие **OnExecute**. Двойным щелчком мыши создайте обработчик:

```
procedure TPictureForm.ActionListUpdate(Action: TBasicAction;
  var Handled: Boolean);
var
  NonEmpty: Boolean;
begin
  NonEmpty := Image.Picture.Graphic <> nil;
  SaveAsAction.Enabled := NonEmpty;
  CloseMenuItem.Enabled := NonEmpty;
  HalfSizeAction.Enabled := NonEmpty;
  NormalSizeAction.Enabled := NonEmpty;
  DoubleSizeAction.Enabled := NonEmpty;
  Handled := True;
end;
```

Шаг 94. Удалите метод **EnableCommands** и обращения к нему из методов **OpenActionExecute** и **CloseActionExecute**. Вот, что должно получиться:

```
procedure TPictureForm.OpenActionExecute(Sender: TObject);
begin
  if OpenDialog.Execute then
  begin
    Image.Picture.LoadFromFile(OpenDialog.FileName);
    NormalSizeAction.Execute;
  end;
  UpdateStatusBar;
end;

procedure TPictureForm.CloseActionExecute(Sender: TObject);
begin
  with Image do
  begin
    Picture.Graphic := nil;
    Width := 0;
    Height := 0;
  end;
  NormalSizeAction.Execute;
  UpdateStatusBar;
end;
```

Шаг 95. Программа полностью готова, выполните компиляцию и запустите ее. Наслаждайтесь результатами своего труда, просматривая рисунки на жестком диске (рисунок 8.92).

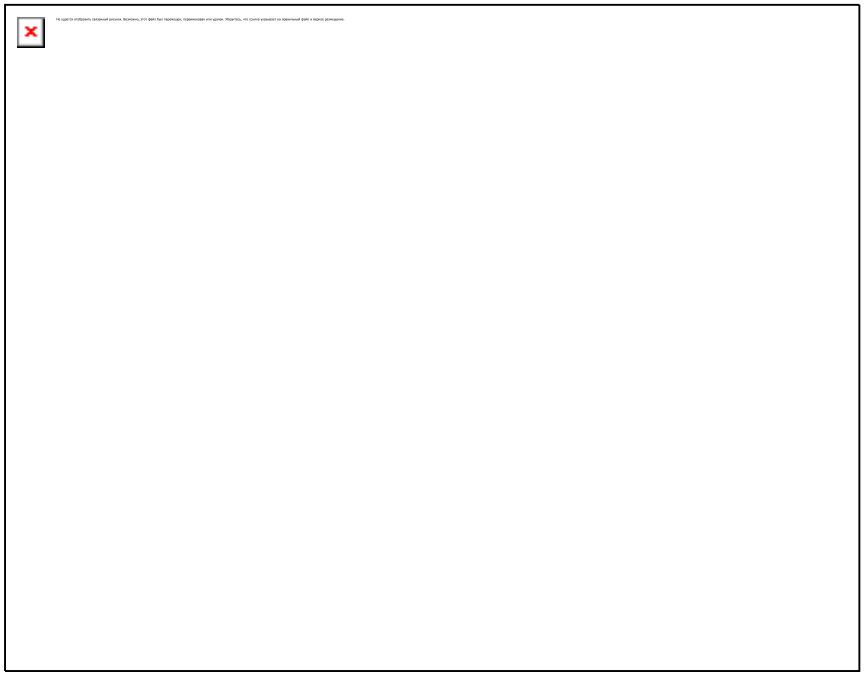


Рисунок 8.92. Окончательный вариант программы для просмотра графических файлов

Напоследок вернитесь к исходному тексту программы и взгляните на то, какими лаконичными стали обработчики событий. В них нет ничего лишнего. Все второстепенные вещи за вас сделали стандартные компоненты среды Delphi, а вы смогли сосредоточиться на главном — логике прикладной задачи.

8.7. Итоги

В этой главе вы в деталях изучили важнейшие средства управления программой — главное и контекстное меню, строку состояния, панель инструментов. Вы умеете их создать и должным образом настроить. Вы способны создать “хребет” любой программы, что и доказали на деле, разработав весьма неплохое приложение для просмотра картинок, которое наверняка пригодиться в практической работе. Храбро вставляйте его в свои проекты и пользуйтесь, постигайте мир графических изображений через окно вашего персонального вьюера. После столь серьезных успехов рекомендуем вам немного передохнуть и перейти к другой важной проблеме — организации диалога между программой и человеком.

Глава 9. Окна диалога

Все мы любим иногда поболтать. Это человеческое свойство передалось программам, и они частенько у вас что-то спрашивают, а вы им что-то отвечаете, иногда невпопад. “Беседа”, правда, идет текстом, а не голосом. Так вот, разговор между программой и пользователем называется диалогом. Организация диалога — важнейшая часть любой программы. Ваша прямая обязанность сделать этот диалог приятным. По форме диалог прост — появляется окно с некоторым сообщением, полем для ввода вашего ответа и кнопкой OK. Вы внимательно читаете сообщение, набираете строку-ответ и нажимаете кнопку OK. Вот и все. Создатели среды Delphi предусмотрели все возможные типы диалогов и создали для вас ряд великолепных “домашних заготовок”.

9.1. Понятие окна диалога

В основе диалога между пользователем и компьютером лежит *окно диалога (dialog box)* — форма, содержащая компоненты для ввода данных: кнопки, текстовые поля, флажки, переключатели, списки и др. С помощью этих компонентов пользователь просматривает и вводит данные. В среде Delphi окно диалога создается на основе обычной формы.

Окна диалога могут работать в одном из двух режимов, *монопольном* (иногда говорят модальном, от англ. modal) и *немонопольном* (немодальном, от англ. modeless).

Монопольное окно диалога не дает пользователю возможности переключиться на другие окна программы до тех пор, пока работа с ним не будет завершена. Сразу заметим, что это не мешает пользователю переключаться на другие программы, например, с помощью панели задач Windows или нажатием комбинации клавиш Alt+Tab. Большинство окон диалога работает в монопольном режиме.

Немонопольные окна диалога предоставляют пользователю свободу выбора, позволяя вводить данные сразу в нескольких окнах.

9.2. Окно "About"

9.2.1. Подготовка формы

Простейшим примером окна диалога является окно About ("О программе"). Как правило, оно открывается по команде меню **Help | About...** , работает в монопольном режиме и служит лишь для информирования пользователя. В предыдущей главе мы рассматривали программу PicView, там как раз не достает окна About. Исправим это упущение и на практике познакомимся с созданием простейших окон диалога.

Шаг 1. Запустите среду Delphi и откройте проект PictureViewer. Добавьте в главное меню пункт **Help** (программный идентификатор **HelpMenuItem**) с командой **About...** (программный идентификатор **AboutMenuItem**). По команде **About...** (рисунок 9.1) будет вызываться окно диалога About, которое мы дальше разработаем.

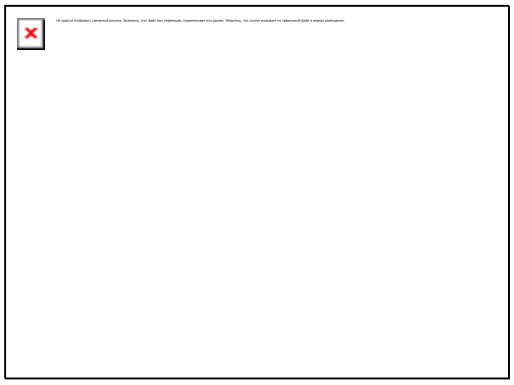


Рисунок 9.1. Пункт меню для вызова окна About

Шаг 2. Добавьте в проект новую форму, переименуйте ее в **AboutForm** и сохраните модуль под именем About.pas. Придайте форме нужные размеры и установите ее заголовок (свойство **Caption**) в значение **About Picture Viewer**. Далее сделаем из этой формы окно диалога.

Шаг 3. Обычная форма имеет много "фитюлек", которые совсем не нужны окну диалога, например раздвижную границу, меню управления окном, кнопки сворачивания и разворачивания окна. Чтобы их убрать, установите свойство **BorderStyle** в значение **bsDialog** (рисунок 9.2).

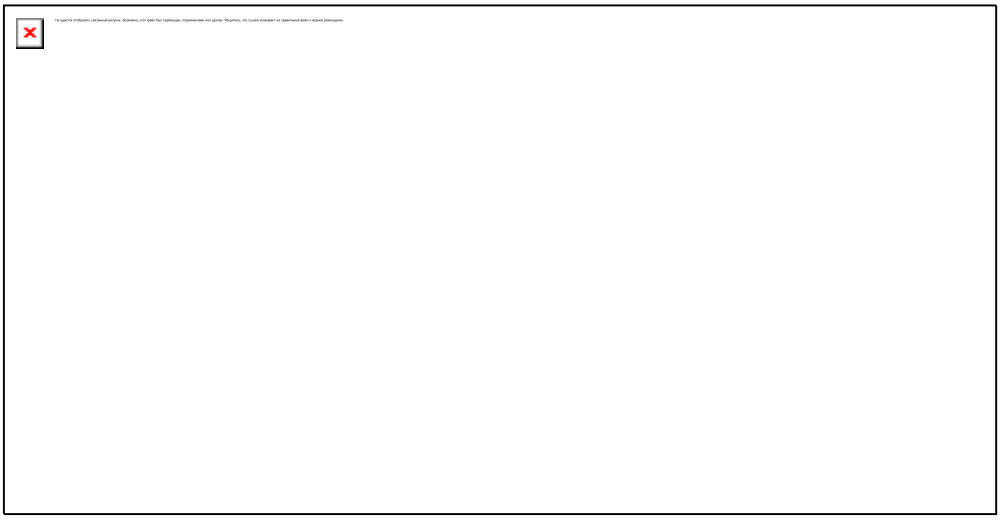


Рисунок 9.2. Превращение формы в окно диалога

Результат сделанного изменения проявится только во время работы программы и будет выражаться в следующем:

- у пользователя не будет возможности изменить размеры формы;
- у формы не будет кнопок сворачивания и разворачивания;
- в управляющем меню будут лишь два пункта: Move и Close.

Шаг 4. Большинство монопольных окон диалога появляются в центре экрана. За это у формы отвечает свойство **Position**. Изначально оно равно **poDesigned** — форма появляется точно в том же месте, где она находится во время разработки. Чтобы центрировать форму на экране, установите свойство **Position** в значение **poScreenCenter**. Заметим, что другие значения

свойства **Position** позволяют центрировать форму относительно главной формы, относительно формы-владельца или вообще не центрировать (см. параграф 7.3.4).

С формой разобрались, займемся компонентами. Окно About обычно содержит красивый рисунок, название программного продукта, замечания по поводу авторских прав или что-нибудь в этом роде и, конечно же, кнопку OK. Начнем с того, что добавим в форму кнопку.

9.2.2. Кнопка

Разрабатывая окно диалога, прежде всего необходимо обеспечить для пользователя возможность завершения диалога по окончание ввода данных. Вот тут как раз и нужны *кнопки (buttons)*. В простейшем окне диалога, каким является окно About, достаточно всего одной кнопки (она обычно называется OK или Close). В более сложных окнах может понадобиться несколько кнопок. Например, в том случае, когда окно диалога принимает от пользователя данные, в него помещают кнопки OK и Cancel, которые позволяют пользователю подтвердить или отменить результат диалога.

Обычная кнопка создается с помощью компонента **Button**, расположенного в палитре компонентов на вкладке **Standard**.

Рисунок 9.3. Компонент Button

Характерные свойства этого компонента кратко описаны в таблице 9.1.

Свойство	Описание
Action	Команда, хранящаяся в компоненте **ActionList** и выполняемая при нажатии кнопки.
Cancel	Если равно значению True, то кнопка срабатывает при нажатии клавиши Esc.
Caption	Текст на кнопке.
Default	Если равно значению True, то кнопка срабатывает при нажатии клавиши Enter. Исключением является ситуация, когда в окне диалога активна другая кнопка — в этом случае срабатывает она.
ModalResult	При нажатии кнопки значение этого свойства копируется в одноименное свойство формы, что приводит к закрытию монопольного окна диалога. Однако для этого значение свойства должно отличаться от mrNone.
WordWrap	Если равно значению True, то работает перенос слов.

Таблица 9.1. Важнейшие свойства компонента Button

Текст на кнопке определяется значением свойства **Caption**. В тексте может присутствовать символ **&**, который не пишется на кнопке, а обеспечивает подчеркивание следующего за ним символа. Нажатие подчеркнутого символа на клавиатуре в комбинации с клавишей Alt вызывает срабатывание кнопки. Например, если свойство **Caption** содержит строку **&Yes**, то

на кнопке будет написано **Yes**, и для нажатия кнопки можно воспользоваться комбинацией клавиш **Alt+Y**.

Пользователи, привыкшие работать с клавиатурой, знают, что одна из кнопок в окне диалога срабатывает при нажатии клавиши Enter. Это происходит при условии, что кнопка содержит значение True в свойстве **Default**. Такая кнопка, как правило, содержит текст OK и внешне отличается от остальных наличием жирной рамки.

Одна из кнопок окна диалога может срабатывать при нажатии клавиши Esc. Это происходит, если кнопка содержит значение True в свойстве Cancel. Как правило, такая кнопка подписывается текстом "Отмена" (Cancel).

Когда пользователь нажимает кнопку, в компоненте **Button** происходит событие **OnClick**. В обработчике этого события вы можете завершить диалог. Завершение немонопольного диалога выполняется вызовом метода **Close** у объекта формы. Завершение монопольного окна диалога выполняется установкой ненулевого значения в свойстве **ModalResult** формы. Впрочем, без этого можно обойтись, если воспользоваться свойством **ModalResult** компонента **Button** (мы не ошиблись, это свойство имеет и форма, и кнопка). Свойство **ModalResult** компонента **Button** устанавливается в окне свойств и по умолчанию равно значению **mrNone**. Если вы выберите другое значение, то нажатие кнопки будет вызывать автоматическое завершение диалога с копированием этого значения в свойство **ModalResult** формы. Анализируя свойство **ModalResult** после завершения диалога, программа узнает, какую кнопку нажал пользователь и в соответствии с этим направляет работу программы в нужное русло.

Помните, что свойство ModalResult работает только в монопольных окнах диалога.

Шаг 5. Перейдем теперь от теории к практике и создадим в нашем окне About кнопку OK (рисунок 9.4). Для этого выберите в палитре компонентов компонент **Button**, опустите его в форму и установите его свойства следующим образом:

Cancel = True

Caption = OK

Default = True

ModalResult = mrOk

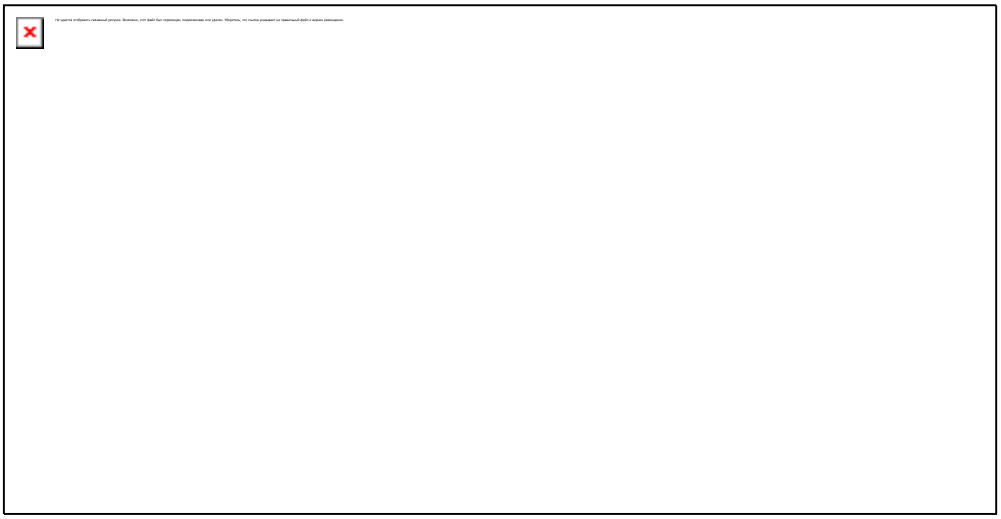


Рисунок 9.4. Кнопка OK в окне About

9.2.3. Кнопка с рисунком

Каждый из нас в душе художник. Поэтому рано или поздно стандартные невзрачные кнопки, содержащие лишь “голый” текст, перестанут вам нравиться. Появится естественное желание

их как-то приукрасить. В этом случае мы советуем вам вместо компонента **Button** воспользоваться компонентом **BitBtn**. Он расположен в палитре компонентов на вкладке **Additional**.

Рисунок 9.5. Компонент BitBtn

Компонент **BitBtn** обладает теми же возможностями, что и компонент **Button**, но кроме текста может содержать значок, который придает кнопке более привлекательный вид. По сравнению с компонентом **Button** компонент **BitBtn** имеет некоторые новые свойства, которые отражены в таблице 9.2.

Свойство	Описание
Glyph	Значок на кнопке.
NumGlyphs	Количество вариантов значка. Компонент делит рисунок Glyph по горизонтали на заданное количество значков и рисует один из них в зависимости от состояния кнопки.
Layout	Положение значка относительно текста: blGlyphLeft — слева, blGlyphRight — справа, blGlyphTop — сверху, blGlyphBottom — снизу.
Margin	Расстояние от границы кнопки до значка. Если оно равно -1, то значок вместе с текстом центрируются на кнопке.
Spacing	Расстояние от значка до текста. Если оно равно -1, то текст центрируется между значком и границей кнопки.
Kind	Задает кнопку стандартного вида. Упрощает создание таких стандартных кнопок, как OK, Cancel, Yes, No, Close, Abort, Retry, Ignore, All, Help.

*Таблица 9.2. Характерные свойства компонента **BitBtn***

Заметим, что кнопка, содержащая значок, принимает наиболее красивый вид, если свойство **Margin** равно значению 4, а свойство **Spacing** равно значению 1.

С помощью компонента **BitBtn** стандартные кнопки OK, Cancel, Yes, No, Close, Abort, Retry, Ignore, All и Help создаются проще, чем при использовании компонента **Button**. Для этого в свойстве **Kind** достаточно выбрать одно из значений, приведенных в таблице 9.3.

Значение	Вид кнопки	Результат установки значения	Пояснения
bkOK		Caption = 'OK' Default = True ModalResult = mrOK	Кнопка, подтверждающая ввод данных.
bkCancel		Caption = 'Cancel' Cancel = True ModalResult =	Кнопка, отменяющая ввод данных.

		mrCancel	
bkYes		Caption = '&Yes' Default = True ModalResult = mrYes	Кнопка для положительного ответа на вопрос.
bkNo		Caption = '&No' Cancel = True ModalResult = mrNo	Кнопка для отрицательного ответа на вопрос.
bkAll		Caption = '&All' ModalResult = mrAll	Кнопка для положительного ответа на все вопросы.
bkAbort		Caption = 'Abort' ModalResult = mrAbort	Кнопка для прерывания операции.
bkRetry		Caption = '&Retry' ModalResult = mrRetry	Кнопка для повторения операции.
bkIgnore		Caption = '&Ignore' ModalResult = mrIgnore	Кнопка для игнорирования произошедших изменений и продолжения начатой операции.
bkHelp		Caption = '&Help'	Кнопка для вызова справочника. Вы можете вызывать справочник в обработчике события OnClick или возложить эту работу на среду Delphi, установив у формы свойство HelpContext в нужное значение.
bkClose		Caption = '&Close'	Кнопка, закрывающая форму.
bkCustom	Любой	Любой	Кнопка для ваших собственных целей.

*Таблица 9.3. Значения свойства **Kind** компонента **BitBtn***

Например, если вам нужна кнопка OK, установите свойство **Kind** в значение bkOK. В результате на кнопке появится зеленая "галочка" и текст "OK", свойство **Default** получит значение True и свойство **ModalResult** получит значение mrOK.

9.2.4. Украшение окна диалога рисунком

Хорошее приложение должно быть не только функционально, но и эстетично. Как говорится, хорошее блюдо должно быть красиво подано. Это относится и к нашему

замечательному приложению, и конечно к его окну About. А что может лучше украсить окно About, чем яркий, запоминающийся рисунок.

Шаг 6. Как вы уже знаете, рисунок создается с помощью компонента **Image**, расположенного в палитре компонентов на вкладке **Additinal**. Выберите этот компонент и поместите его в форму **AboutForm**.

Шаг 7. Установите свойство **AutoSize** в значение True, чтобы компонент автоматически подгонял свои размеры под размеры рисунка, и установите свойство **Transparent** в значение True, чтобы рисунок отображался с прозрачным фоном.

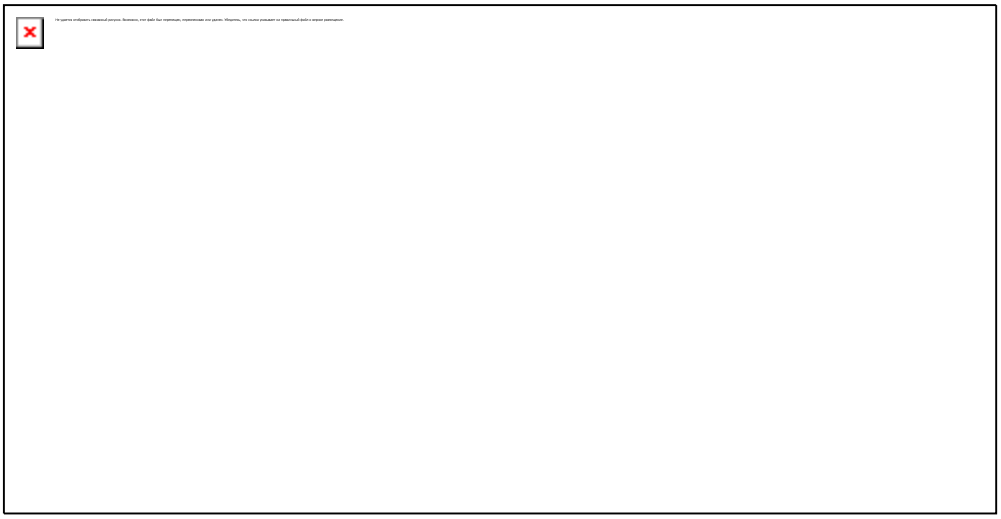


Рисунок 9.6. Прозрачный фон для рисунка задается установкой свойства Transparent в значение True

Шаг 8. Чтобы установить рисунок, перейдите к свойству **Picture** и нажатием кнопки с многоточием откройте окно **Picture Editor**. Это окно должно быть вам уже знакомо. Загрузите файл Athena.bmp из папки "C:\Program Files\Common Files\Borland Shared\Images\Splash\16Color". Рисунок появится в форме (рисунок 9.7).

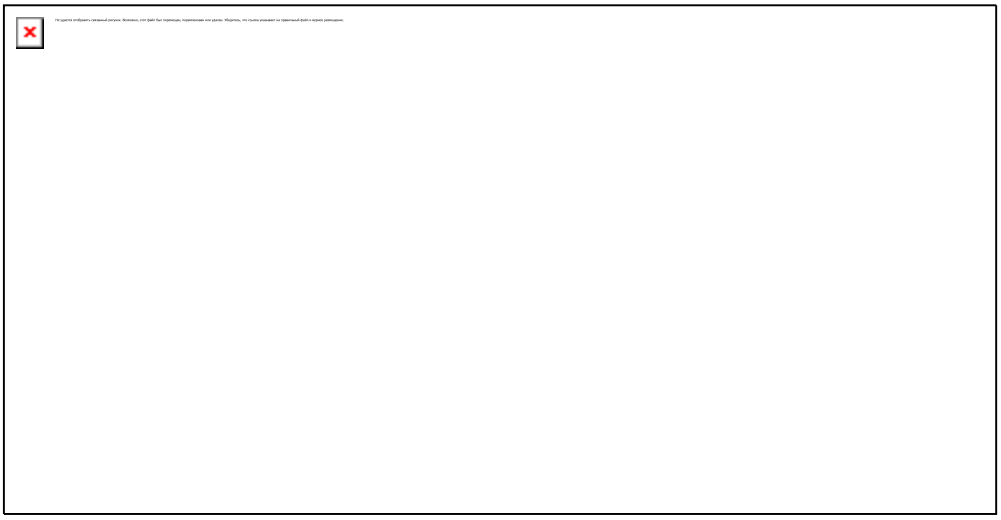


Рисунок 9.7. В компоненте Image загружен рисунок

9.2.5. Текстовая надпись

Сейчас мы подошли к созданию текстовых надписей в окне About. Они будут сообщать пользователю о названии программы и средствах ее разработки. Для нашей учебной задачи этого достаточно, в реальных приложениях окно About может дополнительно содержать

замечание об авторских правах, серийный номер и другую регистрационную или лицензионную информацию.

Следуя традиции, напишем название программы большими жирными буквами, а название средства разработки — обычным мелким шрифтом. Для этого воспользуемся компонентом **Label**, который находится в палитре компонентов на вкладке **Standard**.

Рисунок 9.8. Компонент Label

Характерные свойства компонента **Label** кратко описаны в таблице 9.4.

Свойство	Описание
Align	Способ выравнивания в пределах содержащего компонента.
Alignment	Выравнивание текста в пределах компонента: taLeftJustify — прижат к левой границе, taRightJustify — прижат к правой границе, taCenter — центрирован.
AutoSize	Если равно значению True, то размеры компонента автоматически подгоняются по ширине и высоте текста.
Caption	Текст надписи. С помощью символа **&** в тексте может быть задана “горячая” клавиша.
FocusControl	Компонент формы, получающий фокус ввода при нажатии “горячей” клавиши.
ShowAccelChar	Если равно значению True, то записанный в тексте символ **&** транслируется в подчеркивание следующего за ним символа. Подчеркнутый символ используется в комбинации с Alt как “горячая” клавиша.
Transparent	Если равно значению True, то фон надписи является прозрачным. Прозрачный фон полезен при наложении надписи на рисунок.
WordWrap	Если равно значению True, то работает перенос слов.

*Таблица 9.4. Важнейшие свойства компонента **Label***

Компонент **Label** отображает нередактируемый текст, хранящийся в свойстве **Caption**. Текст выравнивается в пределах компонента одним из трех способов: по левому краю, по правому краю или по центру. Способ выравнивания определяется свойством **Alignment**. Если текст надписи слишком велик, можно организовать его вывод в несколько строк (с переносом слов). Для этого достаточно установить свойство **WordWrap** в значение True. Еще одна удобная возможность — автоматическая подгонка размеров компонента по ширине и высоте текста. Она контролируется свойством **AutoSize** и по умолчанию включена.

Фон надписи можно сделать прозрачным, установив свойство **Transparent** в значение True. Такую надпись можно вынести наверх графического изображения, в результате получится текст на фоне рисунка.

С помощью компонента Label часто создаются подсказки к другим компонентам, в частности к полям ввода. При этом свойство **Caption** содержит не только текст, но и “горячую” клавишу, при выборе которой активизируется связанный с надписью компонент. Активизируемый компонент указывается в свойстве **FocusControl**.

Шаг 9. Вспомним, что компонент **Label** понадобился нам для того, чтобы сделать необходимые надписи в окне About. Опустите в форму первую надпись справа от изображения и установите ее свойства следующим образом:

WordWrap = True

Caption = Picture Viewer

Font Color = clNavy

Font Name = Times New Roman

Font Size = 20

Font Style = [fsBold]

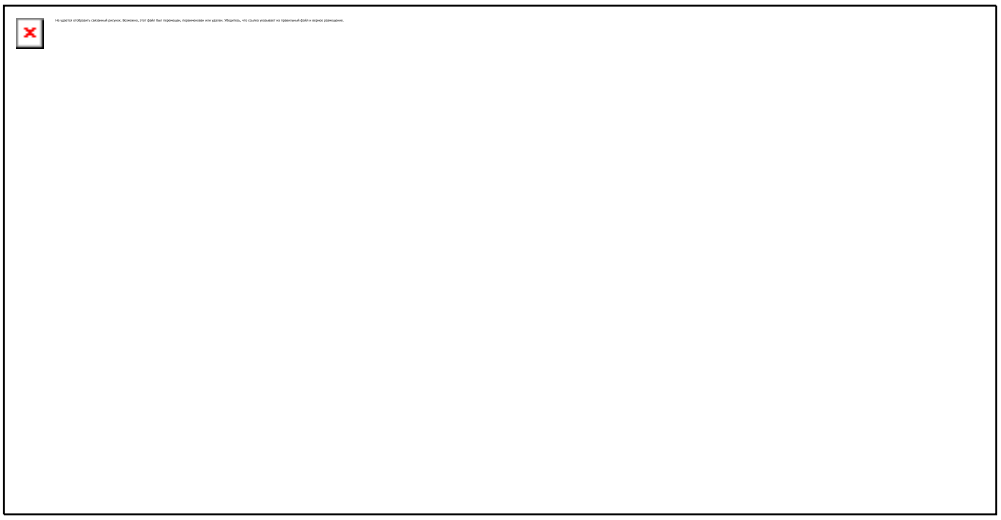


Рисунок 9.9. Надпись выполнена с помощью компонента Label

Шаг 10. После этого поместите в форму еще один компонент **Label** с текстом "Developed in Delphi" в свойстве **Caption**.

9.2.6. Рельефная канавка

Окно диалога почти готово, но для полного ажура не хватает одной мелочи — рельефной канавки вокруг рисунка и надписей (это придаст окну законченность). Для решения подобных задач служит компонент **Bevel**, расположенный в палитре компонентов на вкладке Additional.

Рисунок 9.10. Компонент Bevel

Шаг 11. Поместите в форму компонент **Bevel**, придайте ему нужные размеры и положение, после чего установите свойство **Shape** в значение bsFrame.

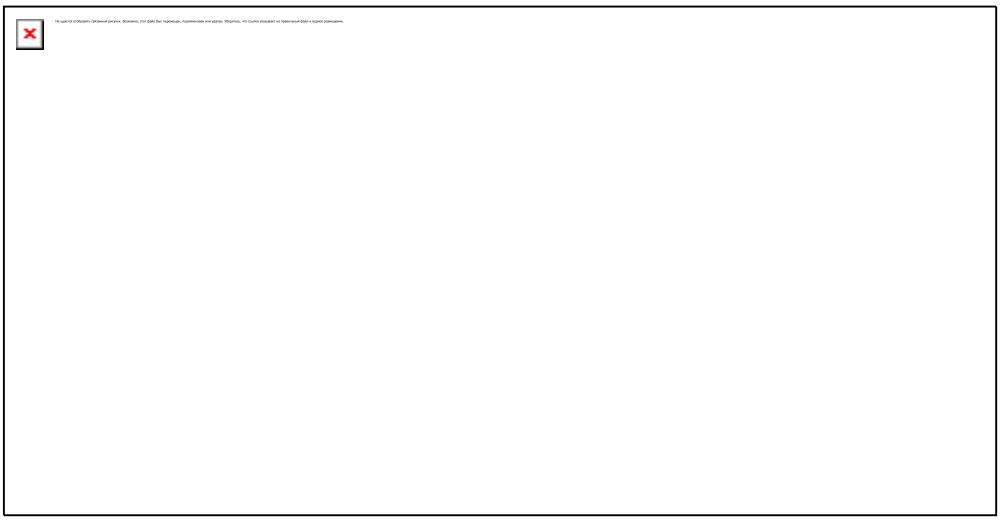


Рисунок 9.11. Компонент Bevel в форме

9.2.7. Рельефная панель

Рельефные канавки удобно создавать с помощью компонента **Bevel**. Однако компонент **Bevel** не может быть контейнером для других компонентов, а следовательно, перемещение рамки не вызывает перемещение компонентов, находящихся внутри нее. Если нужна не просто рамка, а контейнер с рамкой, то пользуются рельефной панелью — компонентом **Panel** (вкладка **Standard** панели инструментов).

Рисунок 9.12. Компонент Panel

Отличительные свойства компонента **Panel** сведены в таблицу 9.5.

Свойство	Описание
Align	Способ выравнивания панели в пределах владельца.
BevelInner	Внутренний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelOuter	Внешний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelWidth	Ширина скосов рельефной рамки.
BorderWidth	Расстояние в пикселях между внутренним и внешним скосами.
BorderStyle	Определяет, имеет ли панель рамку.
Caption	Текст на панели.
DockSite	Определяет, используется ли панель для стыковки

	других компонентов.
FullRepaint	Свойство сохранено для совместимости с предыдущими версиями библиотеки VCL. Оно не влияет на способ перерисовки компонента.
Locked	Если равно False, то OLE-серверу разрешено заменить панель на свою панель инструментов. Если равно True и панель выравнена по какой-нибудь стороне формы, то она остается нетронутой при активизации OLE-сервера по месту.
UseDockManager	Включает режим автоматического размещения стыкуемых компонентов на панели. Компоненты стыкуются методом деления панели по горизонтали и вертикали. Если свойство равно значению False, то программист должен сам позаботиться о размерах и местоположении стыкуемых компонентов, определив обработчики событий **OnGetSiteInfo** и **OnDockDrop**.
OnCanResize	Происходит при попытке изменить размеры панели. Запрос на изменение размеров может исходить от пользователя. Устанавливая значение параметра **Resize**, обработчик события OnCanResize может разрешить или запретить действительное изменение размеров панели.
OnConstrainedResize	Происходит при изменении размеров панели и позволяет на лету изменять ее минимальные и максимальные размеры.
OnGetSiteInfo	Происходит, когда у компонента запрашивается место для стыковки.

Таблица 9.5. Важнейшие свойства компонента Panel

Шаг 12. Уберите компонент **Bevel** из формы и поместите на его место компонент **Panel**. Откорректируйте его положение и размеры и установите свойства следующим образом (рисунок 9.13):

Caption = <пусто>

BevelInner = **bvRaised**

BevelOuter = **bvLowered**

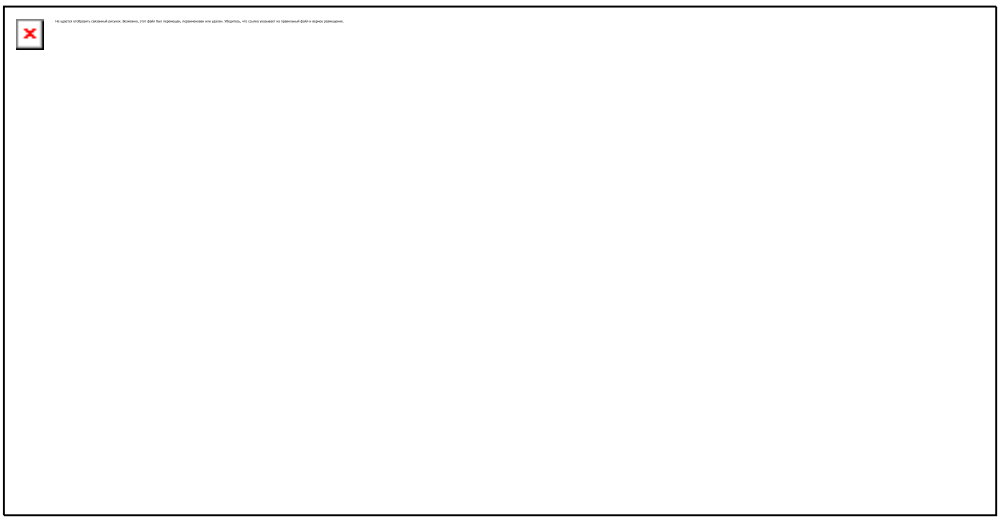


Рисунок 9.13. Компонент Panel заменил в форме компонент Bevel

Шаг 13. С помощью окна **Object TreeView** перенесите компоненты **Image1**, **Label1** и **Label2** на панель **Panel1** (рисунок 9.14).

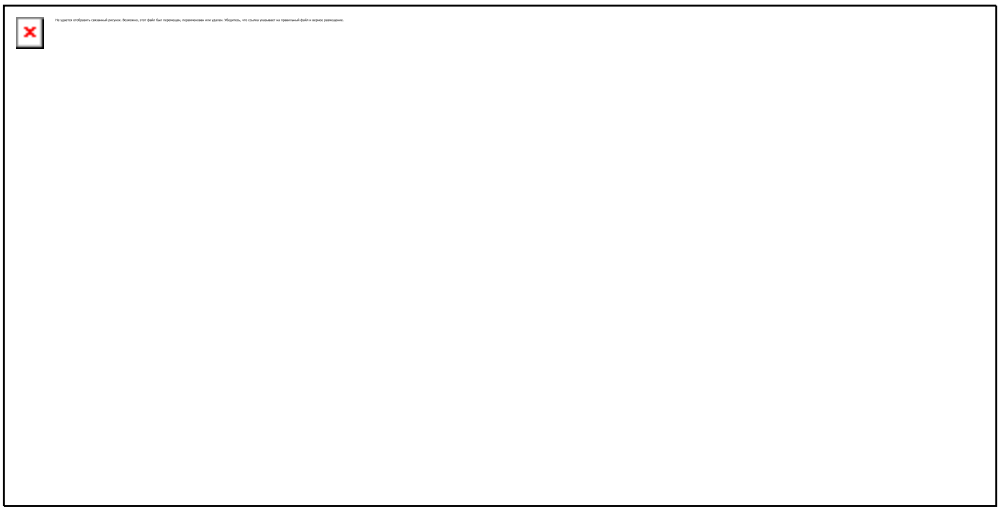


Рисунок 9.14. Компоненты Image1, Label1 и Label2 переносятся на панель Panel1

Теперь рельефная рамка заменена рельефной панелью и при ее перемещении перемещаются все надписи и рисунок (рисунок 9.15).

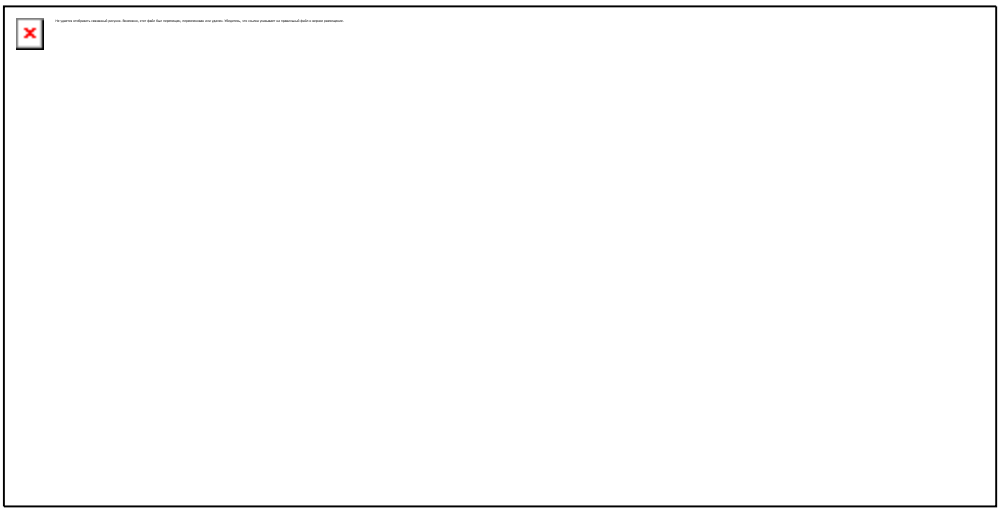


Рисунок 9.15. Компоненты Image1, Label1 и Label2 — теперь на панели Panel1

Будьте аккуратны при удалении панели! Вместе с ней всегда удаляются внутренние компоненты.

9.2.8. Выполнение диалога

Визуальное проектирование окна About закончено, осталось обеспечить его вызов при выборе пользователем команды **Help | About...** главного меню. Для этого нужно сделать следующее:

- создать команду **AboutAction** в компоненте **ActionList** и связать ее с пунктом меню **About...**;
- в модуле MainUnit подключить модуль AboutUnit. Это обеспечит доступ к форме **AboutForm** из главной формы **PictureForm**;
- создать обработчик события **OnExecute** команды **AboutAction** и обеспечить в нем монопольное выполнение диалога.

Шаг 14. Как реализовать первый пункт плана вы уже знаете, поэтому мы не будем на нем останавливаться, и сразу перейдем ко второму пункту плана. Активизируйте **PictureForm**, а затем выберите в меню **File** команду **Use Unit...** . На экране появится окно (рисунок 9.16):

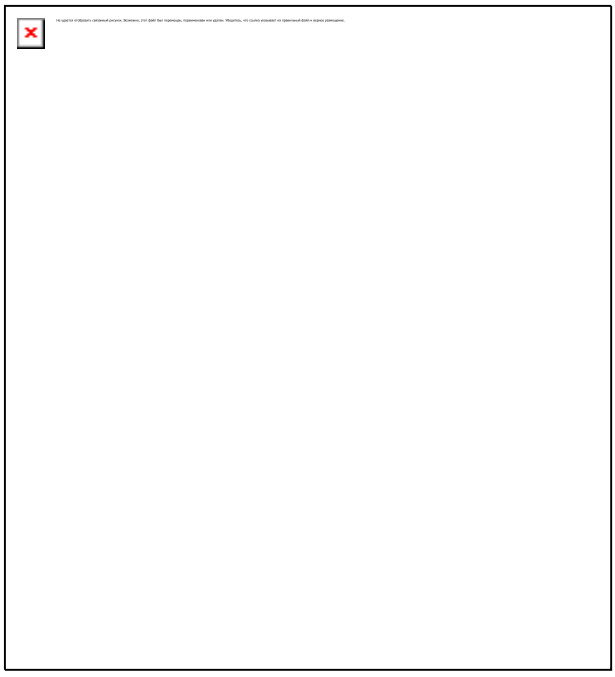


Рисунок 9.16. С помощью окна Use Unit модуль About подключается в модуле Main

Выберите в этом окне модуль AboutUnit и щелкните на кнопке OK. Модуль AboutUnit, содержащий определение формы **AboutForm**, подключится в модуле MainUnit, содержащем определение главной формы **PictureForm**. Чтобы в этом убедиться, перейдите к редактору кода. В разделе **implementation** модуля MainUnit вы обнаружите строку

```
uses AboutUnit;
```

Ее можно было бы набрать и вручную, но мы решили продемонстрировать вам еще одну возможность визуальной среды программирования.

Шаг 15. Выполните теперь второй пункт плана — создайте обработчик события **OnExecute** для команды AboutAction компонента ActionList:

```
procedure TPictureForm.AboutActionExecute(Sender: TObject);
begin
  AboutForm.ShowModal;
end;
```

Метод **ShowModal** запускает окно диалога в монопольном режиме и возвращает управление только после его завершения. В нашем примере метод вызывается как процедура, но в действительности это функция, которая возвращает код завершения диалога. Код берется у формы из свойства **ModalResult**. Напомним, что для выполнения диалога в немонопольном режиме его нужно вызвать с помощью метода **Show**, а не метода **ShowModal**.

Шаг 16. Выполните компиляцию и запустите программу, затем проверьте работу новоиспеченного окна диалога, выполнив команду меню **Help | About...** (рисунок 9.17).

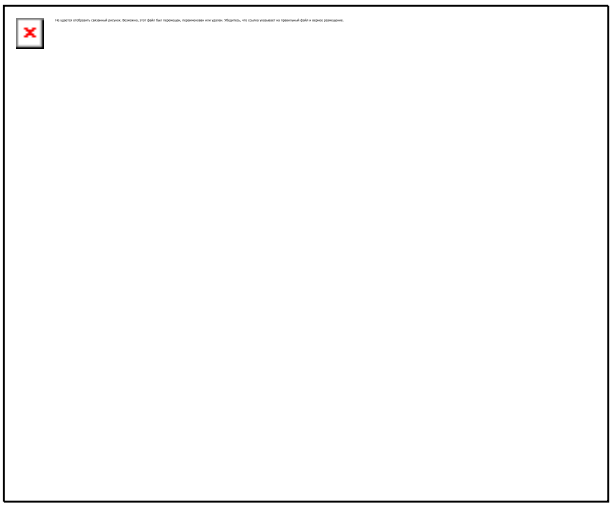


Рисунок 9.17. Диалоговое окно в работе

Все работает правильно, остается только выяснить один вопрос: где, когда и кем конструируется объект **AboutForm**. Ведь никаких усилий мы для этого не предпринимали, и тем не менее использовали объект при вызове метода **ShowModal** как уже существующий.

Объект **AboutForm** создается при запуске программы и существует на протяжении всей ее работы. В этом можно убедиться, заглянув в главный файл программы с помощью команды меню **View | Project Source**. В главном программном блоке вы найдете оператор:

```
Application.CreateForm(TAboutForm, AboutForm);
```

Он-то и обеспечивает "автоматическое" создание объекта формы. Это удобно, но имеет и отрицательные стороны, так как память, выделенная объекту остается занятой даже тогда, когда форма невидима, т.е. до запуска диалога и после его завершения. Если программа имеет одну-две формы, это не так важно, а если много? В этом случае от автоматического создания всех форм, кроме главной, лучше отказаться.

Шаг 17. Чтобы исключить автоматическое создание формы **AboutForm**, откройте окно **Project Options** и на вкладке **Forms** отбуксируйте элемент **AboutForm** из списка **Auto-create forms** в список **Available forms** (рисунок 9.18):

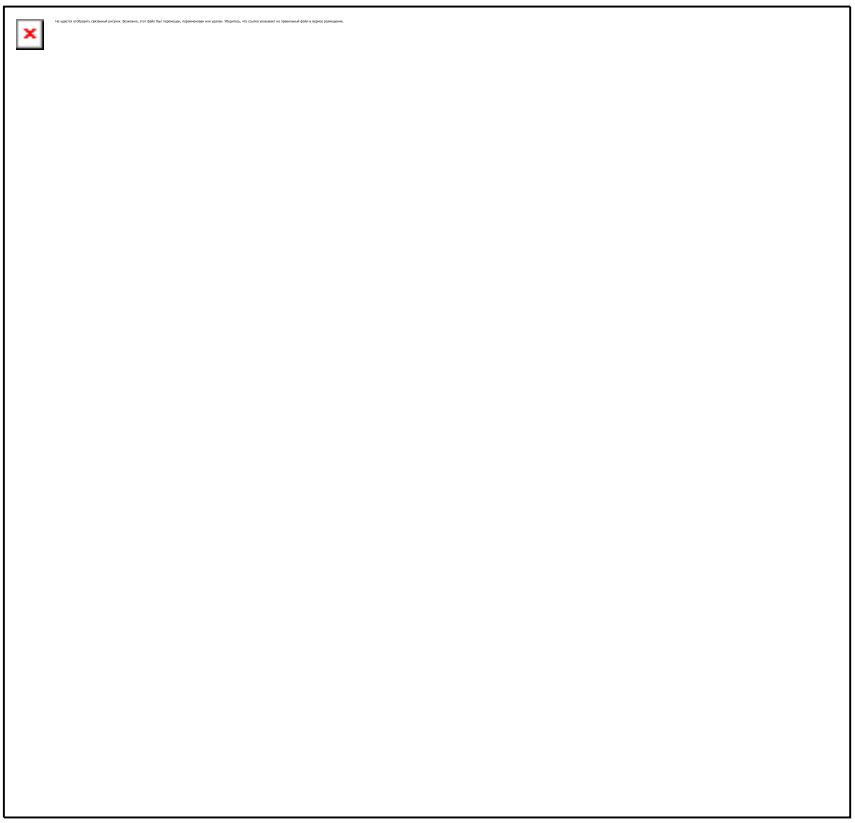


Рисунок 9.18. Форма AboutForm исключена из списка автоматически создаваемых форм

В результате среда Delphi удалит приведенный выше оператор из главного файла программы, переложив заботу о создании формы на ваши плечи.

Шаг 18. Разумеется, что после сделанных действий метод **AboutItemClick**, из которого вызывается окно диалога, нужно переписать:

```
procedure TPictureForm.AboutActionExecute(Sender: TObject);
begin
  AboutForm := TAboutForm.Create(Self);
  try
    AboutForm.ShowModal;
  finally
    AboutForm.Free;
  end;
end;
```

Ну вот, теперь ресурс оперативной памяти используется более рационально. Кстати, обратите внимание, как обеспечивается защита объекта **AboutForm** от исключительных ситуаций, которые могут возникнуть в период его работы (это конечно маловероятно, но чего в этой жизни не бывает!). Если объект **AboutForm** успешно создается, то благодаря оператору **try...finally...end** он всегда корректно освобождается, даже в случае возникновения исключительной ситуации.

9.3. Компоненты для ввода данных

Вы получили первое представление об окнах диалога, научились создавать и выполнять простейшие из них. Но вы еще не знаете, как организовать диалог для получения данных от пользователя. Эта задача решается с помощью компонентов для ввода данных, к изучению которых мы сейчас приступаем.

Использование компонентов для ввода данных рассмотрим на примере приложения Alarms. Эта полезная программа позволит создать список будильников для уведомления о предстоящих событиях. По сигналу будильника в заданные время и день появится окно с сообщением и прозвучит сигнал. Список будильников будет отображаться в главном окне

программы, а установка их параметров будет выполняться в модальном окне диалога. В процессе разработки этого приложения вы познакомитесь с такими компонентами, как **CheckBox**, **RadioButton**, **ComboBox**, **ListBox**, **GroupBox**, **Edit**, **MaskEdit** и некоторыми другими. Итак, приступим.

Шаг 1. Сначала приготовьте новый проект с пустой формой, выбрав команду меню **File | New Application**. Дайте форме идентификатор **MainForm**, скорректируйте ее размеры и установите следующие значения свойств:

Caption = Clock Alarms

BorderIcons = [biSystemMenu,biMinimize]

BorderStyle = bsSingle

Position = poDefaultPosOnly

Сохраните модуль формы под именем MainUnit.pas, а проект — под именем Alarms.dpr.

Шаг 2. В форме **MainForm** будет отображаться список будильников. Для управления списком нужны кнопки: New, Edit и Delete. Для быстрого и удобного закрытия формы нужна еще кнопка Close. Поэтому поместите в форму соответствующее число компонентов **Button** (с идентификаторами **NewButton**, **EditButton**, **DeleteButton**, **CloseButton**) и задайте для них надписи, размеры и положение как на рисунке 9.19.

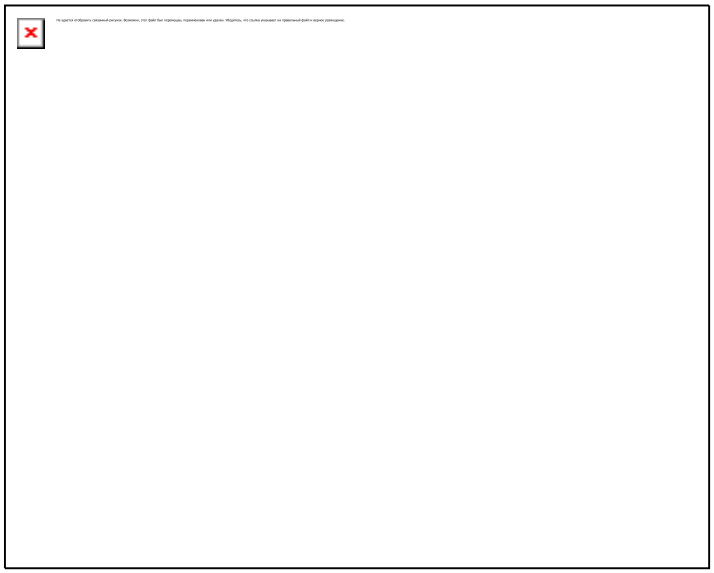


Рисунок 9.19. Кнопки для управления списком будильников

Шаг 3. Установите в компоненте **NewButton** свойство **Default** в значение True, чтобы кнопка срабатывала при нажатии клавиши Enter.

Шаг 4. Для компонента **CloseButton** создайте следующий обработчик события **OnClick**:

```
procedure TMainForm.CloseButtonClick(Sender: TObject);
begin
  Close;
end;
```

Шаг 5. Необходимые подготовительные операции сделаны. Теперь перейдем к разработке окна диалога, предназначенного для ввода параметров будильника. Это окно будет вызываться при нажатии кнопок New и Edit. С этой целью добавьте в проект новую форму, дайте ей идентификатор **AlarmDetailsForm**, скорректируйте размеры и установите следующие значения свойств:

Caption = Alarm Details

BorderStyle = bsDialog

Position = poScreenCenter

Теперь сохраните модуль формы под именем AlarmDetailsUnit.pas.

Шаг 6. Добавьте в форму кнопки OK и Cancel и установите их свойства так, как показано на рисунке 9.20.

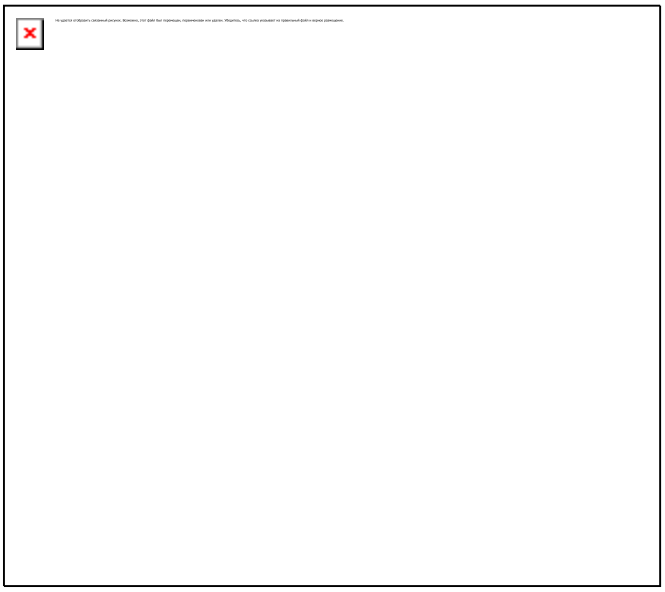


Рисунок 9.20. Стандартные кнопки OK и Cancel и их свойства

Шаг 7. Теперь займемся размещением компонентов для ввода данных. Прежде всего подумаем, какие параметры должны устанавливаться в диалоге. К ним относятся: текстовое сообщение, которое появится по сигналу будильника, время сигнала с точностью до минуты, признак того, нужно ли проигрывать звуковой сигнал, периодичность выдачи сигналов (ежедневно, в заданный день недели или в конкретный день). Учли все? Вроде бы, да. Тогда разместите в форме компоненты, обеспечивающие ввод перечисленных параметров (рисунок 9.21).

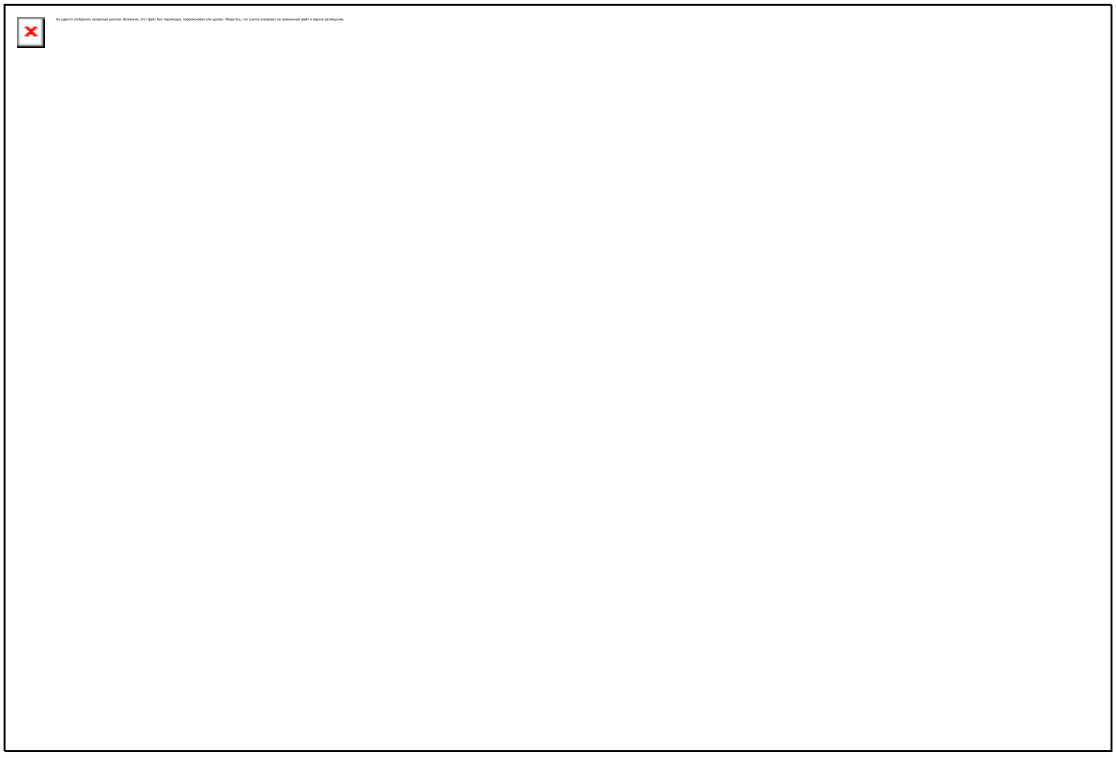


Рисунок 9.21. Эскиз окна для установки параметров будильника

Эскиз окна диалога создан (с помощью выносок на рисунке пояснены названия компонентов). Окинув его взором, вы обнаружите, что знакомых вам компонентов немного — это **Bevel**, **Label** и **Button**. Зато новых — хоть отбавляй: **CheckBox**, **RadioButton**, **Edit**, **MaskEdit**, **GroupBox**, **ComboBox**, **DateTimePicker**. Однако не пугайтесь, мы обо всех расскажем, и вы убедитесь, что обращаться с ними вовсе не сложно.

9.3.1. Фокус ввода

Во время работы программы только один из компонентов принимает клавиатурный ввод в текущий момент времени. Принято говорить, что такой компонент обладает *фокусом ввода* или просто — активен. Передача фокуса ввода осуществляется щелчками компонентов или нажатием клавиш Tab и Shift+Tab на клавиатуре. При использовании клавиатуры фокус ввода передается последовательно от одного компонента другому, причем клавиша Tab обеспечивает перебор элементов в прямом порядке, а сочетание клавиш Shift+Tab – в обратном.

Очередность, в которой компонент получает фокус ввода, задается его свойством TabOrder. Свойство TabOrder действует относительно содержащего компонента, например очередность перебора компонентов **MessageEdit** и **TimeMaskEdit** задается относительно формы, а очередность перебора компонентов **WeeklyComboBox** и **DatePicker** — относительно компонента **GroupBox**.

Если нужно исключить компонент из очереди на фокус ввода, установите свойство **TabStop** в значение False. Однако в этом случае фокус ввода можно насильно передать компоненту с помощью мыши или «горячей» клавиши.

Изначально порядок перебора соответствует порядку добавления компонентов в форму, но его можно изменить, устанавливая значения свойства **TabOrder** в компонентах. Если компонентов в окне диалога очень много, то это занятие может стать довольно утомительным. В этом случае удобнее пользоваться окном **Edit Tab Order**, которое открывается по команде меню **Edit | Tab Order...** (рисунок 9.22).

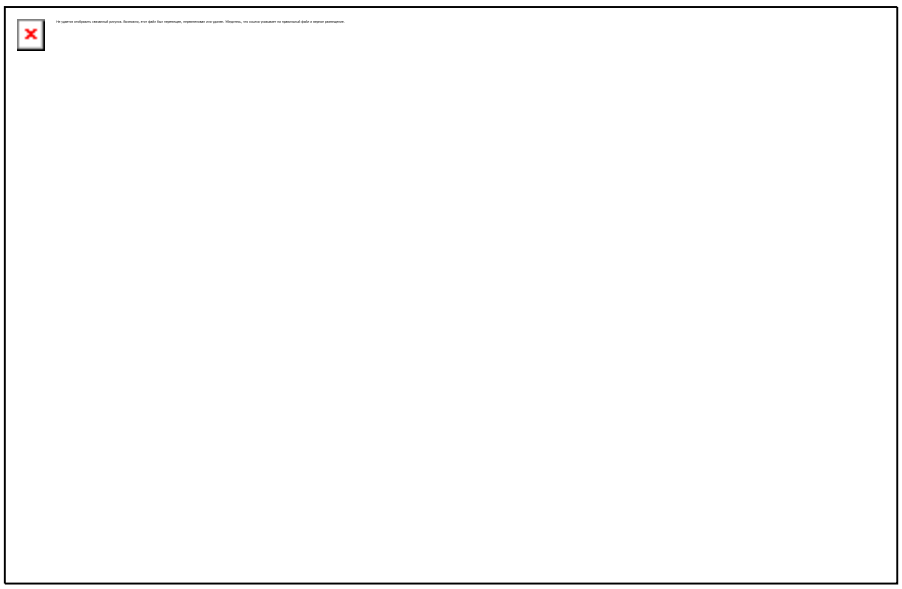


Рисунок 9.22. В окне Edit Tab Order выставляется порядок перебора компонентов формы

Шаг 8. В этом окне задайте порядок перебора компонентов формы **AlarmDetailsForm** таким, как показан на рисунке.

Ну хорошо, скажете вы, допустим, порядок перебора задан. А как управлять передачей фокуса ввода программно? Очень просто. Чтобы передать фокус ввода требуемому

компоненту, у него нужно вызвать метод **SetFocus**. Кстати при управлении фокусом ввода очень полезным может оказаться свойство формы **ActiveControl**, которое указывает активный компонент. Это свойство доступно в окне свойств и часто используется для указания компонента, который первым получит фокус ввода (в обход номера очереди). Если значение свойства не задано, то первым фокус ввода получает компонент, в котором значение свойства **TabOrder** равно нулю.

Шаг 9. Хотя окно диалога **Alarm Details** еще не готово, вам, наверное, не терпится его опробовать и убедиться, что перебор компонентов происходит в нужном порядке. Для этого нужно связать выполнение диалога с нажатием в главной форме кнопки **New...**. Поэтому подключите модуль **AlarmDetails** в модуле **Main** и определите следующий обработчик события **OnClick** для кнопки **NewButton**:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
begin
  AlarmDetailsForm := TAlarmDetailsForm.Create(Self);
  try
    AlarmDetailsForm.ShowModal;
  finally
    AlarmDetailsForm.Free;
  end;
end;
```

Шаг 10. Теперь выполните компиляцию и запустите программу. В окне **Clock Alarms** нажмите кнопку **New...** . Вашему взору предстанет окно диалога **Alarm Settings** (рисунок 9.23).

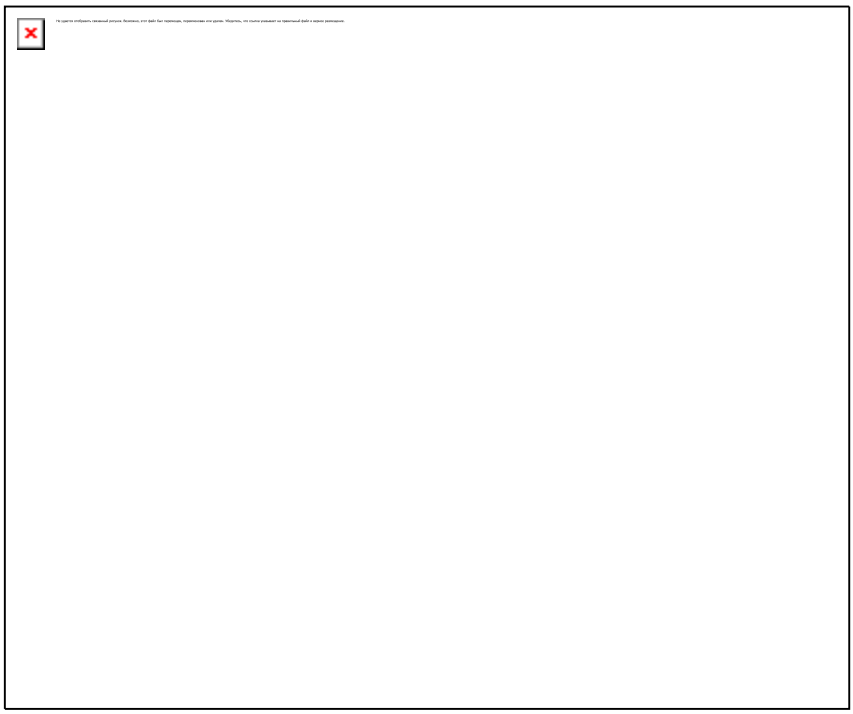


Рисунок 9.23. Рабочий прототип окна для установки параметров будильника

Диалог запускается, фокус ввода перемещается в нужном порядке, но в целом компоненты работают не так, как надо. Ничего удивительного, ведь мы ими почти не занимались. Поэтому дальше мы приступаем к детальному изучению различных типов компонентов. На этом пути полигоном для испытаний нам послужит окно диалога **Alarm Details**.

9.3.2. Переключатели

Переключатели (check boxes) используются для установки параметров, характеризуемых двумя значениями – "Да" и "Нет" (True и False). Переключатели создаются с помощью

компонента **CheckBox**, расположенного в палитре компонентов на вкладке Standard (рисунок 9.24).

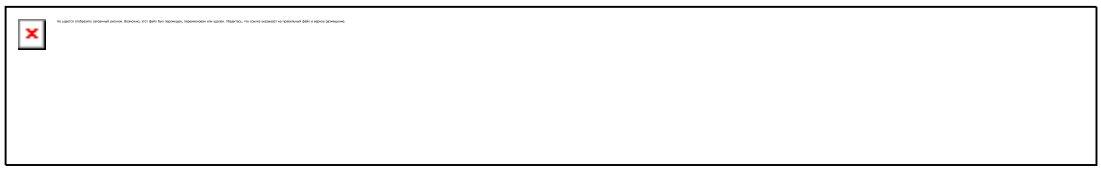


Рисунок 9.24. Компонент CheckBox

Характерные свойства этого компонента собраны в таблице 9.6.

Свойство	Описание
Alignment	Определяет, с какой стороны от переключателя находится текст: taRightJustify – справа, taLeftJustify – слева.
AllowGrayed	Если равно True, то переключатель имеет три состояния.
Caption	Текст рядом с переключателем.
Checked	Определяет, включен ли переключатель.
State	Содержит текущее состояние переключателя.
WordWrap	Если равно значению True, то работает перенос слов.

Таблица 9.6. Важнейшие свойства компонента CheckBox

Обычно переключатель имеет два состояния: *включен* или *выключен*. Текущее состояние определяется значением свойства **Checked**. Если оно равно значению True, то переключатель включен, иначе — выключен. Бывает, что переключатель имеет еще и третье состояние — *неопределенное* (grayed). В этом состоянии переключатель закрашивается серым цветом. Если переключатель имеет три состояния, то вместо свойства **Checked** используется свойство **State**, а в свойстве **AllowGrayed** (разрешает неопределенное состояние) устанавливается значение True.

Когда при работе программы пользователь щелкает переключатель, в нем изменяются значения свойств **Checked** и **State**, а также происходит событие **OnClick**. Обрабатывая это событие, можно установить любую зависимость между состоянием переключателя и состоянием других компонентов.

В нашем примере компонент **CheckBox** используется для установки параметра **Play Sound**, управляющего выдачей звукового сигнала (рисунок 9.25). Мы назвали его **SoundCheckBox**.

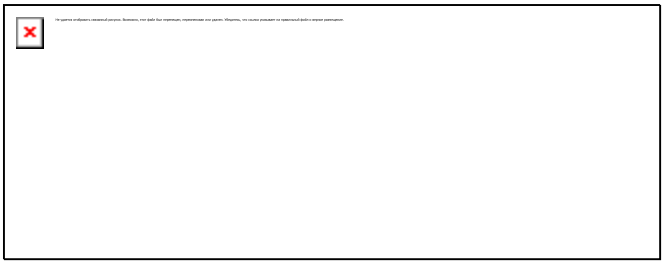


Рисунок 9.25. Переключатель Play Sound представлен компонентом CheckBox

Шаг 11. Чтобы при первом появлении окна диалога, режим **Play Sound** был включен, установите в компоненте **SoundCheck** свойство **Checked** в значение True.

9.3.3. Взаимоисключающие переключатели

Взаимоисключающие переключатели (radio buttons) служат для установки взаимоисключающих параметров. Они обычно объединяются в группы и позволяют пользователю выбрать одно значение из фиксированного (причем немногочисленного) набора вариантов. При включении одного такого переключателя остальные переключатели этой же группы выключаются.

Группа переключателей создается с помощью нескольких компонентов **RadioButton**. Компонент **RadioButton** находится в палитре компонентов на вкладке Standard (рисунок 9.26).

Рисунок 9.26. Компонент RadioButton

Характерные свойства компонента **RadioButton** описаны в таблице 9.7.

Свойство	Описание
Alignment	Определяет, с какой стороны от переключателя находится текст: taRightJustify — справа, taLeftJustify — слева.
Caption	Текст рядом с переключателем.
Checked	Определяет, включен ли переключатель.
WordWrap	Если равно значению True, то работает перенос слов.

Таблица 9.7. Важнейшие свойства компонента RadioButton

В форму всегда помещаются несколько компонентов **RadioButton**, соответствующих возможным значениям устанавливаемого параметра. Например, в форме **AlarmDetailsForm** содержится три таких компонента (**EverydayRadioButton**, **WeeklyRadioButton** и **DateRadioButton**), которые используются для выбора периодичности срабатываний будильника (рисунок 9.27).

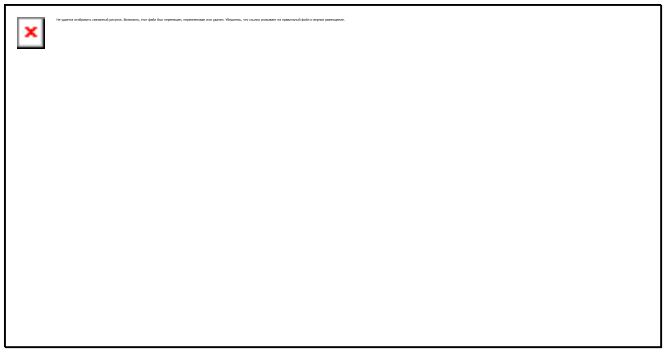


Рисунок 9.27. Переключатели Everyday, Weekly и Date представлены компонентами RadioButton

Внимание! Выбор положения переключателя осуществляется с помощью клавиш со стрелками.

Шаг 12. Текущее состояние переключателя контролируется свойством **Checked**. Если в одном переключателе оно устанавливается в значение True, то во всех остальных переключателях этой же группы оно устанавливается в значение False. Установите в компоненте **EverydayRadioBox** свойство **Checked** в значение True — этот переключатель будет изначально включен.

Шаг 13. Когда пользователь щелкает переключатель, в соответствующем компоненте **RadioButton** свойство **Checked** получает значение True и происходит событие **OnClick**. Обрабатывая это событие, можно установить любую зависимость между состоянием переключателя и состоянием других компонентов формы. Например, когда включен режим **Everyday**, компоненты для ввода дня недели и даты должны быть недоступны. Когда включен режим **Weekly on**, должен быть доступен выбор дня недели, но недоступна установка даты. Наконец, когда включен режим **Date**, установка дня недели должна быть недоступна, а установка даты — доступна. Чтобы достигнуть такой согласованности в работе, определите для переключателей **EverydayRadioButton**, **WeeklyRadioButton** и **DateRadioButton** единый обработчик события **OnClick**:

```
procedure TAlarmDetailsForm.RecurringRadioButtonClick(Sender: TObject);
begin
  WeeklyComboBox.Enabled := Sender = WeeklyRadioButton;
  DatePicker.Enabled := Sender = DateRadioButton;
end;
```

Шаг 14. Разумеется, компоненты **WeeklyComboBox** и **DatePicker** первоначально должны быть недоступны. Поэтому, удерживая клавишу Shift, выберите названную группу компонентов и установите свойство **Enabled** в значение False.

Теперь запустите программу и проверьте правильность ее работы.

9.3.4. Группа взаимоисключающих переключателей

Для быстрой организации группы взаимоисключающих переключателей очень удобен компонент **RadioGroup**, расположенный в палитре компонентов на вкладке Standard (рисунок 9.28).

Рисунок 9.28. Компонент RadioGroup

Его характерные свойства кратко описаны в таблице 9.8.

Свойство	Описание
Align	Способ выравнивания в пределах содержащего компонента.
Caption	Подпись к группе переключателей.
Columns	Число колонок в группе переключателей.
ItemIndex	Номер выбранного элемента, начиная с нуля. Если все переключатели находятся в выключенном состоянии, то значение свойства равно -1.
Items	Подписи к переключателям.

Таблица 9.8. Важнейшие свойства компонента RadioGroup

Компонент **RadioGroup** удобен тем, что заменяет группу компонентов **RadioButton**. Расположение переключателей, которые он отображает, подбирается автоматически с учетом заданного в свойстве **Columns** количества колонок. Номер активного зависимого переключателя хранится в значении свойства **ItemIndex**. Следующий рисунок 9.29 не относится к приложению Alarms, а просто поясняет, что такое компонент **RadioGroup**:

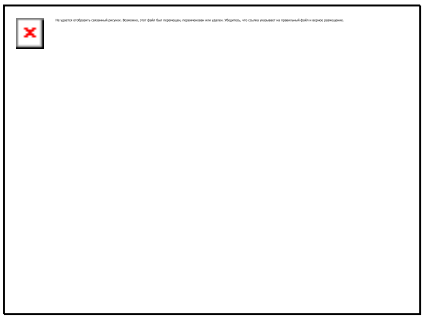


Рисунок 9.29. Четырехпозиционный переключатель представлен компонентом RadioGroup

Компонент **RadioGroup** создает группу, состоящую исключительно из переключателей. Однако он не годится в тех случаях, когда в группе должны присутствовать другие компоненты, что как раз имеет место в нашем примере. В этой ситуации для группировки компонентов применяется компонент **GroupBox**.

9.3.5. Панель группы компонентов

Компонент **GroupBox** служит для группировки компонентов, он расположен в палитре компонентов на вкладке **Standard** (рисунок 9.30).

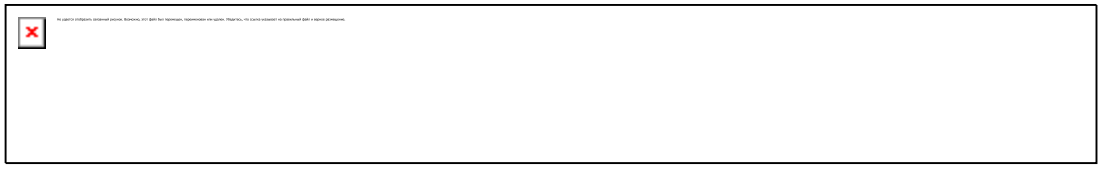


Рисунок 9.30. Компонент GroupBox

Компонент **GroupBox** выглядит как панель с заголовком рисунок 9.31. Текст заголовка задается в свойстве **Caption**.

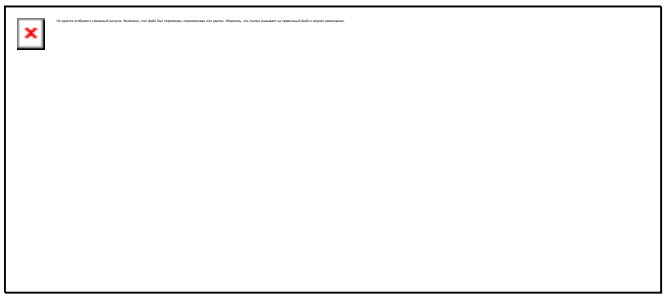


Рисунок 9.31. Панель с заголовком представлена компонентом GroupBox

Компонент **GroupBox** может содержать в себе другие компоненты. Это, например, означает, что установка его свойства **Visible** в значение False прячет группу со всеми расположенными внутри компонентами, а не только рамку с заголовком.

9.3.6. Поле ввода и редактор текста

Для ввода текста предназначены компоненты **Edit** и **Memo**. Они представляют собой соответственно поле ввода и редактор многострочного текста (кроме них существует еще

компонент **RichEdit**, предназначенный для ввода и отображения форматированного текста, но о нем мы поговорим отдельно).

Поле ввода (**Edit**) служит для ввода различных слов, фраз и другого относительно короткого текста. Он не имеет полос прокрутки, но разрешает прокручивать текст по горизонтали клавишами перемещения курсора влево и вправо. Компонент **Edit** расположен в палитре компонентов на вкладке **Standard** (рисунок 9.32).

Рисунок 9.32. Компонент Edit

Характерные свойства компонента **Edit** описаны в таблице 9.9.

Свойство	Описание
AutoSelect	Если равно значению True, то при активизации редактора находящийся в нем текст автоматически выделяется.
AutoSize	Если равно значению True, то высота редактора автоматически подгоняется по высоте текста.
BevelEdges	Вложенные свойства **beLeft**, **beTop**, **beRight** и **beBottom** определяют видимость соответственно левой, верхней, правой и нижней сторон рельефной рамки.
BevelInner	Внутренний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelKind	Вид рельефной рамки: **bkNone** — рамки нет, **bkTile** — рамка с четкими скосами, **bkSoft** — рамка со сглаженными скосами, **bkFlat** — плоская рамка (без скосов).
BevelOuter	Внешний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
CharCase	Преобразует текст к прописным или строчным буквам: ecUpperCase – к прописным буквам, ecLowerCase – к строчным буквам, ecNormal –преобразование символов не выполняется.
HideSelection	Если равно значению True, то при потере редактором активности выделение текста снимается.
MaxLength	Максимальное количество символов, которое пользователь может ввести. Если оно равно 0, то пользователь может ввести текст неограниченной длины.
OEMConvert	Если равно значению True, то символы текста преобразуются в кодовую таблицу OEM.

PasswordChar	Если не равно #0, то указанный в этом свойстве символ отображается вместо каждого символа текста. Применяется для ввода пароля.
ReadOnly	Если равно значению True, то пользователь не сможет изменить текст в редакторе.
Text	Редактируемый текст.
OnChange	Происходит при изменении текста.

Таблица 9.9. Отличительные свойства и события компонента Edit

Редактируемый текст содержится в свойстве **Text**. Его максимальная длина определяется значением свойства **MaxLength**.

Иногда компонент **Edit** используется для отображения нередактируемого текста. Для этого свойство **ReadOnly** устанавливается в значение True. Вы спросите, чем он в таком состоянии лучше компонента **Label**? А тем, что может получать фокус ввода. Кроме того, текст в поле ввода можно выделить и поместить в Буфер Обмена.

Компонент **Edit** легко приспособить для ввода паролей. Для этого достаточно установить в свойстве **PasswordChar** вместо символа #0 какой-нибудь другой символ, обычно символ звездочки (*****). Символ, заданный в свойстве **PasswordChar** отображается вместо реально вводимых символов, что мешает подсмотреть пароль посторонним.

Шаг 15. В форме **SettingsForm** компонент **Edit** применяется для ввода текстового сообщения будильника. Выберите его в окне свойств и в значении свойства **Text** впишите "Reminder !" (рисунок 9.33). Этот текст будет появляться в редакторе при появлении окна диалога.

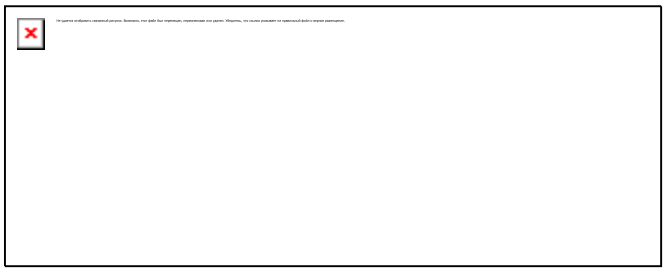


Рисунок 9.33. Компонент Edit используется для ввода текстового сообщения будильника

Шаг 16. Изменение текста во время работы программы приводит к возникновению в компоненте **Edit** события **OnChange**. Обрабатывая это событие, можно, например, устроить работу окна диалога таким образом, что кнопка OK будет недоступна, если в редакторе нет текста. Чтобы реализовать такое поведение нашего диалога, определите для компонента **MessageEdit** следующий обработчик события **OnChange**:

```
procedure TAlarmDetailsForm.MessageEditChange(Sender: TObject);
begin
  OkButton.Enabled := Length(MessageEdit.Text) <> 0;
end;
```

Выполните компиляцию программы и проверьте ее работу.

Прежде чем продолжить обсуждение примера Alarms, сделаем несколько замечаний по поводу редактирования многострочного текста. *Редактор многострочного текста* представлен компонентом **Memo** (рисунок 9.34).

Рисунок 9.34. Компонент Memo

Характерные свойства компонента **Memo** описаны в таблице 9.10.

Свойство	Описание
Align	Способ выравнивания компонента в пределах содержащего компонента.
Alignment	Выравнивание текста: taLeftJustify – прижат к левой границе, taRightJustify – прижат к правой границе, taCenter – центрирован.
HideSelection	Если равно значению True, то при потере редактором фокуса ввода выделение текста снимается.
Lines	Текст в виде массива строк.
MaxLength	Максимальное количество символов, которое пользователь может ввести. Если оно равно 0, то пользователь может ввести текст неограниченной длины.
OEMConvert	Если равно значению True, то символы текста преобразуются в кодовую таблицу OEM.
ReadOnly	Если равно значению True, то пользователь не сможет изменить текст в редакторе.
ScrollBars	Управляет видимостью полос прокрутки: ssNone – полосы прокрутки скрыты, ssBoth – полосы прокрутки видны, ssHorizontal – видна лишь горизонтальная полоса прокрутки, ssVertical – видна лишь вертикальная полоса прокрутки.
WantReturns	Если равно значению True, то клавиша Enter начинает в редакторе новую строку. Иначе нажатие клавиши Enter ассоциируется с нажатием стандартной кнопки окна диалога и для перевода строк применяется сочетание клавиш Ctrl+Enter.
WantTabs	Если равно значению True, то клавиша Tab вставляет в текст символ табуляции, вместо того чтобы передать фокус ввода следующему компоненту.
WordWrap	Если равно значению True, то работает перенос слов.
OnChange	Происходит при изменении текста.

Таблица 9.10. Важнейшие свойства и события компонента Memo

Компонент **Memo** похож на **Edit**, но в отличие от него хранит не одну строку текста, а множество строк. Доступ к строкам обеспечивает свойство **Lines**, представляющее собой объект класса **TStrings** (см. главу 3). С помощью свойства **Lines** строки можно добавлять, вставлять, удалять и т.д. Свойство **Lines** доступно в окне свойств, поэтому на стадии проектирования вы можете заполнить компонент **Memo** некоторым исходным текстом (рисунок 9.35). Этот текст увидит пользователь при появлении формы на экране. Ввод исходного текста осуществляется в специальном редакторе текста (**String list editor**), которое вызывается нажатием кнопки с многоточием в поле значения свойства **Lines**.

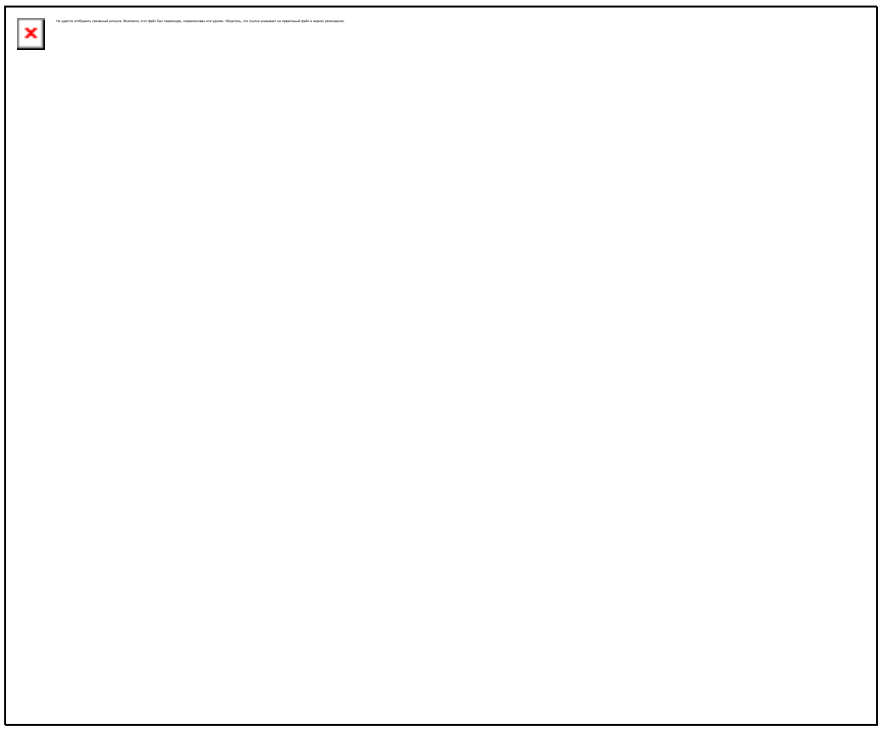


Рисунок 9.35. Окно для ввода многострочного текста, отображаемого компонентом Memo

Компонент **Memo** часто имеет одну или две полосы прокрутки (вертикальную и горизонтальную). Их появление зависит от значения свойства **ScrollBars**.

В нашем приложении Alarms компонент **Memo** не нужен, но вам он безусловно пригодится в других программах.

9.3.7. Редактор с шаблоном

Поскольку компонент **Edit** не проверяет, что вводит пользователь, он не удобен для ввода данных строго определенного формата, например телефонных номеров, времени и др. На этот случай разработчики среды Delphi предусмотрительно создали компонент **MaskEdit**. Он находится в палитре компонентов на вкладке **Additional** (рисунок 9.36).

Рисунок 9.36. Компонент MaskEdit

Компонент **MaskEdit** представляет собой поле ввода, которое вынуждает пользователя вводить данные в строго заданном формате. Во многом аналогичный компоненту **Edit**, он отличается от последнего тем, что имеет свойство **EditMask** и не имеет свойств **HideSelection** и **OEMConvert**.

Свойство **EditMask** задает *шаблон (маску)* для ввода символов текста. Шаблон имеет вид текстовой строки, его символы называются форматными и управляют тем, что вводит пользователь: буквы или цифры, в каком порядке, сколько и т.д. Мы не будем утомлять вас подробным описанием форматных символов, при необходимости обратитесь к справочному руководству. Нас интересует только шаблон для ввода времени с точностью до минуты.

Шаг 17. Выберите компонент **TimeMaskEdit**, затем в окне свойств перейдите к свойству **EditMask** и щелчком кнопки с многоточием откройте специальный редактор для этого свойства — **Input Mask Editor** (рисунок 9.37):

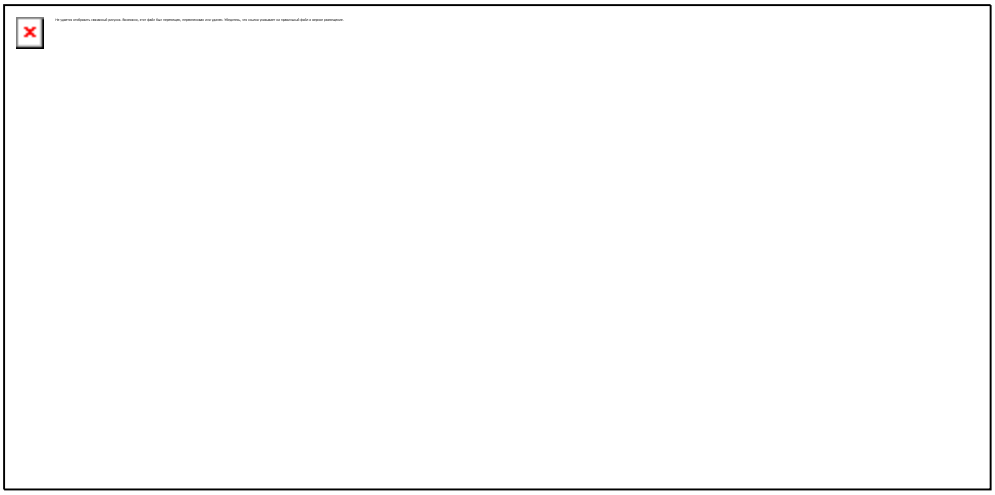


Рисунок 9.37. Выбор шаблона для компонента MaskEdit

В этом окне вы можете ввести шаблон и проверить его работу. Часто используемые шаблоны, например шаблоны телефонных номеров, даты, времени и некоторые другие, можно просто выбрать среди уже имеющихся образцов. Этой возможностью мы и воспользуемся. Выберите в списке **Sample Masks** пункт **Short Time** и щелкните на кнопке OK. Шаблон для ввода времени задан и имеет вид !90:00;1;_ . Обратите внимание, как при этом изменился компонент **TimeMaskEdit** (рисунок 9.38):

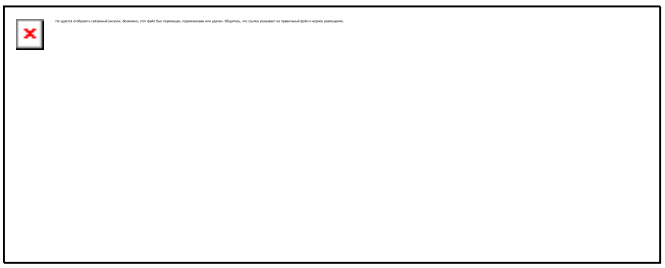


Рисунок 9.38. Компонент MaskEdit настроен для ввода времени

Шаг 18. Теперь давайте сделаем так, чтобы при появлении окна диалога поле ввода **TimeMaskEdit** не было пустым, а содержало текущее время. Для этого создайте у формы обработчик события **OnCreate**:

```
procedure TAlarmDetailsForm.FormCreate(Sender: TObject);
begin
  TimeMaskEdit.Text := FormatDateTime('hh:mm', Time);
end;
```

Готово. Запустите программу и убедитесь, что она работает, как вы того ожидаете.

9.3.8. Раскрывающийся список

Раскрывающийся список (combo box) позволяет пользователю выбрать значение из большого множества альтернатив. Он представляет собой поле ввода, к которому прикреплен раскрывающийся список значений. Редактор служит для ввода нового значения, а список —

для выбора существующего значения. Количество элементов в списке может быть произвольным, причем элементы можно динамически добавлять, удалять, заменять и т.д. Элементами списка обычно служат текстовые строки, но могут быть и графические рисунки (в последнем случае их редактирование невозможно).

Раскрывающийся список представлен компонентом **ComboBox**, который находится в палитре компонентов на вкладке **Standard** (рисунок 9.39):

Рисунок 9.39. Компонент ComboBox

Характерные свойства компонента **ComboBox** собраны в таблице 9.11.

Свойство	Описание
AutoCloseUp	Если равно True, то при вводе пользователем текста, который уже существует в списке **Items**, раскрытый список значений автоматически закрывается.
AutoComplete	Если равно True, то компонент предугадывает вводимый пользователем текст на основе списка **Items**.
AutoDropDown	Если равно True, то при вводе текста автоматически раскрывается список существующих значений.
BevelEdges	Вложенные свойства **beLeft**, **beTop**, **beRight** и **beBottom** определяют видимость соответственно левой, верхней, правой и нижней сторон рельефной рамки.
BevelInner	Внутренний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelKind	Вид рельефной рамки: **bkNone** — рамки нет, **bkTile** — рамка с четкими скосами, **bkSoft** — рамка со сглаженными скосами, **bkFlat** — плоская рамка (без скосов).
BevelOuter	Внешний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
CharCase	Автоматическое преобразование регистра букв: **ecLowerCase** — к строчным буквам, **ecUpperCase** — к заглавным буквам, **ecNormal** — без преобразования.
DropDownCount	Количество одновременно видимых элементов раскрывающегося списка.
ItemHeight	Высота отдельного элемента списка, когда значение свойства **Style** равно csOwnerDrawFixed.

ItemIndex	Порядковый номер выбранного в списке элемента, начиная с нуля.
Items	Элементы раскрывающегося списка.
MaxLength	Максимальное количество символов, которое пользователь может ввести в строке редактора. Если оно равно 0, то пользователь может ввести текст неограниченной длины.
Sorted	Если равно True, то элементы списка сортируются в алфавитном порядке.
Style	Стиль отображения выпадающего списка (см. табл. 7.12).
Text	Текст в строке редактора.
OnChange	Происходит при вводе текста или выборе значения из списка. Не происходит при программном изменении свойства Text.
OnCloseUp	Происходит при закрытии списка значений.
OnDrawItem	Происходит при рисовании элемента раскрывающего списка, но только в том случае, если свойство **Style** содержит значение **csOwnerDrawFixed** или **csOwnerDrawVariable**.
OnDropDown	Происходит при раскрытии списка значений.
OnMeasureItem	Происходит перед рисованием элемента раскрывающего списка для расчета его высоты. Требует, чтобы свойство **Style** содержало значение **csOwnerDrawVariable**.
OnSelect	Происходит при выборе значения в раскрывающемся списке.

Таблица 9.11. Важнейшие свойства компонента ComboBox

Раскрывающийся список умеет отображать себя по-разному в зависимости от значения свойства **Style** (см. таблицу 9.12).

Значение	Описание
csSimple	Редактор и постоянно отображаемый список.
csDropDown	Редактор и ассоциированный с ним раскрывающийся список.
csDropDownList	Раскрывающийся список без редактора. Все элементы списка имеют одинаковую высоту, которая рассчитывается автоматически.
csOwnerDrawFixed	Раскрывающийся список без редактора. Все элементы списка имеют одинаковую высоту, заданную в

	свойстве **ItemHeight**.
csOwnerDrawVariable	Раскрывающийся список без редактора. Элементы списка имеют разную высоту.

Таблица 9.12. Значения свойства Style компонента ComboBox

В двух последних случаях в компоненте **ComboBox** происходит событие **OnDrawItem**. Вы можете его перехватить и рисовать каждый элемент выпадающего списка как вам вздумается. Если элементы списка имеют разную высоту (стиль csOwnerDrawVariable), то компонент генерирует событие **OnMeasureItem**, чтобы узнать высоту каждого элемента. Стили csOwnerDrawFixed и csOwnerDrawVariable применяются в тех случаях, когда элементы списка должны быть рисунками.

Шаг 19. Раскрывающиеся списки пригодились нам в диалоге **Alarm Details** для выбора дня недели (компонент **WeeklyComboBox**). Поскольку все дни недели заранее известны, выберите в свойстве **Style** значение **csDropDownList** (рисунок 9.40).

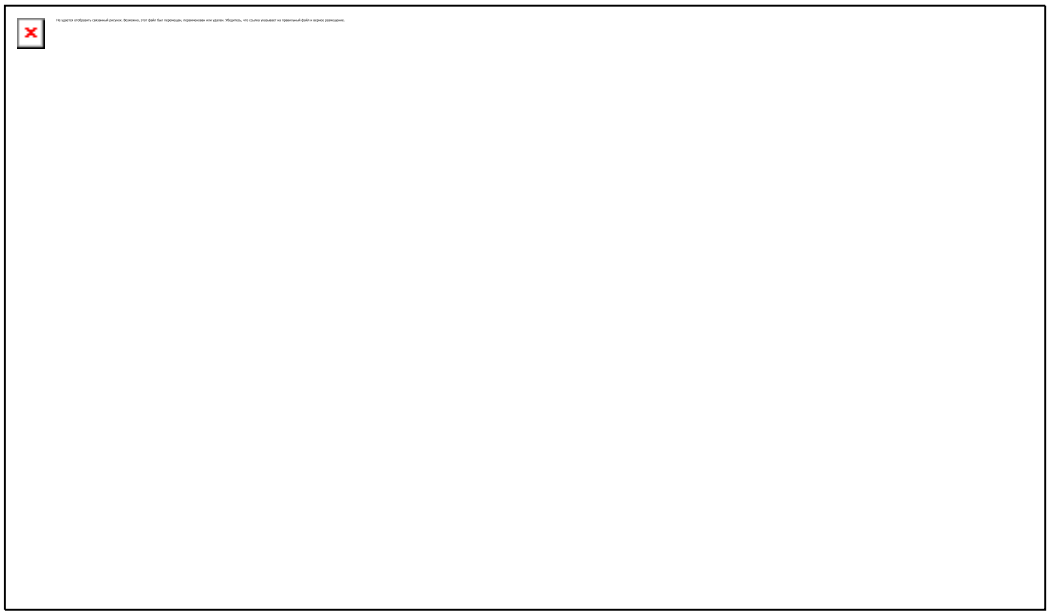


Рисунок 9.40. Для выбора дня недели применяется компонент ComboBox в стиле csDropDownList

Шаг 20. Теперь компонент **WeeklyComboBox** нужно заполнить списком исходных значений. Выберите его в форме, затем в окне свойств перейдите к свойству **Items** и нажмите кнопку с многоточием в поле значения. В появившемся окне введите список строк, как показано на рисунке 9.41.

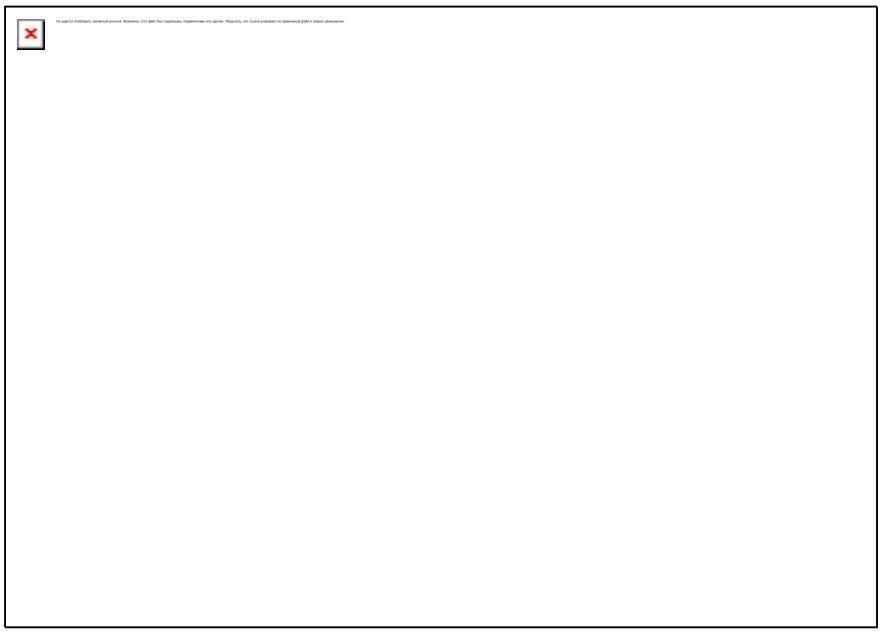


Рисунок 9.41. В этом окне вводятся элементы списка — дни недели

Щелкните кнопку OK — список значений компонента **WeeklyComboBox** задан.

Шаг 21. Выбранный элемент раскрывающегося списка определяется значением свойства **ItemIndex**. Начальное значение свойства равно -1, что означает — ни один элемент не выбран. Однако в компоненте **WeeklyComboBox** должно быть выбрано то значение, которое соответствуют текущей дате. С этой целью нужно доработать у формы обработчик события **OnCreate**. В итоге он будет иметь следующий вид:

```
procedure TAlarmDetailsForm.FormCreate(Sender: TObject);
begin
  // Установка текущего времени
  TimeMaskEdit.Text := FormatDateTime('hh:mm', Time);
  // Установка текущего дня недели
  WeeklyComboBox.ItemIndex := DayOfWeek(DatePicker.Date) - 1;
end;
```

Реализация метода основана на том, что компонент **DatePicker** при создании формы сразу содержит текущую дату.

На этом с визуальной частью диалога **Alarm Details** покончено. Правда, мы ничего не сказали о компоненте **DateTimePicker** (рисунок 9.42).

Рисунок 9.42. Компонент DateTimePicker

Впрочем, он уже работает так, как требуется. В нем нет ничего сложного, и мы надеемся, что вы разберетесь с ним по таблице 9.13.

Свойство	Описание
BevelEdges	Вложенные свойства **beLeft**, **beTop**, **beRight** и **beBottom** определяют видимость соответственно левой, верхней, правой и нижней сторон рельефной рамки.
BevelInner	Внутренний скос рельефной рамки: **bvNone** — скос

	отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelKind	Вид рельефной рамки: **bkNone** — рамки нет, **bkTile** — рамка с четкими скосами, **bkSoft** — рамка со сглаженными скосами, **bkFlat** — плоская рамка (без скосов).
BevelOuter	Внешний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelWidth	Ширина скосов рельефной рамки.
CalAlignment	Способ выравнивания раскрывающегося диалога: **dtaLeft** — по левому краю, **dtaRight** — по правому краю.
CalColors	Цвета: **BackColor** — ни на что не влияет, существует для унификации настройки цвета с другими компонентами; **MonthBackColor** — цвет фона раскрывающего диалога; **TextColor** — цвет текста; **TitleBackColor** — цвет фона заголовка; **TitleTextColor** — цвет текста заголовка; **TrailingTextColor** — цвет текста дат, не принадлежащих текущему месяцу.
Checked	Значение переключателя, который отображается, если свойство **ShowCheckbox** содержит значение True.
Date	Выбранная дата.
DateFormat	Формат даты: **dfShort** — короткий, **dfLong** — длинный.
DateMode	Режим работы компонента: **dmComboBox** — по щелчку кнопки со стрелкой раскрывается окно с месячным календарем, **dmUpDown** — вместо кнопки со стрелкой показывается пара кнопок со стрелками вверх и вниз, щелчки на которых прокручивают день, месяц или год.
Format	Формат даты и времени.
Kind	Если равно значению **dtkDate**, то компонент предназначен для выбора даты, а если значению **dtkTime**, то для выбора времени.
MaxDate	Максимальное значение даты, которое может выбрать пользователь.
MinDate	Минимальное значение даты, которое может выбрать пользователь.
ParseInput	Если равно значению True, то по мере ввода значения пользователем происходит событие **OnUserInput**.
ShowCheckbox	Показывает переключатель (флажок). Значение переключателя определяется свойством **Checked**.

Time	Выбранное время.
OnChange	Происходит при изменении значения даты и времени.
OnCloseUp	Происходит при сворачивании раскрывающегося диалога.
OnUserInput	Происходит по мере ввода данных пользователем.

Таблица 9.13. Важнейшие свойства и события компонента DateTimePicker

В очередной раз выполните компиляцию программы и запустите ее. Откройте окно диалога **Alarm Details** и хорошенько его потестируйте (рисунок 9.43).

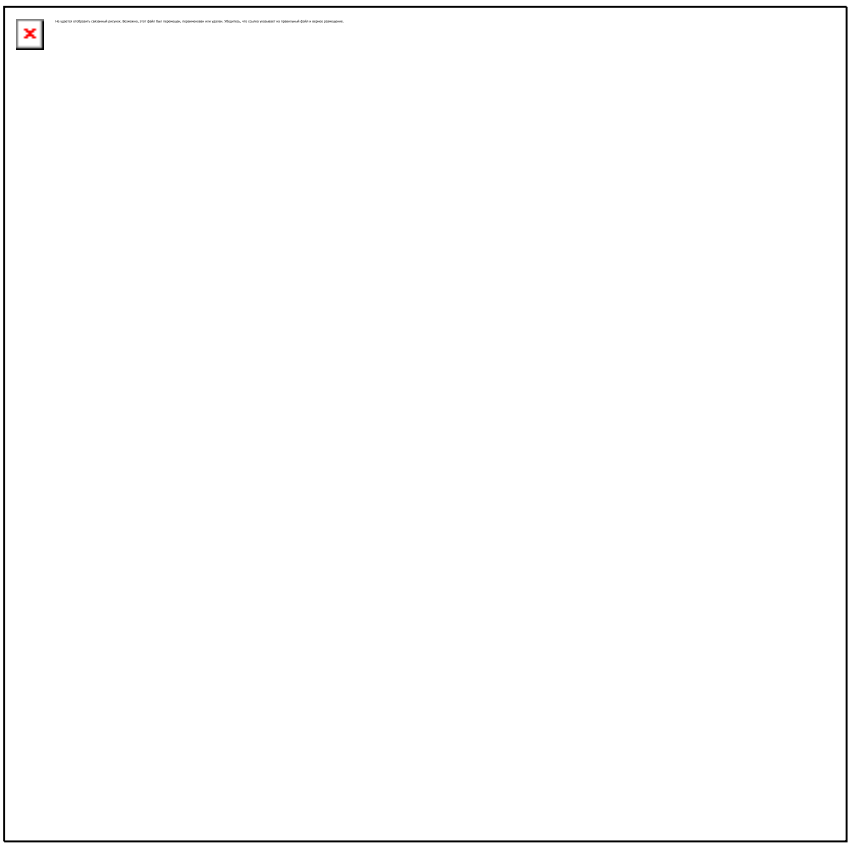


Рисунок 9.43. Тестирование окна Alarm Details

Все компоненты работают правильно. Можете поздравить себя с очередным достижением. Вы создали важную часть приложения Alarms — окно диалога, а заодно разобрались с множеством новых компонентов.

9.3.9. Установка и получение данных

Окно диалога есть, но пользы от него пока нет. Все дело в том, что мы научились устанавливать параметры будильника, но не научились их принимать и хранить.

Шаг 22. Для хранения параметров будильника нам нужна новая структура данных, очевидно класс объектов. Немного поразмыслив, приходим к следующему описанию:

```
type
  TAlarm = class
  private
    Handled: Boolean;
  public
    MsgText: string;
    DateTime: TDateTime;
    PlaySound: Boolean;
    Recurring: Integer;
    function GetAlarmStr: string;
    procedure CheckTime;
  end;
```

Поясним назначение полей и методов. Поле **MsgText** предназначено для хранения текстового сообщения. В поле **DateTime** записывается время и дата сигнала. Значение поля **PlaySound** показывает, требуется ли звуковое сопровождение сообщения. Поле **Recurring** определяет периодичность работы будильника и принимает следующие значения:

- 0 – ежедневно;
- 1..7 – в заданный день недели (1 — Пн, 2 — Вт, ..., 7 — Вс);
- 8 – однажды в заданный день.

В первых двух случаях поле **DateTime** хранит только время, а в последнем еще и дату. Такое ухищрение позволяет организовать компактное хранение данных. Флаг **Handled**, объявленный в секции **private**, является служебным и позволит избежать повторных срабатываний, когда будильник уже прозвенел. Метод **GetAlarmStr** мы определили для удобства. Он будет формировать строку сообщения, содержащую время и текст напоминания. Метод **CheckTime** проверит, пора ли выдать сигнал и если да, то сделает это.

Шаг 23. Поместите описание класса **TAlarm** в раздел **interface** модуля **AlarmDetails**. Затем в разделе **implementation** наберите текст методов **GetAlarmStr** и **CheckTime**:

```
function TAlarm.GetAlarmStr: string;
begin
  Result := FormatDateTime('hh:mm  ', DateTime) + MsgText;
end;

procedure TAlarm.CheckTime;
var
  Hour1, Min1, Sec1, MSec1: Word;
  Hour2, Min2, Sec2, MSec2: Word;
  Match: Boolean;
begin
  // Декодировать текущее время
  DecodeTime(Time, Hour1, Min1, Sec1, MSec1);
  // Раскодировать текущее время будильника
  DecodeTime(DateTime, Hour2, Min2, Sec2, MSec2);
  // Проверить, что текущее время совпадает с временем будильника
  case Recurring of
    0:    // для ежедневной периодичности
      Match := (Hour1 = Hour2) and (Min1 = Min2);
    1..7: // для еженедельной периодичности
      Match := (Hour1 = Hour2) and (Min1 = Min2) and
               (Recurring = DayOfWeek(Date));
    8:    // для конкретной даты
      Match := (Hour1 = Hour2) and (Min1 = Min2) and
               (Int(DateTime) = Date);
    else
      Match := False;
  end;
  // Решить вопрос о выдаче сигнала будильником
  if Match then
  begin
    if not Handled then  // сигнал!
    begin
      Handled := True;   // предотвратить повторные срабатывания
      if PlaySound then Beep;
      MessageDlg(GetAlarmStr, mtWarning, [mbOk], 0);
    end;
  end
  else
    Handled := False; // обеспечить будущие срабатывания
end;
```

Для правильной работы будильника метод **CheckTime** должен вызываться не реже одного раза в минуту. Чем чаще вызывается метод, тем меньше инерционность будильника, но тем больше пустых опросов, а значит выше загруженность операционной системы. Компромиссная частота — два раза в секунду. Так как в одну и ту же минуту метод **CheckTime** будет вызван несколько раз, то для избежания повторных срабатываний используется флаг **Handled**. Будильник выдает сообщение только в том случае, если текущее время совпадает с временем, на которое будильник установлен и при условии, что в данную минуту он еще не звенел.

Шаг 24. Давайте теперь позаботимся о передаче данных в окно диалога перед его запуском и о приеме данных после завершения. Удобнее всего, чтобы за это отвечало само окно диалога, т.е. форма **AlarmDetailsForm**. С этой целью определите в классе **TAlarmDetilasForm** два метода — **GetData** и **SetData**. Методы следует поместить в секцию **public**:

```
type
  TAlarmDetailsForm = class(TForm)
    ...
  public
    procedure GetData(Alarm: TAlarm);
    procedure SetData(Alarm: TAlarm);
  end;
```

В разделе **implementation** наберите программный текст методов:

```
procedure TAlarmDetailsForm.GetData(Alarm: TAlarm);
begin
  with Alarm do
  begin
    // Получить из диалога текст сообщения будильника
    MsgText := MessageEdit.Text;
    // Получить из диалога время срабатывания будильника
    DateTime := StrToTime(TimeMaskEdit.Text);
    // Получить из диалога состояние переключателя звука
    PlaySound := SoundCheckBox.Checked;
    // Получить из диалога периодичность срабатывания будильника
    if EverydayRadioButton.Checked then
      Recurring := 0
    else if WeeklyRadioButton.Checked then
      Recurring := WeeklyComboBox.ItemIndex + 1
    else { DateRadioButton.Checked }
    begin
      Recurring := 8;
      DateTime := DatePicker.Date + DateTime;
    end;
  end;
end;

procedure TAlarmDetailsForm.SetData(Alarm: TAlarm);
begin
  with Alarm do
  begin
    // Установить в окне диалога текст сообщения будильника
    MessageEdit.Text := MsgText;
    // Установить в окне диалога время будильника
    TimeMaskEdit.Text := FormatDateTime('hh:mm', DateTime);
    // Установить в окне диалога состояние переключателя звука
    SoundCheckBox.Checked := PlaySound;
    // Установить в окне диалога периодичность будильника
    case Recurring of
      0:    // ежедневно
        EverydayRadioButton.Checked := True;
      1..7: // еженедельно
        begin
          WeeklyRadioButton.Checked := True;
          WeeklyComboBox.ItemIndex := Recurring - 1;
        end;
      8:    // в конкретный день
        begin
          DateRadioButton.Checked := True;
          DatePicker.Date := Int(DateTime);
        end;
    end;
  end;
end;
```

Метод **GetData** просто заполняет поля переданного в параметре объекта **Alarm** значениями, которые установлены в компонентах окна диалога. Метод **SetData** выполняет обратные действия, заполняя компоненты окна диалога значениями, которые содержатся в полях объекта **Alarm**.

На этом с разработкой модуля AlarmDetails покончено и окно диалога **Alarm Details** полностью готово к использованию. Дальше необходимо обеспечить формирование, редактирование и визуализацию списка будильников. Эта задача решается с помощью компонента **ListBox**.

9.3.10. Список

Компонент **ListBox** отображает *список* элементов, которые пользователь может просматривать и выбирать, но не может непосредственно модифицировать. По умолчанию элементами списка являются строки, но могут быть и графические объекты. Элементы могут располагаться в одну или несколько колонок и автоматически сортироваться. При

необходимости обеспечивается возможность прокрутки списка. Компонент **ListBox** находится в палитре компонентов на вкладке **Standard** (рисунок 9.44).

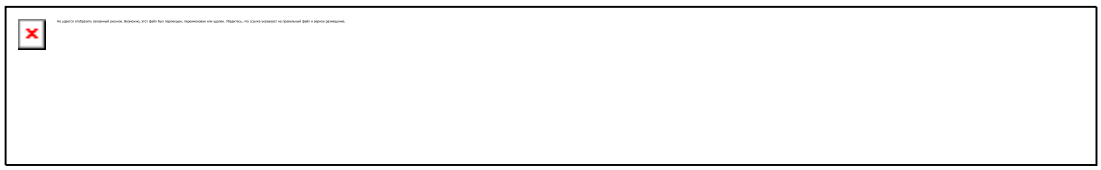


Рисунок 9.44. Компонент ListBox

Его характерные свойства собраны в таблице 9.14.

Свойство	Описание
Align	Способ выравнивания компонента в пределах содержащего компонента.
AutoComplete	Если равно True, то можно быстро выбрать элемент, если начать набирать его текст на клавиатуре.
BevelEdges	Вложенные свойства **beLeft**, **beTop**, **beRight** и **beBottom** определяют видимость соответственно левой, верхней, правой и нижней сторон рельефной рамки.
BevelInner	Внутренний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BevelKind	Вид рельефной рамки: **bkNone** — рамки нет, **bkTile** — рамка с четкими скосами, **bkSoft** — рамка со сглаженными скосами, **bkFlat** — плоская рамка (без скосов).
BevelOuter	Внешний скос рельефной рамки: **bvNone** — скос отсутствует, **bvLowered** — скос внутрь, **bvRaised** — скос наружу; **bvSpace** — скос заменяется отступом.
BorderStyle	Определяет, имеет ли список рамку.
Columns	Количество колонок в списке.
ExtendedSelect	Если равно значению True, то пользователь может выбрать в списке диапазон элементов (однако лишь в том случае, если MultiSelect тоже равно значению True).
IntegralHeight	Если равно значению True, то высота списка автоматически уменьшается, чтобы быть кратной высоте элемента.
ItemHeight	Высота элемента списка, когда значение свойства **Style** равно lbOwnerDrawFixed.
Items	Элементы списка.
MultiSelect	Если равно значению True, то пользователь может выбрать в списке несколько элементов.

ScrollWidth	Логическая ширина списка в пикселях. Если значение свойства **ScrollWidth** больше значения свойства **Width**, то появляется горизонтальная полоса прокрутки. В противном случае полоса прокрутки не показывается.
Sorted	Если равно значению True, то элементы списка сортируются в алфавитном порядке.
Style	Стиль отображения списка (см. табл. 7.14).
OnData	Предназначено для формирования списка элементов перед рисованием. Происходит только в том случае, если свойство **Style** содержит значение **lbVirtual** или **lbVirtualOwnerDraw**.
OnDataFind	Происходит, когда пользователь пытается быстро перейти к элементу, набирая текст элемента на клавиатуре. Обработчик этого события должен на основании введенной пользователем строки вернуть номер соответствующего элемента. Возникает только в том случае, если свойство **Style** содержит значение **lbVirtual** или **lbVirtualOwnerDraw**.
OnDataObject	Происходит при обращении к массиву **Objects** в списке **Items**, но только в том случае, если свойство **Style** содержит значение **lbVirtual** или **lbVirtualOwnerDraw**. Обработчик события должен вернуть соответствующий элементу объект.
OnDrawItem	Происходит при рисовании отдельно взятого элемента списка, но только в том случае, если свойство **Style** содержит одно из следующих значений: **lbOwnerDrawFixed**, **lbOwnerDrawVariable**, **lbVirtualOwnerDraw**.
OnMeasureItem	По замыслу разработчиков событие происходит при расчете высоты отдельно взятого элемента списка перед его рисованием на экране и лишь в том случае, если свойство Style содержит значение **lbOwnerDrawVariable**. Однако из-за дефекта в модуле StdCtrls событие **OnMeasureItem** не срабатывает.

Таблица 9.14. Важнейшие свойства и события компонента ListBox

Особенности хранения и отображения элементов списка определяются свойством **Style**, возможные значения которого описаны в таблице 9.15.

Значение	Описание
LbStandard	Все элементы списка имеют одинаковую высоту, которая рассчитывается исходя из размера шрифта.
LbOwnerDrawFixed	Все элементы списка имеют одинаковую высоту, заданную в свойстве **ItemHeight**. За рисование

	элементов отвечает программист, который должен создать обработчик события **OnDrawItem**.
lbOwnerDrawVariable	По замыслу разработчиков элементы списка имеют разную высоту, определяемую в обработчике события **OnMeasureItem** (из-за дефекта в модуле StdCtrls событие не срабатывает). За рисование элементов отвечает программист, который должен создать обработчик события **OnDrawItem**.
LbVirtual	Элементы списка хранятся отдельно от компонента и запрашиваются с помощью события **OnData**. За рисование элементов отвечает компонент.
LbVirtualOwnerDraw	Элементы списка хранятся отдельно от компонента и запрашиваются с помощью события **OnData**. За рисование элементов отвечает программист, который должен создать обработчик события **OnDrawItem**.

Таблица 9.15. Значения свойства Style компонента ListBox

Шаг 25. Давайте воспользуемся компонентом **ListBox** для организации списка будильников. Активизируйте форму **MainForm**, а затем опустите на нее компонент **ListBox**. Переименуйте компонент в **AlarmListBox** и скорректируйте его местоположение и размеры. Затем установите свойство **TabOrder** в значение 0, чтобы при отображении формы список первым получил фокус ввода (рисунок 9.45).

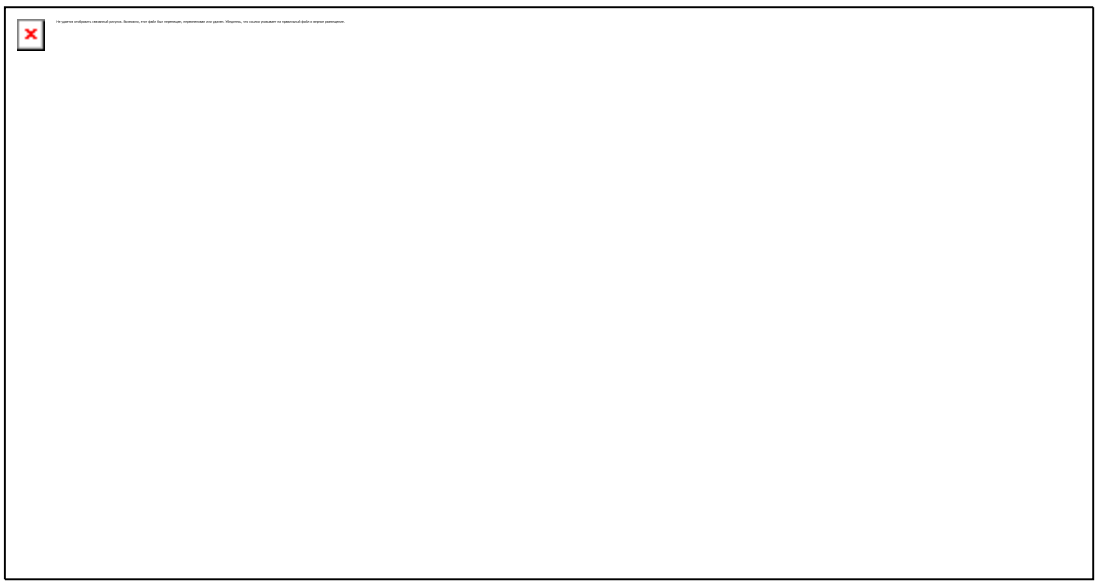


Рисунок 9.45. Компонент ListBox применяется для организации списка будильников

Решим теперь вопрос хранения будильников в компоненте **AlarmListBox**. Для хранения элементов служит свойство **Items**. Свойство **Items** — это объект класса **TStrings**, в нем свойство-массив **Strings** хранит отображаемые строки, а свойство-массив **Objects** — ассоциированные со строками объекты. В нашем примере массив **Strings** будет хранить выдаваемые по сигналу сообщения, а массив **Objects** — соответствующие им объекты класса **TAlarm**.

Теоретически все понятно, осталось реализовать все это практически. Создание, редактирование и удаление будильника осуществляется по щелчкам на кнопках **NewButton**,

EditButton и **DeleteButton** соответственно. Поэтому в них требуется создать обработчики события **OnClick**.

Шаг 26. В кнопке **New...** обработчик события **OnClick** уже существует, но его необходимо доработать:

```
procedure TMainForm.NewButtonClick(Sender: TObject);
var
  Alarm: TAlarm;
begin
  AlarmDetailsForm := TAlarmDetailsForm.Create(Self);
  try
    // Выполнить диалог
    if AlarmDetailsForm.ShowModal = mrOK then
    begin
      // Создать новый объект будильника
      Alarm := TAlarm.Create;
      // Получить параметры будильника из диалога
      AlarmDetailsForm.GetData(Alarm);
      // Добавить будильник в список и выбрать его
      AlarmListBox.ItemIndex := AlarmListBox.Items.AddObject(
        Alarm.GetAlarmStr, Alarm);
    end;
  finally
    AlarmDetailsForm.Free;
  end;
end;
```

Метод **NewButtonClick** создает окно диалога **Alarm Details** и выполняет его в монопольном режиме. Если диалог завершается щелчком кнопки OK, создается новый объект будильника и в него переносятся данные из окна диалога. Затем этот объект добавляется в список **AlarmList** и его номер присваивается свойству списка **ItemIndex**. В результате новый элемент становится выделенным.

Вы, разумеется, хотите проверить работу новоиспеченного метода. Сейчас мы так и сделаем, но прежде нужно решить небольшой вопрос. Дело в том, что при уничтожении блока списка освобождаются только строки, но не освобождаются ассоциированные с ними объекты. Хотя память объектов так или иначе освобождается при завершении приложения, мы рекомендуем всегда освобождать память явно. Это считается "хорошим тоном" программирования и иногда позволяет выявить скрытые ошибки. Освобождение использованных в форме динамических данных осуществляется в обработчике события **OnDestroy**. Для формы **MainForm** он должен быть таким:

```
procedure TMainForm.FormDestroy(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to AlarmListBox.Items.Count - 1 do
    AlarmListBox.Items.Objects[I].Free;
end;
```

После того как вы написали этот обработчик, выполните компиляцию программы и запустите ее. Попытайтесь добавить в список несколько будильников. Если это у вас получилось, перейдем к следующему шагу — программированию реакции на нажатия кнопок **Edit...** и **Delete**.

Шаг 27. Создайте в компоненте **EditButton** обработчик события **OnClick**:

```
procedure TMainForm.EditButtonClick(Sender: TObject);
var
  Alarm: TAlarm;
  SavedIndex: Integer;
begin
  AlarmDetailsForm := TAlarmDetailsForm.Create(Self);
  try
    // Получить выбранный будильник
    with AlarmListBox do Alarm := TAlarm(Items.Objects[ItemIndex]);
    // Установить управляющие элементы диалога в соответствии с
    // параметрами будильника
    AlarmDetailsForm.SetData(Alarm);
    // Выполнить диалог
    if AlarmDetailsForm.ShowModal = mrOK then
    begin
      // Получить из диалога новые параметры будильника
      AlarmDetailsForm.GetData(Alarm);
      with AlarmListBox do
      begin
        // Запомнить номер выбранного в списке элемента
        SavedIndex := ItemIndex;
        // Изменить текст элемента
        // При этом элемент перестает быть выбранным
        Items.Strings[ItemIndex] := Alarm.GetAlarmStr;
        // Восстановить номер выбранного в списке элемента
        ItemIndex := SavedIndex;
      end;
    end;
  finally
    AlarmDetailsForm.Free;
  end;
end;
```

Этот метод создает окно диалога **Alarm Details**, инициализирует его компоненты данными из выбранного в списке объекта будильника, а затем выполняет диалог в монопольном режиме. Если диалог завершился щелчком на кнопке OK, то данные из окна диалога переносятся обратно в объект будильника и соответственно изменяется отображаемая в блоке списка строка. Так как в результате последнего действия в списке пропадает полоса выбора (свойство **ItemIndex** получает значение -1), номер выделенного элемента предварительно сохраняется в локальной переменной **SavedIndex**, а затем восстанавливается.

Шаг 28. Осталось создать обработчик события **OnClick** в компоненте **DeleteButton**:

```
procedure TMainForm.DeleteButtonClick(Sender: TObject);
begin
  with AlarmListBox do
  begin
    // Разрушить объект будильника
    Items.Objects[ItemIndex].Free;
    // Удалить из списка соответствующую объекту строку
    Items.Delete(ItemIndex);
  end;
end;
```

Метод **DeleteButtonClick** удаляет объект будильника и соответствующую ему строку в списке.

Обработчики событий для всех кнопок заданы, однако не спешите запускать приложение. Необходимо позаботиться о том, чтобы кнопки **Edit...** и **Delete** были доступны или недоступны в зависимости от того, выделен в списке элемент или нет. Как бы это сделать попроще? Первое решение, которое напрашивается — это вставить необходимые проверки в обработчики событий кнопок. Это неплохое решение, но оно больше подходит тем, кто привык решать задачу в лоб. Мы пойдем другим путем, воспользовавшись событием **OnIdle** объекта **Application**.

В объекте **Application** происходит событие **OnIdle** в период простоя программы, например во время ожидания пользовательского ввода. Благодаря этому событию программа может выполнять некоторую фоновую работу, которая в нашем случае заключается в управлении состоянием кнопок.

Для создания обработчика события **OnIdle** объекта **Application** воспользуемся уже знакомым вам компонентом **ApplicationEvents** (см. главу 8).

Шаг 29. Поместите в форму компонент **ApplicationEvents**, дайте ему одноименный идентификатор и создайте обработчик события **OnIdle**:

```
procedure TMainForm.ApplicationEventsIdle(Sender: TObject;
  var Done: Boolean);
begin
  EditButton.Enabled := AlarmListBox.ItemIndex <> -1;
  DeleteButton.Enabled := AlarmListBox.ItemIndex <> -1;
  Done := True; // предотвращает непрерывную генерацию события OnIdle
end;
```

В передаваемом по ссылке параметре Done метод возвращает результат своей работы. Значение True показывает, что метод нужно вызывать не постоянно в течении простоя приложения, а только по одному разу в начале каждого периода простоя.

А сейчас выполните компиляцию, запустите программу и тщательно протестируйте работу главной формы (рисунок 9.46).

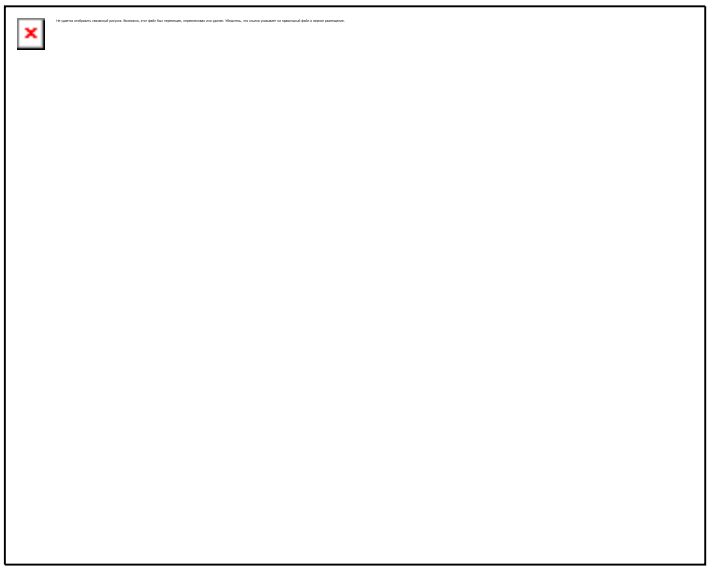


Рисунок 9.46. В этом окне создается список будильников

Будильники можно создавать, добавлять, удалять. Нам осталось сделать последний шаг — заставить будильники "звонить". Для этого нужно периодически вызывать метод **CheckTime** у каждого помещенного в список объекта **TAlarm**. Периодические по времени действия выполняются с помощью таймера, о котором мы дальше и поговорим.

9.4. Законченное приложение для выдачи сигналов в заданные моменты времени

9.4.1. Таймер

Таймер — это системный генератор событий, который периодически сообщает программе о завершении заданного промежутка времени. Интервал времени между событиями таймера может устанавливаться в диапазоне от 1 до 65535 миллисекунд. Используя таймер, учитывайте, что интервалы между этими событиями оказываются неточными из-за накладных расходов механизма обработки событий Windows.

В библиотеке VCL прием событий таймера обеспечивает компонент **Timer**. Он расположен в палитре компонентов на вкладке **System** (рисунок 9.47). Им мы и воспользуемся для "оживления" будильников в приложении Alarms.

Рисунок 9.47. Компонент Timer

Шаг 30. Активизируйте форму **MainForm**. Затем поместите в нее компонент **Timer** (рисунок 9.48). Если хотите, дайте ему любое имя.

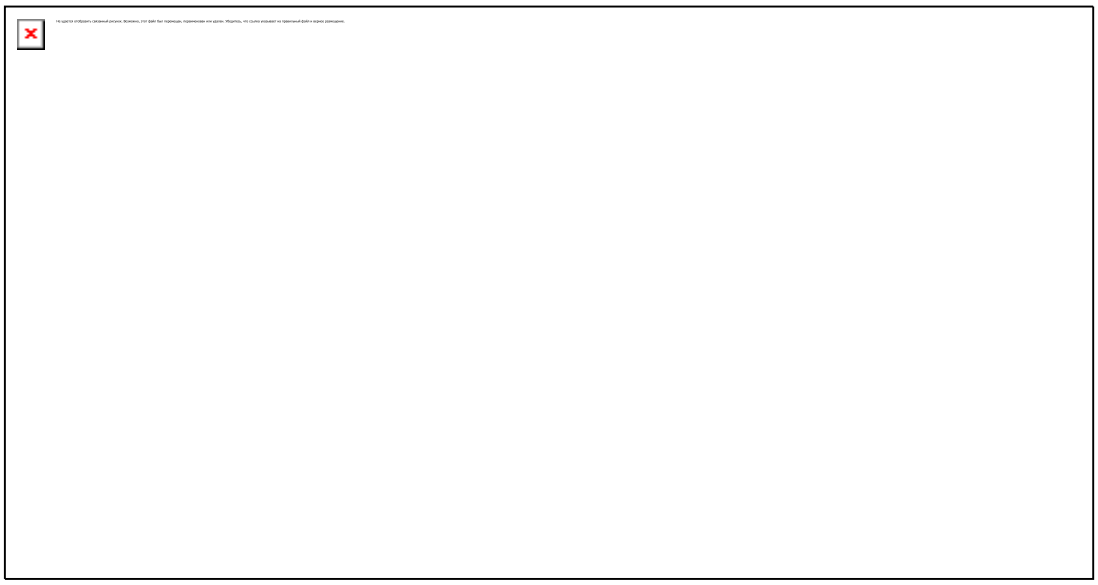


Рисунок 9.48. Компонент Timer — на форме

Шаг 31. Интервал времени между событиями таймера задается в миллисекундах как значение свойства **Interval**. Изначально интервал равен 1000 миллисекунд (1 секунда). Частота контроля будильников должна быть два раза в секунду, поэтому установите свойство **Interval** в значение 500.

Шаг 32. Через заданные в свойстве **Interval** промежутки времени в компоненте **Timer** происходит событие **OnTimer** (единственное событие этого компонента). Для контроля за будильниками нам нужно создать обработчик этого события:

```
procedure TMainForm.TimerTimer(Sender: TObject);
var
  I: Integer;
begin
  for I := 0 to AlarmListBox.Items.Count - 1 do
    with AlarmListBox.Items.Objects[I] as TAlarm do
      CheckTime;
end;
```

Смысл выполняемых действий очевиден: у каждого объекта в списке будильников вызывается метод **CheckTime**. Таким образом, каждый будильник периодически проверяет свое время и, если нужно, выдает предупреждение (рисунок 9.49).

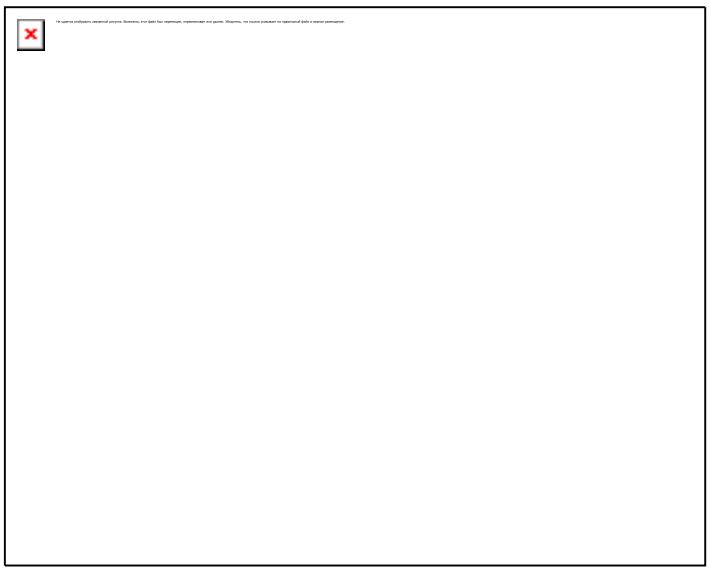


Рисунок 9.49. Когда будильник срабатывает, звучит сигнал и на экран выдается сообщение

Выполните компиляцию и запустите программу, чтобы проверить работу всех элементов программы. Ура! Все работает, терпение и упорство вознаграждены.

9.4.2. Файлы настроек

Программа Alarms имеет один существенный недостаток: после его завершения все установленные будильники теряются, так что при следующем запуске программы их приходится создавать снова. Для решения этой проблемы необходимо, чтобы между сеансами работы программы будильники хранились в конфигурационном файле на диске.

Сохранение и восстановление конфигурации осуществляется в Windows с помощью так называемых файлов настроек. *Файл настроек (initialization file)* — это текстовый файл, состоящий из секций. *Секция* начинается с имени, заключенного в квадратные скобки. В каждой секции содержатся определения некоторых связанных по смыслу параметров, представленные в виде пар Имя=Значение. Примером файла настроек может служить файл настроек проекта в системе Delphi. В нашем проекте это файл Alarms.dof.

Структуру файла настроек для программы Alarms выберем так, чтобы каждому будильнику соответствовала отдельная секция. Число секций, т.е. будильников, будем хранить в параметре AlarmCount секции Global Options. Вот как могло бы выглядеть содержимое файла:

```
[Global Options]
AlarmCount=3

[Alarm1]
Message=Dinner !
Time=13:00
PlaySound=1
Recurring=0

[Alarm2]
Message=Tennis training
Time=16:00
PlaySound=1
Recurring=1

[Alarm3]
Message=My favourite TV-show...
Time=22:30
PlaySound=0
Recurring=8
Date=02/25/96
```

Чтение и запись файла настроек осуществляется с помощью объектов **TIniFile** (заметьте, они не являются компонентами). Класс **TIniFile** описан в модуле **IniFiles**. Этот модуль необходимо самостоятельно добавить в вызывающий модуль с помощью оператора **uses**. При создании объекта **TIniFile** ему в конструктор передается имя INI-файла. Позже это имя можно узнать, обратившись к свойству **FileName**. Если в имени файла маршрут не был указан, считается что INI-файл находится в каталоге системы Windows.

Чтение переменных из INI-файла выполняется с помощью описанных ниже методов. В этих методах название секции передается в параметре *Section*, имя переменной – в параметре *Ident*, а значение по умолчанию – в параметре *Default*.

ReadBool(const Section, Ident: string; Default: Boolean): Boolean — возвращает значение булевской переменной.

ReadInteger(const Section, Ident: string; Default: Longint): Longint — возвращает значение целочисленной переменной.

ReadString(const Section, Ident, Default: string): string — возвращает значение строковой переменной.

ReadSection(const Section: string; Strings: TStrings) — читает из заданной секции имена всех переменных и помещает их в объект класса **TStrings**.

ReadSectionValues(const Section: string; Strings: TStrings) — читает из заданной секции все пары Имя=Значение и помещает их в список. Для доступа к Значению по Имени в объектах класса TStrings существуют свойства-массивы **Names** и **Values**.

При чтении значений из INI-файла может оказаться, что заданный идентификатор или секция отсутствует. В этом случае ошибки не происходит, а функции **ReadBool**, **ReadInteger** и **ReadString** возвращают значение, переданное в параметре *Default*.

Кроме методов чтения существуют также методы записи переменных INI-файла, которые описаны ниже. В этих методах название секции передается в параметре *Section*, имя переменной – в параметре *Ident*, а значение переменной — в параметре *Value*.

WriteBool(const Section, Ident: string; Value: Boolean) – записывает в INI-файл булевское значение.

WriteInteger(const Section, Ident: string; Value: Longint) – записывает в INI-файл целочисленное значение.

WriteString(const Section, Ident, Value: string) – записывает в INI-файл строковое значение.

Если в момент записи значения оказывается, что заданные секция и (или) идентификатор отсутствуют, они создаются.

Удаление секций INI-файла осуществляется с помощью метода **EraseSection**, в который передается единственный параметр — название секции.

Шаг 33. Давайте воспользуемся описанными методами для сохранения и восстановления будильников в программе ALARMS. Работу по сохранению и восстановлению параметров одного будильника лучше всего поручить классу **TAlarm**. Для этого добавьте в его описание два новых метода — **LoadFromIniFile** и **SaveToIniFile**.

```
type
  TAlarm = class
    ...
    procedure LoadFromIniFile(IniFile: TIniFile; const Section: string);
    procedure SaveToIniFile(IniFile: TIniFile; const Section: string);
  end;
```

Метод **LoadFromIniFile** предназначен для чтения из INI-файла полей объекта, а метод **SaveToIniFile** — для записи в INI-файл полей объекта. Секция INI-файла, с которой работают эти методы, передается в параметре Section.

Шаг 34. Наберите программный код методов в разделе **implementation**:

```
procedure TAlarm.LoadFromIniFile(IniFile: TIniFile; const Section: string);
begin
  with IniFile do
  begin
    // Прочитать текст сообщения
    MsgText := ReadString(Section, 'Message', 'Reminder !');
    // Прочитать строковое значение времени и
    // преобразовать его в формат TdateTime
    DateTime := StrToTime(ReadString(Section, 'Time', TimeToStr(Time)));
    // Прочитать состояние переключателя звука
    PlaySound := ReadBool(Section, 'PlaySound', True);
    // Прочитать значение периодичности
    Recurring := ReadInteger(Section, 'Recurring', 0);
    if Recurring = 8 then
      // Прочитать строковое значение даты и
      // преобразовать его в формат TDateTime
      DateTime := StrToDate(
        ReadString(Section, 'Date', DateToStr(Date))) + DateTime;
  end;
end;

procedure TAlarm.SaveToIniFile(IniFile: TIniFile; const Section: string);
begin
  with IniFile do
  begin
    // Записать текст сообщения
    WriteString(Section, 'Message', MsgText);
    // Преобразовать время в строку и записать строку в INI-файл
    WriteString(Section, 'Time', FormatDateTime('hh:mm', DateTime));
    // Записать значение переключателя звука
    WriteBool(Section, 'PlaySound', PlaySound);
    // Записать значение периодичности
    WriteInteger(Section, 'Recurring', Recurring);
    if Recurring = 8 then
      // Преобразовать дату в строку и записать строку в INI-файл
      WriteString(Section, 'Date', DateToStr(DateTime));
  end;
end;
```

Шаг 35. Перейдем теперь от сохранения и восстановления одного будильника к загрузке и восстановлению всего списка. Эти действия следует выполнять соответственно при создании и уничтожении главной формы программы, т.е. в событиях **OnCreate** и **OnDestroy**. Создайте форме **MainForm** обработчик события **OnCreate** и доработайте обработчик события **OnDestroy** (не забудьте подключить модуль **IniFiles**):

```
procedure TMainForm.FormCreate(Sender: TObject);
var
  IniFile: TIniFile;
  Alarm: TAlarm;
  AlarmCount, I: Integer;
begin
  IniFile := TIniFile.Create('Alarms.ini');
  try
    // Прочитать число будильников
    AlarmCount := IniFile.ReadInteger('Global Options', 'AlarmCount', 0);
    // Прочитать список будильников
    for I := 1 to AlarmCount do
    begin
      // Создать будильник
      Alarm := TAlarm.Create;
      // Прочитать параметры будильника из соответствующей секции
      Alarm.LoadFromIniFile(IniFile, 'Alarm' + IntToStr(I));
      // Добавить будильник в список
      AlarmListBox.Items.AddObject(Alarm.GetAlarmStr, Alarm);
    end;
  finally
    IniFile.Free;
  end;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
var
  IniFile: TIniFile;
  I: Integer;
begin
  IniFile := TIniFile.Create('Alarms.ini');
  try
    // Записать число будильников
    IniFile.WriteInteger('Global Options', 'AlarmCount',
      AlarmListBox.Items.Count);
    // Записать список будильников
    for I := 0 to AlarmListBox.Items.Count - 1 do
      with AlarmListBox.Items.Objects[I] as TAlarm do
        // Записать параметры будильника в соответствующую секцию
        SaveToIniFile(IniFile, 'Alarm' + IntToStr(I + 1));
  finally
    IniFile.Free;
    for I := 0 to AlarmListBox.Items.Count - 1 do
      AlarmListBox.Items.Objects[I].Free;
  end;
end;
```

Вот пожалуй и все. Сохраните проект, выполните его компиляцию и запустите программу. Создайте несколько будильников, закройте программу, а затем запустите ее снова... Будильники на месте. Кстати, утилиту Alarms можно поместить в папку **StartUp**, тогда она всегда будет у вас под рукой.

9.5. Многостраничные окна диалога

9.5.1. Страницы с закладками

В сложных компьютерных системах предусмотрена настройка десятков различных параметров, которые пользователь просто обязан установить. Это означает либо перегруженность диалоговых окон, либо слишком большое их количество. Решение этой проблемы заключается в организации многостраничных окон диалога. Они создаются с помощью компонента, который изображается в виде множества снабженных закладками страниц, называемых *вкладками* (рисунок 9.50):

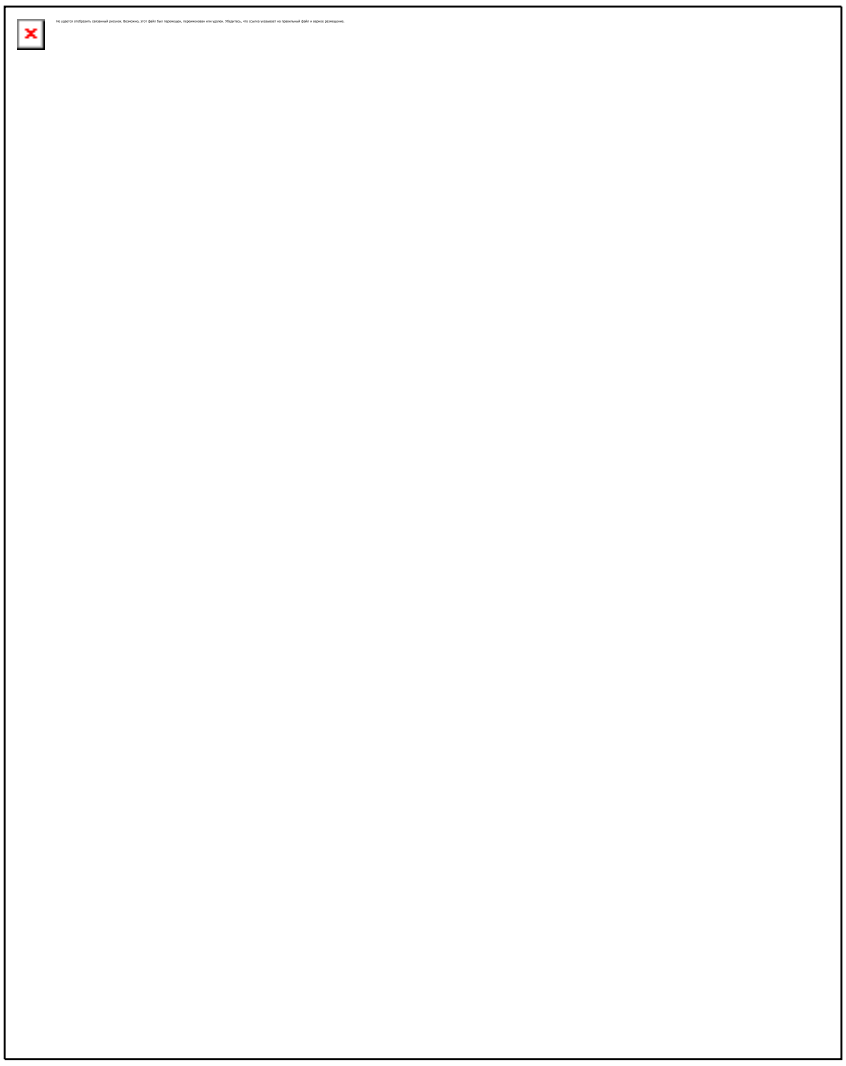


Рисунок 9.50. Пример вкладок в окне диалога

Вкладки относятся к разряду сложных компонентов, но в целом они упрощают пользовательский интерфейс, заменяя несколько связанных по смыслу форм. А чем меньше форм, тем легче пользователю ориентироваться в программе.

Создание вкладок рассмотрим на следующем примере. Представьте, что вкладками являются экзаменационные билеты по трем научным дисциплинам: математике, физике, химии. Каждая вкладка содержит один вопрос с возможными вариантами ответа. Пользователь должен на каждой вкладке выбрать правильный вариант ответа и завершить ввод щелчком на кнопке **Result**. При всех правильных ответах он получит оценку "отлично", при двух ответах — "хорошо", при ответе лишь на один вопрос — "удовлетворительно", а при всех неправильных ответах — "плохо".

Шаг 1. Приступим к реализации примера. Начните новый проект и установите для главной формы следующие свойства:

Name = ExamForm

Caption = Экзамен

BorderStyle = bsDialog

Position = poScreenCenter

Размеры формы подберите по своему усмотрению.

Шаг 2. Теперь поместите в форму компонент **PageControl**. Вы найдете его в палитре компонентов на вкладке **Win32** (рисунок 9.51).

Рисунок 9.51. Компонент PageControl

Характерные свойства компонента **PageControl** кратко описаны в таблице 9.16.

Свойство	Описание
ActivePage	Активная вкладка (страница).
Align	Способ выравнивания компонента в пределах содержащего компонента.
DockSite	Определяет, используется ли компонент PageControl для стыковки других компонентов.
HotTrack	Подсвечивает закладку при наведении на нее указателя мыши.
Images	Список значков, отображаемых на закладках. Свойство **Images** используется совместно со свойством **ImageIndex** компонентов **TabSheet**. Компонент **PageControl** автоматически назначает каждой закладке номер значка в соответствии с очередностью добавления вкладок, однако программист может вручную указать номер значка.
MultiLine	Располагает закладки в несколько рядов.
OwnerDraw	Позволяет программно рисовать закладки в обработчике события **OnDrawTab**. Если свойство **OwnerDraw** равно значению False, то закладки имеют стандартный вид и событие **OnDrawTab** не происходит.
Pages	Массив вкладок (страниц). Каждая вкладка является объектом класса **TTabSheet**. Свойство Pages доступно только из программы.
PageCount	Общее количество вкладок. Доступно только из программы.
RaggedRight	Если равно значению True, то при включенном режиме **MultiLine** закладки не выравниваются на ширину компонента.
ScrollOpposite	Способ организации рядов закладок. Если равно значению False, то все ряды расположены вместе, например вверху. Если равно значению True,

	неактивные ряды переносятся на другую сторону компонента, например вниз.
Style	Стиль закладок: **tsTabs** — обычные трехмерные закладки, **tsFlatButtons** — плоские закладки, **tsButtons** — закладки в виде кнопок.
TabIndex	Номер выбранной закладки (первая закладка имеет номер 0).
TabPosition	Местоположение закладок: **tpTop** — сверху, **tpRight** — справа, **tpLeft** — слева, **tpBottom** — снизу.
TabWidth, TabHeight	Ширина и высота закладок. Если эти свойства равны нулю, то размеры закладок подбираются автоматически, исходя из размеров надписей.
OnChange	Происходит после смены закладки.
OnChanging	Происходит перед сменой закладки.
OnDrawTab	Происходит при рисовании закладки на экране. Требует, чтобы свойство **OwnerDraw** содержало значение True.
OnGetImageIndex	Обработчик этого события должен вернуть номер значка для отображаемой закладки.
OnGetSiteInfo	Происходит, когда у компонента запрашивается место для стыковки.

Таблица 9.16. Важнейшие свойства и события компонента PageControl

Шаг 3. Первоначально компонент **PageControl** не содержит ни единой вкладки. Для создания вкладки щелкните правой кнопкой мыши на компоненте и выберите в контекстном меню команду **New Page** (рисунок 9.52).

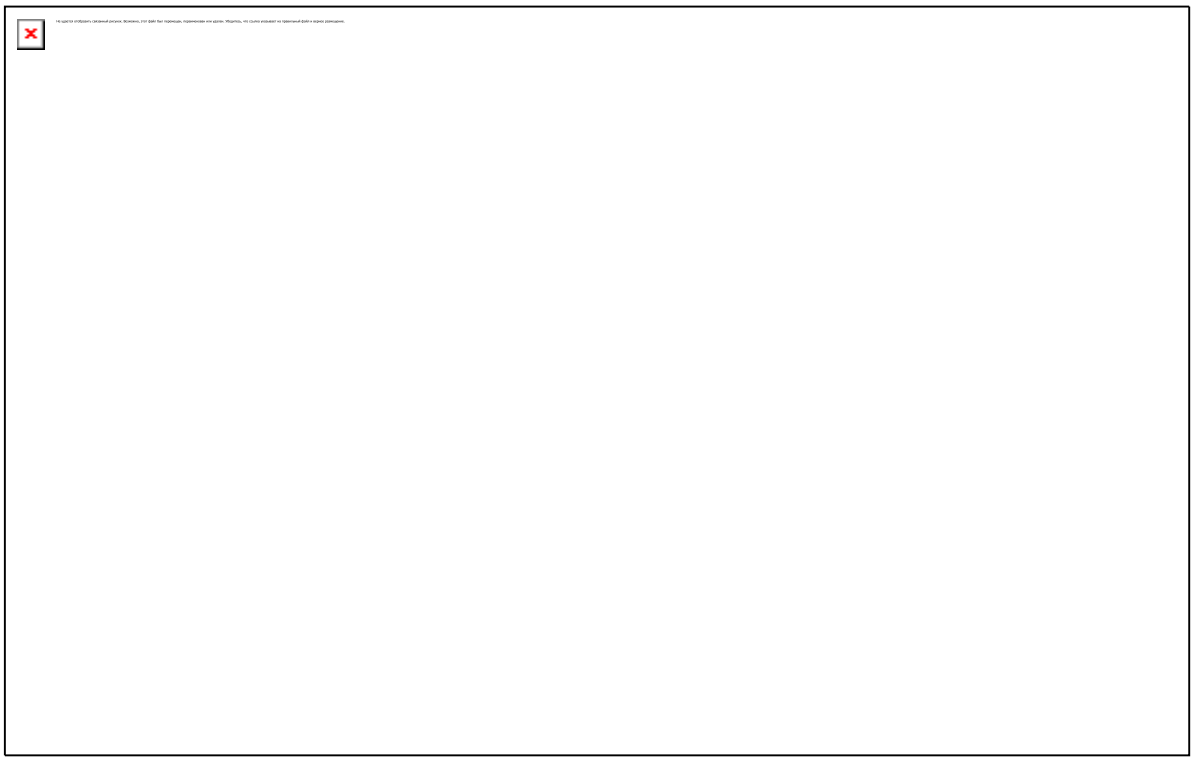


Рисунок 9.52. Создание новой вкладки в компоненте PageControl

Будет создана первая вкладка с заголовком **TabSheet1** (рисунок 9.53).

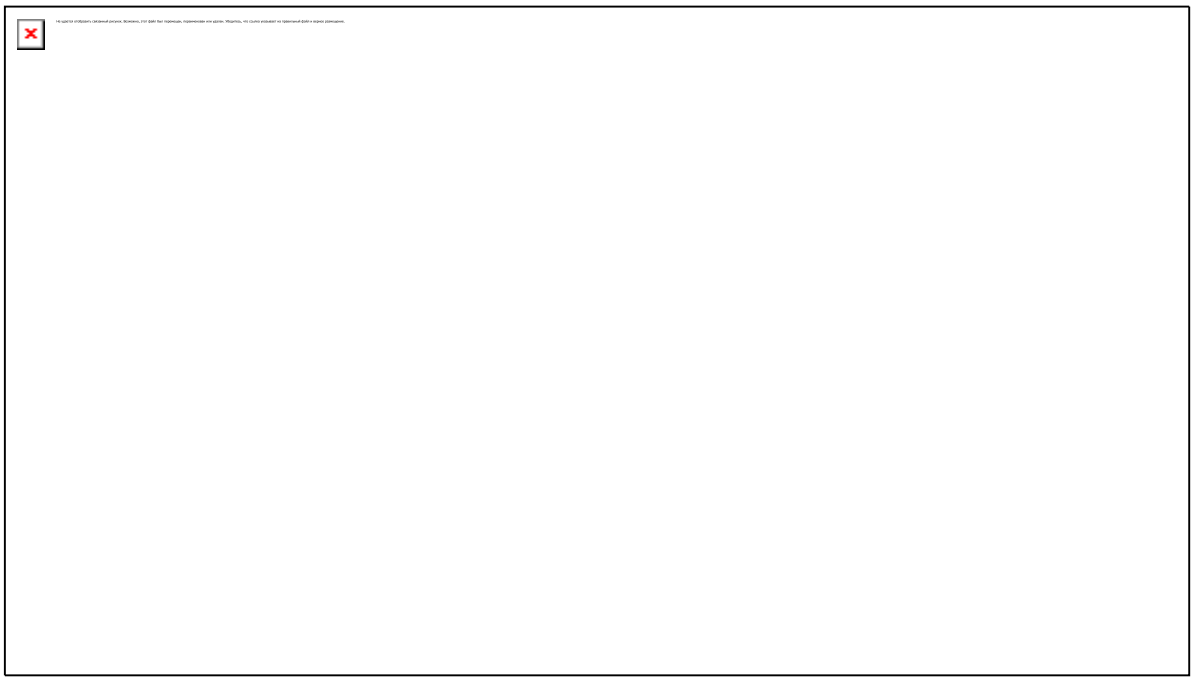


Рисунок 9.53. В компоненте PageControl создана первая вкладка

Каждая вкладка в компоненте **PageControl** представлена объектом класса **TTabSheet**. Свойства отдельной вкладки устанавливаются в окне свойств. Характерные свойства кратко описаны в таблице 9.17.

Свойство	Описание
BorderWidth	Ширина внутреннего отступа.

Caption	Надпись на закладке.
Highlighted	Подсветка закладки цветом.
ImageIndex	Номер значка в списке **Images** компонента **PageControl**. Значок отображается рядом с названием закладки. Отрицательное значение свойства **ImageIndex** говорит о том, что для закладки значок не задан.
PageControl	Ссылка на компонент PageControl, которому принадлежит вкладка. Доступно только из программы.
PageIndex	Номер вкладки в массиве **Pages** компонента **PageControl**.
TabIndex	Номер вкладки среди видимых вкладок. Если вкладка не видна, то свойство TabIndex равно -1. Свойство доступно только программно и только для чтения.
TabVisible	Определяет, видна ли закладка.
OnHide	Происходит при переключении на другую вкладку.
OnShow	Происходит при активизации вкладки.

Таблица 9.17. Важнейшие свойства компонента TTabSheet

Шаг 4. Перейдите к окну свойств и замените текст закладки, вписав в свойстве **Caption** значение **Mathematics**. Действуя аналогично, добавьте вкладки **Physics** и **Chemistry** (рисунок 9.54).

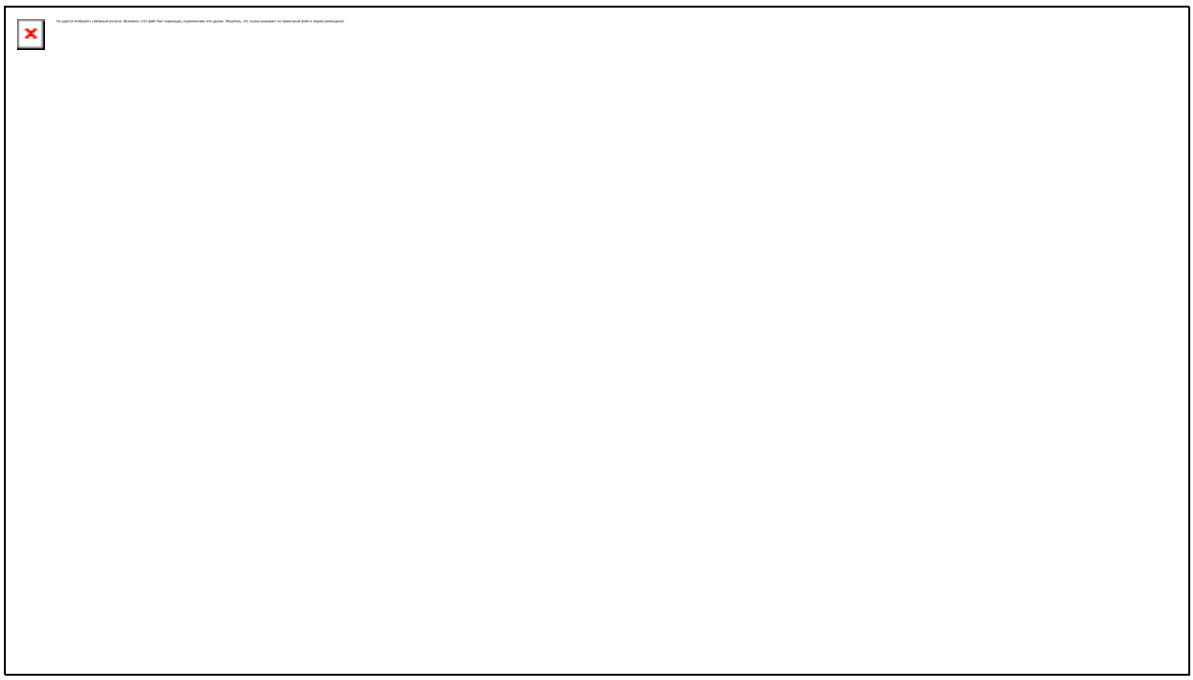


Рисунок 9.54. В компоненте PageControl созданы три вкладки

Шаг 5. Ну вот, у нас уже есть три пустых вкладки и можно приступать к наполнению их содержанием. Сначала щелчком мыши активизируйте вкладку **Mathematics**. Затем поместите на нее группу взаимоисключающих переключателей — компонент **RadioGroup**.

Заголовок группы будет содержать условие вопроса, а текст переключателей — возможные варианты ответа. Вопрос по математике будет из области тригонометрии (рисунок 9.55).

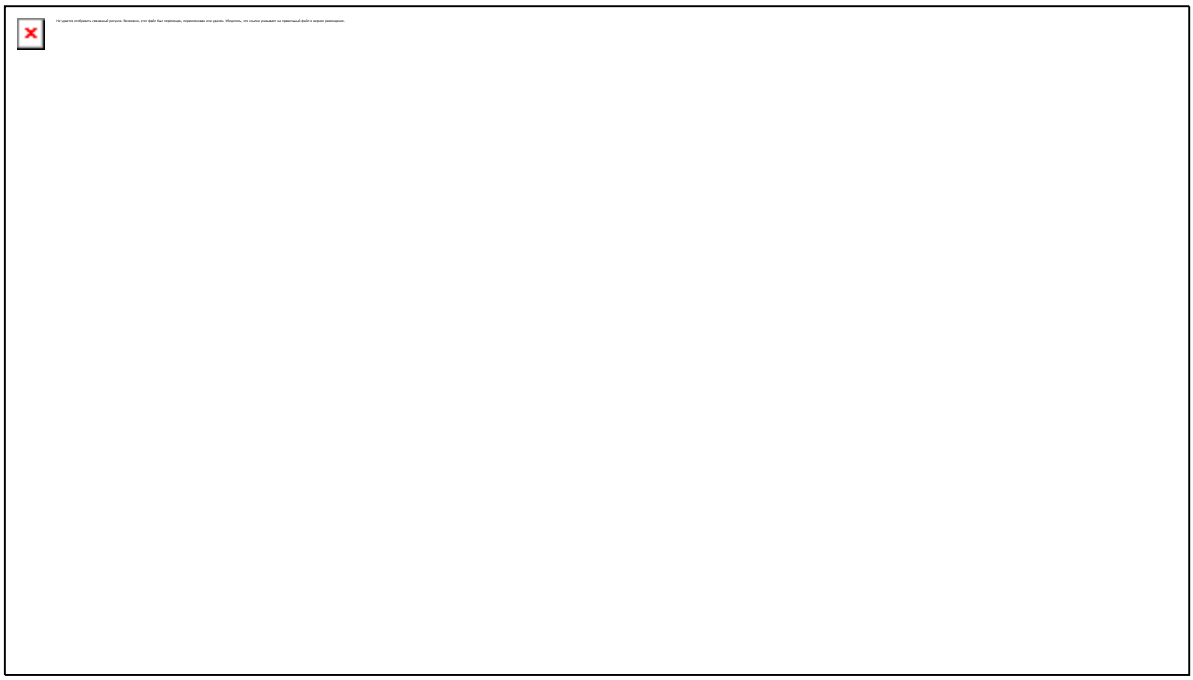


Рисунок 9.55. Содержимое первой вкладки

Шаг 6. Чтобы группа переключателей выглядела так, как на рисунке, подберите ей соответствующие размеры и установите значения следующих свойств:

Caption = The right expression is (правильным является выражение)

Items = sin 50° < cos 50°

sin 50° > cos 50°

sin 50° = cos 50°

ItemIndex = 0 (номер варианта принимаемый по умолчанию)

Tag = 1 (номер правильного варианта, считая от нуля)

Свойство **Tag** не несет смысловой нагрузки в компонентах среды Delphi. Поэтому его можно использовать по своему усмотрению. Помещая в него номер правильного ответа, мы заботимся об упрощении подсчета оценки. Когда пользователь укажет ответ, мы сравним значения свойств **ItemIndex** и **Tag**. Ответ будет считаться правильным лишь в том случае, если номера совпадут.

Шаг 7. Билет по математике готов, приступим к подготовке билета по физике. Активизируйте вкладку **Physics** и поместите на нее компонент **RadioGroup**. Подберите для него подходящие размеры и установите следующие свойства:

Caption = When the ice in water dissolves then (когда лед в воде тает)

Items = the level of water becomes higher (уровень воды поднимается)

the level of water becomes lower (уровень воды понижается)

the level of water remains unchanged (уровень воды остается неизменным)

ItemIndex = 0 (номер варианта принимаемый по умолчанию)

Tag = 2 (номер правильного варианта, считая от нуля)

Результат должен быть таким, как на рисунке 9.56:

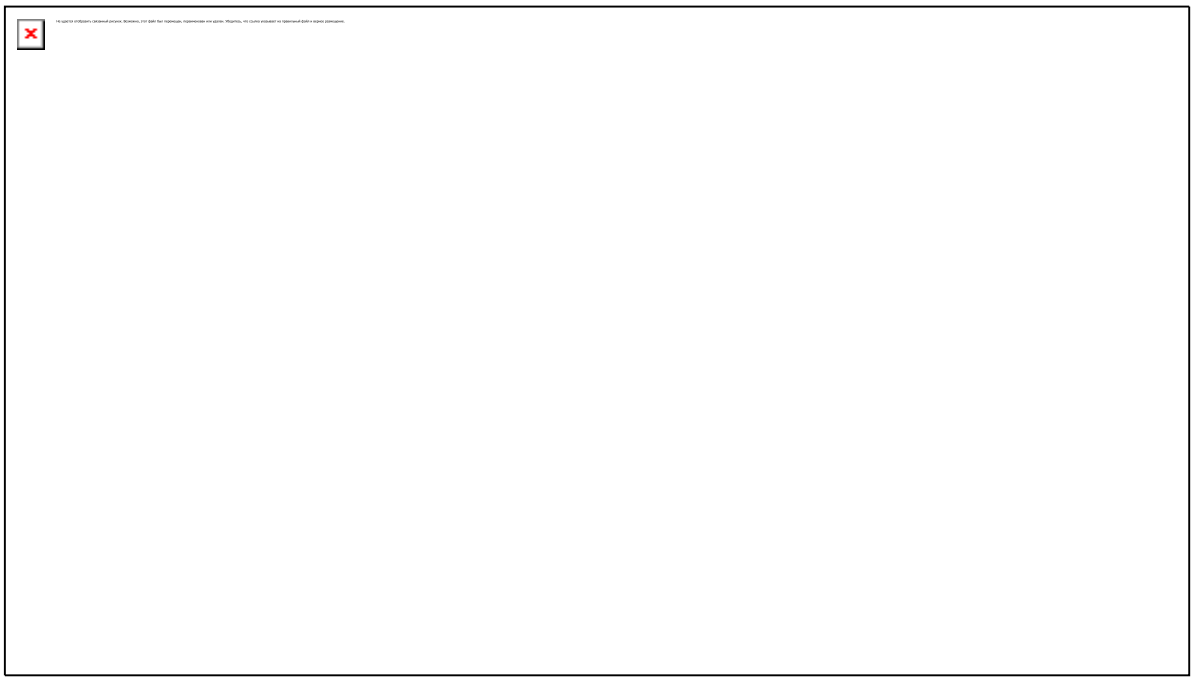


Рисунок 9.56. Содержимое второй вкладки

Шаг 8. Осталось создать билет по химии. Мы надеемся, что после всех предыдущих испытаний это не составит для вас труда. Кратко поясним, что нужно сделать. Активизируйте вкладку **Chemistry** и поместите на нее компонент **RadioGroup**. Подберите для него подходящие размеры и установите следующие свойства:

Caption = The right way of mixing acid and water is (чтобы разбавить кислоту, нужно)

Items = to add acid to water (добавить кислоту в воду)

to add water to acid (добавить воду в кислоту)

ItemIndex = 0 (номер варианта принимаемый по умолчанию)

Tag = 0 (номер правильного варианта, считая от нуля)

После всех ваших действий вкладка **Chemistry** будет выглядеть так, как на рисунке 9.57.

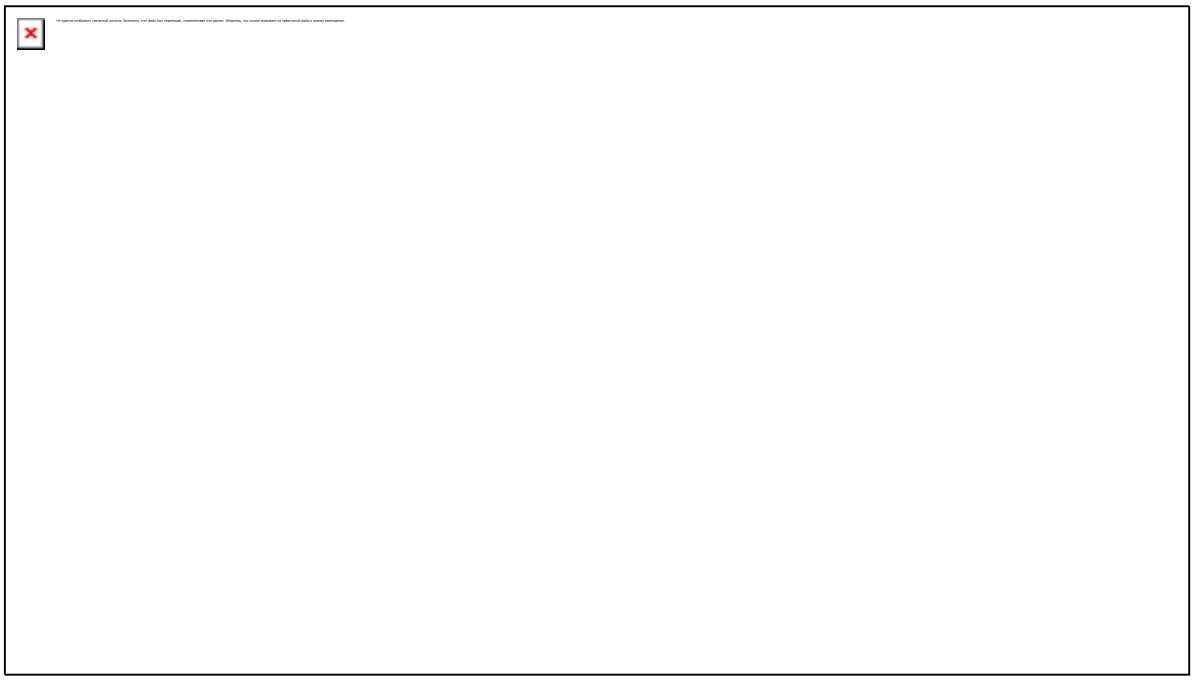


Рисунок 9.57. Содержимое третьей вкладки

Шаг 9. Все вкладки с экзаменационными билетами вроде бы готовы, но как вы считаете, какая из них будет активной при запуске программы? Конечно та, которая осталась активной при проектировании, т.е. вкладка **Chemistry**. А надо, чтобы первой оказалась вкладка **Mathematics**. Поэтому активизируйте ее щелчком мыши. Кроме того, установите у формы свойство **ActiveControl** в значение **RadioGroup1**. Компонент, указанный в свойстве **ActiveControl** первым получает фокус ввода при появлении формы на экране.

Шаг 10. Теперь вас ждет самая ответственная работа — выставление пользователю оценки. Для этого поместите в форму две кнопки (компонент **Button**). Первая кнопка предназначена для выдачи результата экзамена, сделайте ее свойства такими:

Name = ResultButton

Caption = Result

Default = True

Вторая кнопка служит для закрытия окна, ее сделайте такой:

Name = CloseButton

Caption = Close

Cancel = True

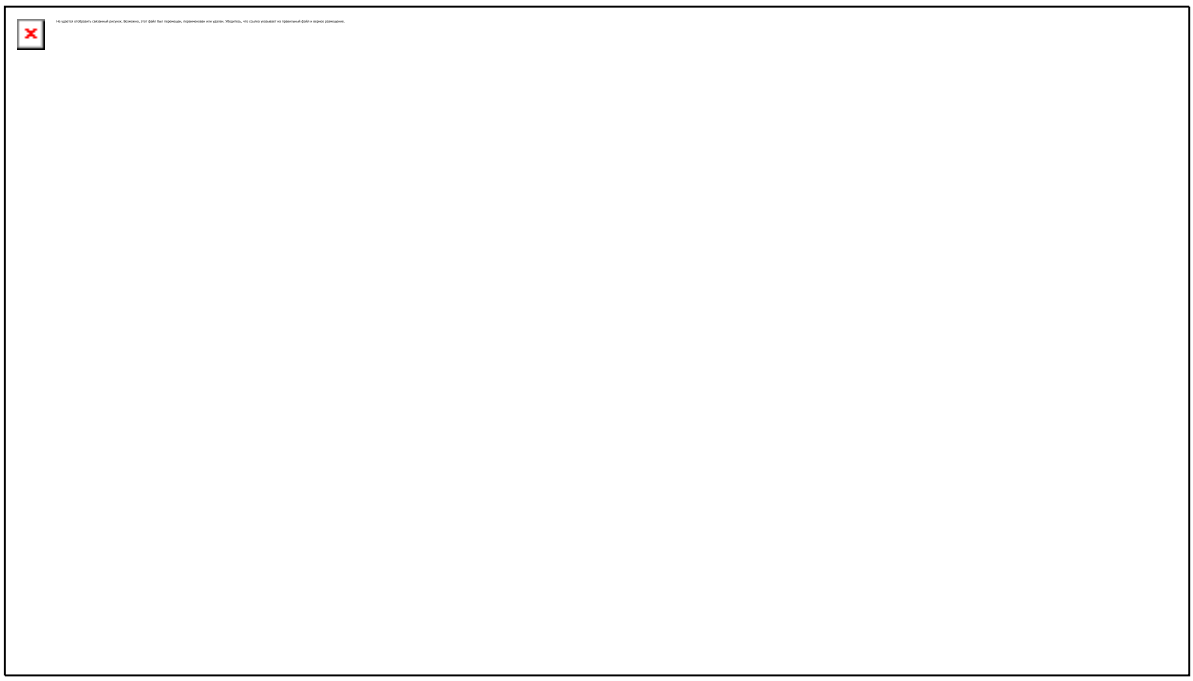


Рисунок 9.58. Кнопка Result выдает оценку, кнопка Close завершает диалог

Шаг 11. Создайте для кнопок следующие обработчики события **OnClick**:

```
procedure TExamForm.ResultButtonClick(Sender: TObject);
const
  MarkText: array[0..3] of string =
    ('Неудовлетворительно', 'Удовлетворительно', 'Хорошо', 'Отлично');
var
  Mark: Integer;
begin
  Mark := 0;
  if RadioGroup1.ItemIndex = RadioGroup1.Tag then
    Mark := Mark + 1;
  if RadioGroup2.ItemIndex = RadioGroup2.Tag then
    Mark := Mark + 1;
  if RadioGroup3.ItemIndex = RadioGroup3.Tag then
    Mark := Mark + 1;
  ShowMessage(MarkText[Mark]);
end;

procedure TExamForm.CloseButtonClick(Sender: TObject);
begin
  Close;
end;
```

Оценка вычисляется элементарно. Сначала предполагается, что она равна нулю (ни одного правильного ответа), а затем она уточняется в соответствии с тем, дал ли пользователь правильный ответ по математике, физике и химии. Под конец вызывается процедура **ShowMessage**, которая выдает в маленьком окне диалога заключение экзаменатора (рисунок 9.59).

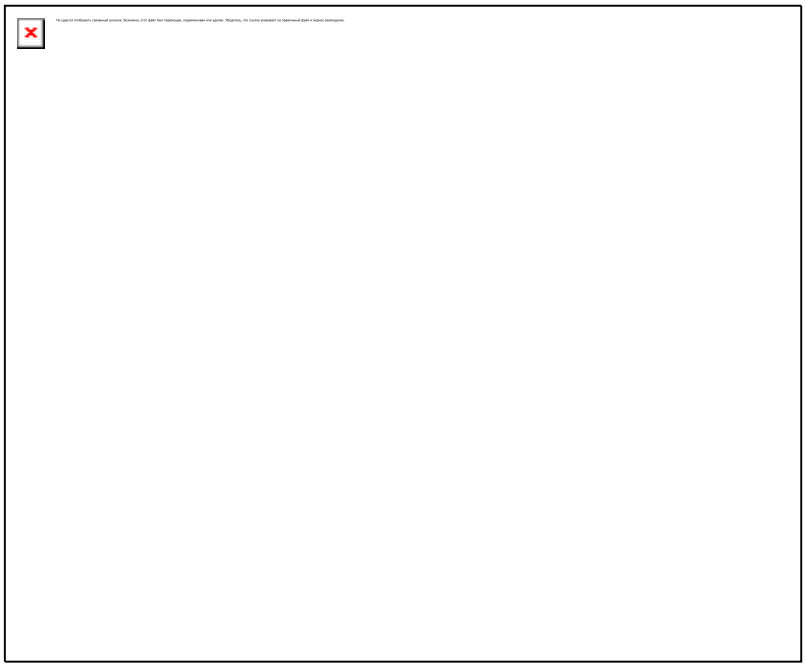


Рисунок 9.59. По щелчку на кнопке Result выставляется оценка

После компиляции и запуска программы предложите своим родственникам или друзьям пройти экзамен. Мы надеемся, что все они получат оценку "отлично".

Поупражняйтесь с компонентом **PageControl**, например, измените размеры закладок (свойства **TabHeight** и **TabWidth**), добавьте больше страниц. Когда закладки перестанут умещаться в одной строке, обнаружится, что их можно прокручивать (рисунок 9.60). Кнопки прокрутки появляются автоматически.

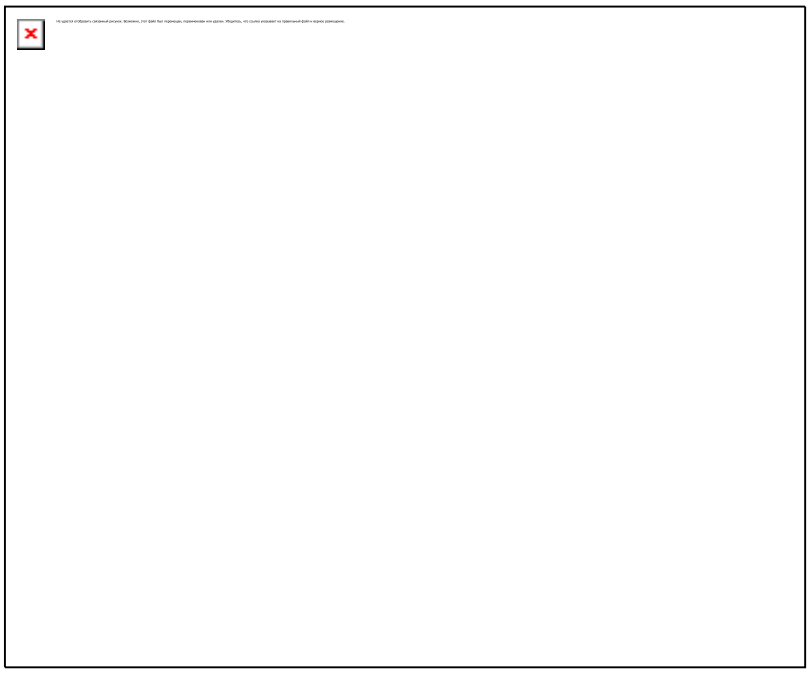


Рисунок 9.60. Вкладки могут прокручиваться с помощью кнопок со стрелками

Если это вам не нравится, закладки можно расположить в несколько рядов, установив свойство **MultiLine** в значение True (рисунок 9.61):

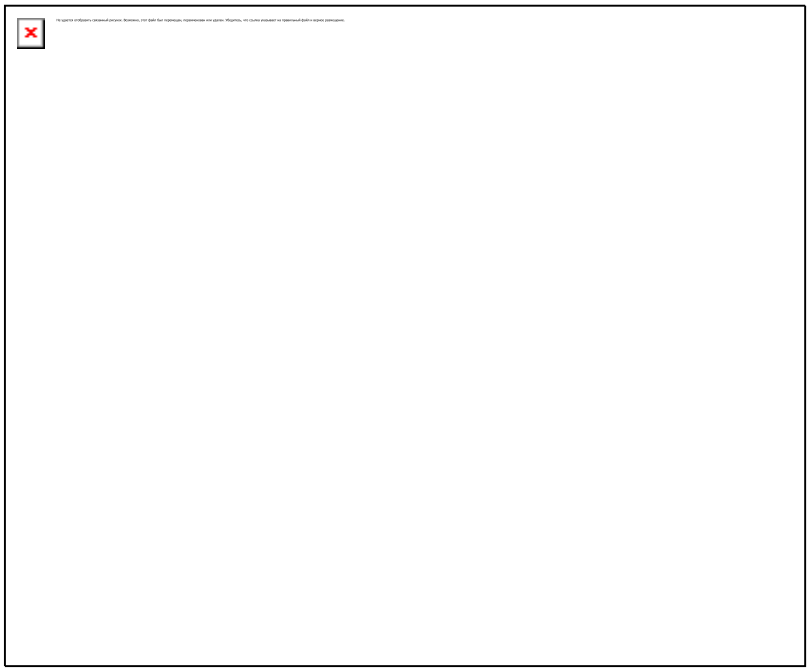


Рисунок 9.61. Вкладки размещены в несколько рядов

А можно ли получить страницы без закладок? Да, для этого в компонентах **TTabSheet** нужно установить свойство **TabVisible** в значение False. Заметьте, это свойство не управляет видимостью вкладки, а влияет лишь на ее заголовок — закладку. Переключение между такими страницами становится вашей заботой и осуществляться программно.

В реальной задаче может потребоваться отследить переключения между страницами. Для этого в компоненте **PageControl** предусмотрены события **OnChanging** и **OnChange**. Первое событие — это запрос на переключение страницы, а второе — уведомление о том, что страница переключилась.

9.5.2. Закладки без страниц

Для создания многостраничных окон диалога иногда используется еще один компонент — **TabControl**, который расположен в палитре компонентов по соседству с компонентом PageControl (рисунок 9.62).

Рисунок 9.62. Компонент TabControl

Характерные свойства компонента **TabControl** описаны в таблице 9.18.

Свойство	Описание
Align	Способ выравнивания компонента в пределах содержащего компонента.
DockSite	Определяет, используется ли компонент TabControl для стыковки других компонентов.
HotTrack	Подсвечивает закладку при наведении на нее указателя мыши.
Images	Список значков, отображаемых на закладках. Каждая закладка получает значок в соответствии со своим порядковым номером.
MultiLine	Располагает закладки в несколько рядов.
MultiSelect	Если равно значению True, то пользователь может выбрать сразу несколько закладок, удерживая клавишу Ctrl. Работает только в том случае, если свойство **Style** содержит значение **tsFlatButtons** или **tsButtons**.
OwnerDraw	Позволяет программно рисовать закладки в обработчике события **OnDrawTab**. Если свойство **OwnerDraw** равно значению False, то закладки имеют стандартный вид и событие **OnDrawTab** не происходит.
RaggedRight	Если равно значению True, то при включенном режиме **MultiLine** закладки не выравниваются на ширину компонента.
ScrollOpposite	Способ организации рядов закладок. Если равно значению False, то все ряды расположены вместе, например вверху. Если равно значению True, неактивные ряды переносятся на другую сторону компонента, например вниз.
Style	Стиль закладок: **tsTabs** — обычные трехмерные закладки, **tsFlatButtons** — плоские закладки, **tsButtons** — закладки в виде кнопок.

Tabs	Закладки в виде списка строк.
TabIndex	Номер выбранной закладки. Если ни одна закладка не выбрана, то значение свойства равно -1.
TabPosition	Местоположение закладок: **tpTop** — сверху, **tpRight** — справа, **tpLeft** — слева, **tpBottom** — снизу.
TabWidth, TabHeight	Ширина и высота закладки. Если эти свойства равны нулю, то ширина и высота каждой закладки подбирается автоматически по ширине и высоте содержащегося на ней текста.
OnChange	Происходит после смены закладки.
OnChanging	Происходит перед сменой закладки.
OnDrawTab	Происходит при рисовании закладки на экране. Требует, чтобы свойство **OwnerDraw** содержало значение True.
OnGetImageIndex	Обработчик этого события должен вернуть номер значка для отображаемой закладки.
OnGetSiteInfo	Происходит, когда у компонента запрашивается место для стыковки.

Таблица 9.18. Важнейшие свойства и события компонента TabControl

Компонент **TabControl** — это фактически одна страница с множеством закладок. Компонент применяется в том случае, если страницы имеют одинаковый вид, а их переключение влечет лишь изменение отображаемых данных. А ведь так произошло с нашими экзаменационными билетами — все страницы содержали по одному единственному компоненту **RadioGroup**.

Каждая вкладка в компоненте **PageControl** потребляет системные ресурсы. Используя компонент **TabControl** вместо компонента **PageControl**, мы значительно снизим потребление оперативной памяти в нашем последнем примере, правда, за счет времени и сил, затраченных на программирование. Давайте не поленимся и переделаем пример с экзаменационными билетами так, чтобы в нем использовался компонент **TabControl**.

Шаг 12. Удалите из формы **ExamForm** компонент **PageControl** и поместите на его место компонент **TabControl**.

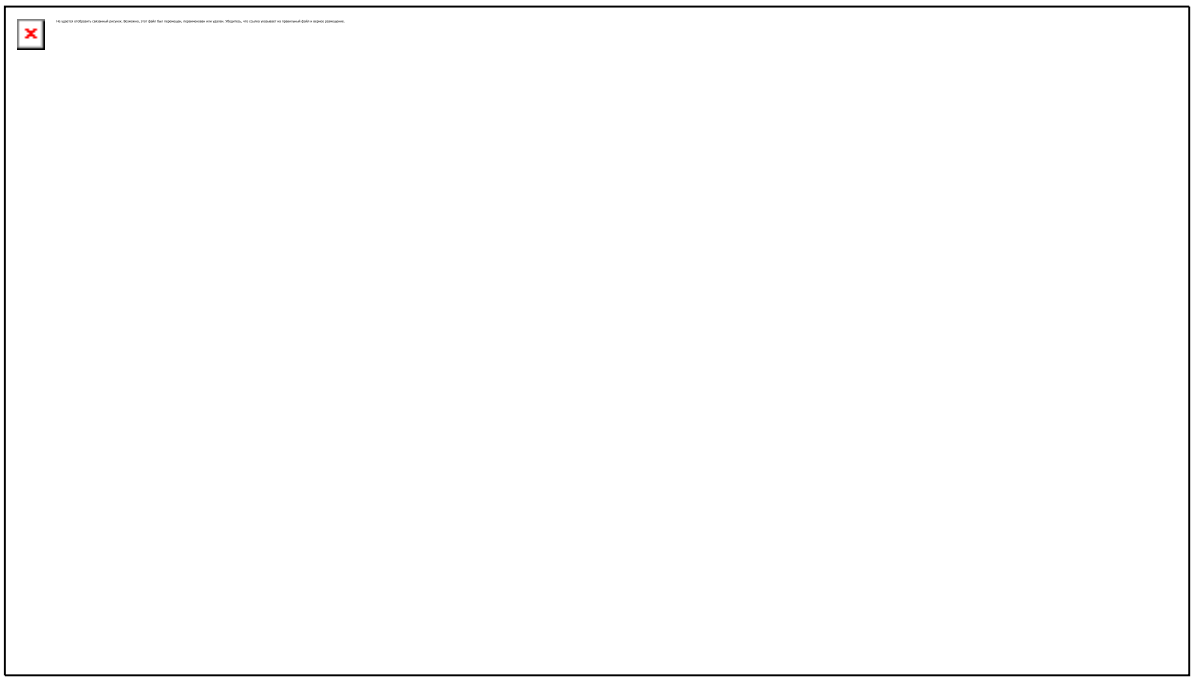


Рисунок 9.63. Компонент TabControl заменил в форме компонент PageControl

Шаг 13. В окне свойств выберите свойство **Tabs** и щелкните кнопку с многоточием. На экране появится редактор строк.

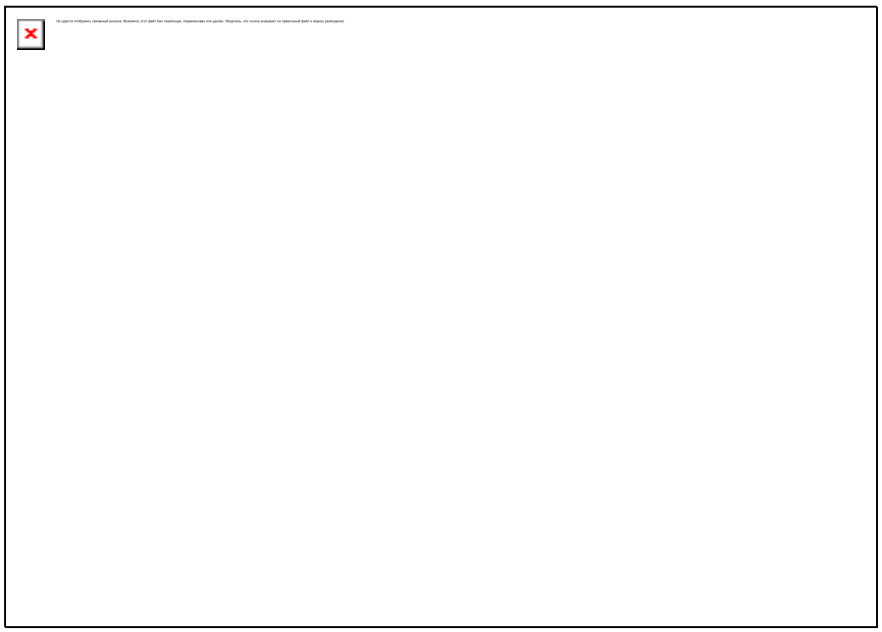


Рисунок 9.64. Список закладок для компонента TabControl

Шаг 14. Введите названия закладок и щелчком кнопки OK закройте окно. Закладки появятся на экране (рисунок 9.65).

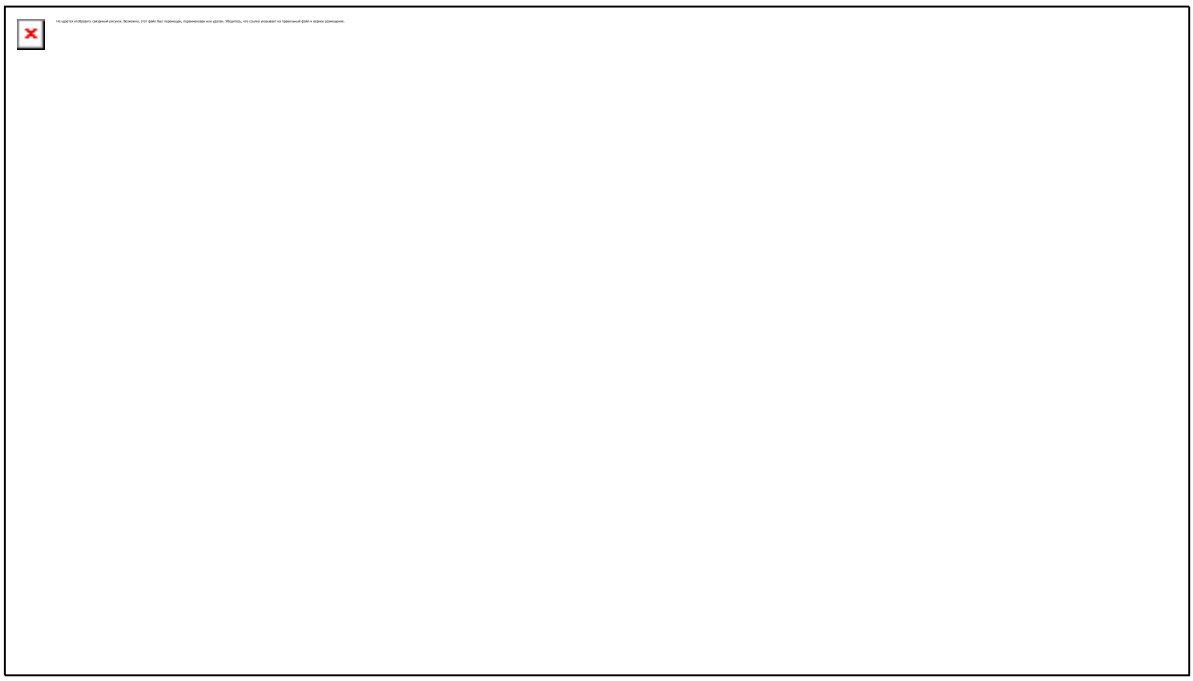


Рисунок 9.65. В компоненте TabControl созданы три закладки

Шаг 15. Теперь внутрь компонента **TabControl** поместите группу взаимоисключающих переключателей и придайте ей соответствующие размеры и положение (рисунок 9.66).

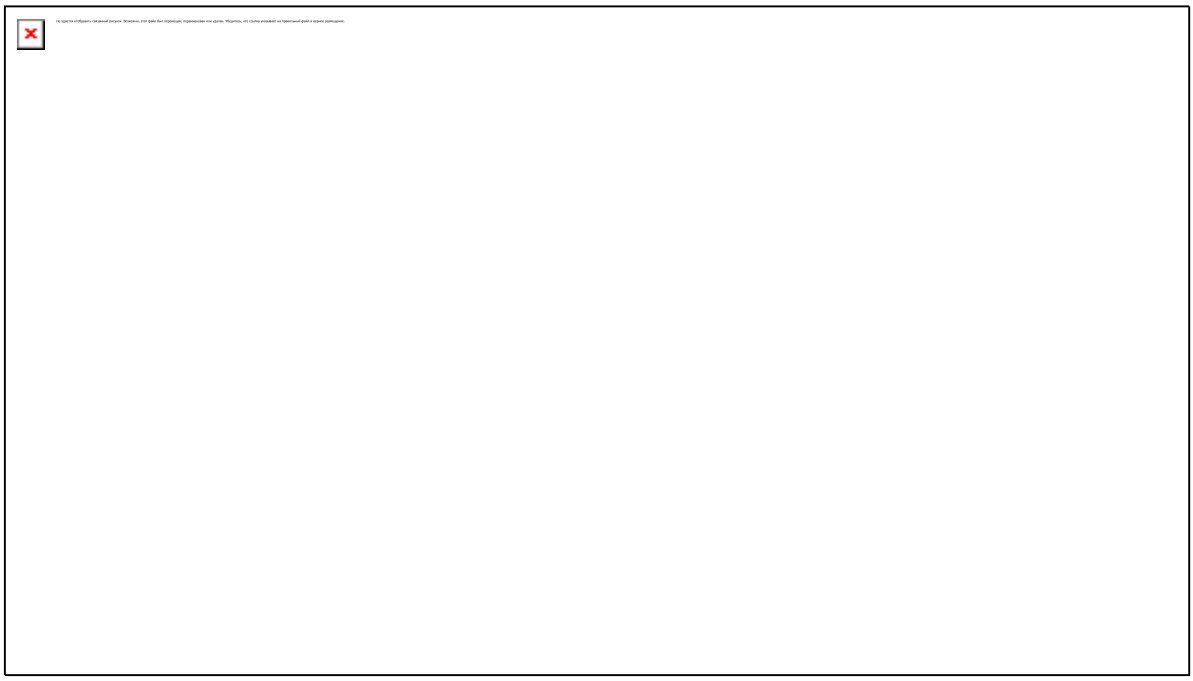


Рисунок 9.66. Группа переключателей RadioGroup1 заготовлена для экзаменационного вопроса с вариантами ответа

Единственная группа взаимоисключающих переключателей будет поочередно играть роль билета по математике, по физике и по химии в зависимости от выбранной закладки.

На этом визуальная часть проектирования закончена. Все остальное придется программировать вручную.

Шаг 16. Сначала нужно позаботится о хранении содержания вопросов и их ответов, поэтому добавьте следующие описания в текст модуля **MainUnit**, поместив их перед всеми обработчиками событий:

```
const
  Questions: array[0..2] of string =
    ('Правильным является выражение',
     'Когда лед в воде тает',
     'Чтобы разбавить кислоту, нужно');
  Answers: array[0..2, 0..2] of string =
    (('sin 50° < cos 50°',
      'sin 50° > cos 50°',
      'sin 50° = cos 50°'),
     ('уровень воды поднимается',
      'уровень воды понижается',
      'уровень воды остается неизменным'),
     ('добавить кислоту в воду',
      'добавить воду в кислоту',
      ''));
  ValidAnswers: array[0..2] of Integer = (1, 2, 0);
```

Шаг 17. Для промежуточного хранения ответов пользователя воспользуемся инициализированной переменной-массивом:

```
var
  UserAnswers: array[0..2] of Integer = (0, 0, 0);
```

Шаг 18. Значения элементов этого массива должны изменяться, когда пользователь выбирает ответ, поэтому создайте компоненту **RadioButton1** обработчик события **OnClick**:

```
procedure TExamForm.RadioGroup1Click(Sender: TObject);
begin
  UserAnswers[TabControl1.TabIndex] := RadioGroup1.ItemIndex;
end;
```

Шаг 19. При смене закладки должен изменяться вопрос экзаменационного билета и возможные варианты ответов. Для этого создайте в компоненте **TabControl1** обработчик события **OnChange**:

```
procedure TExamForm.TabControl1Change(Sender: TObject);
var
  I: Integer;
begin
  // Отобразить новый вопрос
  RadioGroup1.Caption := Questions[TabControl1.TabIndex];
  // Стереть прежние варианты ответа
  RadioGroup1.Items.Clear;
  // Добавить новые варианты ответа в группу переключателей
  for I := 0 to 2 do
    if Length(Answers[TabControl1.TabIndex, I]) > 0 then
      RadioGroup1.Items.Add(Answers[TabControl1.TabIndex, I]);
  // Установить ответ, принимаемый по умолчанию
  RadioGroup1.ItemIndex := UserAnswers[TabControl1.TabIndex];
end;
```

Шаг 20. Все готово? Не совсем. Нужно заполнить компонент **RadioGroup1** данными первого билета при появлении формы на экране. Проще всего это можно сделать, вставив вызов метода **TabControl1Change** в обработчик события создания формы:

```
procedure TExamForm.FormCreate(Sender: TObject);
begin
  TabControl1Change(TabControl1);
end;
```

Шаг 21. Последний штрих — доработка метода выставления оценки:

```
procedure TExamForm.ResultButtonClick(Sender: TObject);
const
  MarkText: array[0..3] of string =
    ('Неудовлетворительно', 'Удовлетворительно', 'Хорошо', 'Отлично');
var
  Mark: Integer;
  I: Integer;
begin
  Mark := 0;
  for I := 0 to 2 do
    if UserAnswers[I] = ValidAnswers[I] then
      Mark := Mark + 1;
  ShowMessage(MarkText[Mark]);
end;
```

А теперь выполните компиляцию и запустите программу. С точки зрения пользователя оно не будет отличаться от программы, созданной ранее с помощью компонента **PageControl**. Думаем, что получив такой практический опыт, вы сможете сами сделать вывод о том, какой из двух компонентов (**PageControl** или **TabControl**) и в каких случаях следует использовать.

9.6. Итоги

Нелегкая тропа создания окон диалога пройдена. Вы в этом деле стали настоящим гуру. Не верите? Да, этот так, ибо вы познали:

- тайны создания монопольных и немонопольных окон диалога,
- технологию работы с разного рода переключателями;
- методы ввода текста и чисел со всеми нюансами, включая шаблоны;
- тайны разработки многостраничных окон диалога с десятками параметров;
- способ хранения параметров в INI-файлах.

Напоследок позволим себе дать два совета: не заставляйте программу болтать лишнее и не загромождайте экран сложными окнами диалога без особой необходимости. Помните, что хороший пользовательский интерфейс должен быть простым.

Часть 2. Программирование на языке C++

Глава 1

1.1. Принципы модульного программирования на языке C++

В языке C++ очень бедные средства модульного программирования, поэтому для достижения модульности программ, следует придерживаться определенных принципов.

Роль программного интерфейса модуля играет h-файл, а cpp-файл — роль реализации этого модуля. Внутрь h-файла включаются h-файлы других модулей, необходимые для компиляции интерфейсной части. Внутрь cpp-файла включаются h-файлы других модулей, необходимые для компиляции cpp- и h-файлов интерфейсной части модуля.

Очевидно, что программисту при включении h-файла другого модуля предоставляется выбор: подключить его в h-файле модуля или в cpp-файле. В данном случае предпочтение следует отдавать части реализации модуля (cpp-файл).

При подключении h-файла следует придерживаться следующей схемы: предположим, что наш модуль называется SysModule и состоит из двух частей: SysModule.h и SysModule.cpp. Рекомендуется следующая схема подключения:

SysModule.h:

```
#include "Config.h" // наш файл конфигурации
                    // подключается первым во всех h-файлах
                    // всех наших проектов
#include "Другой стандартный модуль"
#include "Другой наш модуль"
```

SysModule.cpp:

```
#include "Файл предкомпилированных заголовков"
#include "Еще один наш модуль"
#include "Другой стандартный модуль"
#include "SysModule.h" // подключается последним
```

Поскольку один и тот же h-файл может одновременно включатся в другие h-файлы и несколько раз подключаться при компиляции одного и того же cpp-файла, его следует защищать от повторной компиляции. Для этого в начале любого h-файла вставляются следующие директивы компилятора:

```
#ifndef __SysModule_h__
#define __SysModule_h__
...
#endif //__SysModule_h__
```

Таким образом, в том случае, когда файл подключается несколько раз, скомпилируется он только один раз.

Внимание! Согласно стандарту ISO, любой h- и cpp-файл в C++ должен заканчиваться символом перевода строки.

1.2. Пространства имен

В больших проектах наблюдается серьезная проблема — конфликт идентификаторов. Она решается с помощью пространства имен.

```
namespace Sys
{
  int  var;
  void Proc();
}
```

Внутри пространства имен обращение к определенным внутри переменным и подпрограммам можно осуществлять, используя неполную форму записи:

```
var = 10;
Proc();
```

за пределами – надо использовать полную форму записи:

```
Sys::var = 10;
Sys::Proc();
```

Для того чтобы избежать возможного конфликта идентификаторов, все определения внутри модуля следует помещать в пространство имен. Следует давать небольшой буквенный идентификатор, который будет соответствовать префиксу файла.

Существует возможность открыть пространство имен таким образом, чтобы можно было использовать неполную форму записи. Для этого надо написать строку:

```
using namespace Sys;
```

Но следует отметить, что данная конструкция является причиной многих ошибок, поэтому так писать не стоит.

Существует второй способ открыть пространство имен — это открыть его для конкретного определения:

```
using Sys::Proc();
...
Proc();
...
```

Но рекомендуется использовать Sys::Proc();

Идентификаторы, объявленные вне пространства имен, относятся к так называемому глобальному пространству имен, доступ к которым осуществляется с помощью оператора **::**

```
::Funk();
```

Для того чтобы была возможность закрыть доступ к данным и подпрограммам внутри данного пространства существует пространство имен без имени:

```
namespace
{
  ...
}
```

Пространства имен могут быть вложенными:

```
namespace Sys
{
  namespace Local
  {
    int var;
    ...
  }
  ...
}
Sys::Local::var = 10;
```

Замечание! Когда возникает желание объявить переменный тип данных или подпрограмму внутри пространства имен, а реализовать за пределами (или наоборот), следует поступать так:

SysModul.h:

```
  namespace Sys
  {
    int Proc();
  }
```

SysModul.cpp:

```
  namespace Sys
  {
    int Proc();
    {
       ...
    };
  }
```

1.3. Перегрузка идентификаторов

В C++ можно определить несколько функций с одним и тем же именем. Это явление называется перегрузкой имени — **overloading**.

Эти функции должны отличаться по количеству и типу параметров:

```
void print(int);
void print(const char *);
void print(double);
void print(long);
void print(char);
```

Процесс поиска подходящей функции из множества перегруженных осуществляется путем проверки набора критериев в следующем порядке:

- полное соответствие типов;
- соответствие, достигнутое продвижением скалярных типов данных:

```
bool   - int
char   - int
short  - int
float  - double
double - long double
```

- соответствия, достигнутые путем стандартных преобразований:

```
int    - double
double - int
int    - unsigned int;
```

- соответствия, достигнутые за счет преобразований, определенных пользователем (перегрузка операторов в преобразовании типов);
- соответствия за счет многоточия в объявлении функции.

Если соответствие может быть достигнуто двумя способами на одном и том же уровне, то вызов функции неоднозначен и компилятор выдаст ошибку.

Пример:

```
void TestPrint(char c, int i, short s, float f)
{
   print(c);   // char
   print(i);   // int
   print(s);   // int
   print(f);   // double
   print('a'); // char
   print(49);  // int
   print(0);   // int
   print("a"); // const char*
}
```

Замечание! Перегрузку следует использовать в исключительных случаях. Ее следует использовать по типу параметров, а не по их смыслу или количеству. Так же алгоритм всех перегруженных функций должен быть идентичным (идентичная семантика).

1.4. Переопределенные элементы подпрограмм

Один или несколько последних параметров в заголовке функции могут содержать стандартные значения:

```
void print(int value, int base = 10); // base - система счисления
void print(..., int base)
{
  ...
}
```

При этом функция может вызываться либо print(100, 10) либо print(100).

При вызове данной функции компилятор автоматически подставляет значения для опущенных параметров.

Глава 2

2.1. Классы в C++

Классы в C++ определяются с помощью одного из ключевых слов: **class** или **struct.**

```
class TTextReader
{
  ... // private
};
```

```
struct TTextReader
{
  ... // public
};
```

В C++ доступны атрибуты доступа в классах:

- **public**
- **protected**

- **private**

В Delphi данные секции принято употреблять в порядке: **private...protected...public**. В C++ их можно чередовать.

В работе секций protected и private в Delphi и C++ есть различия:

- В Delphi классы внутри одного модуля могут обращаться к данным и подпрограммам друг друга без ограничений. А действие секций protected и private распространяется только за пределами данного модуля.
- В C++ действие этих секций распространяется на любые два класса. Но установленные ограничения можно обойти с помощью специального оператора **friend**:

```
class TTextReader
{
  friend class TList;
};
```

После этого объект класса TList может обращаться к полям из секций private и protected класса TTextReader.

2.2. Наследование

Наследование класса выполняется следующим образом:

```
class TDelimitedReader: public TTextReader
{
  ...
};
```

При наследовании указываются атрибуты доступа к элементам базового класса (public, protected, private). Для того чтобы понять смысл атрибута доступа к базовому классу, базовый класс следует рассматривать, как поле производного класса.

2.3. Конструкторы и деструкторы

Конструктор создает объект и инициализирует память для него (деструктор — наоборот).

```
class TTextReader
{
  public:
     TTextReader();
    ~TTextReader();
};
```

В Delphi стандартный деструктор является виртуальным. В C++ это определяет программист, если планируется создавать объекты в динамической памяти (по ссылке), деструктор необходимо делать виртуальным:

```
virtual ~TTextReader();
```

Создание объектов:

- по значению (на стеке):

```
TTextReader reader;
```

- по ссылке (в динамической памяти):

```
TTextReader*reader = new TTextReader(); //оператор new служит для размещения
объекта в динамической памяти
```

- с помощью оператора new:

```
TTextReader*reader = new (адрес ) TTextReader;
```

Таким способом объект создается по ссылке по указанному адресу памяти.

Разрушение объектов:

- если объект создан по значению (на стеке), его разрушение выполняется автоматически при выходе переменной за область видимости

```
{
  TTextReader reader;
  ...
} // здесь происходит разрушение объекта reader при автоматическом вызове
деструктора
```

- если объект создан в динамической памяти (по ссылке), он должен быть уничтожен с помощью оператора **delete**:

```
delete reader;
```

При этом сначала происходит вызов деструктора, а затем — освобождение динамической памяти.

Так выглядит динамическое создание и разрушение объектов:

```
new:
  malloc();
  TTextReader();

delete:
  ~TTextReader();
  free();
```

2.4. Стандартные конструкторы

Если программист не определяет в классе конструкторы, то компилятор создает автоматически два конструктора:

- конструктор без параметров
- конструктор копирования

Пример:

```
class TTextReader
{
  public:
    TTextReader(); // конструктор без параметров
    TTextReader(const TTextReader&R); // конструктор копирования
}
```

Внимание! Если программист определил хотя бы один конструктор в класс — компилятор не создаст никаких стандартных конструкторов.

Конструктор без параметров создается для того, чтобы можно было написать:

```
TTextReader R;
```

Конструктор копирования нужен для следующей записи:

```
TTextReader R1 = R2; // означает TTextReader R1(R2);
```

Конструктор копирования вызывается в том случае, когда создаваемый по значению объект создается путем копирования другого уже существующего объекта.

Следует отметить, что запись:

```
TTextReader R1 = R2;
```

и два оператора:

```
TTextReader R1;
R1 = R2;
```

имеют схожий синтаксис с вызовом конструктора копирования, но разную семантику: в первом случае объект создается конструктором копирования, во втором — конструктором без параметров, а затем с помощью оператора ‘=’ выполняется присваивание одного объекта другому (данный вариант требует перегрузки оператора ‘=’ для класса TTextReader)

Работа стандартного конструктора копирования, создаваемого компилятором, заключается в том, чтобы выполнить полное копирование памяти с помощью функции **memcpy**.

2.5. Реализация методов класса

Метод класса может быть реализован по месту или отдельно от класса:

```
class TTextReader  // по месту
{
  public:
     TTextReader() { ... }
    ~TTextReader() { ... }
};

TTextReader::TTextReader() // отдельно от класса
{
  ...
}

TTextReader::~TTextReader()
{
  ...
}
```

Если класс описан в интерфейсной части модуля, его методы рекомендуется реализовывать отдельно от класса в cpp-файле. В том случае, когда некоторый класс надо сделать **inline**-методом, следует писать так:

```
class TTextReader
{
  public:
     TTextReader();
     ~TTextReader();
     int ItemCount();
};

inline int TTextReader::ItemCount()
{
  ...
}
```

2.6. Порядок конструирования и разрушения объектов

По причине автоматичности конструкторов и деструкторов в C++ существует определенный порядок конструирования базовых и агрегированных объектов.

```
class TTextReader
{
  public:
    TTextReader();
    ~TTextReader();
}

class  TDelimitedReader: public TTextReader
{
  public:
    TDelimitedReader();
    ~TDelimitedReader();
}

TDelimitedReader::TDelimitedReader()
{
  ...
}

TDelimitedReader::~TDelimitedReader()
{
  ...
}
```

В конструкторе производного класса конструктор базового класса вызывается автоматически до выполнения первого оператора в теле конструктора.

В деструкторе производного класса деструктор базового класса вызывается автоматически после последнего оператора в теле деструктора.

Если базовый класс содержит конструктор с параметрами или несколько конструкторов, то возникает неопределенность в том, какой конструктор базового класса будет вызван. Эту неопределенность можно устранить следующим образом:

```
TDelimitedReader::TDelimitedReader() : TTextReader(...)
// в скобках записываются параметры для вызова конструктора
{
  ...
}
```

После двоеточия разрешена запись списка операторов, разделенных запятыми. Эти операторы называются списком инициализации.

В C++ поддерживается множественное наследование. В этом случае конструктор базовых классов вызывается автоматически в порядке их упоминания в описании класса. Деструктор же базовых классов вызывается строго в обратном порядке.

Каждый конструктор перед началом своей работы инициализирует указатель **vtable** (в Delphi он называется **VMT**). Конструктор базового класса тоже инициализирует этот указатель. В результате этого объект как бы "рождается", сначала становясь экземпляром базового класса, а затем производного. Деструкторы выполняют противоположную операцию.

В результате этого в конструкторах и деструкторах виртуальные методы работают как невиртуальные.

2.7. Агрегированные объекты

В C++ объекты могут агрегироваться по ссылке и по значению (агрегирование по ссылке похоже на агрегирование в Delphi).

Агрегирование по значению:

```
class TDelimitedReader
{
  public:
    ...
  private:
    std::string m_FileName; // std::string - стандартный класс
                            // для представления строк
};
```

Агрегированные по значению объекты конструируются автоматически в порядке объявления после вызова конструктора базового класса (если не указан другой способ инициализации).

Стандартный способ инициализации можно переопределить до открывающей фигурной скобки конструктора:

```
TTextReader::TDelimitedReader() : TTextReader(), m_FileName("c:/myfile.txt")
{
  ...
}
```

Следует отметить, что данная запись отличается от следующей записи:

```
TDelimitedReader::TDelimitedReader() : TTextReader()
{
  m_FileName = "c:/myfile.txt";
}
```

Во втором случае строка вначале создается пустой, а в теле конструктора переприсваивается.

Объекты, агрегированные по ссылке, нужно создавать вручную с помощью оператора **new**, а удалять — с помощью оператора **delete**:

```
class TDelimitedReader : public TTextReader
{
  ...
  private:
    std::string m_FileName;
    TItems *m_Items;
};

TDelimitedReader::TDelimitedReader() : TTextReader(), m_FileName("c:/myfile.txt")
{
  m_Items = new TItems;
}

TDelimitedReader::~TDelimitedReader()
{
  delete m_Items;
}
```

Правило конструирования агрегированных объектов:

- объекты, агрегированные по значению и константные ссылки инициализируются до тела конструктора.
- объекты, агрегированные по ссылке, инициализируются в теле конструктора.

2.8. Вложенные определения класса

В C++ внутри класса можно определить другой класс:

```
class TTextReader
{
  public:
    class TItems
    {
      ...
    };
  ...
};
```

Эта запись по смыслу соответствует следующей записи:

```
class TTextReader::TItems
{
  ...
};

class TTextReader
{
  friend class TTextReader::TItems; // см. ниже "Друзья класса"
};
```

Таким образом, определения классов и других типов данных внутри класса означает использование имени внешнего класса как пространства имен.

2.9. Друзья класса

Для того, чтобы объекты некоторого класса могли получить доступ в private и protected полям другого класса, используется оператор **friend**, который разрешает доступ ко всем записям и методам класса для того класса, который указан в операторе. Данный оператор используется внутри класса, с его помощью нельзя разрешать доступ к членам класса извне, иначе это нарушит принцип сокрытия данных.

2.10. Статические члены класса

Поля и методы класса могут быть объявлены при помощи слова **static**:

```
class TTextReader
{
  public:
    ...
    static char*ClassName();
    ...
  private:
    static int m_ObjectCount;
    ...
};
```

По смыслу данный код эквивалентен следующему:

```
class TTextReader
{
  friend char*TTextReader::ClassName();
  ...
};

class TTextReader::ClassName()
{
  ...
};

int TTextReader::m_ObjectCount;
```

Если поле объявлено с ключевым словом **static**, то это — обычная глобальная переменная, для которой имя класса используется как пространство имен.

Если метод объявляется с этим словом, то это — обычная глобальная функция, которая является другом класса. Такая функция не имеет псевдо-параметра **this**.

2.11. Множественное наследование

В C++ множественное наследование подразумевает, что у одного класса может быть несколько базовых классов:

```
class TDelimitedReader : public TTextReader, public TStringList
{
  ...
};
```

Объект класса TDelimitedReader содержит все поля и методы базовых классов TTextReader и TStringList. При этом в классе TDelimitedReader можно переопределять виртуальные методы каждого базового класса.

Множественное наследование имеет ряд проблем:

- отсутствие эффективной реализации (неэффективность скрыта от программиста);
- неоднозначность, возникающая из-за того, что в базовых классах могут быть одноименные поля, а также методы с одинаковой сигнатурой;
- повторяющийся базовый класс в иерархии классов.

Неоднозначность при множественном наследовании:

```
class TTextReader
{
  virtual void NextLine();
  ...
};

class TStringList
{
  public:
    virtual void NextLine();
  ...
};

class TDelimitedReader: public TTextReader, public TStringList
{
  ...
};

TDelimitedReader*Reader;
Reader->NextLine(); // Ошибка. Неоднозначность.
```

Неоднозначность возникает потому, что в классе TDelimitedReader существуют две таблицы виртуальных методов и неизвестно, к какой из них надо обращаться за методом NextLine(). Поэтому последний оператор должен быть скорректирован на следующий:

```
Reader->TTextReader::NextLine();
```

или:

```
Reader->TStringList::NextLine();
```

В C++ для классов поддерживается столько таблиц виртуальных методов, сколько у него базовых классов. При перекрытии общего виртуального метода, существующего в нескольких базовых классах, происходит замещение адреса во всех таблицах виртуальных методов.

Перегрузка функций по типам аргументов не приводит к разрешению неоднозначности.

Если функция NextLine() была объявлена с различной сигнатурой в различных классах, то неоднозначность тоже остается.

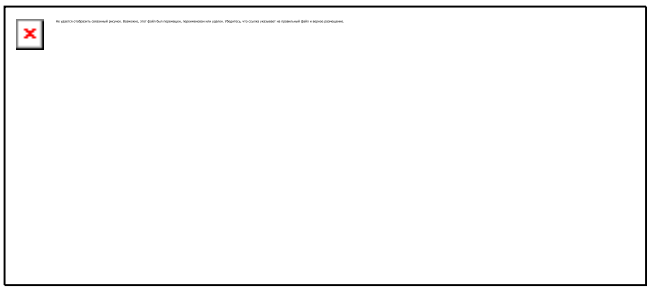
В некоторых случаях наличие в базовых классах функций с одинаковыми именами (но различным количеством параметров или различными типами параметров) является

преднамеренным решением. Чтобы в производном классе открыть нужную функцию нужного базового класса, применяется оператор **using**:

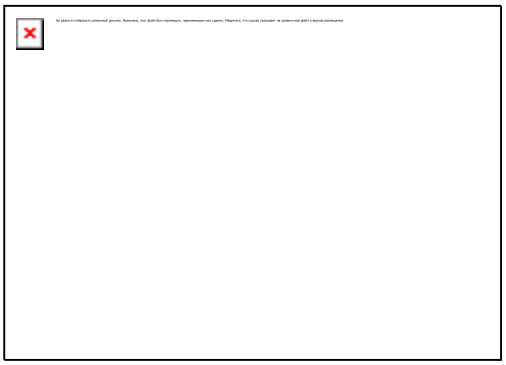
```
class TTextReader
{
  public:
    virtual void NextLine();
  ...
};

class TStringList
{
  public:
    virtual void NextLine(int);
  ...
};

class TDelimitedReader : public TTextReader, public TStringList
{
  public:
    using TStringList::NextLine;
    virtual void NextLine(int);
  ...
};
```

2.12. Проблема повторяющихся базовых классов

Классы TStringList и TTextReader в нашем примере могут иметь одинаковый базовый класс, например TObject. Это приводит к следующей иерархии классов:



В этом случае объект класса TDelimitedReader имеет две копии полей класса TObject.

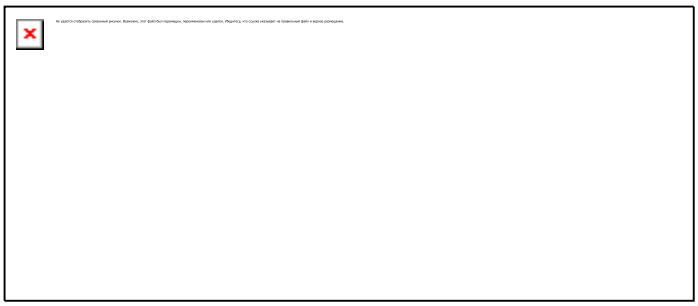


Из-за дублирования полей возникает неоднозначность при обращении к полю класса TObject из метода класса TDelimitedReader. Проблема решается с помощью уточненного имени:

```
TTextReader::m_Field;
TStringList::m_Field;
```

Однако главная проблема состоит в том, что одна сущность дублируется внутри базового класса.

На практике обычно требуется получать следующий результат:

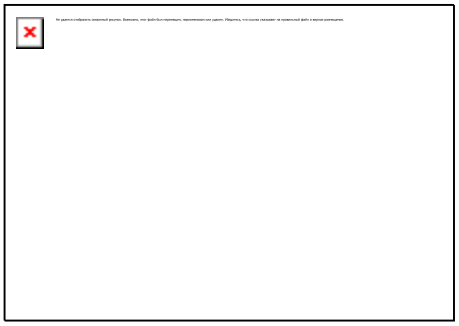


Такой результат достигается при применении виртуальных базовых классов:

```
class TDelimitedReader: public TTextReader, public TStringList
{
  ...
};

class TTextReader: public TObject
{
  ...
};

class TStringList: virtual public TObject
{
  ...
};
```

Обычное наследование соответствует агрегации всех полей базового класса. Виртуальное наследование соответствует агрегации ссылки на поля базового класса.

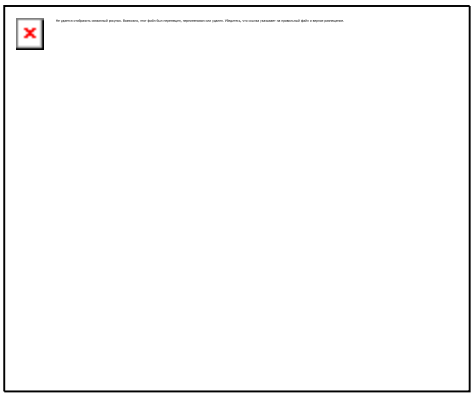
В данном случае структура полей в памяти будет следующей:



Если же при объявлении класса TTextReader мы запишем следующее:

```
class TTextReader: virtual public TObject
{
  ...
};
```

то структура полей будет такой:



Таким образом, множественное наследование таит следующую проблему: заранее неизвестно от каких классов программист захочет унаследовать свой класс. Однако при создании класса использовать виртуальное наследование неэффективно, если наследуются поля, так как доступ к полям всегда будет осуществляться через дополнительный указатель.

Вывод: одинарное наследование в стиле Java, C++, Delphi допустимо только от классов, множественное — от интерфейсов. Иначе можно осуществлять множественное наследование лишь от классов, в которых отсутствуют поля.

Глава 3

3.1. Виртуальные методы

В C++ виртуальные методы определяются при помощи ключевого слова **virtual**:

```
class TTextReader: virtual public TObject
{
  ...
};
```

При перекрытии виртуального метода ключевое слово **virtual** можно записать, а можно и опустить. Синтаксис перекрытия виртуальных методов не предусматривает такие проблемы, как версионность и рефакторинг кода (упрощение программного кода с сохранением функциональности).

Если метод виртуальный следует всегда писать ключевое слово **virtual**.

3.2. Абстрактные классы

В C++ абстрактный класс объявляется следующим образом:

```
class TTextReader
{
  protected:
    virtual void NextLine() = 0;
  ...
};
```

Такой метод называется абстрактным и класс, содержащий данный метод, тоже называется абстрактным.

Виртуальные методы следует объявлять в секции **protected**.

3.3. Подстановочные функции

В C существует способ оптимизировать вызов функций с помощью макросов (#**define**).

В C++ использование этого ключевого слово должно быть минимизировано, так как макросы обрабатываются препроцессором и в большинстве компиляторов символические имена макросов не видны в отладчике. Кроме того, макросы сильно затрудняют отладку программы, так как отладить макрос невозможно.

Вместо макросов в C++ используются подстановочные функции. Они определяются с помощью ключевого слова **inline**:

```
inline int Min(int x, int y)
{
  if (x < y)
    return x;
  else
    return y;
};
```

Тело подстановочной функции в большинстве случаев подставляется в код программы вместо ее вызова. Она должна быть целиком определена в h-файле. Типичной ошибкой

программиста является размещение этой функции в cpp-файле и вынос ее прототипа в h-файл.

Если записана директива **inline**, это еще не означает, что компилятор подставляет тело функции в место ее вызова — он сам решает, что будет более удобным в данном случае.

3.4. Операторы преобразования типа

Существует четыре оператора преобразования типа в C++:

```
reinterpret_cast<тип>(переменная)
static_cast<тип>(переменная)
const_cast<тип>(переменная)
dynamic_cast<тип>(переменная)
```

Первый оператор (**reinterpret_cast**) позволяет отключить контроль типов данных на уровне компилятора, с помощью него любой указатель может быть интерпретирован, как любой другой указатель, а также любая память или переменная может быть интерпретирована иначе.

В программах этот оператор преобразования типа использовать не следует, так как он нарушает переносимость программ. Его наличие свидетельствует о том, что программа является кросс-платформенной. Пример использования:

```
int i;
char *p;
p = reinterpret_cast<char>(&i);
```

Второй оператор (**static_cast**) используется вместо преобразования тип(переменная), (тип)переменная и (тип)(переменная) при работе с классами, структурами и указателями на них.

Оператор static_cast был задуман по причине того, что в C++ выражение тип(переменная) может оказаться вызовом конструктора. Если в программе требуется преобразовать тип, а не вызвать конструктор типа, используется данный оператор. Кроме того, оператор (тип)переменная или (тип)(переменная) может в некоторых случаях оказаться преобразованием reinterpret_cast<тип>(переменная), а при разработке кросс-платформенных программ оператор reinterpret_cast всегда содержит потенциальную опасность неправильной работы программы на другой платформе. Поэтому вместо операторов тип(переменная), (тип)переменная и (тип)(переменная) следует использовать операторы reinterpret_cast и static_cast, которые убирают не явность из преобразования.

Так как оператор static_cast является громоздким, то для простых типов данных допустимо использование форм: (тип)переменная и (тип)(переменная). Форма тип(переменная) не должна использоваться для преобразования типа.

Третий оператор (**const_cast**) используется для приведения не константных указателей к константным и наоборот:

```
void f2(char *s);
void f1(const char *s)
{
  ...
  f1(const char *s);
  ...
  f2(const_cast<char>(s))
  ...
};
```

При объявлении переменных и параметров функций в описании типа может быть указано ключевое слово **const**.

Объявление f(const char *s); означает, что символы, адресуемые указателем s, изменять нельзя.

Объявление f(char const *s) означает, что указатель s изменять нельзя.

Так же можно сделать объявление: f(const char const *s), которое будет означать, что ни указатель, ни переменную изменять нельзя.

Если в программе объект объявлен с помощью модификатора **const**, то у него можно вызывать лишь те методы, которые объявлены с этим же модификатором:

```
class TTextReader
{
  public:
    int ItemCount() const;
  ...
};
```

Наличие константных объектов порождает проблему — огромная избыточность программного кода. Заранее программист не знает, будет ли пользователь (другой программист) его класса создавать константные объекты. Вследствие того, что это не исключено, программист начинает записывать слово **const** в объявление всех методов, в которых его можно записать. Многие методы являются виртуальными или вызывают виртуальные методы. Случается так, что в производных классах виртуальные методы, вызванные константными методами, модифицируют поля объектов (это требуется по условию задачи). Это приводит к логической проблеме, которая решается либо за счет применения оператора **const_cast** к указателю **this** в производных классах, либо за счет объявления полей в производных классах с модификатором **mutable** (записывается при описании полей класса в том случае, если они должны модифицироваться константными методами). Пример:

```
mutable int m_RefCount;
```

Так же решить проблему можно при помощи перегрузки метода класса без модификатора **const**:

```
class TTextReader
{
  public:
    int ItemCount() const;
    int ItemCount();
    int ItemCount() const volatile;
    int ItemCount() volatile;
  ...
};
```

Варианты объявления:

```
volatile TTextReader r;
const volatile TTextReader r;
const TTextReader r;
TTextReader r;
```

Ключевое слово **volatile** запрещает кэшировать значение переменной. Если в программе происходит считывание значения переменной, объявленной с этим ключевым словом, то значение считывается из памяти, а не из регистров, а запись всегда производится в память, в которой размещается данная переменная.

Если ключевое слово **volatile** не указано, то оптимизатор C++ имеет право выполнять регистровые операции (оптимизации) при чтении и записи переменных, а так же размещать их в регистрах.

Четвертый оператор (**dynamic_cast**) соответствует оператору **as** в Delphi. Для работы этого оператора нужно в опциях компилятора включить опцию RTTI. Если это выполнено, то оператор dynamic_cast работает, как static_cast.

Оператор dynamic_cast работает по-разному в зависимости от того, применяется он к ссылке на объект (&) или указателю на объект (*). Если оператор применяется к ссылке на объект,

то преобразование не может быть выполнено и возникает исключительная ситуация. Если он применяется к указателю на объект и преобразование не может быть выполнено, оператор возвращает NULL.

3.5. Размещающий оператор new

Обычно оператор **new** размещает объекты в динамической памяти (**heap**).

```
TDelimitedReader *R = new TDelimitedReader();
```

В данном случае оператор new имеет следующий вид:

```
void *operator new(size_t size);
```

Существует вид оператора new, который позволяет расположить объект по заданному адресу:

```
TDelimitedReader *R = new (Buffer)TDelimitedReader();
```

В этом же случае оператор new имеет следующий вид:

```
void *operator new(size_t size, void *p)
{
  return p;
}
```

Внимание! Комбинировать различные способы выделения и освобождения памяти не рекомендуется.

3.6. Ссылки

Ссылка является альтернативным именем объекта и объявляется следующим образом:

```
int i;
int &r = i;
```

Использование ссылки r эквивалентно использованию переменной i.

Основное применение ссылок — передача параметров в функцию и возврат значения.

В случае, когда ссылка используется в качестве параметра функции, она объявляется неинициализированной:

```
void f(int &i);
```

Во всех остальных случаях ссылка должна инициализироваться при объявлении, как показано ранее.

Если ссылка является полем класса, она должна инициализироваться в конструкторе класса в списке инициализации до тела конструктора.

При использовании в качестве параметров функций ссылки соответствуют **var**-параметрам в языке Delphi:

```
procedure P(var I: Integer)
begin
  ...
end;
```

Константные ссылки соответствуют **const**-параметрам в языке Delphi:

```
procedure P(const I: Integer)
begin
  ...
end;
```

При передаче ссылочного параметра в стек заносится адрес переменной, а не ее копия.

Ссылку следует рассматривать, как псевдоним переменной, которой она инициализирована. Ссылки отличаются от указателей тем, что позволяют компилятору лучше оптимизировать программный код.

Для возврата значений из процедур (через параметры) предпочтение следует отдавать указателям, а не ссылкам. Ссылки следует использовать лишь в тех случаях, когда известно, что возвращаемый объект должен создаваться не в динамической памяти, а на стеке, то есть ссылки применяют при возврате value-type-объектов.

При работе со ссылками существует типовая ошибка — возврат через ссылку переменной, созданной на стеке. Пример ошибочной записи:

```
void f(int *p)
{
  int i;
  p = &i;
}
```

Следующая запись тоже будет ошибочной:

```
void f(int &r)
{
  int i;
  r = i;
}
```

Следующий пример тоже ошибочен, так как нельзя возвращать адрес объекта, созданного на стеке:

```
std::string& GetName(Object* Obj)
{
  const char* str = Obj->GetName();
  return std::string(str);
}
```

3.7. Обработка исключительных ситуаций

В C++ отсутствует аналог блока **try…finally…end**.

На платформе Windows благодаря структурной обработке ОС существуют следующий блок:

```
__try
{
  ...
}
__finally
{
  ...
}
```

Но следует отметить, что для переносимых программ он не подходит.

В C++ существует аналог блока **try…except…end**:

```
try
{
  ...
}
catch(std::ios_base::failure)
{
  ...
}
catch(std::exception)
{
  ...
}
catch(...)
{
  ...
}
```

Распознавание исключительных ситуаций происходит последовательно блоками **catch**, поэтому их последовательность должна быть от частного к общему.

Последний блок **catch** в примере выше ловит любую исключительную ситуацию.

Создание исключительных ситуаций выполняется с помощью оператора **throw** (аналог **raise** в Delphi):

```
throw std::exception("Ошибка");
```

Внутри блока **catch** оператор **throw** возобновляет исключительную ситуацию, как и **raise** в Delphi.

При создании исключительной ситуации при помощи оператора **throw** объект, описывающий исключительную ситуацию, может быть создан в динамической памяти:

```
throw new std::exception("Ошибка");
```

Если применяется такой способ создания исключительной ситуации, ее уничтожение должно происходить следующим образом:

```
try
{
  ...
  throw new std::exception("Ошибка");
}
catch(std::exception *e)
{
  delete e;
}
catch(...)
{
  ...
}
```

Если же записать так:

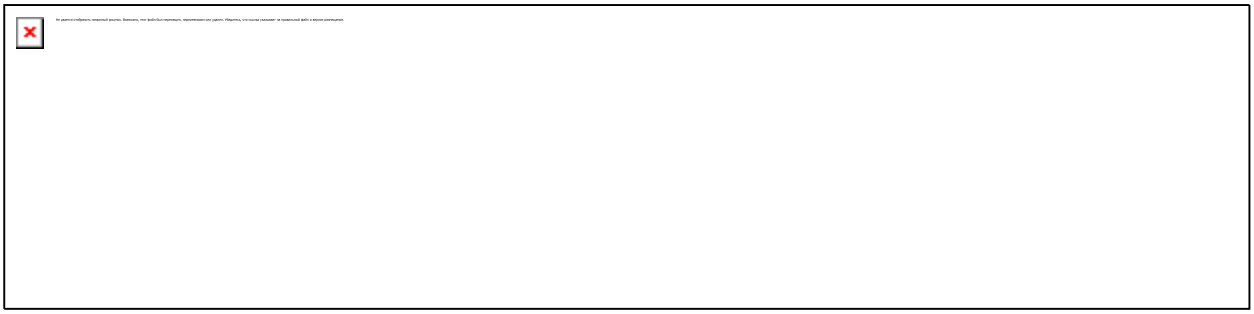
```
try
{
  ...
  throw new std::exception("Ошибка");
}
catch(...)
{
  ...
}
```

то возникнет утечка ресурсов из-за того, что объект std::exception, созданный в динамической памяти, не будет освобожден.

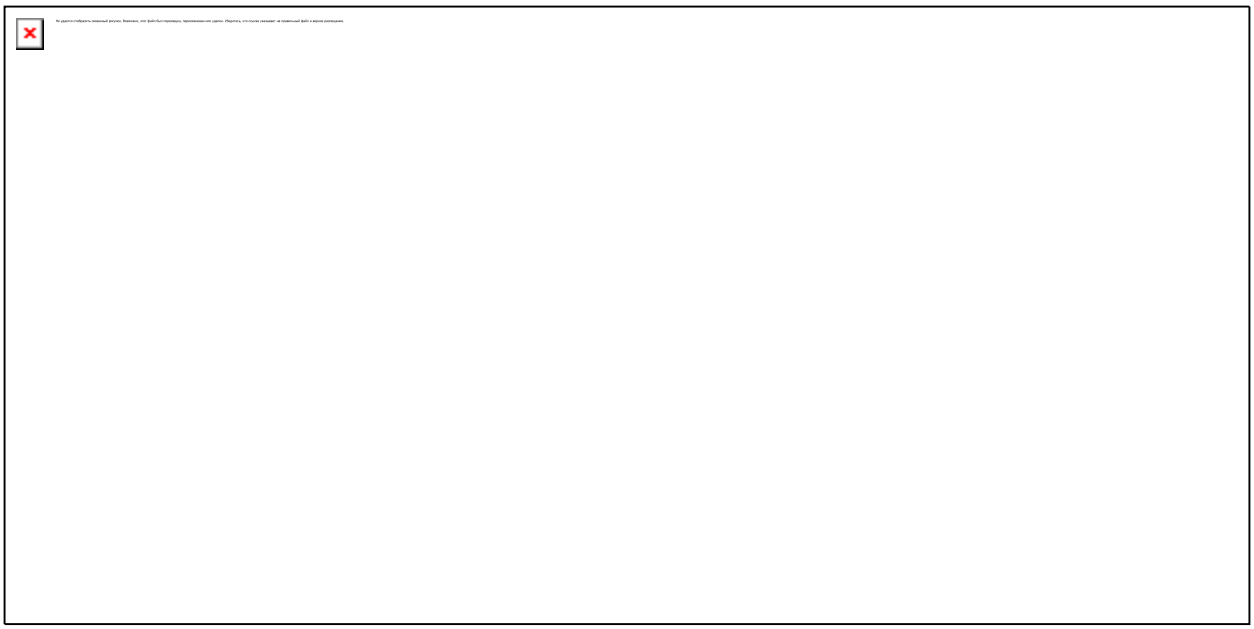
В C++ отсутствует общий базовый класс для исключительных ситуаций, поэтому на верхнем уровне работы программы нужно отлавливать все возможные базовые классы

исключительных ситуаций. Это является препятствием на пути построения расширяемых систем.

Program



Program



3.8. Защита ресурсов от утечки

Поскольку в C+ отсутствует блок **try…finally**, его приходится эмулировать.

```
Object *p = new Object();
try
{
  ...
}
catch(...)
{
  delete p;
  throw;
}
delete p;
```

Данный код эквивалентен следующему:

```
Object *p = new Object();
__try
{
  ...
}
__finally
{
  delete p;
}
```

за исключение того, что второй пример не является переносимым.

Согласно стандарту C++ в деструкторах и операторах delete не должно быть исключительных ситуаций, если же исключительная ситуация произошла, то поведение программы не определено.

Если исключительная ситуация происходит в конструкторе объекта, объект считается не созданным и деструктор для этого объекта не вызывается, но память, выделенная для объекта, освобождается.

Если внутри объекта агрегированы другие объекты, то вызываются деструкторы лишь для тех объектов, которые были полностью созданы к моменту возникновения исключительной ситуации.

Если объект создается в динамической памяти и освобождается в той же самой процедуре, то для защиты от утечки ресурсов можно применять оболочечные объекты — **wrapper** (содержит указатель на динамический объект, который уничтожается в деструкторе оболочечного объекта). Оболочечный элемент создается на стеке, поэтому его деструктор вызывается автоматически, гарантируя тем самым уничтожение агрегированного динамического объекта.

Такие оболочечные объекты в библиотеках программирования называются **AutoPtr**, **SafePtr** и т.д.

```
class AutoPtr
{
  public:
    AutoPtr(int *arr);
    ~AutoPtr();
  private:
    int *m_arr;
};

AutoPtr::AutoPtr(int *arr)
{
  m_arr = arr;
}

AutoPtr::~AutoPtr()
{
  delete[] m_arr;
}

void Proc()
{
  int *arr = new int[100];
  AutoPtr autoc(arr);
  ...
}
```

3.9. Перегрузка операторов

Перегрузка операторов позволяет заменить смысл стандартных операторов (+, –, = и др.) для пользовательских типов данных.

В C++ разрешена перегрузка операторов, выраженных в виде символов, а также операторов:

```
new          delete
new[]        delete[]
```

Запрещена перегрузка следующих операторов:

```
::  .  .*  ?:
```

Перегрузка операторов таит угрозу: она резко усложняет понимание программы, поэтому ей пользоваться нужно очень осторожно. Для стандартных типов данных перегрузка запрещена, хотя бы один из операторов должен принадлежать пользовательскому типу данных.

Бинарные операторы

Бинарный оператор можно определить либо в виде нестатической функции членов с одним аргументом, либо в виде статической функции с двумя аргументами.

Для любого бинарного оператора @ выражение aa@bb интерпретируется как aa.operator@(bb) или operator@(aa, bb). Если определены оба варианта, то применяется механизм разрешения перегрузки функций.

Пример:

```
class X
{
  public:
    void operator +(int);
  X(int);
};

void operator +(X, X);
void operator +(X, double);

void Proc(X, a)
{
  a + 1;   // a.operator +(1)
  1 + a;   // ::operator +(X(1),a)
  a + 1.0; // ::operator +(a, 1.0)
}
```

Унарные операторы

Унарные операторы бывают префиксными и постфиксными.

Унарный оператор можно определить в виде метода класса без аргументов и в виде функции с одним аргументом. Аргумент функции — объект некоторого класса.

Для любого префиксного унарного оператора выражение @aa интерпретируется как:

```
aa.operator @();
operator @(aa);
```

Для любого постфиксного унарного оператора выражение aa@ интерпретируется, как:

```
aa.operator @(int);
operator @(aa, int);
```

Запрещено перегружать операторы, которые нарушают грамматику языка.

Существует три оператора, которые следует определить внутри класса в виде методов:

```
operator =
operator []
operator ->
```

Это гарантирует, что в левой части оператора будет записан **lvalue** (присваиваемое значение).

Операторы преобразования

В C++ существуют операторы преобразования типов. Это является хорошим способом использования конструктора для преобразования типа. Конструктор не может выполнять следующие преобразования:

- неявное преобразование из типа, определяемого пользователем в базовый тип. Это связано с тем, что базовые типы не являются классами.
- преобразование из нового класса в ранее определенный класс, не модифицируя объявление ранее определенного класса.

Оператор преобразования типа:

```
X::operator T() // определяет преобразования класса X в тип данных T (T — класс
или базовый тип)
Пример:
class Number
{
  public:
    operator int();
  operator Complex();
};

Number::operator int()
{
  ...
}

Number::operator Complex()
{
  ...
}
```

Оператор преобразования типа возвращает значение типа T, однако в сигнатуре оператора он не указывается. В этом смысле операторы преобразования типа похожи на конструкторы.

Хотя конструктор не может использоваться для неявного преобразования типа из класса в базовый тип, он может использоваться для неявного преобразования типа из класса в класс.

В программе следует избегать любых неявных преобразований типов, так как это приводит к ошибкам.

С помощью ключевого слова **explicit** можно запретить неявное преобразования типа к конструкторам.

Пример:

```
class File
{
  public:
  explicit File(const char *name); // одновременно не могут быть определены, надо
выбирать один из них
  explicit File(const char *name, int mode = FILE_READ);
};
```

Слово **explicit** записывается лишь для тех конструкторов, которые могут вызываться лишь с одним параметром. Если же они вызываются с несколькими параметрами, то неявное преобразование типов невозможно.

Если объект создается на стеке, то неявное преобразование типа часто бывает необходимо, тогда слово **explicit** писать надо. Так же его надо писать, когда объект создается динамически.

При перегрузке операторов нужно быть внимательным к типу возвращаемого значения: для некоторых операторов объект возвращается по ссылке, для некоторых — по значению:

```
X operator; // по значению
X &operator; // по ссылке
```

Для некоторых операторов возможен и первый и второй вариант перегрузки, поэтому программисту следует определяться с вариантом перегрузки.

Замечание по поводу преобразования типа в тернарном операторе (c ? x : y).

```
class A { ... };
class B: public A { ... };
class C: public A { ... };
```

Запись

```
A* p = cond ? new B : new C;
```

вызовет ошибку компилятора, поскольку между типами выражений "new B" и "new C" выбирается общий тип, а такого нет. Ошибку следует устранить, выполнив преобразование "new B" или "new C" к общему типу, например:

```
A* p = cond ? (A*)new B : new C;
```

или

```
A* p = cond ? new B : (A*)new C;
```

или

```
A* p = cond ? (A*)new B : (A*)new C; // самый лучший вариант
```

Глава 4

4.1. Шаблоны

Шаблоны обеспечивают непосредственную поддержку обобщенного программирования. Они представляют собой параметризованные классы и параметризованные имена функций.

Шаблон определяется с помощью ключевого слова **template**:

```
template <class T>
class basic_string
{
  public:
    basic_string();
    basic_string(const T*);
    basic_string(const basic_string&);
  private:
    T*str;
};

typedef basic_string<char> string;
typedef basic_string<unsigned int> wstring;
```

Вместо слова **typename** часто записывают слово **class**, но параметром шаблона может быть любой тип данных. С точки зрения компилятора, шаблон является макроподстановкой, поэтому шаблонные классы определяются целиком в заголовках файлов (в h-файле, а не в cpp-файле).

Методы шаблона описываются следующим образом:

```
template <class T>
basic_string<T>::basic_string(const *T)
{
  ...
}
```

4.2. Шаблоны функций

Допускается применение шаблонов с целью реализации абстрактных алгоритмов, то есть шаблонов функций.

```
template <class T>
void sort(vector<T>& v);
```

При вызове шаблонных функций компилятор подставляет тип данных и создает новый вариант функции. Если один и тот же тип данных используется несколько раз, то на все типы данных используется несколько раз, то на все типы данных создается один шаблон функции.

При использовании шаблонов существует три больших недостатка:

- шаблоны невозможно отлаживать.

Разработка шаблонов обычно ведется так:

26. разрабатывается класс или функция конкретного типа данных;
27. этот класс или функция параметризуются, то есть создается шаблон.

- существенно замедляется время компиляции. В больших проектах оно может доходить до 30-60 минут.
- очень быстро растут размеры объектных модулей и библиотек на диске.

Компиляция относительно большого проекта в отладочном режиме может потребовать до 10 ГБ.

4.3. Перегрузка шаблонов функций

Можно объявить несколько шаблонов функций с одним и тем же именем, а так же можно объявить комбинацию шаблонов и обычных функций с одинаковым именем:

```
template <class T> T sqrt(T x)

template <class T>
  complex<T> sqrt(complex<T> x);

template <class T>
  double sqrt(double x);

void Proc(complex<double> z)
{
  sqrt(2);   // sqrt<int>(int)
  sqrt(2.0); // sqrt(double)
  sqrt(z);   // sqrt<double>(comlex<double>)
}
```

Шаблонные функции могут вызываться с явным указанием параметра шаблона:

```
sqrt<int>(2);
```

или без него:

```
sqrt(2);
```

В этом случае применяется механизм разрешения перегрузки:

- ищется набор специализации шаблонов функций, которые примут участие в разрешении перегрузки;
- если могут быть вызваны два шаблона функций и один из них более специализирован, то только он и будет рассматриваться;
- разрешается перегрузка для этого набора функций и любых обычных функций. Если аргументы функции шаблона были определены путем выведения по фактическим аргументам шаблона, к ним нельзя применять “продвижение” типа, стандартные и определяемые пользователем преобразования.
- если и обычная функция, и специализация подходят одинаково хорошо, предпочтение отдается обычной функции;
- если ни одного соответствия не найдено, или существует несколько одинаково хорошо подходящих вариантов, то выдается ошибка.

В параметрах шаблонов допустимы стандартные значения, принимаемые по умолчанию:

```
class Allocator
{
  ... // malloc, free, calloc
};

template <class T, class A = Allocator>
class basic_string
{
  ...
};

basic_string<char, MyAllocator> mystr;
basic_string<char> commonstr;
```

4.4. Специализации шаблонов

Как правило, шаблон представляет единственное определение, которое применяется к различным аргументам шаблона. Это не всегда удобно, иногда существует необходимость использовать различные реализации в зависимости от типа.

Например, надо для всех указателей использовать особую реализацию шаблона, а для всех базовых типов данных — обычную реализацию. Это делается с помощью специализации шаблона:

```
template <class T>
class vector
{
  ...
};

void Proc()
{
  vector<int> vi;
  vector<void*> vp;
  ...
}
```

```
template <>
class vector<void*>
{
   ...
};
```

Также может применяться частичная специализация шаблонов:

```
template <class T>
class vector<T*> : private vector<void*>
{
  ...
};

template <>
class vector<void*>
{
  ...
};
```

Специализация шаблонов, как правило, используется для сокращения объема программного кода. Если шаблон создается для указателей на какие-то объекты и класс объекта не так важен, то при использовании обычных шаблонов без специализации возникает многократное дублирование одного и того же кода. Это связано с тем, что в машинных кодах работа со всеми указателями строится одинаково. Чтобы избежать дублирования кода в случае использования указателей следует создавать специализации шаблонов.

4.5. Использование шаблонов при создании новых типов данных

На основе шаблонов можно создать новый типа данных, использование которого позволит упростить программный код и избежать возможных ошибок.

Существует два способа создания нового типа данных на базе шаблона:

- можно воспользоваться оператором **typedef**:

```
typedef basic_string<char> string;
```

Далее типом данных string можно пользоваться, забыв, что это шаблон.

На самом деле для компилятора оператор **typedef** является как бы макроподстановкой, то есть при использовании типа string компилятор всегда подставляет значение типа basic_string<char>.

Если шаблонные классы агрегируют другие шаблонные классы и порождаются от шаблонных классов, то длина программного идентификатора внутри объектного модуля может быть очень велика. Порой она превышает 4 КБ. Некоторые компиляторы на платформе Unix имеют ограничения на длину программного идентификатора внутри объектного модуля. Поэтому при использовании шаблонов (в особенности библиотеки **stl**) нередко возникает проблема с компиляцией.

Например, выражение:

```
string::iterator;
```

на самом деле в объектном модуле выглядит так:

```
basic_string<char>::iterator;
```

Оператор **typedef** по существу отличается от раздела type в Delphi. В C++ упрощение записи — макроподстановка.

- наследование

```
class string : public basic_string<char>
{
  ...
};
```

При наследовании создается именно новый тип данных, поэтому в объектный модуль в данном случае помещается идентификатор следующего вида:

```
string::iterator;
```

Таким образом, использование наследования позволяет решить проблему длины идентификаторов при работе с шаблонами.

4.6. Стандартная библиотека шаблонов

Перечислим, что содержится в стандартной библиотеке шаблонов:

- **Классы и шаблоны для организации потоков ввода/вывода**

В языке C++ вместо функций **printf** и **scanf** предлагается использовать объекты потоковых классов:

```
std::cout;
std::cin;
```

Вывод осуществляется с помощью оператора сдвига:

```
std::cout << "Hello!";
int n;
std::cin >> n;
```

Чтобы перевести строку надо сделать следующую запись:

```
std::cout << "Hello!" << std::endl; // или "\n"
```

Объекты **cout** и **cin** являются экземплярами классов **ostream** и **istream**. Существуют также классы **iostream** (класс для ввода/вывода) и **streambuf** (позволяет выполнить буферизованный ввод/вывод).

В программе не следует смешивать потоковый ввод/вывод с функциями **printf** и **scanf**. Если все же это происходит, То между блоками кода, использующими тот или иной подход, надо выполнять вызов функции **fflush** — сброс буферов.

- **Ссылки**

Для работы с ссылками подключается файл:

```
#include <string>
```

Расширение .h у файлов стандартной библиотеки отсутствует.

Среди строк наибольшей популярностью пользуются следующие классы:

```
std::string
std::wstring
```

- **Контейнеры**

В языке C++ существуют следующие контейнеры:

```
std::vector<T> // обычный массив
std::list<T>   // список, реализация не уточняется.
std::queve<T>  // FIFO
std::deque<T>  // двунаправленная очередь
std::stack<T>  // LIFO
std::map<K,T>  // список объектов T, индексированных ключами K
std::set<T>    // множество
std::bitset    // массив битов
```

Для использования любого из шаблонов надо подключить заголовочный файл, название которого совпадает с названием подключаемого шаблона.

В примере выше кроме параметра <T> есть еще, как правило, три параметра. Здесь они преднамеренно опущены и для них заданы стандартные значения.

Например, среди этих параметров есть **allocator** — это класс, который отвечает за создание или удаление элементов контейнера. Благодаря нему, можно создавать элементы в любой области памяти.

- **Итераторы**

```
#include <iterator>
```

Итератор — абстракция указателя на элемент контейнера.

Пример использования:

```
#include <iterator>
void strlen(const cahr *str)
{
  const char *p = str;
  while(*p != 0)
    {
      ...
      ++p;
    }
  ...
}
```

Указатель в строке — это итератор по строке.

Можно сказать, что в примере выше типы данных char* и const char* являются итераторами строки (обычной 0-терминированной).

С помощью оператора **typedef** можно определить такой тип данных и затем их использовать:

```
typedef char *iterator;
typedef const char *const_iterator;
```

Внутри каждого контейнера стандартной библиотеки C++ определены два типа данных: **iterator** и **const_iterator**, которые фактически являются указателями на элемент контейнера.

Работа с этими типами данных происходит следующим образом:

```
void Find(std::vector<MyObject*> &v, const char *s)
{
  typename std::vector<MyObject*>::iterator it = v.begin();
  while (it != v.end())
  {
    const char *szName = (*it).Name;
    if (strcmp(szName, s) == 0)
      return true;
    ++it;
  }
  return false;
}
```

В стандартном контейнере существуют функции begin() и end(), которые возвращают соответственно итераторы на первый и последний элементы контейнера. Физически функция end() возвращает **NULL**.

Если итератор адресует объект, то доступ к полям следует осуществлять с помощью оператора *. Допустимо использование и оператора ->, но он может быть переопределен и поэтому работа операторов *. и -> может отличаться. Переход к следующему элементу контейнера выполняется префиксным инкрементом ++it. Допустимо использование постфиксного оператора it++, но в последнем случае может возникнуть неоднозначность в том, что увеличивается на единицу — итератор или значение, возвращаемое итератором.

- **Алгоритмы**

```
#include <algorithm>
#include <cstdlib>
```

В стандартной библиотеке существует несколько стандартных алгоритмов, которые следует изучить до того, как разрабатывать собственные алгоритмы над контейнерами.

```
find()
sort()
*bsearch() // бинарный поиск
*qsort()   // быстрая сорировка
```

Недостатком **bsearch** является то, что не возвращается место вставки элемента.

- **Утилиты**

```
#include <utility>
#include <functional>
#include <memory>
#include <ctime>
```

В **utility** переопределен шаблон pair<F,S>.

В **fuctional** определены объекты функций — это такие объекты, которые могут использоваться как функции. Объекты функций применяются в алгоритмах. Например, выполняется обход контейнера и вызывается некоторая функция для каждого элемента контейнера. Вместо функции можно подставить объект-функцию. В этом и состоит смысл объекта-функции.

memory — распределение памяти для контейнера (здесь находятся стандартные алгоритмы).

В **ctime** находятся функции времени и даты.

- **Диагностика**

```
#include <stdexcept> // классы исключительных ситуаций
#include <cassert>   // макрос assert. Вместо него лучше использовать
                     // исключительные ситуации
#include <cerrno>    // файлы обработки ошибок в стиле C, то есть функции,
                     // которые возвращают коды ошибок
```

- **Локализация**

Поддержка различных языков:

```
#include <locale>
#include <clocale>
```

- **Поддержка языка программирования C++**

Крайние граничные значения типов данных:

```
#include <limits>
#include <climits>
#include <typeinfo>  // поддержка RTTI
#include <exception> // поддержка исключительных ситуаций
```

- **Работа с числами**

```
#include <complex>
#include <volarray>
#include <numeric>
#include <cmath>
#include <cstdlib>
```