

Глава 4. Визуальная модель Delphi.....	44
4.1 Процедурное программирование.....	44
4.2 Объектно-ориентированное программирование.....	47
4.3 Компонентная модель.....	51
4.4 Наследственность.....	51
Глава 5. Основы языка программирования Delphi.....	54
5.1 «Hello World» или из чего состоит проект.....	54
5.2 Язык программирования Delphi.....	62
5.3 Типы данных в Delphi.....	66
5.3.1 Целочисленные типы данных.....	67
5.3.2. Вещественные типы данных.....	67
5.3.3 Символьные типы данных.....	68
5.3.4. Булевы типы.....	72
5.4.5. Массивы.....	73
5.4.6. Странный PChar.....	74
5.4 Процедуры и функции в Delphi.....	75
5.5 Рекурсивный вызов процедур.....	80
5.6 Встроенные процедуры.....	82



Глава 4. Визуальная модель Delphi.



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

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

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

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

4.1 Процедурное программирование.

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

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

Команда 1
Команда 2
Команда 3
...
...
Команда N.

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

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

...

Команда N.

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

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

Начало процедуры 1

Команда 1

Команда 2

Конец процедуры 1

Начало программы

Команда 1

Команда 2

Если выполнено условие, то выполнить код процедуры 1.

Команда 3

Конец программы.

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

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

Начало процедуры 1 (Переменная 1: строка)

Команда 1

Команда 2

Конец процедуры 1

Начало программы

Команда 1

Команда 2

Если выполнено условие, то выполнить код процедуры 1.

Команда 3

Конец программы.

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



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

Давай рассмотрим пример процедуры приближённый к реальности:

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

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

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

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

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

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

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

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

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

Заметь, что после двоеточия идёт описание типа возвращаемого значения.

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

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

Начало программы

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

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

4.2 Объектно-ориентированное программирование.

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

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

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

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

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

Простой объект выглядит так:

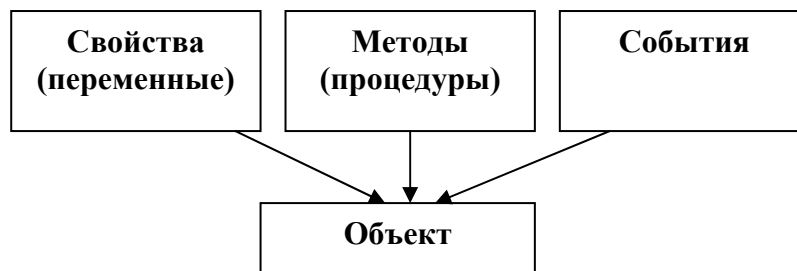


Рис 4.2.1 Графическое представление объекта.

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

Свойства:

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

Методы:

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

События:

1. Кнопка нажата.
2. Заголовок кнопки изменён.

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

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

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



Процесс создания объекта называется инициализацией.

Теперь рассмотрим процесс применения нашей кнопки. Он выглядит следующим образом:

1. Создание кнопки с помощью вызова метода *«Создать кнопку»*.
2. Изменение необходимых свойств.

Всё. Наша кнопка готова к работе.



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

На языке программирования это будет выглядеть немного сложнее:

1. Объявить переменную типа Кнопка.
2. В эту переменную проинициализировать объект.
3. Можно использовать объект.

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

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

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

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

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

Начало программы.

Переменные:

Объект1 – Кнопка;

Начало кода

Объект1:= Кнопка.Создать объект

Объект1.Заголовок:='Привет'

Объект1.Уничтожить объект.

Конец кода

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

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

Начало программы.

Переменные:

Объект1 – Кнопка;

Начало кода

Объект1.Заголовок:='Привет'

Конец кода

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

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



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

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

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

Начало программы.

Переменные:

Объект1 – Кнопка;

Объект2 – Кнопка;

Начало кода

Объект1:= Кнопка.Создать объект

Объект2:= Кнопка.Создать объект

Объект1.Заголовок:='Привет'

Объект2.Заголовок:='Пока'

Объект1.Уничтожить объект.

Объект2.Уничтожить объект.

Конец кода



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

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

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

Объект1.Free.

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

4.3 Компонентная модель.

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

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

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

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

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

4.4 Наследственность.

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

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

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

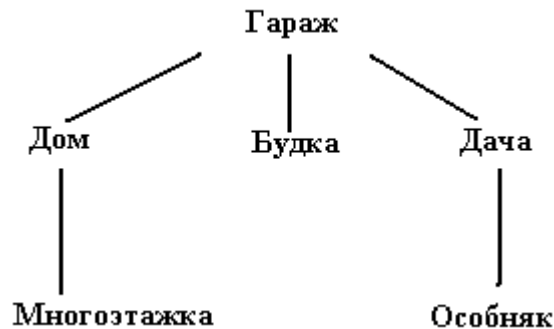


Рис 4.4.1 Иерархия гаража.

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

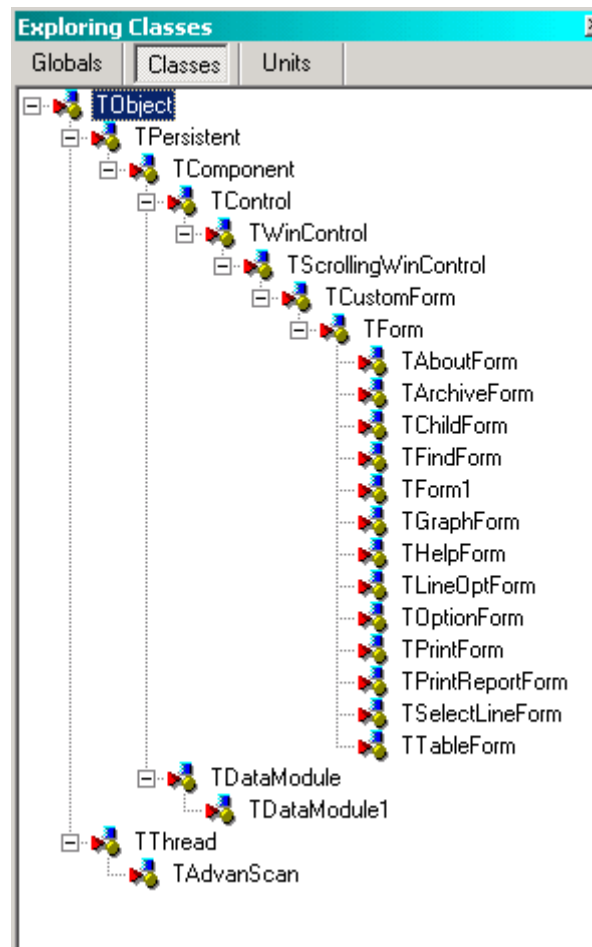


Рис 4.4.2 Иерархия ООП в Delphi.

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

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

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

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

Процедура отвечающая за создание стен у гаража.

Начало

Создать стены

Конец

Процедура отвечающая за создание стен у дома.

Начало

Вызвать объект предка.

Повесить на стены картины.

Конец

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



Глава 5. Основы языка программирования Delphi.



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

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

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

оба термина идентичны.

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

5.1 «Hello World» или из чего состоит проект.

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

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

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

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

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

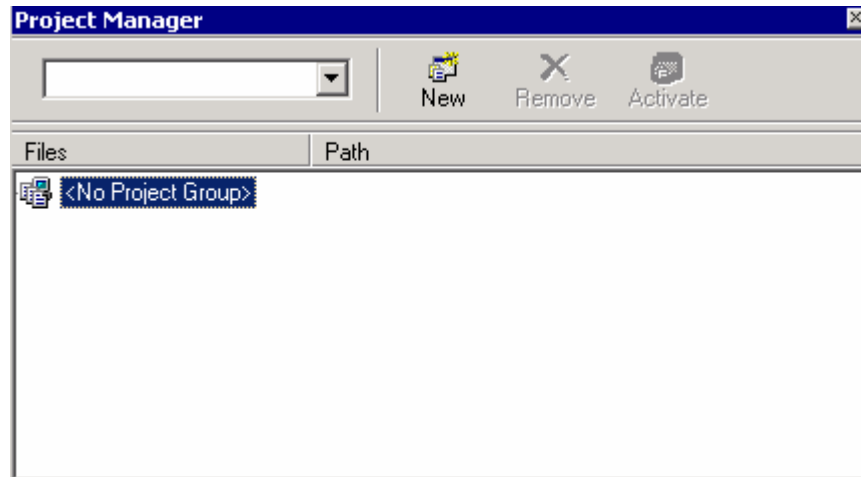


Рис. 5.1.1 Менеджер проектов.

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

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

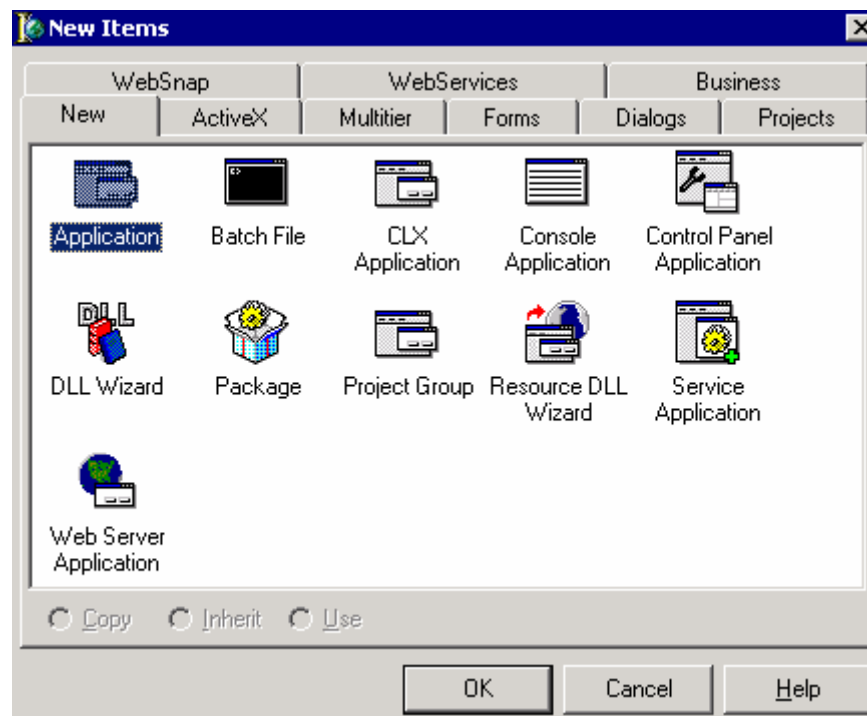


Рис. 5.1.2 Создание нового проекта.



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

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

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

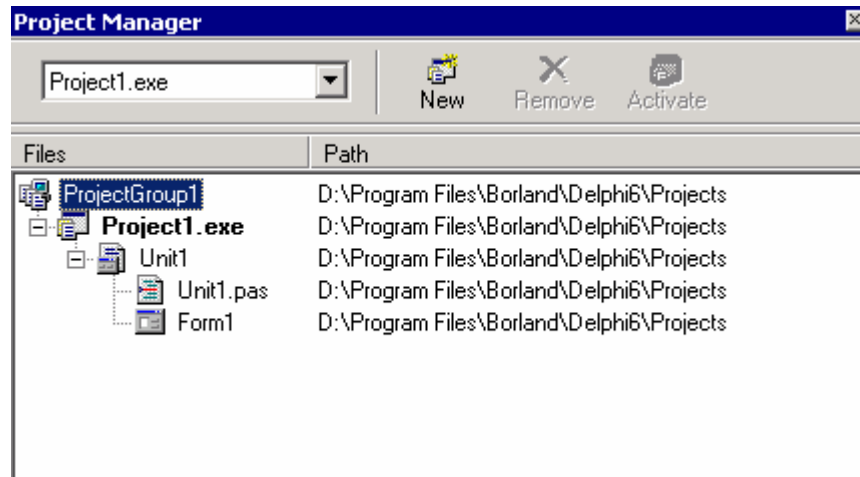


Рис. 5.1.3 Менеджер проекта при созданном проекте.

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

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

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

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

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



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

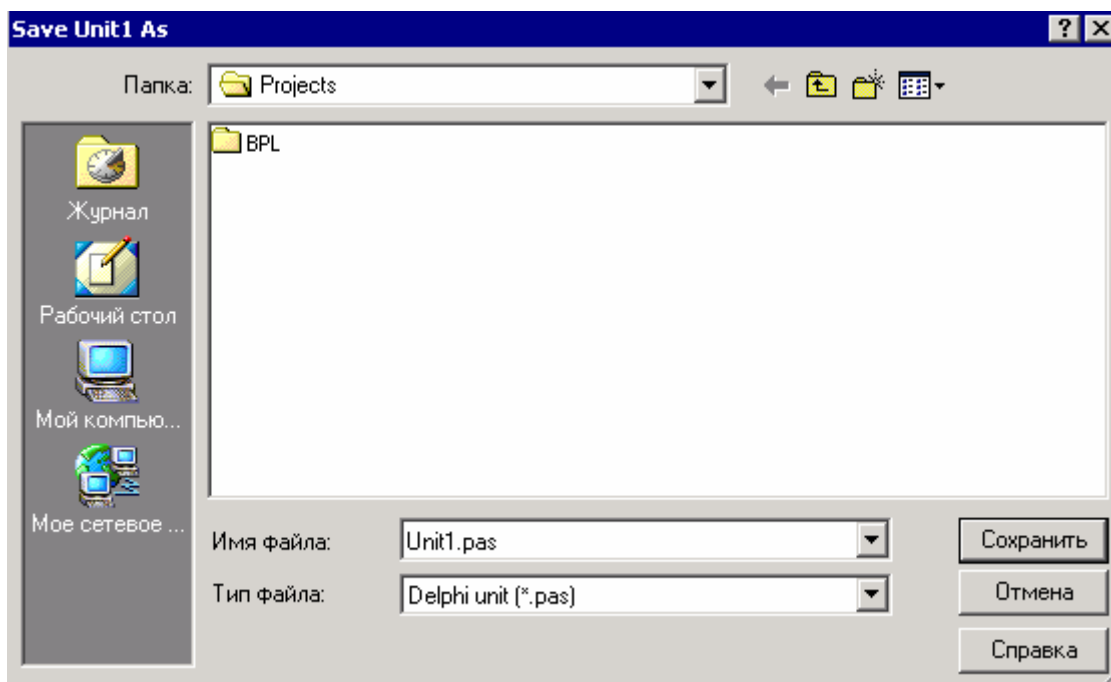


Рис. 5.1.4 Запрос ввода имени модуля.

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

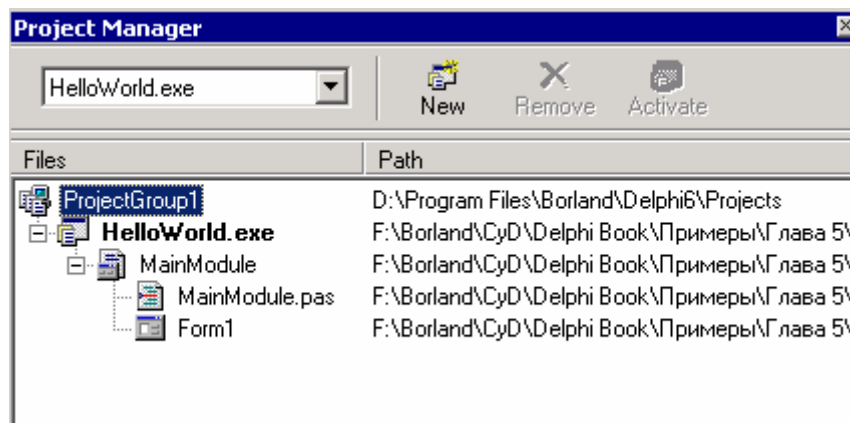


Рис. 5.1.5 Менеджер проекта.

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

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

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

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

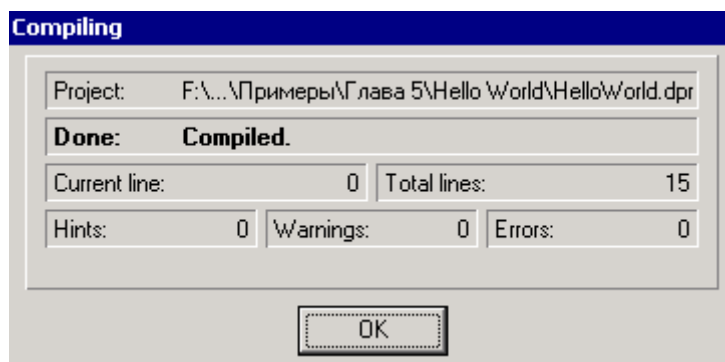


Рис. 5.1.6 Результат компиляции.

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

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

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

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

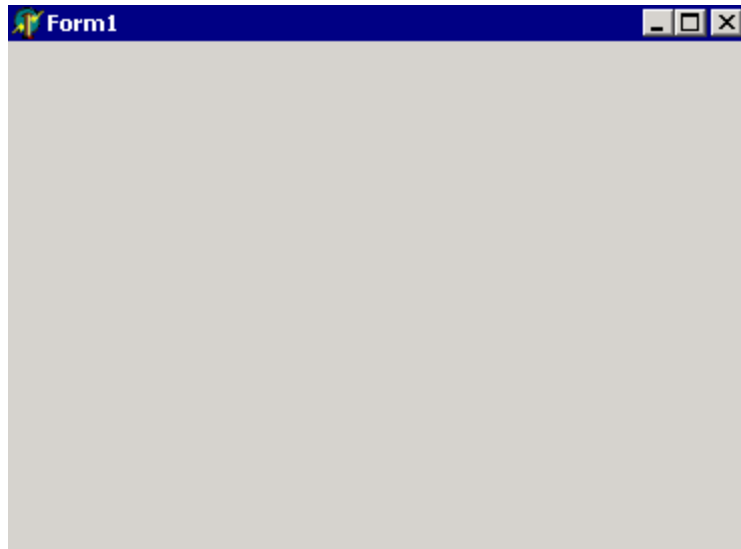


Рис. 5.1.7 Программа в действии.

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

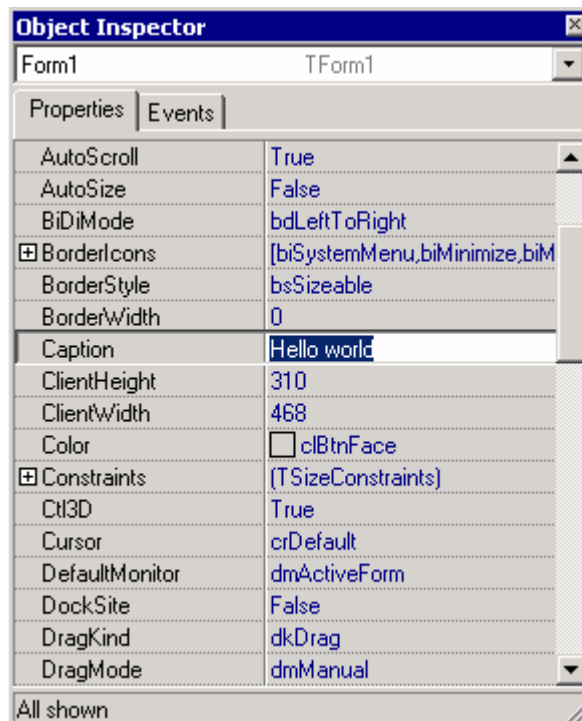


Рис. 5.1.8 Объектный инспектор.

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

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

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

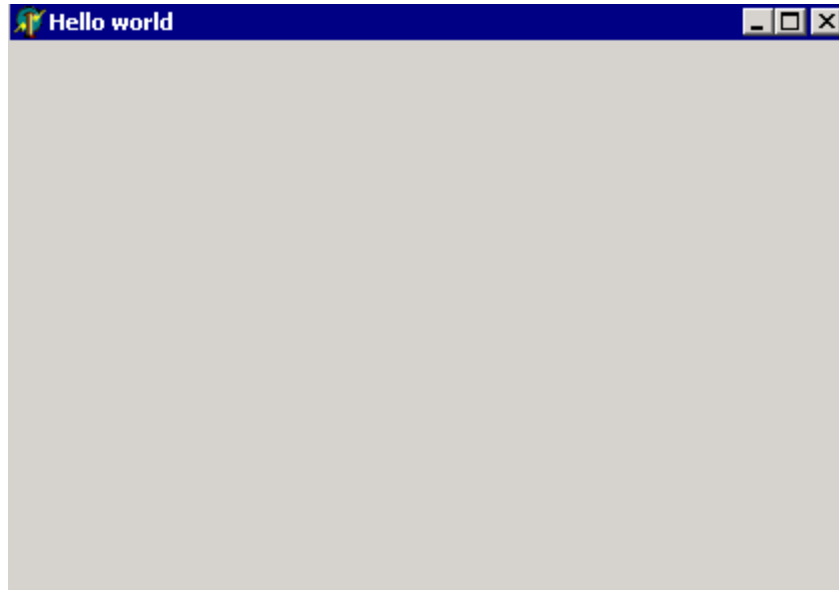


Рис. 5.1.9 Hello World в действии.



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

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

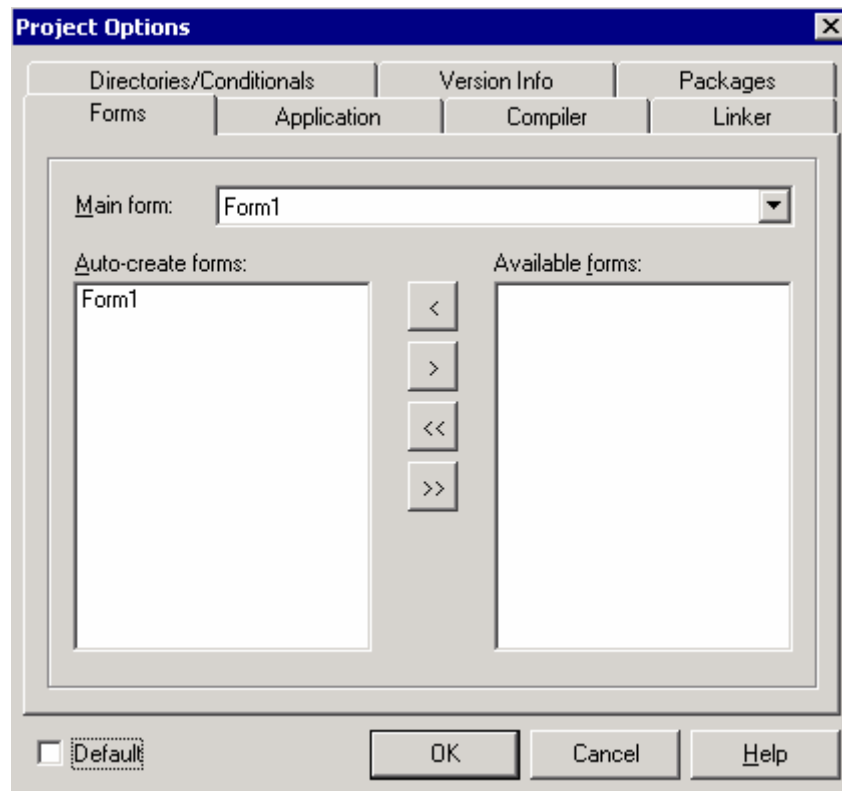


Рис. 5.1.10 Свойства проекта

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

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

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

Title – заголовок, который будет отображаться в панели задач.

Help file – имя файла помощи.

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

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

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

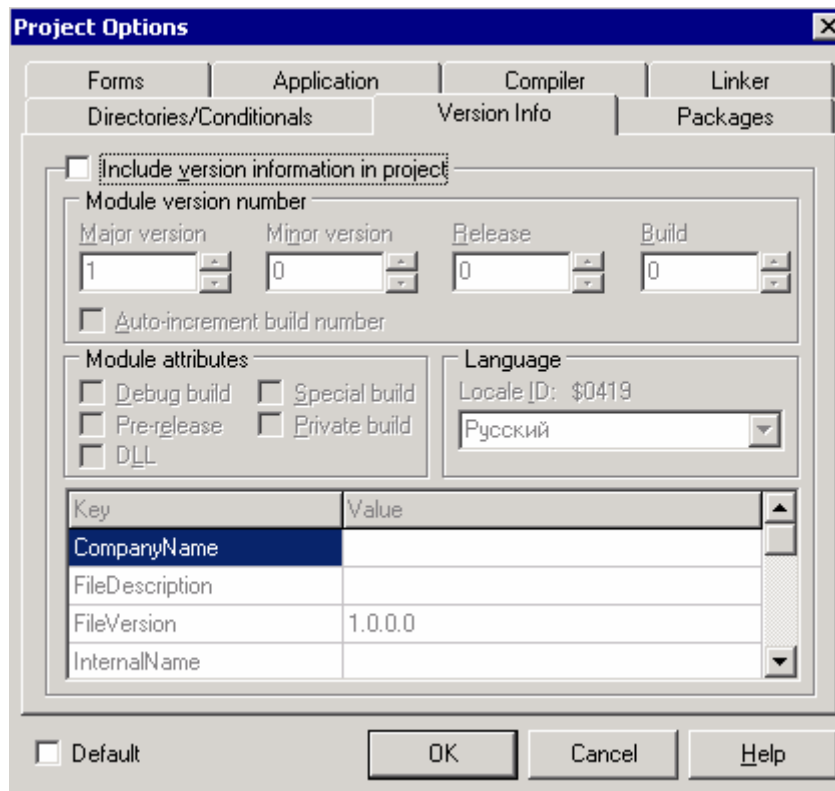



Рис. 5.1.11 Свойства проекта

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

5.2 Язык программирования Delphi.

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

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

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

Рассмотрим пример:

//Это комментарий.

Это уже не комментарий

{Это снова комментарий

И это тоже}

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

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

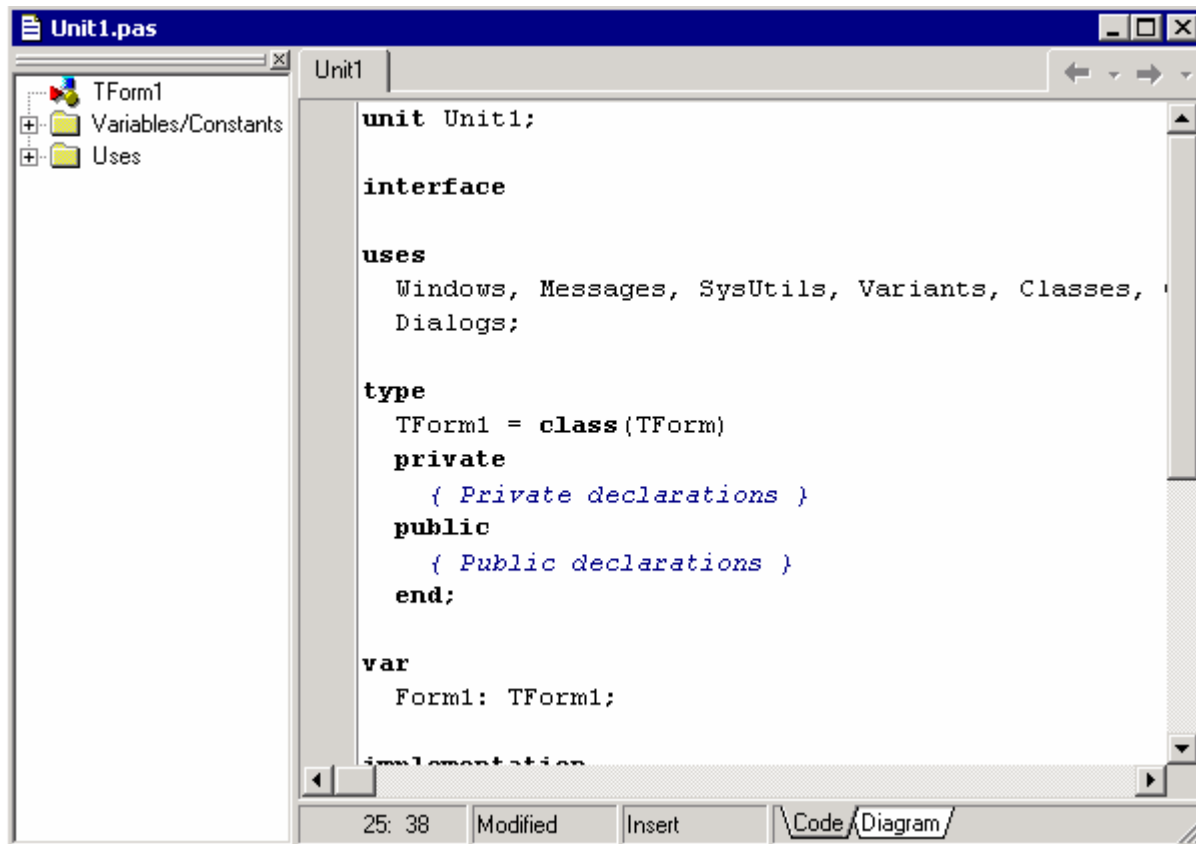


Рис. 5.2.1 Hello World в действии.

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

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

```
unit Unit1; //Имя модуля
```

```
interface
```

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

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

```
//Здесь описываются компоненты и события
```

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

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

```
public //После этого слова можно описывать открытые данные объекта  
  { Public declarations } //Подсказка, которую сгенерировал Delphi  
  
  {Здесь можно описывать переменные и методы доступные из любого другого модуля}  
end;  
  
var //Объявление глобальных переменных  
  Form1: TForm1; //Это описана переменная Form1 типа объекта TForm1  
  
implementation  
  
{$R *.dfm} //Подключение .dfm файла (файл с данными о визуальных объектах)  
  
end. // end с точкой - конец модуля
```

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

```
Имя объекта = class(Имя предка)

  //Здесь описываются компоненты и события

  private //После этого слова можно описывать закрытые данные объекта

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

    Переменная1: Integer;
    Procedure Proc1;

  public //После этого слова можно описывать открытые данные объекта

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

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

private

Переменная1: Integer;

Procedure Proc1; *// Это процедура, после неё не может быть переменных*

Переменная2: String; *// Это ошибка. Уже объявлена одна процедура.*

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

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

var *// Объявление глобальных переменных*

Form1: TForm1; *// Это описана переменная Form1 типа объекта TForm1*

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

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

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

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

5.3 Типы данных в Delphi.

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

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

Рассмотрим, какие бывают типы переменных.

5.3.1 Целочисленные типы данных.

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

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

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

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

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

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

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

5.3.2. Вещественные типы данных.

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

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

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



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

5.3.3 Символьные типы данных.

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

10 14 18

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

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

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

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

Таблица ANSI

Симво Номе Симво Номе Симво Номе Симво Номе Симво Номе

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

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

207 208 200 194 210

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

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

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

В Delphi присутствуют следующие основные типы строк:

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

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

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

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



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.3.4. Булевы типы.

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

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

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

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

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

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

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

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



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

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

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

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

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

5.4.5. Массивы.

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

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

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

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

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

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

5.4.6. Странный PChar.

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

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

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

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

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

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

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

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

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

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

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

5.4 Процедуры и функции в Delphi

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

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

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

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

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

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

Для вызова процедуры нужно написать только Exampl.

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

```
procedure TForm1.Examp2;  
begin  
  Examp1; //Вызываем процедуру Examp1 написанную ранее.  
end;
```

В этой процедуре Examp2 я вызываю написанную ранее Examp1. Все имена процедуры должны начинаться с латинских букв и могут заканчиваться цифрами. Нельзя использовать русские буквы и имена процедур не могут начинаться с цифр.

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

```
procedure Examp2;  
begin  
  Examp1; //Произойдёт ошибка, потому что процедура Examp1; описана ниже  
end;  
  
procedure Examp1;  
var  
  i:Integer;  
begin  
  i:=10;  
end;  
  
procedure Examp4;  
begin  
  Examp1; //Здесь ошибки не будет, потому что Examp1; описан выше.  
end;
```

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

```
type  
  TForm1 = class(TForm)  
  private:  
    procedure Examp1;  
    procedure Examp2;  
  public:  
    procedure Examp3;  
    procedure Examp4;  
  end;
```

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

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

```
function Examp1:Integer;  
var  
  i:Integer;//Объявление локальной переменной  
begin  
  i:=10;//Присваиваю переменной значение  
  Result:=i; // Возвращаю значение i  
end;
```

Я объявил функцию, которая будет возвращать значение типа integer (целое число) function Examp1: Integer. Для возврата значения, его нужно присвоить к переменной Result, как я это делаю в примере выше.

Для вызова функции можно делать так:

```
procedure TForm1.Examp2;  
var  
  x:Integer;  
begin  
  x:=Examp1; //Вызываем процедуру Examp1 написанную ранее.  
end;
```

В этом примере я присваиваю переменной x значение, возвращаемое функцией Examp1.

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

```
function Examp1(index:Integer):Integer;  
begin  
  Result:=index*2; // Возвращаю переданное значение index  
                  // умноженное на 2  
end;
```

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

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

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

```
function Examp1(index:Integer):Integer;
```

```
begin
  Examp1:=index*2; // Возвращаю переданное значение index
                  // умноженное на 2
end;
```

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

Вызов моей функции будет такой:

```
procedure TForm1.Examp2;
var
  x:Integer;
begin
  x:=Examp1(20); //Вызываем процедуру Examp1 написанную ранее.
end;
```

Я предаю в функцию Examp1 значение 20, а она мне вернёт 20 умноженное на 2, то есть 40. Это я показал пример с функцией, но точно так же можно поступать и с процедурами, передавая им значения.

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

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

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

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

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

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

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

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

```
function Examp1(index1, index2:Integer):Real;
begin
  Examp1:=index1/ index2;
end;

procedure Examp2;
var
  x: Real;
begin
  x:=Examp1(20, 10); //Вызываем процедуру Examp1 написанную ранее.
end;
```

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

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

```
function Examp1(index1, index2:Integer):Real;
begin
  Если index2 равен 0, то exit.
  Examp1:=index1/ index2;
end;

procedure Examp2;
var
  x:Real;
begin
  x:=Examp1(20, 0); //Вызываем процедуру Examp1 написанную ранее.
  x:=x*2; // Здесь произойдёт ошибка
end;
```

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

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

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

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

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

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

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

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

5.5 Рекурсивный вызов процедур

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

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

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

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

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

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

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

Мы сами должны написать код, который будет прерывать рекурсию:

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

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

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

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

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

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

5.6 Встроенные процедуры

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

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

```
procedure Examp1(Sender: TObject);
```

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

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

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

```
procedure Examp1(Sender: TObject);
```

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

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

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


Автор: Horrific aka Фленов Михаил e-mail: vr_online@cydsoft.com

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

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