

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Казанский государственный
энергетический университет

В. Е. ЛЕОНТЬЕВ

Утверждено
учебным управлением КГЭУ
в качестве учебного пособия
для студентов

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ В СРЕДЕ BORLAND DELPHI

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

по курсам

Программное обеспечение измерительных процессов
Защита информации

Казань 2006

УДК 681.325.5
ББК 32.81
Л 47

Леонтьев В. Е.

Прикладное программирование в среде Borland Delphi: Лабораторный практикум / Казань: Казан. гос. энерг. ун-т, 2006.

Настоящее пособие представляет собой лабораторный практикум по фундаментальным основам создания прикладных программ в современных средах разработки. Каждая лабораторная работа в этом пособии предваряется значительным теоретическим материалом, который предназначен как для аудиторного, так и самостоятельного обучения и призван восполнить возможные пробелы в изучении лекционного курса по данной дисциплине. Пособие состоит из двух частей и содержит восемь лабораторных работ.

Первая часть включает пять работ, которые посвящены изучению основ использования интегрированной среды разработки Borland Delphi, языка программирования Object Pascal, базовых технологий и приемов программирования прикладных программ для операционной системы Windows. Практическая часть лабораторных работ направлена на изучение приемов программирования, принципов построения алгоритмических структур. В одной из работ этого цикла рассматриваются практические аспекты реализации программ, управляющих внешними устройствами, которые подключаются к ПЭВМ.

Во второй части лабораторного практикума рассматривается применение криптографических алгоритмов и протоколов для реализации встраиваемых подсистем безопасности прикладных программ. В трех лабораторных работах этого цикла даются практические навыки использования профессиональной библиотеки криптографических алгоритмов для Delphi – LockBox2 компании TurboPower Professional. Приводятся примеры использования симметричных алгоритмов DES и AES, асимметричного алгоритма RSA и алгоритма цифровой подписи DSA, а также различных односторонних функций шифрования.

Предназначено для студентов, углубленно изучающих технологии программирования и защиты информации по специальностям 200106 – «Информационно-измерительная техника» и 230401 – «Прикладная математика».

Рецензенты

Д-р. физ.-матем. наук, ведущий науч. сотрудник КазФТИ
им. Е. К. Завойского КазНЦ РАН

Р. М. Баязитов

Канд. пед. наук, доцент каф. ПЭ КГЭУ

Л. В. Ахметвалеева

Рекомендовано секцией РИС факультета энергомашиностроения

Председатель секции С.Р. Сидоренко

© Казанский государственный энергетический университет, 2006 г.

ЧАСТЬ I. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ В СРЕДЕ BORLAND DELPHI

Лабораторная работа №1

ОСНОВЫ ИСПОЛЬЗОВАНИЯ ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ DELPHI ПРИ ПОСТРОЕНИЯ ПРОСТЕЙШИХ WINDOWS- ПРИЛОЖЕНИЙ

1. Цель работы

Основной целью данной лабораторной работы является изучение студентами ключевых положений современного компонентно-ориентированного программирования на примере высокоразвитой интегрированной среды разработки Delphi. Вместе с этим студенты должны получить элементарные практические навыки создания современного программного обеспечения: научиться работать с дизайнером форм, редактором кода, инспектором объектов, а также создавать примитивные Windows-приложения.

2. Основные теоретические сведения

2.1. Введение в интегрированные среды разработки программ

Современные среды разработки программного обеспечения предоставляют разработчику-программисту высокоразвитые инструменты, которые существенным образом сокращают время проектирования и реализации программ, а также позволяют сделать разрабатываемое программное обеспечение значительно более качественным. Наряду с этим использование интегрированной среды разработки существенным образом облегчает труд программиста.

Под *интегрированной средой разработки (integrated developer environment – IDE)* понимают комплекс взаимосвязанных программных средств, который реализует развитый программный интерфейс между *компилятором (compiler)*, *компоновщиком (linker)* и другими служебными программами, участвующими в процессе формирования бинарного кода приложения с одной стороны и разработчиком программ с другой стороны. Этот программный интерфейс реализуется в виде следующих элементов среды, таких, как: редактор кода, редак-

тор оконного интерфейса (конструктор форм), менеджер проектов, инспектор объектов программы, палитра визуальных компонентов. Среда разработки позволяет управлять не только процессом кодирования приложения, но и берет на себя функции управления процессом компоновки и отладки приложения. Так, все современные среды разработки имеют встроенные средства пошаговой отладки, расстановки контрольных точек в программе и точек останова.

В настоящее время существует большое число современных высокоразвитых сред разработки приложений, ориентированных на компонентное объектно-ориентированное программирование. Различные среды разработки программного обеспечения ориентированы на различные сферы применения, используют в качестве языкового инструмента различные алгоритмические языки программирования. Однако, несмотря на функциональные различия, все современные среды разработки программ реализуют схожие концепции и подходы к разработке программного обеспечения и имеют в своем составе схожие инструментальные средства разработчика.

Хорошо известно, что на сегодняшний день законодателем «мод» в области разработки системного и прикладного программного обеспечения в целом и интегрированных сред разработки в частности является корпорация Microsoft. Наиболее известными и используемыми программными продуктами этого производителя является линейка сред разработки из состава пакета Visual Studio: Visual Basic, Visual C++, Visual FoxPro, Visual J++ и Visual InterDev.

Другим крупным «игроком» в области создания высокоразвитых сред разработки является компания Borland Enterprise. Ее программные продукты, такие, как Borland Delphi, Borland C++ Bulder, Borland JBulder и другие, являются наиболее популярными средами разработки у разработчиков проектов среднего и высокого уровней сложности. Интегрированные среды разработки от компании Borland позиционируются как инструменты для создания клиентских и серверных приложений, в том числе распределенных для систем управления базами данных. Важно при этом отметить, что эти программные продукты имеют наиболее развитые инструментальные, языковые и компонентные средства.

При изучении современных концепций высокоуровневого программирования оптимальным выбором является среда разработки Borland Delphi, которая использует в качестве базового языка программирования Object Pascal, а также развитую объектно-ориентированную компонентную библиотеку классов Visual Component Library, реализующую для программиста основные строительные блоки приложения.

2.2. Основные компоненты среды Delphi

Внешний вид главного окна среды разработки Delphi представлен на рис. 1.1. С точки зрения пользовательского интерфейса Delphi состоит из нескольких оконных областей.

В первом окне (1) располагается главное меню программы, палитры инструментов, команды которых дублируют команды главного меню. Наряду с этим в основном окне среды располагаются закладки с палитрами компонентов, которые используются как строительные блоки создаваемого приложения. К таким визуальным компонентам относятся как стандартные элементы управления Windows, такие, как кнопки, строки редактирования, списки и комбинированные списки, так и сложные нестандартные элементы управления, реализованные в компонентной библиотеке Delphi.

Второе окно (2) располагает в своей клиентской области *редактор исходного кода (Code Editor)* (3) создаваемого приложения, а также закладки (слева) *менеджера проекта (Project Manager)* (4) и *навигатора кода (Code Explorer)*. В нижней части данного окна располагается область вывода сообщений среды (5).

Следующая оконная область среды представляет собой образ окна проектируемого приложения и является одновременно редактором оконного интерфейса – это так называемый *дизайнер форм (Form Designer)* (6). Это окно является контейнером для визуальных компонентов из которых строится интерфейс приложения. Визуальные компоненты добавляются на форму во время разработки приложения из палитры компонентов.

Другим инструментальным средством Delphi является *инспектор объектов (Object Inspector)* (7). Инспектор объектов содержит список свойств объектных визуальных компонентов, которые используются во время разработки приложения, а также позволяет изменять значения этих свойств и ассоциировать программный код (обработчики событий) с этими компонентами.

Рассмотрим более подробно функциональные возможности и использование каждого из упомянутых элементов среды разработки.

При первом запуске среда разработки Delphi автоматически создает *проект (project)* по умолчанию, состоящий из одного модуля исходного кода, с которым связана *форма (form)* – главное окно будущего приложения.

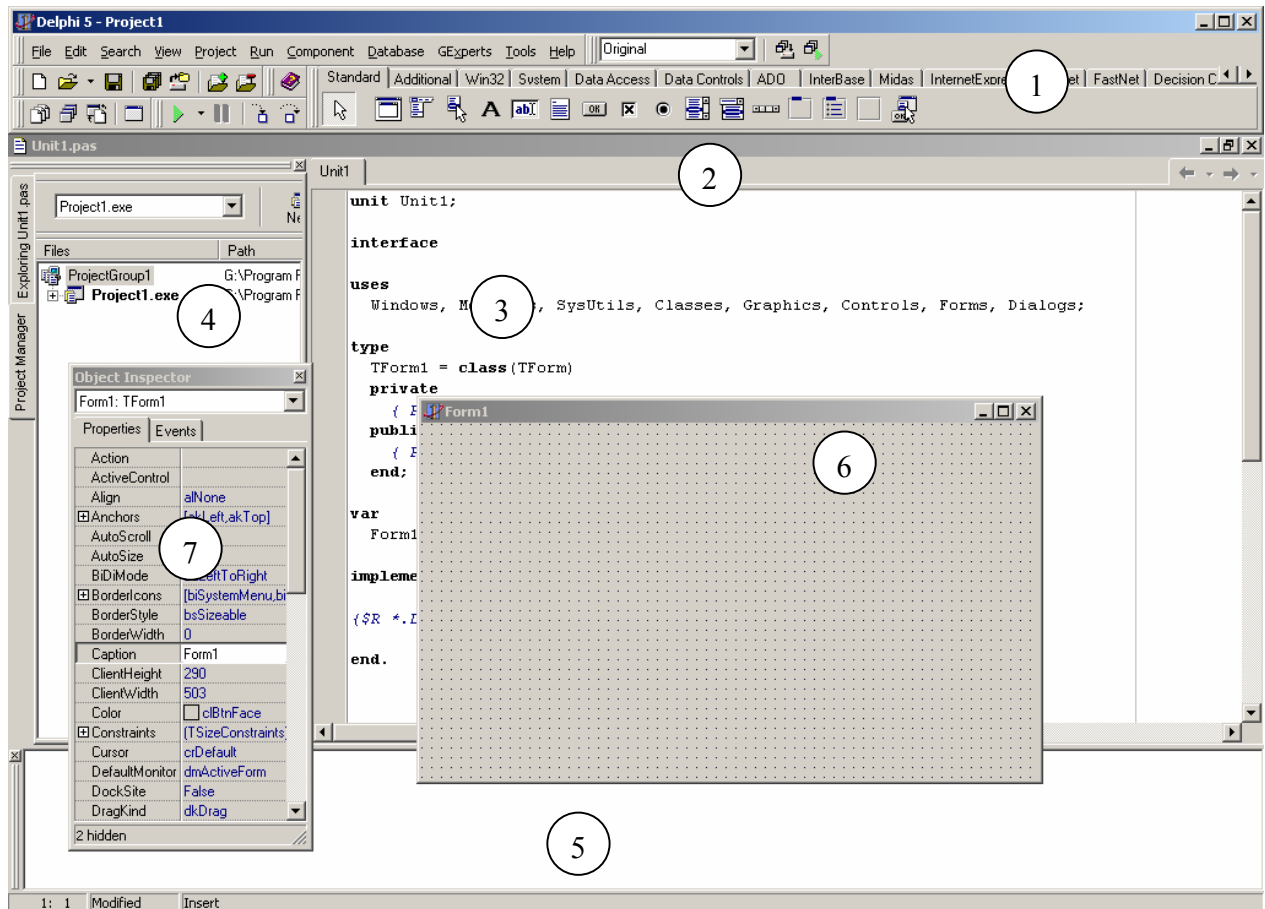


Рис. 1.1. Внешний вид среды разработки Delphi

Как и любая другая среда разработки, Delphi использует абстракцию так называемого *проекта (project, design)*, который объединяет в единое целое файлы, относящиеся к одному разрабатываемому приложению. Иерархия файлов, составляющих проект, отображается в менеджере проектов (рис. 1.2) в виде дерева структурных элементов проекта: модулей и форм. В вершине такой иерархии находится *группа проектов (project group)*, которая может объединять несколько одновременно разрабатываемых приложений. На следующем уровне располагается проект, который в свою очередь содержит модули проекта. Если модуль ассоциирован с формой приложения (окном), то для модуля доступен отдельно исходный текст в редакторе кода и форма в конструкторе форм. Для доступа и открытия элемента проекта пользователь должен либо выбрать команду контекстного меню, либо выполнить двойной нажатие левой кнопки мыши. Менеджер проектов позволяет добавить и удалить модули проекта, вы-

полнить компиляцию отдельного проекта и настроить опции каждого проекта в отдельности.

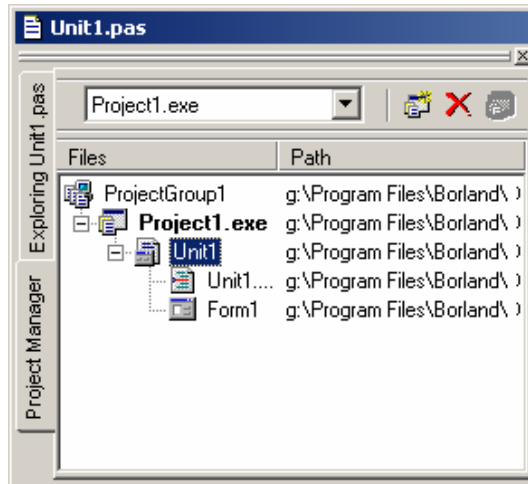


Рис. 1.2. Внешний вид менеджера проектов

Создание приложения в среде Delphi, как правило, начинается с конструирования *пользовательского интерфейса (user interface)*. Как уже отмечалось, для его формирования используется дизайнер форм. Для этого на этапе разработки приложения необходимый элемент управления должен быть выбран из палитры компонентов (рис. 1.3) и размещен в области формы (рис. 1.4).



Рис. 1.3. Выбранный компонент Button из палитра компонентов Standard

Размещенный на форме и выделенный компонент можно перемещать для его расположения в нужной позиции, копировать на данную или другую форму приложения, изменять его линейные размеры и удалять с формы. Для этого используются команды главного или/и контекстных меню и комбинации клавиш, аналогичные другим средам разработки и графическим редакторам. (рис. 1.4).

Одни компоненты отображаются на форме в том виде, как они будут выглядеть в окне работающего приложения (кнопки, списки, строки редактирования, панели и д.р.), то есть на этапе исполнения программы, тогда как другие отображаются в виде пиктограмм и не являются визуальными компонентами.

Невизуальные компоненты могут представлять в дизайнера специальные элементы управления, например, главное и контекстное меню, или предоставлять разработчику функциональность времени исполнения.

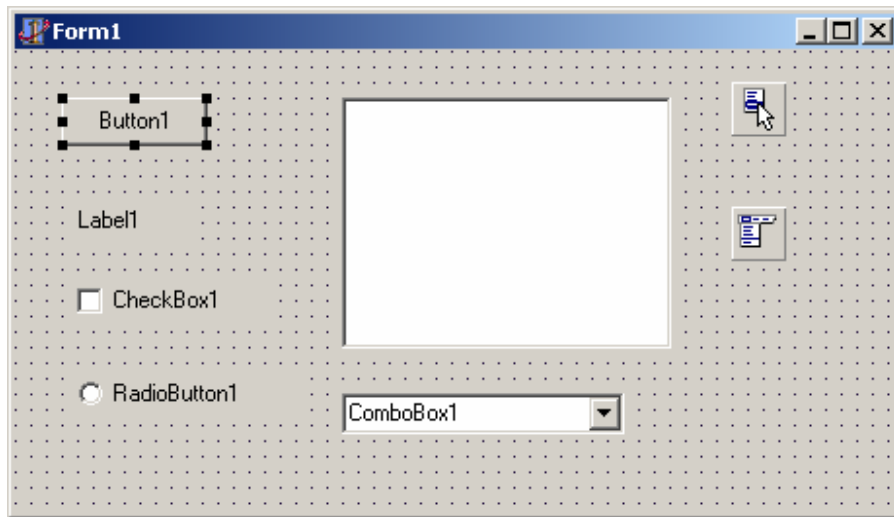


Рис. 1.4. Форма (окно) приложения на этапе разработки пользовательского интерфейса

После размещения компонентов на форме, в ходе проектировании приложения необходимо установить свойства компонентов и определить обработчики событий для них. Задание свойств и событий позволяет изменить внешний вид компонента или его поведение на этапе выполнения приложения. Если компонент выделен на форме, его свойства и события отображаются в инспекторе объектов (7). Инспектор объектов является связующим звеном между кодом создаваемого приложения и визуальной средой разработки. Инспектор объектов имеет две закладки (рис. 1.5): Properties (Свойства) и Events (События). Первая служит для отображения значений различных свойств компонента приложения, а вторая содержит список всех возможных событий, реакция на которые в виде подпрограмм может быть задана для этого компонента.

Под *свойством (property)* можно понимать некоторую характеристику визуального компонента. Большинство свойств являются простыми значениями, такими как имена цветов, типов курсора, стилей или шрифтов, целыми, булевыми или строчными значениями, задающие характеристики компонентов, например ширину, высоту, название и т.п. Некоторые свойства имеют ассоциированные редакторы, которые служат для установки сложных значений. Если

на форме выделено сразу несколько компонентов, то в инспекторе объектов отображаются только присущие всем выделенным компонента свойства и события.

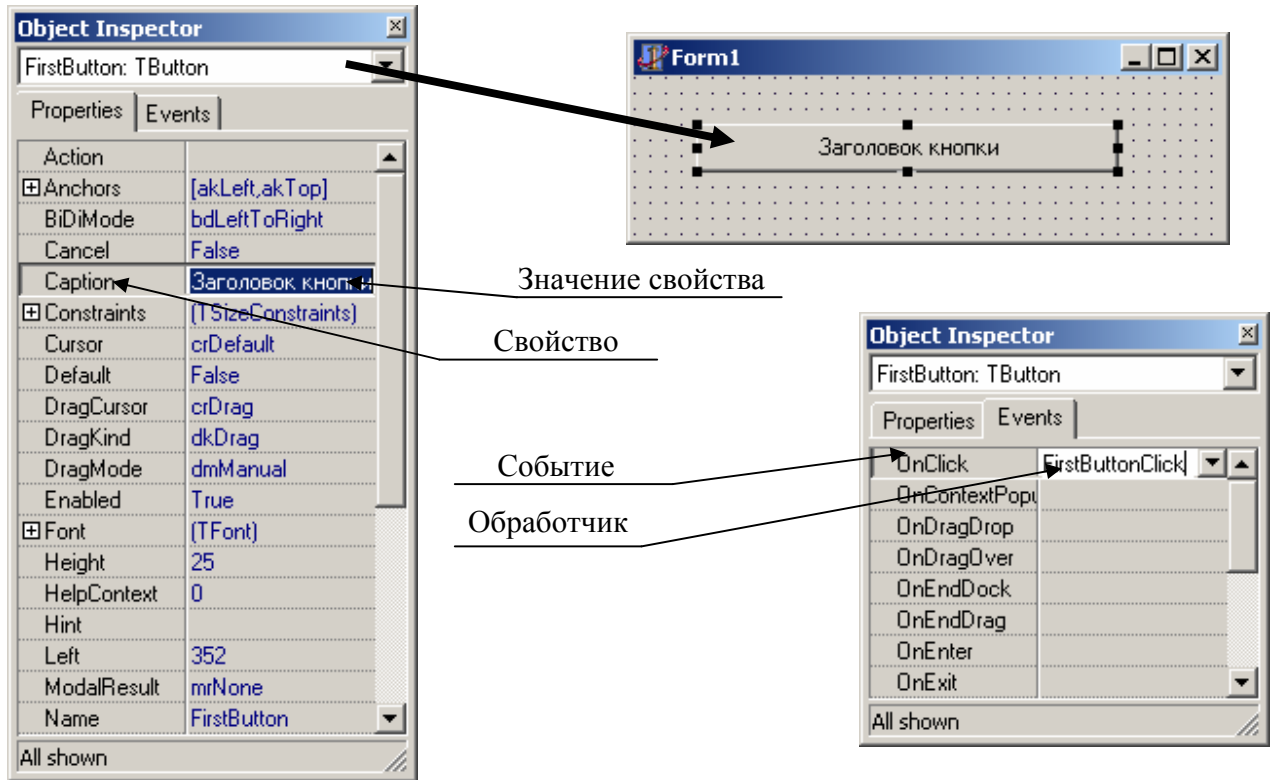


Рис. 1.5. Инспектор объектов

Несмотря на возможности построения интерфейса приложения в конструкторе форм с использованием визуальной технологии, логика приложения реализуется только путем написания исходного кода программы на высокоуровневом языке программирования в редакторе кода среды разработки. Поэтому редактор кода занимает центральное место в любой интегрированной среде разработки. Программист-разработчик большую часть времени работает в редакторе кода и функциональные возможности именно этого инструмента во многом определяют эффективность и скорость разработки. К важнейшим функциям редактора кода Delphi можно отнести следующие:

- *подсветка синтаксиса (Syntax Highlight)* – автоматическое выделение цветом или стилем шрифта ключевых слов языка, операторов, разделителей строчных и числовых литералов;

• *технология Code Insight* – автоматическое отображение в редакторе кода списка параметров подпрограмм и параметров методов класса (Code Parameters), списков методов, свойств и событий объекта класса (Code Completion). Эта функция активизируется автоматически при вводе идентификаторов объектов и подпрограмм (рис. 1.6). Технология Code Insight позволяет автоматически завершать ввод языковых конструкций и создавать их шаблоны, а также генерировать каркас методов класса по его интерфейсу (Code Template).

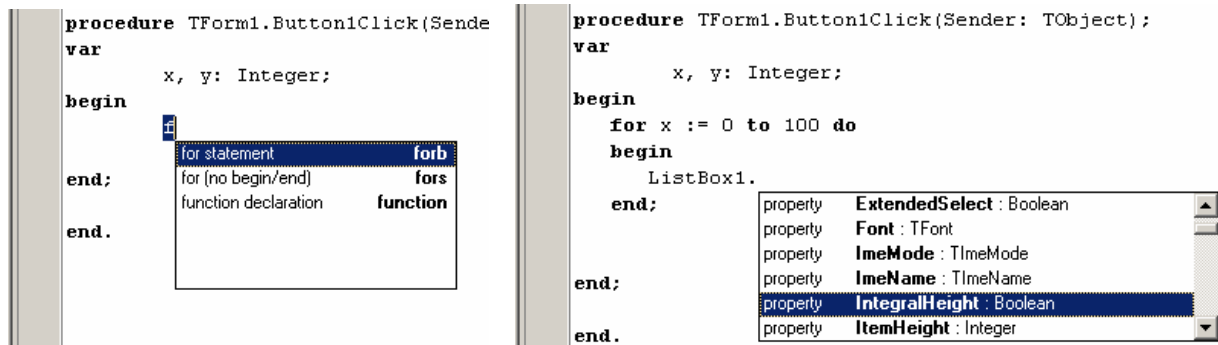


Рис. 1.6. Технология Code Insight в редакторе кода Delphi

Ядром любого приложения, созданного в среде Delphi, является код, реализующий функциональность различных компонентов, которые являются элементарными строительными блоками, в том числе образующими пользовательский интерфейс приложения. Этот код образует *библиотеку классов и подпрограмм Delphi* (Visual Component Library – VCL), которая является частью среды разработки. Для создания же бизнес-логики приложения, т.е. некоторой собственной функциональности программы, разработчику необходимо с некоторыми компонентами, из которых строится приложение, связать программный код в виде подпрограмм, выполняющийся как реакция на действия пользователя. Механизм связывания программного кода с компонентами приложения и свойства специального типа, с которыми ассоциируется этот код, получили название *событий (events)*. К примитивным событиям, на которые могут реагировать визуальные компоненты, относятся, например, одинарное или двойное нажатие кнопки мыши, нажатие и отпускание клавиши клавиатуры и т.д. В большинстве случаев с некоторыми визуальными компонентами, необходимо связать одну или несколько подпрограмм-обработчиков соответствующих событий. Среда Delphi позволяет быстро создать каркас подпрограммы-обработчика, используя инспектор объектов.

На второй закладке инспектора (Events) находится список всех возможных событий выделенного на форме компонента. Двойное нажатие в поле события этой закладки инспектора объектов создает соответствующий обработчик и переключает среду в редактор кода (рис. 1.7).

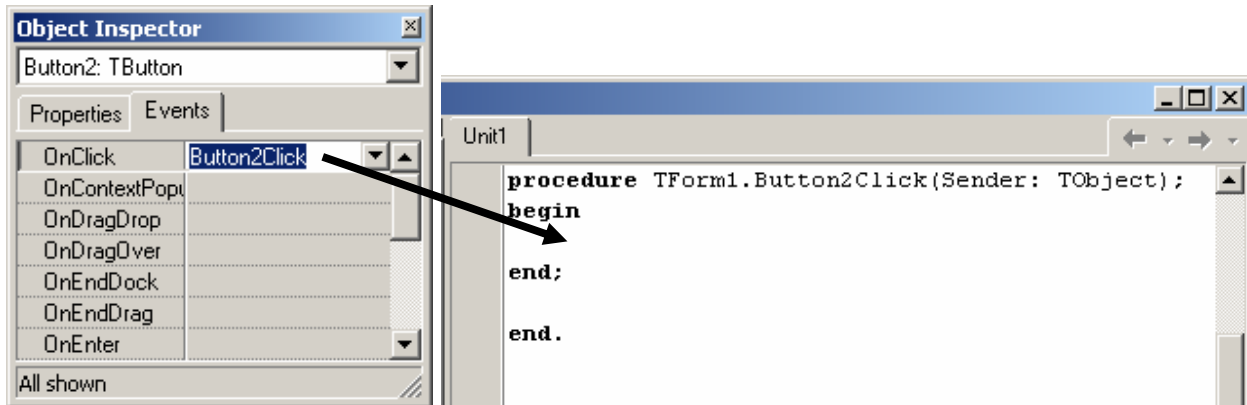


Рис. 1.7. Создание обработчика события с помощью инспектора объектов

Таким образом, в самом простейшем случае задача программиста, при реализации логики приложения, состоит в том, чтобы реализовать код некоторого количества обработчиков событий и дополнительных подпрограмм.

После того как написан код приложения, проект должен быть скомпилирован и отлажен. Среда Delphi предоставляет возможность компиляции и сборки проекта с помощью команд из главного меню или панели инструментов. При отсутствии синтаксических ошибок в коде приложения оно может быть запущено под управлением интегрированного отладчика Delphi.

Интегрированный отладчик является мощнейшим средством среды Delphi. Он позволяет выполнять пошаговую отладку с заходом и обходом подпрограмм, просмотр переменных, изменение их значений в ходе отладки, установку точек останова, в том числе условных. Для отладки в Delphi используются также средства наблюдения за *стеком вызовов (call stack)* подпрограмм и методов объектов, *журнализации отладочных сообщений (event log)*, просмотра *состояния активных потоков приложения (thread status)* и загружаемых модулей. Для низкоуровневой отладки разработчик может воспользоваться таким отладочным инструментом как интерактивный дизассемблер, позволяющий наблюдать за ходом выполнения программы на уровне инструкций процессора и просматривать регистры и флаги ЦПУ.

Несмотря на то, что данный материал крайне кратко рассматривает только некоторые основные моменты использования среды Delphi, его достаточно для понимания сути основополагающих концепций построения программного обеспечения с использованием высокоразвитых современных интегрированных сред разработки. Важно отметить, что для современного программиста наиважнейшим фактором успешной реализации задач является не только знание синтаксиса используемого языка программирования и математических основ алгоритмизации, но и в еще большей степени владение технологиями программирования, умение работать в интегрированных средах разработки, а также владение методами использования компонентной базой современного программирования.

3. Практическая часть


3.1. Первое простейшее приложение

Создадим в Delphi простейшее приложение, которое демонстрирует возможности основных функциональных элементов интегрированной среды разработки, а также знакомит с простейшими компонентами библиотеки VCL.

Для более эффективной работы со средой необходимо запомнить несколько функциональных клавиш: F12 – переключение между конструктором форм и редактором кода; F11 – вызов инспектора объектов; F9 – сборка и последующий запуск приложения; CTRL-F9 – сборка проекта без запуска приложения.

Для создания проекта выполним следующую последовательность действий:

1. Создадим новый проект приложения. Для этого выберем команду New Application из меню File (New Application | File).

2. Перенесем на форму в дизайнера форм компонент Button  (кнопка – класс TButton) из палитры компонентов Standard.

Размещение компонентов на форме может быть выполнено двумя способами: либо двойным нажатием по пиктограмме компонента в палитре компонентов, либо с помощью одинарного нажатия по пиктограмме компонента и последующего указания щелчком левой кнопки мыши места расположения компонента на форме.

3. Выделим на форме компонент Button и в инспекторе объектов выберем свойство Caption (заголовок) и зададим его новое значение – «Добавить» (рис. 1.8).

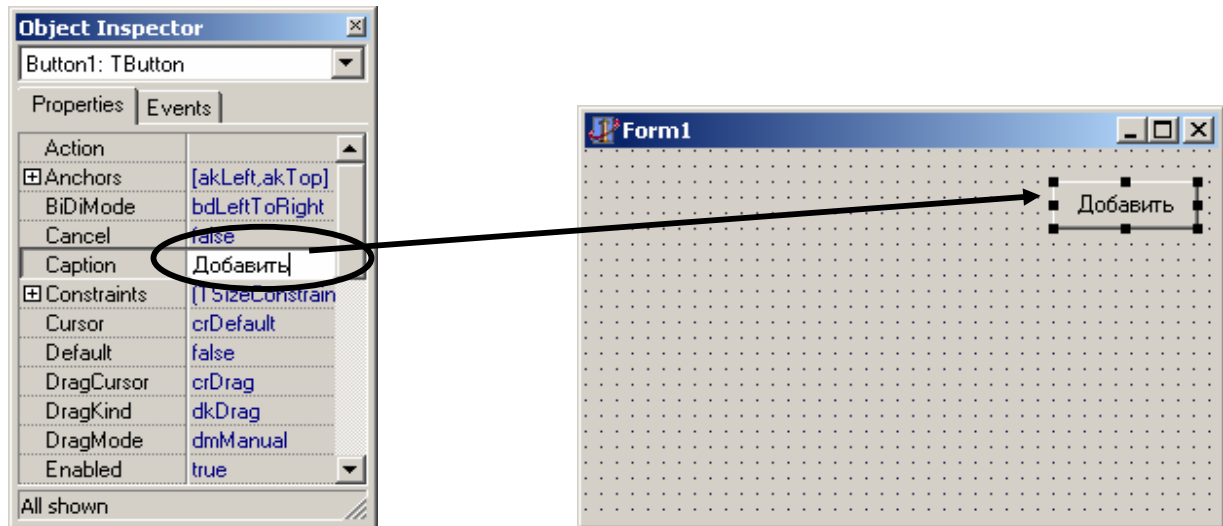



Рис. 1.8. Изменение заголовка кнопки

4. Зададим также новое значение для свойства Name (Имя) данной кнопки – «AddButton». Это свойство определяет идентификатор соответствующего объекта в программе, в данном случае объекта класса TButton. Следует обратить внимание, что неточное и несогласованное с исходным текстом программы воспроизведение значения этого свойства сделает невозможным успешную сборку проекта приложения.

5. Активизируем форму приложения с помощью одиночного нажатия левой кнопки мыши по клиентской области формы. При этом в инспекторе объектов отобразятся свойства и события компонента Form1 (форма – класс TForm). Зададим новые значения для свойств Caption – «Первая программа», Name – «MainForm» и BorderStyle (стиль границы окна) – «bsDialog».

6. Разместим на форме компонент ListBox  (список – класс TListBox) и установим новые значения для свойства Name этого компонента – «SubjectListBox».

7. Выберем в инспекторе объектов закладку Events и выделим на форме кнопку AddButton. Двойным нажатием мыши по пункту OnClick («при нажатии») перейдем в редактор кода и введем код внутри процедуры обработчика события OnClick. Событие OnClick возникает, когда пользователь нажимает и

отпускает левую кнопку мыши в области элемента управления, в данном случае кнопки типа Button. Исходный текст обработчика имеет следующий вид:

```
procedure TMainForm.AddButtonClick(Sender: TObject);
{
var
    i: Integer;
begin
    {
for i := 1 to 10 do
        begin
            SubjectListBox.Items.Add ('Строка ' + IntToStr (i + 1));
        end;
    }
end;
```

Здесь необходимо обратить внимание, на то, что каркас метода, т.е его заголовок и тело в секции реализации, а также прототип в секции интерфейса, (см. раздел 2.2 лабораторной работы № 3) будет сформирован редактором кода автоматически. В исходном тексте, который показан выше, эти фрагменты выделены наклонным шрифтом (**их вводить не нужно**), тогда как блоки кода, добавляемые программистом заключены в фигурные скобки.

8. Запустим приложение с помощью команды Run | Run или с помощью клавиши F9.

9. После запуска приложения нажмем на кнопку Добавить (не в конструкторе, а в запущенной программе). Результат работы данного приложения показан на рис. 1.9.

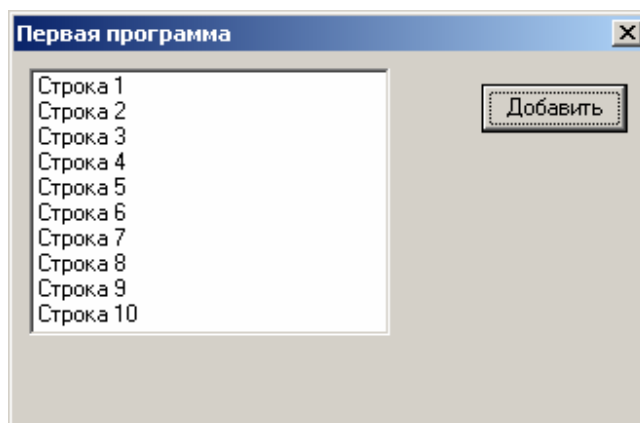



Рис. 1.9. Скомпилированное и запущенное приложение

3.2. Первое простейшее приложение (продолжение)

Используем ранее созданный проект для его дальнейшего развития и выполним над проектом следующие шаги:

1. Добавим на форму компонент Edit  (строка редактирования – класс TEdit). Установим в инспекторе объектов свойству Name этого компонента значение «SubjectEdit», а свойству Text этого компонента зададим значение пустой строки.

2. Создадим еще одну кнопку и определим для нее идентификатор (свойству Name) «DeleteButton», а свойству Caption зададим значение «Удалить».

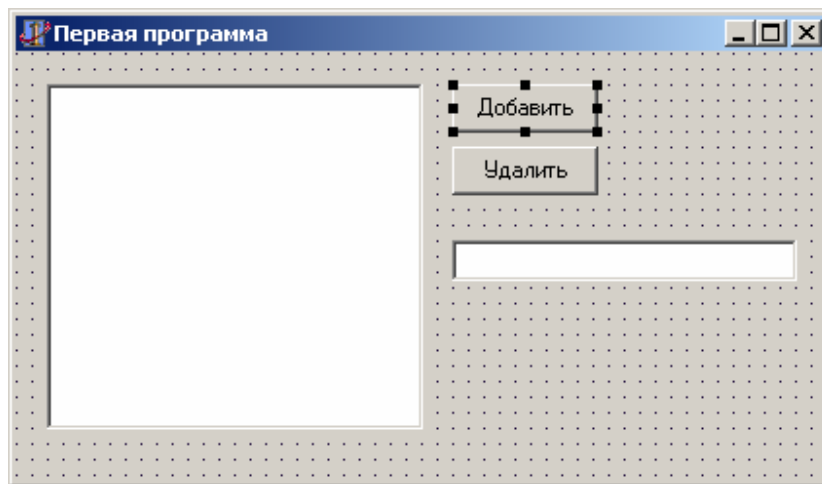



Рис. 1.10. Окно программы на этапе разработки

3. Определим обработчик события OnClick для кнопки DeleteButton, так как это было проделано в предыдущем примере. Этот обработчик имеет следующий вид:

```
procedure TMainForm.DeleteButtonClick(Sender: TObject);
begin
    {
        // Если пункт списка выделен
        if SubjectListBox.ItemIndex <> -1 then
            // мы его удалим
            SubjectListBox.Items.Delete(SubjectListBox.ItemIndex);
    }
end;
```

5. Выполним сборку приложения и протестируем его работу.

6. Далее завершив работу программы, перенесем на форму приложения еще один новый не визуальный компонент – PopupMenu (контекстное меню – класс TPopupMenu) .

Контекстное меню (contextual menu) – это широко распространенный элемент управления Windows, который представляет собой всплывающее меню, которое появляется над другим элементом управления при одиночном нажатии правой кнопкой мыши в клиентской области этого элемента и содержит, как правило, команды, дублирующие команды главного меню или кнопок из палитры инструментов программы.

Для создания пунктов любого меню, в том числе и контекстного, в арсенале средств среды Delphi имеется *дизайнер меню (Menu Designer)*, который позволяет легко добавлять, удалять и перемещать пункты меню, а также создавать подменю. Дизайнер меню вызывается двойным нажатием по компоненту PopupMenu или MainMenu (основное меню) в конструкторе форм.

7. Назначим идентификатор «ListBoxPopup» для компонента PopupMenu.

8. Чтобы контекстное меню «всплывало» над нужным элементом управления программы, компонент ListBoxPopup необходимо ассоциировать с этим элементом управления на этапе разработки.

Назначим всплывающее меню компоненту SubjectListBox (список). Для этого необходимо выделить на форме SubjectListBox и перейти в инспектор объектов. Затем для свойства PopupMenu задать значение «ListBoxPopup» из комбинированного списка (рис. 1.11).

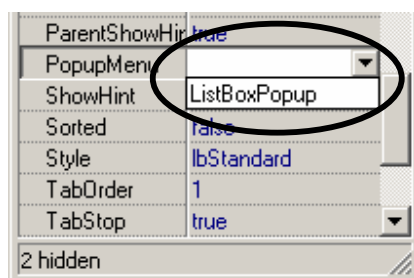


Рис. 1.11. Назначение контекстного меню элементу управления

9. Используя дизайнер меню, добавим в ListBoxPopup два пункта меню – «Добавить» и «Удалить». Для этого необходимо выделить в дизайнере пустой пункт меню и, перейдя в инспектор объектов, назначить свойству Caption новое

значение. После этого вновь выбрать следующий пустой пункт меню и также заполнить свойство Caption (рис. 1.12).

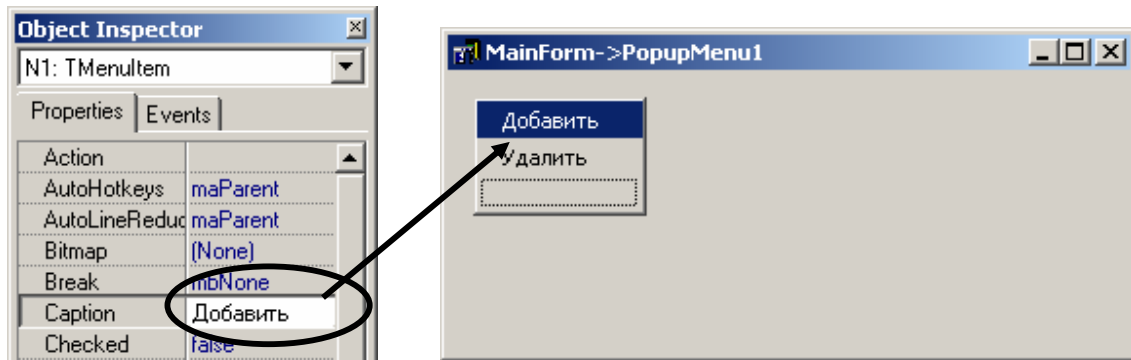


Рис. 1.12. Использование дизайнера меню

10. Задействуем пункты меню для выполнения команд. Для этого выберем нужный пункт меню и перейдем в закладку Events инспектора объектов. Выделим событие `OnClick`, ассоциированное с данным пунктом меню, и назначим нужный обработчик события выбрав его из комбинированного списка: для команды «Добавить» процедуру-обработчик `AddButtonClick`, а для команды «Удалить» – `DeleteButtonClick` (рис. 1.13).

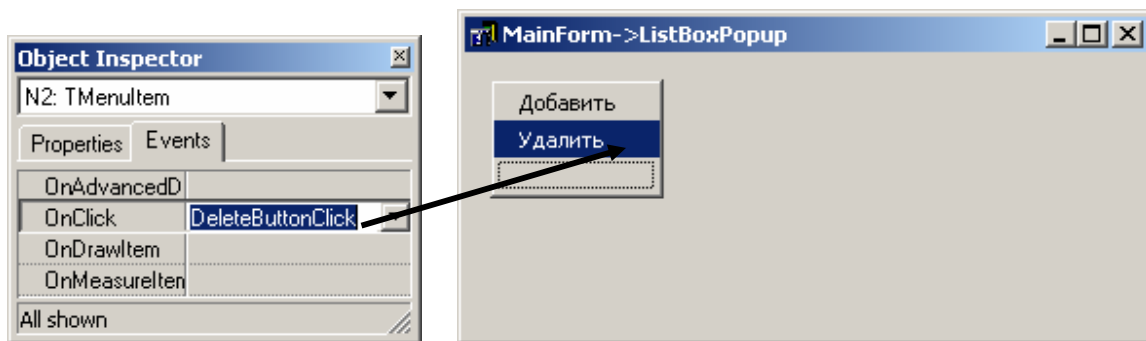


Рис. 1.13. Назначение пунктам меню обработчиков событий

Назначение командам меню уже имеющихся в программе обработчиков событий, которые назначены для кнопок, позволяет сократить объем кода и сделать код программы более читаемым.

11. Изменим обработчик события `OnClick` кнопки `AddButton` таким образом, чтобы выполнять добавление в список содержимого строки редактирования (компонент `SubjectEdit`):

```

procedure TMainForm.AddButtonClick(Sender: TObject);
begin
    {
        // Если содержимое строки редактирования не пустая строка
        if SubjectEdit.Text <> '' then
            // тогда добавляем содержимое в список
            SubjectListBox.Items.Add (SubjectEdit.Text);
    }
end;

```

12. Выполним сборку проекта и протестируем работу приложения.

3.3. Задания

1. Внимательно изучить теоретический материал из разделов 2.1 и 2.2.
2. Выполнить все этапы из практической части лабораторной работы.
3. Добавить еще одну командную кнопку с заголовком «Сортировка» и связать с кнопкой действия по сортировке содержимого списка SubjectListBox. Использовать для выполнения сортировки свойство Sorted класса TListBox.

4. Контрольные вопросы

1. Назовите базовые компоненты интегрированной среды разработки.
2. Что такое событие с точки зрения модели программирования языка Object Pascal ?
3. Какие типы свойств могут иметь компоненты VCL?

Лабораторная работа № 2

ПОНЯТИЕ О ПАРАДИГМАХ ПРОГРАММИРОВАНИЯ. ОСНОВЫ СИНТАКСИСА И СЕМАНТИКИ ЯЗЫКА ОБЪЕКТ PASCAL

1. Цель работы

Цель данной работы состоит в том, чтобы познакомить студентов с основами синтаксиса и семантики языка программирования Object Pascal. В рамках

данной работы студент должен изучить основные лексические элементы языка, суть работы простых и структурных операторов ОП, познакомиться с простыми (скалярными) типами данных, а также базовыми структурами данных – массивами и записями.

2. Основные теоретические сведения

2.1. Основные понятия и система принципов алгоритмических языков программирования

Слово «*программа*» имеет греческое происхождение и означает «объявление, распоряжение». В современном понимании, относящемся к программированию как разделу информатики, программа определяет план действий, подлежащих выполнению исполнителем, обычно вычислительной машиной (компьютером). Программа выглядит как конечная совокупность инструкций, каждая из которых приводит к выполнению некоторой элементарной операции над данными, хранящимися в памяти ВМ. Автоматизм исполнения достигается тем, что любая текущая команда, однозначно указывает на команду, которая должна быть выполнена следующей. Особенностью процесса исполнения команд является наличие команд *ветвления*, которые на основании и в зависимости от выполнения некоторых условий передают управление на ту или другую следующую инструкцию программы. Другой особенностью программ является возможность многократного, *циклического* выполнения некоторого числа инструкций. Эти особенности приводят к тому, что длина выполнения программы, т.е. количество выполненных инструкций, и время ее выполнения должны однозначно определяться некоторым набором входных данных.

Таким образом, программа является с математической точки зрения конечным объектом, которая заставляет вычислительную машину закономерно реагировать на потенциально бесконечное многообразие входных данных. Тем самым программа так же, как и теоремы и их доказательства, реализует всеобщность математических закономерностей.

Из вышесказанного следует, что языки для записи программ являются наряду теоретическим программированием одним из важнейших разделов математики и информатики.

Программы, исполняемые ВМ, должны формулироваться с использованием специальных знаковых систем, которые получили название *языков программирования* (*programming language, computer language*).

Язык программирования (ЯП) – это формальная знаковая система, служащая средством решения формализуемых вычислительных задач. Основное назначение ЯП – это формулирование программ, подлежащих исполнению вычислительной машиной. Программа, написанная на ЯП представляет собой своеобразную операционную (т.е. какие действия вычислительного характера производить) и информационную (над какими объектами данных производить эти действия) модель закономерности внешнего мира, причем программа фиксирует ее в точной и воспроизводимой форме.

Наиболее распространенным видом языков программирования являются алгоритмические языки. Среди важнейших понятий алгоритмических ЯП выделяются некоторые базовые конструкции:

- *описания* объектов программы (*definition, declaration*) – определяют имена или *идентификаторы* программных информационных и операционных сущностей, таких, как типы данных, классы, интерфейсы, шаблоны, скалярные переменные, объекты классов, подпрограммы и др. Например:

```
var
```

```
x: integer; // описание переменной на языке Pascal
```

- *выражения* (*expression*) – являются источником значений для программных информационных сущностей (объектов) и представляются в программе в виде комбинации имен объектов и операторов. Например:

```
// выражение является источником значения реального типа
```

```
x + y * sin(x) / x - z
```

- *операторы* (*statements*) – представляют в программе элементарные вычислительные действия (математические операции, логические операции, операции сравнения) или алгоритмические действия (операция приваривания, передача управления от одного оператора к другому, повторяющееся исполнение одного или нескольких операторов).

Например:

```
x := y + 10; // оператор присваивания(:=) в языке Pascal
```

Базовыми операторами, на основе которых могут быть сформированы законченные предложения, подлежащие исполнению в алгоритмических языках программирования, являются операторы *присваивания* значения (*assignment statement*), *передачи управления* (*transfer-of-control statement*), *вызова подпрограммы* (*subroutine call statement*) и *цикла* (*cycle statement*). Операторы других видов должны использоваться в контексте этих базовых операторов и не могут формировать исполняемые предложения программы.

В разных ЯП реализованы различные принципы или системы понятий, следуя которым строится законченная программа – это так называемая *парадигма программирования*. С точки зрения структурирования программы, т.е. разделения ее на составные части, в современном программировании выделяют несколько парадигм программирования:

- *неструктурного* программирования (*chaos paradigm*);
- *структурного* программирования (*structured*);
- *процедурного* программирования (*procedure-oriented*);
- *объектно-ориентированного* программирования (*object-oriented*).

Неструктурное программирование отличается тем, что программа представляет собой последовательность алгоритмических конструкций, никак не разделяемых по принципу решаемых алгоритмических задач. Отличительной чертой неструктурной программы является отсутствие подпрограмм и наличие большого числа безусловных передач управления от одного фрагмента программы к другому (безусловных переходов).

Структурное программирование доказывает избыточность использования безусловных переходов и то, что алгоритм любой сложности может быть построен на основе линейной комбинации трех базовых алгоритмических структур – *линейной*, *ветвления* и *циклической*. *Алгоритмической структурой* называют способ представления последовательности исполнения предложений программы. В линейной алгоритмической структуре предложения программы исполняются последовательно одно за другим без изменения естественного порядка следования предложений (рис. 2.1 а). В структуре ветвления присутствует оператор условной передачи управления, который позволяет изменить линейный ход исполнения программы. Этот оператор передает управление одному или другому предложению программы, в зависимости от выполнения или невыполнения некоторого условия (рис. 2.1 б). Наконец, структура повторения (рис. 2.1 в) или циклическая структура разрешает повторное исполнение одного или нескольких предложений программы, до тех пор пока выполняется или не

выполняется некоторое условие в программе. Структура повторения может быть реализована комбинацией операторов условной и безусловной передачи управления. Весьма важным здесь является то, что реализация принципов структурного программирования становится возможной в алгоритмическом языке только тогда, когда в нем имеется самостоятельный оператор цикла, поскольку использование оператора безусловной передачи управления нарушает эти принципы.

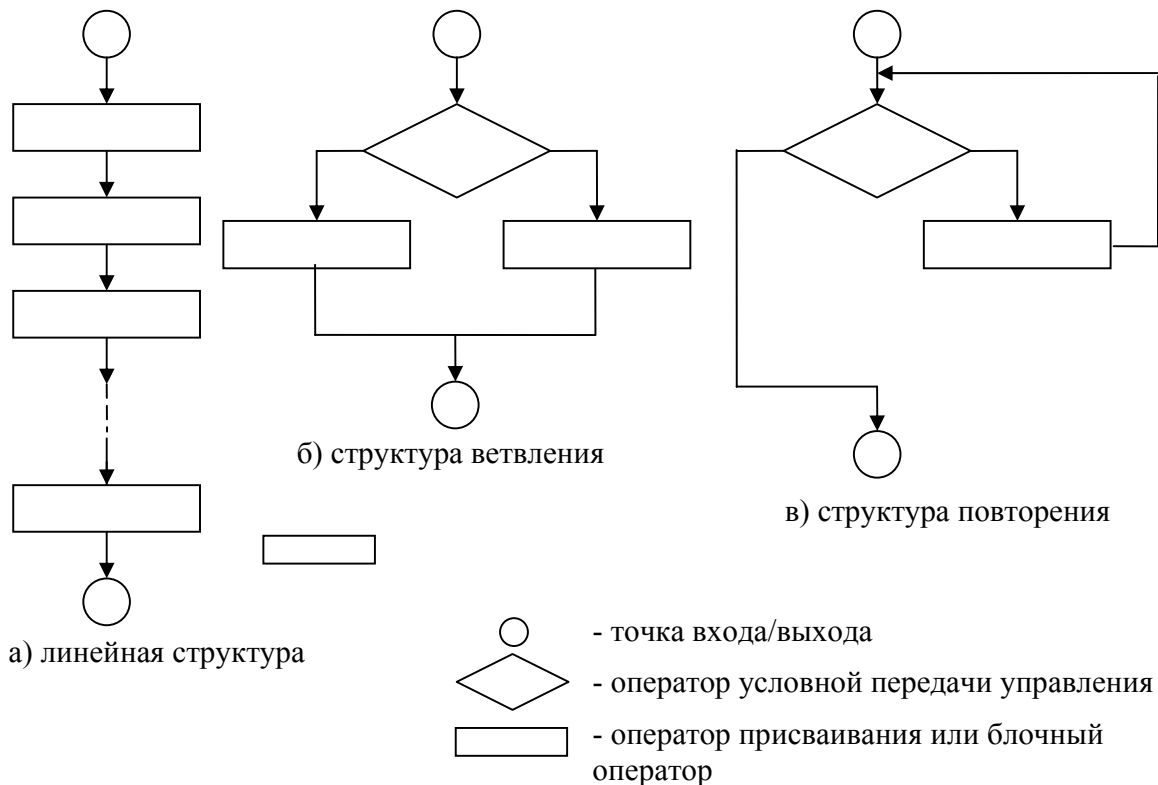


Рис. 2.1. Типы алгоритмических структур

Парадигма процедурного программирования вводит понятие *подпрограммы* (*subroutine*) как именованной последовательности операторов, совокупность действий которых направлена на решение в общем случае небольшой, но конкретной алгоритмической задачи. Именно в процедурном программировании была впервые реализована идея *повторного использования кода* (*code reusing*), основной смысл которой заключается в многократном использовании кода подпрограммы в рамках одного или нескольких программных проектов. Реализация в алгоритмических ЯП принципов процедурного программирования

ния стала качественным прорывом в технологии программирования, основными результатами которого было увеличение производительности труда программиста, относительного уменьшения объема исходного кода и повышение его читабельности. Идеи процедурного подхода к программированию легли в основу наиболее развитой на сегодняшний день технологии программирования – объектно-ориентированного программирования и соответствующих ЯП.

Основным недостатком процедурной парадигмы было то, что данные и подпрограммы их обработки были логически отделены друг от друга, что создавало определенные проблемы при построении программ высокой степени сложности. Следующим шагом в развитии ЯП стало развитие представлений об *абстрактных типах данных (abstract data type, АДТ)*. АДТ в отличие от скалярных типов (целых чисел, строк, чисел с плавающей точкой) являются не только и не столько простым контейнером для данных на физическом уровне, которому сопоставлены одна или несколько ячеек памяти, но наряду с данными содержит определенные методы в виде подпрограмм для обработки этих данных. Такие АДТ получили название *классов*. Класс сродни обычным скалярным типам и на его основе также создаются особые переменные, называемые *объектами*.

2.2. Лексические единицы языков программирования

Основой любого естественного или искусственного языка является алфавит – набор допустимых в языке элементарных знаков (букв, цифр и служебных знаков). Знаки могут объединяться в слова или *лексемы (lexeme)*, рассматриваемые в тексте (программе) как неделимые части, имеющие определенный смысл. Словарный состав языка вместе с описанием способов их представления составляют *лексику* языка. Слова в языке объединяются в более сложные конструкции – *предложения (clause)*. В языках программирования предложения представляются в простейшем случае операторами. Предложения строятся из слов и более простых предложений по правилам синтаксиса. *Синтаксис (syntax)* языка представляет собой описание правильных предложений, тогда как описание смысла предложений, т.е. значений слов и их внутренних связей, составляют *семантику (semantics)* языка.

Любой язык программирования имеет схожие лексические единицы, к которым принято относить следующие: идентификаторы, ключевые слова, литералы, операторы, разделители.

Идентификатором (identifier) является последовательность латинских символов (в некоторых случаях и символов кириллицы), в том числе и цифр, начинающаяся с буквы или символа подчеркивания, который именуется некоторый объект в программе, уникально идентифицируя его в пределах некоторого блока программы. Идентификатор не должен содержать знака пробела и некоторых других служебных символов. В некоторых языках программирования, таких, как C/C++, Java, C# при именовании объектов программы имеет значение регистр символов, тогда как в Object Pascal и Visual Basic регистр символов не имеет значения.

Ниже приведены примеры правильных идентификаторов:

```
AppPath, x, y, _lerosoft, matrix1, matrix2
```

Неправильными идентификаторами являются следующие:

```
4sale      // начинается с цифры
#23        // не начинается с буквы
Str$       // содержит неразрешенный знак
```

При именовании объектов в программе, как правило, придерживаются определенного стиля и *соглашения о именовании (naming convention)*, т.е. правил, следуя которым строятся идентификаторы, в рамках одного или нескольких программных проектов. Принято, хотя это и необязательно, идентификаторы начинать с префикса, за которым следует сокращенное англоязычное выражение (не содержащее пробелов), отражающее смысл объекта. Например:

```
iPay, lngShift, bFlag, objActiveSheet, objWorkBook, hAppHandle
```

Для сокращения слишком длинных имен можно исключать из слов, составляющих идентификатор, гласные буквы, поскольку в основном только согласные буквы определяют смысл слова. Например:

```
bFlg, objActvSheet, objWrkBook, hAppHndl
```

Ключевыми словами (keyword) являются лексические единицы, имеющие вид постоянных идентификаторов. Эти слова зарезервированы в языке про-

граммирования для специального использования и не могут применяться в качестве идентификаторов пользовательских объектов программы, таких, как переменные, подпрограммы и т.п. С помощью отдельных ключевых слов или их комбинаций представляются основные алгоритмические конструкции, т.е. структурные и простые операторы. Каждый язык программирования имеет собственный уникальный набор ключевых слов. Как правило, ключевое слово во всех языках программирования представляется короткими англоязычным словом или сокращением, отражающим смысл представляемого оператора. К распространенным ключевым словам многих языков программирования можно отнести, например, следующие:

```
for, if, then, do, while, begin, end, class, type
```

Операторы позволяют выразить операционную модель языка программирования. С помощью операторов выражаются не объекты, а действия над ними. Например:

```
x := 1;           // оператор приваривания
x := x shl 6;    // shl оператор логического сдвига влево
```

Литералами (literal) являются такие символы, которые в рамках принятой интерпретации определяют все свои особенности, в том числе и собственное значение. Литералы подразделяются на числовые, с плавающей точкой и строки.

Числовые литералы могут быть в виде десятичных, шестнадцатеричных, а в некоторых языках программирования двоичных и восьмеричных чисел. Например:

```
10, 10, 120, 1234 // десятичные числа
0xA, $0A, &H0A   // шестнадцатеричные в C, Pascal и Basic
```

Литералы с плавающей точкой представляют вещественные числа. Они состоят из целой части, точки и дробной части. Например:

```
0.1345, 23.45, 3.14159265358979, 1.1
```

Числа с плавающей точкой могут быть представлены также в виде целого числа целой части и показателя, имеющего также вид целого числа со знаком, перед которым находится прописная или строчная буква E. Например:

```
1345E-1 2345E-2, 34159265358979E-15, 1E-1
```

Наконец, строчные литералы или *строки* (*string*) представляют собой последовательность символов, заключенных в кавычки (в языке Pascal в апострофы). Строчный литерал интерпретируется как массив символов. Например:

```
'Hello world', 'Bye world', "Привет мир"
```

В языках программирования помимо разнообразных лексических единиц имеются вспомогательные синтаксические элементы, такие, как разделители, комментарии и директивы.

Разделители (*separator*) – это особый вид синтаксических элементов, реализующих разделение нескольких операторов («;») или других элементов языковых конструкций («», «()», «[]»).

```
// ";" - разделяет операторы в ОР, С/С+, С# и Java
x := x + 1;
// "(" - разделяют части выражения
if (x > 10) and (z = 0) then Exit;
// "," - разделяют элементы множества
if x in [10, 12, 13, 16] then x := x * 3;
```

К специфическим лексическим элементам языков программирования можно отнести также *директивы* (*directives*), которые не являются исполняемыми, но позволяют управлять процессом трансляции программы и обеспечивают взаимодействие программы со средой разработки.

Непременным атрибутом всех современных языков программирования является *комментарии* (*comment*). Комментарий не является лексической единицей, не интерпретируется языком программирования и служит в качестве средства документирования программы. Комментарии обычно объясняют сложные или нестандартные фрагменты кода, смысл и назначение объектов программы, описывают проблему и подходы к ее решению. Начало коммента-

рия в программе отмечается специальными символами, за которыми следует текст комментария. Например:

```
` комментарий в языке Basic
// комментарий открытого типа в C/C++ и Pascal
/* комментарий закрытого типа в C/C++ */
{комментарий закрытого типа в Pascal}
```

2.3. Переменные и их декларация. Типы данных. Операция присваивания и чтения значения переменной

Базовой концепцией всех языков программирования является концепция памяти, суть которой заключается в возможности хранения данных в особых объектах программы. Эти объекты получили название *переменных (variables)*. Переменная используется для временного хранения некоторого значения в течение исполнения программы. Это значение может изменяться в ходе работы программы.

Каждой переменной в физической памяти распределенной для программы сопоставлены одна или несколько ячеек памяти, в зависимости от типа данных, который имеет переменная. Например, целая 32-битная переменная $x = 23435345$ с адресом n будет занимать четыре смежных ячейки памяти (рис. 2.2).

$n + 3$	$n + 3$	$n + 2$	$n + 1$	n	$n - 1$
	00000001	01100101	10011000	01010001	

Рис. 2.2. Физическое представление переменной в физической памяти

Переменной сопоставлено имя, т.е. идентификатор, который используется в программе для ссылки на значение, хранимое в этой переменной. Наряду с идентификатором каждой переменной должен быть сопоставлен определенный тип данных. *Тип данных (data type)* – это характеристика переменной, определяющая возможные значения, которые может принимать переменная, а также операции, допустимые над ней. К фундаментальным простейшим типам, которые также называются *скалярными типами (scalar type)*, относятся целые знаковые и беззнаковые типы, типы с плавающей точкой различной точности (по количеству значащих верных разрядов после десятичной точки) и строчные ти-

пы. Например, беззнаковая двухбайтная переменная целого типа может принимать значения от 0 до 65565 и для нее имеют смысл такие операции, как умножение, деление, сложение и вычитание, операции сдвига и битовые логические операции. В свою очередь переменная строчного типа занимает в памяти объем соответствующий числу символов этой строки, и для нее имеет смысл из выше перечисленных операций только операция сложения (конкатенации) строк.

Object Pascal поддерживает широкий набор predefined типов, которые распознаются компилятором языка автоматически, без их предварительной декларации (объявления). Типы языка Object Pascal подразделяются на *простые (simple)*, *строчные (string)*, *указатели (pointer)*, *процедурные (procedural)* и *вариантные (variant)*. К простым типам принадлежат *порядковые типы (ordinal)*, которые включают *целые* (табл. 2.1) и *реальные* (табл. 2.2) типы.

Таблица 2.1. Целые типы языка ОР

Идентификатор типа	Диапазон	Разрядность
Integer	-2147483648..2147483647	знаковое 32-битное
Cardinal	0..4294967295	беззнаковое 32-битное
Shortint	-128..127	знаковое 8-битное
Smallint	-32768..32767	знаковое 16-битное
Longint	-2147483648..2147483647	знаковое 32-битное
Int64	$-2^{63}..2^{63}-1$	знаковое 64-битное
Byte	0..255	беззнаковое 8-битное
Word	0..65535	беззнаковое 16-битное
Longword	0..4294967295	беззнаковое 32-битное

Таблица 2.2. Реальные типы языка ОР

Идентификатор типа	Диапазон	Разрядность (в байтах)
Real48	$2.9 \cdot 10^{-39} .. 1.7 \cdot 10^{38}$	6
Single	$1.5 \cdot 10^{-45} .. 3.4 \cdot 10^{38}$	4
Double	$5.0 \cdot 10^{-324} .. 1.7 \cdot 10^{308}$	8
Extended	$3.6 \cdot 10^{-4951} .. 1.1 \cdot 10^{4932}$	10

Comp	$-2^{63}+1 \dots 2^{63}-1$	8
Currency	$-922337203685477.5808 \dots$ 922337203685477.5807	8
Real	$5.0 \cdot 10^{-324} \dots 1.7 \cdot 10^{308}$	8

К порядковым типам относится также *булевский* тип данных (*boolean*). Булевский тип данных в ОР представлен типами BOOLEAN (1 байт), BYTEBOOL (1 байт), WORDBOOL (2 байта) и LONGBOOL (4 байта). Переменная булевского типа может принимать только два значения: «истина» (TRUE) и «ложь» (FALSE). Например:

```
var OK: Boolean
...
if X <> 0 then OK := True;
```

Строчные типы служат для объявления переменных, которые хранят последовательности символов. В ОР имеется несколько predefined строчных типов (табл. 2.3).

Таблица 2.3. Строчные типы языка ОР

Идентификатор типа	Максимальное число символов	Кодировка
ShortString	255	ANSI
AnsiString	$\sim 2^{31}$	ANSI
WideString	$\sim 2^{30}$	UNICODE

В ОР предусмотрено ключевое слово *string*, которое служит идентификатором типа и в зависимости от настроек компилятора может задавать переменной тип SHORTSTRING либо ANSISTRING.

К структурным типам относятся *наборы* (*set*), *массивы* (*array*), *записи* (*record*), *файлы* (*file*) и *классы* (*class*).

Значение переменной любого типа может изменяться в ходе работы программы. Основополагающими действиями, которые определены над всеми переменными, являются операции присваивания значения и возвращения значе-

ния. Этим операциям сопоставлен оператор присваивания, который обозначается в ОР в виде «:=». Оператор присваивания имеет следующую форму:

```
<variable> := <expression>;
```

где <variable> – идентификатор переменной любого типа, а <expression> – любое выражение, совместимое по типу возвращаемого значения. Например:

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
I := Sqr(J) - I * K;
```

Семантика оператора присваивания такова, что если переменная находится слева от него, то этой переменной *присваивается* значение, которое *возвращает* слева от себя выражение или переменная стоящая, справа от оператора присваивания. Переменная, стоящая справа от оператора присваивания или участвующая в выражении, всегда только возвращает значение.

Язык ОР является *строго типизированным* языком (*strong typing*), т.е. каждая переменная должна иметь определенный тип данных и объявлена до своего первого упоминания в программе или подпрограмме. Базовый синтаксис декларации переменной имеет следующий вид:

```
var <identifierList>: <type>;
```

где <identifierList> – это список идентификаторов, разделенных запятыми, а <type> – любой predefined или ранее задекларированный пользовательский тип. Например:

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;    // диапазонный тип
  Okay: Boolean;
```

Переменные, декларируемые внутри процедуры или функции, называются локальными, а вне процедуры или функции – уровня модуля. Переменные уровня модуля могут получать начальное значение при декларации:

```
var I: Integer = 7;
```

2.4. Структурные операторы. Операторный блок

Основным признаком структурного оператора является возможность размещать внутри себя другие операторы. Структурные операторы реализуют базовые управляющие алгоритмические структуры: последовательность, ветвление и повторение.

Простейшим структурным оператором является *составной* оператор (*compound*) или так называемые *операторные скобки* (*statement brackets*). В ОР этот оператор реализован с помощью ключевых слов языка BEGIN и END, между которыми заключаются любые другие операторы, разделяемые символом «;». Например:

```
begin
    Z := X;
    X := Y;
    Y := Z;
end;
```

Структурный оператор является неотъемлемой частью практически любой синтаксической конструкции ОР: операторного блока главного модуля программы, функции, процедуры или структурного оператора. Например:

```
begin
|
|   I := SomeConstant;
|   while I > 0 do
|       begin
|           ...
|           I := I - 1;
|       end;
|   end;
end;
```

Другим важнейшим структурным оператором ОП является оператор ветвления, который имеет две формы IF ... THEN и IF ... THEN ... ELSE. Синтаксис оператора IF ... THEN имеет следующий вид:

```
if <expression> then <statement>
```

где <expression> – выражение, возвращающее значение булевского типа, а <statement> – любой простой или составной оператор. Если <expression> возвращает значение TRUE, то оператор <statement> будет выполнен, в противном случае – не выполнен.

Синтаксис оператора IF ... THEN ... ELSE имеет следующий вид:

```
if <expression> then <statement1> else <statement2>
```

В этом операторе, если <expression> возвращает значение TRUE, то будет выполнен оператор <statement1>, в противном случае – <statement2>.

Например:

```
if J = 0 then
    Exit          // ВНИМАНИЕ! Разделитель не нужен
else
    Result := I / J;
```

ИЛИ

```
if J <> 0 then
begin
    Result := I/J;
    Count := Count + 1;
end          // ВНИМАНИЕ! Разделитель также не нужен
else if Count = Last then
    Done := True
else
    Exit;
```


Наконец, последним из базовых структурных операторов ОП является оператор повторения или цикла. ОП имеет три вида циклических операторов: оператор REPEAT, оператор WHILE и оператор FOR. Важной особенностью всех циклических операторов является возможность досрочного выхода из цикла с помощью специальной процедуры Break и пропуск очередного шага (итерации) цикла с помощью другой процедуры – Continue. Необходимо отметить, что в других современных языках программирования средства завершения и пропуска итерации цикла реализованы в виде операторов языка, (C/C++, Java, VB), а не подпрограмм как в ОП.

Операторы REPEAT и WHILE относятся к типу условных циклов, продолжение или завершение которых зависит от состояния **выражения возвращающего значение булевского типа**.

Оператор REPEAT имеет следующий синтаксис:

```
repeat
    <statement1>;
    ...;
    <statementn>;
until <expression>
```

где <expression> – выражение возвращающее булевское значение. Цикл выполняется, до тех пор пока <expression> возвращает значение FALSE. Например:

```
repeat
    K := I mod J;
    I := J;
    J := K;
until J = 0;

repeat
    Write(' Введите значение (0..9): ');
    Readln(I);
until (I >= 0) and (I <= 9);
```

Цикл на основе REPEAT получил название цикла с постусловием. Тело такого цикла всегда выполняется хотя бы один раз.

Оператор WHILE подобен оператору REPEAT, за исключением того, что проверка условия окончания цикла осуществляется в нем до начала тела цикла. Оператор имеет следующий синтаксис:

```
while <expression> do <statement>
```

Если выражение <expression> возвращает ложное значение, то цикл завершается. Данный тип цикла также является условным и называется циклом с предусловием.

```
while Data[I] <> X do I := I + 1;
```

```
while I > 0 do
```

```
begin
```

```
    if Odd(I) then Z := Z * X; // Odd – проверка на нечетность
```

```
    I := I div 2; // div – оператор целочисленного деления
```

```
    X := Sqr(X);
```

```
end;
```

Оператор FOR в отличие от операторов REPEAT и WHILE требует явного указания числа итераций цикла. Синтаксис этого оператора имеет следующий вид:

```
for <counter> := <initialValue> to <finalValue> do <statement>
```

или

```
for <counter> := <initialValue> downto <finalValue> do <statement>
```

где <counter> – локальная переменная порядкового типа, задекларированная в блоке, который содержит оператор FOR, <initialValue> и <finalValue> – значения диапазона изменения переменной <counter>, <statement> – простой или структурный оператор, который *не изменяет* значение переменной <counter>. На каждом шаге цикла значение переменной <counter> увеличивается на 1 от начального значения <initialValue>. Когда <counter> достигает значения <finalValue>, цикл завершается.

При использовании оператора в форме FOR...DOWNTO происходит обратное измерение переменной <counter> с шагом -1.

```

for I := 2 to 63 do
    if Data[I] > Max then
        Max := Data[I];

for I := 1 to 10 do
    for J := 1 to 10 do
    begin
        X := 0;
        for K := 1 to 10 do
            X := X + Mat1[I, K] * Mat2[K, J];
        Mat[I, J] := X;
    end;

```

2.5. Массивы

Массив является базовой структурой данных во всех ЯП, которая поддерживается на уровне встроенных примитивов языка. Массив является в общем случае линейной статической структурой данных, которая может быть представлена как последовательность ячеек памяти, с которыми ассоциировано одинаковое имя и сопоставлен одинаковый тип данных. Для ссылки на одну из ячеек такой структуры необходимо указать имя массива и номер соответствующей ячейки. Такой номер называется индексом элемента массива.

Основным назначением такой структуры данных, как массив в ЯП, является то, что с помощью массивов в значительной степени упрощается обработка потоков данных в программе, код программы делается более компактным, а большинство операций над данными, структурированными в виде массива, формализуются в виде циклических алгоритмов. Массивы могут быть использованы для реализации таких линейных структур данных, как списки, стеки, очереди, а также нелинейных, таких, например, как графы различных типов представляя матрицы смежности или таблицы ребер графов.

Каждый массив имеет определенную верхнюю и нижнюю границы, и индексы элементов массива должны находиться в этих четко заданных границах. Во всех ЯП пространство памяти под массив выделяется при его определении,

даже если он не содержит ни одного элемента. В этом и проявляется статичная природа массива, делая его в некоторых случаях непригодным для решения задач, связанных с обработкой больших объемов данных переменной длины. Однако массив идеально подходит для работы с данными небольшой длины.

Тип данных элементов массива может произвольным, но, как уже отмечалось, одинаковым для всех. Элементы массива могут быть скалярными величинами или иметь структурный или объектный тип (указатель).

Так же, как в языке Visual Basic, на уровне синтаксиса языка ОР поддерживается два вида массивов: статические и динамические. Однако ОР имеет важное семантическое отличие от других ЯП, которое состоит в том, что массив является переменной особого типа – типа «массив». Тип статического массива объявляется с помощью ключевого слова ARRAY:

```
array [<lowerindex_1>..<upperindex_1>, . . . ] of <basetype>
```

где <lowerindex_1> определяет нижнюю индекса первой размерности массива, <upperindex_1> – верхнюю границу той же размерности, а <basetype> указывает на идентификатор встроенного или пользовательского типа, который будут принимать все элементы объявляемого массива.

Размер массива в ОР ограничен объемом памяти 2 Гб. Значения нижней и верхней границы индекса соответствуют целому знаковому типу INTEGER. Простейшая переменная типа одномерный массив может быть определена следующим образом:

```
var MyArray: array[1..100] of Byte;
```

Такой массив содержит 100 элементов каждый из которых имеет байтный беззнаковый тип. При объявлении такого массива его элементы имеют неопределенное значение. В общем случае всякая неинициализированная переменная в ОР также принимает случайное значение, при этом говорят об инициализации «мусором». Такой тип инициализации объектов программы характерен также для таких языков программирования как C/C++, Java и C#.

Многомерные массивы в ОР могут быть объявлены двумя способами. В первом случае перечисляются диапазоны индексов всех размерностей:

```
var MyMatrix: array[1..100, 1..100] of Real;
```

во втором случае может быть объявлен массив массивов

```
var MyMatrix: array[1..100] of array [1..100] of Real;
```

Для доступа к элементам массива здесь используется следующий синтаксис:

```
MyArray[33] := 10;
MyMatrix[12, 33] := 12.34;
MyMatrix[15][36] := 1412.3464;
```

Стандартные функции `OP Low` и `High` позволяют найти значения нижней и верхней границ первой размерности переменной-массива.

В современном программировании весьма часто используются динамические массивы, не имеющие фиксированного размера, память для которых выделяется динамически. Тип динамического одномерного массива определяется следующим образом:

```
array of <basetype>
```

а соответствующая переменная типа динамический массив:

```
var MyFlexibleArray: array of Real;
```

Объявление такой переменной означает задание начального адреса будущего динамического массива. Для создания самого массива и выделения для него пространства в области динамической памяти в исполняемом блоке программы или подпрограммы должна быть вызвана функция `SetLength`, аргументами которой является имя массива и число выделяемых ячеек для массива:

```
var
    A, B: array of Integer;
begin
    SetLength(A, 1);
    SetLength(B, 1);
```

```

A[0] := 2;
B[0] := 2;
end;

```

Нижняя граница индекса динамического массива всегда равна нулю. Для декларации многомерных динамических многомерных массивов используется повторяющаяся конструкция ARRAY OF. К примеру, определение самостоятельного типа динамический массив и переменной этого типа выполняется следующим образом:

```

type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;

```

При этом декларируется тип двумерного динамического массива и соответствующая переменная. Для создания массива и выделения памяти под его ячейки так же, как и в случае одномерных массивов, используется функция SetLength:

```

SetLength(Msgs, I, J);

```

3. Практическая часть

3.1. Примеры использования циклических операторов в ОР

Создадим новый программный проект, в котором на законченных примерах изучим применение базовых алгоритмических структур языка ОР. Рассмотрим также практику использования некоторых новых компонентов библиотеки VCL.

1. Закроем с помощью команды Close All | File проект, созданный по умолчанию при загрузке среды. Выберем команду New... | File, выполнение которой приводит к открытию диалогового окна репозитория объектов.

Репозиторий объектов является хранилищем шаблонов различных типов проектов, которые поддерживаются средой разработки, а также шаблонов различных стандартных диалоговых окон, компонентов, модулей данных, отчетов, ActiveX-компонентов и др.

Среда Delphi позволяет разработчику создавать различные типы приложений на платформе Win32, такие, как обычное оконное приложение Windows (Application), консольное приложение (Console Application), динамическую библиотеку Windows (DLL), контрольную панель Windows (Control Panel Module), приложение для Web сервера (Web Server Application), а также службу Windows (Service Application). Репозиторий объектов позволяет разработчику быстро сгенерировать каркас приложения того или иного типа, сокращая для него объем кодирования. Помимо автоматизированного создания каркаса приложений репозиторий позволяет добавлять в проект уже созданного приложения модули различных типов, такие, как простой модуль (Unit), модуль контейнера-данных (Data Module), модуль класса потока (Thread Object), наконец модуль формы (Form). Таким образом, репозиторий объектов является еще одним мощным инструментом среды разработки, который существенным образом позволяет облегчить труд разработчика.

Итак, используя репозиторий объектов, создадим новый проект, выбрав двойным нажатием левой кнопки мыши пункт Application из закладки New (рис. 2.3).

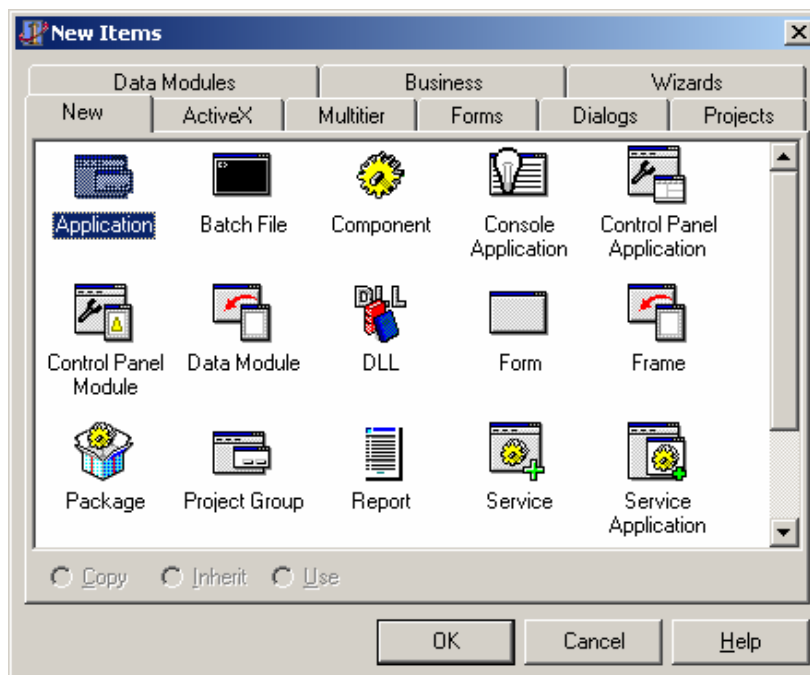




Рис. 2.3. Внешний вид репозитория объектов среды Delphi

2. В инспекторе объектов назначим новое имя (свойство Name) форме приложения – «MainForm», а также установим заголовок формы – «Изучение языка ОР» (свойство Caption).

3. В палитре компонентов Standard выберем компонент MainMenu, представляющий главное меню приложения (1)  (класс TMainMenu), и перенесем его на форму приложения. Изменим имя по умолчанию (свойство Name) этого компонента на «MainFormMenu». Компонент типа MainMenu позволяет создавать в приложении главное меню, размещаемое ниже заголовка окна приложения. Для формирования элементов главного меню, также как для работы с пунктами контекстного меню в Delphi применяется редактор (дизайнер) меню, основные моменты работы с которым были рассмотрены в первой лабораторной работе.

4. Добавим на форму компонент Мемо (2)  (класс TМемо), который представляет собой простой многострочный редактор, подобный тому, который используется в редакторе Notepad. Зададим этому компоненту имя «MessageView». Затем в инспекторе объектов выделим свойство Align (выравнивать) и выбрав в комбинированном списке из нескольких predefined установим ему назначение «alClient». В результате компонент Мемо автоматически «распространится» на всю клиентскую область окна-контейнера (рис. 2.4) и будет изменять свои размеры в соответствии с размерами этого окна.

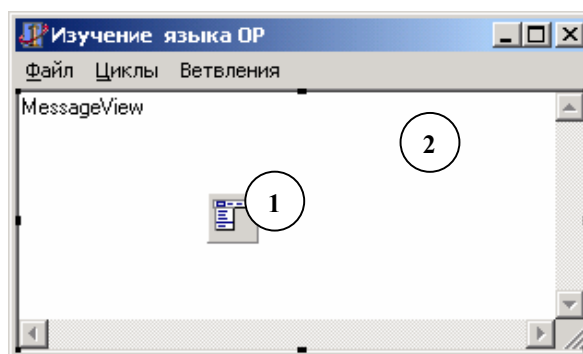



Рис. 2.4. Использование свойства Align

Для создания сложных многокомпонентных интерфейсов среда Delphi и ее компонентная библиотека реализует важное свойство визуальных компонентов – так называемое *выравнивание (align)*. Это свойство задает привязку компонента к границе материнского окна: слева (alLeft), справа (alRight), сверху

(alTop), снизу (alBottom) или по ширине клиентской области (alClient). Использование этой возможности существенным образом упрощает построение пользовательского интерфейса поскольку, выровненный таким образом компонент сохраняет свое выравнивание и позицию при масштабировании окна, **изменяя автоматически соответствующий типу привязки собственный размер**. При наличии нескольких компонентов на форме они могут быть выровнены по отношению к границам друг друга.

5. Для удаления надписи «MessageView» в многострочном редакторе необходимо, используя свойство Lines,  открыть редактор String List editor и удалить соответствующую строку.

6. Используя редактор меню (двойное нажатие левой кнопкой мыши по компоненту меню), создадим элемент меню с заголовком «&Файл» и элемент подменю данного меню с заголовком «&Выход».

7. Назначим обработчик события OnClick для элемента меню «Выход» и добавим в процедуру обработки следующий код (выделен фигурной скобкой):

```
procedure TMainForm.N2Click(Sender: TObject);
begin
  { Application.Terminate; // завершение работы приложения }
end;
```

8. Выполним сборку проекта и запустим приложение используя команду Run | Run. Протестируем приложение.

9. Создадим новое меню и команды, как показано на рис. 2.5.

Эти меню используем для обработчиков, которые демонстрируют работу циклических операторов языка ОР.

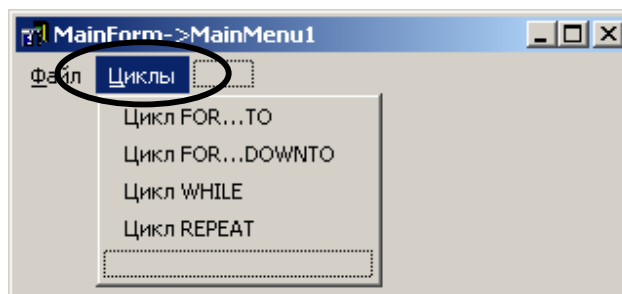


Рис. 2.5. Команды меню «Циклы» в дизайнера меню

Следует еще раз напомнить, что для задания обработчика события компонента необходимо:

- выделить на форме компонент или, если это элемент меню, выбрать его в дизайнере меню;
- в инспекторе объектов перейти на закладку Events;
- путем двойного нажатия левой кнопкой мыши на пункте соответствующего события в списке создать каркас процедуры-обработчика;
- ввести исполняемый код, соответствующий обработчику.

Каркас процедуры-обработчика создается **автоматически** и в простейшем случае выглядит следующим образом:

```
// обработчик нажатия кнопки мыши по кнопке
procedure TMainForm.MyButtonClick(Sender: TObject); // формируется Delphi
begin
    // ВНИМАНИЕ! здесь размещается пользовательский код
end;
```

Таким образом, для команды меню «Цикл FOR...TO» необходимо назначить следующий обработчик:

```
procedure TMainForm.FORTO1Click(Sender: TObject);
{
    var
        // для цикла for счетчик только порядкового типа
        // переменная типа Byte
        i: Byte;
begin
    {
        MessageView.Lines.Clear; // очистка содержимого редактора
        for i := 0 to 99 do
            begin
                // Добавляем строку в коллекцию строк редактора
                MessageView.Lines.Add (IntToStr(i) + Chr(9) + IntToStr(i + 100));
                // Добавляем пустую строку
                MessageView.Lines.Add('');
            end;
        end;
    }
end;
```

Для команды меню «Цикл FOR...DOWNTO»:

```

procedure TMainForm.FORDOWNTO1Click(Sender: TObject);
{
  var
  // переменная типа Byte
  i: Byte;
begin
  {
    MessageView.Lines.Clear; // очистка содержимого редактора
    for i := 99 downto 0 do
    begin
      // Добавляем строку в коллекцию строк редактора
      MessageView.Lines.Add (IntToStr(i) + Chr(9) + IntToStr(i + 100));
      // Добавляем пустую строку
      MessageView.Lines.Add('');
    end;
  }
end;

```

Для команды меню «Цикл WHILE»:

```

procedure TMainForm.WHILE1Click(Sender: TObject);
{
  var
  // переменная для управления условным циклом
  // необязательно должна быть порядковой
  i: Single;
begin
  {
    MessageView.Lines.Clear; // очистка содержимого редактора
    i := 0;
    while i <= 1 do
    begin
      // Добавляем строку в коллекцию строк редактора
      MessageView.Lines.Add (FloatToStr(i) );
      // Добавляем пустую строку
      MessageView.Lines.Add('');
      // Инкрементируем счетную переменную
      i := i + 0.01;
    end;
  }
end;

```

end;

Для команды меню «Цикл REPEAT»:

```
procedure TMainForm.REPEAT1Click(Sender: TObject);
var
  i: Double;
begin
  {
  MessageView.Lines.Clear;
  i := 0;
  repeat
    MessageView.Lines.Add (FloatToStr(i) );
    i := i + 0.2
  // Условие выхода из цикла противоположно циклу WHILE
  until i >= 10
  }
end;
```

10. Запустим и протестируем приложение.

3.2. Примеры использования условных операторов ОР

Модифицируем разработанное в данной лабораторной работе приложение для изучения условных операторов языка ОР и массивов.

1. Аналогично п. 9 раздела 3.1 создадим новый элемент меню и добавим команды, как показано на рис. 2.6.

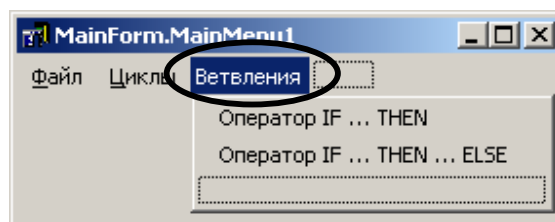


Рис. 2.6. Команды меню «Ветвления» в дизайнера меню

Используем команды меню для подпрограмм-обработчиков, которые демонстрируют работу основных условных операторов IF, а также простые операции с массивами.

С командой меню «Оператор IF...THEN» ассоциируем следующий обработчик:

```

procedure TMainForm.IF1Click(Sender: TObject);
var
    matrix: array[1..10, 1..10] of Integer;
    i, j: ShortInt;
    sTemp: string;
begin
    MessageView.Lines.Clear;
    MessageView.Lines.Add('Исходный массив');
    MessageView.Lines.Add('');
    // сформируем массив из случайных чисел
    for i := 1 to 10 do
        begin
            sTemp := '';
            for j := 1 to 10 do
                begin
                    // Сцепляем с i - 1 строкой случайное число и знак табуляции
                    matrix[i, j] := Round(10*Random + 1);
                    // формируем i-ю строку матрицы
                    sTemp := sTemp + IntToStr(matrix[i, j]) + Chr (9);
                end;
            // выводим i-ю строку матрицы
            MessageView.Lines.Add(sTemp);
        end;
    MessageView.Lines.Add('');
    // Заменяем все диагональные элементы на нули
    MessageView.Lines.Add('Преобразованный массив');
    MessageView.Lines.Add('');
    for i := 1 to 10 do
        begin
            sTemp := '';
            for j := 1 to 10 do
                begin
                    if i = j then

```

```

        matrix [i, j] := 0;
        // формируем i-ю строку матрицы
        sTemp := sTemp + IntToStr(matrix[i, j]) + Chr (9);
    end;
    // выводим i-ю строку матрицы
    MessageView.Lines.Add(sTemp);
end;
end;

```

В обработчике процедуры (IF1Click) показан один из возможных способов задания начальных значения элементам массива с использованием функции-генератора случайных чисел, а также тривиальный алгоритм поиска элементов массива по заданному критерию.

Элементу меню «Оператор IF...THEN...ELSE» назначим обработчик, исходный код которого показан ниже:

```

procedure TMainForm.IFELSE1Click(Sender: TObject);
var
    matrix: array[1..10, 1..10] of Integer;
    i, j: ShortInt;
    sTemp: string;
    NegativeCount, PositiveCount: Integer;
begin
    MessageView.Lines.Clear;

    MessageView.Lines.Add('Исходный массив');
    MessageView.Lines.Add('');
    // сформируем массив из случайных чисел (см. предыдущий пример)
    . . .
    NegativeCount := 0; PositiveCount := 0;
    for i := 1 to 10 do
    begin
        for j := 1 to 10 do
        begin
            if matrix [i, j] <= 0 then
                NegativeCount := NegativeCount + 1
            else

```

```

        PositiveCount := PositiveCount + 1;
    end;
end;
MessageView.Lines.Add ('');
MessageView.Lines.Add ('Число положительных элементов: ' +
    IntToStr(PositiveCount));
MessageView.Lines.Add ('Число отрицательных элементов: ' +
    IntToStr(NegativeCount));
end;

```

Эта подпрограмма, обработчик IFELSE1Click, демонстрирует пример подсчета числа отрицательных и положительных элементов массива.

2. Запустим и протестируем приложение (рис. 2.7).

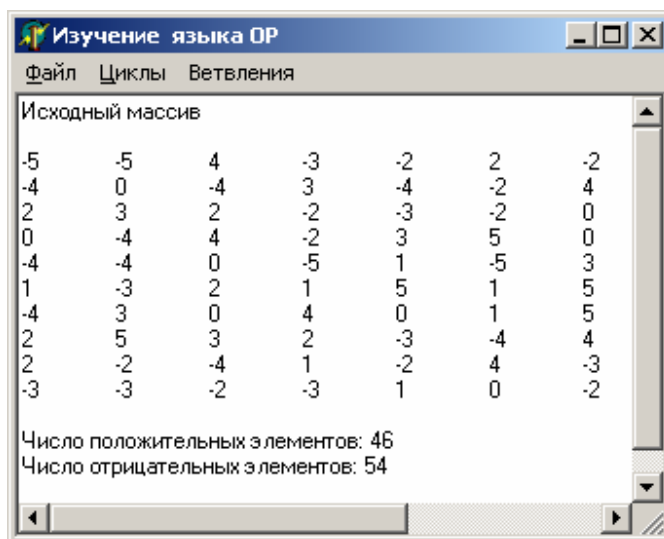


Рис. 2.7. Вид главного окна приложения и результаты его работы

3.3. Элементы работы с интегрированным отладчиком Delphi

Программирование даже простых приложений, состоящих из небольших и несложных подпрограмм, реализующих тривиальные алгоритмы обработки данных и отвечающих за интерактивное взаимодействие с пользователем, сопряжено с опасностью появления в логике их работы многочисленных ошибок.

Такие ошибки, в отличие от неверного синтаксиса не выявляются компилятором языка программирования на этапе сборки проекта и их локализация сопряжена с анализом фрагментов исходного текста алгоритмов программы. Для упрощения этой задачи все современные среды разработки, в том числе и Delphi, имеют интегрированные в состав IDE интерактивные отладочные средства, которые предоставляют разработчику многочисленные инструменты и механизмы отладки программы.

Используем в качестве примера ранее разработанный проект программы для изучения некоторых основополагающих приемов работы с интегрированным отладчиком. Выполним над проектом следующую последовательность действий.

1. Для обеспечения возможности выполнения отладки приложения в исполняемый файл программы должна быть добавлена так называемая *отладочная информация (debug information)*. Для этого необходимо используя команду главного меню Options ... | Project открыть диалоговое окно Project Options (Опции проекта). Выбрав в этом окне закладку Compiler необходимо в разделе Debugging установить следующие опции: Debug Information, Local Symbol и Reference Info.

Первая опция разрешает компилятору включать в объектные модули проекта (DCU файлы – Delphi Compiled Unit) базовую отладочную информацию, которая делает возможным пошаговую отладку и расстановку точек останова. Опция Local Symbol позволяет внедрить в объектные модули и в конечном итоге в исполняемый файл информацию о так называемых локальных идентификаторах (символах) программы. Установка этой опции даст возможность при дизассемблировании наблюдать соответствие предложений и операторов программы инструкциям процессора. Последняя установка не является обязательной для выполнения отладки, однако бывает весьма полезной, поскольку дает возможность разработчику интерактивно переходить к определению переменных и декларации классов объектов при нажатии левой кнопкой мыши на их идентификаторах в редакторе исходного текста при удерживаемой клавиши CTRL (рис. 2.8). Включение рассмотренных опций не позволяет выполнять отладку приложения, до тех пор пока не будет выполнена повторная компиляция модулей и последующая сборка проекта (Build | Project).

2. В простейшем случае для выполнения отладки некоторого фрагмента кода предварительно в одном или нескольких операторах этого фрагмента должны быть установлены так называемые *точки останова (breakpoint)*. Точка

останова – это средство указания отладчику приостанавливать нормальный ход исполнения программы при прохождении предложения программы с которой ассоциирована такая точка останова. Точки останова устанавливаются непосредственно в редакторе исходного кода.

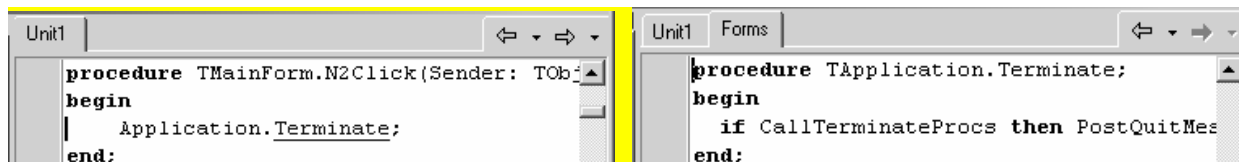


Рис.2.8. Результат включения опции Reference Info (до пехода и после перехода)

Перейдем в редакторе кода на обработчик IFELSE1Click ассоциированный с элементом меню «Оцератор IF...THEN...ELSE». Установим точку останова одинарным нажатием на маркерной колонке (полоска слева). При этом строка исходного текста будет выделена инверсным красным цветом, а слева отобразится круглый красный маркер, отмечающий точку останова.

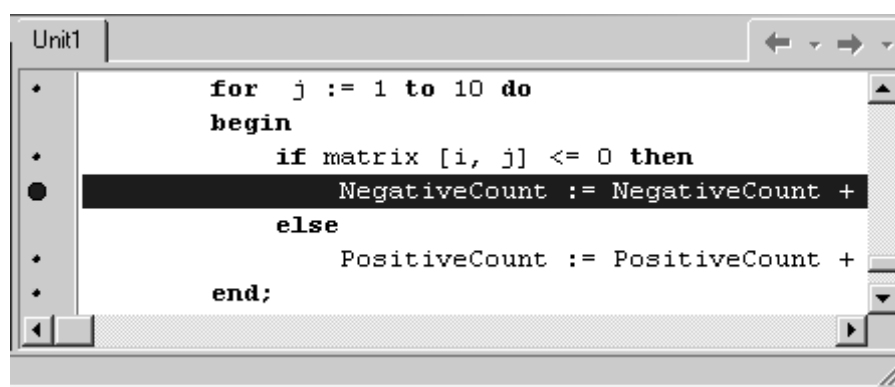


Рис. 2.9. Установка точек останова в редакторе кода

Простая точка останова может быть также определена с использованием быстрой клавишной комбинации F5.

Необходимо отметить, что возможности интегрированного отладчика Delphi не ограничиваются только работой с простыми точками останова. Дополнительная функциональность точек останова может быть задана с помощью диалогового окна Source Breakpoint Properties, которое доступно по команде Breakpoint properties в контекстном меню редактора кода (рис. 2.10).

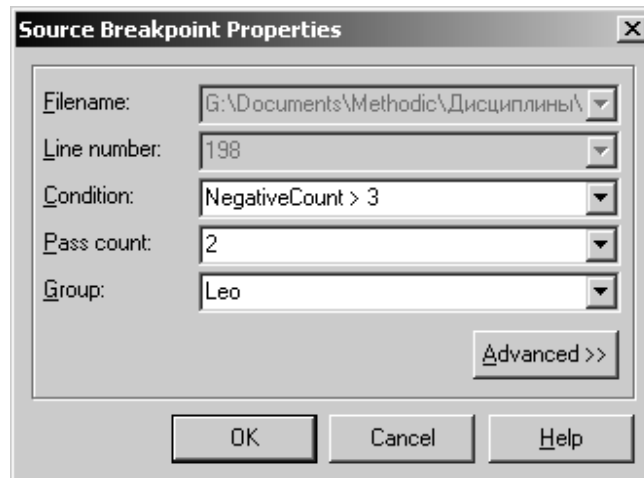


Рис. 2.10. Диалоговое окно свойств точки останова

С любой точкой останова можно ассоциировать условие, задаваемое в опции Condition в виде выражения возвращающего булевское значение. Вычисление в ходе исполнения программы истинного значения этого выражения приводит к активизации данной точки останова. Такое выражение может содержать переменные соответствующие области видимости данной точки останова. Для точки останова можно определить момент срабатывания после ее прохождения некоторого количества раз, что задается параметром Pass count. Задание числа проходов точки останова бывает полезным при отладке циклических алгоритмов. Наконец, несколько точек останова могут быть объединены в группу, для их согласованного управления. Идентификатор групп для точки останова устанавливается с помощью опции Group.

Для заданной ранее точки останова укажем свойства как показано на рис. 2.10.

3. При выполнении отладки необходимо бывает наблюдать на рядом параметров алгоритма и самое главное за содержимым переменных и свойств объектов. Для этого в арсенале интегрированного отладчика Delphi имеется специальное средство – *окно наблюдения за объектами программы (Watch List window)*. Для добавления переменной в окно просмотра, для наблюдения за ней в процессе отладки, ее необходимо выделить в редакторе кода и выбрать из контекстного меню команду Add Watch At Cursor | Debug или использовать клавиатурную комбинацию CTRL-F5.

В качестве примера добавим в окно просмотра переменные NegativeCount и PositiveCount и будем наблюдать за ними в ходе отладки программы.

4. Использование точек останова бывает необходимо для быстрой локализации фрагмента программы, в котором разработчик предполагает наличие логической ошибки. Однако для более детального понимания работы алгоритма крайне важно бывает пошаговое выполнение операторов, когда разработчик имеет возможность перейти от текущего исполняемого предложения к следующему. Отладчик Delphi предоставляет два вида пошагового исполнения программы в ходе ее отладки: Step Over – без захода в подпрограммы (F8) и Trace Into – с заходом в подпрограммы (F7).

После того как установлена точка останова и заданы ее параметры, а в окне просмотра добавлены переменные, проект приложения может быть повторно собран и запущен на выполнение.

Активизируем в работающей программе команду Оператор IF...THEN ...ELSE | Ветвления. Это приведет к приостановке исполнения программы и переход в редактор кода, в момент когда будут выполнены условия срабатывания установленной точки останова. Текущая исполняемая строка программы при этом будет помечена в маркерной колонке специальным знаком в виде зеленой стрелки ↗.

Перейдя из приостановленного приложения в отладчик выполним пошаговую отладку, используя для этого многократное выполнение команды Step Over (F8). При этом можно будет наблюдать за переходом отладчика от одного оператора программы к другому, а также просматривать изменение содержимого переменных в окне просмотра. Если окно просмотра в текущий момент неактивно, то оно может быть активизировано командой Watches | Debug Windows | View или клавишной комбинацией CTRL-ALT-W.

Рассмотренные элементарные приемы использования интегрированного отладчика не исчерпывают всех многочисленных возможностей этого мощнейшего инструмента в составе среды Delphi. Перечислим только некоторые важнейшие из них: вывод отладочных сообщений различных видов в журнал событий (Event Log), наблюдение за стеком вызовов, автоматический переход при возникновении исключений различных типов, дизассемблирование и пошаговое исполнение кода на уровне машинных команд, автоматическое формирование точки останова при условном изменении переменных в окне просмотра.

Владение этим инструментом разработки имеет исключительную важность для разработчика, поскольку при реализации даже небольших программных проектов имеют место трудно выявляемые логические ошибки различных уровней проектирования.

3.3. Задания

1. Внимательно изучить теоретический материал из раздела 2.
2. Выполнить все этапы из практической части лабораторной работы и проанализировать исходный код приведенных примеров.
3. Самостоятельно изучить синтаксис и работу условного оператора CASE.
4. Самостоятельно изучить логические (битовые) и булевские операторы, а также арифметические операторы MOD и DIV.
5. Составить фрагмент программы, реализующий алгоритм «пузырьковой» сортировки одномерного массива с выводом исходного и отсортированного массива аналогично приведенным примерам.
6. Составить фрагмент программы для вычисления частного q и остатка r при делении двух натуральных (целых неотрицательных) a на d , не используя операции целочисленного деления (DIV) и вычисления остатка (MOD).
7. Составить фрагмент программы, в которой, не используя других массивов, необходимо переставить элементы массива в обратном порядке.

4. Контрольные вопросы

1. Назовите основные типы данных языка Object Pascal.
2. В чем состоит принципиальное отличие операторов условных циклов WHILE и REPEAT?
3. Какие типы данных могут быть использованы для переменной-счетчика счетного цикла FOR?
4. Чем отличаются статические и динамические массивы и каким образом регулируется размер динамического массива?
5. Какие преимущества дает использование свойства Align для визуальных компонентов VCL?

Лабораторная работа № 3

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ В ОБЪЕКТ PASCAL. ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ. ПРОГРАММИРОВАНИЕ ФАЙЛОВЫХ ОПЕРАЦИЙ

1. Цель работы

Первичной целью данной работы является раскрытие сущности такого фундаментального понятия современного программирования, как подпрограмма. В работе рассматриваются основополагающие вопросы процедурного программирования, такие, как интерфейс и реализация подпрограммы, формальные и фактически параметры в подпрограммах, передача параметров по значению и по ссылке. Наряду с этим, в теоретическом материале приводятся основные сведения о структуре модуля в ОР. В практической части работы приводятся несколько примеров реализации и использования процедур и функций, которые иллюстрируют теоретический материал.

2. Основные теоретические сведения

2.1. Фундаментальные идеи процедурного программирования

Главной проблемой структурного программирования и алгоритмических языков первых поколений стало отсутствие языковых средств структурирования исходного текста и, как следствие, крайне большая избыточность кода программ. Решением этих задач стало введение в языки программирования понятия подпрограммы как фундаментального средства ее структурирования. Подпрограмма – это средство представления исходного кода в виде логически связанных, но самостоятельных программных единиц, каждая из которых решат небольшую специфическую алгоритмическую задачу. Подпрограммы в общем случае бывают двух видов: *процедуры (procedure)* и *функции (function)*.

Подпрограмма представляет собой особым образом оформленный фрагмент исходного кода, имеющий собственный идентификатор, ограничивающий область видимости своих внутренних объектов. Упоминание идентификатора подпрограммы в исходном тексте приводит к активизации подпрограммы и называется ее *вызовом (call)*. Сразу после активизации подпрограммы, распреде-

ляется память под ее внутренние переменные и начинают выполняться входящие в нее операторы. После окончания выполнения последнего из них управление передается обратно в *точку вызова* подпрограммы (*invocation point*) и продолжают выполняться операторы, стоящие непосредственно за оператором вызова подпрограммы (рис. 3.1).

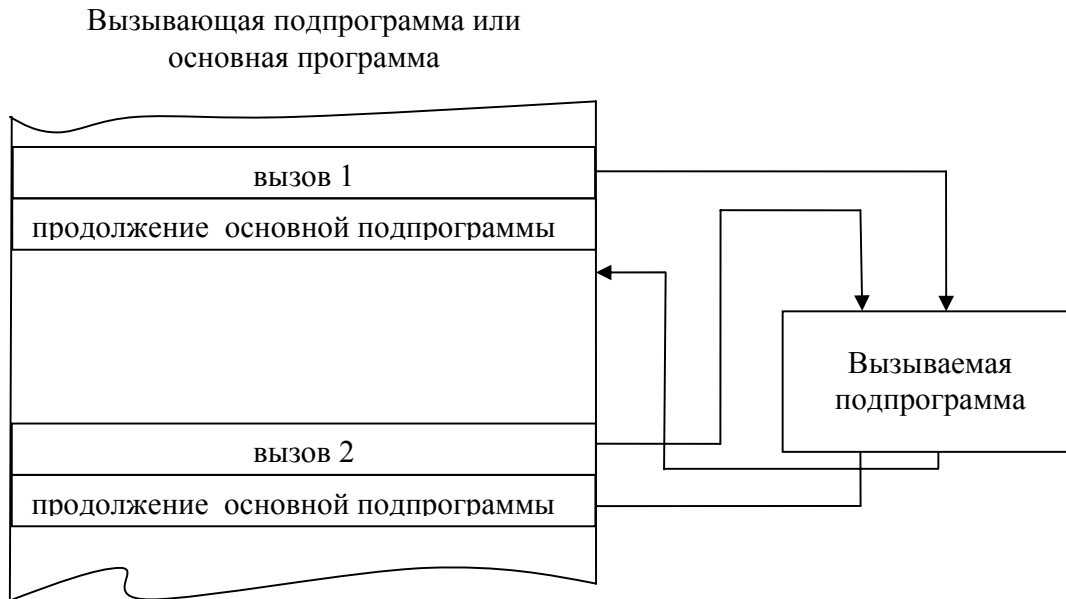


Рис. 3.1. Схематичная модель вызова подпрограммы

Подпрограмма может быть вызвана из других фрагментов кода множеством раз. При этом ее код не копируется в точку вызова и существует в единственном экземпляре, что позволяет существенным образом сократить объем программы, путем выделения в ней некоторого количества подпрограмм.

Для обмена информацией между основной программой и подпрограммой используется один или несколько *параметров вызова* (*call parameter*). Параметр представляет собой некоторое значение, которое передается в подпрограмму и используется в ней посредством своего идентификатора. Синтаксически передача параметров выражается путем указания в операторе вызова после имени подпрограммы непосредственных значений параметров (в виде литералов) или идентификаторов переменных, которые задают значение этих параметров.

Процедура — это подпрограмма, которая не возвращает слева от себя какое-либо значение и применяется для выполнения некоторых алгоритмических действий. В следующем примере показан вызов процедуры, первый параметр в

которую передается в виде непосредственного значения, в второй в виде переменной:

```
DoSomething (12, x); // вызов процедуры DoSomething
```

Параметры передаваемые в виде переменных могут быть использованы для передачи значения из вызываемой подпрограммы в вызывающую.

Напротив, функция, подобно переменной, может участвовать в выражениях и возвращает слева от себя некоторое значение заданного в ее определении типа.

```
// вызов функции sin, возвращающая значение вещественного типа
y := sin(10 * x + 10);
```

Наряду с подпрограммой, другим важным понятием в парадигме процедурного программирования является принятие *программного модуля* (*program module*). Программный модуль является средством структуризации программного проекта в целом. В самом общем случае, модуль предназначен для *инкапсуляции* (*encapsulation*), т.е. размещения внутри себя, набора из нескольких подпрограмм. Подпрограммы размещаются в модуле либо по признаку решения однотипных задачи, например, модуль с подпрограммами для обработки строк, модуль подпрограмм для работы с датами или модуль с математическими подпрограммами, либо модуль содержит в себе подпрограммы, реализующие части одной крупной алгоритмической задачи.

На уровне программного модуля в ЯП определено понятие *видимости* (*visibility*) подпрограммы, которое регламентирует доступность подпрограммы из других модулей программного проекта. Подпрограммы так же, как и переменные, объявленные в модуле, в самом простейшем случае могут быть либо доступны только внутри модуля (*private*) или доступны как внутри, так и вне модуля, т.е. в других модулях проекта (*public*).

Физически модуль программы представляет собой файл с исходным тестом подпрограмм, он имеет имя, на основании которого на подпрограммы внутри него ссылаются другие модули.

2.2. Реализация принципов процедурного программирования в ОР

Любая подпрограмма языка Object Pascal до того как будет впервые использована, посредством ее вызова, должна быть алгоритмически описана. Описание подпрограммы в ОП состоит из заголовка и тела подпрограммы. Для объявления процедур и функций, то есть задания их заголовков, используются зарезервированные ключевые слова PROCEDURE и FUNCTION соответственно. Заголовок процедуры имеет вид:

```
procedure    <subname>    [([(<param_1>:    <type>;    ...    <param_n>):
<type>)]];
```

Тогда как заголовок функции представляется следующим образом

```
function    <subname>    [([(<param_1>:    <type>;    ...    <param_n>):    <type>)]]:
<type>;
```

Синтаксический элемент <subname> является идентификатором подпрограммы, <param_xxx> – идентификатором параметра, а <type> задает тип параметра или возвращаемого функцией значения. Сразу за заголовком подпрограммы может следовать одна или несколько стандартных директив языка, которые уточняют действия компилятора. Параметры, указываемые при декларации подпрограммы, называются *формальными параметрами (formal argument)*. С точки зрения семантики языка ОП формальный параметр – это фактически внутренняя переменная подпрограммы, которой будет присвоено значение при вызове подпрограммы. Значения, указываемые при вызове подпрограммы, напротив, называются *фактическими параметрами (actual argument)*.

Список формальных параметров необязателен и может отсутствовать. Если же он есть, то в нем должны быть перечислены имена формальных параметров и их типы, например:

```
procedure    MyProc (a: real; b: integer; c: char);
```

Параметры в списке отделяются друг от друга точками с запятой. Несколько следующих подряд однотипных параметров можно объединять в подсписки, например:

```
function    MyFunc (a, b: integer); real;
```


Операторы тела подпрограммы, как уже упоминалось, рассматривают список формальных параметров как своеобразное расширение раздела описания переменных: все переменные из этого списка могут использоваться в любых выражениях внутри подпрограммы.

Для понимания деталей реализации процедурного программирования в Object Pascal необходимо учитывать существенное отличие этого языка от других ЯП, которое связано с синтаксически жесткой структурой модуля программы. Модуль в ОР получил название *юнита* (*unit*) и имеет следующую структуру:

```

unit <name>;
interface
[uses <unitslist>;]

    <declaration>;
    . . .
implementation
[uses <unitslist>;]
    <implementation>;
    . . .
[initialization
    <initialization section>;]
[finalization
    <finalization section>;]
end.

```

Модуль состоит из заголовка, интерфейсной секции, секции реализации, секции инициализации и секции финализации.

Заголовок (*unit header*) модуля определяет идентификатор модуля. Он состоит из зарезервированного ключевого слова UNIT за которым следует идентификатор <name> и разделитель. Идентификатор модуля является тем синтаксическим элементом, на основе которого осуществляется *связывание* (*binding*) нескольких модулей в единую исполняемую программу. Важно отметить, что в ОР имя файла модуля всегда должно отличаться от идентификатора модуля.

Секция интерфейса (interface section) модуля начинается с ключевого слова INTERFACE и продолжается до ключевого слова IMPLEMENTATION. Этот раздел предназначен для объявления констант, типов, переменных, а также процедур и функций, которые будут доступны для клиентов этого модуля, т.е. других модулей программы, которые подключают этот модуль. Эти записи называются открытыми (*public*). В раздел интерфейса при объявлении процедур и функций включаются только их заголовки (идентификаторы и списки параметров). Этот раздел модуля может содержать список подключаемых, внешних модулей по отношению к данному модулю. Этот список следует за ключевым словом USES и состоит из идентификаторов модулей, разделенных запятыми.

Секция реализации (implementation section) модуля начинается с ключевого слова IMPLEMENTATION и заканчивается ключевым словом END, если в модуле отсутствуют необязательные секции инициализации и финализации. Секция реализации алгоритмически определяет те процедуры и функции, которые были задекларированы (описаны) в секции интерфейса. В этом разделе модуля подпрограммы реализуют свою функциональность. Подпрограммы в этой секции могут располагаться в произвольном порядке. Список параметров открытых подпрограмм в секции реализации может быть опущен. Однако в случае указания списка параметров он должен полностью совпадать с декларацией подпрограммы в секции интерфейса.

В дополнение к определению открытых процедур и функций в секции реализации могут быть определены константы, типы (в том числе и классы), переменные и подпрограммы, которые будут закрытыми для модулей, являющихся клиентами данного модуля. Секция реализации может иметь собственное предложение USES, которое должно следовать непосредственно за ключевым словом IMPLEMENTATION.

Подраздел инициализации модуля в ОР не является обязательным элементом. Он начинается с ключевого слова INITIALIZATION и продолжается до начала подраздела финализации или, если он отсутствует, до конца модуля. Раздел инициализации содержит операторы, которые выполняются при упоминании идентификатора данного модуля в предложении USES модуля-клиента или в основной программе.

Раздел финализации также является необязательным. Он содержит операторы, которые будут выполняться, когда основная программа завершает свое выполнение. Секция финализации, как правило, выполняется в порядке противоположном инициализации.

Указание заголовка подпрограммы в разделе интерфейса модуля означает только то, что подпрограмма может быть доступна в других модулях проекта. После объявления подпрограмма обязательно должна быть реализована в секции реализации модуля.

Реализация подпрограммы состоит из заголовка и тела. Тело подпрограммы, то есть ее алгоритм, в ОР представляется в виде операторных скобок, **внутри которых заключены операторы реализующие этот алгоритм**. Между заголовком и телом процедуры может располагаться блок описаний переменных, констант и меток.

Синтаксически шаблон реализации процедуры выглядит следующим образом:

```

procedure <procname> [((<param_1>: <type>; . . .));
const
    <constname> = <value>;
    . . .
var
    <varname>: <type>;
    . . .
begin
    <statements>
end;

```

Аналогичным образом в ОР представляется блок описаний и тело подпрограммы для функции.

Рассмотрим пример функции, реализующей операцию возведения в степень, которая не представлена в ОР в виде отдельного оператора:

```

function Power (a, b: Real): Real;
begin
    if a > 0 then
        Power := exp(b*ln(a))
    Else if a < 0 then
        Power := exp(b*ln(abs((a))))
    Else if b = 0 then
        Power := 1

```

```

    Else
        Power := 0;
end;

```

Из примера видно, что в области видимости функции неявно определена одноименная идентификатору функции переменная (в примере – Power), совпадающая по типу с возвращаемым значением функции. Эта переменная используется для возвращения результирующего значения из функции.

```

y := Power(10, 3); // результат работы функции - 1000

```

Уже отмечалось что, указание в модуле программы как заголовка, так и реализации подпрограммы делает ее доступной в других модулях, то есть процедура или функция одного модуля может быть вызвана из подпрограмм другого модуля. Это свойство дает возможность объединять произвольное количество модулей в один проект приложения. Рассмотрим пример организации связи между двумя модулями.

В первом модуле с именем MyUnit1 реализованы две подпрограммы: функция PublicFunc и процедура PrivateProc. Функция PublicFunc к тому же еще и описана в секции итерфейса, что делает ее открытой для других модулей.

```

unit MyUnit1;
{СЕКЦИЯ ИНТЕРФЕЙСА}
interface
    function PublicFunc (Param1: Integer; Param2: Integer); Real;
// открытые переменные модуля
var MyPublicVar: string;
// модули используемые данным
uses OtherUnit1, OtherUnit2;
{СЕКЦИЯ РЕАЛИЗАЦИИ}
Implementation
// закрытые переменные модуля
var MyPrivateVar: string;
// реализация функции PublicProc
function PublicFunc (Param1: Integer; Param2: Integer); Real;
begin

```

```

        PublicFunc := (Param1 + Param2) / 3.14;
end;
// реализация процедуры PrivateProc
procedure PrivateProc (Param: Integer);
begin
    . . .
end;

```

Второй модуль MyUnit2 является *клиентом* (*consumer*) первого модуля, поскольку в предложении USES его секции реализации приведен идентификатор первого модуля. Все процедуры и функции модуля MyUnit2 могут вызывать открытую функцию PublicFunc, но им невидима закрытая процедура PrivateProc.

```

unit MyUnit2;
interface
    procedure DoSomething (P1, P2: Integer);
implementation
uses MyUnit1; // перечисляем модули, которые будут использованы
procedure DoSomething (P1, P2: Integer);
var y: Real;
begin
    // вызов открытой функции из модуля MyUnit1
    y := PublicFunc(P1, P2);
end;

```

Одним из свойств параметров является способ их передачи в подпрограмму. В ОР имеется три вида прохождения параметров в подпрограмму: *по значению* (*by value*), *по ссылке* (*by reference*) и *по константному значению* (*by constant value*). При передаче параметра по значению содержимое параметра копируется внутри подпрограммы в новую ячейку памяти. Все изменения внутри подпрограммы над содержимым параметра, переданного по значению, не отражаются на исходных значениях. По умолчанию параметры в ОР передаются именно таким образом. Напротив, при передаче параметра по ссылке в подпрограмму передается не копия передаваемого значения, а адрес переменной, указанной в качестве фактического параметра. Значение переменной, пере-

данной в качестве параметра по ссылке, может изменяться в подпрограмме и это изменение станет доступным на стороне вызывающей подпрограммы, так как ячейка памяти переданной переменной *разделяется (share)* одновременно двумя подпрограммами. Для определения параметра по ссылке перед именем параметра указывается ключевое слово VAR. Для параметра, объявленного по ссылке, нельзя указывать в качестве фактического аргумента константное значение.

Следующий пример наглядно демонстрирует различия между передачей параметров по ссылке и по значению:

```

function FuncByVal(param: Integer): Integer;
begin
    param := param + 33;
    FuncByVal := param;
end;
function FuncByRef(var param: Integer): Integer;
begin
    param := param + 33;
    FuncByRef := param;
end;
procedure CallingProc;
var
    x, y: Integer;
begin
    x := 123;
    ShowMessage('Исходное значение x: ' + IntToStr(x));
    y := FuncByVal (x);
    ShowMessage(x после передачи по значению: ' + IntToStr(x));
    y := FuncByRef (x);
    ShowMessage('x после передачи по ссылке: ' + IntToStr(x));
end;

```

В данном примере после вызова второй функции значение переменной *x* в вызывающей подпрограмме станет равным 156.

В случае передачи параметра как константы ключевое слова VAR заменяется на CONST. При передаче параметра как константы, как и при прохождении

параметра по ссылке, в подпрограмму передается ее адрес. Однако компилятор блокирует любые присваивания параметру-константе нового значения в теле подпрограммы.

Передача значений по ссылке выполняется существенно быстрее, чем по значению, так как отсутствует необходимость копирования значения в новую ячейку памяти. Несмотря на это надо отметить, что описание формальных параметров как параметров-переменных может иметь отрицательные последствия.

Во-первых, это исключает возможность вызова подпрограммы с фактическими параметрами в виде выражений и литералов, что делает программу менее компактной. Например:

```
y := FuncByRef (x + 1); // ошибка
y := FuncByRef (124); // ошибка
```

Во-вторых, в подпрограмме существует возможность случайного изменения параметра, которое доступно на стороне вызывающей подпрограммы, что может повлиять на ее работу.

Использование параметров-значений также имеет свои недостатки, например, при передаче параметров большого размера (или большого числа параметров-значений), происходит их копирование, что расходует память в стеке подпрограммы и требует значительного времени на копирование. В случае передачи параметров большого размера оптимальным является использование параметров-констант, которые не копируются в стек подпрограммы и в то же время не могут быть изменены.

Важной особенностью многих развитых современных языков программирования, в том числе и языка Object Pascal, является возможность определения нескольких подпрограмм (процедур или функций) с одинаковыми именами, но различными наборами параметров. Такая возможность получила название *перегрузки процедур и функций* (*overloading procedures and functions*). При перегрузке подпрограммы должны декларироваться с ключевым словом (директивой) **OVERLOAD** и иметь различные списки параметров. Например:

```
// подпрограмма деления вещественных чисел
function Divide(X, Y: Real): Real; overload;
begin
```

```

    Result := X/Y;
end;
// подпрограмма деления целых чисел
function Divide(X, Y: Integer): Integer; overload;
begin
    Result := X div Y;
end;

```

Другой уникальной особенностью языка ОР является возможность декларации в теле одной подпрограммы другой вложенной подпрограммы. Область видимости вложенной подпрограммы **при этой оказывается ограниченной** телом подпрограммы верхнего уровня. В свою очередь вложенные подпрограммы могут иметь свои собственные вложенные процедуры и функции. Например:

```

procedure DoSomething(S: string);
var
    X, Y: Integer;
    // вложенная процедура
    procedure NestedProc(S: string);
    begin
        ...
    end;
begin
    ...
    // вызов вложенной процедуры
    NestedProc(S);
    ...
end;

```

2.3. Записи и файловый тип данных

К структурным типам данных, наряду с рассмотренными ранее массивами, относятся *записи (record)*. Запись аналогична *структурам (structure)* в языках C/C++ и представляет собой гетерогенный набор элементов. Каждый элемент называется *полем (field)*. Объявление (декларация) типа записи определяет

имена и типы данных каждого отдельного поля. Синтаксис определения типа записи выглядит следующим образом:

```
type <recordTypeName> = record

    <field1>: <type1>;
    ...
    <fieldN>: <typeN>;
end
```

где <recordTypeName> – любой правильный идентификатор, задающий имя декларируемого типа, <fieldxxx> – идентификатор соответствующего поля, <type> – идентификатор типа поля.

Например, следующая декларация создает запись типа TDateRec, в которой имеет три поля:

```
type
    TDateRec = record
        Year: Integer; // год
        Month: 1..12; // месяц
        Day: 1..31; // день
end;
```

Переменная типа запись создается аналогично переменной любого другого типа. Например:

```
var MyBirthDay: TDataRec;
```

Для доступа к полям переменной типа запись используется специальный оператор, называемый *селектором поля (field selector)*, который синтаксически представляется символом «точка». Например:

```
MyBirthDay.Year := 1975;
MyBirthDay.Month := 5;
MyBirthDay.Day := 9;
```

Доступ к полям записи в компактной и удобной форме также обеспечивает оператор WITH ... DO:

```
with MyBirthDay do
begin
    Year := 1975;
    Month := 5;
    Day := 9;
end;
```

Для однотипных переменных типа «запись» допустима операция присваивания:

```
var
    MyBirthDay, CurrentDay: TDataRec;
. . .
MyBirthDay := CurrentDay;
```

Другим важнейшим структурным типом в языке ОР является *файловый тип* данных. Файл с точки зрения языка ОР представляет собой упорядоченный набор элементов некоторого одинакового типа. Для декларации файлового типа используется следующий синтаксис:

```
type <fileName> = file of <basetype>
```

где <fileName> определяет имя декларируемого файлового типа, <basetype> задает тип элемента файла. Базовым типом может быть любой тип данных, кроме объектного и файлового типа. Например, для определения переменной файлового типа – файла целых чисел необходимо записать:

```
type IntegerFile = file of Integer; // определяем файловый тип
var fTempFile: IntegerFile; // декларируем переменную
```

Таким образом, в ОР файл, которому может быть ассоциирован файл файловой системы ОС, представляется переменной особого файлового типа.

Работа с файлами в ОР состоит из трех типичных операций:

- создание или открытие файла по его имени и получение от ОС дескриптора файла;
- запись или чтение всего файла или его части;
- закрытие файла и освобождение файлового дескриптора.

Для получения файлового дескриптора и ассоциирования файла на диске с файловой переменной в ОР используется процедура из стандартной библиотеки времени исполнения – процедура `AssignFile`, которая в качестве параметров принимает файловую переменную и путь к файла. Например:

```
AssignFile (fTempFile, 'C:\leo.int'); // получаем дескриптор
```

Вызов данной процедуры не создает нового файла, а только получает файловый дескриптор. Для создания нового файла и его открытия используется другая процедура – `Rewrite`, которая получает в качестве параметра файловую переменную ранее инициированную вызовом процедуры `AssignFile`.

```
Rewrite(fTempFile); // создаем файл
```

Если файл с таким же именем уже имелся на диске, то в результате вызова данной процедуры файл будет повторно создан, а его прежнее содержимое потеряно.

Открытие ранее созданного файла без потери содержимого с возможностью чтения файла или добавления элементов в файл осуществляется путем вызова другой процедуры библиотеки подпрограмм Delphi – процедуры `Reset`, которая, как и `Rewrite`, использует в качестве параметра файловую переменную.

```
Reset (fTempFile); // открываем
```

Наконец, для чтения и записи в файл используются стандартные процедуры ввода/вывода (в/в) языка ОР – процедуры `Read` и `Write`, которые позволяют поэлементно считать или записать данные соответственно. В качестве параметров процедуры получают файловую переменную и переменную-элемент, куда будет считано значение из файла или откуда будет записано в файл. Следующий пример демонстрирует запись 300 элементов – целых чисел, в типизированный файл:

```
for i := 200 to 499 do
    Write(fTempFile, 100);
```

После окончания выполнения действий над файлом он должен быть закрыт. При этом ранее задействованный файловый дескриптор будет возвращен ОС. Файл закрывается с помощью процедуры CloseFile, которой, как и ранее описанным процедурам, передается файловая переменная, с которой ассоциирован закрываемый файл.

```
CloseFile(fTempFile);
```

Кроме рассмотренных выше подпрограмм для работы с файлами в составе библиотек подпрограмм Delphi имеется масса других файловых функций и процедур. Описание некоторых из них представлено в табл. 3.1.

Таблица 3.1. Некоторые основные подпрограммы для работы с файлами

Имя	Формат	Описание
Eof	function Eof(var F): Boolean;	Определяет, достигнут ли конец файла F (F – файловая переменная). Функция возвращает True, если текущая позиция файлового указателя находится за границей последнего символа (элемента) файла. В противном случае функция возвращает False.
FilePos	function FilePos(var F): Longint;	Возвращает текущую позицию файлового указателя.
FileSize	function FileSize(var F): Integer;	Возвращает размер файла в байтах или в количестве записей для типизированного файла.
MkDir	procedure MkDir(S: string);	Создает новую директорию, путь которой указан в виде параметра процедуры.
Rename	procedure Rename(var F; NewName: string);	Изменяет имя файла с файловой переменной F на имя указанное в каче-

		стве параметра Newname.
Seek	procedure Seek(var F; N: Longint);	Перемещает текущий указатель элемента файла F на N позиций.

Для работы с обыкновенными текстовыми файлами в стандартной библиотеке подпрограмм имеется предопределенный файловый тип `TextFile`. В этом случае при создании файловой переменной ключевое слово `FILE` не указывается:

```
var fTextFile: TextFile;
```

Остальные принципы работы с текстовыми файлами аналогичны типизированным файлам, где в качестве элемента данных используются строковые переменные. Для записи и чтения в текстовый файл построчно используются процедуры `WriteLn` и `ReadLn` соответственно.

3. Практическая часть


3.1. Использование файлов и файловых операций


Рассмотрим пример создания прототипа примитивного приложения для управления адресной книгой. Данный пример демонстрирует общие принципы использования подпрограмм, а также последовательность действий при программировании файловых операций для загрузки типизированных данных в программу и сохранения их из программы в файл.



1. Создадим в интегрированной среде Delphi новый проект и сохраним, задав ему произвольное имя используя команду `Save Project As | File`.

2. Выполним следующие действия по настройке компонентов проекта:

- изменим исходное имя формы (1) на «MainForm» (свойство `Name`);

- добавим на форму компонент `StringGrid` (класс `TStringGrid`)  (2) из закладки `Additional` палитры компонентов и установим для него новое имя – «AddressBookGrid» (рис. 3.2);

- добавим компонент `GroupBox` (класс `TGroupBox`)  из закладки `Standard` (3);

- добавим в область компонента GroupBox три простых однострочных редактора типа Edit (4)  и один типа MaskEdit  (5). Зададим следующие идентификаторы этим компонентам: «FIOEdit», «AddressEdit», «PhoneEdit» и «BirthDayEdit» (для TMaskEdit);

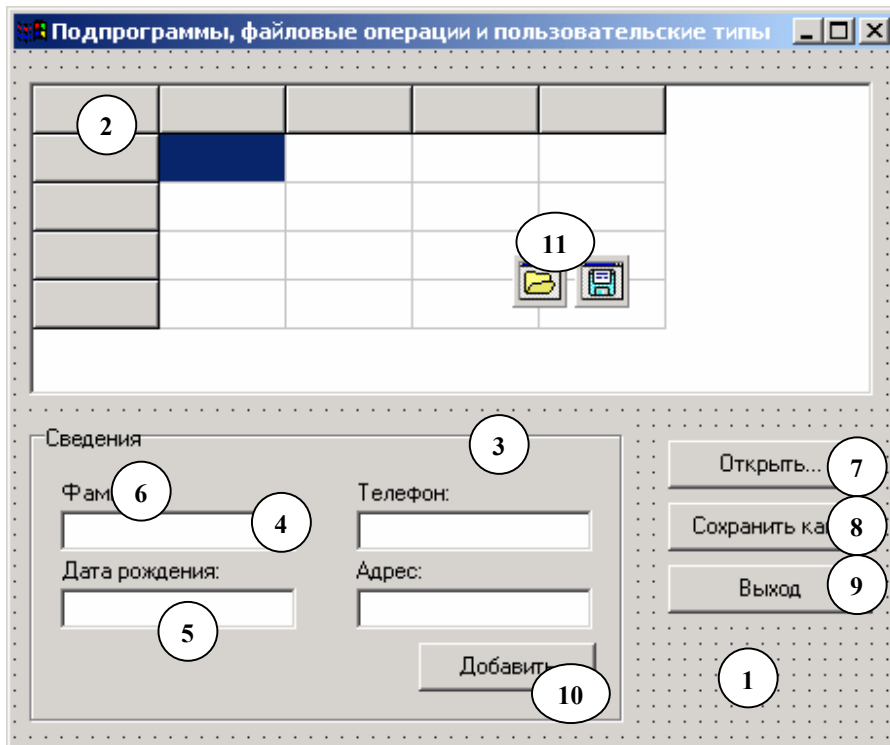






Рис. 3.2. Внешний вид главного окна приложения в режиме разработки

- выделив в дизайнера строку редактирования BirthDayEdit установим для нее шаблон-маску ввода типа Date, используя для этого редактор свойства EditMask .
- расположим строки редактирования, как это показано на рис. 3.2., а также добавим соответствующие подписи к ним в виде компонентов Label (класс TLabel)  (6);
- добавим на форму четыре командных кнопки типа Button и установим заголовки для них, как это показано на рис. 3.2, а также зададим кнопкам соответствующие имена: «OpenButton» для кнопки «Открыть...» (7), «SaveButton» для кнопки «Сохранить как...» (8), «ExitButton» для кнопки «Выход» (9) и наконец «AddButton» для кнопки «Добавить» (10);

• присоединим к форме два новых невизуальных компонента из закладки Dialogs палитры компонентов: OpenFileDialog (класс TOpenDialog)  и SaveDialog (класс TOpenDialog)  (11), которым установим соответствующие имена «OpenDialog» и «SaveDialog».

Реализуем бизнес-логику приложения, добавив код соответствующих подпрограмм в модуль главного окна.

3. В секцию интерфейса добавим определения нового структурного пользовательского типа (тип запись). Этот тип данных будет использован в качестве базового типа для определения файловой переменной:

```
type TAddressBookRec = record
    FIO: string[64];
    Address: string[64];
    Phone: string[12];
    BirthDay: TDateTime;
end;
```

4. Также в секцию интерфейса, в раздел PRIVATE определения типа класса формы TMainForm, добавим декларации двух процедур, которые станут закрытыми процедурами (методами) данного класса:

```
procedure OpenFile(FileName: string);
procedure SaveFile(FileName: string);
```

5. Эти процедуры должны быть реализованы в секции реализации данного модуля. Они предназначены для выполнения загрузки данных из файла и сохранения в файл. Исходный текст данных подпрограмм с подробными комментариями приведен ниже:

```
// подпрограмма загрузки данных из файла в «решетку» (StringGrid)
procedure TMainForm.OpenFile(FileName: string);
var
    // файловая переменная типа TAddressBookRec
    fAddressBookFile: file of TAddressBookRec;
    // буфер для чтения элемента из файла
```

```

    AddressBookRecBuff: TAddressBookRec;
begin
    // ассоциируем дисковый файл с файловой переменной
    AssignFile(fAddressBookFile, OpenFileDialog.FileName);
    // открываем файл для чтения
    Reset(fAddressBookFile);
    // сканируем файл, используя цикл с предусловием
    // функция Eof возвращает True, если достигнут конец файла
    while not Eof(fAddressBookFile) do
    begin
        // читаем элемент файла в буфер
        Read(fAddressBookFile, AddressBookRecBuff);
        // используем оператор with для компактности записи
        with AddressBookGrid, AddressBookRecBuff do
        begin
            // копируем из буфера в ячейки «решетки»
            Cells[0, RowCount-1] := FIO;
            Cells[1, RowCount-1] := Address;
            Cells[2, RowCount-1] := Phone;
            Cells[3, RowCount-1] := DateToStr(BirthDay) ;
            // добавляем очередную строку
            RowCount := RowCount + 1;
        end;
    end;
    // закрываем файл
    CloseFile(fAddressBookFile);
end;

// подпрограмма записи данных из «решетки» в файл
procedure TMainForm.SaveFile(FileName: string);
var
    // файловая переменная
    fAddressBookFile: file of TAddressBookRec;
    // буфер для записываемого в файл элемента
    AddressBookRecBuff: TAddressBookRec;

```



```

    i: Integer; // счетчик
begin
    AssignFile(fAddressBookFile, SaveDialog.FileName);
    // перезаписываем или создаем новый файл
    Rewrite(fAddressBookFile);
    // сканируем «решетку», используя счетный цикл
    // не захватывая последнюю пустую строку
    for i := 1 to AddressBookGrid.RowCount - 2 do
    begin
        // конвертируем запись в TAddressBookRec
        AddressBookRecBuff.FIO :=
            AddressBookGrid.Cells[0, i];
        AddressBookRecBuff.Address :=
            AddressBookGrid.Cells[1, i];
        AddressBookRecBuff.Phone :=
            AddressBookGrid.Cells[2, i];
        AddressBookRecBuff.BirthDay :=
            StrToDate(AddressBookGrid.Cells[3, i]);
        // и пишем в файл
        Write(fAddressBookFile, AddressBookRecBuff);
    end;
    // закрываем файл
    CloseFile(fAddressBookFile);
end;

```

6. Реализуем код подпрограмм, которые будут определять взаимодействие пользователя с элементами оконного интерфейса приложения. Такими подпрограммами являются обработчики одиночного нажатия (событие `OnClick`) на кнопках. Нажатие кнопок `SaveButton` и `OpenButton` приводит к отображению стандартных диалогов `Windows` для открытия и сохранения файлов:

```

procedure TMainForm.SaveButtonClick(Sender: TObject);
begin
    // если в стандартном диалоге выбрали имя файла
    // и нажали СОХРАНИТЬ
    if SaveDialog.Execute then

```

```

begin
    // вызываем подпрограмму записи в файл
    SaveFile(SaveDialog.FileName);
end;
end;

procedure TMainForm.OpenButtonClick(Sender: TObject);
var
    i: Integer;
begin
    // чистим решетку
    for i := 1 to AddressBookGrid.RowCount - 1 do
        AddressBookGrid.Rows[i].Clear;
    // установим число строк в решетке
    AddressBookGrid.RowCount := 2;

    // если в стандартном диалоге выбрали имя файла
    // и нажали ОТКРЫТЬ
    if OpenFileDialog.Execute then
        begin
            // вызываем подпрограмму OpenFile,
            // которая загрузит данные из файла в решетку
            OpenFile(OpenDialog.FileName);
        end;
end;

```

7. Обработчик нажатия кнопки AddButton содержит код, который позволяет добавить новую запись, на основе данных из строк редактирования, в таблицу:

```

procedure TMainForm.AddButtonClick(Sender: TObject);
var
    i: Integer;
begin
    // Если не введена фамилия, то выводим из обработчика
    if FIOEdit.Text = '' then

```

```

        Exit;
    with AddressBookGrid do
    begin
        // переносим данные в ячейки последней строки таблицы
        Cells[0, RowCount-1] := FIOEdit.Text;
        Cells[1, RowCount-1] := AddressEdit.Text;
        Cells[2, RowCount-1] := PhoneEdit.Text;
        Cells[3, RowCount-1] := BirthDayEdit.Text;
        // добавляет еще одну строку
        RowCount := RowCount + 1;
    end;

    // перебираем все компоненты, принадлежащие данной форме
    for i := 0 to Self.ComponentCount - 1 do
        // если компонент является наследником
        // класса TCustomEdit
        if Self.Components[i] is TCustomEdit then
            // очищаем его
            (Self.Components[i] as TCustomEdit).Clear;
        end;
    end;

```

8. Наконец, кнопка ExitButton имеет следующий обработчик:

```

procedure TMainForm.ExitButtonClick(Sender: TObject);
begin
    // выход из приложения
    Application.Terminate;
end;

```

9. Форма (класс TForm) также является компонентом библиотеки классов VCL, который имеет широкий набор различных событий. Наиболее часто используемым событием является событие OnCreate, возбуждаемое сразу после создания окна и компонентов, которые на нем были расположены, но до отображения самого окна. Обработчик этого события может быть использован для задания начальных значений свойствам формы и ее компонентов. В данном

примере этот обработчик используется для задания внешнего вида компонента AddressBookGrid.

```
procedure TMainForm.FormCreate(Sender: TObject);  
begin  
    // Настройка грида в режиме run-time  
    with AddressBookGrid do  
        begin  
            // число не редактируемых столбцов и строк  
            FixedCols := 0;  
            FixedRows := 1;  
            // общее число столбцов и строк  
            ColCount := 4;  
            RowCount := 2;  
            // ширина столбцов (зависит от разрешения экрана)  
            ColWidths[0] := 120;  
            ColWidths[1] := 120;  
            ColWidths[2] := 64;  
            ColWidths[3] := 120;  
            // выводим подписи столбцов в верхнюю строку  
            Cells[0, 0] := 'ФИО';  
            Cells[1, 0] := 'Адрес';  
            Cells[2, 0] := 'Телефон';  
            Cells[3, 0] := 'Дата рождения';  
        end;  
    end;
```

10. Выполним сборку проекта и запустим разработанное приложение. Добавим несколько тестовых записей в таблицу, после чего сохраним их в файл. Затем, закроем приложение и повторно его запустим, загрузив из ранее созданного файла сохраненные данные (рис. 3.3).

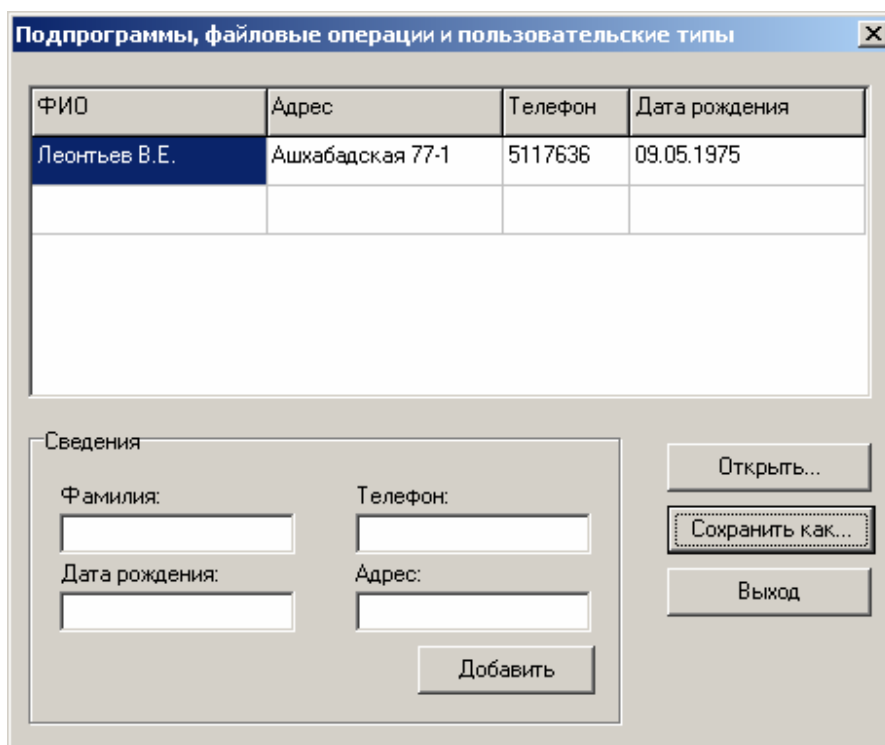


Рис. 3.3. Внешний вид разработанной программы на этапе исполнения

3.3. Задания

1. Внимательно изучить теоретический материал из раздела 2. Воспроизвести в среде разработки код примеров, приведенных в теоретической части работы.
2. Выполнить все этапы из практической части лабораторной работы.
3. Самостоятельно изучить функциональность компонента StringGrid, назначение его свойств, методов и событий.
4. Реализовать в рассмотренном примере функцию удаления записей.
4. Разработать программу табулирования сложной функции с возможностью записи результатов табуляции в файл.

4. Контрольные вопросы

1. В чем преимущества и недостатки различных способов передачи параметров в подпрограммы?
2. В чем состоит смысл операции перегрузки процедур и функций?

3. Может ли в качестве поля одной записи быть использована другая запись (запись записей)?

Лабораторная работа № 4

ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА ПРОГРАММИРОВАНИЯ В ОБЪЕКТ PASCAL. КЛАССЫ И ОБЪЕКТЫ

1. Цель работы

Цель данной лабораторной работы состоит в том, чтобы дать студентам ясные представления об основополагающих концепциях объектно-ориентированного программирования. В рамках данной работы студент должен освоить элементарные навыки разработки программного обеспечения с использование приемов объектно-ориентированного программирования.

2. Основные теоретические сведения

2.1. Реализация объектно-ориентированной парадигмы в Object Pascal

Одной из основных проблем, которые присущи процедурной парадигме программирования, является то, что при подобном подходе отсутствуют языковые средства, связывающие данные (информацию) и алгоритмические процессы, необходимые для их обработки. Все данные и процессы при процедурном подходе к программированию находятся «в свободном плавании», и на программиста возлагается задача обеспечить правильность использования каждого элемента. Поскольку такая взаимосвязь на уровне языка отсутствует, может возникнуть проблема конфликтов использования данных несколькими алгоритмическими процессами (подпрограммами).

Другой проблемой является невозможность программной реализации абстракции сложных процессов обработки данных. Объекты реального мира, отражаемые в виде программного кода и данных, не могут быть адекватно представлены просто как подпрограммы и переменные, так как такие объекты мыслятся как единое целое.

Наконец немаловажным является то, что только подпрограммы универсального назначения могут быть повторно использованы в нескольких проектах, тогда как код, относящийся к реализации большинства бизнес-процессов, оказывается практически не переносим от одного проекта к другому. Таким образом, процедурное программирование, хотя и имеет огромные преимущества перед непроцедурным подходом, оставляет неразрешенными более сложные задачи программирования. Эти проблемы снимаются в рамках *объектно-ориентированного программирования (ООП)*.

Современные ЯП реализуют объектно-ориентированную концепцию программирования, основная идея которой состоит в том, что данные (переменные) должны быть неразрывно связаны с методами обработки этих данных (подпрограммами). Ключевым элементом данной концепции программирования является особый программный элемент, который получил название *объекта (object)*. Объект также называется *абстрактной структурой данных*, основным назначением которой является обеспечение «обертки» для некоторого набора данных, представленных в простейшем случае в виде переменных, и набора подпрограмм. Предоставление такой оболочки для кода и данных и создание единого целого с точки зрения синтаксиса и семантики языка программирования называется *объектной инкапсуляцией*. При этом работа с данными и детали ее реализации сокрыты от внешнего пользователя объекта.

С точки зрения классического объектно-ориентированного программирования каждый объект является реализацией или воплощением некоторого *абстрактного типа данных*, подобно тому, как скалярная переменная является воплощением обычного скалярного типа. Абстрактный тип данных называют также *классом объектов*. Класс представляет собой программную единицу, которая реализует структуры данных и подпрограммы для их обработки. Данные, описанные в классе объектов, принято называть *полями*, а подпрограммы – *методами*. В ООП также определено понятие *свойства* как специального метода, основным назначением которого является корректный доступ к полю класса объекта. Поля, методы и свойства называются компонентами класса или его *членами (class members)*.

Под термином поля класса по существу понимают переменную класса, предназначенную для хранения данных, которые *раздельно существуют для каждого экземпляра класса*. Метод в свою очередь является процедурой или функцией, которая ассоциирована с классом в целом и *разделяется всеми его*

объектами. Все методы, за исключением особых, которые называются *методами класса*, создаются вместе с экземпляром класса.

Класс должен быть всегда задекларирован перед созданием первого экземпляра класса, т.е. объекта. Класс может быть определен только в области видимости программы или модуля, до декларации процедуры или функции. Декларация класса в ОР имеет следующую форму:

```
type <classname> = class (<ancestorclass>)
    <memberlist>;
end;
```

где <classname> – любой синтаксически правильный идентификатор, представляющий имя класса, <ancestorclass> – имя родительского класса от которого данный класс наследует свою функциональность, <memberlist> – декларация членов класса. Родительский класс должен быть реализован, до которого как декларируется класс-наследник. Если имя родительского класса не указано, то класс в языке ОР наследует функциональность непосредственно базового класса TObject, который является «прародителем» всех объектов в ОР.

Рассмотрим пример простого класса, функциональность которого заключается в хранении символьной строки и возможности отображения этой строки в виде сообщения во всплывающем диалоговом окне Windows. Декларация данного класса выглядит следующим образом:

```
type
    TCustomMsgBox = class (TObject)
        FMsg: string;
        procedure Show;
end;
```

Если внутри класса описан один или несколько методов, то они должны быть реализованы в секции реализации в том же модуле, где был объявлен класс. Например:

```
procedure TCustomMsgBox.Show;
begin
    ShowMessage (FMsg) ;
```



```
end;
```

Важно отметить, что в отличие от реализации простой процедуры или функции имя метода предваряется именем класса, которому он принадлежит, и отделяется от него «точкой» – селектором члена класса. В области видимости любого метода данного класса доступны все члены этого класса, как поля, так и методы.

Правильно задекларированный и реализованный класс может быть использован только посредством создания объекта этого класса. Для создания такого объекта, во-первых, необходимо объявить переменную данного класса.

Здесь крайне важно отметить, что объекты имеют *ссылочную природу*, т.е. переменная класса или объект физически представляют собой 32-х разрядную ячейку памяти, которая содержит только адрес памяти, с которого начинается объект (рис. 4.1). По этому адресу располагаются поля данного экземпляра класса и таблица, содержащая адреса методов, которые, как уже отмечалось принадлежат классу в целом.

Переменная класса объявляется так же, как и обычная скалярная переменная в блоке объявления переменных в области видимости декларации класса или в модуле, имеющем ссылку на модуль, в котором реализован данный класс.

```
var objCustomMsgBox: TCustomMsgBox;
```

Второе, что необходимо сделать для использования функциональности класса – сконструировать объявленный объект. Это означает, что для полей объекта и методов класса будет выделено адресное пространство, а адрес начала объекта в памяти будет записан в объектную переменную. Для выполнения этого действия каждый класс имеет особый метод, который называется *конструктором* (*constructor*).

Конструктор создает объект и инициализирует его поля, задавая числовым полям нулевое значение, строчным – пустую строку, а указателям (адресным переменным) – специальное значение пустого указателя NIL.

Принято, что конструктор имеет идентификатор Create и представляет собой вид функции, которая возвращает адрес объекта (указатель на объект). Например:

```
objCustomMsgBox:= TCustomMsgBox.Create;
```

Как видно из примера, конструктор вызывается от имени класса, что может на первый взгляд показаться абсурдным. Однако это совсем не так, поскольку конструктор вызывается еще до того момента, как будет создан сам объект. Конструктор, а также другие методы, которые вызываются от имени класса называются *методами класса* или *статическими методами*.

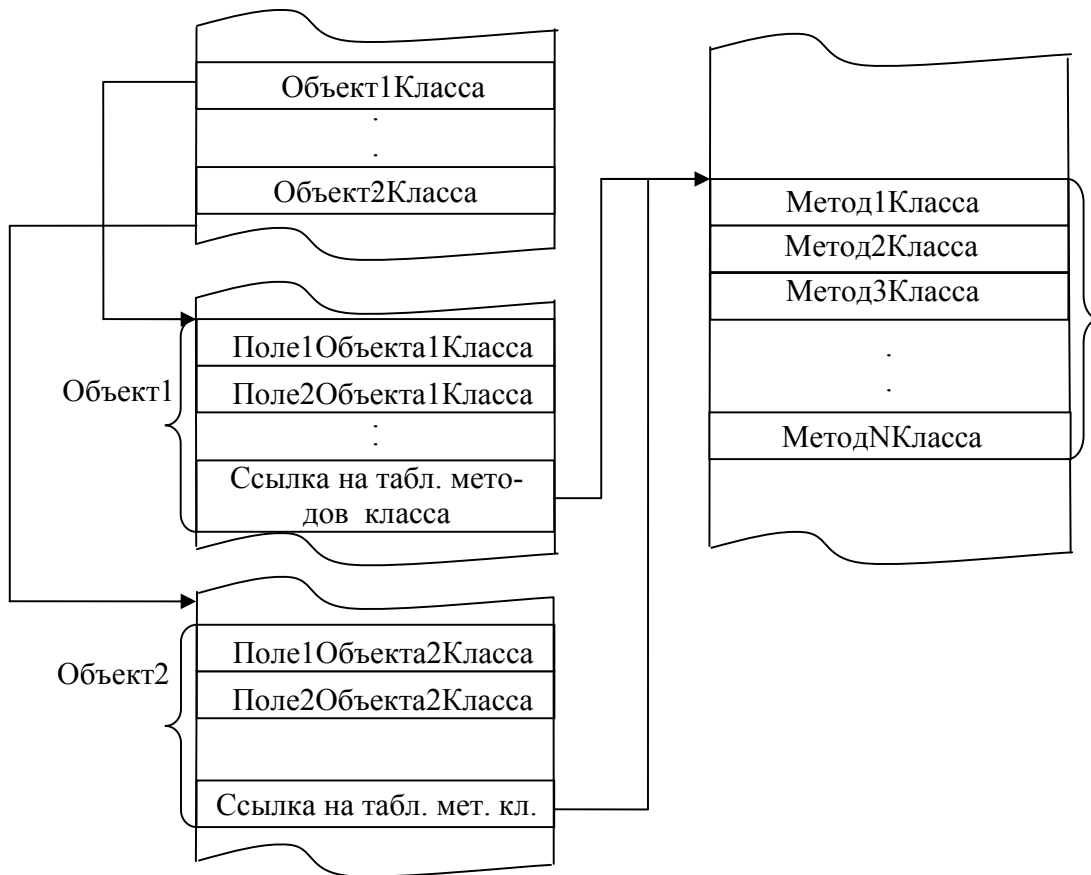


Рис. 4.1. Ссылочная природа объектов

Для доступа к членам класса используется специальный оператор, подобный оператору доступа к полю записи. Этот оператор получил название *селектора члена класса* (*member selector*) и синтаксически представлен символом «точки». Например, для использования объекта в рассматриваемом примере необходим следующий код:

```
begin
```

```

. . .
objCustomMsgBox := TCustomMsgBox.Create; // конструируем
// присваиваем значение свойству
objCustomMsgBox.FMsg := 'Привет мир ООП!';
objCustomMsgBox.Show; // вызываем метод Show
objCustomMsgBox.Free; // вызываем деструктор
end;

```

Результатом работы этого кода будет всплывающее окно с сообщением (рис. 4.2.)

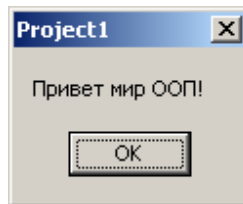


Рис. 4.2. Результат работы метода Show класса TCustomMsgBox

Важным аспектом использования объектов в ООП языка ОР является необходимость их принудительного разрушения и освобождения используемой ими памяти. Этот процесс называется *освобождением объекта (object releasing)*. Освобождение выполняется посредством вызова специального метода – метода Free, работа которого заключается в безопасном вызове специального метода называемого *деструктором (destructor)* объекта – метода Destroy.

Необходимость освобождения объекта заключается также в его ссылочной природе. Дело в том, что при выходе из области видимости объектной переменной автоматически освобождается только сама объектная переменная, которая содержала ссылку на объект. В результате теряется адрес объекта, а память распределенная для полей объекта, не освобождается. Такая ситуация называется *утечкой памяти (memory leak)*.

Наряду с инкапсуляцией другой важнейшей концепцией ООП является *наследование* классов объектов. Под наследованием понимается возможность создания новых классов и соответственно объектов на их основе, которые обладают свойствами и поведением родительских классов. Такая концепция позволяет создавать иерархии классов, включающие наборы классов, порожденных

от одного общего предка и обладающих все большей специализацией и функциональностью по сравнению со своими предшественниками.

Как было упомянуто ранее, для организации наследования необходимо в скобках после ключевого слова `CLASS` указать имя класса предка. Рассмотрим пример, в котором реализуем наследование вновь создаваемого класса от ранее созданного класса `TCustomMsgBox`. Например:

```
// расположим объявление нового класса в секции Interface
type
  TMsgBox = class (TCustomMsgBox)
    FCaption: string;
    // переопределили конструктор по умолчанию
    constructor Create(InitMsg: string); overload;
    procedure ShowEx;
end;

. . .

// разместим реализацию класса в секции Implementation
procedure TMsgBox.ShowEx;
begin
  Application.MessageBox(PChar(FMsg), PChar(FCaption), 1);
end;
constructor TMsgBox.Create(InitMsg: string);
begin
  inherited Create; // вызываем конструктор предка
  FMsg := InitMsg;
end;
```

Класса `TMsgBox` расширяет функциональность базового класса `TCustomMsgBox` за счет введения нового поля и метода. Новые члены класса `TMsgBox` будут принадлежать только этому вновь созданному классу и его объектам, тогда как все члены класса предка `TCustomMsgBox` доступны для использования в том числе и в классе-наследнике. Важное отличие класса `TmsgBox` – это определение нового конструктора. Декларация конструктора начинается с ключевого слова `CONSTRUCTOR`. Указываемый после этого ключе-

вого слова идентификатор конструктора может быть произвольным, однако общепринятым является идентификатор `Create`. Конструктор может иметь параметры. Для сохранения доступа в классе-потомке к конструктору класса предка в конце декларации должно быть указано ключевое слово `OVERLOAD`. Это позволяет «перегрузить» конструктор (доопределить новый конструктор в дополнение к старому), вместо его «перекрытия» (замены конструктора предка новым собственным конструктором потомка) конструктором класса-потомка.

Рассмотрим пример создания и использования объекта нового класса:

```
var
    objMsg1, objMsg2: TMsgBox;

    . . .
    // создаем первый объект, используя новый конструктор
    objMsg1 := TMsgBox.Create('Привет, многоликий мир ООП!');
    objMsg1.FCaption := 'Изучение ООП';
    objMsg1.ShowEx;

    // Создаем второй объект, используя конструктор предка
    objMsg2 := TMsgBox.Create;
    // используем для вывода сообщения средства класса-предка
    objMsg2.FMsg := ('Привет тебе, непостижимый мир ООП!')
    objMsg2.Show;
```

Еще одним важным проявлением ссылочной природы объектов является возможность присваивания однотипным объектным переменным ссылок на другие объекты. Продолжая предыдущий пример рассмотрим следующий код:

```
. . .
// разрушаем второй объект
objMsg2.Free;
// присваиваем objMsg2 ссылке на первый объект
objMsg2 := objMsg1;
// попыбует использовать первый объект от имени objMsg2
objMsg2.ShowEx
// разрушаем первый объект от имени objMsg2
```

```
objMsg2.Free;
```

```
. . .
```

В этом примере объект `objMsg2` разрушается путем вызова метода `Free`. Однако при этом объектная переменная остается доступной для использования до выхода из области видимости подпрограммы. Возникает ситуация, когда объектная переменная еще «жива», а объект, ранее ассоциированный с ней, уже «мертв». При присваивании `objMsg2` ссылки на первый объект оказывается, что обе объектные переменные теперь указывают на один объект, т.е. *являются различными идентификаторами одной программной сущности*. Поэтому первый объект может быть разрушен с помощью вызова метода `Free` от имени второй объектной переменной.

Одним из следствий принципа объектной инкапсуляции является ограничение видимости членов класса по отношению к потребителям класса, а также по отношению к классам-потомкам. При объявлении класса его членам может быть задан специальный атрибут, который определяет их видимость. В ОР имеются четыре различных ключевых слова, которые задают область видимости: `PRIVATE`, `PROTECTED`, `PUBLIC` и `PUBLISHED`.

Директива `PRIVATE` задает минимальную область видимости члена класса, делая его доступным только в модуле, где он определен, т.е. для самого класса. Такие члены класса называются закрытыми. Значения таких полей и методов, как правило, оказывают решающее значение для поведения объектов, или данные, ассоциированные с этими членами, должны жестко синхронизироваться с другим членами класса, поскольку неосторожная их синхронизация может привести к нежелательным последствиям.

Директива `PROTECTED` несколько ослабляет защиту. Вне модуля, в котором определен конкретный класс, его члены остаются невидимыми и доступны только в классах-потомках — так называемом *доме* класса. Если класс-потомок не определен, такие члены остаются невидимыми, как будто они были объявлены как `PRIVATE`.

Директива `PUBLIC` открывает доступ членам класса, делая их видимыми везде, где виден сам класс.

С точки зрения задания видимости директива `PUBLISHED` аналогична `PUBLIC`, однако для членов класса из раздела `PUBLISHED` генерируется информация о *типах времени исполнения* (*runtime type information*). Информацию о типе поля класса из раздела `PUBLISHED` может получить внешняя программа.

Информацию о типах времени исполнения использует также инспектор объектов, для того чтобы отображать в **среде разработки** список свойств и событий компонентов и визуальных элементов управления. Инспектор объектов извлекает эти данные из *упаковок компонентов (design-time package)* и скомпилированных модулей. Информация о типах времени исполнения раскрывает широчайшие возможности для разработчика, например обеспечивает доступ к данным и методам класса на этапе времени исполнения, то есть когда обращение к ним не было предусмотрено алгоритмом программы. Доступ к полям и методам классов, которые имеют информацию о типах времени исполнения, возможен из любой Delphi-программы. Модуль TypInfo RTL-библиотеки Delphi содержит большое число подпрограмм и описаний специальных типов, которые обеспечивают этот доступ.

При построении класса и задании видимости его членов в классическом ООП принято скрывать все поля класса, оставляя доступными для пользователя класса только методы. Доступ к значениям закрытых данных осуществляется посредством специальных методов доступа – **так называемого механизма свойств**.

Переработаем ранее рассмотренный пример класса TCustomMsgBox в соответствии с принципом сокрытия полей класса от клиентов:

```

type
    TCustomMsgBox = class (TObject)
    private
        FMsg: string;
    public
        procedure Show;
end;

```

Из данного определения класса становится очевидно, что при простом сокрытии поля FMsg в разделе PRIVATE класс полностью теряет свою работоспособность. Для обеспечения доступа к полю необходимо предусмотреть, по крайней мере, два дополнительных открытых метода, один из которых присваивает значение, а другой возвращает его. Например:

```

type
    TCustomMsgBox = class (TObject)

```

```

private
    FMsg: string;
public
    procedure SetMsg (NewValue: string);
    function GetMsg (): string;
    procedure Show;
end;

```

Реализация этих новых методов может быть тривиальной:

```

procedure TCustomMsgBox.SetMsg (NewValue: string);
begin
    FMsg := NewValue;
end;

function TCustomMsgBox.GetMsg (): string;
begin
    GetMsg := FMsg;
end;

```

Использование методов для доступа к данным может на первый взгляд показаться несколько неуклюжим и громоздким. С точки зрения эффективности такой способ действительно не является предпочтительным. Однако использование методов доступа к данным внутри класса имеет и свои неоспоримые достоинства. Так, методы доступа позволяют управлять процессом задания значения полю, например, проверяя новое устанавливаемое значение на предмет удовлетворения некоторым критериям. Например:

```

procedure TCustomMsgBox.SetMsg (NewValue: string);
begin
    // проверяем длину сообщения
    if Length (NewValue) > 128 then
        // если она более 128 символов
        FMsg := '' {присваиваем пустое значение}
    else
        // в противном случае

```



```

    FMsg := NewValue; {устанавливаем новое значение}
end;

```

Метод, ответственный за получение клиентом класса значения поля, также может подвергаться его модификации в зависимости от некоторых условий или независимо от них:

```

function TCustomMsgBox.GetMsg(): string;
begin
    // возвращаем значение поля в верхнем регистре
    GetMsg := UpperCase (FMsg);
end;

```

Несмотря на преимущества, которые дает введение методов доступа к атрибутам (полям) класса, этот способ является синтаксически неудобным и избыточным. Вместо использования простого оператора присваивания здесь используются вызовы методов:

```

objCustomMsgBox.SetMsg('Привет тебе, о ужасный мир ООП!');
objCustomMsgBox.Show();

```

Однако язык программирования ОР предлагает разработчику решение этой проблемы посредством языкового инструмента, который получил название *свойства (property)*. Свойства – это особый объектно-ориентированный механизм, организующий доступ к атрибутам класса посредством методов доступа с использованием синтаксиса и семантики оператора присваивания. Свойства объявляются в разделе PUBLIC декларации класса с помощью ключевого слова PROPERTY:

```

property <propertyname>: <type> read <getmethod> write <setmethod>;

```

Для ранее рассмотренного примера декларация свойства может выглядеть следующим образом:

```

property Msg: string read GetMsg write SetMsg;

```

Поскольку свойство только лишь регламентирует правила доступа к одному из атрибутов класса оно не требует никакой реализации. При этом использование объекта класса `TCustomMsgBox` будет иметь вид:

```
// используем свойство так, как если бы это было простое поле
objCustomMsgBox.Msg := 'Привет тебе, о прекрасный мир ООП!';
objCustomMsgBox.Show();
```


Во всех методах объекта доступна неявная переменная `Self`, представляющая собой указатель на этот экземпляр объекта, который был использован при данном вызове этого метода. Переменная `Self` передается компилятором всем методам класса в качестве скрытого параметра.

3. Практическая часть

3.1. «Жизнь» и «смерть» объектов в режиме run-time. Динамическое создание визуальных компонентов

Рассмотрим простое приложение, основу интерфейса которого составляет главное окно и несколько элементов управления, которые создаются и отображаются динамически в ходе работы программы. В данном примере демонстрируются правила динамического создания объектов различных классов и их последующего разрушения.

1. Создадим новый проект простого Windows-приложения. В инспекторе объектов назначим имя «MainForm» главному окну приложения (свойство `Name`). Установим, используя свойство `Caption`, заголовок главного окна – «Управление окнами и динамическое создание объектов».

2. Используя дизайнер поместим на форму главного окна компонент `MainMenu` .

Создадим структуру меню и необходимые команды, как это показано в табл. 4.1. В ячейках таблицы указаны заголовки элементов меню, а в скобках даны их идентификаторы.

Таблица 4.1. Структура меню приложения

Меню	Файл (File-Menu)	Действия (ActMenu)	Сервисы (Service-Menu)	Помощь (HelpMenu)
Команды	Выход (File-ExitMenu)	Строка состояния (ActStatusMenu)	Опции... (ServiceOptionMenu)	О программе... (HelpAbout Menu)
		Панель слева (ActPanelMenu)		
		График (ActImageAreaMenu)	Панель (ServicePanelMenu)	
		Кнопка на панели (ActButtonOnPanelAreaMenu)		

3. Добавим в предложение USES секции интерфейса модуля главного окна имена следующих модулей библиотеки VCL: ComCtrls и ExtCtrls. В этих модулях сосредоточена реализация таких компонентов VCL, как StatusBar (строка состояния), Panel (панель), ProgressBar (строка выполнения процесса) и др.

Определим для каждой команды меню собственный обработчик события OnClick. Совокупность этих обработчиков составит основу реализации программной логики данного приложения.

3.1. Для команды «Выход» определим следующий обработчик:

```
procedure TMainForm.FileExitMenuClick(Sender: TObject);
begin
    Application.Terminate
end;
```

Вся работа этой процедуры сводится к вызову метода Terminate глобального объекта Application. Действие этого метода приводит к завершению работы приложения.

Объект Application создается автоматически компилятором при создании главного окна приложения. Ссылка на этот объект так же, как и сам класс TApplication, определены в модуле Forms VCL. Объект класса TApplication инкапсулирует в себе поведение приложения в целом и отвечает за хранение дескриптора приложения, дескриптора главного окна, заголовка приложения, ссылки на битовую матрицу иконки приложения и т.п.. В классе TApplication также определены события, которые наступают при получении и потере фокуса приложением, при простое приложения, при вызове справки приложения и т.д.

Перед тем, как реализовать следующие обработчики, определим несколько открытых членов класса главного окна TMainForm (раздел PUBLIC определения класса TMainForm):

```

type TMainForm = class (TForm)
    . . .
    private
        {Private declarations}
    public
        StatusBar: TStatusBar;
        LeftPanel: TPanel;
        Image: TImage;
    end;

```

3.2. Далее, определим обработчик для команды «Строка состояния» | «Действия». Исходный текст данной процедуры представлен ниже:

```

procedure TMainForm.StatusBarActMenuClick(Sender: TObject);
begin
    // динамически создаем строку состояния
    StatusBar := TStatusBar.Create(nil);
    // указываем, что родительским окном
    // будет главное окно приложения
    StatusBar.Parent := Self;
    // чтобы повторно не создать еще одну строку состояния
    // деактивируем текущий элемент меню
    (Sender as TMenuItem).Enabled := False;
end;

```

Работа данной процедуры состоит в том, чтобы создать визуальный элемент управления Windows – строку состояния (рис. 4.3).



Рис. 4.3. Элемент управления Windows типа строка состояний

Функциональность элементов интерфейса Windows доступна приложениям, создаваемым в среде Delphi посредством специальных классов, объекты которых получили название *визуальных элементов управления (controls)*. Визуальные элементы управления могут использоваться в дизайнерах форм на этапе разработки приложения и переноситься на форму из палитры компонентов. Наиболее часто используемые элементы управления собраны в закладках палитры компонентов Standard, Additional и Win32.

Однако, несмотря на то, что элементы управления можно использовать в конструкторе форм, они могут создаваться динамически в ходе работы приложения и разрушаться, после того как необходимость в их использовании отпадает.

Для создания элемента управления необходимо определить объектную переменную соответствующего типа, что и было сделано ранее в описании класса окна. После этого вызвать конструктор соответствующего класса:

```
StatusBar := TStatusBar.Create(nil);
```

Одной из особенностей использования объекта, который представляет собой визуальный элемент управления, является указание в его конструкторе ссылки на элемент управления, который возьмет на себя обязанность корректно разрушить данный элемент управления, т.е. освободить память, используемую им после окончания работы приложения. Такой элемент управления называется *владельцем (owner)* другого элемента управления. Если программист берет на себя обязанность разрушить объект, сопоставленный создаваемому элементу управления, то вместо ссылки на владельца может быть указан пустой указатель NIL, как это и показано в примере.

Наряду с этим, при создании элемента управления, необходимо указать на элемент управления, который станет контейнером для данного, т.е. того, кто разместит элемент управления в своей клиентской области. Для этого все элементы управления в VCL имеют свойство Parent, определенное в предке всех элементов управления, классе TControl. Для указания родительского элемента управления свойству Parent присваивается ссылка на объект, который должен быть также оконным элементом управления, т.е. наследником класса TWinControl. В рассматриваемом обработчике это сделано следующим образом:

```
StatusBar.Parent := Self;
```

Здесь ссылка на «родитель» указывает на объект главного окна посредством ключевого идентификатора `Self`.

Далее в обработчике события заблокируется текущая команда меню, т.е. команду, которая инициировала вызов текущего обработчика события:

```
(Sender as TMenuItem).Enabled := False;
```

Здесь используется *оператор приведения* типа `AS`, который позволяет обратиться к методам и свойствам объекта класса-наследника от имени объекта класса-предка. В данном случае это объект `Sender` класса `TObject`, реально указывающий на объект, который сгенерировал обрабатываемое событие, т.е. объект класса `TMenuItem` (пункт меню). Возможность использования оператора приведения типа является следствием еще одной важнейшей концепции ООП – *полиморфизма*.

Аналогичного эффекта блокировки команды меню можно было добиться использованием явного указания на объект команды меню:

```
ActStatusMenu.Enabled := False;
```

3.3. Следующий обработчик реализуем для команды «Панель слева» | «Действия». Результатом его работы является динамическое создание панели, пристыкованной к левой границе главного окна, и специального элемента управления – **сплиттера**, который позволяет изменять размер областей, занимаемых элементами управления. Исходный текст данной процедуры приведен ниже:

```
procedure TMainForm.ActPanelMenuClick(Sender: TObject);
var
    // ссылка на этот объект будет потеряна
    splitter: TSplitter;
begin
    LeftPanel := TPanel.Create(nil);
    LeftPanel.Parent := Self;
    // ширина панели будет равна трети от ширины окна
    LeftPanel.Width := Self.ClientWidth div 3;
```

```

// панель будет иметь «вдавленный» вид
LeftPanel.BevelOuter := bvLowered;
// при создании сплиттера в его конструкторе
// в качестве владельца укажем ссылку на
// главное окно, чтобы деструктор главного окна
// позаботился о разрушении сплиттера,
// так как ссылка на него будет безвозвратно утеряна
splitter := TSplitter.Create(Self) ;
splitter.Parent := Self;
// стыкуем сплиттер слева
splitter.Align := alLeft;
// стыкуем панель слева
LeftPanel.Align := alLeft;
// блокируем и это меню
(Sender as TMenuItem).Enabled:= False;
end;

```

В данном фрагменте кода ссылка на объект сплиттера теряется при выходе из области видимости подпрограммы, поэтому он не может быть корректно разрушен программистом. Конструктору класса TSplitter передается указатель на объект главного окна, который и разрушит данный объект, **как и любой другой собственный «дочерний» компонент.**

3.4. Обработчик для команды «График» | «Действия» выполняет действия, схожие с ранее рассмотренными обработчиками, и динамически создает объект класса TImage, функциональность которого позволяет выполнять рисование в его клиентской области, например, выводить графики, изображения из файлов графических форматов.

```

procedure TMainForm.ActImageAreaMenuClick(Sender: TObject);
begin
    Image := TImage.Create(nil);
    // вместо Self можно явно использовать MainForm
    Image.Parent := MainForm;
    // растягиваем на всю клиентскую область окна
    Image.Align := alClient;
    // автоматическое растяжение битовой матрицы

```

```

Image.Stretch := True;
  // блокируем меню
(Sender as TMenuItem).Enabled := False;
end;

```

3.5. Определим заголовок и реализуем новый метод в классе TMainForm – процедуру ButtonClick. Этот метод будет служить обработчиком события OnClick кнопки, подпрограмма создания которой будет реализована в следующем пункте. Процедура ButtonClick выполняет действия по рисованию в клиентской области компонента Image синусоиды, путем копирования битовой матрицы из памяти. Исходный текст подпрограммы представлен ниже:

```

procedure TMainForm.ButtonClick(Sender: TObject);
var
  x, y: Double;
  i: Integer;
  scale, displacement: Integer;
  bmp: TBitmap; // битовая матрица в памяти
begin
  // создаем матрицу
  bmp := TBitmap.Create;
  // обязательно настроим ее размеры
  // соответственно размерам клиентской области Image
  bmp.Width := Image.ClientWidth;
  bmp.Height := Image.ClientHeight;

  // масштабы и смещения
  scale := Image.ClientHeight div 5;
  displacement := Image.ClientHeight div 2 ;
  // рисуем синусоиду на битовой матрице
  for i := 0 to 500000 do
  begin
    y := scale * sin ( x / 10 ) + displacement;
    bmp.Canvas.Pixels[Round (x),Round (y)] := clBlack;
    x := x + 0.001
  end;
end;

```



```

    // битовую матрицу нужно скопировать на канву Image
    // скопировать матрицу можно двумя способами
    // первый способ
    //Image.Picture.Bitmap.Assign(btmap);
    // второй способ
    Image.Canvas.Draw(0,0, btmap );
    // поскольку битовая матрица в памяти уже не нужна
    // можно ее разрушить
    btmap.Free;
end;
```

3.6. Наконец реализуем последний обработчик для команды «Кнопка на панель» меню «Действия». Эта процедура динамически создает простую командную кнопку, размещая ее на панели:

```

procedure TMainForm.ActButtonMenuClick(Sender: TObject);
var
    btnTmp: TButton;
begin
    // проверим, создана ли панель
    if LeftPanel <> nil then
        begin
            // разрушать кнопку будет деструктор главного окна
            btnTmp := TButton.Create(Self);
            // но родительским окном для кнопки будет панель
            btnTmp.Parent := LeftPanel;
            // стыкуем кнопку сверху
            btnTmp.Align := alTop;
            btnTmp.Caption := 'Функция Sin(x)';
            // в режиме run-time назначаем обработчик
            //события OnClick для созданной кнопки
            btnTmp.OnClick := ButtonClick;
            // блокируем меню
            (Sender as TMenuItem).Enabled := False;
        end;
    end;
end;
```

Особенностью данного кода является назначение динамически созданному элементу управления обработчика события. Для этого специальному свойству `OnClick` объекта класса `TButton` присваивается указатель на метод `ButtonClick`, который должен быть заранее определен и реализован (см. п. 3.5).

Данный фрагмент кода демонстрирует, что объектная модель ОР позволяет не только создавать элементы управления во время выполнения, но и изменять поведение элементов управления, назначая им различные обработчики событий, что открывает фактически безграничные возможности при создании прикладных программ высокого уровня сложности.

3.7. Выполним сборку проекта и протестируем приложение. Выполним последовательно команды из меню «Действия», а затем активизируем кнопку, расположенную на левой панели.

Результат работы приложения показан на рис. 4.4.



Рис. 4.4. Результаты работы программы

3.2. Модальные диалоговые окна

Любая, даже самая простая прикладная программа предполагает наличие в своем интерфейсе нескольких вспомогательных диалоговых окон, которые отображаются по команде пользователя и предоставляют ему дополнительные функции, например, открытие и сохранение файлов, настройку конфигурации

приложения, управление данными и др. Вспомогательные окна Windows-приложения в самом общем случае могут быть двух видов: модальные и немодальные окна. Все диалоговые окна создаются другим окном, не обязательно главным окном приложения, и принадлежат ему подобно обычным элементам управления. Созданное и отображенное модальное окно в отличие от немодального не позволяет переключиться на родительское окно, до тех пор пока не будет закрыто. Немодальное окно может свободно «плавать» в клиентской области основного окна приложения или за его пределами, **позволяя пользователю переходить на родительское окно.**

Вспомогательные диалоговые окна приложения, созданного в среде Delphi представляются в программе, как и любые другие окна, объектами класса, наследуемого от предопределенного в VCL класса TForm. Поэтому для использования любого диалогового окна необходимо определить класс-наследник TForm и создать соответствующий объект. После окончания использования окна, объект, ассоциированный с ним, должен быть разрушен, а память, занимаемая им освобождена.

Среда Delphi позволяет автоматически создать новое окно и подключать его к разрабатываемому проекту. Для этого используется репозиторий объектов Delphi, с помощью которого создается модуль класса окна и его бинарный образ в виде DFM-файла (Delphi Form). DFM-файл содержит двоичные ресурсы и служебные данные, необходимые для сборки проекта при включении данного окна в приложение. **Этот файл также используется дизайнером форм благодаря чему присоединенное к проекту окно, которое в терминологии Delphi называется формой, становится доступным в среде разработки.**

Продолжим разработку приложения, которое было рассмотрено в предыдущем разделе. Добавим в этот проект несколько вспомогательных диалоговых окон и рассмотрим особенности работы с ними: статическое и динамическое создание окон, разрушение модальных окон, передача данных от одного окна к другому.

1. Добавим в проект новую форму. Для этого можно воспользоваться репозиторием объектов или выбрать команду File | New Form. Данная форма будет служить для отображения информации о названии приложения, его версии и авторских правах (так называемый About). Переименуем в инспекторе объектов объект формы, задав ему новое имя «AboutForm», а также установим новый заголовок формы – «О программе».

Для отображения вспомогательного окна имеет существенное значение стиль границы, который может быть выбран установкой свойства `BorderStyle`, принимающего следующие значения: `bsDialog` – окно не изменяет размеры и поддерживает стиль стандартного диалогового окна Windows; `bsSingle` – окно также не позволяет изменять размер, но имеет более тонкую рамку; `bsNone` – граница окна невидима, невидим в том числе и стандартный заголовок окна; `bsSizeable` – стандартное масштабируемое окно; `bsToolWindow` – окно подобно `bsSingle`, но имеет уменьшенный заголовок; `bsSizeToolWin` – окно подобно `bsSizeable`, но имеет также уменьшенный заголовок.

В нашем случае установим для свойства `BorderStyle` значение `bsDialog`.

2. При добавлении нового окна среда Delphi создает новый модуль с интерфейсом и реализацией класса окна – класса-наследника от `TForm`, в разделе интерфейса которого имеется декларация объектной переменной этого класса, то есть объекта данной формы. Например, в данном случае:

```

unit Unit2;

interface
uses
    Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
    Dialogs;

type
    TAboutForm = class (TForm)
    private
        { Private declarations }
    public
        { Public declarations }
    end;

var
    AboutForm: TAboutForm;

implementation
    {$R *.DFM}
end.

```

Для создания объекта данной формы необходимо возвращаемый конструктором класса указатель на объект присвоить объектной переменной `AboutForm`.

Однако, по умолчанию, все вспомогательные окна приложения создаются автоматически при запуске самого приложения. Такие формы называются также автосоздаваемыми (auto-create forms). Список автосоздаваемых форм можно увидеть в закладке `Forms` диалогового окна `Project Options` (`Options | Project`) (рис. 4.4).

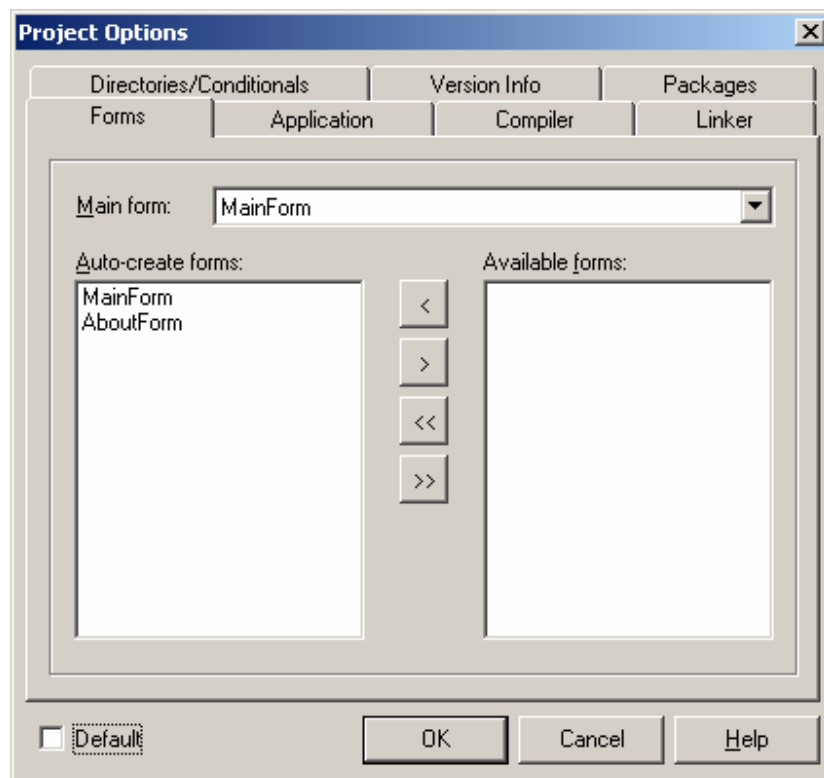


Рис. 4.4. Список автоматически создаваемых форм

Таким образом, для использования вспомогательного окна достаточно получить доступ к ее объектной переменной. Как уже отмечалось, за отображение окна, как правило, отвечает главное окно приложения, поэтому в предложение `USES` модуля главного окна необходимо включить идентификатор модуля диалогового окна:

```

. . .
var

```

```
MainForm: TMainForm;
```

```
implementation
```

```
uses Unit2;
```

```
{ $R *.DFM }
```

```
...
```

3. Используя дизайнер форм, реализуем простейший интерфейс данного диалогового окна, например, следующего вида:

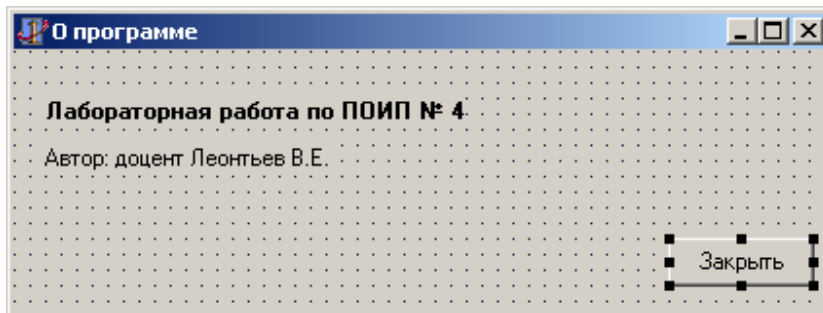


Рис. 4.5. Внешний вид диалогового окна «О программе» на этапе разработки

В инспекторе объектов для кнопки «Закреть» установим свойству `Modal-Result` значение `mrCancel`, что позволит использовать кнопку для закрытия этого диалогового окна.

4. Далее вновь перейдем в модуль главного окна приложения и определим обработчик для команды «О программе» меню «Помощь», код которого должен отображать диалоговое окно.

Клиенту объекта класса наследника `TForm` доступны несколько методов, которые позволяют отобразить на экране окно. К таким методам относятся `Show` и `ShowModal`. Первый метод отображает диалоговое окно как немодальное, а второй – как модальное. Таким образом, исходный текст обработчика будет иметь следующий вид:

```
procedure TMainForm.HelpAboutMenuClick(Sender: TObject);
begin
    AboutForm.ShowModal;
end;
```

end;

5. Выполним сборку приложения и протестируем его (рис. 4.6). Выполнение команды «О программе» должно приводить к отображению модального диалогового окна «О программе».

Важным и отрицательным следствием того, что добавляемые в проект новые формы автоматически создаются при запуске приложения, является то, что для них расходуется динамическая память приложения (место в «куче») еще до того, как то или иное окно может понадобиться. Причем после использования и последующего закрытия такого окна, объект с ним ассоциированный не разрушается и память не освобождается. С другой стороны, преимуществом использования автоматически создаваемых окон является то, что операции по их отображению происходят намного быстрее, так как окно уже создано, но только невидимо.

В большинстве случаев в реальных приложениях целесообразнее использовать динамически создаваемые окна, так как основная масса вспомогательных диалоговых окон используется крайне редко.

6. Первое, что необходимо – это убрать нужную форму из списка автосоздаваемых форм. Этого можно добиться двумя способами: используя диалог Project Option переместив идентификатор формы в список доступных форм (available) или открыв модуль самого проекта с помощью команды View Source | Project. Модуль проекта имеет следующий типичный исходный текст:

```
program Project2;

uses
  Forms,
  Unit1 in 'Unit1.pas' {MainForm},
  Unit2 in 'Unit2.pas' {AboutForm};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TMainForm, MainForm);
  Application.CreateForm(TAboutForm, AboutForm); // строку удалить
```

```
Application.Run;
end.
```

Чтобы форма не создавалась автоматически, необходимо удалить соответствующий вызов метода `CreateForm` класса `Application` (подчеркнутая строка в исходном тексте).

7. Динамическое создание формы выполняется аналогично тому, как создается любой другой объект. Например, код обработчика команды «О программе» может выглядеть следующим образом:

```
procedure TMainForm.HelpAboutMenuClick(Sender: TObject);
begin
    AboutForm := TAboutForm.Create(nil);
    AboutForm.ShowModal;
    AboutForm.Free;
end;
```

Созданная таким образом форма динамически создается, а после ее использования и закрытия разрушается, освобождая память.

В свете рассматриваемой темы о динамическом создании объектов крайне важно отметить, что язык ОР и исполняемая среда Delphi поддерживает специальные средства, входящие в состав системы структурной обработки исключительных ситуаций Delphi для безопасного использования динамически создаваемых объектов. В ОР имеется специальный блочный (структурный) оператор TRY ... FINALLY, который позволяет избежать утечки памяти при исключительных ситуациях во время динамического создания объектов. Этот оператор имеет следующий вид:

```
try
    <statements>;
finally
    <finalization>;
end;
```


где `<statements>` – потенциально опасный код, связанный с использованием динамически создаваемого объекта, `<finalization>` – код, выполняющий разрушение объекта. Например:





```
procedure TMainForm.HelpAboutMenuClick(Sender: TObject);
begin
    AboutForm := TAboutForm.Create(nil);
    try
        AboutForm.ShowModal;
        // действия по использованию объекта AboutForm
    finally
        AboutForm.Free;
    end;
end;
```

Данный код является более безопасным, так как при возникновении исключительной ситуации в блоке TRY до аварийного выхода из процедуры будет обязательно выполнен код в блоке FINALLY, что позволит избежать утечки памяти. Данная техника кодирования с использованием оператора TRY ... FINALLY применима и крайне желательна при динамическом создании любых других объектов. Использование оператора TRY ... FINALLY так же, как и других операторов структурной обработки исключительных ситуаций, позволяет создавать безопасный и надежный программный код.

Рассмотрим один возможных способов передачи информации из модального диалога в главное окно программы, объект которого, как правило, является основным координатором потоков данных в программе.

8. Присоединим к проекту еще одну новую форму, которая также станет модальным диалоговым окном. Установим для свойства BorderStyle значение bsDialog. Переименуем объект формы задав ему новый идентификатор (свойство Name) – «OptionsForm». Данное диалоговое окно будет выполнять роль диалога для измерения настроек (опций) нашей программы, в частности позволит устанавливать цвет линии синусоиды, выводимой в главном окне.

9. Удалим новую форму из списка автосоздаваемых форм. Далее, используя конструктор форм, добавим на форму несколько элементов управления: над-

пись (класс TLabel) , три полосы прокрутки типа ScrollBar (класс TScrollBar) , две кнопки  и панель  (класс TPanel), как это показано на рис. 4.6.

В инспекторе объектов для кнопки «ОК» установим свойству ModalResult значение mrOK, а для кнопки «Отмена» – mrCancel. Свойству Max (максимальная позиция «ползунка») всех трех полос прокрутки (ScrollBar) установим значение 255. Свойствам BevelInner (внутренняя кромка) и BevelOuter (внешняя кромка) панели (Panel) установим значение bvNone.

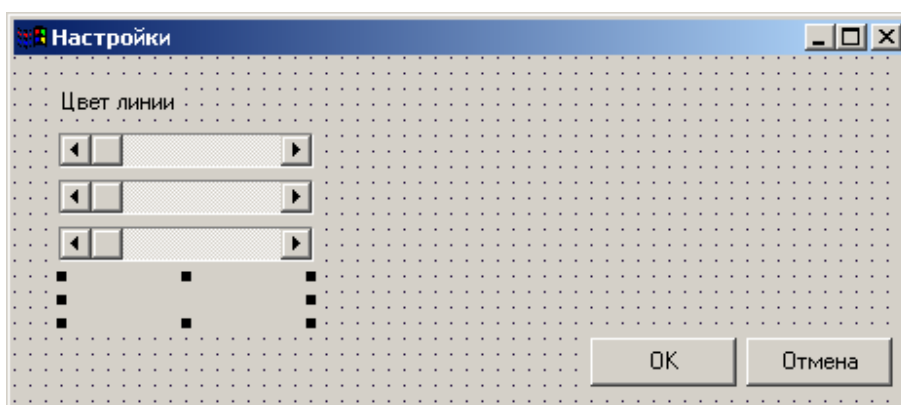


Рис. 4.6. Внешний вид окна настроек

10. Реализуем программную логику в классе TOptionsForm. Для этого добавим новое открытое поле для этого класса с идентификатором OptionLineColor типа TColor:

```

. . .
public
    OptionLineColor: TColor; {Public declarations}

```

Далее определим обработчик события OnCreate формы OptionsForm. Эта подпрограмма реализует восстановление значения опции (цвета линии) в диалоговом окне, в момент его создания, по текущему установленному значению этого параметра в главном окне.

```

procedure TOptionForm.FormCreate(Sender: TObject);
begin
    // читаем текущее значение цвета линии из главного окна

```

```

// и присваиваем это значение полю OptionLineColor
OptionLineColor := MainForm.LineColor;
// расщепляем TColor на три RGB-составляющих и устанавливаем
// в соответствии со этими значениями позиции «ползунков»
ScrollBar1.Position := GetRValue(OptionLineColor); // RED
ScrollBar2.Position := GetGValue(OptionLineColor); // GREEN
ScrollBar3.Position := GetBValue(OptionLineColor); // BLUE
// назначаем полосам прокрутки обработчик OnChange
ScrollBar1.OnChange := ScrollBar1Change;
ScrollBar2.OnChange := ScrollBar1Change;
ScrollBar3.OnChange := ScrollBar1Change;
// подсветим панель текущим цветом
Panell1.Color := OptionLineColor;
end;

```

Используя инспектор объектов назначим компоненту `ScrollBar1` обработчик события `OnChange` и реализуем в нем следующий код:

```

procedure TOptionForm.ScrollBar1Change(Sender: TObject);
begin
    // когда двигаем ползунки
    Panell1.Color := RGB(ScrollBar1.Position, ScrollBar2.Position,
        ScrollBar3.Position);
    OptionLineColor := Panell1.Color;
end;

```

Этот обработчик будет вызываться всякий раз при попытке пользователя изменить позицию «ползунка» одной из полос прокрутки. При этом обработчик сформирует значение цвета на основании положений «ползунков» всех трех полос прокрутки. Числовое значение позиции «ползунка» при этом соответствует RGB-составляющей цвета и лежит в диапазоне от 0 до 255. Полученное значение цвета используется для установки свойства `Color` компонента `Panell1`, то есть фактически для его закрашивания.

11. Теперь осталось реализовать код на стороне клиента, то есть в классе главного окна приложения. Сначала, для этого необходимо в предложении `USES` секции интерфейса модуля главного окна добавить идентификатор моду-

ля формы OptionForm. Главное окно должно создавать это диалоговое окно и получать от него сформированное значение цвета. Эта функциональность реализована в коде обработчика команды «Опции» меню «Сервисы», исходный текст которого представлен ниже.

```

procedure TMainForm.ServiceOptionMenuClick(Sender: TObject);
begin
    OptionForm := TOptionForm.Create(nil);
    try
        if OptionForm.ShowModal = mrOk then
            LineColor := OptionForm.OptionLineColor;
    finally
        OptionForm.Free;
    end;
end

```

Деятельность этой подпрограммы состоит в том, чтобы проанализировать значение, возвращаемое функцией ShowModal. Если работа диалога закончилась нажатием кнопки «ОК», то полю LineColor объекта класса TMainForm присваивается значение, возвращаемое полем OptionLineColor формы OptionForm.

Поле LineColor типа TColor должно быть предварительно объявлено в модуле формы MainForm.

```

. . .
public
    LineColor: TColor; {Public declarations}

```

Для того чтобы синусоида отрисовывалась установленным цветом в процедуре ButtonClick в необходимо идентификатор clBlack заменить на LineColor.

```

bmp.Canvas.Pixels[Round (x),Round (y)] := LineColor;

```

12. Выполним сборку проекта и протестируем работу приложения, устанавливая различные значения цвета линии.

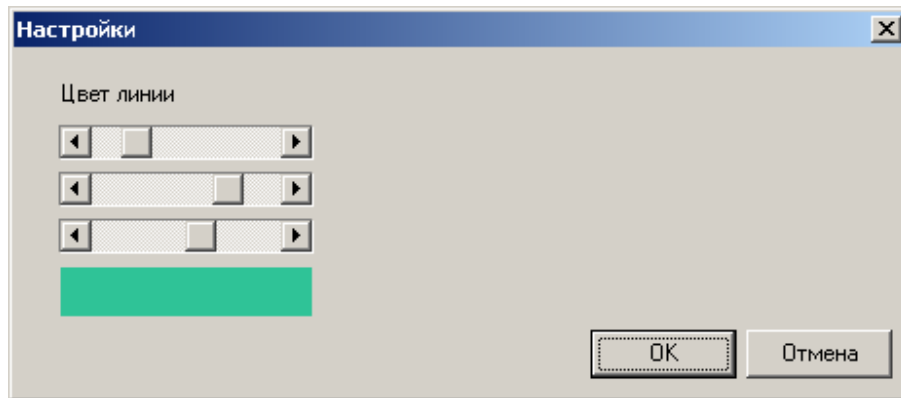


Рис. 4.7. Использование окна настроек

3.3. Немодальные диалоговые окна

Немодальные окна в прикладных программах Windows используются значительно более редко по сравнению с модальными окнами. Немодальные окна, как правило, предназначаются для отображения разнообразных плавающих палитр инструментов и информационных панелей и т.п.

Модифицируем ранее разработанное приложение с целью создания плавающего немодального окна, в котором будут отображаться экранные координаты курсора, перемещаемого в области компонента Image. В связи с этим выполним следующие действия по дальнейшей модификации проекта.

1. Присоединим к проекту еще одну новую форму. Переименуем ее как «PanelForm» и установим свойству FormStyle значение fsStayOnTop (всегда оставаться поверх других окон). Так же, как и ранее, удалим данную форму из списка автоматически создаваемых форм.

2. Добавим на форму две надписи типа Label, задав им имена «XLabel» и «YLabel» (рис. 4.8).



Рис. 4.8. Внешний вид немодальной формы на этапе разработки

2. Включим в предложение USES секции интерфейса модуля главного окна ссылку на имя модуля формы PanelForm. Определим обработчик события OnClick для команды «Панель» из меню «Сервисы» главного окна:


```

procedure TMainForm.ServicePanelMenuClick(Sender: TObject);
begin
    // проверим, не является ли указатель объекта «пустым»
    // функция Assigned эквивалентна оператору « <> nil»
    if Assigned (PanelForm) then
        // Немодальное окно уже создано, тогда
        // оно только получит фокус
        PanelForm.Show
    // в противном случае
    else
        begin
            try
                // создаем объект формы
                PanelForm := TPanelForm.Create(Self) ;
                // и показываем ее
                PanelForm.Show;
            // блок except будет выполнен, если
            // в блоке try произошла ошибка
            except
                // если создать не смогли, то обнуляем указатель
                PanelForm := nil;
            end;
        end;
    end;

```

Данная процедура позволяет безопасно создать немодальное диалоговое окно. Основная задача, которая возникает при этом – проверить, был ли создан объект формы ранее. Задача эта обусловлена тем, что пользователь однажды открыв немодальное окно, может еще выполнить ту же команду позднее, поскольку пользователю доступно главное окно приложения. В связи с этим процедура проверяет указатель объекта PanelForm на равенство NIL. Если форма не

была создана ранее, ее объектная переменная равна значению пустого указателя NIL.

3. В модуле новой формы определим обработчик OnClose объекта формы. Этим обработчик вызывается когда форма закрывается пользователем, например, путем нажатия системной кнопки в заголовке окна .

```
procedure TPanelForm.FormClose(Sender: TObject; var Action:
                                TCloseAction);

begin
    // указываем на то, что форма будет
    // автоматически уничтожена своим владельцем
    // после ее закрытия
    Action := caFree;
    PanelForm := nil;
end;
```

4. При перемещении курсора мыши в области какого-либо элемента управления операционная система (ОС) автоматически формирует сообщение, в структуре которого содержится информация о экранных координатах курсора. Это сообщение может регистрироваться и обрабатываться прикладной программой.

В Delphi за это отвечает механизм обработчиков событий, поскольку все сообщения ОС трактуются как события компонентов, которые могут быть обработаны с помощью специальных процедур.

Определим в секции интерфейса класса TMainForm и реализуем в секции реализации следующую процедуру, которая и будет назначена в качестве обработчика события OnMouseMove (при перемещении курсора мыши) компонента Image.

```
procedure TMainForm.ImageMouseMove(Sender: TObject; Shift: TShiftState;
X, Y: Integer);

begin
    // если немодальное окно на месте
    if Assigned(PanelForm) then
        begin
            // цвет точки под курсором совпадает с цветом линии синусоиды
```

```

if Image.Picture.Bitmap.Canvas.Pixels[x,y] = LineColor then
begin
    // подсвечиваем надписи в окне PanelForm красным цветом
    PanelForm.XLabel.Font.Color := clRed;
    PanelForm.YLabel.Font.Color := clRed;
end
// если курсор не попал на линию
else
begin
    // подсвечиваем надписи черным цветом
    PanelForm.XLabel.Font.Color := clWindowText;
    PanelForm.YLabel.Font.Color := clWindowText;
end;
// выводим текущие экранные координаты под курсором
PanelForm.XLabel.Caption := 'X = ' + IntToStr(x);
PanelForm.YLabel.Caption := 'Y = ' + IntToStr(y);
end;
end;

```

5. В процедуру ActImageAreaMenuClick (реализует динамическое создание компонента Image) необходимо добавить следующий код, который динамически назначает процедуру обработки событию OnMouseMove компонента Image

```

. . .
Image.OnMouseMove := ImageMouseMove;

```

6. Для корректной работы программы необходимо в начале выполнять инициализацию поля LineColor класса TMainForm. Это можно сделать в обработчике события OnCreate объекта формы MainForm. Данное событие наступит сразу же после окончания работы конструктора класса формы или конструктора предка, если собственный конструктор формы не определен.

```

procedure TMainForm.FormCreate(Sender: TObject);
begin
    LineColor := clBlack;
end;

```


7. Наконец, после внесения указанных выше изменений приложение может собрано и протестировано. Перемещение указатель мыши в области компонента Image при открытой форме PanelForm приводит к отображению экранных координат под курсором. Координаты пересечения курсора с линией тренда графика отображаются красным цветом (рис. 4.9).



Рис. 4.9. Результаты работы программы

3.4. Задания

1. Внимательно изучить теоретический материал из раздела 2.1.
2. Воспроизведите код всех примеров, приведенных в теоретической части работы.
3. Выполните все этапы из практической части лабораторной работы.

4. Контрольные вопросы

1. Какие основополагающие задачи позволяет решить технология объектно-ориентированного программирования?
2. В чем состоит смысл ссылочной природы объектов в ОП?
3. Что такое метод класса и в чем его отличие от обычных методов?
4. Какие преимущества дает технология наследования ООП?

Лабораторная работа № 5

ВЗАИМОДЕЙСТВИЕ ПРИКЛАДНОЙ WINDOWS- ПРОГРАММЫ С ДРАЙВЕРАМИ УСТРОЙСТВ

1. Цель работы

Цель данной лабораторной работы состоит в том чтобы, ознакомить студентов с практическими аспектами построения прикладных программ, в которых предполагается организация программного взаимодействия с аппаратными средствами с использованием драйверов устройств, функционирующих в режиме ядра ОС.

В рамках данной работы студент должен также кратко ознакомиться с основными положениями архитектуры драйверной модели Windows. На примере использования простейшего драйвера логического устройства управляющего, LPT-портом, студент должен изучить основные подходы к реализации программ управления аппаратурой и приобрести соответствующие практические навыки.

2. Основные теоретические сведения

2.1 Компоненты подсистемы ввода/вывода Windows 2000

Согласно целям, поставленным при разработке Windows 2000, подсистема ввода/вывода должна обеспечивать приложениям абстракцию устройств – как физических, так и программных (виртуальных и логических) – и при этом предоставлять следующую функциональность:

- стандартные средства безопасности и именованя устройств для защиты разделяемых ресурсов;
- высокопроизводительный асинхронный пакетный ввод/вывод для поддержки масштабируемых приложений;
- сервисы для написания драйверов на языках программирования высокого уровня;
- поддержку многоуровневой модели и расширяемости для добавления драйверов, модифицирующих поведение других драйверов или устройств без внесения изменений в них;

- динамическую загрузку и выгрузку драйверов для рационального расходования системных ресурсов;
- поддержка технологии Plug and Play, благодаря которой система находит и устанавливает драйверы для нового оборудования и выделяет им необходимые аппаратные ресурсы.

Для реализации этой функциональности подсистема ввода/вывода состоит из нескольких компонентов исполнительной системы ядра и драйверов устройств (рис. 5.1):

- центральное место в этой подсистеме занимает *диспетчер ввода/вывода (Input-Output Manager)*. Он подключает приложения и системные компоненты к виртуальным, физическим и логическим устройствам драйверов, а также реализует инфраструктуру, поддерживающую драйверы устройств.

- *Драйверы устройств (device drivers)*, которые, как правило, предоставляет интерфейс ввода-вывода для устройств конкретного типа. Драйверы принимают от диспетчера ввода-вывода команды, предназначенные управляемым ими устройствам и уведомляют диспетчер ввода-вывода о выполнении этих команд. Драйверы часто используют этот диспетчер для пересылки команд ввода/вывода другим драйверам, задействованным в реализации интерфейса того же устройства и участвующим в управлении им.

- *Диспетчер PnP (PnP manager)* работает в тесном взаимодействии с диспетчером ввода-вывода и *драйвером шины (bus driver)* – одной из разновидностей драйверов устройств. Он управляет выделением аппаратных ресурсов, а также распознает устройства и реагирует на их подключение или отключение. Диспетчер PnP и драйверы шины отвечают за загрузку соответствующего драйвера при обнаружении нового устройства.

- *Реестр (Registry)* служит в качестве базы данных, в которой хранится описание основных устройств, подключенных к системе, а также параметры инициализации драйверов и конфигурационные настройки.

- Для установки драйверов используется INF-файлы. Они связывают конкретное аппаратное устройство с драйвером, который берет на себя ведущую роль в управлении этим устройством. Содержимое INF-файла состоит из инструкций, описывающих соответствующее устройство, исходное и целевое местоположение файлов драйвера, измерения, которые нужно внести в реестр, при установке драйвера, и информацию о зависимостях драйвера.

- Уровень аппаратных абстракций (HAL) изолирует драйверы от специфических особенностей конкретных процессоров и контроллеров прерываний,

поддерживая интерфейс прикладного программирования, скрывающий межплатформенные различия. В сущности HAL является драйвером шины для тех устройств на материнской плате компьютера, которые не контролируются другими драйверами.

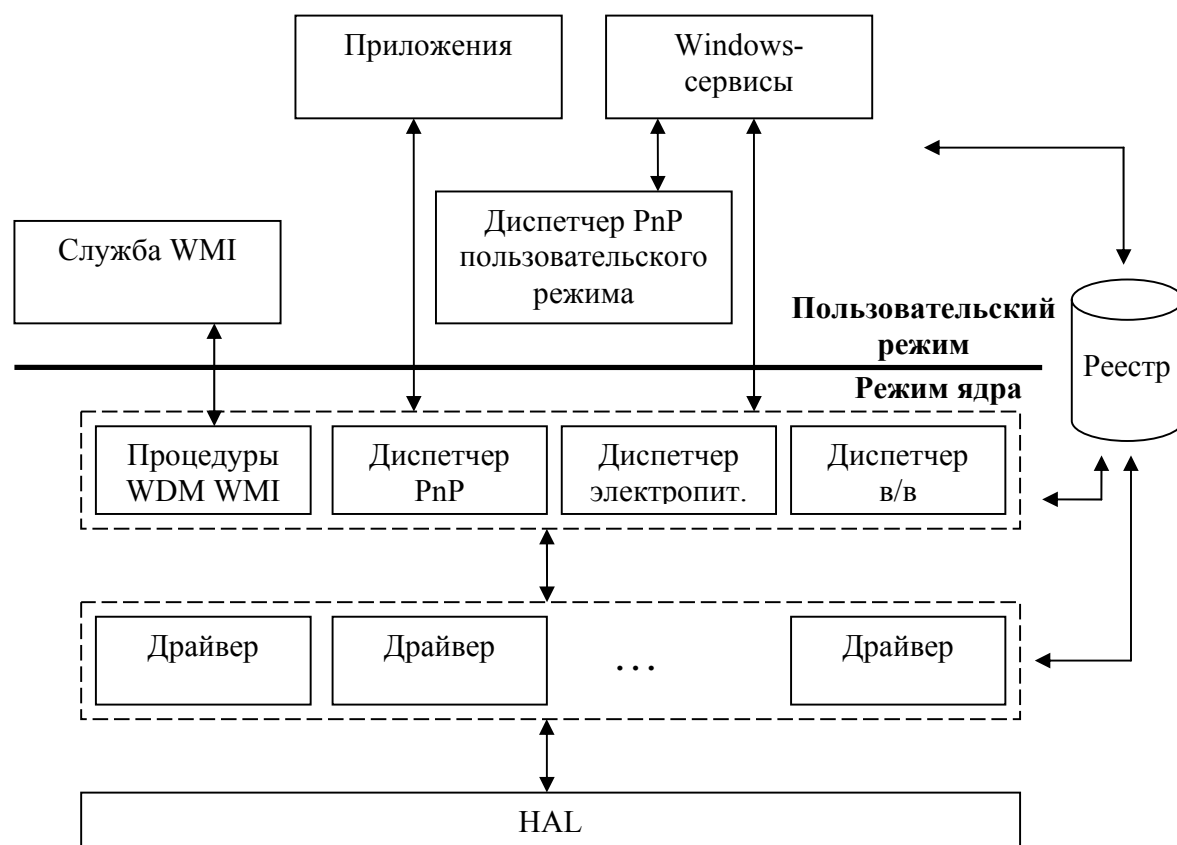


Рис. 5.1. Компоненты подсистемы ввода/вывода

2.2. Диспетчер ввода/вывода и типовая обработка ввода/вывода

Диспетчер ввода/вывода (в/в) определяет модель доставки запросов на ввод/вывод драйверам устройств. Подсистема в/в управляется пакетами. Большинство запросов в/в представляется *пакетами запросов в/в* (*I/O request packets, IRP*), передаваемых от одного компонента подсистемы ввода-вывода другому. Подсистема в/в позволяет индивидуальному потоку приложения управлять сразу несколькими запросами на в/в. IRP – это структура данных, которая содержит информацию, полностью описывающую запрос ввода/вывода. В том числе, со-

держит номер специальной диспетчерской функции драйвера, которая и будет выполнена в ответ на этот запрос.

Диспетчер ввода-вывода создает IRP, представляющий операцию в/в, передает указатель на IRP соответствующему драйверу и удаляет пакет по завершении операции. Драйвер, получивший IRP, выполняет указанную в пакете операцию и возвращает IRP диспетчеру в/в, чтобы тот завершил эту операцию, либо передал пакет другому драйверу для дальнейшей обработки.

Диспетчер в/в не только создает и уничтожает IRP, но и содержит общий для различных драйверов код, который они используют для обработки в/в. Драйверы устройств могут вызывать около сотни функций, предоставляемых диспетчером в/в.

Унифицированный модульный интерфейс драйверов позволяет диспетчеру вызывать любой драйвер, ничего не зная о его структуре и его внутреннем устройстве. Операционная система обрабатывает запросы на в/в, так будто они адресованы файлам. Драйвер преобразует запросы к виртуальному файлу в запросы, специфичные для устройства. Драйверы также могут вызывать друг друга, используя диспетчер обеспечивая, многоуровневую независимую обработку запросов в/в.

Большинство операций в/в не требует участия всех компонентов подсистемы. Запрос на в/в формируется приложением, выполняющим соответствующую операцию в/в (например чтение данных с устройства). Такие операции обрабатываются диспетчером в/в, одним или несколькими драйверами устройств и HAL. Приложения выполняют любые операции в/в так, как если бы это были операции в/в в обыкновенный файл дисковой файловой системы. ОС абстрагирует все запросы на в/в, скрывая тот факт, что конечное устройство в/в может и не быть устройством с файловой структурой. Это позволяет обобщить интерфейс между приложениями и устройствами.

Таким образом, любой источник или приемник в/в представляется для приложения как некий виртуальный файл. Все считываемые и записываемые данные передаются простыми потоками байтов, направляемыми в виртуальные файлы. Приложения пользовательского режима вызывают документированные функции, которые в свою очередь обращаются к внутренним функциям подсистемы в/в. Запросы, адресованные виртуальным файлам, диспетчер в/в динамически направляет драйверам устройств в виде IRP. Базовая схема обработки запроса на в/в может быть представлена в виде рис. 5.2.



Рис. 5.2. Общая схема прохождения операции в/в в Windows

2.3. Драйверы устройств и их классификация

Драйверы могут работать в двух режимах: в пользовательском или режиме ядра. Windows поддерживает несколько типов драйверов пользовательского режима, в частности: драйверы виртуальных устройств и драйверы принтеров.

Драйверы виртуальных устройств используются для эмуляции устройств для 16-разрядных программ MS-DOS при обращении таких программ к портам в/в. Эти драйверы транслируют вызовы программ MS-DOS в вызовы Windows-функций, которые передаются реальным драйверам устройств режима ядра. Наличие таких драйверов обусловлено тем, что Windows работает в защищенном режиме и прямое обращение приложений к аппаратным средствам как имеет место быть в MS-DOS, невозможно.

Драйверы принтеров транслируют аппаратно-независимые запросы на графические операции в команды, специфичные для принтера. Далее эти команды перенаправляются драйверу режима ядра, например драйверу парал-

лельного порта (paraport.sys) или драйверу порта принтера на шине USB (USB-print.sys).

Драйверы режима ядра можно разбить на несколько основных категорий: драйверы файловых систем, PnP-драйверы и драйверы, не отвечающие спецификации PnP.

Драйверы файловой системы принимают запросы на в/в и выполняют их, формируя специфические запросы драйверам устройств массовой памяти или сетевым драйверам.

PnP-драйверы работают с оборудованием и связаны с диспетчером электропитания и диспетчером PnP. В их число входят драйверы для устройств массовой памяти, видеоадаптеров, устройств ввода и сетевых адаптеров.

Драйверы, не отвечающие спецификации PnP, также называются расширениями ядра. Расширяют функциональность системы, предоставляя доступ из пользовательского режима к сервисам и драйверам режима ядра.

Подкатегория драйверов режима ядра также подразделяется на группы в зависимости от модели, на которой они основаны. По этому признаку можно выделить унаследованные драйверы и WDM-драйверы.

Унаследованные драйверы написаны для Windows NT 4, но могут быть использованы в старших версиях ОС на основе ядра NT (Windows 2000, Windows XP, Windows 2003 Server). Такие драйверы не поддерживают многоуровневой обработки запросов и не могут быть интегрированы с диспетчером электропитания и диспетчером PnP.

WDM-драйверы отвечают спецификации Windows Driver Model (WDM). Эта модель требует от драйверов поддержки управления электропитанием, Plug and Play. Существует три типа драйверов WDM в зависимости от их роли в обслуживании запросов к устройствам: драйверы шин, функциональные драйверы и драйверы фильтров.

Драйверы шин управляют логическими или физическими шинами, такими как, PCMCIA, PCI, USB, IEEE 1394, ISA, SCSI и IDE. Драйвер шины отвечает за распознавание устройств, подключенных к управляемой ими шине, оповещение о них диспетчера PnP и управление параметрами электропитания шины.

Функциональные драйверы – управляют конкретным типом устройств, подсоединенным к той или иной шине. Драйверы шин предоставляют функциональным драйверам доступ к своим устройствам через диспетчер PnP. Функциональным считается драйвер, экспортирующий рабочий интерфейс устрой-

ства операционной системе. Как правило, это драйвер, больше других знающий о функционировании определенного устройства.

Драйверы фильтров занимают более высокий логический уровень, чем функциональные драйверы, они дополняют функциональность или изменяют поведение устройства, либо другого драйвера.

2.4. Унифицированная структура драйвера режима ядра

Драйвер устройства состоит из набора процедур, вызываемых на различных этапах обработки запроса диспетчером в/в. Основные процедуры драйвера показаны на рис. 5.3.

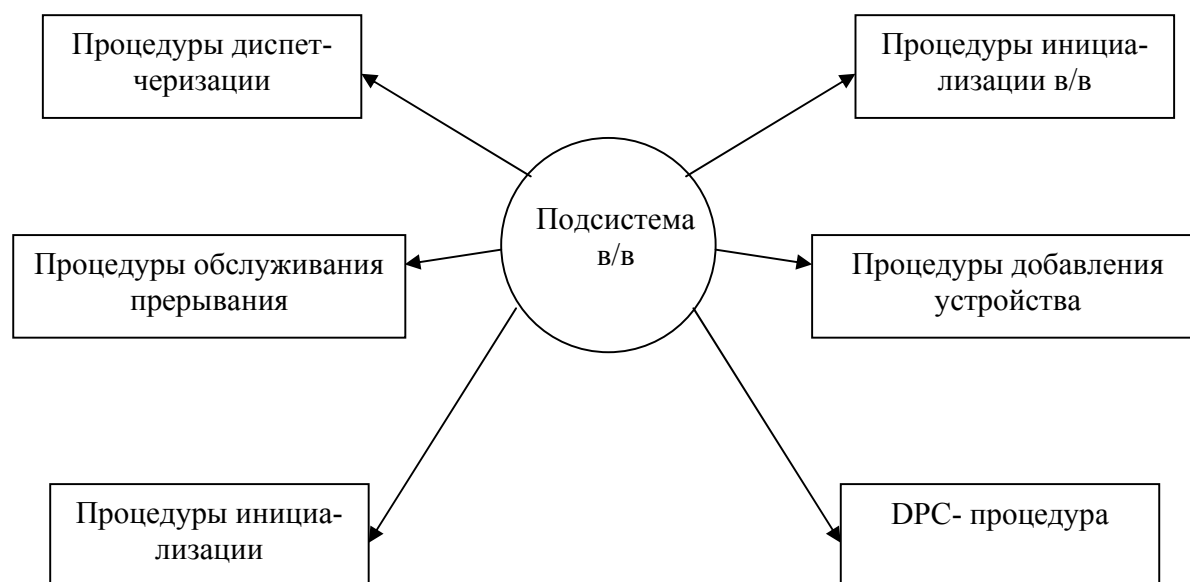


Рис. 5.3. Основные процедуры драйвера устройства

Инициализирующая процедура вызывается диспетчером в/в при загрузке этого драйвера в операционную систему. Данная процедура регистрирует остальные процедуры драйвера в диспетчере в/в, заполняя соответствующей информацией системные структуры данных, и выполняет необходимую глобальную инициализацию.

Процедура добавления устройств реализуется только в драйверах, поддерживающих PnP. Вызывая эту процедуру, диспетчер PnP посылает уведомление при обнаружении устройства, за которое отвечает данный драйвер. Выполняя эту процедуру, драйвер создает специальный объект ОС типа «устройство».

Процедуры диспетчеризации – это основные функции, предоставляемые драйвером, например для открытия и закрытия устройства, чтения и записи данных, а также реализации других возможностей устройства, файловой системы или сети. Диспетчер в/в, вызванный для выполнения операции в/в, генерирует IRP и обращается к драйверу через одну из его процедур диспетчеризации. Запросы в/в в виде IRP обрабатываются именно этими диспетчерскими процедурами.

Процедура обслуживания прерывания (Interrupt Service Routine, ISR) вызывается, когда устройство генерирует прерывания и диспетчер прерываний ядра передает управление этой процедуре. В модели в/в Windows ISR выполняется с высоким приоритетом (Device IRQL), поэтому они должны выполнять минимум действий во избежание слишком длительной блокировки прерываний более низкого уровня. Для выполнения остальной части обработки прерывания процедура ISR должна поставить в очередь соответствующую ей так называемую процедуру *отложенного вызова (Deferred Procedure Call – DPC)*.

DPC-процедура обработки прерывания выполняет основную часть обработки прерывания обработки прерывания, оставшуюся после выполнения ISR.

2.5. Объект «драйвер» и объект «устройство». Виды устройств. Интерфейс с прикладной программой

Для управления подсистемой в/в и ее компонентами в Windows используется абстракция объектов, когда драйвер и устройства, за управление которыми он отвечает, представляется объектами ОС. Объект «драйвер» логически представляет отдельный драйвер в системе. Именно от этого объекта диспетчер в/в получает адрес точки входа в драйвер и адреса процедур диспетчеризации запросов в/в.

В свою очередь объект «устройство» представляет физическое или логическое устройство в системе и описывает его характеристики и реализует действия по его обслуживанию.

Диспетчер в/в создает объект «драйвер» при загрузке в систему соответствующего драйвера и вызывает его инициализирующую процедуру, которая записывает в атрибуты объекта точку входа этого драйвера.

После загрузки драйвер может создавать одни или несколько объектов типа «устройство» для представления устройств.

В одном случае объект «устройство» может быть ассоциирован с физическим устройством и должен реализовывать программный интерфейс с аппаратными средствами этого физического устройства (памятью, портами в/в). Кроме интерфейса с физическим устройством объект «устройство» драйвера может, опираясь на функциональность физического устройства (соответствующего объекта типа «устройство») реализовать виртуальное устройство, т.е. имеющее как будто некоторые физические свойства, которых на самом деле отсутствуют. И наконец, объект «устройство» может реализовать некоторое воображаемое логическое устройство, выступающее для приложения как некоторое устройство с определенным набором функциональности, которая не имеет никакого отношения к аппаратуре.

Объекты устройств отвечают за реализацию набора абстрактных сервисов (чтение, запись, функции управления), которые предоставляют доступ к имеющимся в системе аппаратным средствам, виртуальным или логическим устройствам. **Причем такой интерфейс является строго унифицированным, поэтому нет никакого различия в запросах на операции в/в, применяемых к устройствам различных видов.**

Большинство устройств являются именованными, несмотря на то, что разрешены и безымянные устройства. Устройство может взаимодействовать с другим устройством, если имя другого устройства известно. Для того, чтобы приложение могло устанавливать связь с устройством, устройство должно ссылаться на одну или более символических ссылок. Для того, чтобы понять, что такое *символическая ссылка* (*symbolic link*), нужно знать, что система поддерживает специальную директорию системных объектов различных видов, таких, как драйверы, устройства, события, **семафоры и, в том числе, символические ссылки.** Директория системных объектов может просматриваться с помощью специальных утилит, например утилиты, написанной Марком Руссиновичем – WinObj (рис. 5.4).

Символическая ссылка является специальным типом объектов, которые обеспечивают косвенную ссылку на другой объект в системе. Специальная поддиректория объектной системной иерархии резервируется для символических ссылок на объекты устройств. Символические ссылки в этой поддиректории видимы приложениями через Win32 API. Для заданного устройства здесь может быть несколько символических ссылок, которые указывают на него, или их может и не быть вовсе.

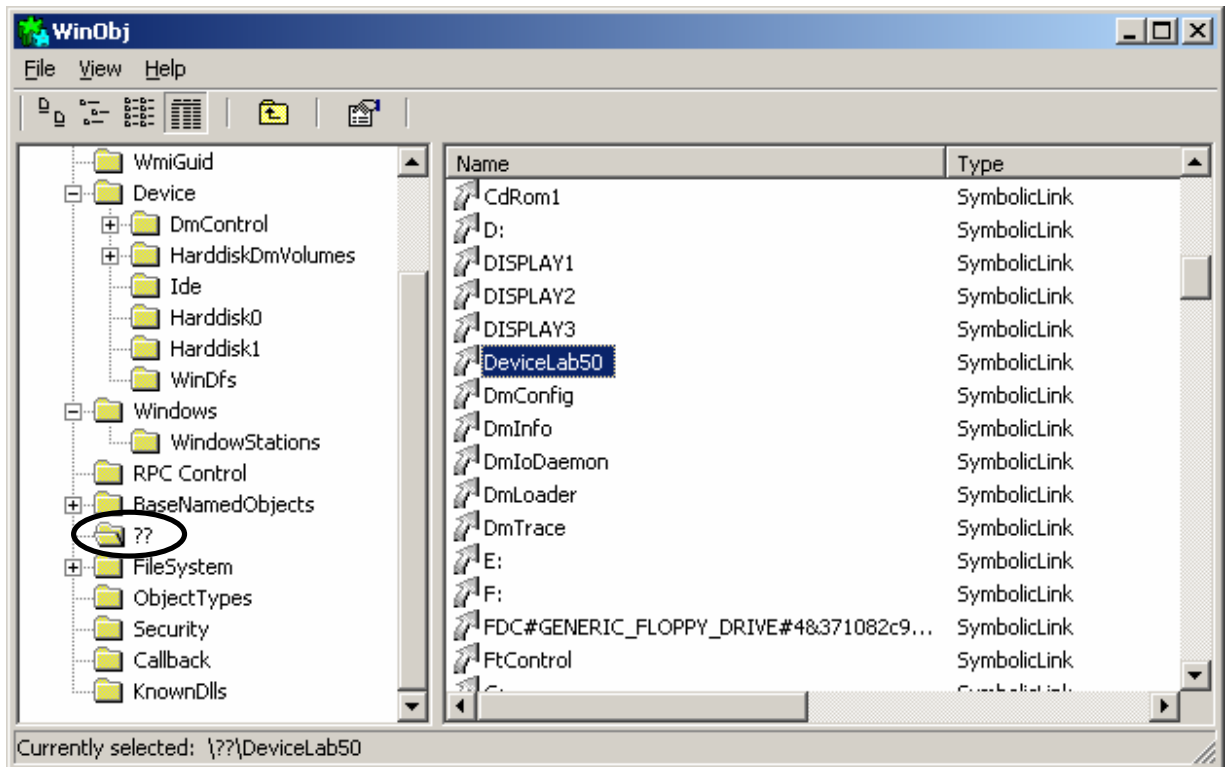


Рис. 5.4. Директория имен системных объектов

Используя такую символическую ссылку на объект устройства определенного драйвера, прикладная программа может получить доступ к этому устройству с помощью API функции CreateFile. Например:

```

var
    // указатель на открываемое устройство драйвера
    hLAB5: THandle;
    // буфер на 16 байт для операций чтения/записи
    Buffer: array[1..16] of char;
    // число байтов в буфере после операции
    NumberOfBytesWritten: Cardinal;
...
// открываем устройство драйвера и получаем его дескриптор
hLAB5 := CreateFile('\\.\DeviceLab50', GENERIC_READ,
                    FILE_SHARE_READ,
                    nil, OPEN_EXISTING, 0, 0);

```

Необходимо отметить, что в качестве первого параметра функции `CreateFile` указывается не имя файла драйвера, а символическая ссылка на объект устройства драйвера. В этом случае при вызове `CreateFile` вместо создания или открытия дискового файла открывается устройство драйвера, которое имеет символическую ссылку с названием `DeviceLab50`. При этом диспетчер в/в формирует IRP со специальным кодом (`IRP_MJ_CREATE`) и передает его драйверу, которому принадлежит устройство. К этому моменту драйвер должен уже быть загружен в системе, поскольку вызов `CreateFile` приводит не к загрузке драйвера, а, что важно для понимания, к получению доступа к экземпляру объекта устройства драйвера. В ответ на вызов `CreateFile` драйвер вызывает диспетчерскую функцию отвечающую за обработку данного IRP объектом устройства, и в конечном итоге возвращает дескриптор пользовательского режима для устройства, который позволит приложению выполнять операции в/в с ним.

Дальнейшие действия с открытым таким образом устройством драйвера зависят от типа операций, которые поддерживаются им (чтение, запись, операции управления), и типов управления в/в (буферизированный или небуферизированный).

Если устройство драйвера поддерживает операцию чтения и, соответственно, может обслуживать IRP с основным кодом функции `IRP_MJ_READ`, то доступ к этой операции становится возможным с помощью функции Windows API – `ReadFile`:

```
// перед выполнением операции проверим, было ли открыто устройство
if hLAB5 = INVALID_HANDLE_VALUE then
begin
    ShowMessage('Устройство DeviceLab50 не найдено');
    Exit;
end;
// пытаемся считать 16 байт из устройства в буфер приложения
if not ReadFile(hLAB5, Buffer, 16, NumberOfBytesWritten, nil) then
    // если ошибка, то выдаем системное сообщение
    MessageDlg(SysErrorMessage(GetLastError), mtError, [mbOK], 0);

// в противном случае отображает полученные данные в ASCII формате
ShowMessage('Данные полученные от драйвера: ' + string(Buffer));
```

В случае если, устройство имеет диспетчерскую функцию, отвечающую за обработку операцию записи (IRP_MJ_WRITE), то для доступа к этой операции необходимо использовать функцию Windows API – WriteFile, которая позволяет записать данные из буфера пользовательской программы в устройство.

Если устройство было открыто в прикладной программе, то по окончании работы с ним дескриптор на устройство должен быть освобожден с помощью функции API – CloseHandle. Например:

```
// закрываем логическое устройство
// (ВНИМАНИЕ! мы не выгружает при этом драйвер)
CloseHandle(hLAB5);
```

Невыполнение этого действия может привести к невозможности использования устройства драйвера другими пользовательским потоками, в случае если драйвер может создавать только один экземпляр устройства.

3. Практическая часть

3.1. Управление простыми устройствами с использованием драйвера режима ядра

В качестве примера возьмем простой одноуровневый драйвер, реализующий логическое устройство, функциональность которого заключается в выполнении операций с регистром данных LPT-порта компьютера. Данный драйвер поддерживает прямые небуферизированные операции в/в и имеет четыре диспетчерские функции объекта типа «устройство»: Create, Read, Write и Close. Указанные функции отвечают за обработку запросов в/в с кодами основных функций IRP_MJ_CREATE, IRP_MJ_READ, IRP_MJ_WRITE и IRP_MJ_CLOSE.

Данный драйвер написан с использованием библиотеки классов для языка C++ DriverWorks компании Compuware Corporation, которая входит в состав продукта DriverStudio.

Данный программный комплекс разработчика включает в свой состав программное окружение, утилиты и программную инфраструктуру для разработки драйверов устройств Windows. DriverStudio позволяет использовать Microsoft Visual Studio в качестве платформы и интегрированной среды разработ-

ки. Кроме того она включает несколько инструментов, упрощающих разработку драйверов, таких как DriverMonitor, EzDriverInstaller, Wdm Sniffer.

Самым главным является то, что DriverStudio также включает библиотеки классов C++, такие как DriverWorks и DriverNetworks, которые упрощают разработку драйверов.

Классы, входящие в состав этих библиотек инкапсулируют функциональность таких объектов ОС, как драйверы, устройства, пакеты запроса в/в, прерывания, объекты синхронизации и др. Использование таких библиотек существенно упрощает программирование драйверов устройств, позволяя разработчику сосредоточиться на сути решаемой задачи, избежав утомительного кодирования типовых фрагментов программного интерфейса драйвера.

Ниже приведены отдельные фрагменты исходного текста драйвера написанные на C++ в Microsoft Visual Studio .NET 2003, включающие диспетчерские функции, отвечающие за обработку запросов основных операций в/в. Данный драйвер будет использован для реализации программы управления простым устройством (линейка светодиодов), подключаемым к LPT-порту компьютера.

```
// объявление членов класса CLeoLPTDevice
protected:
    . . .
    // диапазон регистров в/в устройства
    // в данном случае LPT на шине ISA
    KIoRange m_LeoLPT;
    . . .
//-----
// ДИСПЕТЧЕРСКАЯ ФУНКЦИЯ CREATE
NTSTATUS CLeoLPTDevice::Create(KIrp I)
{
    I.Information() = 0;
    // инициализируем объект типа «диапазон адресов в/в»
    // на шине ISA (мост PCI-ISA) с базовым адресом 0x378 (LPT)
    // из трех однобайтных регистров
    m_LPT.Initialize(Isa, 0, 0x0378, 3);
    return I.Complete(STATUS_SUCCESS);
}
```

```

//-----
// ДИСПЕТЧЕРСКАЯ ФУНКЦИЯ READ
NTSTATUS CLeoLPTDevice::Read(KIrp I)
{
    // здесь можно проверить корректность передаваемых параметров
    if (FALSE)
    {
        I.Information() = 0;
        return I.Complete(STATUS_INVALID_PARAMETER);
    }

    // Если попытка читать 0 элементов, то прекращаем
    // дальнейшую обработку
    if (I.ReadSize() == 0)
    {
        I.Information() = 0;
        return I.Complete(STATUS_SUCCESS);
    }

    // по умолчанию статус типа «все хорошо»
    NTSTATUS status = STATUS_SUCCESS;
    // Декларируем объект типа "память" и иницилируем его с помощью
    // структуры Memory Descriptor List (MDL) из IRP
    KMemory Mem(I.Mdl());

    // Используем этот объект для создания указателя на буфер
    // вызывающей прикладной программы
    PUCCHAR pBuffer = (PUCCHAR) Mem.VirtualAddress();
    // Запрашиваемое число читаемых байтов
    ULONG dwTotalSize = I.ReadSize(CURRENT);
    // Число прочитанных байтов
    ULONG dwBytesRead = 0;

    // используя объект типа KIoRange
    // читаем из порта один байт и помещаем его в буфер
    m_LeoLPT.inb(0, pBuffer, 1);
    // всегда читаем только один байт

```

```

    dwBytesRead = 1;
    I.Information() = dwBytesRead;
    return I.Complete(status);
}

//-----
// ДИСПЕТЧЕРСКАЯ ФУНКЦИЯ WRITE
NTSTATUS CLeoLPTDevice::Write(KIrp I)
{
    if (FALSE)
    {
        I.Information() = 0;
        return I.Complete(STATUS_INVALID_PARAMETER);
    }
    if (I.WriteSize() == 0)
    {
        I.Information() = 0;
        return I.Complete(STATUS_SUCCESS);
    }
    NTSTATUS status = STATUS_SUCCESS;

    KMemory Mem(I.Mdl());
    PCHAR pBuffer = (PCHAR) Mem.VirtualAddress();
    ULONG dwTotalSize = I.WriteSize(CURRENT);
    ULONG dwBytesSent = 0;
    // вывод в порт содержимого pBuffer
    m_LeoLPT.outb(0, *pBuffer);
    I.Information() = dwBytesSent;
    return I.Complete(status);
}

```

Разработаем клиентскую прикладную программу, которая будет управлять устройством, посылая в LPT-порт, через драйвер устройства управляющие данные. Для реализации программы указанную ниже последовательность действий:


1. Создадим новый проект приложения в среде Delphi (рис. 5.5). Переименуем основное окно (1) приложения задав, имя в инспекторе объектов – «MainForm». Установим свойству BorderStyle данного окна значение bsDialog.




Рис. 5.5. Внешний вид окна приложения на этапе разработки

2. Добавим на форму главного окна приложения следующие элементы управления и зададим им необходимые свойства:

- две простых командных кнопки типа Button (2 и 3) с именами «WriteButton» и «ReadButton» соответственно;

- три радиокнопки (4, 5, 6) типа RadioButton (класс TRadioButton)  с именами «LeftFireRadioButton», «RightFireRadioButton» и «IncrementalFireRadioButton»

- группу (7) из восьми флажков типа CheckButton (класс TCheckButton)  с произвольными именами по умолчанию. Начиная с крайнего правого, назначим свойству Tag каждого флажка из группы последовательные числовые значения от 0 до 7.

- подпись к группе флажков типа Label, как показано на рис. 5.5.

3. Реализуем подпрограммы и обработчики, отвечающие за реализацию логики работы программы.

3.1. С кнопкой WriteButton ассоциируем обработчик события OnClick, исходный текст которого представлен ниже:

```
procedure TMainForm.WriteButtonClick(Sender: TObject);
var
```

```

hLeoLPT: THandle;           // дескриптор устройства
NumBytesWritten: Cardinal; // число записанных байтов
i: Byte;                   // счетчик цикла
Buffer: Byte;              // передаваемый байт
begin
    // открываем устройство драйвера
    hLeoLPT := CreateFile('\.\LeoLPTDevice0',GENERIC_WRITE,
                          FILE_SHARE_READ,
                          nil, OPEN_EXISTING, 0, 0);
    // если драйвер не загружен
    if (hLeoLPT = INVALID_HANDLE_VALUE) then
    begin
        ShowMessage ('Устройство LeoLPTDevice0 не найдено');
        Exit;
    end;
    Buffer := 1;
    // записываем в регистр данных (0x378) LPT порта
    // с интервалом в 50 мс
    for i := 0 to 255 do
    begin
        // записываем побайтно в устройство
        if not WriteFile(hLeoLPT, Buffer, 1, NumBytesWritten, nil) then
            MessageDlg(SysErrorMessage(GetLastError), mtError, [mbOK], 0);
        // установка битов воображаемого порта на экране программы
        SetBitVirtualPort(Buffer);
        // проверяем тип клиентской операции с буфером
        // для выполнения операций используем ассемблерные вставки
        if LeftFireRadioButton.Checked then
        asm
            // циклический сдвиг влево через перенос на один бит
            ROL Buffer, 1
        end
        else if RightFireRadioButton.Checked then
        asm
            // циклический сдвиг влево через перенос на один бит
            ROR Buffer, 1
    end

```

```

end
else if IncrementalFireRadioButton.Checked then
asm
    // инкремент буфера
    INC Buffer
end;
    // позволяем приложению обрабатывать сообщения (перерисовка)
    // в противном случае экран не будет перерисовываться
    // во время работы цикла
Application.ProcessMessages;
    // ждем 100 мс
    Sleep(100);
end;
    // закрываем логическое устройство
    CloseHandle(hLeoLPT);
end;

```

3.2. В секции интерфейса класса TMainForm в разделе PRIVATE задекларируем метод SetBitVirtualPort и реализуем его в секции реализации. Данный метод позволяет установить экранный флажок воображаемого регистра порта в соответствии с состоянием битов буфера, записываемого в реальный физический регистр LPT-порта. Исходный текст данного метода представлен ниже:

```

procedure TMainForm.SetBitVirtualPort(Data: Byte);
var
    i, temp: Byte;
begin
    //сканируем все компоненты на форме
    //эта возможность обусловлена тем, что Delphi
    //обеспечивает доступ к информации времени исполнения,
    //и потому что все компоненты принадлежат к общему предку - TComponent
for i:= 0 to Self.ComponentCount - 1 do
begin
    // если компонент принадлежит к классу TCheckBox
    if Self.Components[i] is TCheckBox then
begin

```

```

    // копируем в temp содержимое битового поля,
    // чтобы не потерять установленные биты
temp := Data;
    // ставим галочку на CheckBox, если бит с номером,
    // совпадающим с его тегом в temp установлен в 1
    (Self.Components[i] as TCheckBox).Checked :=
(temp and (1 shl (Self.Components[i] as TCheckBox).Tag)) <> 0
    end;
end;
end;

```

3.3. Назначим обработчик события `OnClick` кнопке `ReadButton`. Данная процедура открывает устройство драйвера и читает один байт из него, отображая состояния прочитанного байта на экране. Исходный текст данного обработчика приведен ниже:

```

procedure TMainForm.ReadButtonClick(Sender: TObject);
var
    hLeoLPT: THandle;           // дескриптор устройства
    NumBytesWritten: Cardinal; // число записанных байтов
    Buffer, i: Byte;           // счетчик цикла и буфер
begin
    // открываем устройство драйвера
    hLeoLPT := CreateFile('\.\LeoLPTDevice0', GENERIC_READ,
        FILE_SHARE_READ, nil, OPEN_EXISTING, 0, 0);
    // если драйвер не загружен
    if (hLeoLPT = INVALID_HANDLE_VALUE) then
        begin
            ShowMessage ('Устройство LeoLPTDevice0 не найдено');
            Exit;
        end;
    // побайтно читаем из устройства
    if not ReadFile(hLeoLPT, Buffer, 1, NumBytesWritten, nil) then
        MessageDlg(SysErrorMessage(GetLastError), mtError, [mbOK], 0);
    // установка битов виртуального порта на экране программы
    SetBitVirtualPort(Buffer);

```

```
// закрываем логическое устройство
CloseHandle(hLeoLPT);
```

end;

4. Выполним сборку проекта. Перед началом тестирования разработанной программы в операционной системе должен быть установлен драйвер (LeoLPT.sys) устройства, с которым будет работать приложение. Для установки драйвера в системе можно воспользоваться простой утилитой, входящей в комплект поставки свободно распространяемой программы PortTalk (универсальный драйвер портов и API для работы с ним), которая называется Driver Install Utility (рис. 5.6). В данной утилите необходимо указать путь к файлу драйвера и произвольное имя для создания учетной записи о драйвере в системном реестре. Затем, выбрав команду Install, создать запись в реестре, а с помощью команды Start загрузить драйвер.

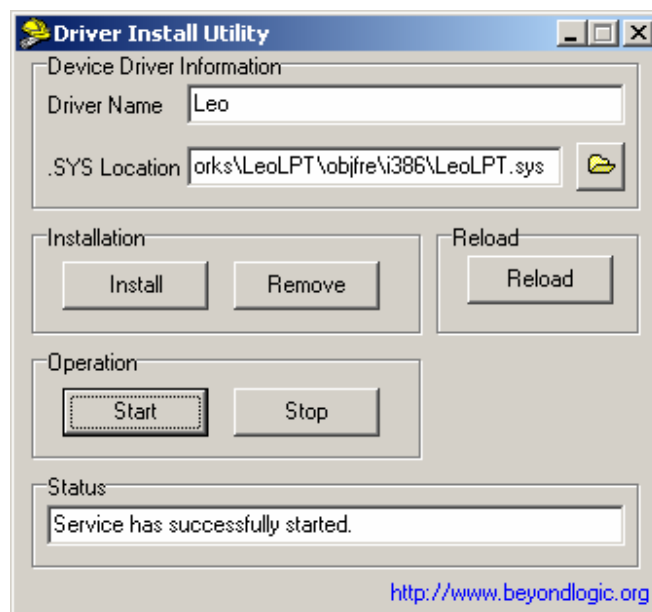


Рис. 5.6. Программа Driver Install Utility

После этого приложение полностью работоспособно. Однако для более убедительной демонстрации функциональности драйвера и разработанной управляющей программы к LPT порту можно подключить тривиальное устройство, представляющее собой линейку из восьми светодиодов с ограничительными резисторами. Линейка подключена к выводам принтерного разъема, которые соответствуют линиям данных интерфейса Centronics (рис. 5.7).

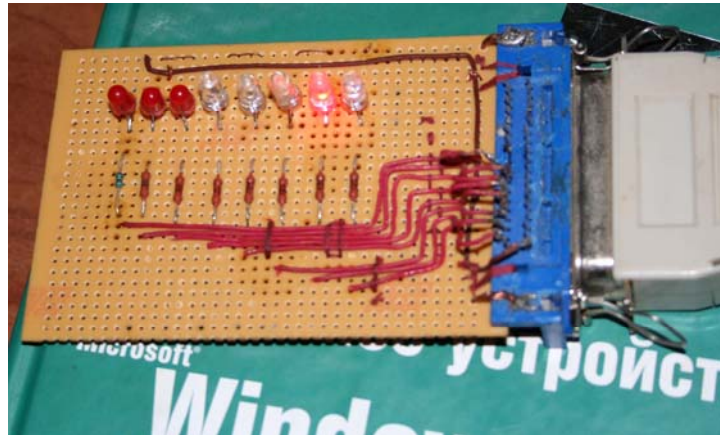


Рис. 5.7. Устройство для тестирования работы драйвера и программы

При нормальном функционировании драйвера, программы и устройства после запуска программы и выбора команды циклической записи можно будет наблюдать «бегущий огонь» слева направо или справа налево в зависимости от установленной опции или же поразрядное зажигание светодиодов в соответствии в инкрементом выводимых в порт данных.

3.2. Задания

1. Внимательно изучить теоретический материал из раздела 2. Повторить кодирование примеров, приведенных в теоретической части работы.
2. Выполните все этапы из практической части лабораторной работы.

4. Контрольные вопросы

1. Какие разновидности драйверов выделяют в ОС Windows 2000?
2. Назовите основные компоненты подсистемы ввода/вывода ОС.
3. Перечислите основные процедуры драйвера режима ядра.

ЧАСТЬ II. ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИЙ ЗАЩИТЫ ИНФОРМАЦИИ

Лабораторная работа № 6

**СИММЕТРИЧНАЯ КРИПТОГРАФИЯ.
ПРИМЕНЕНИЕ АЛГОРИТМОВ С ЗАКРЫТЫМ КЛЮЧОМ ДЛЯ ЗАЩИТЫ
ИНФОРМАЦИИ В ПРИКЛАДНЫХ ПРОГРАММАХ.
БИБЛИОТЕКА АЛГОРИТМОВ ШИФРОВАНИЯ LOCKBOX2**

1. Цель работы

Цель данной лабораторной работы состоит в том, чтобы познакомить студентов с практическими аспектами применения распространенных криптографических алгоритмов. В работе рассматривается использование широко известной криптографической библиотеки функций и классов для среды разработки Delphi – библиотеки LockBox2 компании TurboPower Professional. В работе рассматриваются примеры использования подпрограмм для формирования ключей, а также подпрограммы симметричного шифрования.

2. Основные теоретические сведения

2.1. Основные понятия криптографической защиты данных

Криптография (cryptography) представляет собой совокупность методов преобразования данных, направленных на то, чтобы защитить эти данные, сделав их бесполезными для пользователей, которым они не предназначены. Методы криптографии позволяют решить две основные проблемы: проблему *конфиденциальности* и проблему *целостности*.

Криптография имеет многовековую историю развития и использования, поэтому в этой области компьютерных наук имеется устойчивая терминология.

Исходное сообщение называется *открытым текстом (plain text)*. Процесс маскирования сообщения, позволяющий скрыть его суть, принято называть *зашифрованием (encipher)*. Зашифрованное сообщение называется *шифртекстом (ciphertext)*. Процедура обратного превращения шифртекста в открытый текст называется *расшифрованием (decipher)*. Науку защиты сообщений называют криптографией, тогда как наука взломов шифртекстов называется *криптоанализом*.

Открытый текст принято обозначать буквой *M (message – сообщение)* или *P (plaintext – открытый текст)*. Открытым текстом может быть поток битов ин-

формации, текстовый файл, точечный рисунок, оцифрованный звук или видео-изображение. В компьютерной криптографии M – это просто двоичные данные. Шифртекст принято обозначать буквой C . Это также двоичные данные, иногда того же размера, что и M , а иногда большего. Функция зашифрования E , оперируя M , создает C , т.е.

$$E(M) = C.$$

В обратном процессе функция расшифрования D , оперируя C , восстанавливает M :

$$D(C) = M.$$

Криптографический алгоритм, называемый также шифром, представляет собой математическую функцию, которая используется для зашифрования и расшифрования информации. Обычно это две связанные функции: одна для зашифрования, а другая для расшифрования. Если защита, обеспечиваемая алгоритмом, основана на сохранении в тайне самого алгоритма, это так называемый *ограниченный* алгоритм. В свою очередь, современная криптография решает проблемы обеспечения секретности не с помощью сокрытия алгоритма, а с помощью *криптографического ключа (key)*, обозначаемого буквой K . Такой ключ может быть любым значением, выбранным из большого множества. Множество возможных ключей называют *пространством ключей*. Ключ используется в обеих операциях – как зашифрования, так и расшифрования, т.е. они зависят от ключа, и это обстоятельство указывается в виде индексов K . Таким образом, теперь функции зашифрования и расшифрования принимают вид:

$$E_K(M) = C.$$

$$D_K(C) = M.$$

При этом справедливо следующее равенство:

$$D_K(E_K(M)) = M.$$

В некоторых алгоритмах для зашифрования и расшифрования используются различные ключи. Иными словами, ключ зашифрования $K1$ отличается от соответствующего ключа расшифрования $K2$. В этом случае:

$$E_{K1}(M) = C.$$

$$D_{K2}(C) = M.$$

При этом справедливо следующее равенство:

$$D_{K2}(E_{K1}(M)) = M.$$

Надежность этих алгоритмов полностью зависит от ключа или ключей, а не от самих алгоритмов. Это означает, что алгоритм может быть опубликован и проанализирован. Программные продукты, использующие этот алгоритм, могут широко распространяться. Причем знание алгоритма злоумышленником не имеет значения – отсутствие конкретного ключа не дает возможность читать сообщения.

Таким образом, *криптосистема* представляет собой совокупность алгоритма, всех возможных открытых текстов, шифртекстов и ключей.

Обобщенная схема криптосистемы может быть представлена в виде (рис. 6.1):

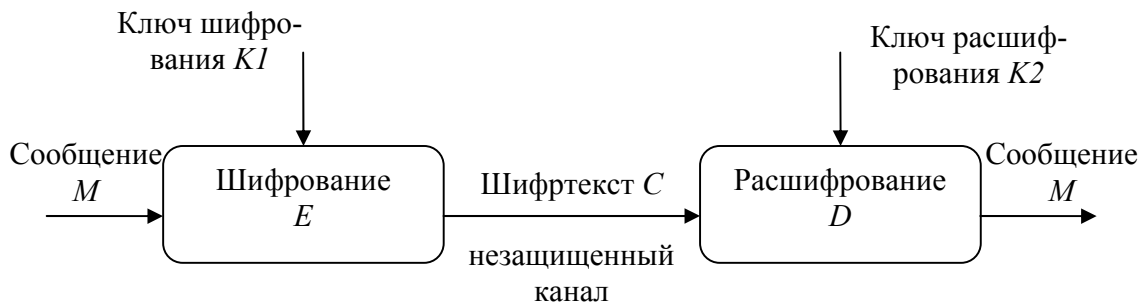


Рис. 6.1. Обобщенная схема криптосистемы

Известны два основных типа алгоритмов, основанных на использовании ключей: *симметричные и асимметричные алгоритмы (symmetric and asymmetric encryption)*.

Симметричные алгоритмы, иногда называемые условными алгоритмами, это те, в которых ключ зашифрования может быть вычислен из ключа расшифрования, и наоборот. В большинстве симметричных алгоритмов ключи зашифрования и расшифрования одинаковы. Эти алгоритмы также называются алгоритмами с секретным или закрытым ключом. Защита, обеспечиваемая симметричными алгоритмами, определяется ключом; раскрытие ключа означает, что шифровать и расшифровывать сообщения может кто угодно. Симметричные алгоритмы подразделяются на две категории. Одни алгоритмы обрабатывают открытый текст побитово (иногда побайтово). Такие алгоритмы называются потоковыми алгоритмами или шифрами. Другие алгоритмы обрабатывают группы битов открытого текста. Эти группы называют блоками, а алгоритмы – блочными шифрами. В современных компьютерных алгоритмах типичный размер блока составляет 64 бита. Это достаточно большое значение, чтобы затруднить анализ, и в тоже время достаточно малое, чтобы быть удобным для работы.

Задача обеспечения конфиденциальности передачи электронных документов с помощью симметричной криптосистемы сводится к обеспечению конфиденциальности ключа шифрования. Обычно ключ шифрования представляет собой файл или массив данных и хранится на персональном ключевом носителе (дискета или смарт-карта). Симметричное шифрование идеально подходит для шифрования «для себя», например, с целью предотвратить несанкционированный доступ к ней в отсутствие владельца. Это может быть как архивное шифрование выбранных файлов, так и прозрачное (автоматическое) целых логических или физических дисков.

Симметричное шифрование неудобно тем, что перед началом обмена зашифрованными данными необходимо обменяться секретными ключами со всеми адресатами. Передача секретного ключа симметричной криптосистемы не может быть осуществлена по общедоступным каналам связи, секретный ключ надо передавать отправителю и получателю по защищенному каналу распространения ключей.

Существуют реализации алгоритмов симметричного шифрования для абонентского шифрования данных и отправки через Интернет. Использование одного ключа для всех абонентов подобной криптосети недопустимо по соображения безопасности. При компрометации ключа под угрозой безопасности будет находиться документооборот всех абонентов. В этом случае может быть использована *матрица ключей*.

Матрица ключей представляет собой таблицу, содержащую ключи парной связи абонентов. Каждый элемент таблицы K_{ij} предназначен для связи i и j абонентов и доступен только двум данным абонентам. Соответственно, для всех элементов матрицы ключей соблюдается равенство $K_{ij} = K_{ji}$. Такие сетевые наборы ключей должны распространяться только по защищенным каналам связи.

Асимметричные алгоритмы или алгоритмы с открытым ключом устроены таким образом, что ключ, используемый для зашифрования, отличается от ключа расшифрования. Более того, ключ расшифрования не может вычислен, по крайней мере в течении разумного периода времени, из ключа расшифрования. Ключ зашифрования, называемый *открытым ключом* (*public key*), может быть открытым, поскольку не дает возможности расшифровывать шифртекст, тогда как ключ расшифрования, называемый *закрытым ключом* (*private key*), не должен быть свободно доступным.

В некоторых случаях сообщения следует зашифровывать закрытым ключом, а расшифровывать – открытым. Такой метод используют в цифровых подписях.

2.2. Введение в библиотеку LockBox2

Программный продукт LockBox2 фирмы TurboPower Professional представляет собой библиотеку функций и классов с исходными текстами, полностью написанными на языке Object Pascal. Библиотека LockBox2 предоставляет программисту развитую сервисы, которые позволяет встраивать в прикладные программы инструментарий современной криптографии. Особенностью LockBox2 является то, что эта функциональность может быть использована без каких-либо серьезных начальных знаний криптографических алгоритмов или их реализации, подобно тому как можно с успехом использовать графическую библиотеку без знания реализации аппаратных средств графической подсистемы.

Имеется несколько способов использования библиотеки LockBox2. Для опытных разработчиков, которые имеют точные требования при реализации программного обеспечения и понимают фундаментальные основы криптоалгоритмов, библиотека LockBox2 предоставляет в распоряжения широкий спектр низкоуровневых функций (*low-level routines*). Для быстрого и упрощенного использования криптоалгоритмов в состав LockBox2 входят несколько шифровальных компонентов (*encryption components*) и компонентов хэширования (*hashing components*). Разработчики, которым необходимо использовать алго-

ритмы и протоколы цифровых подписей для подписывания документов и файлов, могут воспользоваться функциональностью компонентов цифровых подписей (*digital signature components*).

В последней версии LockBox2 наряду с алгоритмами симметричного шифрования (DES, 3DES, Blowfish, AES) реализованы алгоритмы шифрования с открытым ключом (RSA, DSA), в том числе функции и классы для генерирования пар связанных ключей для алгоритма RSA, а также алгоритмы, реализующие модульную арифметику и арифметику сверхбольших чисел.

В дополнение ко всему прочему LockBox2 поддерживает алгоритмы 32- и 64-битного потокового шифрования (RNG32 и RNG64) для файлов и потоков байтов в памяти.

Библиотека полностью реализована на языке Object Pascal и может быть использована при разработки приложений в средах Borland Delphi или Borland C++Builder фактически любых версий. Библиотека содержит около 15-ти модулей (в зависимости от версии библиотеки), к основным из которых относятся следующие (табл. 6.1):

Таблица 6.1. Описание основных модулей библиотеки LockBox2

Имя модуля	Назначение
LbProc	Высокоуровневые функции шифрования файлов и файловых потоков с использованием алгоритмов симметричной криптографии (BlowFish, DES, TripleDES, Rijndael, оригинальные алгоритмы от TurboPower).
LbCipher	Наиболее низкоуровневые подпрограммы, реализующие базовые алгоритмы симметричной криптографии, а также алгоритмы однонаправленных функций шифрования. Модуль содержит большое количество вспомогательной функциональности: описания базовых структур данных, подпрограммы генерации ключей для симметричных алгоритмов, функции хэширования строк и др. Данный модуль составляет ядро библиотеки в части реализации шифрования с закрытым ключом.
LbClass	В модуле реализованы классы-обертки для различных алгоритмов симметричного блочного шифрования, хэширования и поточных шифров.
LbString	Функции шифрования строк на основе симметричной крипто-

	графии, а также функции кодирования строк в формат Base64.
LbUtils	Функции преобразования содержимого буферов памяти в шестнадцатеричный строчный формат и обратно, а также функции сравнения буферов.
LbRSA	Низкоуровневые функции и процедуры, реализующие алгоритм RSA, алгоритм цифровой подписи на основе RSA, а также функции генерации ключей для RSA. В модуле реализованы высокоуровневые классы-обертки для алгоритма RSA и алгоритма цифровой подписи на базе RSA.
LbAsym	Абстрактные базовые классы для классов асимметричных шифров и ключей асимметричных алгоритмов (TLbSignature, TLbAsymmetricKey, TLbAsymmetricCipher).
LbBigInt	Низкоуровневые подпрограммы для работы со сверхбольшими числами, которые используются в алгоритмах RSA и DSA, а также класс-обертка (TLbBigInt).
LbDSA	Классы и компоненты, реализующие функциональность алгоритма цифровой подписи DSA.

Для использования в проектах функциональных возможностей библиотеки LockBox2 идентификаторы данных модулей должны быть перечислены в предложении USES секции интерфейса или секции реализации.

2.3. Методы генерации ключей в LockBox2

Все распространенные современные блочные шифры, как симметричные, так и асимметричные предполагают наличие криптографического ключа или пары ключей. Библиотека LockBox2 предоставляет в распоряжение разработчика функции генерации, в том числе связанных ключей. Для генерации простых ключей используются процедуры GenerateLMDKey, GenerateMD5Key, GenerateRandomKey.

Наиболее простой функцией генерации ключа для симметричного шифра является процедура GenerateRandomKey. Декларация этой процедуры в модуле LbCipher имеет следующий вид:

```
procedure GenerateRandomKey (var Key; KeySize : Integer);
```

Эта процедура создает ключ, используя стандартный генератор случайных чисел, который реализован в модуле System RTL библиотеки Delphi. Процедура генерирует ключ размещая, его по адресу буфера Key, размером KeySize байт. Реализация данной подпрограммы приведена ниже:

```
procedure GenerateRandomKey(var Key; KeySize : Integer);
var
  I: Integer;
begin
  Randomize;
  for I := 0 to KeySize - 1 do
    TByteArray(Key) [I] := System.Random(256);
end;
```

Деятельность этой процедуры заключается в том, что она последовательно, в цикле, заполняет массив байтов заданным количеством случайных чисел и возвращает указатель на него.

В следующем примере генерируется 8-байтный ключ, который преобразуется к строчному шестнадцатеричному виду:

```
var
  Key: array[0...7] of Byte;
begin
  GenerateRandomKey(Key, SizeOf(Key));
  // показываем строчное шестнадцатеричное представление ключа
  ShowMessage(BufferToHex (Key, SizeOf(Key)));
end;
```

При построении приложения часто необходимо сгенерировать ключ на основе введенного пользователем пароля (ключевого слова). Наиболее предпочтительным вариантом в этом случае является использование процедуры GenerateMD5Key.

```
procedure GenerateMD5Key (var Key: TKey128; const Str : string);
TKey128 = array [0..15] of Byte;
```

Процедура `GenerateMD5Key` формирует 128-битный ключ с применением алгоритма хеширования MD5 на основе открытого текста теоретически любой произвольной длины.

Функция хеширования (хеш-функция) представляет собой преобразование, на вход которого подается сообщение переменной длины M , а выходом является строка фиксированной длины $h(M)$ – так называемый дайджест сообщения M , то есть сжатое двоичное представление основного сообщения M произвольной длины.

Наконец, процедура `GenerateLMDKey` генерирует ключ, используя алгоритм LockBox Message Digest (LMD). В отличие от дайджеста MD5 дайджест LMD может иметь переменную длину и соответственно позволяет генерировать ключи различного размера. Прототип процедуры имеет следующий вид:

```
procedure GenerateLMDKey (var Key; KeySize : Integer;
                        const Str : string);
```

Следующий пример демонстрирует генерацию 256-битного ключа на основе открытого текста:

```
var
    key: array[0...31] of Byte;
    MyPassword: string;
begin
    MyPassword := 'backbreaking job';
    GenerateLMDKey(key, SizeOf(key), MyPassWord);
    ShowMessage(BufferToHex(key, SizeOf(key)));
end;
```

Необходимо заметить, что используемая в процедуре `GenerateLMDKey` функция хэширования LMD не обладает хорошей криптостойкостью, поэтому применять в программах, критичных к безопасности, этот алгоритм и его производные следует с осторожностью.

Все три рассмотренные процедуры определены в модуле `LbCipher` библиотеки `LockBox2`.

2.4. Программные средства симметричного

шифрования в LockBox2

В библиотеке реализованы несколько наиболее распространенных алгоритмов симметричного шифрования: алгоритм Data Encryption Standard (DES), тройной DES и наиболее разитый на сегодняшний день симметричный шифр Rijndael, который образует основу нового стандарта шифрования США – Advanced Encryption Standard (AES).

2.4.1. Процедуры и функции поддержки DES-шифрования

Наиболее простым для понимания является алгоритм DES. Шифр DES является первым в мире алгоритмом, который был принят в качестве открытого стандарта шифрования. Данный алгоритм был предложен корпорацией IBM Национальному бюро стандартов США как вариант государственного стандарта шифрования. Шифр представлял собой алгоритм, созданный на базе шифра LUCIFER. В 1976 этот алгоритм был принят в качестве основы и после анализа Агентством Национальной безопасности принят как стандарт.

Различные виды криптоанализа (линейного и дифференциального) показывают, что внутренняя структура алгоритма может обеспечить значительную криптостойкость. Однако исходная реализация алгоритма предполагает использование 64-битного ключа, из которого реально используются только 56 бит. Поэтому такой сравнительно короткий ключ не обеспечивает защиту от «лобовой» атаки путем прямого перебора, что обусловлено стремительным развитием современных вычислительных средств.

Проблема короткого ключа может быть нейтрализована использованием другого варианта алгоритма DES – тройного DES (triple DES).

Библиотека LockBox2 предоставляет в распоряжение разработчика как набор низкоуровневых функций, ориентированных на шифрование с помощью DES отдельного блока, строк произвольной длины, файловых потоков и файлов, так и специализированные классы поддержки шифра DES.

Шифрование и расшифрование блока шифра DES в режиме электронной кодовой книги (блок открытого текста соответствует блоку шифртекста) выполняется с помощью процедуры EncryptDES:

```
procedure EncryptDES(const Context : TDESContext;  
                    var Block : TDESBlock);
```


где формальный параметр `Block` представляет собой массив из восьми байтов, `Context` – так называемый DES-контекст – является упакованной записью, содержащей представления трансформированных ключей раундов алгоритма. Перед шифрованием первого блока DES-контекст должен быть инициализирован вызовом специальной процедуры `InitEncryptDES`, которая выполняет действия по формированию ключей раундов на основе значения исходного ключа:

```
procedure InitEncryptDES(const Key : TKey64;
                        var Context : TDESContext; Encrypt : Boolean);
```

До вызова этой процедуры должен быть сформирован одной из функций генерации ключей 64-битный ключ `Key`, передаваемый в качестве первого параметра. Второй параметр представляет собой ссылку на контекст, подлежащий инициализации, а третий аргумент `Encrypt` указывает на то, что контекст будет использоваться для зашифрования (`True`) или расшифрования (`False`), поскольку от этого зависит порядок расположения ключей раундов.

Рассмотрим пример, в котором наглядно показана последовательность действий и необходимые переменные при выполнении шифрования блока с помощью низкоуровневых DES-процедур библиотеки `LockBox2`:

```
var
    DESBlock : TDESBlock;           // array[0..7] of Byte
    DESContext : TDESContext;
    key: TKey128;                   // array[0..15] of Byte
    DESKey: TKey64;                 // array[0..7] of Byte
    OpenText: Pointer;

begin
    GenerateMD5Key(key, 'Leo'); // генерируем ключ
    // копируем младшую половину 128-битного ключа в 64-битный ключ
    Move(Key, DESKey, sizeof(DESKey));
    // инициализация DES-контекста (готовит ключи раундов)
    InitEncryptDES(DESKey, DESContext, True);
    // готовим 8-ми байтный блок данных для шифрования
    OpenText :=PChar('LeoSoft!');
    // Копируем открытый текст в DES-блок
```

```

Move(OpenText, DESBlock, 8);
// выполняем зашифрование блока
EncryptDES(DESContext, DESBlock);
// показываем шестнадцатеричное представление
// блока после шифрования
ShowMessage(BufferToHex(DESBlock, SizeOf(DESBlock)));
end;

```

Рассмотренные функции блочного DES-шифрования реализованы в модуле LbCipher.

К примитивным, но весьма полезным низкоуровневым процедурам и функциям DES-шифрования относятся подпрограммы шифрования строк, которые реализованы в модуле LbString: DESEncryptString, DESEncryptStringEx, DESEncryptStringCBC, DESEncryptStringCBCEx. Первые две подпрограммы выполняют DES-шифрование в режиме электронной кодовой книги (Electronic Code Book – ECB), тогда как вторая пара подпрограмм использует режим сцепления блоков шифртекста (Chaining Block Cipher – CBC). Подпрограммы с префиксом «Ex» представляют собой функции и возвращают слева от себя значение зашифрованной строки. Прототипы двух первых подпрограмм имеют следующий вид:

```

procedure DESEncryptString(const InString : string;
                        var OutString : string;
                        const Key : TKey64; Encrypt : Boolean);

function DESEncryptStringEx(const InString : string;
                        const Key : TKey64;
                        Encrypt : Boolean) : string;

```

Обе подпрограммы получают в качестве параметра строку открытого текста InString и исходный ключ шифрования Key. В зависимости от параметра Encrypt входной текст будет зашифрован или расшифрован.

Необходимо отметить, что при зашифровании открытый текст сначала преобразуется к формату Base64 (чтобы содержал только печатные символы) и только затем шифруется. Наоборот, при расшифровании закрытый текст преобразуется из формата Base64, а затем расшифровывается. Для работы с форматом

Base64 в модуле LbString имеются две процедуры LbEncodeBase64 и LbDecodeBase64.

Рассмотрим пример использования функции DESEncryptStringEx:

```
var
    DESKey: TKey64;
begin
    GenerateRandomKey(DESKey, SizeOf(TKey64));
    // шифруем строку "Leo" и показываем результат шифрования
    ShowMessage(DESEncryptStringEx('Leo', DESKey, True));
end;
```

Другой группой подпрограмм DES-шифрования являются процедуры шифрования файлов. Это весьма удобные в использовании подпрограммы, которые позволяют зашифровывать и расшифровывать целые файлы, записывая результат операции в новый выходной файл. К таким подпрограммам относятся процедуры DESEncryptFile и DESEncryptFileCBC, которые реализованы в модуле LbProc библиотеки LockBox2:

```
procedure DESEncryptFile(const InFile, OutFile : string;
                        Key : TKey64; Encrypt : Boolean);
procedure DESEncryptFileCBC(const InFile, OutFile : string;
                            Key : TKey64; Encrypt : Boolean);
```

В качестве параметра InFile передается полный путь к исходному файлу, а OutFile – к результирующему файлу. Параметр Key определяет 64-битный исходный ключ DES алгоритма. Если параметр Encrypt принимает значение True, то содержимое файла InFile зашифровывается и записывается в OutFile, в противном случае файл InFile расшифровывается и результат записывается в OutFile.

2.4.2. Процедуры и функции поддержки AES-шифрования

Весьма полезным свойством библиотеки LockBox2 является значительная степень унифицированности интерфейсов всех подпрограмм симметричного

шифрования, что позволяет перейти в прикладных программах от использования одних алгоритмов шифрования к другим.

В последней версии библиотеки LockBox2 были реализованы подпрограммы шифрования на базе алгоритма Rijndael (читается как «Райндел»). Как уже отмечалось шифр Rijndael является наиболее развитым симметричным блочным шифром, криптостойкость которого многократно превышает криптостойкость любых других симметричных шифров. Алгоритм уже около десяти лет успешно противостоит многократным попыткам взлома с использованием методов дифференциального и линейного криптоанализа. Шифр утвержден министерством торговли США в качестве федерального стандарта шифрования 4 декабря 2001 года. Шифр разработан бельгийскими учеными Йоаном Даменом и Винсентом Райменом. В отличие от DES новый шифр не использует методологию сети Фейстеля для организации раудов шифрования. Вместо сети Фейстеля в шифре Rijndael реализована архитектура типа «квадрат». Размер блока в алгоритме является переменным и может составлять 128, 192 или 256 битов. Однако в качестве стандарта принят вариант шифра только с размером блока 128 бит. Размер ключа также может устанавливаться равным 128, 192 или 256 битов. Прямое и обратное преобразования в шифре имеют одинаковую алгоритмическую структуру.

Подпрограммы Rijndael-шифрования в библиотеки подразделяются на группы, подобно группам подпрограмм алгоритма DES: блочные, строковые, потоковые и файловые. Подпрограммы шифрования блока (EncryptRDL и EncryptRDLCBC) и подпрограммы инициализации контекста (InitEncryptRDL), а также остальные процедуры и функции Rijndael-шифрования имеют интерфейсы, схожие с интерфейсами подпрограмм DES-шифрования.

```
procedure EncryptRDL(const Context: TRDLContext;
                    var Block : TRDLBlock);
```

Перед шифрованием первого блока контекст шифра Rijndael должен быть инициализирован вызовом процедуры InitEncryptRDL, которая выполняет действия по инициализации так называемых векторов шифра Rijndael на основе значения исходного ключа:

```
procedure InitEncryptRDL(const Key; KeySize : Longint;
                        var Context: TRDLContext; Encrypt : Boolean);
```

Ниже приведен пример шифрования блока с использованием этих процедур:

```

var
    RDLBlock: TRDLBlock; RDLContext: TRDLContext;
    key: TKey128;
    OpenText: pointer;
begin
    GenerateMD5Key(key, 'Leo');
    InitEncryptRDL(key, SizeOf(key), RDLContext, True);
    // получим указатель на блок открытого текста
    OpenText := PChar('LeoSoft Solution'); // ровно 16 байт
    // копируем открытый текст в RDLBlock
    Move(OpenText, RDLBlock, 16);
    // шифруем
    EncryptRDL(RDLContext, RDLBlock);
    // покажем в шестнадцатичном виде
    ShowMessage(BufferToHex(RDLBlock, SizeOf(RDLBlock)));
end;

```

Принцип использования других групп процедур и функций Rijndael шифрования подобен использованию аналогичных подпрограмм для DES-шифрования.


3. Практическая часть

Исследуем особенности практики применения библиотеки LockBox2 при построении прикладных программ, использующих элементы криптографии для защиты информации.

3.1. Генерация ключей

Рассмотрим на примерах применение процедур генерации криптографических ключей. Выполним следующие этапы для построения проекта приложения в среде разработки Delphi.

1. Создадим новый проект стандартного Windows приложения, основу которого составляет главное окно. Используя инспектор объектов, заменим имя (свойство Name) формы по умолчанию на «MainForm», а также установим значение стиля рамки окна (свойство BorderStyle) равным bsDialog.

2. Расположим в центре клиентской области формы главного окна компонент типа «закладки» (класса TPageControl)  (рис. 6.1). Данный компонент доступен через закладку Win32 палитры компонентов Delphi.

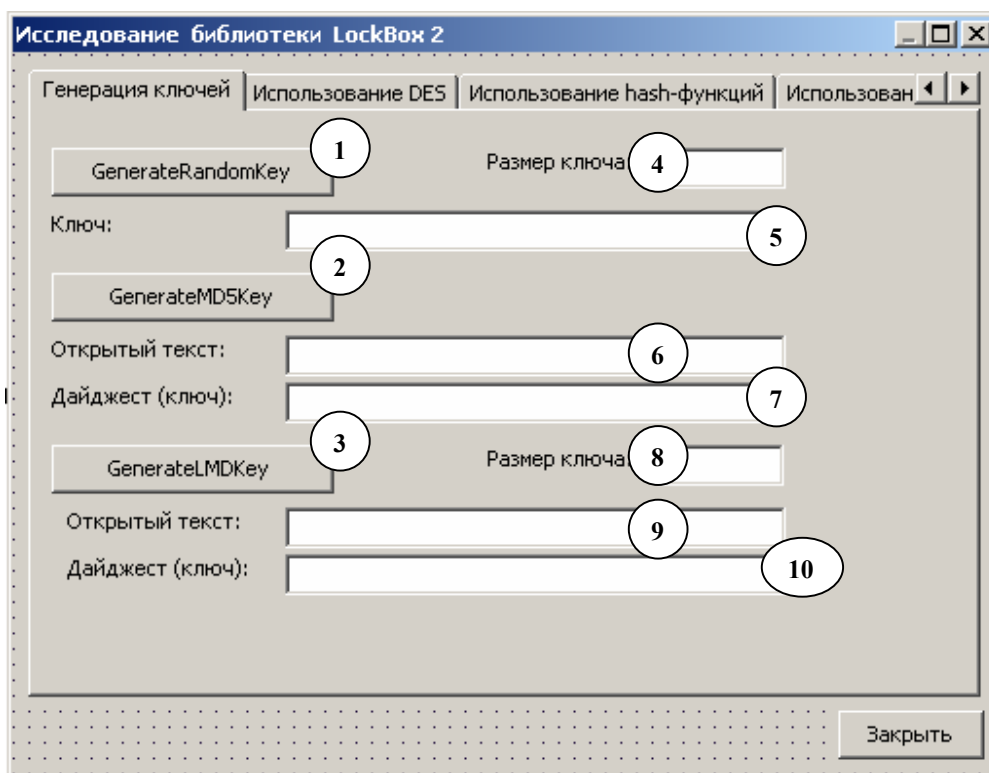




Рис. 6.1. Внешний вид окна приложения (первая закладка)


3. В дизайнера форм выберем команду New Page из контекстного меню, ассоциированного с компонентом типа PageControl, и добавим одну закладку (класс TTabSheet). Установим заголовок первой страницы (свойства Caption), как показано рис. 6.1.

4. Добавим командную кнопку типа Button  в область формы окна приложения, определив для нее идентификатор (имя) «AppCloseButton».

5. В область первой закладки («Генерация ключей») добавим следующие элементы управления:

- три простых командных кнопки типа Button (1-3) с именами «RandomKeyButton», «MD5KeyButton» и «LMDKeyButton» соответственно и заголовками, как показано на рис. 6.1;

- семь строк редактирования  (4-11) с идентификаторами (именами) «RandomKeySizeEdit» (4), «RandomKeyEdit» (5), «OpenTextMD5Edit» (6), «MD5KeyEdit» (7), «LMDKeySizeEdit» (8), «OpenTextLMDEdit» (9) и «LMDKeyEdit» (10);

- подписи к элементам управления типа Label , в соответствии с тем, как показано на рис. 6.1.

6. Добавим необходимый код в виде обработчиков событий OnClick (при нажатии) для трех командных кнопок, расположенных ранее на закладке «Генерация ключей»

6.1. Для кнопки RandomKeyButton код обработчика нажатия будет выглядеть следующим образом:

```
procedure TMainForm.RandomKeyButtonClick(Sender: TObject);
var
    PKey: PByte;           // указатель на байтовую ячейку
    KeySize: Integer;     // размер ключа
begin
    // выясним размер ключа, указанный пользователем
    KeySize := StrToInt(RandomKeySizeEdit.Text);
    // распределим память под массив байтов
    GetMem(PKey, SizeOf(Byte)* KeySize);
try
    // ВНИМАНИЕ! не забудем разыменовать указатель на массив
    // ключа PKey иначе будем работать с указателем указателя
    // и получим ACCESS VIOLATION
    GenerateRandomKey(PKey^, KeySize);
    // показываем в шестнадцатичном виде
    RandomKeyEdit.Text := BufferToHex (PKey^,KeySize);
finally
    // сборка "мусора" (garbage collection)
    // размер не указываем, т.к. менеджер памяти Delphi
    // сам опеределляет количество памяти под указателем
```

```

        FreeMem(PKey);
    end
end;

```

Этот код позволяет сформировать криптографический ключ на основе результатов работы генератора случайных чисел длиной, которая указана пользователем в строке редактирования RandomKeySizeEdit.

6.2. Для кнопки LMDKeyButton код обработчика OnClick представлен ниже:

```

procedure TMainForm.LMDKeyButtonClick(Sender: TObject);
var
    PKey: PByte;
    KeySize: Integer;
begin
    KeySize := StrToInt(LMDKeySizeEdit.Text);
    GetMem(PKey, SizeOf(Byte) * KeySize);
    try
        GenerateLMDKey(PKey^, KeySize, OpenTextLMDEdit.Text);
        LMDKeyEdit.Text := BufferToHex (PKey^,KeySize);
    finally
        FreeMem(PKey);
    end
end;

```

В этом фрагменте кода показано, как с помощью процедуры GenerateLMDKey формируется ключ заданного пользователем размера на основе дайджеста LMD.

6.3. В отличие от двух других процедур генерации ключей процедура GenerateMD5Key возвращает ключ фиксированной длины (128 бит). Однако ключ, сформированный этой процедурой, отличается высоким качеством, т.е. не представляется возможным восстановления ключевой фразы на основе сформированного ключа.

```

procedure TMainForm.MD5KeyButtonClick(Sender: TObject);
var

```



```

    key: TKey128;
begin
    GenerateMD5Key(key, OpenTextMD5Edit.Text );
    MD5KeyEdit.Text:= BufferToHex(key, SizeOf(key));
end;

```

7. Назначим в качестве обработчика `OnClick` кнопки «Заккрыть» следующий код:

```

procedure TMainForm.AppCloseButtonClick(Sender: TObject);
begin
    Application.Terminate;
end;

```

8. Для сборки данного приложения необходимо выполнение следующих условий:

- расположение исходных файлов или скомпилированных модулей библиотеки `LockBox2` должно быть предварительно указано в настройках среды разработки Delphi. Это может быть выполнено с помощью команды `Environment Options | Tools` в закладке `Library` (опция `Library Path`).

- в предложении `USES` секции интерфейса модуля главного окна необходимо перечислить все модули библиотеки `LockBox2`, процедуры и функции которых были использованы в проекте. Поскольку предполагается в дальнейшем использовать различные модули `LockBox2`, то можно перечислить сразу следующие идентификаторы модулей: `LbCipher`, `LbProc`, `LBString`, `LbRSA`, `LbClass`, `LbUtils`, `LBDSA`, `LBAsym` и `LBBigInt` или же указывать их по мере необходимости.

Если эти условия выполнены, то может быть выполнена сборка проекта и его тестирование. Необходимо отметить, что перед запуском обработчиков генерации ключей на основе случайных чисел или дайжеста LMD предварительно, во избежание ошибки преобразования в функции `StrToInt`, должен быть указан размер ключа в соответствующих строках редактирования.

3.2. Использование шифра AES

На примере зашифрования файлов с помощью криптоалгоритма AES разберем особенности применения шифровальных процедур библиотеки LockBox2. Выполним следующие шаги для развития проекта:

1. Используя дизайнер форм, присоединим к компоненту типа PageControl новую страницу (закладку) с заголовком (свойство Caption) «Использование AES».

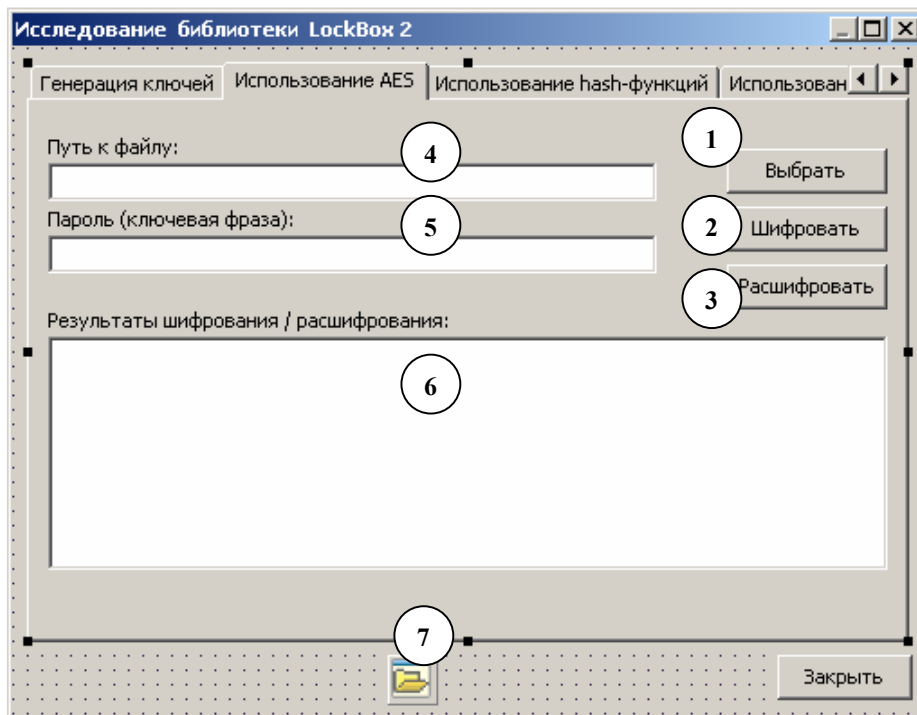




Рис. 6.2. Внешний вид окна приложения (вторая закладка)

2. В клиентскую область второй вкладки поместим следующие компоненты и установим им необходимые свойства:

- три командных кнопки (1-3) с идентификаторами `SelectFileButton`, `EncryptAESButton` и `DecryptAESButton` в соответствии с указанными номерами на рис. 6.2, а также назначим этим кнопкам соответствующие заголовки;

- две строки редактирования типа `Edit`  (4-5) с идентификаторами `EncryptFileNameEdit` и `PassEdit`;

- многострочный редактор (6) типа `Мемо`  с именем `OutputMemo`;

- присоединим к форме невизуальный компонент `OpenDialog` (7)  из закладки `Dialogs` палитры компонентов, которому установим имя «`OpenDialog`»;

- подпиши к элементам управления, как показано на рис. 6.2.

3. Реализуем логику работы этой части программы, назначив кнопкам из второй закладки следующие обработчики события `OnClick`:

3.1. Кнопка `SelectFileButton` позволяет задать полный путь к файлу, который должен быть впоследствии зашифрован. Путь файла, выбранного с помощью стандартного диалога `Windows`, при этом размещается в строке редактирования `EncryptFileNameEdit`. Компонент `OpenDialog` настроен на открытие только текстовых файлов с расширениями `TXT`, `C`, `PAS`, `ASM`, `CPP`. Текст данного обработчика приведен ниже:

```

procedure TMainForm.SelectFileButtonClick(Sender: TObject);
begin
    // позволяем открывать только текстовые файлы
    OpenDialog.Filter :=
        'Текстовые файлы|*.txt;*.pas;*.asm;*.c;*.cpp';
    // открываем стандартный диалог ОС для открытия файлов
    if OpenDialog.Execute = True then
        begin
            // если файл был выбран, то запоминаем имя исходного файла
            EncryptFileNameEdit.Text := OpenDialog.FileName;
            FileName:= EncryptFileNameEdit.Text;
            // формируем имя зашифрованного и расшифрованного файла
            EncFile := FileName + '.encrypt';
            DecFile := FileName + '.decrypt';
        end;
    end;

```

В исходном тексте данного обработчика упоминаются три строчных переменных `FileName`, `EncFile` и `DecFile`, которые предназначены для хранения пути к исходному файлу, файлу с результатами зашифрования и расшифрования соответственно. Эти переменные должны быть определены как члены класса главного окна `TMainForm`:

```

private
{ Private declarations }
FileName, EncFile, DecFile: string;
...

```

3.2. Нажатие кнопки EncryptAESButton приводит к шифрованию выбранного файла с помощью шифра AES. Содержимое зашифрованного файла после зашифрования помещается в многострочный редактор OutputMemo (рис. 6.3).

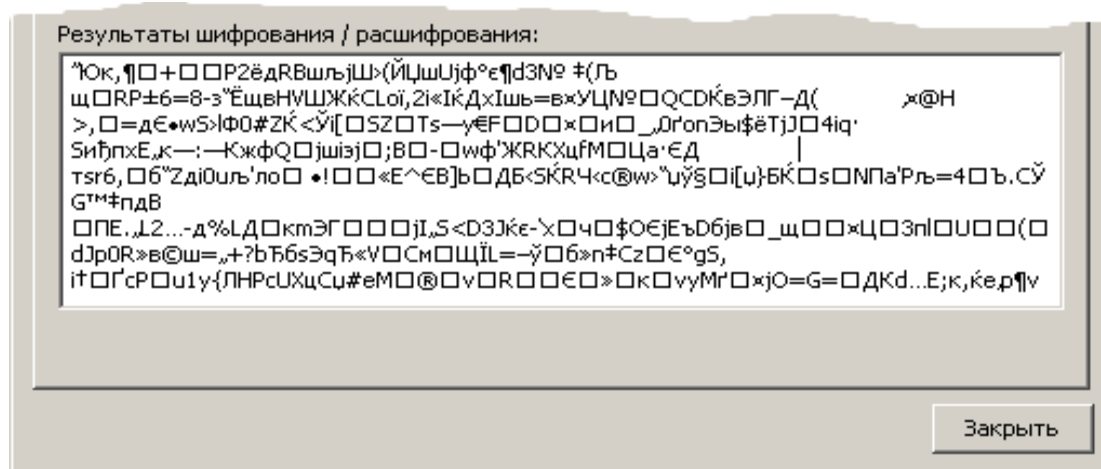


Рис. 6.3. Вывод результатов зашифрования файла

Для успешного шифрования в строке редактирования PassEdit предварительно должна быть указана ключевая фраза, на основе которой будет сформирован криптографический ключ для продпрограммы зашифрования. Исходный текст данной процедуры с подробными комментариями показан ниже:

```

procedure TMainForm.EncryptAESButtonClick(Sender: TObject);
var
    Key: TKey128; // здесь разместим ключ
begin
    // проверим наличие введенного пароля
    // и выбран ли файл для зашифрования
    if (PassEdit.Text <> '') and (FileName <> '') then
    begin
        // генерируем 128-битный ключ зашифрования

```

```

GenerateMD5Key(Key, PassEdit.Text);
// шифруем файл в режиме сцепления блоков шифртекста
RDLEncryptFileCBC(FileName, EncFile, Key,
                  SizeOf(TKey128), True);
// выводим на просмотр
OutputMemo.Lines.LoadFromFile(EncFile);
end
else
    ShowMessage('Не определен пароль или файл для шифрования');
end;

```

3.3. Подпрограмма обработки события `OnClick`, ассоциированная с кнопкой `DecryptAESButton`, выполняет восстановление (расшифрование) ранее зашифрованного файла, используя ту же ключевую фразу (соответственно ключ), что и примененную ранее для зашифрования. Восстановленное содержимое расшифрованного файла помещается также в многострочный редактор `OutputMemo` (рис. 6.4).

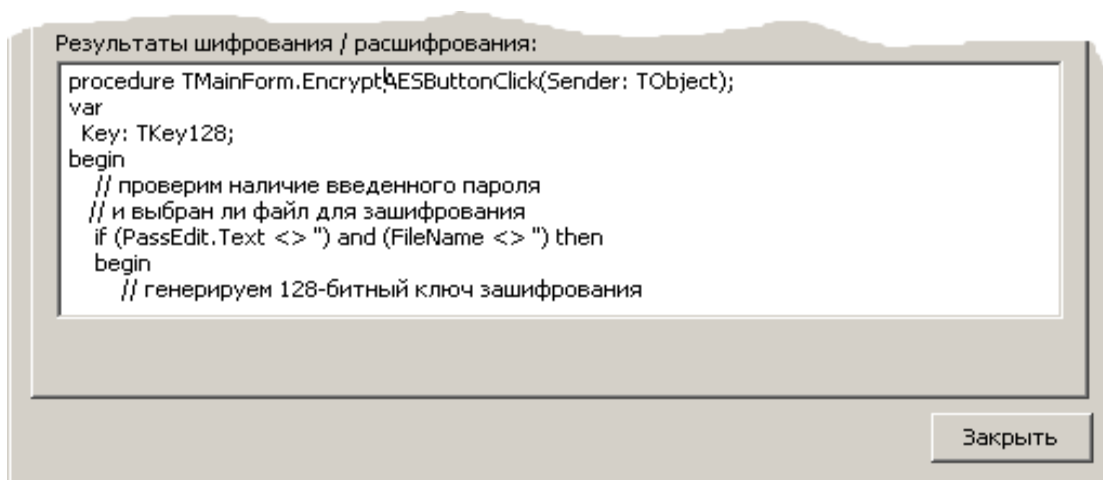


Рис. 6.4. Вывод результатов расшифрования ранее зашифрованного файла

Перед началом расшифрования в подпрограмме выполняется проверка существования на диске соответствующего зашифрованного файла с помощью функции `FileExists`.

```

procedure TMainForm.DecryptAESButtonClick(Sender: TObject);

```

```

var
  Key: TKey128;
begin
  // существует ли зашифрованный файл
  if FileExists(EncFile) then
    begin
      // вновь генерируем ключ
      // если ключевая фраза не изменилась
      // ключи будут совпадать
      GenerateMD5Key(Key, PassEdit.Text);
      // расшифровываем
      RDLEncryptFileCBC(EncFile, DecFile, Key,
                        SizeOf(TKey128), False);
      // выводим на просмотр результат расшифрования
      OutputMemo.Lines.LoadFromFile(DecFile);
    end
  else
    ShowMessage('Не найден файл для расшифрования');
end;

```

4. Выполним сохранение проекта, поскольку проект будет использован в следующих работах практикума, а также его сборку и тестирование.

В связи с работой данного приложения необходимо отметить следующую особенность. Повторное шифрование уже выбранного и зашифрованного файла (повторное нажатие на кнопку «Шифровать») даже при неизменном значении ключевой фразы и, соответственно, ключа приводит к выводу в многострочном редакторе новых результатов зашифрования. Это обусловлено особенностью шифрования в режиме сцепления блоков шифртекста (СВС): первый блок для сцепления, так называемая синхропосылка, формируется псевдослучайным образом и определяет содержимое последующих блоков. Такая синхропосылка при каждом зашифровании формируется заново, что и обуславливает различие в результатах шифрования. Однако результаты расшифрования всегда совпадают при неизменном значении ключевой фразы.

3.3. Задания

1. Исследовать поведение программы при замене шифровальной процедуры `RDLEncryptFileCBC` на процедуру `RDLEncryptFile` (режим ECB).
2. Разработать фрагменты программы для шифрования файлов с помощью алгоритма тройного-DES, используя процедуры библиотеки `LockBox2`.

4. Контрольные вопросы

1. В чем состоит фундаментально важное отличие алгоритмов шифрования с закрытым и открытым ключом?
2. Какие еще профессиональные криптографические библиотеки и криптографические API, кроме `LockBox2`, вы знаете?

Лабораторная работа № 7

АСИММЕТРИЧНАЯ КРИПТОГРАФИЯ. ПРИМЕНЕНИЕ АЛГОРИТМОВ С ОТКРЫТЫМ КЛЮЧОМ ДЛЯ ЗАЩИТЫ ИНФОРМАЦИИ В ПРИКЛАДНЫХ ПРОГРАММАХ

1. Цель работы

Цель данной лабораторной работы состоит в том, чтобы рассмотреть на практике использование криптографических алгоритмов с открытым ключом. В работе рассматриваются примеры использования библиотеки `LockBox2`, в частности, подпрограмм формирования ключей асимметричного шифрования, а также подпрограмм асимметричного RSA-шифрования.

2. Основные теоретические сведения

2.1. Асимметричные криптосистемы

Принципиальное отличие асимметричной криптосистемы от симметричных алгоритмов состоит в том, что для шифрования информации и ее последующего расшифрования используются различные ключи:

- открытый ключ K используется для зашифрования информации и вычисляется из секретного ключа k ;

- секретный ключ k используется для расшифровки зашифрованной информации с помощью парного ему открытого ключа K .

Секретный и открытый ключ генерируются попарно. Секретный ключ должен оставаться у его владельца; его необходимо надежно защитить от несанкционированного доступа. Копия открытого ключа должна находиться у каждого абонента криптографической сети, с которым обменивается информацией владелец секретного ключа. Для организации двунаправленного защищенного информационного обмена между несколькими абонентами криптосети, каждый абонент должен переслать по открытому каналу связи всем другим свой открытый ключ и получить открытые ключи других абонентов криптосети (рис. 7.1)

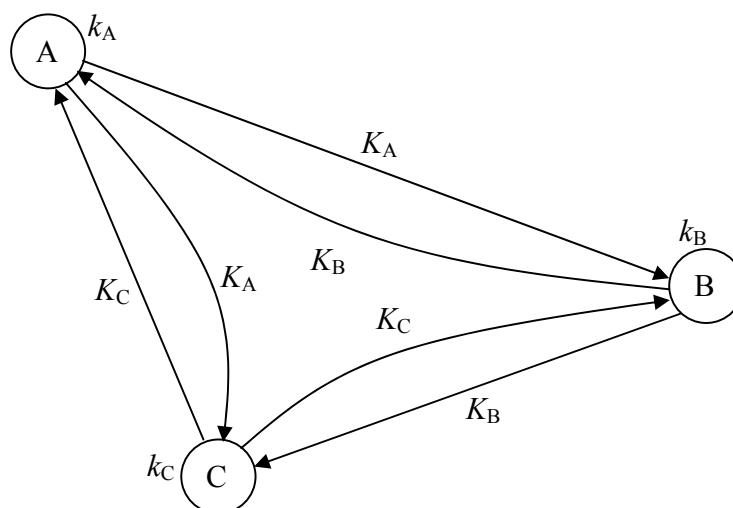


Рис. 7.1. Схема обмена открытыми ключами между абонентами криптосети при использовании асимметричной криптографии

Процесс передачи зашифрованной информации в асимметричной криптосистеме осуществляется следующим образом:

1. Абонент B генерирует пару ключей: секретный k_B и открытый K_B . Открытый ключ K_B отсылается абоненту A и остальным абонентам.

2. Абонент A зашифровывает с помощью ключа K_B и отправляет шифртекст абоненту B .

3. Абонент B расшифровывает сообщение ключом k_B . Никто, кроме B , не может расшифровать сообщение, даже абонент A , пославший это сообщение, так как не имеет секретного ключа.

Таким образом защита информации в асимметричной криптосистеме основана на секретности ключа k_B получателя сообщения.

Основные требования, выполнение которых обеспечивает безопасность асимметричной криптосистемы, можно сформулировать следующим образом:

- простое вычисление пары ключей;
- математически тривиальное получение криптограммы;
- элементарное восстановление исходного сообщения;
- знание злоумышленником открытого ключа K_B не дает ему возможность вычислить секретный ключ k_B , так как это представляет собой непреодолимую вычислительную задачу;
- знание злоумышленником открытого ключа K_B и шифртекста C также не позволяет вычислить исходное сообщение M в связи с непреодолимой вычислительной проблемой.

Концепция современных асимметричных криптосистем с открытым ключом основана на применении однонаправленных функций, которые обладают свойством: существует алгоритм вычисления $Y = F(X)$, но нет алгоритма вычисления обратной функции или он слишком вычислительно сложен, т.е. невозможно найти исходное X при известном Y и известном алгоритме функции F .

2.2. Реализация алгоритма асимметричного шифрования RSA

Из всех предложенных за последние несколько десятилетий алгоритмов с открытым ключом именно алгоритм RSA проще всего понять и реализовать, при этом алгоритм RSA является наиболее популярным и широко используемым. Свое название алгоритм получил в честь своих изобретателей Р. Райвеста, А. Шамира, Л. Адельмана. Этот алгоритм многие годы противостоит многочисленным попыткам криптоаналитического вскрытия. Он может применяться как для шифрования данных, так и для цифровой подписи документов.

Безопасность алгоритма RSA основана на трудоемкости разложения на множители (факторизации) больших чисел. Открытый и закрытый ключи являются функциями двух больших простых чисел (целое положительное число, большее единицы, не имеющее других делителей, кроме единицы и самого себя 2, 3, 5, 7, 11, 13, 17, 19...), разрядностью 100-200 десятичных цифр или даже больше. Предполагается, что восстановление открытого текста по шифртексту и открытому ключу равносильно разложению числа на два больших простых множителя.

Для генерации двух ключей применяется два больших случайных простых числа p и q . Для максимальной безопасности p и q должны иметь равную длину. Далее вычисляется произведение $n = pq$. Затем случайным образом выбирается ключ шифрования e , такой, что e и $(p-1)(q-1)$ являются взаимно простыми числами (то есть целые числа, не имеющие общих простых делителей). Наконец, с помощью расширенного алгоритма Евклида вычисляется ключ расшифрования d , такой что:

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)},$$

то есть:

$$d = e^{-1} \pmod{(p-1)(q-1)}$$

Заметим, что d и n также взаимно простые числа.

Таким образом, числа e и n составляют открытый ключ, а числа d и n – закрытый. Два простых числа p и q больше не нужны. Они могут быть отброшены, но не должны быть раскрыты.

При шифровании сообщение M сначала разбивается на цифровые блоки m_i , размерами меньше числа n . То есть если p и q имеют 100 разрядов, то n будет иметь 200 разрядов и каждый блок сообщения m_i должен иметь около, но не более, 200 разрядов в длину. Зашифрованное сообщение C также будет состоять из блоков c_i такой же длины. Формула зашифрования выглядит следующим образом:

$$c_i = m_i^e \pmod{n}.$$

При расшифровке сообщения для каждого зашифрованного блока c_i вычисляется

$$m_i = c_i^d \pmod{n}.$$

2.3. Средства поддержки асимметричного шифрования в LockBox2

Параметр `Callback` является указателем на функцию, вызов которой внутри процедуры позволяет досрочно завершить генерацию ключа. Рассмотрим пример использования этой процедуры:

```

var
    OpenRSAKey, CloseRSAKey: TLbRSAKey;
begin
    //формируем короткий ключ (128 бит) с малым числом итераций (5)
    GenerateRSAKeysEx(CloseRSAKey, OpenRSAKey, aks128,5, nil);
    // выводим компоненты ключей
    ShowMessage('Закрытый ключ: ' +
        #13#13 + CloseRSAKey.ModulusAsString + #13#13
        + CloseRSAKey.ExponentAsString);
    ShowMessage('Открытый ключ: ' +
        #13#13 + OpenRSAKey.ModulusAsString + #13#13
        + OpenRSAKey.ExponentAsString);
end;

```

В классе `TLbRSAKey` реализован метод `StoreToFile`, позволяющий сохранять содержимое криптографического ключа в файле:

```

CloseRSAKey.StoreToFile('G:\Leo.key');

```

Имеется также метод загрузки компонентов ранее сохраненного ключа из файла на диске – метод `LoadFromFile`:

```

var
    key: TLbRSAKey;
begin
    // создаем новый объект ключа
    key := TLbRSAKey.Create(aks128);
    // загружаем
    key.LoadFromFile('G:\Leo.key');
    // показываем компоненты ключа
    Edit1.Text := key.ModulusAsString ;
    Edit2.Text := key.ExponentAsString;

```

```
end;
```

После того как сгенерированы пары ключей, могут быть использованы подпрограммы для зашифрования и расшифрования с использованием алгоритма RSA. В библиотеке LockBox2 доступны несколько подпрограмм для RSA шифрования – это подпрограммы зашифрования и расшифрования блоков EncryptRSA и DecryptRSA, файлов и потоков – RSAEncryptFile и RSAEncryptStream, а также строк RSAEncryptString.

Простейшими подпрограммами RSA-шифрования являются функции блочного зашифрования и расшифрования. Для зашифрования блока открытого текста в LockBox2 используется функция EncryptRSA, которая в качестве параметров получает открытый ключ PublicKey, содержимое блока открытого текста InBlock, а также ссылку на буфер-приемник блока шифртекста OutBlock:

```
function EncryptRSA(PublicKey: TLbRSAKey;
                    const InBlock : TRSAPlainBlock;
                    var OutBlock : TRSACipherBlock) : Longint;
```

Блок открытого текста представляется массивом из 53 байтов (424 бита), а блок шифртекста – массивом из 64 байтов (512 бит):

```
TRSAPlainBlock = array[0..52] of Byte;
TRSACipherBlock = array[0..63] of Byte;
```

В отличие от рассмотренных ранее подпрограмм симметричного шифрования, процедуры которых объединяют в себе операцию зашифрования и расшифрования, расшифрование для шифра RSA реализовано в виде отдельной функции DecryptRSA. Интерфейс этой функции отличается от интерфейса EncryptRSA, тем, что в качестве первого параметра передается закрытый ключ и меняется порядок передачи шифровальных блоков: сначала блок шифртекста и затем блок-приемник результатов расшифрования открытого текста:

```
function DecryptRSA(PrivateKey: TLbRSAKey; const InBlock:
                    TRSACipherBlock; var OutBlock : TRSAPlainBlock) : Longint;
```

Рассмотрим пример применения блочных функций RSA-шифрования. В следующем примере зашифровывается строчный буфер произвольного размера, который размещается в блоке открытого текста. Затем блок шифртекста расшифровывается.

```

procedure TMainForm.Button6Click(Sender: TObject);
var
    CloseRSAKey, OpenRSAKey: TLbRSAKey;
    Text: string;
    OpenText: TByteArray;
    PlainBlock: TRSAPlainBlock;
    CipherBlock: TRSACipherBlock ;
    i: Integer;
begin
    // открытый шифруемый текст
    Text := 'Преимуществом RSA является высокая криптостойкость';
    // перемещаем побайтно в буфер шифровальной функции
    // длина строки не должна превышать размер буфера PlainBlock
    for i := 0 to Length(Text)-1 do
        begin
            PlainBlock[i] := Byte(Text[i+1]);
        end;
    // генерируем ключи по умолчанию
    GenerateRSAKeys(CloseRSAKey, OpenRSAKey);
    // шифруем
    EncryptRSA(OpenRSAKey, PlainBlock, CipherBlock);
    // показываем зашифрованный блок
    ShowMessage(BufferToHex(CipherBlock, SizeOf(CipherBlock)));
    // очищаем блок открытого текста
    FillChar(PlainBlock, SizeOf(TRSAPlainBlock), 0);
    Text := '';
    // расшифровываем
    DecryptRSA(CloseRSAKey, CipherBlock, PlainBlock);
    // копируем расшифрованный блок в строку
    for i := 0 to SizeOf(PlainBlock)-1 do
        begin

```

```

    Text := Text + Char(PlainBlock[i]);
  end;
  // показываем
  ShowMessage(Text);
end;

```

Наиболее полезными подпрограммами RSA-шифрования, которые отличаются простотой и эффективностью использования, являются процедуры шифрования файлов и строк :

```

procedure RSAEncryptFile(const InFile, OutFile : string;
    Key: TLbRSAKey; Encrypt : Boolean);

function RSAEncryptString(const InString : string;
    Key : TLbRSAKey; Encrypt : Boolean) : string;

```

Обе подпрограммы являются двунаправленными. Параметр `Encrypt` указывает на направление шифрования: если `Encrypt` равен `True`, выполняется зашифрование, иначе расшифрование. В зависимости от направления шифрования в качестве параметра `Key` ключа должен быть указан либо открытый ключ (зашифрование), либо закрытый ключ (расшифрование).

Кроме описанных выше низкоуровневых функций и процедур RSA-шифрования в составе библиотеки `LockBox2` имеется класс `TLbRSA`, который предоставляет программисту объектно-ориентированную оболочку к этим подпрограммам. Прародителем класса `TLbRSA` является класс `TComponent`. Поэтому объекты класса `TLbRSA` в терминологии Delphi являются компонентами и могут размещаться на форме на этапе разработки интерфейса программы. Использование в программе объектов класса `TLbRSA` существенно упрощает использование библиотеки для разработчиков, которые имеют только общие сведения о криптографических алгоритмах и протоколах.

3. Практическая часть

Разберем законченный пример использования средств асимметричной криптографии и однонаправленных функций шифрования из библиотеки


LockBox2. За основу возьмем проект приложения, разработанный к предыдущей лабораторной работе.

3.1. Использование асимметричного шифра RSA

1. Откроем ранее сохраненный проект (Open Project | File). Для реализации функциональности новой части приложения будем использовать уже существующие основные компоненты приложения.

2. Присоединим к компоненту PageControl еще одну закладку типа TabSheet (команда New Page контекстного меню PageControl). В инспекторе объектов зададим свойству Caption значение «Использование RSA» (рис. 7.1).

3. В область созданной закладки необходимо добавить несколько элементов управления и правильно задать их идентификаторы и установить некоторые свойства:

- четыре командные кнопки типа Button  (1-4) с идентификаторами (свойство Name) RSAKeyButton (1), SelectFileRSAButton (2), RSAEncryptButton (3), RSADecryptButton (4), которые имеют соответствующие заголовки, как это показано на рис. 7.2;

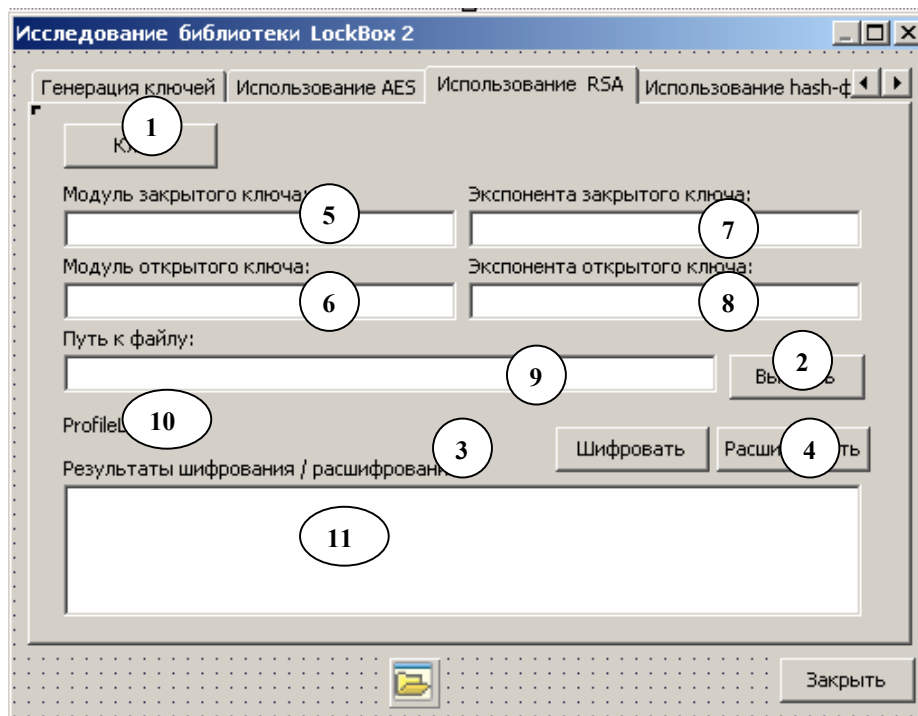




Рис. 7.2. Внешний вид окна приложения (третья вкладка)

- пять строк редактирования (5-9) типа Edit  с идентификаторами RSAPrKeyModEdit, RSAPuKeyModEdit, RSAPrKeyExpEdit, RSAPuKeyExpEdit и EncryptRSAFileNameEdit. Строки редактирования имеют ассоциированные необязательные подписи, указывающие на их предназначение в программе.

- многострочный редактор типа Мемо , который предназначен для вывода результатов зашифрования и расшифрования. Этому компоненту необходимо назначить идентификатор OutputRSAMemo.

- подпись типа Label **A** (10) с идентификатором ProfileLabel, которая предназначена для вывода сообщений о времени формирования ключей и продолжительности операций шифрования.

4. Каждой из добавленных на новую закладку кнопок назначим обработчик события OnClick. Код этих подпрограмм составит функциональную основу этой части приложения.

4.1. Кнопка RSAKeyButton предназначена для запуска процедуры генерации ключей:

```

procedure TMainForm.RSAKeyButtonClick(Sender: TObject);
var
    // два локальных ключа типа TLbRSAKey
    RSAPrivateKey, RSAPublicKey: TLbRSAKey;
begin
    // примитивное профилирование
    StartTick := GetTickCount;    // старт
    // генерируем ключи с размерами по умолчанию (512 бит)
    GenerateRSAKeys(RSAPrivateKey, RSAPublicKey);
    StopTick := GetTickCount;    // стоп
    // показываем результаты профилирования
    ProfileLabel.Caption :=
        Format ('Время формирования ключей: %f с',
            [(StopTick-StartTick) / 1000] );
    // показываем в строках редактирования
    // компоненты закрытого и открытого ключей
    RSAPrKeyModEdit.Text := RSAPrivateKey.ModulusAsString;
    RSAPrKeyExpEdit.Text := RSAPrivateKey.ExponentAsString;
    RSAPuKeyModEdit.Text := RSAPublicKey.ModulusAsString;

```

```

RSAPuKeyExpEdit.Text := RSAPublicKey.ExponentAsString;
// компоненты ключей могут быть сохранены в формате ASN.1
RSAPrivateKey.StoreToFile('Leo.privatekey');
RSAPublicKey.StoreToFile('Leo.openkey');
end;

```

Пара криптографических ключей алгоритма RSA генерируются с параметрами по умолчанию: 512-битный ключ с числом раундов проверки на простоту исходных чисел p и q , равным 20. Поскольку генерация ключей отнимает значительное время и является величиной непостоянной в силу особенностей вышеупомянутой проверки, то значение этого интервала представляет определенный интерес. Для выяснения времени генерации ключа можно воспользоваться функцией `GetTickCount` Windows API, которая возвращает число миллисекунд, прошедших с момента запуска ОС. Для такого примитивного профилирования понадобятся две переменные `StartTick` и `StopTick`, которые необходимо определить как члены класса `TMainForm`:

```

private
    { Private declarations }
    . . .
    StartTick, StopTick: Cardinal;

```

Разность тиков, полученных до начала и после завершения генерации ключей, составляет искомый временной интервал, значение которого отображается на экране. Компоненты (модуль и экспонента) сформированных ключей выводятся в строки редактирования и записываются в файл для возможности их последующего использования.

4.2. Кнопка `SelectFileRSAButton` предназначена для выбора полного пути к исходному файлу для зашифрования:

```

procedure TMainForm.SelectFileRSAButtonClick(Sender: TObject);
begin
    OpenFileDialog.Filter :=
        'Текстовые файлы|*.txt;*.pas;*.asm;*.c;*.cpp';
    if OpenFileDialog.Execute() then
        begin

```

```

// если файл был выбран то
// запоминаем имя исходного файла
EncryptRSAFileNameEdit.Text := OpenFileDialog.FileName;
FileName:= EncryptRSAFileNameEdit.Text;
// формируем имя зашифрованного и расшифрованного файла
EncFile := FileName + '.encrypt';
DecFile := FileName + '.decrypt';
    end;
end;

```

4.3. Двум оставшимся кнопкам `RSASaveButton` и `RSADecryptButton` можно назначить одну общую для них процедуру обработки. Для выполнения операций зашифрования и расшифрования воспользуемся классом `TLbRSA`, который служит «оберткой» для рассмотренных ранее процедур RSA-шифрования.

Поскольку для зашифрования и расшифрования применяются две разных подпрограммы, то при использовании одно обработчика необходимо предусмотреть способ вызова разных методов класса `TLbRSA` с различными параметрами (входной и выходной файлы) и различными ключами в зависимости от того, какая кнопка инициировала событие, связанное с этим обработчиком.

Оптимальным способом решения такой задачи является использование указателей (адресов) на подпрограммы и методы объектов. Object Pascal так же как и C/C++ поддерживает работу с указателями различных программных сущностей, как информационных, так и операционных. Такие свойства языка позволяют вызывать подпрограммы, используя не их идентификаторы (имена процедур и функций), а их адреса в виртуальной памяти процесса. Ограничением при этом является то, что от имени одного указателя на подпрограммы можно вызывать только подпрограммы с одинаковым интерфейсом.

Рассматриваемая процедура обработки дает некоторые представления об указателях на подпрограммы в Object Pascal и показывает их практическое применение.

```

procedure TMainForm.RSASaveButtonClick(Sender: TObject);
var
    // объект криптомашины
    RSACryptEngine: TLbRSA;

```

```

// объектная переменная будет служить адресом ключей криптомашины
Key: TLbRSAKey;
// файл ключа, входной и выходной файлы криптомашины
KeyFileName, InFile, OutFile: string;
// указатель на нужный шифровальный метод криптомашины
Crypter: procedure(const InFile, OutFile : string) of object;
mess: string;
begin
// создаем криптомашину
RSACryptEngine := TLbRSA.Create(nil);
try
// выясняем, кто инициировал запуск обработчика
// если кнопка "Шифровать"
if (Sender as TButton).Caption = 'Шифровать' then
begin
// выбираем файл с открытым ключом
KeyFileName := 'Leo.openkey';
// исходным является выбранный файл для шифрования
// а выходным зашифрованный
InFile := FileName; OutFile := EncFile;
// получаем указатель на метод (зашифровальную процедуру)
Crypter := RSACryptEngine.EncryptFile;
// получаем указатель на объектное свойство объекта
Key := RSACryptEngine.PublicKey;
mess := 'зашифрования';
end
// если кнопка "Расшифровать"
else
begin
// ссылаемся на файл с закрытым ключом
KeyFileName := 'Leo.privatekey';
// теперь исходным является зашифрованный файл
// а выходным расшифрованный
InFile := EncFile; OutFile := DecFile;
// получаем указатель на метод (расшифровальную процедуру)
Crypter := RSACryptEngine.DecryptFile;

```

```

    // получаем указатель на объектное свойство объекта
    Key := RSACryptEngine.PrivateKey;
    mess := 'расшифрования';
end;
// проверим наличие входного файла и файла ключа
if not (FileExists(InFile) and FileExists(KeyFileName)) then
begin
    // если не нашли, то даем отбой
    ShowMessage('Входной или ключевой файл не найден');
    // ВНИМАНИЕ! «убивать» объект RSACryptEngine
    // здесь не нужно, поскольку все равно попадем в finally
    // иначе ACCESS VIOLATION -
    // "жизнь" объекта после "смерти" невозможна
    Exit;
end;
// адресная природа программных сущностей
// во всем своем великолепии
Key.LoadFromFile(KeyFileName);
// используем примитивное профилирование
StartTick := GetTickCount; // старт
Crypter(InFile, OutFile);
StopTick := GetTickCount; // стоп
// используем функцию форматирования строк
// для вывода информации о профилировании
ProfileLabel.Caption :=
    Format('Время %s: %f', [mess, (StopTick - StartTick)/1000]);
// выводим содержимое выходного файла в редактор
OutputRSAMemo.Lines.LoadFromFile(OutFile);
finally
    // разрушаем машину
    RSACryptEngine.Free;
end;
end;

```

Данный обработчик должен быть назначен обеим кнопкам. Это можно сделать, используя инспектор объектов. Сначала, выбрав кнопку RSAEncrypt-

Button, необходимо создать новый обработчик события OnClick и реализовать его, а затем, выбрав кнопку RSADecryptButton, назначить ей тот же самый обработчик, выбрав из списка доступных в инспекторе объектов процедуру RSAEncryptButtonClick.

В этой процедуре используется одно важное и весьма полезное свойство событийной модели программирования, которая поддерживается библиотекой классов VCL: любая процедура обработки события от оконного элемента управления имеет формальный параметр типа TObject с идентификатором Sender, через который в эту подпрограмму передается ссылка на объект (элемент управления), инициировавший обработку данного события. Получив в распоряжение эту ссылку подпрограмма обработки события может в период времени исполнения модифицировать свое поведение, определив кто именно послал событие. Выявить инициатора обработки можно легко, если использовать свойство полиморфности объектов и специальные операторы приведения объектного типа (оператор AS) и проверки объекта на принадлежность к определенному классу (оператор IS). Например, чтобы проверить, что объект является кнопкой, т.е. объектом класса TButton или класса потомка от TButton, нужно выполнить следующий код с использованием оператора IS:

```
if Sender Is TButton then
begin
    . . .
end;
```

Оператор IS возвращает истинное булевское значение, если операнд справа действительно является объектом класса, указанного в качестве операнда слева.

Оператор AS в свою очередь позволяет разрешить объектную ссылку в объект определенного класса, если такая ссылка действительно указывает на объект заданного класса или класса предка. Например:

```
(Sender As TButton).Caption := 'Моя кнопка';
```

Использование свойства полиморфности и оператора AS в рассматриваемом примере позволяет выявить инициатора вызова обработчика простейшим

способом: на основе анализа содержимого свойства Caption кнопки, т.е. ее заголовка.

5. Наконец, выполним сохранение проекта, поскольку проект будет использован в следующих работах практикума, и его сборку. При тестировании проекта необходимо учитывать особенность алгоритма RSA, шифрование в котором основано на выполнении операций возведения в степень сверхбольших чисел. Эта особенность состоит в том, что операции над сверхбольшими числами реализуются программным способом, а не прямыми командами процессора, что требует огромного количества времени. Время шифрования с помощью алгоритма RSA на три порядка больше времени шифрования с помощью симметричных шифров. При этом время зашифрования в RSA в десятки раз меньше времени расшифрования (рис. 7.3 и 7.4).

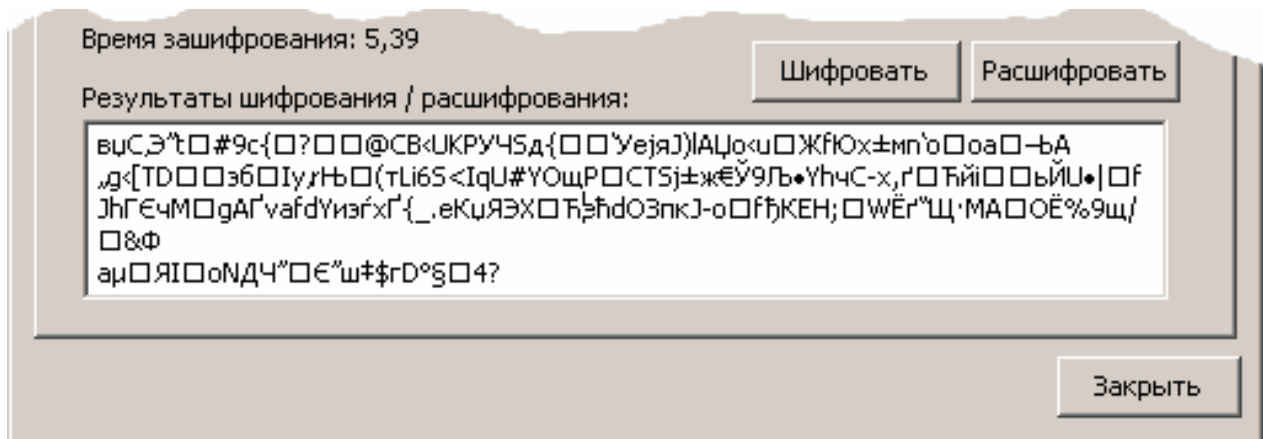


Рис. 7.3. Результаты выполнения RSA-зашифрования текста проекта

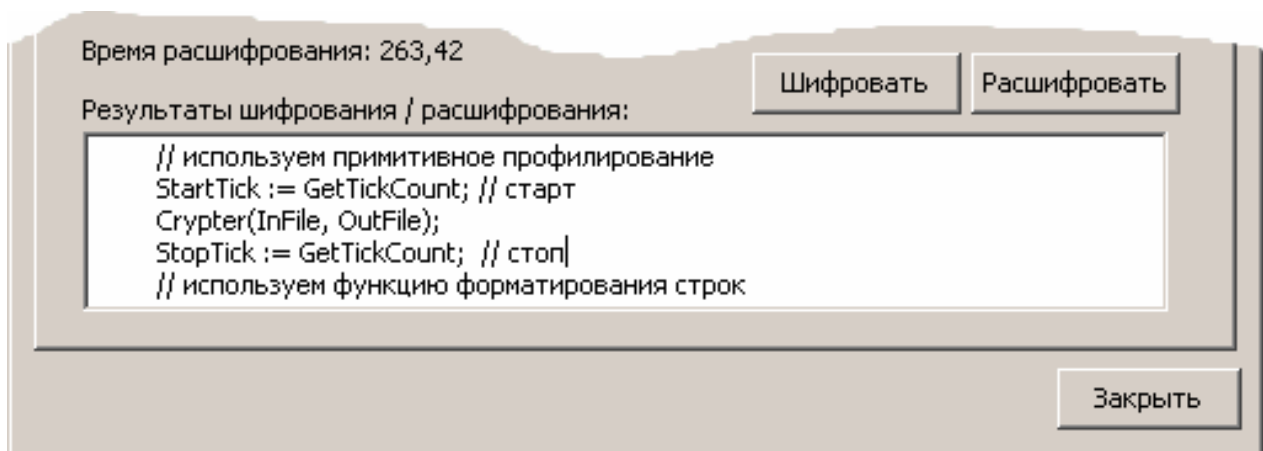


Рис. 7.4. Результаты выполнения RSA-расшифрования текста проекта

Тестирование данного приложения и результаты временного профилирования участков кода, в которых выполняются операции шифрования позволяют сделать следующие выводы: асимметричное шифрование имеет ограниченное применение при шифровании массивов данных большого объема.

3.2. Задания

1. Исследовать зависимость временных затрат в операциях шифрования алгоритма RSA от длины используемого ключа и длины шифруемого блока. Использовать для этого подпрограмму генерации ключей `GenerateRSAKeysEx` и подпрограммы шифрования блока заданной длины `EncryptRSAEx` и `DecryptRSAEx`. Прототипы процедур и их реализация находятся в модуле `LbRSA` библиотеки `LockBox2`.

4. Контрольные вопросы

1. Какие существуют схемы реализации криптосистем асимметричного шифрования?
2. На чем основана высокая криптостойкость шифра RSA?

Лабораторная работа № 8

АСИММЕТРИЧНАЯ КРИПТОГРАФИЯ. АЛГОРИТМЫ ЦИФРОВОЙ ПОДПИСИ И ИХ ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ

1. Цель работы

Целью этой работы является изучение основ методологии применения алгоритмов электронной цифровой подписи для аутентификации файлов и документов. В ходе данной лабораторной работы студент должен познакомиться с основными теоретическими положениями в области однонаправленных функций шифрования и электронной цифровой подписи. В практической части обучающийся должен приобрести навыки применения криптографических прото-

колов формирования и проверки ЭЦП на примере алгоритме DSA, используя при этом программные средства библиотеки LockBox2.

2. Основные теоретические сведения

2.1. Криптографические хэш-функции

Функция хэширования (*hash function*) представляет собой преобразование, на вход которого подается сообщение переменной длины M , а выходом является строка фиксированной длины $h(M)$. Иначе говоря, хэш-функция h принимает в качестве аргумента сообщение (документ) M произвольной длины и возвращает хэш-значение $H = h(M)$ фиксированной длины.

Хэш-значение $h(M)$ – это так называемый дайджест сообщения M , то есть сжатое двоичное представление основного сообщения произвольной длины. Криптографические функции хэширования позволяют сжать подписываемый документ до 128 или более бит, тогда как M может быть размером в несколько мегабайт. Следует отметить, что значение хэш-функции $h(M)$ зависит сложным образом от документа M и не позволяет восстановить сам документ M .

Функция хэширования должна обладать следующими свойствами:

- может быть применима к аргументу любого размера;
- выходное значение хэш-функции имеет фиксированный размер;
- достаточно просто вычислить для любого M .
- чувствительна к всевозможным изменениям в тексте M , таким, как вставки, выбросы, перестановки и т.п.
- хэш-функция должна обладать свойством необратимости, то есть задача подбора документа M' , который обладал бы требуемым значением хэш-функции, должна быть вычислительно не разрешима;
- вероятность того, что значения хэш-функций различных документов совпадут, должна быть ничтожно мала.

Таким образом, функция хэширования может использоваться для обнаружения изменений сообщения, то есть она может служить для формирования *криптографической контрольной суммы*, также называемой *кодом аутентификации сообщения*. В этом качестве хэш-функция используется для контроля целостности сообщения, например, при формировании и проверке *электронной цифровой подписи*. **Хэш-функции также широко применяются в подсистемах**

безопасности прикладных программ и операционных систем при реализации протоколов аутентификации пользователей.

Наиболее популярными хэш-функциями являются MD2, MD4, MD5 и SHA. MD-алгоритмы (Message Digest) были разработаны Р. Райвестом. Каждый из них вырабатывает 128-битный хэш-код. Алгоритм MD2 самый медленный, MD4 – самый быстрый. MD5 является модификацией MD4, в котором пожертвовали скоростью ради увеличения безопасности. SHA (Secure Hash Algorithm) – алгоритм вычисления дайжеста сообщений, вырабатывающий 160-битовый хэш-код. Алгоритм SHA несколько надежнее алгоритмов MD4 и MD5.

На практике однонаправленные хэш-функции конструируются на основе так называемых сжимающих функций. Созданная таким образом однонаправленная функция возвращает хэш-значение разрядности n при заданных входных данных, больших размеров – разрядности m . Входными данными сжимающей функции являются блок сообщения и выходные данные предыдущих блоков текста. Выход представляет собой хэш-значение всех блоков, обработанных до текущего момента. То есть хэш-значение блока M_i равно:

$$h_i = f(M_i, h_{i-1}).$$

Это хэш-значение вместе со следующим блоком сообщения образуют следующие входные данные сжимающей функции (рис. 8.1).

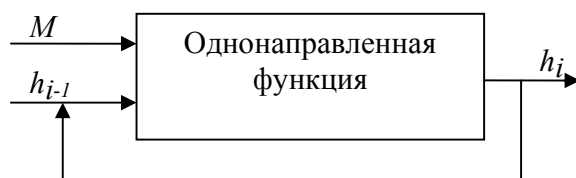


Рис. 8.1. Схема реализации однонаправленной функции шифрования

Хэш-значением всего сообщения является хэш-значение, полученное при обработке последнего блока.

2.2. Программные средства поддержки однонаправленных функций шифрования в LockBox2

В библиотеке LockBox2 поддерживаются несколько различных алгоритмов однонаправленного шифрования, в частности криптостойкие и широко распространенные MD5, SHA-1, а также несекретные и малоизвестные ELF, MIX128 и LockBox Message Digest (LMD).

MD5 является 128 битным алгоритмом хэширования. Входные данные в этом алгоритме разделяются на 512-битные блоки, каждый из которых на стадии смешивания разделяется на 32-битные части. Все операции в MD5 выполняются над 32-битными целыми числами, поэтому алгоритм весьма эффективен при применении на 32-битный процессорах.

Алгоритм MD5 требует для своей работы как отдельной фазы инициализации, так и отдельной стадии финализации. Несмотря на то, что LockBox2 предоставляет разработчику отдельные подпрограммы для инициализации (InitMD5), собственно хэширования (UpdateMD5) и финализации (FinalizeMD5), разработчикам рекомендовано использовать для вычисления хэш-значения единую подпрограмму HashMD5. Однако, если требуется полный контроль над процессом хэширования (например, в случае если хэшируемые данные разделены на части и появляются в различные временные отрезки и не могут быть объединены в одном буфере), то можно применить эти отдельные процедуры. Сначала вызывается InitMD5 для подготовки структуры контекста используемой двумя другими подпрограммами. Затем вызывается процедура UpdateMD5, которая выполняется наибольшее время для хэширования каждой отдельной части данных. Наконец, на последнем этапе структура контекста преобразуется в хэш-значение вызовом FinalizeMD5. Следующий пример показывает хэширование простого буфера:

```
var
    Context : TMD5Context;
    Digest  : TMD5Digest;
begin
    InitMD5(Context);
    UpdateMD5(Context, Buffer, sizeof(Buffer));
    FinalizeMD5(Context, Digest);
    // Digest содержит хэш-значение Buffer
end;
```

Процедура HashMD5 позволяет разработчику абстрагироваться от деталей инициализации и использования контекста алгоритма MD5. Эта процедура находит хэш-значение некоторого буфера Buf, размером BufSize, и размещает результат в Digest.

```
procedure HashMD5(var Digest: TMD5Digest; const Buf;
                  BufSize : LongInt);
TMD5Digest = array[0..15] of Byte;
```

Для своей реализации процедура HashMD5 использует рассмотренные ранее подпрограммы InitMD5, UpdateMD5 и FinalizeMD5.

Алгоритм SHA-1 (Security Hash Algorithm) был изобретен Национальным институтом стандартов и технологии (NIST) совместно с Агентством национальной безопасности США (NSA) для его использования в алгоритме цифровой подписи DSA (Digital Signature Algorithm).

SHA-1 генерирует 160-битный (20 байтный) дайжест. Входными блоками алгоритма являются 512-битные блоки. Как и в MD5, входные блоки на стадии микширования разделяются на подблоки по 32 бита, что делает этот алгоритм также весьма эффективным на 32-битных машинах.

По аналогии с MD5 алгоритм SHA-1 требует отдельных стадий: инициализации (InitSHA1), вычисления значения текущего блока (UpdateSHA1) и финализации (FinalizeSHA1).

Для упрощения операций имеется процедура HashSHA1, которая объединяет в себе все три операции:

```
procedure HashSHA1(var Digest: TSHA1Digest; const Buf;
                   BufSize : Longint);
TSHA1Digest = array[0..19] of byte;
```

Алгоритм SHA-1 является одним наиболее надежных из известных алгоритмов хэширования, даже несколько лучшим, нежели алгоритм MD5.

Реализованный в LockBox2 собственный алгоритм хэширования LMD предположительно обеспечивает секретность, но он не был подвергнут всестороннему криптоанализу, поэтому использовать его нужно с осторожностью, особенно в критических к секретности подсистемах программы. Идеологически LMD весьма схож с другими известными алгоритмами. Он также имеет раз-

дельные фазы и аналогичные по назначению библиотечные процедуры. Основным отличием является возможность формирования хэш-значения переменной длины от 1 до 256 байт. Для указания размера выходного дайжеста в процедуре HashLMD имеется отдельный параметр (DigestSize):

```
procedure HashLMD (var Digest; DigestSize: LongInt;
                   const Buf; BufSize : LongInt);
```

Простейший несекретный алгоритм хэширования ELF (Executable and Linking Format) был изначально разработан для вычисления 32-битных дайжестов строчных данных при компоновке объектных файлов. Этот алгоритм нашел широкое применение для реализации таких структур данных, как словари и хэш-таблицы. В LockBox2 реализованы две подпрограммы для поддержки этого алгоритма. Первая процедура HashELF вычисляет 32-битный дайжест буфера заданного размера:

```
procedure HashELF(var Digest : LongInt; const Buf;
                  BufSize : LongInt);
```

вторая подпрограмма StringHashELF находит ELF-дайжест строки:

```
procedure StringHashELF(var Digest : LongInt; const Str : string);
```

Другим примитивным алгоритмом хэширования, который также поддерживается библиотекой LockBox2, является алгоритм Mix128. Этот алгоритм так же, как и ELF, формирует 32-битное хэш-значение. Алгоритм является весьма быстрым и может быть использован для эффективной реализации словарей и хэш-таблиц. В LockBox2 имеются две подпрограммы поддержки этого алгоритма: HashMix128 и StringHashMix128, которые семантически эквиваленты соответствующим подпрограммам поддержки ELF-алгоритма.

2.3. Алгоритм DSA. Математические основы реализация

В августе 1991 года Национальное агентство по стандартам и торговле США в своем стандарте DSS (Digital Signature Standart) предложило алгоритм цифровой подписи DSA (Digital Signature Algorithm). Как было опубликовано позднее, алгоритм был разработан непосредственно Агентством национальной безопасности США по заказу НАСТ. Алгоритм DSA не является международ-

ным стандартом и используется в основном в США. Однако, несмотря на то, что DSA разработан АНБ США, его криптостойкость не вызывает сомнения.

Согласно спецификации алгоритма DSA отправитель и получатель электронного сообщения используют для формирования ключей большие целые числа: g и p – простые числа длиной L бит каждое, причем $L \in (512, 1024)$, а также q – простое число длиной 160 бит, которое является делителем числа $p - 1$. Исходные числа g , p и q являются открытыми и могут быть общими для всех пользователей криптосети.

Отправитель сообщения, формирующий электронную цифровую подпись (ЭЦП) выбирает случайное целое число x , такое, что $x \in (1, q)$. Число x является секретным ключом отправителя, которое используется для формирования ЭЦП.

Затем отправитель вычисляет значение $y = g^x \bmod p$, которое будет представлять собой открытый ключ для проверки подписи отправителя. Число y передается всем получателям документов, которые будут выполнять их аутентификацию. Алгоритм DSA предусматривает использование односторонней функции шифрования, в качестве которой определен алгоритм безопасного хэширования SHA.

Для того чтобы подписать сообщение M , отправитель выполняет следующие шаги:

1. Выбирать случайное целое число k в интервале $[1, q-1]$.
2. Вычислить $r = (g^k \bmod p) \bmod q$.
3. Вычислить $s = k^{-1} \{h(M) + xr\} \bmod q$.
4. Если s получается равным нулю, тогда выбирается новое k и процесс повторяется сначала.
5. Если s оказывается ненулевым, то подпись сообщения M признается сформированной верной и представляется в виде пары чисел r и s .

Для того чтобы проверить аутентичность подписанного документа M , получатель должен выполнить следующие стадии проверки:

1. Получить подлинную копию открытого ключа отправителя y .
2. Вычислить $w = s^{-1} \bmod q$ и хэш-значение $h(M)$.
3. Вычислить два значения $u_1 = h(M)w$ и $u_2 = (rw) \bmod q$.
4. Далее, используя открытый ключ вычислить $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$.
5. Подпись документа M признается подлинной, если значения v и r совпадают.

Алгоритм DSA представляет собой вариант алгоритмов подписи Шнорра и Эль-Гамала. Безопасность алгоритма цифровой подписи DSA базируется на трудностях задачи дискретного логарифмирования. При использовании с ключом 512 бит алгоритм DSA недостаточно стоек для обеспечения длительной безопасности, но вполне надежен при ключе 1024 бита.

2.4. Использование LockBox2 для формирования ЭЦП с помощью алгоритма DSA

В LockBox2 поддержка алгоритмов ЭЦП появилась в только в последней версии. Специфика протоколов подписывания и проверки ЭЦП обусловила то, что функциональность ЭЦП в LockBox2 реализована в виде классов, инкапсулирующих связанные методы и свойства.

В библиотеке имеются два класса поддержки ЭЦП: класс TLbDSA, обеспечивающий работу с ЭЦП на основе алгоритма DSA, и класс TLbRSASSA, в котором используется алгоритм RSASSA (RSA Signature Scheme with Appendix). Оба класса являются наследниками класса TLbSignature. Класс TLbSignature управляет базовой функциональностью подписывающей «машины», в частности вычислением цифровой подписи буфера, файла, байтового потока и строк. Эта функциональность представлена в классе TLbSignature в виде виртуальных методов, которые должны перегружаться в классах-наследниках TLbDSA и TLbRSASSA. Таким образом, класс TLbSignature создает некоторый универсальный интерфейс доступа к подписывающим «машинам», которые реализованы в классах-наследниках. Рассмотрим основные методы этого класса.

Поскольку все алгоритмы цифровой подписи являются разновидностями алгоритмов шифрования с открытым ключом, поэтому в них также перед началом подписывания данных должна быть сформирована пара криптографических ключей: *закрытый* – для подписывания и *открытый* – для проверки подписи. Класс TLbSignature имеет виртуальный абстрактный метод GenerateKeyPair, который предназначен для вычисления таких ключей:

```
procedure GenerateKeyPair; virtual; abstract;
```

Напомним, что абстрактным методом называется такой метод класса, который не имеет реализации (только заголовок) и обязательно должен быть реализован в производном классе. Фактически виртуальный абстрактный метод

является своего рода «контрактом» между родительским и дочерним классом, который все дочерние классы обязаны выполнить, т.е. наполнить такой метод некоторой функциональностью. Ключевое слово **VIRTUAL** в определении метода класса указывает на возможность перегружать (создавать собственный одноименный метод) такие методы в классах-потомках, делая возможным на этапе времени исполнения вызывать метод текущего класса или одноименный метод какого-либо из его предков.

Для подписывания и проверки цифровой подписи в классе имеются пары однотипных методов (также абстрактных). Для работы с буферами данных в памяти предназначается пара методов `SignBuffer` и `VerifyBuffer`:

```
procedure SignBuffer(const Buf; BufLen: Cardinal); virtual; abstract;
function VerifyBuffer(const Buf; BufLen: Cardinal): Boolean; virtual; abstract;
```

где `Buf` – указатель на буфер с данными, `BufLen` – размер буфера в байтах.

Для работы с файлами имеются методы `SignFile` и `VerifyFile`:

```
procedure SignFile(const AFileName: string); virtual; abstract;
function VerifyFile(const AFileName: string) : Boolean; virtual;
abstract;
```

где `AFileName` – полный путь к подписываемому или проверяемому файлу.

Формирование и проверка ЭЦП для строк выполняется с помощью методов `SignString` и `VerifyString`:

```
procedure SignString(const AStr: string); virtual; abstract;
function VerifyString(const AStr: string): Boolean; virtual; abstract;
```

где параметр `AStr` указывает на строчные данные. Класс `TLbSignature` также имеет методы `SignStream` и `VerifyStream` для подписывания и проверки данных в виде байтовых потоков.

Рассматриваемый класс реализует функциональность одного важного свойства – `KeySize`, которое имеет тип `TLbAsymKeySize`. Это свойство позволя-

ет устанавливать размер ключа в алгоритмах ЭЦП. Это свойство может принимать значения, определяемые перечислимым типом `TLbAsymKeySize`:

```
TLbAsymKeySize = (aks128, aks256, aks512, aks768, aks1024);
```

задавая соответствующий размер ключа: 128, 256, 768 или 1024 бита.

Класс `TLbDSA` является наследником класса `TLbSignature` и реализует все его абстрактные методы, а также дополняет собственными интерфейсами и функциональностью, которая обусловлена особенностью реализации алгоритма DSA. Среди свойств класса `TLbDSA` можно выделить свойства `PrivateKey` и `PublicKey`, которые предназначены для хранения пары криптографических ключей, а также свойства `SignatureR` и `SignatureS`, которые содержат компоненты цифровой подписи. Свойство `PrimeTestIterations` определяет число итераций проверки исходных чисел p , q и g на то, являются ли они простыми. По умолчанию это свойство устанавливается при вызове конструктора и получает значение, равное 20. Увеличение значения этого параметра снижает риск генерации составных чисел p , q и g , что может повлиять на раскрытие ключей ЭЦП. Однако большое значение `PrimeTestIterations` резко увеличивает время генерации ключей.

Указанные исходные разделяемые параметры DSA могут быть сформированы отдельно с помощью функции класса `TLbDSA` – `GeneratePQG`. В результате работы этой функции создаются новые параметры p , q и g , которые могут быть использованы позднее для генерации серии пар криптографических ключей x и y . Раздельная генерация ключей возможна с помощью вызова другого метода класса `TLbDSA` – `GenerateXY`. По умолчанию `GenerateXY` использует псевдослучайное число в качестве «затравки».

Рассмотрим простейший пример, который демонстрирует общую схему использования объекта класса `TLbDSA`:

```
var
    DSA: TLbDSA;
begin
    DSA := TLbDSA.Create(nil);
    try
        // используем 512 ключ
        DSA.KeySize := aks512;
```

```

// генерируем параметры и криптографические ключи
DSA.GenerateKeyPair;
// подписываем текст
// после этого объект заполняет свойства
// SignatureR и SignatureS
DSA.SignString('LeoSoft Solution');
// выполняем проверку
if not DSA.VerifyString('LeoSoft Solution') then
    // если не выполнена возбуждаем исключение
    raise Exception.Create('Сигнатура не совпадает');
finally
    DSA.Free;
end;
end;

```

3. Практическая часть

3.1. Пример практического использования алгоритма ЭЦП DSA


Рассмотрим на примере подписывания файла с данными использование алгоритма DSA и соответствующих средств библиотеки LockBox2.

1. Используем проект приложения, разработка которого была начата в первых двух лабораторных работах второй части практикума (№ 6 и № 7).

2. Используем некоторые уже имеющиеся элементы управления в приложении, в частности воспользуемся компонентом PageControl и создадим еще одну закладку типа TabSheet (команда New Page контекстного меню PageControl). В инспекторе объектов изменим заголовки закладки по умолчанию на новый, задав свойству Caption значение «Алгоритм DSA» (рис. 8.2).

3. В клиентскую область новой закладки добавим несколько элементов управления и определим их идентификаторы, а также установим некоторые свойства:

- три командные кнопки типа Button  (1-3) с идентификаторами (свойство Name) SelectFileDSAButton (1), DSASignButton (2), DSAVerifyButton (3)

• три строки редактирования (4-6) типа Edit  с идентификаторами SignedDSAFileNameEdit, RCompotentSignEdit, SCompotentSignEdit. Строки редактирования имеют ассоциированные необязательные подписи, указывающие на их предназначение в программе.

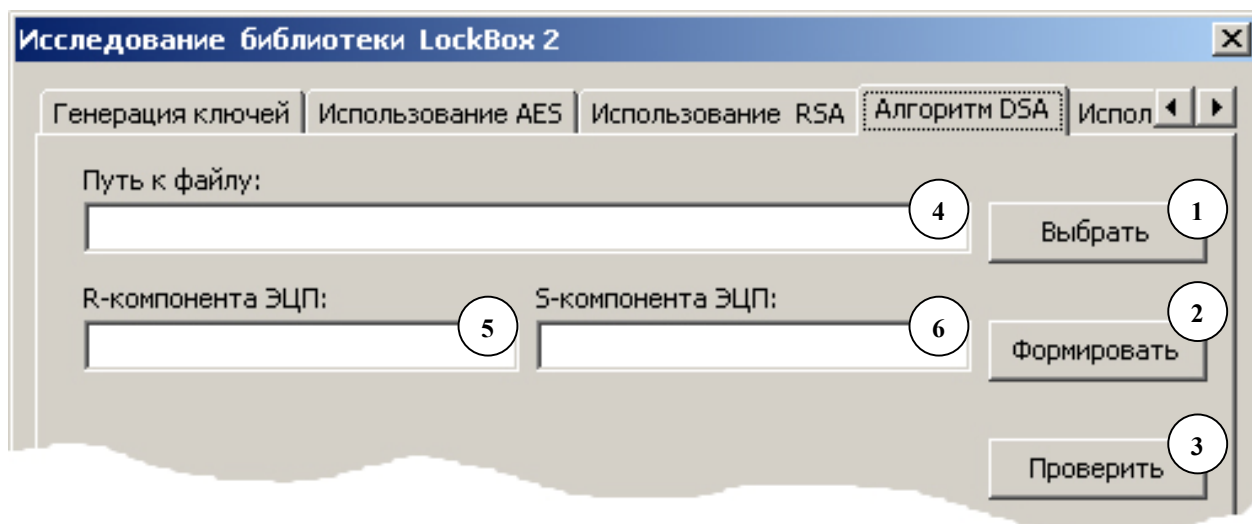


Рис. 8.2. Внешний вид окна приложения (четвертая вкладка)

4. Командным кнопкам назначим обработчики события OnClick. Код этих подпрограмм составит функциональную основу этой части приложения:

4.1. Кнопка SelectFileDSAButton предназначена для выбора полного пути к подписываемому файлу. В процедуре для выбора файла используется компонент OpenFileDialog.

```
procedure TMainForm.SelectFileDSAButtonClick(Sender: TObject);
begin
    OpenFileDialog.Filter :=
        'Текстовые файлы|*.txt;*.pas;*.asm;*.c;*.cpp';
    if OpenFileDialog.Execute() then
        begin
            // если файл был выбран, то запоминаем имя исходного файла
            SignedDSAFileNameEdit.Text := OpenFileDialog.FileName;
            FileName:= SignedDSAFileNameEdit.Text;
        end;
end;
end;
```

4.2. Обработчик нажатия на кнопку DSASignButton выполняет действия, связанные с генерацией криптографических ключей и формирования ЭЦП:

```

procedure TMainForm.DSASignButtonClick(Sender: TObject);
var
    // подписывающая машина
    DSA: TLbDSA;
    // 20-ти байтный буфер для временного
    // размещения R и S компонент подписи
    Block: TLbDSABlock;
    // файл для хранения R и S компонент подписи
    SignFile: file of TLbDSABlock;
begin
    // проверяем был ли выбран файл
    if not FileExists(FileName) then
        begin
            ShowMessage('Не выбран файл или файл не существует');
            Exit;
        end;
    // создаем объект класса TLbDSA
    DSA := TLbDSA.Create(nil);
    try
        // показываем песочные часы перед длительной операцией
        Screen.Cursor := crHourGlass;
        // короткий ключ и малое число итераций
        DSA.KeySize := aks128;
        DSA.PrimeTestIterations := 5;
        // генерируем ключевую пару XY и разделяемые параметры PQG
        DSA.GenerateKeyPair;
        // сохраним открытый ключ в файле
        // он понадобится впоследствии для проверки ЭЦП
        DSA.PublicKey.StoreToFile(FileName + '.DSA.key');
        // подписываем файл, используя закрытый ключ
        DSA.SignFile (FileName);
        // показываем R и S компоненты сигнатуры
        RCompotentSignEdit.Text := DSA.SignatureR.IntStr;
    finally
        DSA.Free;
    end;

```

```

SCompotentSignEdit.Text := DSA.SignatureS.IntStr;
// теперь нужно сохранить компоненты ЭЦП в файле
// но увы, готового метода нет, поэтому делаем это руками
// получаем файловый дескриптор
AssignFile(SignFile, FileName + '.DSA.sign');
// открываем для перезаписи
Rewrite(SignFile);
// помещает в буфер R компоненту
DSA.SignatureR.ToBuffer(Block, SizeOf(Block));
// записываем буфер в файл
Write(SignFile, Block);
// помещает в буфер S компоненту
DSA.SignatureS.ToBuffer(Block, SizeOf(Block));
// записываем буфер в файл
Write(SignFile, Block);
// закрываем файл
CloseFile(SignFile);

finally
    // разрушает подписывающую машину
    DSA.Free;
    // курсор по умолчанию
    Screen.Cursor := crDefault;
end;
end;

```

Вторая половина ключевой пары, т.е. открытый ключ с помощью которого должна выполняться проверка ЭЦП, записывается в файл с использованием метода StoreToFile объекта класса TLbAsymmetricKey. Для проверки ЭЦП также необходима сама цифровая подпись, поэтому ее компоненты также сохраняются в отдельный файл, однако при этом используются стандартные для Delphi операции в/в для типизированных файлов.

4.3. Проверка ЭЦП выполняется после нажатия кнопки DSAVerifyButton, исходный текст обработчика OnClick для которой представлен ниже:

```

procedure TMainForm.DSAVerifyButtonClick(Sender: TObject);
var

```

```

    DSA: TLbDSA;
    Block: TLbDSABlock;
    SignFile: file of TLbDSABlock;
begin
    DSA := TLbDSA.Create(nil);
    try
        DSA.KeySize := aks128;
        DSA.PublicKey.LoadFromFile(FileName + '.DSA.key');
        // считываем из файла компоненты цифровой подписи
        AssignFile(SignFile, FileName + '.DSA.sign');
        // открываем для чтения
        Reset(SignFile);
        // считываем блок с S-компонентой
        Read(SignFile, Block);
        // копируем его в подписывающую машину
        DSA.SignatureR.CopyBuffer(Block, SizeOf(Block));
        // считываем блок с R-компонентой
        Read(SignFile, Block);
        // копируем его в подписывающую машину
        DSA.SignatureS.CopyBuffer(Block, SizeOf(Block));
        CloseFile(SignFile);
        // проверяем подпись под файлом
        if DSA.VerifyFile(FileName) then
            ShowMessage('Подпись верна')
        else
            ShowMessage('Подпись неверна');
    finally
        DSA.Free;
    end;
end;

```

Для того чтобы операции формирования и проверки были независимы друг от друга, открытый ключ и компоненты ЭЦП записываются в отдельные файлы на диске. Поэтому объект класса TLbDSA в обработчике DSAVerifyButtonClick должен инициализировать открытый ключ и считать из файла компоненты ЭЦП.

3.2. Задания

1. Самостоятельно исследовать функциональность класса TLbRSASS и реализовать программу подсистему формирования и проверки цифровой подписи с использованием алгоритма RSASS.
2. Проанализировать время формирования хэш-значения в криптостойких однонаправленных функциях шифрования (MD5, SHA-1).

4. Контрольные вопросы

1. Какую задачу призвана решать электронная цифровая подпись в реализации технологии защиты информации?
2. Что составляет информационную основу электронной цифровой подписи?
3. При реализации каких структур данных применяются некриптографические функции однонаправленного шифрования (функциями хеширования)?
4. Какой из алгоритмов электронной цифровой подписи DSA или RSASS обеспечивает более надежное подписывание информации?



БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Миллер Т., Пауэл Д. Использование Delphi 3. Специальное издание.: Пер. с англ. – К.: Диалектика, 1997. – 768 с.
2. Бакнелл Д. Фундаментальные алгоритмы и структуры данных в Delphi.: Пер. с англ. – СПб.: ДиаСофтЮП, 2003. – 560 с.
3. Макконнел С. Совершенный код. Мастер-класс.: Пер. с англ. – СПб.: Питер, 2005. – 896 с.
4. Грегори К. Использование Visual C++ 6. Специальное издание.: Пер. с англ. – М.; Спб.; К.: Вильямс, 1999. – 864 с.
5. Руссинович М., Соломон Д. Внутреннее устройство Microsoft Windows: Microsoft Windows Server 2003, Windows XP и Windows 2000.: Пер. с англ. – СПб.: Питер, 2005. – 992 с.
6. Шнайер Б. Прикладная криптография. Протоколы, алгоритмы, исходные тексты на языке Си. : Пер. с англ. – М.: Триумф, 2003. – 816 с.
7. Галицкий А.В., Рябко С. Д., Шаньгин В.Ф. Защита информации в сети – анализ технологий и синтез решений – М.: ДМК Пресс, 2004. – 616 с.

ОГЛАВЛЕНИЕ

ЧАСТЬ I. ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ В СРЕДЕ DELPHI

Лабораторная работа №1. ОСНОВЫ ИСПОЛЬЗОВАНИЯ ИНТЕГРИРОВАННОЙ СРЕДЫ РАЗРАБОТКИ DELPHI ПРИ ПОСТРОЕНИИ ПРОСТЕЙШИХ WINDOWS- ПРИЛОЖЕНИЙ

1. Цель работы
2. Основные теоретические сведения
- 2.1. Введение в интегрированные среды разработки программ.....
- 2.2. Основные компоненты среды Delphi
3. Практическая часть
- 3.1. Первое простейшее приложение
- 3.2. Первое простейшее приложение (продолжение).....
- 3.2. Задания
4. Контрольные вопросы

Лабораторная работа № 2. ПОНЯТИЕ О ПАРАДИГМАХ ПРОГРАММИРОВАНИЯ. ОСНОВЫ СИНТАКСИСА И СЕМАНТИКИ ЯЗЫКА ОБЪЕКТ PASCAL

1. Цель работы
2. Основные теоретические сведения
- 2.1. Основные понятия и система принципов алгоритмических языков программирования
- 2.2. Лексические единицы языков программирования
- 2.3. Переменные и их декларация. Типы данных. Операция присваивания и чтения значений переменной
- 2.4. Структурные операторы. Операторный блок
- 2.5. Массивы
3. Практическая часть
- 3.1. Примеры использования циклических операторов в ОР
- 3.2. Примеры использования условных операторов ОР
- 3.3. Задания
4. Контрольные вопросы

Лабораторная работа № 3. ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ В ОБЪЕКТ PASCAL. ПОЛЬЗОВАТЕЛЬСКИЕ ТИПЫ ДАННЫХ. ПРОГРАММИРОВАНИЕ ФАЙЛОВЫХ ОПЕРАЦИЙ

1. Цель работы
2. Основные теоретические сведения

2.1. Фундаментальные идеи процедурного программирования.....	
2.2. Реализация принципов процедурного программирования в ОР	
2.3. Записи и файловый тип данных	
3. Практическая часть	
3.1. Организация циклической структуры с использованием инструкций условных переходов	
3.1. Использование файлов и файловых операций	
3.2. Задания	
4. Контрольные вопросы	

Лабораторная работа № 4. ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ ПАРАДИГМА ПРОГРАММИРОВАНИЯ В ОБЪЕКТ PASCAL. КЛАССЫ И ОБЪЕКТЫ.

1. Цель работы	
2. Основные теоретические сведения	
2.1. Реализация объектно-ориентированной парадигмы в Object Pascal	
3. Практическая часть	
3.1. «Жизнь» и «смерть» объектов в режиме run-time. Динамическое соз- дание визуальных компонентов	
3.2. Модальные диалоговые окна	
3.3. Немодальные диалоговые окна	
3.4. Задания	
4. Контрольные вопросы	

Лабораторная работа № 5. ВЗАИМОДЕЙСТВИЕ ПРИКЛАДНОЙ WINDOWS-ПРОГРАММЫ С ДРАЙВЕРАМИ УСТРОЙСТВ

1. Цель работы	
2. Основные теоретические сведения	
2.1. Компоненты подсистемы ввода/вывода Windows 2000	
2.2. Диспетчер ввода/вывода и типовая обработка ввода/вывода	
2.3. Драйверы устройств и их классификация	
2.4. Унифицированная структура драйвера режима ядра	
2.5. Объект «драйвер» и объект «устройство». Виды устройств. Интерфейс с прикладной программой	
3. Практическая часть	
3.1. Управление простыми устройствами с использованием драйверов ре- жима ядра	
3.2. Задания	
4. Контрольные вопросы	

ЧАСТЬ II. ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ С ПРИМЕНЕНИЕМ ТЕХНОЛОГИЙ ЗАЩИТЫ ИНФОРМАЦИИ

Лабораторная работа № 6. СИММЕТРИЧНАЯ КРИПТОГРАФИЯ. ПРИМЕНЕНИЕ АЛГОРИТМОВ С ЗАКРЫТЫМ КЛЮЧОМ ДЛЯ ЗАЩИТЫ ИНФОРМАЦИИ В ПРИКЛАДНЫХ ПРОГРАММАХ. БИБЛИОТЕКА АЛГОРИТМОВ ШИФРОВАНИЯ LOCKBOX2

1. Цель работы
2. Основные теоретические сведения
- 2.1. Основные понятия криптографической защиты данных
- 2.2. Введение в библиотеку LockBox2
- 2.3. Методы генерации ключей в LockBox2
- 2.4. Программные средства симметричного шифрования в LockBox2
 - 2.4.1. Процедуры и функции поддержки DES-шифрования.....
 - 2.4.1. Процедуры и функции поддержки AES-шифрования.....
3. Практическая часть
- 3.1. Генерация ключей
- 3.2. Использование шифра AES
- 3.3. Задания
4. Контрольные вопросы

Лабораторная работа № 7. АСИММЕТРИЧНАЯ КРИПТОГРАФИЯ ПРИМЕНЕНИЕ АЛГОРИТМОВ С ОТКРЫТЫМ КЛЮЧОМ ДЛЯ ЗАЩИТЫ ИНФОРМАЦИИ В ПРИКЛАДНЫХ ПРОГРАММАХ.....

1. Цель работы
2. Основные теоретические сведения
- 2.1. Асимметричные криптосистемы.....
- 2.2. Реализация алгоритма асимметричного шифрования RSA
- 2.3. Средства поддержки асимметричного шифрования в LockBox2...
3. Практическая часть
- 3.1 Использование асимметричного шифра RSA
- 3.2. Задания
4. Контрольные вопросы

Лабораторная работа № 8. АСИММЕТРИЧНАЯ КРИПТОГРАФИЯ. АЛГОРИТМЫ ЦИФРОВОЙ ПОДПИСИ И ИХ ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ

1. Цель работы	
2. Основные теоретические сведения	
2.1. Криптографические хэш-функции	
2.2. Программы средства поддержки однонаправленных функций шифрования в LockBox2	
2.3 Алгоритм DSA. Математические основы реализации	
2.4. Использование LockBox2 для формирования ЭЦП с помощью алгоритма DSA.	
3. Практическая часть	
3.1.Пример практического использования алгоритма ЭЦП DSA	
3.2. Задания	
4. Контрольные вопросы	
БИБЛИОГРАФИЧЕСКИЙ СПИСОК	

Вячеслав Евгеньевич Леонтьев

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ В СРЕДЕ BORLAND DELPHI

Лабораторный практикум
по курсу
«Программное обеспечение измерительных процессов»
«Защита информации»

(Кафедра информатики и информационных управляющих систем КГЭУ)

Редактор издательского отдела Н.И. Волокитина

Изд. лиц. № 03480 от 8.12.00

Подписано к печати

Формат 60 x 84/16

Гарнитура «Times»

Вид печати РОМ

Бумага «Business»

Физ. печ. л.

Усл. печ. л.

Уч-изд. л.

Тираж 70

Заказ

Издательский отдел КГЭУ
420066, Казань, Красносельская, 51

Типография КГЭУ
420066, Казань, Красносельская, 51