Министерство образования Российской Федерации
Томский политехнический университет

В.Г. Букреев, Н.В. Гусев

DELPHI-6 – СРЕДА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ СИСТЕМ ПРОМЫШЛЕННОЙ АВТОМАТИЗАЦИИ

Учебное пособие

Издательство ТПУ
Томск 2004

ББК 24.7
УДК 621.3.011.1: 658.011.56
Б 90

Букреев В.Г., Гусев Н.В.

Б 90 Delphi-6 – среда разработки программного обеспечения для систем промышленной автоматизации: Учебное пособие. – Томск: Изд-во ТПУ, 2004. – 103 с.

ISBN

В пособии изложены основы разработки программного обеспечения для систем промышленной автоматизации в среде Delphi-6. Приведено описание языка программирования Object Pascal. Рассмотрены примеры моделирования сложных электромеханических систем, построения траекторий движения многокоординатных электроприводов промышленных механизмов.

Пособие подготовлено на кафедре электропривода и электрооборудования и предназначено для студентов специальности 180400 «Электропривод и автоматика промышленных установок и технологических комплексов».

ББК 24.7
УДК 621.3.011.1: 658.011.56

Рекомендовано к печати Редакционно-издательским советом Томского политехнического университета

Рецензенты

Кандидат технических наук заведующий отделом НИИ автоматики и электромеханики при ТУСУРЕ
Ю.Н. Андреев

Кандидат технических наук, заместитель руководителя отдела электрофизических технологий обработки воды при Конструкторско-технологическом центре Томского научного центра Сибирского отделения СО РАН
Н.П. Поляков

ISBN

ВВЕДЕНИЕ

Современные приложения для систем промышленной автоматизации – это, прежде всего такие приложения, которые должны работать в режиме реального времени, непрерывно опрашивать состояние объекта управления и выдавать управляющие воздействия на объект управления. Это, в свою очередь, ставит перед разработчиком приложения специфичные задачи:

- обеспечение стабильной частоты опроса контролируемых параметров;
- представление информации о состоянии всего объекта управления и отдельных узлов;
- в режиме тестирования оператор должен иметь возможность подать команду на каждый исполнительный механизм объекта управления;
- информация о работе объекта управления должна сохраняться в архивах и по запросам оператора представлена в виде таблиц и графиков.

В зависимости от требований, предъявляемых к быстродействию и надежности системы, приложение может работать в режиме “жесткого реального” времени либо в режиме "нежесткого" реального времени. При работе систем управления в режиме “жесткого реального” времени используются специальные операционные системы, такие как QNX, RTOS-32, VxWorks и ряд других. В случае управления объектом, работающим в режиме "нежесткого" реального времени, наиболее актуальным на сегодняшний день является применение операционной системы линии Windows NT. Это обусловлено простотой разработки приложений под данную операционную систему, широкой ее распространенностью и большим количеством объектов управления не требующих “жесткого реального” времени.

В данном пособии рассмотрены основные вопросы разработки приложений для АСУ в среде Delphi под управлением операционной системы Windows NT, приведены некоторые практические рекомендации и советы по решению тех или иных задач, возникающих в процессе создания программного обеспечения для систем промышленной автоматизации.

1. ВВЕДЕНИЕ В DELPHI

1.1. Среда разработчика

Среда разработчика представляет собой полнофункциональный инструмент, объединяющий все средства, необходимые для создания приложений : редактор исходных текстов, менеджер проектов, палитру компонентов, инспектор объектов и ряд дополнительных утилит (встроенный отладчик, браузер объектов, дизайнер меню и т.п.). Более того, она обеспечивает возможность подключения к среде внешних утилит.

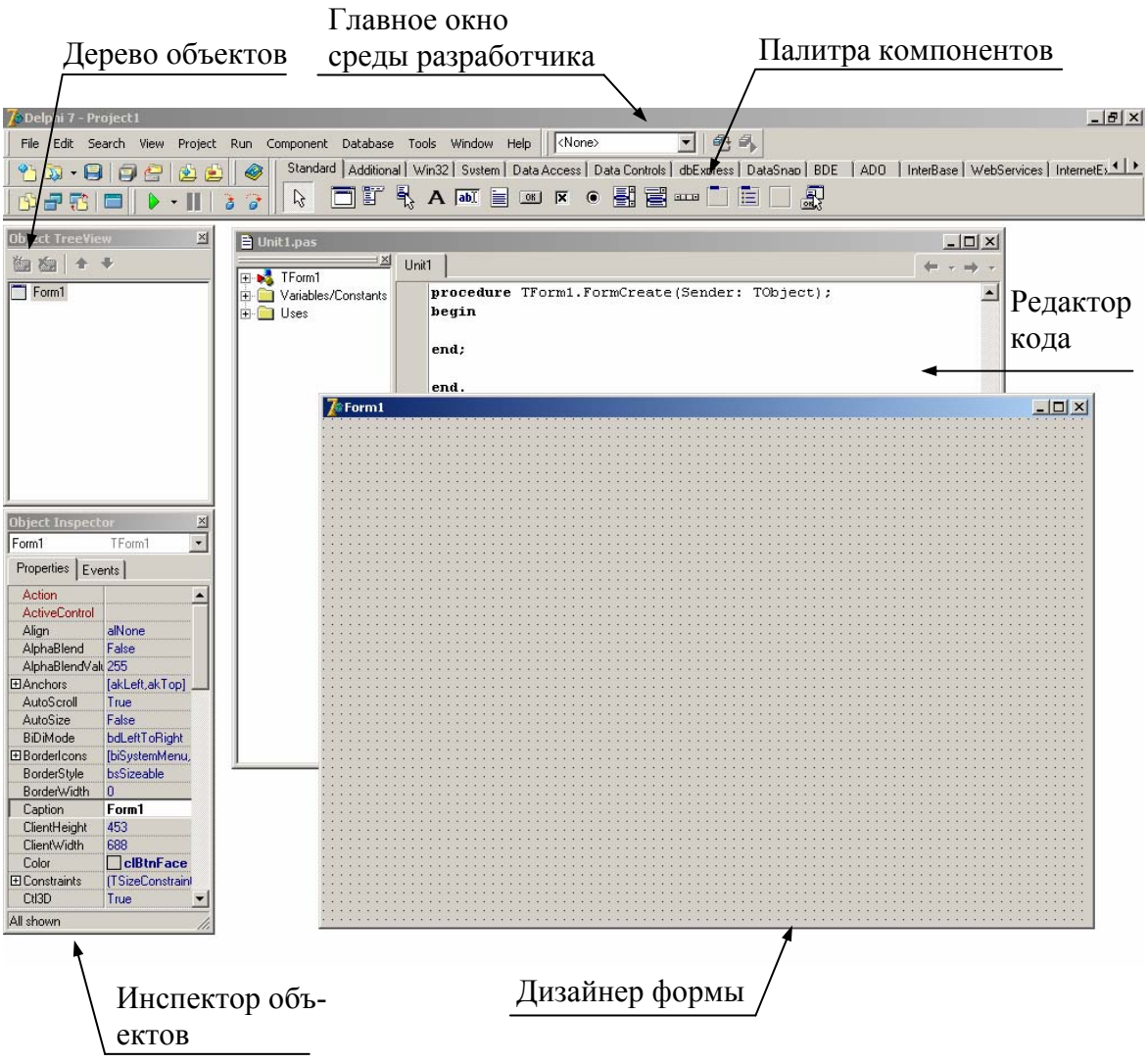


Рис. 1.1. Интегрированная среда разработчика

Главное окно

Помимо традиционного меню, содержащего базовые команды File (Файл), Edit (Правка), Search (Поиск), View (Вид), Project (Проект), Run

(Запуск), Component (Компонент), Database (База данных), Tools (Инструменты), Window (Окно), Help (Помощь), главное окно среды разработчика включает полосу быстрого доступа к командам и палитру компонентов.

Палитра компонентов

Палитра компонентов позволяет выбрать визуальные и другие компоненты, которые будут присутствовать в нашем приложении. Компоненты, располагаемые в палитре, представлены в виде кнопок. Компонентами могут быть, например, кнопки, списки (визуальные компоненты) или таблицы для доступа к базам данных (невизуальные компоненты).

Дерево объектов

Это окно появилось в 6-й версии Delphi и предназначено для наглядного отображения связей между отдельными компонентами, размещенными на активной форме или в активном модуле данных. Щелчок по любому компоненту в этом окне активизирует соответствующий компонент в окне формы и отображает свойства этого компонента в окне инспектора объектов. Двойной щелчок приводит к срабатыванию механизма Code Insight (кодового проникновения), который вставляет в окно кода заготовку для обработчика события OnClick. Наконец, компонент можно “перетащить” в окне и таким образом поменять его владельца (свойство parent).

Редактор кода

Редактор кода – это еще один обязательный элемент среды разработчика, он используется для непосредственного написания кода. Создаваемый инспектором объектов код типа шаблона обработчика события, а также код, создаваемый самой средой Delphi, помещаются в окне редактора, они доступны для редактирования в любой момент создания программы. Редактор позволяет осуществлять выделение синтаксиса цветом, имеет неограниченную возможность отмены действий и возможность переключения между всеми исходными файлами, входящими в проект.

Инспектор объектов

Инспектор объектов позволяет устанавливать свойства объектов и назначать методы-обработчики событий во время разработки программы. Расположив необходимые объекты в форме, вы можете изменять их свойства – список свойств каждого объекта отображается в инспекторе

объектов. Инспектор объектов отображает также список событий, обрабатываемых объектом, и код для каждого обработчика. Такой подход существенно упрощает связь кода с интерфейсными элементами, а также с неотображаемыми элементами типа компонентов для управления базами данных.

Окно инспектора объектов содержит две страницы – Properties (Свойства) и Events (События). Страница Properties служит для установки нужных свойств компонента, страница Events позволяет определить реакцию компонента на то или иное событие. Совокупность свойств отображает видимую сторону компонента: положение относительно левого верхнего угла рабочей области формы, его размеры и цвет, шрифт и текст надписи на нем и т. п.; совокупность событий – его поведенческую сторону: будет ли компонент реагировать на щелчок мыши или на нажатие клавиш, как он будет вести себя в момент появления на экране или в момент изменения размеров окна и т. п.

Встроенный отладчик

Любая система создания приложений была бы неполной без средства отладки программ. Среда разработчика Delphi включает в себя интегрированный отладчик, который позволяет выполнять пошаговую трассировку кода, устанавливать различные точки останова (break points), узнавать значения различных выражений и просматривать стек вызовов.

Дизайнер формы

Дизайнер формы представляет собой чистую форму, на которой разработчик устанавливает необходимые ему компоненты.

Вся рабочая область окна формы обычно заполнена точками координатной сетки, служащей для упорядочения размещаемых на форме компонентов (вы можете убрать эти точки, вызвав с помощью меню Tools | Environment options соответствующее окно настроек и убрав флажок в переключателе Display Grid на окне, связанном с закладкой Preferences).

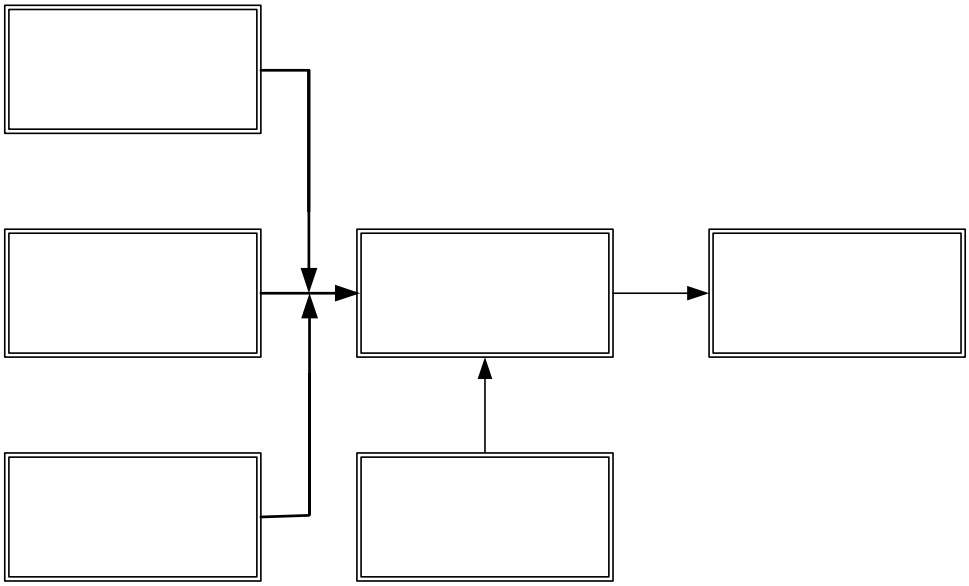
Приложение может включать в себя несколько форм.

1.2. Проекты

В среде Delphi вы работаете с проектами – наборами файлов, из которых состоит создаваемое вами приложение. Ими могут быть файлы, создаваемые в среде Delphi, – файлы с исходным текстом модулей

(расширение *.PAS) формы, являющиеся графическим представлением вашего приложения (расширение *.DFM), и сами файлы проектов (расширение *.DPR). Следует отметить, что каждому файлу формы обязательно соответствует файл с исходным текстом модуля, но файл с исходным текстом модуля не обязательно должен иметь соответствующую ему форму.

Файл проекта связывает вместе все файлы, из которых состоит приложение, и, таким образом, среда Delphi «знает», какие файлы необходимы для создания (сборки) приложения.



1.2. Процесс создания исполняемого файла

Файл проектов

Когда вы начинаете создавать новый проект, выбрав команду File | New Project, среда Delphi создает файл проекта и управляет им в процессе создания приложения. Файл проекта сохраняется с расширением *.DPR, и для каждого проекта может быть только один такой файл. По умолчанию проект имеет название PROJECT1. Для каждого проекта файл проекта может выглядеть следующим образом:

Файлы форм
***.DFM**

Модули
***.PAS**

Про
***.D**

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
```

```
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

Отметим, что использование модуля Forms обязательно для всех программ, создаваемых в среде Delphi, так как в этом модуле содержится описание класса TApplication, который лежит в основе всех Delphi-приложений. В приведенном примере модуль UNIT1.PAS описывает главную форму, включенную в данное приложение. Название формы (ее идентификатор), которое должно отличаться от названия модуля, описывающего ее, приводится в фигурных скобках. Это название соответствует значению свойства Name формы. Директива in указывает на то, что модуль является обязательной частью проекта, а не просто файлом, используемым в проекте. Директива $R подключает к создаваемому файлу (проекту) все необходимые ресурсы (все файлы с расширением *.RES). Отметим, что само изображение формы хранится в виде Windows-ресурса, но имеет расширение *.DFM (Delphi Form). Вызов метода Application.Initialize приводит к инициализации самого приложения, представленного экземпляром класса TApplication. Метод Application.CreateForm загружает и инициализирует форму, а метод Application.Run начинает выполнение приложения и загружает главную форму приложения. Каждый раз, когда вы добавляете к проекту новую форму или новый модуль, Delphi автоматически добавляет директиву uses в файл проекта. Точно так же в файл проекта автоматически добавляются вызовы метода Application.CreateForm для загрузки и инициализации дополнительных форм, используемых в данном приложении.

Так как среда Delphi автоматически управляет файлами проектов, нет необходимости в самостоятельном внесении изменений в эти файлы. Более того, этого делать не рекомендуется во избежание потери целостности всего приложения.

Модули

Модули – это программные единицы, предназначенные для размещений фрагментов программ. Как уже было отмечено, для каждой формы, включаемой в проект, создается отдельный модуль (файл с исходным текстом, имеющий расширение *.PAS) Именно в этом файле хранится код, который вы пишете в процессе создания приложения, – объявления переменных, типов, код обработчиков сообщений для интерфейсных элементов, дополнительный код и т.п. В проект можно включать и модули, не связанные с формами.

Ниже показан код, создаваемый для формы в самом начале работы с новым проектом:

```
unit Unit1;

interface
// Секция интерфейсных объявлений

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

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
// Секция реализации
{$R *.dfm}
end.
```

В секции интерфейсных объявлений Interface описываются программные элементы (типы, классы, процедуры и функции), которые будут “видны” другим программным модулям, а в секции реализации Implementation раскрывается механизм работы этих элементов. Разделение модуля на две секции обеспечивает удобный механизм обмена алгоритмами между отдельными частями одной программы. Он также реализует средство обмена программными разработками между отдельными программистами. Получив откомпилированный “посторонний” модуль, программист получает доступ только к его интерфейсной части, в которой, как уже говорилось, содержатся объявления элементов. Детали реализации объявленных процедур, функций, классов скрыты в секции реализации и недоступны другим модулям.

Рассмотрим отдельные составные части модуля более подробно. В списке используемых модулей

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms,
  Dialogs;
```

перечислены основные модули, определяющие функциональность приложения создаваемого в среде Delphi. В секции описания типов описывается класс, соответствующий форме. Далее создается переменная, представляющая собой экземпляр класса:

```
var
  Form1: TForm1;
```

Директива компилятора $R используется для подключения двоичного образа формы (который сохраняется в файле с расширением *.DFM и имеет формат файла ресурсов Windows).

Файл форм

Форма сохраняется в виде Windows-ресурсов в файле с расширением *.DFM. Этот файл – бинарный, и он подключается непосредственно к исполняемому файлу в момент компиляции программы.

Файл формы содержит список свойств всех компонентов, включенных в форму, значения которых были изменены по сравнению со значениями, заданными по умолчанию (в конструкторе соответствующего объекта).

Файл формы имеет еще одно важное назначение. Он связывает графическое представление формы с обработчиками сообщений.

Файл опций проекта

Для каждого проекта создается файл опций (файл с расширением *.DOF), в который записываются значения опций компилятора, компоновщика и названия рабочих каталогов.

1.3. Управление проектами

При загрузке Delphi автоматически создается новый проект. Вы можете использовать этот проект для создания нового приложения или открыть уже существующий проект, либо использовать один из предоставляемых средой шаблонов в качестве основы для нового приложения. Для того чтобы создать новый проект или открыть уже существующий, сохранить проект или закрыть его, воспользуйтесь командами, распо-

ложенными в меню File: New, Open, Save, Save As, Save Project As, Save All, Close и Close All.

1.4. Компиляция, сборка и выполнение программ

Результатом компиляции всех Delphi-проектов является исполняемый файл. Это может быть либо программа (файл с расширением *.EXE), либо динамически загружаемая библиотека (файл с расширением *.DLL). Отметим, что ваша программа может быть откомпилирована и выполнена на любой стадии создания. Это бывает удобно для проверки работы интерфейсных элементов и правильности их взаимодействия, а также для проверки функциональности отдельных фрагментов создаваемого кода.

Компиляция проекта

Для компиляции исходных файлов, входящих в проект, используется команда Project | Compile главного меню интегрированной среды разработчика или комбинация клавиш Ctrl+F9. При этом выполняются следующие действия:

- компилируются файлы с исходным текстом всех модулей, содержимое которых изменялось после последней компиляции. В результате для каждого файла с исходным текстом модуля создается файл с расширением *.DCU. Если исходный текст модуля по каким-то причинам недоступен компилятору, то модуль не перекомпилируется;
- если были внесены изменения в интерфейсную часть модуля, то перекомпилируется не только этот модуль, но и модули, использующие его (через директиву uses);

Сборка проекта

При сборке проекта (в отличие от компиляции) компилируются все файлы, входящие в проект, вне зависимости от того, были в них внесены изменения после предыдущей компиляции или нет. Для сборки проекта используется команда Project | Build All главного меню интегрированной среды разработчика.

Выполнение программ

Для выполнения программы используется команда Run | Run главного меню интегрированной среды разработчика или клавиша F9.

При вызове этой команды происходят те же действия, что и при вызове команды Project | Compile, но после компиляции программа запускается на выполнение.

1.5. Основы визуального программирования

Программирование в Delphi строится на тесном взаимодействии двух процессов: процесса конструирования визуального проявления программы (т. е. ее Windows-окна) и процесса написания кода, придающего элементам этого окна и программе в целом необходимую функциональность. Для написания кода используется окно кода, для конструирования программы – остальные окна Delphi, и прежде всего – окно формы.

Между содержимым окон формы и кода существует неразрывная связь, которая строго отслеживается Delphi. Это означает, что размещение на форме компонента приводит к автоматическому изменению кода программы и наоборот – удаление тех или иных автоматически вставленных фрагментов кода может привести к удалению соответствующих компонентов. Помня об этом, программисты вначале конструируют форму, размещая на ней очередной компонент, а уже только после этого переходят, если это необходимо, к писанию фрагмента кода, обеспечивающего требуемое поведение компонента в работающей программе.

1.5.1. Пустая форма и ее модификация

Как уже говорилось, окно формы содержит проект Windows-окна программы. Важно помнить, что с самого начала работы над новой программой Delphi создает минимально необходимый код, обеспечивающий ее нормальное функционирование в Windows. Таким образом, простейшая программа готова сразу после выбора опции File | New | Application, и нам остается просто запустить программу. Однако до этого необходимо выполнить две важные вещи: создать собственный рабочий каталог (папку) и нужным образом настроить Delphi.

Настройка Delphi

В процессе работы над проектами программ, описываемых в этой книге, вам понадобится создать множество форм и модулей. Полезно сохранять эти данные в виде дисковых файлов в отдельной папке. Более того, для каждой программы в этой папке имеет смысл создать свою

вложенную папку. Тогда, чтобы освободить место на диске для серьезной программы, вам будет достаточно уничтожить основную папку, а чтобы передать ту или иную учебную программу своему коллеге – переписать на дискету содержимое соответствующей вложенной папки. Создайте папку с именем, например, my_delph, а в нем – вложенную папку для вашей первой программы.

Второе, что нам предстоит сделать, – это внести изменения в стандартную настройку среды Delphi. Это необходимо для того, чтобы среда автоматически сохраняла на диске последнюю версию создаваемой вами программы. Выберите опцию меню Tools | Environment options и убедитесь, что в появившемся диалоговом окне активна страница Preferences. В левом верхнем углу этой страницы в группе Autosave | Options есть переключатели Editor Files и Desktop. Активизация переключателей приведет к автоматическому сохранению текста окна кода программы и общего расположения окон Delphi перед началом очередного прогона создаваемой программы, что избавит вас от возможных потерь в случае “зависания” программы. Советую активизировать эти переключатели, щелкнув по каждому мышью. Чтобы следить за ходом компиляции, активизируйте также переключатель Show Compiler progress в группе Compiling and Running. Следует также отметить, что при написании достаточно больших и сложных программ полезно сохранять в коде программы комментарии – текстовые фрагменты, которые не влияют на работу программы, но делают ее текст более понятным. Для этого следует выбрать соответствующий шрифт для отображения кода программы. По умолчанию редактор Delphi использует шрифт Courier New, в котором может не быть символов кириллицы. В этом случае выберите опцию Tools | Editor options и на странице Display в строке Editor Font установите Courier New Cyr.

После установки необходимых настроек все готово для прогона вашей программы. Щелкните мышью по кнопке «  » в главном окне или, что проще, нажмите клавишу F9, именно таким способом подготовленная Delphi-программа последовательно проходит три главных этапа своего жизненного цикла – этапы компиляции, компоновки и исполнения. На этапе компиляции осуществляется преобразование подготовленного в окне кода текста программы на языке Object Pascal в последовательность машинных инструкций, на этапе компоновки к ней подключаются необходимые вспомогательные подпрограммы, а на этапе исполнения готовая программа загружается в оперативную память и ей передается исполнение.

Как только вы нажмете F9, появится диалоговое окно Save Unit1 As, в котором Delphi попросит вас указать имя файла для модуля Unit1.pas и папку его размещения. По умолчанию Delphi предлагает разместить файл модуля и проекта в системной папке BIN. Поскольку эта папка содержит жизненно важные для Delphi файлы, обязательно измените ее на вашу рабочую папку (например, MY_DELPH).

Если в окне Save Uniti As вы укажете имя Unit1.pas и нажмете Enter, Delphi, потребует задать имя еще и для проекта программы в целом. Под этим именем будет создан исполняемый EXE-файл.

Изменение свойств формы

После создания нашей программы путем выбора пункта меню File | New | Application попробуем модифицировать ее, например изменим заголовок окна. По умолчанию заголовок окна совпадает с заголовком формы: Form1. Чтобы изменить заголовок, нужно обратиться к окну инспектора объектов (Object Inspector).

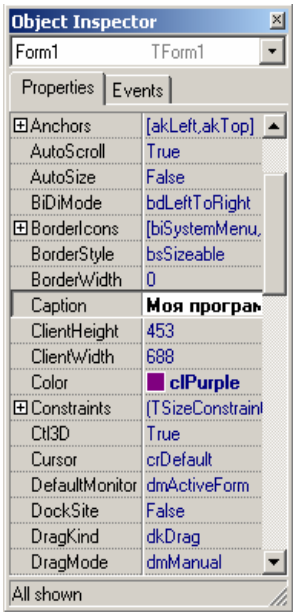


Рис. 1.3. Окно инспектора объектов

Для этого сделайте активным окно нашей программы и в закладке Properties (Свойства) Инспектора объектов (рис. 1.3) элементу Caption присвоить имя – заголовок формы, например “Моя программа”. Заголовок окна вашей программы теперь будет иметь введенное вами имя.

Итак, простым изменением содержимого строки в окне Инспектора объектов мы добились важной перемены: изменили одно из свойств окна программы – его заголовок. Таким же образом можно изменять

любое другое свойство формы, например цвет фона окна, тип курсора мыши, размер формы, имя формы, иконку формы и т.д.

1.5.2. Размещение нового компонента

Для размещения нового компонента на форме нужно сначала его выбрать (щелкнуть по нему мышью) в палитре компонентов, а затем щелкнуть мышью по точке рабочего пространства формы, где должен располагаться левый верхний угол компонента.

В качестве примера рассмотрим, как размещается компонент Label на рабочей форме, предназначенный для отображения различного рода надписей. Убедитесь в том, что в палитре компонентов (рис. 1.4) выбрана страница Standard, и щелкните мышью по кнопке «  ». Теперь щелкните мышью по форме так, чтобы компонент появился на форме и располагался левее и выше ее центра (рис. 1.5).

Рис. 1.4. Страница Standard палитры компонентов

Новый компонент имеет стандартное имя “Label1”, и надпись на нем повторяет это имя. Изменим эту надпись: с помощью строки Caption окна Инспектора объектов введите надпись: Я программирую на Delphi. Как только вы начнете вводить новую надпись, вид компонента на форме начнет меняться, динамически отражая все изменения, производимые вами в окне Инспектора объектов.

Выделим надпись цветом и сделаем ее шрифт более крупным. Для этого щелкните мышью по свойству Font окна Инспектора объектов и с помощью кнопки в правой части строки раскройте диалоговое окно настройки шрифта. В списке Size (Размер) этого окна выберите высоту шрифта 24 пункта, а с помощью списка Color (Цвет) выберите нужный цвет, после чего закройте окно кнопкой Ok. Надпись на компоненте в окне формы тут же соответствующим образом изменит свои свойства. Delphi обладает замечательной способностью визуальной реализации любых изменений свойств компонента не только на этапе прогона программы, но и на этапе проектирования формы.

Щелкните мышью внутри обрамляющих надпись черных прямоугольников и, не отпуская левую кнопку мыши, сместите ее указатель так, чтобы он расположился левее в центре окна, после чего отпустите

кнопку. Таким способом можно буксировать компонент по форме, добиваясь нужного его положения.

С помощью обрамляющих черных квадратиков можно изменять размеры компонента. Для этого следует поместить острие указателя

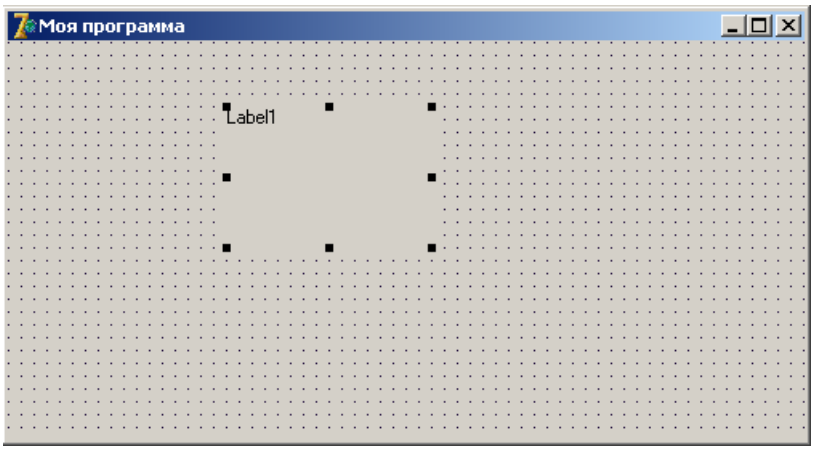


Рис. 1.5. Размещения компонента Label1 на форме

мыши над одним из них (в этот момент указатель меняет свою форму на двунаправленную стрелку), затем нажать левую кнопку мыши и, не отпуская ее, буксировать сторону или угол компонента в нужном направлении, после чего отпустить кнопку.

Отметим, что все видимые компоненты имеют свойства Left (Слева), Top (Сверху), Width (Ширина) и Height (Высота), числовые значения которых определяют положение левого верхнего угла компонента и его размеры в так называемых пикселях, т. е. в минимальных по размеру точках экрана, светимостью которых может управлять программа. При буксировании компонента или изменении его размеров мышью эти значения автоматически меняются, и наоборот – изменение этих свойств в окне Инспектора объектов приводит к соответствующему изменению положения и размеров компонента. В Delphi-4, 5 и 6 значения Left и Top автоматически появляются в небольшом окне рядом с указателем мыши при буксировке компонента по форме.

1.5.3. Реакция на события

Как уже говорилось, функциональность программы определяется совокупностью ее реакций на те или иные события. В связи с этим каждый компонент помимо свойств характеризуется также набором событий, на которые он может реагировать.

В качестве примера реакции на события рассмотрим работу нашей программы при наличии на ее форме компонента Button (кнопка). Для этого проведем очередную модернизацию нашей первой программы: вставим в ее форму еще один компонент – кнопку, и заставим ее откликаться на событие, связанное с нажатием левой кнопки мыши.

Компонент “кнопка” изображается пиктограммой «  » на странице Standard палитры компонентов. Поместите этот компонент на форму и расположите его ниже компонента Label и посередине формы (рис. 1.6).

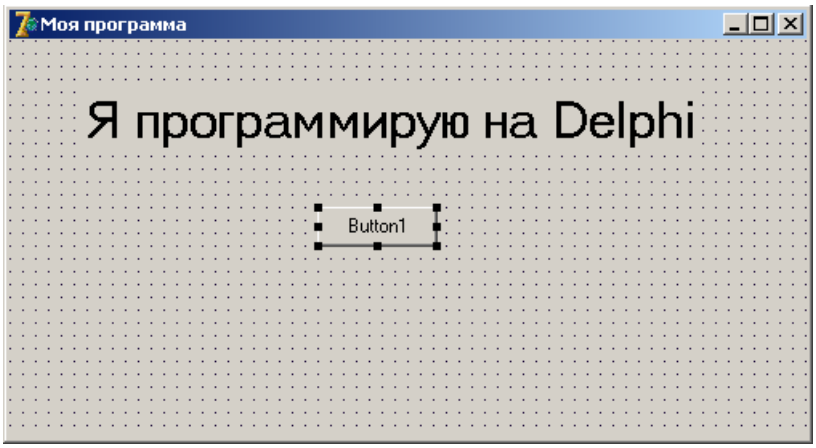


Рис. 1.6. Форма с вставленной кнопкой

Обработчик события OnClick

При щелчке по кнопке мышью в работающей программе возникает событие OnClick. (По щелчку). Пока это событие никак не обрабатывается программой, и поэтому нажатие кнопки не приведет ни к каким последствиям. Чтобы заставить программу реагировать на нажатие кнопки, необходимо написать на языке Object Pascal фрагмент программы, который называется обработчиком события.

Этот фрагмент должен представлять собой последовательность текстовых строк, в которых программист указывает, что именно должна делать программа в ответ на нажатие кнопки. Фрагмент оформляется в виде специальной подпрограммы языка Object Pascal – процедуры.

Чтобы заставить Delphi самостоятельно сделать заготовку для процедуры обработчика события OnClick, дважды подряд без заметной паузы щелкните мышью по вновь вставленному компоненту. В ответ Delphi активизирует окно кода, и вы увидите в нем такой текстовый фрагмент:

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

end;
```

В приведенном выше коде слово procedure извещает компилятор о начале подпрограммы-процедуры (в Delphi могут использоваться также подпрограммы-функции; в этом случае вместо procedure (процедура) используется слово function (функция)). За ним следует имя процедуры TForm1.Button1Click. Это имя – составное: оно состоит из имени класса TForm1 и собственно имени процедуры Button1Click.

Классами в Delphi называются функционально законченные фрагменты программ, служащие образцами для создания подобных себе экземпляров. Однажды, создав класс, программист может включать его экземпляры (копии) в разные программы или в разные места одной и той же программы. Такой подход способствует максимально высокой продуктивности программирования за счет использования ранее написанных фрагментов программ. В состав Delphi входит несколько сотен классов, созданных программистами корпорации Borland (так называемых стандартных классов). Совокупность стандартных классов определяет мощные возможности этой системы программирования.

Каждый компонент принадлежит к строго определенному классу, а все конкретные экземпляры компонентов, вставляемые в форму, получают имя класса с добавленным числовым индексом. По используемому в Delphi соглашению все имена классов начинаются с буквы T. Таким образом, имя TForm1 означает имя класса, созданного по образцу стандартного класса TForm. Если вы посмотрите начало текста в окне кода, то увидите следующие строки:

```
type
  TForm1 = class(TForm)
    Label1: TLabel;
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
```

Строка

```
TForm1 = class(TForm)
```

определяет новый класс TForm1, который порожден (создан по образцу) от стандартного класса TForm. Строка

```
Form1: TForm1;
```

создает экземпляр этого класса с именем Form1. Стандартный класс TForm описывает пустое Windows-окно, в то время как класс TForm1 описывает окно с уже вставленными в него компонентами “метка” и “кнопка”. Описание этих компонентов содержат строки

```
Label1: TLabel;
Button1: TButton;
```

Они указывают, что компонент Button1 представляет собой экземпляр стандартного класса TButton, а компонент Label1 – экземпляр класса TLabel.
За именем процедуры Button1Click в круглых скобках следует описание параметра вызова:

```
Sender: TObject
```

(параметр с именем Sender принадлежит классу TObject). Как мы увидим дальше, процедуры могут иметь не один, а несколько параметров вызова или не иметь их вовсе. Параметры вызова (если они есть) служат для настройки реализованного в процедуре алгоритма на выполнение конкретной работы. Параметр Sender вставлен Delphi “на всякий случай”: с его помощью подпрограмма Button1Click может при желании определить, какой именно компонент создал событие OnClick. Вся строка в целом

```
procedure TForm1.Button1Click(Sender: TObject);
```

называется заголовком процедуры. Ее завершает символ “;”. Этот символ играет важную роль в Object Pascal, т.к. показывает компилятору на конец предложения языка. Из отдельных предложений составляется весь текст программы. В конце каждого предложения нужно ставить точку с запятой – это обязательное требование синтаксиса языка. Три следующие строки определяют тело процедуры:

```
begin

end;
```

Слово begin (начало) сигнализирует компилятору о начале последовательности предложений, описывающих алгоритм работы процедуры, а слово end (конец) – о конце этой последовательности. В нашем случае тело процедуры пока еще не содержит описания каких-либо действий, что и неудивительно: Delphi лишь создала заготовку для процедуры, но она ничего “не знает” о том, для чего эта процедура предназначена. Наполнить тело нужными предложениями – задача программиста.

Для нас важно то обстоятельство, что каждый раз при нажатии кнопки Button1 управление будет передаваться в тело процедуры, а значит, между словами begin и end мы можем написать фрагмент программы, который будет выполняться в ответ на это событие. Чтобы убедиться в этом, добавим между словами begin и end следующую строку:

```
ShowMessage('Вы нажали кнопку Button1');
```

Теперь в ответ на нажатие кнопки Button1 на экране появится сообщение “Вы нажали кнопку Button1” (рис. 1.7).

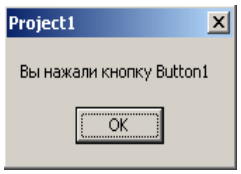


Рис. 1.7. Информационное сообщение Windows-приложения

Динамическое изменение свойств компонента

Наиболее простой пример динамического изменения свойств компонента – это изменение свойства Caption кнопки Button1. Проще всего это сделать с помощью окна формы и Инспектора объектов. Для этого создадим обработчик события OnCreate (По созданию) для формы и изменим в нем это свойство. Событие OnCreate возникает после создания windows-окна, но до появления этого окна на экране. Чтобы создать обработчик этого события, раскройте список компонентов в верхней части окна Инспектора объектов, выберите компонент Form1 и дважды щелкните по свойству OnCreate на странице Events этого компонента (щелкать нужно по правой части строки Oncreate). В ответ Delphi вновь активизирует окно кода и покажет вам заготовку для процедуры TForm1.FormCreate. Отредактируйте ее следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Button1.Caption:='Кнопка';
end;
```

Единственная вставленная нами строка представляет собой так называемый оператор присваивания языка Object Pascal. В левой части оператора указывается свойство Button1.Caption, а в правой части – значение 'Кнопка ', которое мы хотим придать этому свойству. Связывает обе части комбинация символов “: =”, которая читается как “присвоить значение”. Символы “:=” всегда пишутся слитно, без разделяющих пробелов, хотя перед двоеточием и после знака равенства можно для лучшей читаемости программы вставлять пробелы. Как и любое другое предложение языка, оператор присваивания завершается точкой с запятой.

Составное имя Button1.Caption необходимо для точного указания компилятору, о каком свойстве идет речь: в нашей программе используются три компонента (включая саму форму), каждый из которых имеет свойство Caption; уточняющий префикс Button1 заставит изменить это свойство у кнопки, а не у метки или формы. Присваиваемое свойству значение является текстовой строкой. По правилам Object Pascal текстовая строка должна заключаться в обрамляющие апострофы. Внутри апострофов можно написать любое количество произвольных символов – именно они (без обрамляющих апострофов) будут определять новую надпись на кнопке.

После очередного прогона программы вы увидите измененную надпись на кнопке, а мы сделаем важный вывод: любое свойство любого компонента можно изменять динамически, т. е. в ходе исполнения программы.

1.6. Знакомство с компонентами

Компоненты представляют собой элементы, из которых конструируется видимое изображение, создаваемое работающей программой. При этом следует заметить, что существует значительное количество компонентов, которые не создают видимого изображения, но которые, тем не менее, играют важную роль в тех или иных случаях. Правильнее думать о компонентах как о заранее приготовленных для вас фрагментах программы, которые можно вставлять, если в этом есть необходимость, в разрабатываемую программу. В этом разделе приводится на-

чальный обзор компонентов, который даст вам самое общее представление о богатстве возможностей Delphi.

Библиотеки компонентов для разных версий Delphi строятся по принципу расширения: в первой версии было около 70 компонентов, в то время как в состав Delphi 6 входит более 300 компонентов. Ниже перечисляются лишь функциональные группы (страницы палитры) компонентов и описываются те из них, которые наиболее часто используются при разработке Delphi-приложений.

1.6.1. Страница Standard

На странице Standard палитры компонентов сосредоточены стандартные для Windows интерфейсные элементы, без которых не обходится практически ни одна программа.

Frame – рама. Наравне с формой служит контейнером для размещения других компонентов. В отличие от формы может размещаться в палитре компонентов, создавая заготовки компонентов.

MainMenu – главное меню программы. Компонент способен создавать и обслуживать сложные иерархические меню.

PopupMenu – вспомогательное или локальное меню. Обычно это меню появляется в отдельном окне после нажатия правой кнопки мыши.

Label – метка. Этот компонент используется для размещения в окне не очень длинных однострочных надписей.

Рис. 1.8. Стандартные компоненты

Edit – строка ввода. Предназначена для ввода, отображения или редактирования одной текстовой строки.

Memo – многострочный текстовый редактор. Используется для ввода и/или отображения многострочного текста.

Button – командная кнопка. Обработчик события OnClick этого компонента обычно используется для реализации некоторой команды.

CheckBox – независимый переключатель. Щелчок мышью на этом компоненте в работающей программе изменяет его логическое свойство Checked.

RadioButton – зависимый переключатель. Обычно объединяется, как минимум, еще с одним таким же компонентом в группу. Щелчок по

переключателю приводит к автоматическому освобождению ранее выбранного переключателя в той же группе.

ListBox – список выбора. Содержит список предлагаемых вариантов (опций) и дает возможность проконтролировать текущий выбор.

ComboBox – комбинированный список выбора. Представляет собой комбинацию списка выбора и текстового редактора.

ScrollBar – полоса управления. Представляет собой вертикальную или горизонтальную полосу, напоминающую полосы прокрутки по бокам Windows-окна.

GroupBox – группа элементов. Этот компонент используется для группировки нескольких связанных по смыслу компонентов.

RadioGroup – группа зависимых переключателей. Содержит специальные свойства для обслуживания нескольких связанных зависимых переключателей.

Panel – панель. Этот компонент, как и GroupBox, служит для объединения нескольких компонентов. Содержит внутреннюю и внешнюю кромки, что позволяет создать эффекты “вдавленности” и “выпуклости”.

Actiontist – список действий. Служит для централизованной реакции программы на действия пользователя, связанные с выбором одного из группы однотипных управляющих элементов, таких как опции меню, пиктографические кнопки и т. п.

1.6.2. Страница Additional

В страницы Additional размещены 18 дополнительных компонентов, с помощью которых можно разнообразить вид диалоговых окон.

Рис. 1.9. Дополнительные компоненты

BitBtn – командная кнопка с надписью и пиктограммой.

SpeedButton – пиктографическая кнопка. Обычно используется для быстрого доступа к тем или иным опциям главного меню.

MaskEdit – специальный текстовый редактор. Способен фильтровать вводимый текст, например для правильного ввода даты.

StringGrid – таблица строк. Этот компонент обладает мощными возможностями для представления текстовой информации в табличном виде.

DrawGrid – произвольная таблица. В отличие от StringGrid ячейки этого компонента могут содержать произвольную информацию, в том числе и рисунки.

Image – рисунок. Этот компонент предназначен для отображения рисунков, в том числе пиктограмм и метафайлов.

Shape – фигура. С помощью этого компонента вы можете вставить в окно правильную геометрическую фигуру – прямоугольник, эллипс, окружность и т. п.

Bevel – кромка. Служит для выделения отдельных частей окна трехмерными рамками или полосами.

ScrollBox – панель с полосами прокрутки. В отличие от компонента Panel автоматически вставляет полосы прокрутки, если размещенные в нем компоненты отсекаются его границами.

CheckListBox – список множественного выбора. Отличается от стандартного компонента ListBox наличием рядом с каждой опцией независимого переключателя типа CheckBox, облегчающего выбор сразу нескольких опций.

Splitter – граница. Этот компонент размещается на форме между двумя другими видимыми компонентами и дает возможность пользователю во время прогона программы перемещать границу, отделяющую компоненты друг от друга.

StaticText – статический текст. Отличается от стандартного компонента Label наличием собственного windows-окна, что позволяет обводить текст рамкой или выделять его в виде “вдавленной” части формы.

ControiBar – полоса управления. Служит контейнером для “причаливаемых” компонентов в технологии Drag&Dock.

ApplicationEvents – получатель события. Если этот компонент помещен на форму, он будет получать все предназначенные для программы сообщения Windows (без этого компонента сообщения принимает глобальный объект-программа Application).

ValueListEditor – редактор строк, содержащих пары имя-значение. Пары такого типа широко используются в Windows, например в файлах инициации, в системном реестре и т. п.

LabeledEdit – комбинация однострочного редактора и метки.

ColorBox – специальный вариант ComboBox для выбора одного из системных цветов.

Chart – диаграмма. Этот компонент облегчает создание специальных панелей для графического представления данных.

ActionManager – менеджер действий. Совместно с тремя следующими компонентами обеспечивает создание приложений, интерфейс

которых (главное меню и инструментальные кнопки) может настраиваться пользователем.

ActionMainMenuBar – полоса меню, опции которого создаются с помощью компонента ActionManager.

ActionToolBar – полоса для размещения пиктографических кнопок, создаваемых с помощью компонента ActionManager.

CustomizeDig – диалог настройки. С помощью этого компонента пользователь может по своему вкусу настроить интерфейс работающей программы.

1.6.3. Страница Win32

Страница Win32 содержит интерфейсные элементы для 32-разрядных операционных систем Windows 95/98/NT/2000/XP.

Рис. 1.10. Компоненты Win32

TabControl – набор закладок. Каждая закладка представляет собой прямоугольное поле с надписью и/или рисунком. Выбор той или иной закладки распознается программой и используется для управления содержимым окна компонента.

PageControl – набор панелей с закладками. Каждая панель может содержать свой набор интерфейсных элементов и выбирается щелчком по связанной с ней закладке.

ImageList – набор рисунков. Представляет собой хранилище для нескольких рисунков одинакового размера.

RichEdit – многострочный редактор форматированного текста. В отличие от компонента Memo страницы Standard текст в компоненте RichEdit подчиняется правилам Расширенного Текстового Формата (RTF – Rich Text Format) и может изменять такие свои характеристики, как шрифт, цвет, выравнивание и т. д.

TrackBar – регулятор. Используется для управления значениями некоторых величин в программах. Например, с его помощью удобно изменять громкость звучания в мультимедийных программах.

ProgressBar – индикатор процесса. С помощью этого компонента можно отображать ход исполнения достаточно длительного по времени процесса, например процесса переноса данных на дискету.

UpDown – цифровой регулятор. Две кнопки этого компонента предназначены для увеличения (верхняя) или уменьшения (нижняя) связанной с компонентом числовой величины.

HotKey – управляющая клавиша. Компонент используется для ввода управляющих клавиш, таких как F1, Alt+A, Ctrl+Shift+1 и т. п.

Animate – мультипликатор. Предназначен для отображения последовательно сменяющих друг друга кадров движущихся изображений (видеоклипов). Компонент не может сопровождать видеоклип звуком.

DateTimePicker – селектор времени/даты. Этот компонент предназначен для ввода и отображения даты или времени.

TreeView – дерево выбора. Представляет собой совокупность связанных в древовидную структуру пиктограмм. Обычно используется для просмотра структуры каталогов (папок) и других подобных элементов, связанных иерархическими отношениями.

ListView – панель пиктограмм. Организует просмотр нескольких пиктограмм и выбор нужной. Этот компонент способен располагать пиктограммы в горизонтальных или вертикальных рядах и показывать их в крупном или мелком масштабе.

HeaderControl – управляющий заголовок. Представляет собой горизонтальную или вертикальную полосу, разделенную на ряд смежных секций с надписями. Размеры секций можно менять мышью на этапе работы программы. Обычно используется для изменения размеров столбцов или строк в разного рода таблицах.

StatusBar – панель статуса. Предназначена для размещения разного рода служебной информации в окнах редактирования. Посмотрите на нижнюю часть рамки окна кода Delphi или текстового редактора Word, и вы увидите этот компонент в действии.

ToolBar – инструментальная панель. Этот компонент служит контейнером для командных кнопок BitBtn и способен автоматически изменять их размеры и положение при удалении кнопок или при добавлении новых.

CoolBar – инструментальная панель. В отличие от ToolBar используется как контейнер для размещения стандартных интерфейсных компонентов Windows, таких как Edit, ListBox, ComdoBox и т. д.

PageScroller – прокручиваемая панель. Служит для размещения узких инструментальных панелей. При необходимости автоматически создает по краям панели стрелки прокрутки.

ComboBoxEx – компонент в функциональном отношении подобен comboBox (страница standard), но может отображать в выпадающем списке небольшие изображения.

1.6.4. Страница System

На странице System представлены компоненты, которые имеют различное функциональное назначение, в том числе компоненты, поддерживающие стандартные для Windows технологии межпрограммного обмена данными OLE (Object Linking and Embedding – связывание и внедрение объектов) и DDE (Dynamic Data Exchange – динамический обмен данными). Технология DDE в современных программах полностью вытеснена гораздо более мощной технологией COM, и поэтому соответствующие им компоненты не рассматриваются.

Timer – таймер. Этот компонент служит для отсчета интервалов реального времени.

PaintBox – окно для рисования. Создает прямоугольную область, предназначенную для прорисовки графических изображений.

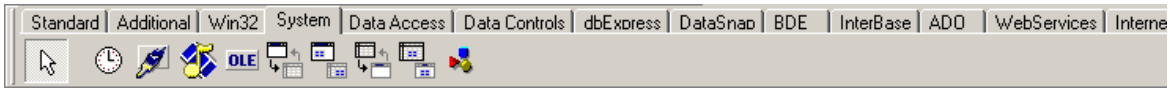


Рис. 1.11. Компоненты страницы System

MediaPlayer – мультимедийный проигрыватель. С помощью этого компонента можно управлять различными мультимедийными устройствами.

OleContainer – OLE-контейнер. Служит приемником связываемых или внедряемых объектов.

Компоненты этой страницы имеются во всех предыдущих версиях Delphi.

1.6.5. Страница Dialogs

Компоненты страницы Dialogs реализуют стандартные для Windows диалоговые окна.

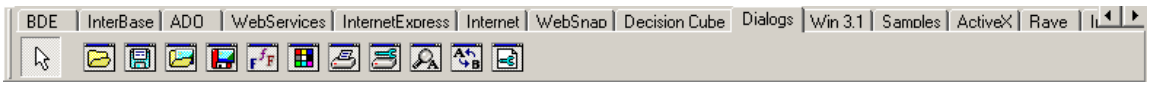


Рис. 1.12. Компоненты страницы Dialogs

OpenDialog – открыть. Реализует стандартное диалоговое окно “Открыть файл”.

SaveDialog – сохранить. Реализует стандартное диалоговое окно “Сохранить файл”.

OpenPictureDialog – открыть рисунок. Реализует специальное окно выбора графических файлов с возможностью предварительного просмотра рисунков.

SavePictureDialog – сохранить рисунок. Реализует специальное окно сохранения графических файлов с возможностью предварительного просмотра рисунков.

FontDialog – шрифт. Реализует стандартное диалоговое окно выбора шрифта.

ColorDialog – цвет. Реализует стандартное диалоговое окно выбора цвета.

PrintDialog – печать. Реализует стандартное диалоговое окно выбора параметров для печати документа.

PrinterSetupDialog – настройка принтера. Реализует стандартное диалоговое окно для настройки печатающего устройства.

FindDialog – поиск. Реализует стандартное диалоговое окно поиска текстового фрагмента.

ReplaceDialog – замена. Реализует стандартное диалоговое окно поиска и замены текстового фрагмента.

1.6.6. Страница Win3.1

Большинство компонентов страницы Win3.1 введены для совместимости с версией 1. В современных программах вместо них рекомендуется использовать соответствующие компоненты страницы Win32.

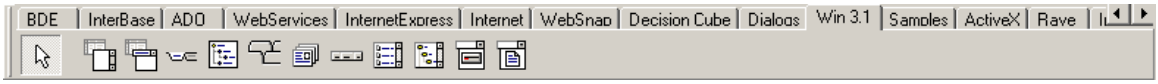


Рис. 1.13. Компоненты страницы Win3.1

TabSet – набор закладок. В приложениях для Windows 32 вместо него рекомендуется использовать компонент TabControl.

OutLine – дерево выбора. В приложениях для Windows 32 вместо него рекомендуется использовать компонент Treeview.

TabbedNotebook – набор панелей с закладками. В приложениях для Windows 32 вместо него рекомендуется использовать компонент PageControl.

Notebook – набор панелей без закладок. В приложениях для Windows 32 вместо него рекомендуется использовать компонент PageControl.

Header – управляющий заголовок. В приложениях для Windows 32 вместо него рекомендуется использовать компонент Header-Control.

FileListBox – панель выбора файлов.

DirectorybistBox – панель выбора каталогов.

DiriveComboBox – панель выбора дисков.

FilterComboBox – панель фильтрации файлов.

Компоненты FileListBox, DirectoryListBox, DriveComboBox и FilterComboBox впервые появились в версии 3. Их функции реализованы элементами стандартных окон OpenDialog и SaveDialog, которые и рекомендуется использовать в Windows 32.

1.6.7. Страница Samples

Страница Samples содержит компоненты разного назначения.

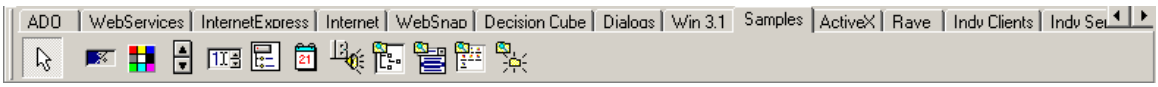


Рис. 1.14. Компоненты страницы Samples

Gauge – индикатор состояния. Подобен компоненту ProgressBar (страница Win32), но отличается большим разнообразием форм.

ColorGrid – таблица цветов. Этот компонент предназначен для выбора основного и фонового цветов из 16-цветной палитры.

SpinButton – двойная кнопка. Дает удобное средство управления некоторой числовой величиной.

SpinEdit – редактор числа. Обеспечивает отображение и редактирование целого числа с возможностью его изменения с помощью двойной кнопки.

DirectoryOutLine – список каталогов. Отображает в иерархическом виде структуру каталогов дискового накопителя.

Calendar – календарь. Предназначен для показа и выбора дня в месяце.

1.6.8. Страница ActiveX

Компоненты ActiveX являются “чужими” для Delphi: они создаются другими инструментальными средствами разработки программ (например, C++ или Visual Basic) и внедряются в Delphi с помощью технологии OLE. На странице ActiveX представлены лишь 4 из велико-

го множества ActiveX-компонентов, разрабатываемых повсюду в мире компаниями-производителями программных средств и отдельными программистами.

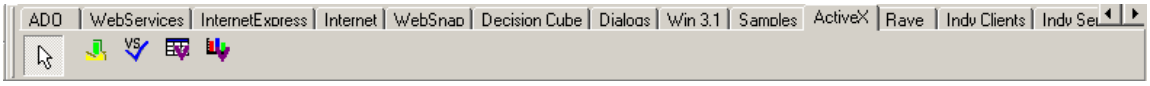


Рис. 1.15. Компоненты страницы ActiveX

Chartfx – интерактивный график. Дает программисту удобное средство включения в программу интерактивных (диалоговых) графиков.

VSSpell – спеллер. Осуществляет орфографическую проверку правильности написания английских слов.

F1Book – электронная таблица. Позволяет создавать и использовать рабочие книги электронных таблиц, подобно тому как это делает MS Excel.

VtChart – мастер диаграмм. Обеспечивает мощные средства построения двух- и трехмерных диаграмм по результатам табличных вычислений.

2. ЯЗЫК OBJECT PASCAL

2.1. Алфавит

Алфавит языка Object Pascal включает буквы, цифры, шестнадцатеричные цифры, специальные символы, пробелы и зарезервированные слова.

Буквы – это буквы латинского алфавита от a до z и от A до Z, а также знак подчеркивания "_". В языке нет различия между заглавными и строчными буквами алфавита, если только они не входят в символьные и строковые выражения.

Цифры – арабские цифры от 0 до 9. Каждая шестнадцатеричная цифра имеет значение от 0 до 15. Первые 10 значений обозначаются арабскими цифрами 0... 9, остальные шесть – латинскими буквами a ... f или A ... F.

Специальные символы Object Pascal – это символы :
+ - * / = , ' . : ; < > [ ] ( ) { } " @ $ #.
К специальным символам относятся также следующие пары символов:
< > , < = , > = , : = , (* , *) , (. , .) , / / .
В программе эти пары символов нельзя разделять пробелами, если они используются как знаки операций отношения или ограничители комментария.

Особое место в алфавите языка занимают пробелы, к которым относятся любые символы в диапазоне кодов от 0 до 32. Эти символы рассматриваются как ограничители идентификаторов, констант, чисел, зарезервированных слов. Несколько следующих друг за другом пробелов считаются одним пробелом (последнее не относится к строковым константам).

2.2. Элементы программы

Элементы программы – это минимальные неделимые ее части, несущие в себе определенную значимость для компилятора. К элементам относятся:

- зарезервированные слова;
- идентификаторы;
- типы;

- константы;
- переменные;
- метки;
- подпрограммы;
- комментарии.

Зарезервированные слова – это английские слова, указывающие компилятору на необходимость выполнения определенных действий. Зарезервированные слова не могут использоваться в программе ни для каких иных целей, кроме тех, для которых они предназначены. Например, зарезервированное слово begin означает для компилятора начало составного оператора. Программист не может создать в программе переменную с именем begin, константу begin, метку begin или вообще какой бы то ни было другой элемент программы с именем begin.

Список зарезервированных слов в Object Pascal приведен в таблице 2.1.

Таблица 2.1

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

Идентификаторы – это слова, которыми программист обозначает любой другой элемент программы, кроме зарезервированного слова, идентификатора или комментария. Идентификаторы в Object Pascal могут состоять из латинских букв, арабских цифр и знака подчеркивания. Никакие другие символы или специальные знаки не могут входить в идентификатор. Из этого простого правила следует, что идентификато-

ры не могут состоять из нескольких слов (нельзя использовать пробел) или включать в себя символы кириллицы (русского алфавита).

Типы – это специальные конструкции языка, которые рассматриваются компилятором как образцы для создания других элементов программы, таких как переменные, константы и функции. Любой тип определяет две важные для компилятора вещи: объем памяти, выделяемый для размещения элемента (константы, переменной или результата, возвращаемого функцией), и набор допустимых действий, которые программист может совершать над элементами данного типа. Любой определяемый программистом идентификатор должен быть описан в разделе описаний (перед началом исполняемых операторов). Это означает, что компилятор должен знать тот тип (образец), по которому создается определяемый идентификатором элемент.

Константы определяют области памяти, которые не могут изменять своего значения в ходе работы программы. Как и любые другие элементы программы, константы могут иметь свои собственные имена. Объявлению имен констант должно предшествовать зарезервированное слово const (от англ. constants – константы). Например, мы можем определить константы

```
Const
Kbyte = 1024;
Mbyte = Kbyte*Kbyte;
Gbyte = 1024*Mbyte;
```

Тип константы определяется способом ее записи и легко распознается компилятором в тексте программы, поэтому программист может не использовать именованные константы (т. е. не объявлять их в программе явно).

Переменные связаны с изменяемыми областями памяти, т. е. с такими ее участками, содержимое которых будет меняться в ходе работы программы. В отличие от констант переменные всегда объявляются в программе. Для этого после идентификатора переменной ставится двоеточие и имя типа, по образу которого должна строиться переменная. Разделу объявления переменной (переменных) должно предшествовать слово var. Например:

```
var
inValue: Integer;
```

```
byValue: Byte;
```

Идентификатор inValue объявляется как переменная типа integer, а идентификатор byValue – как переменная типа Byte. Стандартный (т. е. заранее определенный в Object Pascal) тип integer определяет четырехбайтный участок памяти, содержимое которого рассматривается как целое число в диапазоне от –2 147 483 648 до +2 147 483 647, а стандартный тип Byte – участок памяти длиной 1 байт, в котором размещается беззнаковое целое число в диапазоне от 0 до 255[4].

Метки – это имена операторов программы. Метки используются очень редко и только для того, чтобы программист смог указать компилятору, какой оператор программы должен выполнятся следующим. Метки, как и переменные, всегда объявляются в программе. Разделу объявлений меток предшествует зарезервированное слово label. Например:

```
label
m1;
begin
Goto m1;
// Программист требует передать управление
// оператору, помеченному меткой m1. .....
// Эти операторы будут пропущены
m1:
// Оператору, идущему за этой меткой,
.....
// будет передано управление
end;
```

Подпрограммы – это специальным образом оформленные фрагменты программы. Особенностью подпрограмм является их значительная независимость от остального текста программы. Говорят, что свойства подпрограммы локализуются в ее теле. Это означает, что, если программист что-либо изменит в подпрограмме, ему, как правило, не понадобится в связи с этим изменять что-либо вне подпрограммы. Таким образом, подпрограммы являются средством структурирования программ, т. е. расчленения программ на ряд во многом независимых фрагментов. Структурирование неизбежно для крупных программных проектов, поэтому подпрограммы используются в Delphi-программах очень часто.

В Object Pascal есть два вида подпрограмм: процедуры и функции. Функция отличается от процедуры только тем, что ее идентификатор

можно наряду с константами и переменными использовать в выражениях, т. к. функция имеет выходной результат определенного типа.

Рассмотрим пример реализации функции:

```
Function MyFunction: Integer;
var x: Integer;
begin
x:= 2*MyFunction-1;
end;
```

2.3. Выражения и операции

Основными элементами, из которых конструируется исполняемая часть программы, являются константы, переменные и обращения к функциям. Каждый из этих элементов характеризуется своим значением и принадлежит к какому-либо типу данных. С помощью знаков операций и скобок из них можно составлять выражения, которые фактически представляют собой правила получения новых значений.

Частным случаем выражения может быть просто одиночный элемент, т. е. константа, переменная или обращение к функции. Значение такого выражения имеет, естественно, тот же тип, что и сам элемент. В общем случае выражение состоит из нескольких элементов (операндов) и знаков операций, а тип его значения определяется типом операндов и видом примененных к ним операций.

Примеры выражений:

```
(a + b) * c
sin(t)
a > 2
```

В Object Pascal определены следующие операции:

- унарные not, @ ;
- мультипликативные *, /, div, mod, and, shi, shr;
- аддитивные +, -, or, xor;
- отношения =, <>, <, >, <=, >=, in.

Приоритет операций убывает в указанном порядке, т. е. наивысшим приоритетом обладают унарные операции, низшим – операции отношения. Порядок выполнения нескольких операций равного приоритета устанавливается компилятором из условия оптимизации кода программы и не обязательно слева направо. При исчислении логических

выражений операции равного приоритета всегда вычисляются слева направо.

Правила использования операций с операндами различного типа приводятся в таблице 2.2.

Таблица 2.2

Операция	Действие	Тип операндов	Тип
not	Отрицание	Логический	Логический
not	То же	Любой целый	Тип операнда
@	Адрес	Любой	Указатель
*	Умножение	Любой целый	Наименьший целый
*	То же	Любой вещественный	Extended
*	Пересечение множеств	Множественный	Множественный
/	Деление	Любой вещественный	Extended
div	Целочисленное деление	Любой целый	Наименьший целый
mod	Остаток от деления	То же	То же
and	Логическое И	Логический	Логический
and	То же	Любой целый	Наименьший целый
shl	Левый сдвиг	То же	То же
shr	Правый сдвиг	То же	То же
+	Сложение	То же	То же
+	То же	Любой вещественный	Extended
+	Объединение множеств	Множественный	Множественный
+	Сцепление строк	Строковый	Строковый
-	Вычитание	Любой целый	Наименьший целый
-	То же	Любой вещественный	Extenden
or	Логическое или	Логический	Логический
or	То же	Любой целый	Наименьший целый
=	Равно	Любой простой или строковый	Логический
0	Не равно	То же	То же
<	Меньше	Логический	Логический
<=	Меньше или равно	То же	То же
>	Больше	То же	То же
>=	Больше или равно	То же	То же

Логические операции применимы к операндам целого и логического типов. Если операнды – целые числа, то результат логической операции есть тоже целое число, биты которого (двоичные разряды) формируются из битов операндов по правилам, указанным в табл. 2.3.

Таблица 2.3

Операнд 1	Операнд 2	not	and	or	Xor
1	–	0	–	–	–
0	–	1	–	–	–
0	0	–	0	0	0
0	1	–	0	1	1
1	0	–	0	1	1
1	1	–	1	1	0

Также к логическим операциям в Object Pascal обычно относятся и две сдвиговые операции над целыми числами:

i shl j – сдвиг содержимого i на j разрядов влево; освободившиеся младшие разряды заполняются нулями;

i shr j – сдвиг содержимого i на j разрядов вправо; освободившиеся старшие разряды заполняются нулями.

В этих операциях i и y – выражения любого целого типа.

2.4. Типы данных

Любые данные (константы, переменные, свойства, значения функций или выражения) в Object Pascal характеризуются своими типами. Тип определяет множество допустимых значений, которые может иметь тот или иной объект, а также множество допустимых операций, которые применимы к нему. Кроме того, тип определяет также и формат внутреннего представления данных в памяти ПК.

Object Pascal характеризуется разветвленной структурой типов данных (рис. 2.1). В языке предусмотрен механизм создания новых типов, благодаря чему общее количество используемых в программе типов может быть сколь угодно большим.

2.4.1. Порядковые типы

К порядковым типам относятся целые, логические, символьный, перечисляемый и тип-диапазон. К любому из них применима функция Ord(x), которая возвращает порядковый номер значения выражения X.

Для целых типов функция ord(x) возвращает само значение x, т. е. Ord(X) = x для x, принадлежащего любому целому типу. Применение Ord(x) к логическому, символьному и перечисляемому типам дает положительное целое число в диапазоне от 0 до 1 (логический тип), от 0 до 255 (символьный), от 0 до 65535 (перечисляемый). Тип-диапазон сохраняет все свойства базового порядкового типа, поэтому результат применения к нему функции ord(X) зависит от свойств этого типа.

К порядковым типам можно также применять функции:

pred(x) – возвращает предыдущее значение порядкового типа (значение, которое соответствует порядковому номеру

succ(x) – возвращает следующее значение порядкового типа, которое соответствует порядковому номеру ord (x) +1.

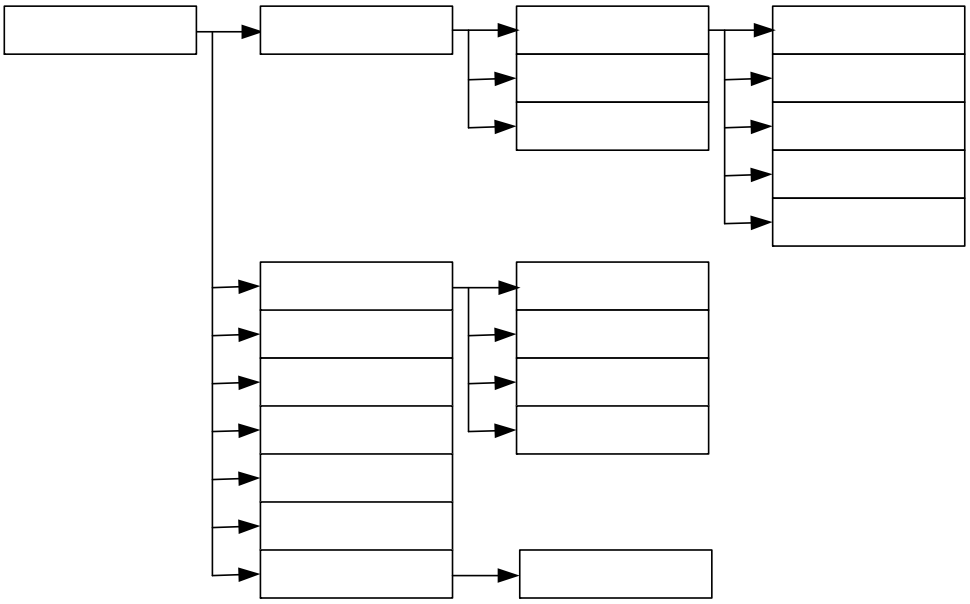


Рис. 2.1. Структура типов данных

Целые типы

Диапазон возможных значений целых типов зависит от их внутреннего представления, которое может занимать один, два, четыре или восемь байтов. В табл. 2.4 приводятся названия целых типов, длина их внутреннего представления в байтах и диапазон возможных значений.

При использовании процедур и функций с целочисленными параметрами следует руководствоваться “вложенностью” типов, т. е. везде, где может использоваться word, допускается использование Byte (но не наоборот), в Longint “входит” Smallint, который, в свою очередь, включает в себя Shortint.

Таблица 2.4

Название	Длина, байт	Диапазон значений
Cardinal	4	0. .. 2 147 483 647
Byte	1	0...255
Shortint	1	–128...+127
Smallint	2	–32 768...+32 767
Word	2	0...65 535
Integer	4	–2 147 483 648...+2 147 483 647
Longint	4	–2 147 483 648...+2 147 483 647
Int64	8	–9*1018. ..+9*1018
LongWord	4	0. . .4 294 967 295

Перечень процедур и функций, применимых к целочисленным типам, приведен в табл. 2.5. Буквами b, s, w, i, l обозначены выражения, соответственно, типа Byte, Shortint, Word, Integer и Longint, x – выражение любого из этих типов. В квадратных скобках указывается необязательный параметр.

Таблица 2.5

Обращение	Тип результата	Действие
abs (x)	X	Возвращает модуль x
chr(b)	Char	Возвращает символ по его коду
dec (vx [, i])	–	Уменьшает значение vx на i, а при отсутствии i – на 1
inc(vx[,i])	–	Увеличивает значение vx на i, а при отсутствии i – на 1
hi(w)	Byte	Возвращает старший бант аргумента
hi(I)	То же	Возвращает третий по счету байт
lo(i)	“	Возвращает младший байт аргумента
lo(w)	“	То же
odd(l)	Boolean	Возвращает True, если аргумент – нечетное число
random(w)	Как у параметра	Возвращает псевдослучайное число, равномерно распределенное в диапазоне 0...(w–l)
sqr(x)	X	Возвращает квадрат аргумента
swap(i)	Integer	Меняет местами байты в слове
swap (w)	Word	То же

При действиях с целыми числами тип результата будет соответствовать типу операндов, а если операнды относятся к различным целым типам – общему типу, который включает в себя оба операнда. Например,

при действиях с shortint и word общим будет тип integer. В стандартной настройке компилятор Delphi не вырабатывает код, осуществляющий контроль за возможной проверкой выхода значения из допустимого диапазона, что может привести к недоразумениям.

Логические типы

К логическим относятся типы Boolean, ByteBool, wordBool и LongBool. В стандартном Паскале определен только тип Boolean, остальные логические типы введены в Object Pascal для совместимости с Windows: типы Boolean и ByteBool занимают по одному байту каждый, Bool и WordBool – по 2 байта, LongBool – 4 байта. Значениями логического типа может быть одна из предварительно объявленных констант False (ложь) или True (истина). Для них справедливы правила:

```
Ord(False) = 0;
Ord(True) <> 0;
Succ(False)= True;
Pred(True) = False.
```

Поскольку логический тип относится к порядковым типам, его можно использовать в операторе цикла счетного типа, например:

```
var
l : Boolean;
begin
for l := False to True do ....
```

Символьный тип

Значениями символьного типа является множество всех символов ПК. Каждому символу приписывается целое число в диапазоне 0...255. Это число служит кодом внутреннего представления символа, его возвращает функция ord.

Для кодировки в Windows используется код ANSI (назван по имени American National Standard Institute – Американского Национального Института Стандартизации, предложившего этот код).

К типу char применимы операции отношения, а также встроенные функции:

Chr (в) – функция типа char; преобразует выражение типа Byte в символ и возвращает его своим значением;

UpCase(CH) – функция типа char; возвращает прописную букву, если CH – строчная латинская буква, в противном случае возвращает сам символ CH (для кириллицы возвращает исходный символ).

Перечисляемый тип

Перечисляемый тип задается перечислением тех значений, которые он может получать. Каждое значение именуется некоторым идентификатором и располагается в списке, обрамленном круглыми скобками, например:

```
type
colors = (red, white, blue);
```

Применение перечисляемых типов делает программы нагляднее. Если, например, в программе используются данные, связанные с месяцами года, то такой фрагмент программы будет иметь вид:

```
type
TMonth=(jan,feb,mar,may,jun,jul,aug,sep,oct,nov,dec);
var
Month : TMonth;
begin
if Month = aug then
IbOutput.Caption := 'Хорошо бы поехать к морю!';
end.
```

Соответствие между значениями перечисляемого типа и порядковыми номерами этих значений устанавливается порядком перечисления: первое значение в списке получает порядковый номер 0, второе – 1 и т.д. Максимальная мощность перечисляемого типа составляет 65536 значений, поэтому фактически перечисляемый тип задает некоторое подмножество целого типа word и может рассматриваться как компактное объявление сразу группы целочисленных констант со значениями 0, 1 и т. д.

Тип-диапазон

Тип-диапазон есть подмножество своего базового типа, в качестве которого может выступать любой порядковый тип, кроме типа-диапазона.

Тип-диапазон задается границами своих значений внутри базового типа:

```
<мин.знач.>..<макс.знач.>
```

Здесь <мин.знач. > – минимальное значение типа-диапазона;

<макс. знач. > – максимальное его значение.

Например:

```
type
digit = '0'..'9';
dig2 = 48 .. 57;
```

Тип-диапазон не обязательно описывать в разделе type, а можно указывать непосредственно при объявлении переменной, например:

```
var
date : 1. .31;
month: 1..12;
Ichr : 'A'..'Z';
```

При определении типа-диапазона нужно руководствоваться следующими правилами:

- два символа ".." рассматриваются как один символ, поэтому между ними недопустимы пробелы;
- левая граница диапазона не должна превышать его правую границу.

В стандартную библиотеку Object Pascal включены две функции, поддерживающие работу с типами-диапазонами:

High(x) – возвращает максимальное значение типа-диапазона, к которому принадлежит переменная x;

Low(x) – возвращает минимальное значение типа-диапазона.

Вещественные типы

В отличие от порядковых типов, значения которых всегда сопоставляются с рядом целых чисел и, следовательно, представляются в ПК абсолютно точно, значения вещественных типов определяют произвольное число лишь с некоторой конечной точностью, зависящей от внутреннего формата вещественного числа. Вещественные типы, реализованные в Object Pascal, приведены в таблице 2.6.

Таблица 2.6

Длина, байт	Название	Количество значащих цифр	Диапазон значений
8	Real	15…16	5.0*10e-324…1.7*10e308
4	Single	7…8	1.5*10e-45…3.4*10e38
8	Double	15…16	5.0*10e324…1.7*10e308
10	Extended	19…20	3.4*10-4951…1.1*10e4932
8	Comp	19…20	2e63…+2e63-1
8	Currency	19…20	+/-922 337 203 685477,5807

Следует отметить, что арифметический сопроцессор всегда обрабатывает числа в формате Extended, а три других вещественных типа в этом случае получаются простым усечением результатов до нужных размеров и применяются в основном для экономии памяти.

Особое положение в Object Pascal занимают типы Comp и Currency, которые трактуются как вещественные числа с дробными частями фиксированной длины: в Comp дробная часть имеет длину 0 разрядов, т. е. просто отсутствует, в Currency длина дробной части – 4 десятичных разряда. Фактически оба типа определяют большое целое число со знаком, сохраняющее 19...20 значащих десятичных цифр (во внутреннем представлении они занимают 8 смежных байт). В то же время в выражениях Comp и Currency полностью совместимы с любыми другими вещественными типами: над ними определены все вещественные операции, они могут использоваться как аргументы математических функций и т. д. Наиболее подходящей областью применения этих типов являются бухгалтерские расчеты.

Для работы с вещественными данными могут использоваться встроенные математические функции, представленные в табл. 2.7. В этой таблице Real означает любой вещественный тип, integer – любой целый тип.

Таблица 2.7

Обращение	Тип параметра	Тип результата	Примечание
abs(x)	Real, Integer	Тип аргумента Real	Модуль аргумента
pi	–	<<	П =3.141592653...
arcTan(x)			Арктангенс (значение в радианах)
cos(x)	То же <<	То же <<	Косинус, угол в радианах
exp(x)	<<	<<	Экспонента
frac(x)	<<	<<	Дробная часть числа
int(x)	<<	<<	Целая часть числа
ln(x)	<<	<<	Логарифм натуральный
random	–	<<	Псевдослучайное число, равномерно распределенное в диапазоне 0...[1]
random.fx)	Integer	Integer	Псевдослучайное целое число, равномерно распределенное в диапазоне 0...(x–1)
randomize	–	–	Инициация генератора псевдослучайных чисел
sin (x)	Real	Real	Синус, угол в радианах

Окончание таблицы 2.7

sqr(x)	То же	То же	Квадрат аргумента
sqrt(x)	<<	<<	Корень квадратный

Тип дата–время

Тип дата–время определяется стандартным идентификатором TDateTime и предназначен для одновременного хранения и даты, и времени. Во внутреннем представлении он занимает 8 байт и подобно Currency представляет собой вещественное число с фиксированной дробной частью: в целой части числа хранится дата, в дробной – время.

Над данными типа TDateTime определены те же операции, что и над вещественными числами, а в выражениях этого типа могут участвовать константы и переменные целого и вещественного типов.

Для работы с датой и временем используются подпрограммы, перечисленные в табл. 2.8.

Таблица 2.8

Function Date: TDateTime;	Возвращает текущую дату
Function DateToStr (D: TDateTime): String;	Преобразует дату в строку символов
Function DateTimeToStr (D: TDateTime): String;	Преобразует дату и время в строку символов
Function FormatDateTime (Format: String; Value: TDateTime): String;	Преобразует дату и время из параметра value в строку символов в соответствии со спецификаторами параметра Format
Function Now: TDateTime;	Возвращает текущую дату и время
Function Time: TDateTime;	Возвращает текущее время
Function TimeToStrfT: TDateTime): String;	Преобразует время в строку

Поскольку тип TDateTime совместим с форматом вещественных чисел, можно без труда определить дату, отстоящую от заданной на сколько-то дней вперед или назад: для этого достаточно соответственно прибавить к заданной дате или отнять от нее нужное целое число.

2.4.2. Структурированные типы

Любой из структурированных типов (в Object Pascal их четыре: массивы, записи, множества и файлы) характеризуется множественностью образующих этот тип элементов. Каждый элемент, в свою очередь, может принадлежать структурированному типу, что позволяет говорить

о возможной вложенности типов. В Object Pascal допускается произвольная глубина вложенности типов, однако суммарная длина любого из них во внутреннем представлении не должна превышать 2 Гбайт.

Массивы

Массивы в Object Pascal во многом схожи с аналогичными типами данных в других языках программирования. Отличительная особенность массивов заключается в том, что все их компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием его порядкового номера.

Описание типа массива задается следующим образом:

```
<имя типа> = array [ <сп.инд.типов> ] of <тип>;
```

Здесь <имя типа> – правильный идентификатор; array, of – зарезервированные слова (массив, из); <сп. инд. типов> – список из одного или нескольких индексных типов, разделенных запятыми; квадратные скобки, обрамляющие список, – требование синтаксиса.

Определить переменную как массив можно непосредственно при описании этой переменной, без предварительного описания типа массива, например:

```
var
a,b : array [1..10] of Real;
```

Обычно в качестве индексного типа используется тип-диапазон, в котором задаются границы изменения индексов. Так как тип <тип>, идущий в описании массива за словом of, – любой тип Object Pascal, то он может быть, в частности, и другим массивом, например:

```
type
mat = array [0..5] of array [-2..2] of array [Char] of Byte;
//Такую запись можно заменить более компактной:
type
mat = array [0..5,-2..2,char] of Byte;
```

В Object Pascal можно одним оператором присваивания передать все элементы одного массива другому массиву того же типа, например:

```
var
a,b : array [1..5] of Single;
begin
a := b;
end.
```

Над массивами не определены операции отношения. Поэтому сравнить два массива нужно поэлементно.

В случае необходимости при объявлении массивов в программе можно не указывать границы индексов. Полученные таким образом массивы называются динамическими, например:

```
var
A: array of Integer;
B: array of array of Char;
C: array of array of array of Real;
```

В приведенном примере массив A имеет одно измерение, массив B – два, массив C – три измерения. Распределение памяти и указание границ индексов по каждому измерению динамических массивов осуществляется в ходе выполнения программы путем инициации массива с помощью функции SetLength.

Например:

```
SetLength(A,3);
```

одномерный динамический массив, A будет инициирован, т. е. получит память, достаточную для размещения трех целочисленных значений. Нижняя граница индексов по любому измерению динамического массива всегда равна 0, поэтому верхней границей индексов для A станет 2.

Фактически идентификатор динамического массива ссылается на указатель, содержащий адрес первого байта памяти, выделенной для размещения массива. Поэтому для освобождения этой памяти достаточно присвоить идентификатору значение nil (другим способом является использование процедуры Finalize):

```
var
A,B: array of Integer;
begin
// Распределяем память:
SetLength(A,10);
SetLength(B,20);
// Используем массивы:
// Освобождаем память:
A := NIL;
Finalize(B);
end;
```

При работе с многомерными массивами сначала устанавливается длина его первого измерения, затем второго, третьего и т. д. Например:

```
var
A: array of array of Integer;//Двумерный динамический массив
begin
//Устанавливаем длину первого измерения (количество столб-
цов):
SetLength(A,3);
//Задаем длину каждого столбца:
SetLength(A[0],3);
SetLength(A[l],3);
SetLength(A[2] ,3);
end;
```

Записи

Запись – это структура данных, состоящая из фиксированного количества компонентов, называемых полями записи. В отличие от массива компоненты (поля) записи могут быть различного типа. Чтобы можно было ссылаться на тот или иной компонент записи, поля именуются.

Структура объявления типа записи имеет следующий вид:

```
<имя типа> = record <сп.полей> end;
```

Здесь <имя типа> – правильный идентификатор; record/ end – зарезервированные слова (запись, конец); <сп.полей> – список полей; представляет собой последовательность разделов записи, между которыми ставится точка с запятой. Каждый раздел записи состоит из одного или нескольких идентификаторов полей, отделяемых друг от друга запятыми. За идентификатором ставится двоеточие и описание типа поля, например:

```
type
BirthDay = record Day, Month: Byte;
Year : Word end;
var
a,b : Birthday;
```

В этом примере тип BirthDay есть запись с полями Day, Month и Year; переменные *a* и *b* содержат записи типа BirthDay.

Как и в массиве, значения переменных типа записи можно присваивать другим переменным того же типа, например

```
a := b;
```

К каждому из компонентов записи можно получить доступ, если использовать составное имя, т. е. указать имя переменной, затем точку и имя поля:

```
a.day := 27;
b.year := 1939;
```

Чтобы упростить доступ к полям записи, используется оператор присоединения with:

```
with <переменная> do <оператор>;
```

Здесь with, do – зарезервированные слова (с, делать);
<переменная> – имя переменной типа запись, за которой возможно следует список вложенных полей;
<оператор> – любой оператор Object Pascal.
Пример использования оператора with:

```
c.Bd.Month := 9;
// Это эквивалентно
with c.Bd do Month := 9;
```

Множества

Множества – это наборы однотипных логически связанных друг с другом объектов. Характер связей между объектами лишь подразумевается программистом и никак не контролируется Object Pascal. Количество элементов, входящих в множество, может меняться в пределах от 0 до 256 (множество, не содержащее элементов, называется пустым). Именно непостоянством количества своих элементов множества отличаются от массивов и записей.

Два множества считаются эквивалентными только тогда, когда все их элементы одинаковы, причем порядок следования элементов в множестве безразличен. Если все элементы одного множества входят также и в другое, говорят о включении первого множества во второе. Пустое множество включается в любое другое.

Описание типа множества имеет вид:

```
<имя типа> = set of <базовый тип>;
```

Здесь <имя типа> – правильный идентификатор;
set, of – зарезервированные слова (множество, из);

<базовый тип> – базовый тип элементов множества, в качестве которого может использоваться любой порядковый тип, кроме Word, Integer, Longint, Int64.

Пример определения и задания множеств:

```
type
digitChar = set of '0'..'9';
digit = set of 0. .9;
var
sl,s2,s3 : digitChar;
s4,s5,s6 : digit;
begin
si = ['1', '2', '3'];
s2 = ['3', '2', '1'];
s3 = ['2', '3'];
s4 = [0..3, 6];
s5 = [4, 5];
s6 = [3..9];
end.
```

Для задания множества используется так называемый конструктор множества: список спецификаций элементов множества, отделенных друг от друга запятыми; список обрамляется квадратными скобками. Спецификациями элементов могут быть константы или выражения базового типа, а также тип-диапазон того же базового типа.

Над множествами определены следующие операции:

* пересечение множеств; результат содержит элементы, общие для обоих множеств; например, s4*s6 содержит [3], s4*s5 – пустое множество (см. выше);

+ объединение множеств; результат содержит элементы первого множества, дополненные недостающими элементами из второго множества:

S4+S5 содержит [0,1,2,3,4,5,6];

S5+S6 содержит [3, 4, 5, 6, 7, 8, 9] ;

разность множеств; результат содержит элементы из первого множества, которые не принадлежат второму:

S6–S5 содержит [3,6,7,8,9];

S4–S5 содержит [0,1, 2, 3, 6] ;

= проверка эквивалентности; возвращает True, если оба множества эквивалентны;

<> проверка неэквивалентности; возвращает True, если оба множества неэквивалентны;

<= проверка вхождения; возвращает True, если первое множество включено во второе;

>= проверка вхождения; возвращает True, если второе множество включено в первое;
in проверка принадлежности; в этой бинарной операции первый элемент – выражение, а второй – множество одного и того же типа; возвращает True, если выражение имеет значение, принадлежащее множеству:

```
3 in s6 // возвращает True;
2*2 in s1 // возвращает False.
```

Дополнительно к этим операциям можно использовать две процедуры.
include – включает новый элемент во множество.
Обращение к процедуре: Include(S,I).
Здесь s – множество, состоящее из элементов базового типа TSetBase;
I – элемент типа TSetBase, который необходимо включить во множество.
exclude – исключает элемент из множества.
Обращение: exclude(S,I).
Параметры обращения – такие же, как у процедуры include. В отличие от операций + и –, реализующих аналогичные действия над двумя множествами, процедуры оптимизированы для работы с одиночными элементами множества и поэтому отличаются высокой скоростью выполнения.

Строки

Для обработки текстов в Object Pascal используются следующие типы:

- короткая строка shortString или string [n] , где n <= 255;
- длинная строка string;
- широкая строка WideString;
- нуль-терминальная строка pchar.

Общим для этих типов является то, что каждая строка трактуется как одномерный массив символов, количество символов в котором может меняться в работающей программе: для string [n] длина строки меняется от 0 до n, для string и pchar – от 0 до 2 Гбайт.

В Windows широко используются нуль-терминальные строки, представляющие собой цепочки символов, ограниченные символом #0. Максимальная длина такой строки лимитируется только доступной памятью и может быть очень большой. Следует отметить, что символы в нуль-терминальных строках нумеруются с 0.

В 32-разрядных версиях Delphi введен новый тип string, сочетающий в себе удобства обоих типов. При работе с этим типом память вы-

деляется по мере надобности (динамически) и ограничена имеющейся в распоряжении программы доступной памятью.

Для совместимости с компонентами, основывающимися на OLE-технологии, в Delphi–32 введены также широкие строки, объявляемые стандартным типом wideString. По своим свойствам они идентичны длинным строкам string, но отличаются от них тем, что для представления каждого символа используются не один, а два байта.
Примеры объявлений строковых типов:

```
var
ssS: String[250];// Короткая строка длиной до 250 символов
ssMax: ShortString;// Короткая строка длиной до 255 символов
stS: String; // Длинная строка
swS: WideString;// Широкая строка
pcS: PChar; // Ссылка на нуль-терминальную строку
acS: array [0..1000] of Char; // Нуль-терминальная строка
// длиной до 1000 символов
```

Типы String и ShortString

Несмотря на разницу во внутреннем представлении, короткие строки ShortString и длинные строки string имеют для программиста одинаковые свойства.

Текущую длину строки можно получить с помощью функции Length. Например:

```
Var
S:string;
Num:integer;
Begin
S:='Test';
Num:=Length(S); //Значение Num=4
End;
```

Требуемая длина строки устанавливается с помощью процедуры SetLength. При этом надо учесть, что если длина вводимой строки больше установленной, то “лишние” символы отбрасываются.

Стандартные подпрограммы преобразования строк в другие типы, а также другие процедуры и функции для работы со строками приведены в приложении 2.

Нуль-терминальные строки

Нуль-терминальные строки широко используются при обращениях к так называемым API-функциям Windows (API – Application Program

Interface – интерфейс прикладных программ). Поскольку компоненты Delphi берут на себя все проблемы связи с API-функциями Windows, программисту редко приходится прибегать к нуль-терминальным строкам.

Более подробное описание приведено в [4].

2.5. Операторы языка

2.5.1. Составной оператор и пустой оператор

Составной оператор – это последовательность произвольных операторов программы, заключенная в операторные скобки – зарезервированные слова begin ... end. Составные операторы – важный инструмент Object Pascal, дающий возможность писать программы по современной технологии структурного программирования (без операторов перехода goto).

Object Pascal не накладывает никаких ограничений на характер операторов, входящих в составной оператор. Среди них могут быть и другие составные операторы – язык Object Pascal допускает произвольную глубину их вложенности.

Пример использования составного оператора:

```
begin
     begin
          begin
          end;
     end;
end;
```

Фактически весь раздел операторов, обрамленный словами begin ... end, представляет собой один составной оператор. Поскольку зарезервированное слово end является закрывающей операторной скобкой, оно одновременно указывает и конец предыдущего оператора.

2.5.2. Условный оператор

Условный оператор позволяет проверить некоторое условие и, в зависимости от результатов проверки, выполнить то или иное действие. Таким образом, условный оператор – это средство ветвления вычислительного процесса.

Структура условного оператора имеет следующий вид:

```
if <условие> then <оператор1> else <оператор2>;
```

где if, then, else – зарезервированные слова (если, то, иначе);
<условие> – произвольное выражение логического типа;
<оператор1>, <оператор2> – любые операторы языка Object Pascal.

Условный оператор работает по следующему алгоритму. Вначале вычисляется условное выражение <условие>. Если результат есть True (истина), то выполняется <оператор1>, а <оператор2> пропускается; если результат есть False (ложь), наоборот, <оператор1> пропускается, а выполняется <оператор2>. Например:

```
var
X, Y, Max: Integer;
begin .
if X > Max then
Y := Max else
Y := X;
….
end;
```

При выполнении этого фрагмента переменная *y* получит значение переменной *x*, если только это значение не превышает *max*, в противном случае *y* станет равно *max*.

Условными называются выражения, имеющие одно из двух возможных значений: истина или ложь. Такие выражения чаще всего получаются при сравнении переменных с помощью операций отношения =, <>, >, >=, <, <=. Сложные логические выражения составляются с использованием логических операций and (логическое И), or (логическое ИЛИ) и not (логическое НЕ). Например:

```
if (a > b) and (b <> 0) then ...
```

В отличие от других языков программирования в Object Pascal приоритет операций отношения меньше, чем у логических операций, поэтому отдельные составные части сложного логического выражения заключаются в скобки. Например, такая запись предыдущего оператора будет неверной:

```
if a>b and b <> 0 then ...// Ошибка, так как фактически (с
учетом приоритета операции) компилятор будет транслировать
такую строку:
if a>(b and b)<>0 then...
```

2.5.3. Операторы повторений

В языке Object Pascal имеются три различных оператора, с помощью которых можно запрограммировать повторяющиеся фрагменты программ.

Счетный оператор цикла FOR имеет такую структуру:

```
for <параметр_цикла> := <нач_знач> to <кон_знач> do <опера-
тор>;
```

Здесь for, to, do – зарезервированные слова (для, до, выполнить);
<параметр_цикла> – переменная типа Integer;
<нач_знач> – начальное значение – выражение того же типа;
<кон_знач> – конечное значение – выражение того же типа;
<оператор> – произвольный оператор Object Pascal.
При выполнении оператора for вначале вычисляется выражение <нач_знач> и осуществляется присваивание <параметр_цикла> := <нач_знач>. После этого циклически повторяется:

- проверка условия <параметр_цикла> <= <кон_знач>; если условие не выполнено, оператор for завершает свою работу;
- выполнение оператора <оператор>;
- наращивание переменной <параметр_цикла> на единицу.

Другая форма записи оператора:

```
for <пар_цик>: = <нач_знач>downto <кон_знач>do <оператор>;
```

Замена зарезервированного слова to на downto означает, что шаг наращивания параметра цикла равен (–1), а управляющее условие приобретает вид <параметр__цикла> = <кон_знач>.

Оператор цикла WHILE с предпроверкой условия имеет структуру:

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

Здесь while, do – зарезервированные слова (пока [выполняется условие], делать);
<условие> – выражение логического типа;
<оператор> – произвольный оператор Object Pascal.
Если выражение <условие> имеет значение True, то выполняется <оператор>, после чего вычисление выражения <условие> и его проверка

повторяются. Если <условие> имеет значение False, оператор while прекращает свою работу.

Оператор цикла REPEAT... UNTIL с постпроверкой условия имеет структуру:

```
repeat <тело цикла> Until <условие>;
```

Здесь repeat, until – зарезервированные слова (повторять [до тех пор], пока [не будет выполнено условие]);
<тело_цикла> – произвольная последовательность операторов Object Pascal;
<условие> – выражение логического типа.
Операторы <тело_цикла> выполняются хотя бы один раз, после чего вычисляется выражение <условие>: если его значение есть False, операторы <тело_цикла> повторяются, в противном случае оператор repeat... until завершает свою работу.

Обратите внимание: пара repeat... unti1 подобна операторным скобкам begin ... end, поэтому перед until ставить точку с запятой необязательно.

Для гибкого управления циклическими операторами for, while и repeat в состав Object Pascal включены две процедуры без параметров:
break – реализует немедленный выход из цикла; действие процедуры заключается в передаче управления оператору, стоящему сразу за концом циклического оператора;
continue – обеспечивает досрочное завершение очередного прохода цикла; эквивалент передачи управления в самый конец циклического оператора.

2.5.4. Оператор выбора

Оператор выбора позволяет выбрать одно из нескольких возможных продолжений программы. Параметром, по которому осуществляется выбор, служит ключ выбора – выражение любого порядкового типа (к порядковым относятся типы integer, char, логический и др.).

Структура оператора выбора:

```
case <ключ_выбора> of <список_выбора> [else <операторы>] end;
```

Здесь case, of, else, end – зарезервированные слова (случай, из, иначе, конец); <ключ_выбора> – ключ выбора (выражение порядкового типа);
<список_выбора> – одна или более конструкций следующего вида:

<константа_выбора> – <оператор>;
<константа_выбора> – константа того же типа, что и выражение <ключ_выбора>; <оператор> – произвольный оператор Object Pascal.
Оператор выбора работает следующим образом. Вначале вычисляется значение выражения <ключ_выбора>, а затем в последовательности операторов <список_выбора> отыскивается такой, которому предшествует константа, равная вычисленному значению. Найденный оператор выполняется, после чего оператор выбора завершает свою работу. Если в списке выбора не будет найдена константа, соответствующая вычисленному значению ключа выбора, управление передается операторам, стоящим за словом else. Часть else <операторы> можно опускать. Тогда при отсутствии в списке выбора нужной константы ничего не произойдет, и оператор выбора просто завершит свою работу.

Любому из операторов списка выбора может предшествовать не одна, а несколько констант выбора, разделенных запятыми. Например:

```
var
ch : Char;
begin
case ch of
'n','N','Н','Н': IbOutput.Caption := 'Нет';
'y','Y','д','Д': IbOutput.Caption := 'Да';
end
end;
```

2.5.5. Метки и операторы перехода

В некоторых случаях использование операторов перехода может упростить программу. Однако следует отметить, что современная технология структурного программирования основана на принципе “программировать без GOTO”. Считается, что злоупотребление операторами перехода затрудняет понимание программы, делает ее запутанной и сложной в отладке.

В общем виде оператор перехода имеет вид:

```
goto <метка>;
```

Здесь goto – зарезервированное слово (перейти [на метку]);
<метка> – метка.
Метка в Object Pascal – это произвольный идентификатор, позволяющий именовать некоторый оператор программы и таким образом ссылаться на него. В целях совместимости со стандартным языком Паскаль

в Object Pascal допускается в качестве меток использование также целых чисел без знака. Метка располагается непосредственно перед помечаемым оператором и отделяется от него двоеточием. Оператор можно помечать несколькими метками, которые в этом случае отделяются друг от друга двоеточием. Перед тем как появиться в программе, метка должна быть описана. Описание меток состоит из зарезервированного слова label, за которым следует список меток:

```
Label
loop, Ibl, lb2;
begin
goto Ibl;

loop: .........
Ibl:lb2: ......
......
goto lb2;
end;
```

Действие оператора goto состоит в передаче управления соответствующему помеченному оператору. При использовании меток необходимо руководствоваться следующими правилами:

- метка, на которую ссылается оператор goto, должна быть описана в разделе описаний, и она обязательно должна встретиться где-нибудь в теле программы;
- метки, описанные в подпрограмме, локализуются в ней, поэтому передача управления извне подпрограммы на метку внутри нее невозможна.

2.6. Процедуры и функции

Процедурой в Object Pascal называется особым образом оформленный фрагмент программы, имеющий собственное имя. Упоминание этого имени в тексте программы приводит к активизации процедуры и называется ее вызовом. Сразу после активизации процедуры начинают выполняться входящие в нее операторы, после выполнения последнего из них управление возвращается обратно в основную программу, и выполняются операторы, стоящие непосредственно за оператором вызова процедуры.

Для обмена информацией между основной программой и процедурой используется один или несколько параметров вызова. Процедуры

могут иметь и другой механизм обмена данными с вызывающей программой, так что параметры вызова могут и не использоваться. Если они есть, то они перечисляются в круглых скобках за именем процедуры и вместе с ним образуют оператор вызова процедуры.

Функция отличается от процедуры тем, что результат ее работы возвращается в виде значения этой функции, и, следовательно, вызов функции может использоваться наряду с другими операндами в выражениях.

Стандартные процедуры и функции (общее название – подпрограммы), реализующие математические операции, приведены в приложении 1.

Описать подпрограмму – значит указать ее заголовок и тело. В заголовке объявляются имя подпрограммы и формальные параметры, если они есть. Для функции, кроме того, указывается тип возвращаемого ею результата. За заголовком следует тело подпрограммы, которое, подобно программе, состоит из раздела описаний и раздела исполняемых операторов. В разделе описаний подпрограммы могут встретиться описания подпрограмм низшего уровня, а в них – описания других подпрограмм и т. д.

При входе в подпрограмму низшего уровня становятся доступными не только объявленные в ней имена, но и сохраняется доступ ко всем именам верхнего уровня. Образно говоря, любая подпрограмма как бы окружена полупрозрачными стенками: снаружи подпрограммы мы не видим ее внутренности, но, попав в подпрограмму, можем наблюдать все, что делается снаружи, например:

```
var V1 : ... ;
Procedure A;
var V2 : ...;
end {A};
Procedure B;
var V3 : . . . ;
     Procedure B1;
     var V4 : . . . ;
          Procedure B11;
          var V5;
          begin
          end;
```

Из приведенного примера видно, что из процедуры B11 доступны все пять переменных v1,...,v5, из процедуры B1 доступны переменные v1,...,v4, из центральной программы только v1.

Отметим, что любая подпрограмма может вызвать саму себя – такой способ вызова называется рекурсией.

В общем виде описание подпрограммы состоит из заголовка и тела подпрограммы. Так заголовок процедуры имеет вид:

```
PROCEDURE <имя> [(<сп.ф.п.>)] ;
```

Заголовок функции:

```
FUNCTION <имя> [(<сп.ф.п.>)] : <тип>;
```

где <имя> – имя подпрограммы (правильный идентификатор);
<сп.ф.п.> – список формальных параметров;
<тип> – тип возвращаемого функцией результата.

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 MyProcedure(x: Real; y: Real; z:integer);
Function MyFunction(a: Real; b: Real): Real;
```

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

```
Procedure MyProcedure(x, y: Real; z:integer);
Function MyFunction(a, b: Real): Real;
```

Сразу за заголовком подпрограммы может следовать одна из стандартных директив assembler, external, far, forward, inline, interrupt, near. Эти директивы уточняют действия компилятора и распространяются только на всю подпрограмму, т. е., если за подпрограммой следует другая подпрограмма, стандартная директива, указанная за заголовком первой, не распространяется на вторую.

assembler – эта директива отменяет стандартную последовательность машинных инструкций, вырабатываемых при входе в процедуру и перед выходом из нее. Тело подпрограммы в этом случае должно реализоваться с помощью команд встроенного Ассемблера.

external – с помощью этой директивы объявляется внешняя подпрограмма.

far – компилятор должен создавать код подпрограммы, рассчитанный на дальнюю модель вызова.

near – заставит компилятор создать код, рассчитанный на ближнюю модель памяти. Введены для совместимости с Delphi 1, которая использовала сегментную модель памяти.

forward – используется при опережающем описании для сообщения компилятору, что описание подпрограммы следует где-то дальше по тексту программы (но в пределах текущего программного модуля).

inline – указывает на то, что тело подпрограммы реализуется с помощью встроенных машинных инструкций.

interrupt – используется при создании процедур обработки прерываний.

2.6.1. Параметры

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 SB(a: Real; b: Integer; c: Char);
```

Как видно из примера, параметры в списке отделяются друг от друга точками с запятой. Несколько следующих подряд однотипных параметров можно объединять в подсписки, например, вместо

```
Function F(a: Real; b: Real): Real;
```

можно написать проще:

```
Function F(a,b: Real): Real;
```

Операторы тела подпрограммы рассматривают список формальных параметров как своеобразное расширение раздела описаний: все переменные из этого списка могут использоваться в любых выражениях внутри подпрограммы. Таким способом осуществляется настройка алгоритма подпрограммы на конкретную задачу.

Рассмотрим такой полезный пример. В Object Pascal не предусмотрена операция возведения вещественного числа в произвольную степень (начиная с версии 2 с Delphi поставляется модуль Match, в котором есть соответствующая функция.). Тем не менее эту задачу можно решить с использованием стандартных математических функций Exp и Ln по следующему алгоритму:

$$X^Y = e^{(Y \cdot Ln(X))}.$$

Создадим функцию с именем power и двумя вещественными параметрами A и B, которая будет возвращать результат возведения A в степень B. Обработчик события bbRunClick нашей учебной формы читает из компонента edInput текст и пытается выделить из него два числа, разделенных хотя бы одним пробелом. Если это удалось сделать, он обращается к функции power дважды: сначала возводит первое число x в степень второго числа y, затем x возводится в степень y.

```
procedure TfmExample.bbRunClick(Sender: TObject);
Function Power(A, B: Real): Real;
{Функция возводит число A в степень B. Поскольку логарифм
отрицательного числа не существует, реализуется проверка
значения A: отрицательное значение заменяется на положитель-
ное, для нулевого числа результат равен нулю. Кроме того,
любое число в нулевой степени дает единицу.}
begin
if A > 0 then
Result:= Exp(B * Ln(A)) else if A < 0 then
Result:= Exp(B * Ln(Abs(A))) else if B = 0 then
Result:= 1 else
Result:= 0;
end; // Power var
S: String;
X, Y: Real; begin
{Читаем строку из edinput и выделяем из нее два вещественных
числа, разделенных хотя бы одним пробелом.}
S:= edinput.Text;
if (S = '') or (pos(' ' ,S) = 0) then
Exit; // Нет текста или в нем нет пробела - прекращаем
      // дальнейшую работу
try
// Выделяем первое число:
X:= StrToFloat(copy(S, I, pos(' ', S) - 1));
// Если успешно, удаляем символы до пробела // и выделяем
второе число:
Delete (S, 1, pos (' ', S));
Y:= StrToFloat(Trim(S));
except
Exit; // Завершаем работу при ошибке преобразования
end;
mmOutput.Lines.Add(FloatToStr(Power(X, Y)));
mmOutput.Lines.Add(FloatToStr(Power(X, -Y)));
end;
```

Для вызова функции Power мы просто указали ее в качестве параметра при обращении к стандартной функции преобразования вещественного числа в строку FloatToStr. Параметры x и y в момент обращения к функции power – это фактические параметры. Они подставляются вместо формальных параметров A и B в заголовке функции, и затем над ними осуществляются нужные действия. Полученный результат присваивается специальной переменной с именем Resuit, которая в теле любой функции интерпретируется как то значение, которое вернет функция после окончания своей работы. В программе функция power вызывается дважды – сначала с параметрами x и y, а затем x и y, поэтому будут получены два разных результата.

Механизм замены формальных параметров на фактические позволяет нужным образом настроить алгоритм, реализованный в подпрограмме. Object Pascal следит за тем, чтобы количество и типы формальных параметров строго соответствовали количеству и типам фактических параметров в момент обращения к подпрограмме. Смысл используемых фактических параметров зависит от того, в каком порядке они перечислены при вызове подпрограммы. В нашем примере первый по порядку фактический параметр будет возводиться в степень, задаваемую вторым параметром, а не наоборот. Программист должен сам следить за правильным порядком перечисления фактических параметров при обращении к подпрограмме.

Любой из формальных параметров подпрограммы может быть либо параметром-значением, либо параметром-переменной, либо, наконец, параметром-константой.

В предыдущем примере параметры A и B определены как параметры-значения. Если параметры определяются как параметры-переменные, перед ними необходимо ставить зарезервированное слово var, а если это параметры-константы – слово const, например:

```
Procedure MyProcedure(var A: Real; B: Real; const C: String);
```

Здесь A – параметр-переменная, B – параметр-значение, а C – параметр-константа. Определение формального параметра тем или иным способом существенно в основном только для вызывающей программы: если формальный параметр объявлен как параметр-переменная, то при вызове подпрограммы ему должен соответствовать фактический параметр в виде переменной нужного типа; если формальный параметр объявлен как параметр-значение или параметр-константа, то при вызове ему может соответствовать произвольное выражение. Контроль за соблюдени-

ем этого правила осуществляется компилятором Object Pascal. Если бы для предыдущего примера был использован такой заголовок функции:

```
Function Power (A: Real; var B : Real): Real;
```

то при втором обращении к функции компилятор указал бы на несоответствие типа фактических и формальных параметров (параметр обращения Y есть выражение, в то время как соответствующий ему формальный параметр B описан как параметр-переменная).

Для того чтобы понять, в каких случаях использовать тот или иной тип параметров, рассмотрим, как осуществляется замена формальных параметров на фактические в момент обращения к подпрограмме.

Если параметр определен как параметр-значение, то перед вызовом подпрограммы это значение вычисляется, полученный результат копируется во временную память (стек) и передается подпрограмме. Важно учесть, что даже если в качестве фактического параметра указано простейшее выражение в виде переменной или константы, все равно подпрограмме будет передана лишь копия переменной (константы). Любые возможные изменения в подпрограмме параметра-значения никак не воспринимаются вызывающей программой, так как в этом случае изменяется копия фактического параметра.

Если параметр определен как параметр-переменная, то при вызове подпрограммы передается сама переменная, а не ее копия (фактически в этом случае подпрограмме передается адрес переменной). Изменение параметра-переменной приводит к изменению фактического параметра в вызывающей программе.

В случае параметра-константы в подпрограмму также передается адрес области памяти, в которой располагается переменная или вычисленное значение. Однако компилятор блокирует любые присваивания параметру-константе нового значения в тексте подпрограммы.

Итак, параметры-переменные используются как средство связи алгоритма, реализованного в подпрограмме, с внешним миром: с помощью этих параметров подпрограмма может передавать результаты своей работы вызывающей программе. Разумеется, в распоряжении программиста всегда есть и другой способ передачи результатов – через глобальные переменные. Однако злоупотребление глобальными связями делает программу, как правило, запутанной, трудной в понимании и сложной в отладке. В соответствии с требованиями хорошего стиля программирования рекомендуется там, где это возможно, использовать передачу результатов через фактические параметры-переменные.

С другой стороны, описание всех формальных параметров как параметров-переменных нежелательно по двум причинам. Во-первых, это исключает возможность вызова подпрограммы с фактическими параметрами в виде выражений, что делает программу менее компактной. Во-вторых, в подпрограмме возможно случайное использование формального параметра, например для временного хранения промежуточного результата, т. е. всегда существует опасность непреднамеренно испортить фактическую переменную. Вот почему параметрами-переменными следует объявлять только те, через которые подпрограмма в действительности передает результаты вызывающей программе. Чем меньше параметров объявлено параметрами-переменными и чем меньше в подпрограмме используется глобальных переменных, тем меньше опасность получения не предусмотренных программистом побочных эффектов, связанных с вызовом подпрограммы, тем проще программа в понимании и отладке. По той же причине не рекомендуется использовать параметры-переменные в заголовке функции: если результатом работы функции не может быть единственное значение, то логичнее использовать процедуру или нужным образом декомпозировать алгоритм на несколько подпрограмм.

Существует еще одно обстоятельство, которое следует учитывать при выборе вида формальных параметров. Как уже говорилось, при объявлении параметра-значения осуществляется копирование фактического параметра во временную память. Если этим параметром будет массив большой размерности, то существенные затраты времени и памяти на копирование при многократных обращениях к подпрограмме можно минимизировать, объявив этот параметр параметром-константой. Параметр-константа не копируется во временную область памяти, что сокращает затраты времени на вызов подпрограммы, однако любые его изменения в теле подпрограммы невозможны – за этим строго следит компилятор.

Еще одно свойство Object Pascal – возможность использования нетипизированных параметров. Параметр считается нетипизированным, если тип формального параметра-переменной в заголовке подпрограммы не указан, при этом соответствующий ему фактический параметр может быть переменной любого типа. Заметим, что нетипизированными могут быть только параметры-переменные:

```
Procedure MyProc(var aParametr);
```

Нетипизированные параметры обычно используются в случае, когда тип данных несущественен. Такие ситуации чаще всего возникают при

разного рода копированиях одной области памяти в другую, например, с помощью процедур BlockRead, BlockWrite, и тд.

2.7. Классы и интерфейсы

Классами в Object Pascal называются специальные типы, которые содержат поля, методы и свойства. Как и любой другой тип, класс служит лишь образцом для создания конкретных экземпляров реализации, которые называются объектами.

Важным отличием классов от других типов является то, что объекты класса всегда распределяются в куче, поэтому объект-переменная фактически представляет собой лишь указатель на динамическую область памяти.

Пример описания собственного класса:

```
type
TMyClass = class(TObject)
  Field: Integer;
end;
var
MyClass: TMyClass;
begin
  MyClass.Field := 0; // Запишем значение 0 в поле Field
класса MyClass
end;
```

В основе классов лежат три фундаментальных принципа, которые называются инкапсуляция, наследование и полиморфизм.

Инкапсуляция

Класс представляет собой единство трех сущностей – полей, методов и свойств. Объединение этих сущностей в единое целое и называется инкапсуляцией. Инкапсуляция позволяет во многом изолировать класс от остальных частей программы, сделать его “самодостаточным” для решения конкретной задачи. В результате класс всегда несет в себе некоторую функциональность. Например, класс TForm содержит (инкапсулирует в себе) все необходимое для создания Windows-окна, класс TMemo представляет собой полнофункциональный текстовый редактор.

Инкапсуляция представляет собой мощное средство обмена готовыми к работе программными заготовками. Библиотека классов Delphi – это фактически набор “кирпичиков”, созданных программистами Borland для построения ваших программ.

Наследование

Любой класс может быть порожден от другого класса. Для этого при его объявлении указывается имя класса-родителя:

```
TChildClass = class (TParentClass)
```

Порожденный класс автоматически наследует поля, методы и свойства своего родителя и может дополнять их новыми. Таким образом, принцип наследования обеспечивает поэтапное создание сложных классов и разработку собственных библиотек классов.

Все классы Object Pascal порождены от единственного родителя класса TObject. Этот класс не имеет полей и свойств, но включает в себя методы самого общего назначения, обеспечивающие весь жизненный цикл любых объектов – от их создания до уничтожения. Программист не может создать класс, который не был бы дочерним классом TObject. Так следующие два объявления идентичны:

```
TaClass = class(TObject)
TaClass = class
```

Полиморфизм

Полиморфизм – это свойство классов решать схожие по смыслу проблемы разными способами. В рамках Object Pascal поведенческие свойства класса определяются набором входящих в него методов. Изменяя алгоритм того или иного метода в потомках класса, программист может придавать этим потомкам отсутствующие у родителя специфические свойства. Для изменения метода необходимо перекрыть его в потомке, т. е. объявить в потомке одноименный метод и реализовать в нем нужные действия. В результате в объекте-родителе и объекте-потомке будут действовать два одноименных метода, имеющих разную алгоритмическую основу и, следовательно, придающих объектам разные свойства. Это и называется полиморфизмом объектов.

2.7.1. Составляющие класса

Полями – называются инкапсулированные в классе данные. Поля могут быть любого типа, в том числе и классами, например:

```
type TMyClass = class
aIntField: Integer;
```

```
aStrField: String;
aObjField: TObject;
end;
```

Каждый объект получает уникальный набор полей, но общий для всех объектов данного класса набор методов и свойств. Фундаментальный принцип инкапсуляции требует обращаться к полям только с помощью методов и свойств класса. Однако в Object Pascal разрешается обращаться к полям и напрямую:

```
type
TMyClass = class
FIntField: Integer;
FStrField: String; end;
var
aObject: TMyClass;
begin
aObject.FIntField := 0;
aObject.FStrField := 'Строка символов';
end;
```

Класс-потомок получает все поля всех своих предков и может дополнять их своими, но он не может переопределять их или удалять. Таким образом, чем ниже в дереве иерархии располагается класс, тем больше данных получают в свое распоряжение его объекты.

Методами называются инкапсулированные в классе процедуры и функции. Они объявляются так же, как и обычные подпрограммы:

```
type
TMyClass = class
Function MyFunc(aPar: Integer): Integer;
Procedure MyProc;
end;
```

Доступ к методам класса, как и к его полям, возможен с помощью составных имен:

```
var
aObject: TMyClass;
begin
aObject.MyProc;
end;
```

Как уже говорилось, методы класса могут перекрываться в потомках. Например:

```
type
TParentClass = class Procedure DoWork;
end;
TChildClass = class(TParentClass) Procedure DoWork;
end;
```

Потомки обоих классов могут выполнять сходную по названию процедуру DoWork, но, в общем случае, будут это делать по-разному. Такое замещение методов называется статическим, т. к. реализуется компилятором.

В Object Pascal гораздо чаще используется динамическое замещение методов на этапе прогона программы. Для реализации этого метода, замещаемый в родительском классе, должен объявляться как динамический (с директивой dynamic) или виртуальный (virtual). Встретив такое объявление, компилятор создаст две таблицы – DMT (Dynamic Method Table) и VMT (Virtual Method Table) и поместит в них адреса точек входа, соответственно, динамических и виртуальных методов. При каждом обращении к замещаемому методу компилятор вставляет код, позволяющий извлечь адрес точки входа в подпрограмму из той или иной таблицы. В классе-потомке замещающий метод объявляется с директивой override (перекрыть). Получив это указание, компилятор создаст код, который на этапе прогона программы поместит в родительскую таблицу точку входа метода класса-потомка, что позволит родителю выполнить нужное действие с помощью нового метода.

Пусть, например, родительский класс с помощью методов show и Hide соответственно показывает что-то на экране или прячет изображение. Для создания изображения он использует метод Draw с логическим параметром:

```
type
TVisualObject = class(TWinControl)
Procedure Hide;
Procedure Show;
Procedure Draw(IsShow: Boolean); virtual;
end;
TVisualChildObject = class(TVisualObject)
Procedure Draw(IsShow: Boolean); override;
end;
// Реализация методов Show и Hide:
Procedure TVisualObject.Show;
begin
```

```
Draw(True) ;
end;
Procedure TVisualObject.Hide;
begin
Draw(False) ;
end;
```

Методы Draw у родителя и потомка имеют разную реализацию и создают разные изображения. В результате родительские методы – Show и Hide – прятать или показывать те или иные изображения будут в зависимости от конкретной реализации метода Draw у любого из своих потомков. Динамическое связывание в полной мере реализует полиморфизм классов.

Разница между динамическими и виртуальными методами состоит в том, что таблица динамических методов DMT содержит адреса только тех методов, которые объявлены как dynamic в данном классе, в то время как таблица VMT содержит адреса виртуальных методов не только данного класса, но и всех его родителей. Значительно большая по размеру таблица VMT обеспечивает более быстрый поиск, в то время как при обращении к динамическому методу программа сначала просматривает таблицу DMT у объекта, затем у его родительского класса и так далее, пока не будет найдена нужная точка входа.

В состав любого класса входят два специальных метода – конструктор и деструктор. У класса TObject эти методы называются Create и Destroy, так же они называются в подавляющем большинстве его потомков. Конструктор распределяет объект в динамической памяти и помещает адрес этой памяти в переменную self, которая автоматически объявляется в классе. Деструктор удаляет объект из кучи. Обращение к конструктору должно предварять любое обращение к полям и некоторым методам объекта. По своей форме конструкторы и деструкторы являются процедурами, но объявляются с помощью зарезервированных слов Constructor и Destructor:

```
type
TMyClass = class
IntField: Integer;
Constructor Create(Value: Integer);
Destructor Destroy;
end;
```

Любые поля объекта, а также методы класса, оперирующие с его полями, могут вызываться только после создания объекта с помощью вызова конструктора, т. к. конструкторы распределяют объект в дина-

мической памяти и делают действительным содержащийся в объекте указатель:

```
var
MyObject: TMyClass;
begin
MyObject.IntField := 0;
// { Ошибка! Объект не создан конструктором!}
MyObject := TMyClass.Create;
// Надо так: создаем объект
MyObject.IntField := 0;
// и обращаемся к его полю
MyObect.Free;
// Уничтожаем ненужный объект
end;
```

В базовом классе TObject определен метод Free, который сначала проверяет действительность адреса объекта и лишь затем вызывает деструктор Destroy. Обращение к деструктору объекта будет ошибочным, если объект не создан конструктором, поэтому для уничтожения ненужного объекта следует вызывать метод Free.

Некоторые методы могут вызываться без создания и инициации объекта. Такие методы называются методами класса, они объявляются с помощью зарезервированного слова class:

```
type
TMyClass = class(TObject)
class Function GetClassName: String;
end;
var
S: String;
begin
S := TMyClass.GetClassName;
end;
```

Методы класса не должны обращаться к полям, т. к. в общем случае вызываются без создания объекта, а следовательно, в момент вызова полей просто не существует. Обычно они возвращают служебную информацию о классе – имя класса, имя его родительского класса, адрес метода и т. п.

Свойства – это специальный механизм классов, регулирующий доступ к полям. Свойства объявляются с помощью зарезервированных слов property, read и write (слова read и write считаются зарезервирован-

ными только в контексте объявления свойства). Обычно свойство связано с некоторым полем и указывает те методы класса, которые должны использоваться при записи в это поле или при чтении из него. Например:

```
type
TaClass = class
IntField: Integer;
Function GetField: Integer;
Procedure SetField (Value: Integers);
Property IntegerValue: Integer
read GetField
write SetField;
end;
```

В контексте программы свойство ведет себя как обычное поле. Например, мы могли бы написать такие операторы:

```
var
aClass: TaClass;
Value: Integer;
begin
aClass := TaClass.Create;
// Обязательное обращение к конструктору перед обращением к
полю или свойству! aClass.IntegerValue := 0;
Value := aClass.IntegerValue;
aClass.Destroy; // Удаление ненужного объекта
end;
```

Более того, возможен и такой оператор присваивания:

```
aClass.IntField := NewValue;
```

Разница между этим оператором и оператором

```
aClass.IntegerValue := NewValue;
```

заключается в том, что при обращении к свойству автоматически подключается метод setField, в котором могут реализовываться специфичные действия.

Если необходимо, чтобы свойство было доступно только для чтения или только для записи, следует опустить соответственно часть write или read. Вообще говоря, свойство может и не связываться с полем. Фактически оно описывает один или два метода, которые осуществляют некоторые действия над данными того же типа, что и свойство.

Наиболее полно работа с классами изложена в [7, 8].

2.7.2. Объявление класса

Любой вновь создаваемый класс может содержать секции (разделы), определяемые зарезервированными словами **published** (опубликованные), **private** (закрытые), **protected** (защищенные), **public** (доступные) и **automated** (автоматизированные). Внутри каждой секции вначале определяются поля, а затем – методы и свойства.

Секции определяют области видимости элементов описания класса. Секция public не накладывает ограничений на область видимости перечисляемых в ней полей, методов и свойств – их можно вызывать в любом другом модуле программы. Секция published также не ограничивает область видимости, однако в ней перечисляются свойства, которые должны быть доступны не только на этапе исполнения, но и на этапе конструирования программы (т. е. в окне Инспектора объектов). Секция published используется только при разработке нестандартных компонентов. Заметим, что среда Delphi помещает описания компонентов, вставленных в форму, в специальную секцию без названия, которая располагается сразу за заголовком класса и продолжается до первой объявленной секции. Эта секция – published. Программисту не следует помещать в нее собственные элементы описания класса или удалять из нее элементы, вставленные средой. Секция private сужает область видимости до минимума: закрытые элементы описания доступны только внутри методов данного класса и подпрограммах, находящихся в том же модуле, где описан класс. Элемент, объявленный в секции private, становится недоступным даже ближайшим потомкам класса, если они размещаются в других модулях. Секция protected доступна только методам самого класса, а также любым его потомкам, независимо от того, находятся ли они в том же модуле или нет. Наконец, секция automated используется только для объявления свойств и методов, которые будут добавлены к так называемому интерфейсу OLE-объектов автоматизации; область видимости членов этой секции не ограничена.

В Object Pascal разрешается сколько угодно раз объявлять любую секцию, причем порядок следования секций не имеет значения. Любая секция может быть пустой.

Следующий фрагмент кода поясняет области видимости.

```
Unit Unit1;
Interface
Uses Controls, Forms;
type
TForm1 = class(TForm)
```

```
Button1: TButton; // Эта секция обслуживается Delphi
                  // Ее элементы доступны всем
private
// Эта секция доступна в модуле Uniti
FIntField: Integers
Procedure SetValue(Value: Integers);
Function GetValue: Integer;
published
// Эта секция доступна в любом модуле
Property IntField: read GetValue write SetValue;
protected // Эта секция доступна классам-потомкам
Procedure Proc1;
public // Эта секция доступна в любом модуле Procedure Proc2;
end;
var
Form1: TForm1;
Implementation Procedure TForm1.Proc1 ;
Button1.Color := clBtnFace;
// Так можно
FIntField := 0;
// Так можно
IntField := 0;
end;
begin
Form1.Button1.Color := clBtnFace; // Так можно
Form1.FIntField := 0; // Так можно
Form1.IntField := 0; // Так можно
Form1.Proc1; // Так нельзя!
Form1.Proc2; // Так можно
end.
Unit Unit2;
Interface
Uses Controls, Unit1;
type
TForm2 = class(TFormI)
Button2: TButton;
Procedure Button2Click(Sender: TObject);
end;
var
Form2: TForm2;
Implementation
Procedure TForm2.Button2Click(Sender: TObject);
begin
Button1.Color := clBtnFace; // Так можно
FIntField := 0; // Так нельзя!
IntField := 0; // Так можно
Proc1; // Так можно
Proc2; // Так можно
```

```
end;
begin
Form1.Button1.Color := clBtnFace; // Так можно
Form1.FIntField := 0; // Так нельзя!
Form1.IntField := 0; // Так можно
Form1.Proc1; //Так нельзя!
Form1.Proc2; // Так можно
end.
```

При объявлении класса-потомка разрешается перемещать элементы класса из одной области видимости в другую. Для предыдущего примера допустимо такое объявление:

```
type
TForm2 = class(Tform1)
Public
Procedure Proc1;
end;
```

После этого в модуле unit2 возможно такое обращение:

```
Form2.Proc1;
```

После перемещения в секцию private элемент объявления становится невидим потомкам (если потомок, как это обычно бывает, объявляется в другом модуле), и, следовательно, его уже нельзя переместить в другую секцию.

Класс может объявляться только в интерфейсной области модуля или в самом начале области реализации. Нельзя определять классы в разделе описаний подпрограмм.

2.7.3. Интерфейсы

Интерфейсы играют главную роль в технологиях COM (Component Object Model – компонентная модель объектов), CORBA (Common Object Request Broker Architecture – архитектура с брокером требуемых общих объектов) и связанных с ними технологиях удаленного доступа, т. е. технологиях доступа к объектам, расположенным (и выполняющимся) на другой машине. Их основная задача – описать свойства, методы и события удаленного объекта в терминах машины клиента, т. е. на используемом при разработке клиентского приложения языке программирования. С помощью интерфейсов программа клиента обращает-

ся к удаленному объекту так, как если бы он был ее собственным объектом.

Интерфейсы представляют собой частный случай описания типов. Они объявляются с помощью зарезервированного слова interface. Например:

```
type
IEdit = interface
procedure Copy; stdcall;
procedure Cut; stdcall;
procedure Paste; stdcall;
function Undo: Boolean; stdcall;
end;
```

Такое объявление эквивалентно описанию абстрактного класса в том смысле, что провозглашение интерфейса не требует расшифровки объявленных в нем свойств и методов.

В отличие от классов интерфейс не может содержать поля, и следовательно, объявляемые в нем свойства в разделах read и write могут ссылаться только на методы. Все объявляемые в интерфейсе члены размещаются в единственной секции public. Методы не могут быть абстрактными (abstract), виртуальными (virtual), динамическими (dynamic) или перекрываемыми (override). Интерфейсы не могут иметь конструкторов или деструкторов, т. к. описываемые в них методы реализуются только в рамках поддерживающих их классов, которые называются интерфейсными.

Если какой-либо класс поддерживает интерфейс (т. е. является интерфейсным), имя этого интерфейса указывается при объявлении класса в списке его родителей:

```
TEditor = class(TInterfacedObject,IEdit)
procedure Copy; stdcall;
procedure Cut; stdcall;
procedure Paste; stdcall;
function Undo: Boolean; stdcall;
end;
```

В отличие от обычного класса интерфейсный класс может иметь более одного родительского интерфейса:

```
type
IMylnterface = interface procedure Delete; stdcall;
end;
TMyEditor = class(TInterfacedObiect, lEdit, IMylnterface)
```

```
procedure Copy; stdcall;
procedure Cut; stdcall;
procedure Paste; stdcall;
function Undo:, Boolean; stdcall;
procedure Delete; stdcall;
end;
```

В любом случае в разделе реализации интерфейсного класса необходимо описать соответствующие интерфейсные методы. Если, например, объявлен интерфейс

```
IPaint = interface
procedure CirclePaint(Canva: TCanvas; X,Y,R: Integer);
procedure RectPaint(Canva: TCanvas; X1,Y1,X2,Y2: Integer);
end;
```

и использующий его интерфейсный класс

```
TPainter = class(TInterfacedObject,IPaint)
procedure CirclePaint(Canva: TCanvas; X,Y,R: Integers);
procedure RectPaint(Canva: TCanvas; X1,Y1,X2,Y2: Integer);
end;
```

то в разделе implementation следует указать реализацию методов:

```
procedure TPainter.CirclePaint(Canva: TCanvas;
X,Y,R: Integers);
begin
with Canva do
Ellipse(X, Y, X+2*R, Y+2*R);
end;
procedure TPainter.RectPaint(Canva: TCanvas;
X1,Y1,X2,Y2: Integer);
begin
with Canva do
Rectangle(XI, Yl, X2, Y2)
end;
```

Теперь можно объявить интерфейсный объект класса TPainter, чтобы с его помощью нарисовать окружность и квадрат:

```
procedure TFormI.PaintBoxIPaint(Sender: TObject);
var
Painter: IPaint;
begin
Painter:= TPainter.Create;
Painter.CirclePaint(PaintBoxl.Canvas,10,0,10);
```

```
Painter.RectPaint(PaintBoxl.Canvas,40,0,60,20);
end;
```

Несмотря на то что интерфейс всегда объявляется до объявления использующего его интерфейсного класса и, следовательно, известен компилятору, его методы обязательно должны быть перечислены в объявлении класса. В нашем случае простое указание

```
type
TPainter = class(TInterfacedObject, IPaint)
end;
```

было бы ошибкой: компилятор потребовал бы вставить описание методов CirclePaint и RectPaint.

Подобно тому, как все классы в Object Pascal порождены от единственного родителя TObject, все интерфейсные классы порождены от общего предка TInterfacedObject. Этот предок умеет распределять память для интерфейсных объектов и использует глобальный интерфейс lunknow:

```
type
TInterfacedObject = class(TObject, lUnknown) private
FRefCount: Integer;
protected
function Querylnterface(const IID: TGUID; out Obj): Integer;
stdcall;
function _AddRef: Integer; stdcall;
function _Release: Integer; stdcall;
public
property RefCount: Integer read FRefCount;
end;
```

Если бы в предыдущем примере класс TPainter был описан так:

```
TPainter = class(IPaint)
procedure CirclePaint(Canva: TCanvas; X,Y,R: Integer);
procedure RectPaint(Canva: TCanvas; X1,Y1,X2,Y2: Integer);
end;
```

компилятор потребовал бы описать недостающие методы Queryinterface, _AddRef и _Release класса TInterfacedObject. Поле RefCount этого класса служит счетчиком вызовов интерфейсного объекта и используется по принятой в Windows схеме: при каждом обращении к методу Add интерфейса IUnknow счетчик наращивается на единицу, при каждом обращении к Release – на единицу сбрасывается. Когда значение этого по-

ля становится равно 0, интерфейсный объект уничтожается и освобождается занимаемая им память. Если интерфейс предполагается использовать в технологиях COM/DCOM или CORBA, его методы должны описывать с директивой stdcall или (для объектов автоматизации) safecall

К интерфейсному объекту можно применить оператор приведения типов as, чтобы использовать нужный интерфейс:

```
procedure PaintObjects(P: TInterfacedObject) var X: IPaint;
begin
try
X:= P as IPaint;
X.CirclePaint(PaintBoxl.Canvas,0,0,20)
except
ShowMessage('Объект не поддерживает интерфейс IPaint')
end
end;
```

Встретив такое присваивание, компилятор создаст код, с помощью которого вызывается метод Queryinterface интерфейса IUnknow с требованием вернуть ссылку на интерфейс IPaint. Если объект не поддерживает указанный интерфейс, возникает исключительная ситуация.

Интерфейсы, рассчитанные на использование в удаленных объектах, должны снабжаться глобально-уникальным идентификатором (guiD). Наиболее подробно работа с интерфейсами рассмотрена в [1, 5, 7, 8].

2.8. Файлы

Под файлом понимается именованная область внешней памяти ПК (жесткого диска, гибкой дискеты, диска CD-ROM).

Любой файл имеет три характерные особенности:

- у файла есть имя, что дает возможность программе работать одновременно с несколькими файлами;
- файл содержит компоненты одного типа. Типом компонентов может быть любой тип Object Pascal, кроме файлов. Иными словами, нельзя создать “файл файлов”;
- длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип можно задать одним из трех способов:

```
<имя> = File of <тип>;
<имя> = TextFile;
<имя> = File;
```

где <имя> – имя файлового типа (правильный идентификатор);
File, of – зарезервированные слова (файл, из);
TextFile – имя стандартного типа текстовых файлов;
<тип> – любой тип Object Pascal, кроме файлов. Например:

```
TextSO = File of String[80];
var
F1: File of Char;
F2: TextFile;
F3: File;
F4: TextSO;
F5: File of Product
```

В зависимости от способа объявления можно выделить три вида файлов:

- типизированные файлы (задаются предложением File of ...);
- текстовые файлы (определяются типом TextFile);
- нетипизированные файлы (определяются типом File).

В приведенном выше примере F1, F4 и F5 – типизированные файлы, F2 – текстовый файл, F3 – нетипизированный файл. Вид файла, вообще говоря, определяет способ хранения в нем информации. Однако в Object Pascal нет средств контроля вида ранее созданных файлов. При объявлении уже существующих файлов программист должен сам следить за соответствием вида объявления характеру хранящихся в файле данных.

2.8.1. Доступ к файлам

Файлы становятся доступны программе только после выполнения особой процедуры открытия файла. Эта процедура заключается в связывании ранее объявленной файловой переменной с именем существующего или вновь создаваемого файла, а также в указании направления обмена информацией: чтение из файла или запись в него.

Файловая переменная связывается с именем файла в результате обращения к стандартной процедуре AssignFile :

```
AssignFile (<ф.п.>, <имя файла>);
```

где <ф.п.> – файловая переменная (правильный идентификатор, объявленный в программе как переменная файлового типа);
<имя файла > – текстовое выражение, содержащее имя файла и, если это необходимо, маршрут доступа к нему.

Инициировать файл означает указать для этого файла направление передачи данных. В Object Pascal можно открыть файл для чтения, для записи информации, а также для чтения и записи одновременно. Для чтения файл инициируется с помощью стандартной процедуры Reset:

```
Reset (<ф.п.>);
```

где <ф. п. > – файловая переменная, связанная ранее процедурой AssignFile с уже существующим файлом. При выполнении этой процедуры дисковый файл подготавливается к чтению информации. В результате специальная переменная-указатель, связанная с этим файлом, будет указывать на начало файла, т. е. на компонент с порядковым номером 0.

Чтобы исключить попытку открытия несуществующего файла, используют стандартную функцию FileExists:

```
begin
if FileExists(FileName) then
..... // Файл существует
else ..... // Файл не существует
end;
```

В Object Pascal разрешается обращаться к типизированным файлам, открытым процедурой Reset (для чтения информации), с помощью процедуры write (для записи информации). Такая возможность позволяет легко обновлять ранее созданные типизированные файлы и при необходимости расширять их. Для текстовых файлов, открытых процедурой Reset, нельзя использовать процедуру Write ИЛИ WriteLn.

Стандартная процедура

```
Rewrite (<ф.п.>);
```

инициирует запись информации в файл, связанный с файловой переменной <ф.п.>. Процедурой Rewrite нельзя инициировать запись информации в ранее существовавший дисковый файл: при выполнении этой процедуры старый файл (если он был) уничтожается и никаких сообщений об этом в программу не передается. Новый файл подготавливается к приему информации, и его указатель принимает значение 0.
Стандартная процедура

```
Append (<ф.п.>)
```

инициирует запись в ранее существовавший текстовый файл для его расширения, при этом указатель файла устанавливается в его конец. Процедура Append применима только к текстовым файлам, т. е. их файловая переменная должна иметь тип TextFile. Процедурой Append нельзя инициировать запись в типизированный или нетипизированный файл. Если текстовый файл ранее уже был открыт с помощью Reset или Rewrite, использование процедуры Append приведет к закрытию этого файла и открытию его вновь, но уже для добавления записей.

2.8.2. Работа с текстовыми файлами

Текстовый файл трактуется в Object Pascal как совокупность строк переменной длины. Доступ к каждой строке возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой строки ставится специальный признак eoln (End Of LiNe – конец строки), а в конце всего файла – признак eof (End Of File – конец файла). Эти признаки можно протестировать одноименными логическими функциями. При формировании текстовых файлов используются следующие системные соглашения.

Для доступа к записям применяются процедуры Read, ReadLn, write, writebn. Они отличаются возможностью обращения к ним с переменным числом фактических параметров, в качестве которых могут использоваться символы, строки и числа. Первым параметром в любой из перечисленных процедур должна стоять файловая переменная. Обращение осуществляется к дисковому файлу, связанному с переменной процедурой AssignFile.

Подпрограммы для работы с текстовыми файлами приведены в таблице 2.9.

Таблица 2.9

Function Eoln(var F: TextFile): Boolean;	Тестирует маркер конца строки и возвращает True, если конец строки достигнут
Procedure Read(var F: TextFile; V1 [,V2,...,Vn]);	Читает из текстового файла последовательность символьных представлений переменных Vi типа char. String, а также любого целого или вещественного типа, игнорируя признаки EOLN

Окончание таблицы 2.9

Procedure ReadLn(var F: TextFile; [VI [, V2, ...,Vn]]);	Читает из текстового файла последовательность символьных представлений переменных Vi типа char, String, а также любого целого или вещественного типа с учетом границ строк
Function SeekEof(var F:Text): Boolean;	Пропускает все пробелы, знаки табуляции и маркеры конца строки eoln до маркера конца файла eof или до первого значащего символа и возвращает True, если маркер eof обнаружен
Function SeekEoln (var F: TextFile): Boolean;	Пропускает все пробелы и знаки табуляции до маркера конца строки eoln или до первого значащего символа и возвращает True, если маркер обнаружен
Procedure Write(var F: Text; PI [, P2,..., Pn]) ;	Записывает символьные представления параметров Pi в текстовый файл
Procedure WriteLn (var F: Text; [PI [, P2, ..., Pn]]);	Записывает символьные представления параметров Pi и признак конца строки eoln в текстовый файл

Процедура Read предназначена для последовательного чтения из текстового файла символьных представлений переменных Vi. При чтении переменных типа char выполняется чтение одного символа и присваивание считанного значения переменной. Если перед выполнением чтения указатель файла достиг конца очередной строки, то результатом чтения будет символ cr (код #13), а если достигнут конец файла, то символ eof (код #26). Процедуру Read не рекомендуется использовать для ввода переменных типа string, т. к. она не способна “перепрыгнуть” через разделитель строк eoln и читает только первую строку текстового файла. Для ввода последовательности строк нужно использовать процедуру ReadLn.

При обращении к процедуре Read, за вводом очередного целого или вещественного числа процедура “перескакивает” маркеры конца строк, т. е. фактически весь файл рассматривается ею как одна длинная строка, содержащая текстовые представления чисел. В сочетании с проверкой конца файла функцией eof процедура Read позволяет организовать простой ввод массивов данных, например, так:

```
const
N = 1000; // Максимальная длина ввода
var
F : TextFile;
M : array [1..N] of Real;
```

```
i : Integer;
begin
AssignFile(F, 'prog.dat');
Reset(F);
i := 1;
while not EOF(F) and (i <= N) do
begin
Read(F, M[i]);
inc (i) end;
CloseFile(F) ;
end.
```

Пример записи информации в текстовый файл приведен ниже:

```
var
F : TextFile;
X : array [1..N] of Real;
i : Integer;
egin
AssignFile(F,'MyDat.txt');
Rewrite(F);//Перезапись файла
For i:=0 to N do
begin
Writeln(F,Floattostr(X[i])
end;
CloseFile(F);//Закрыть файл
```

Полное описание работы с файлами представлено в [8].

2.9. Пример программной реализации метода интерполяции функций сплайном третьего порядка на Delphi-6

Пусть в нашем случае задана исходная функция $Z = Q^2$, где $Q = \sqrt{X^2 + Y^2}$.

Число узловых точек и интервалов интерполяции равно 5.

Разместим на рабочей форме необходимые для проведения исследования кубического сплайна компоненты:

1. Компонент StringGrid «  » (закладка палитры компонентов Additional) представляет собой таблицу с данными, посредством которой будут вводиться значения функции в узловых точках. Используя инспектор объектов Object Inspector, установим количество столбцов RowCount и строк ColCount таблицы равное 5.

2. Компонент TabbedNotebook «  » (закладка палитры компонентов Win 3.1). В нашей программе он будет использоваться как группа закладок, содержащая графики интерполированной и заданной функций, а также график изменения ошибки интерполирования $\gamma = f(Q)$. Для добавления новых закладок необходимо в инспекторе объектов выбрать свойство Pages и в появившемся окне нажать кнопку Add.
3. Два компонента Chart «  » (закладка палитры компонентов Additional) для отображения графиков. Двойной щелчок мышью открывает диалоговое окно, в котором можно установить все необходимые параметры отображения графиков. Добавление новой графической зависимости $y = f(x)$ можно осуществить, нажав кнопку Add и предварительно выбрав закладку Chart/Series.

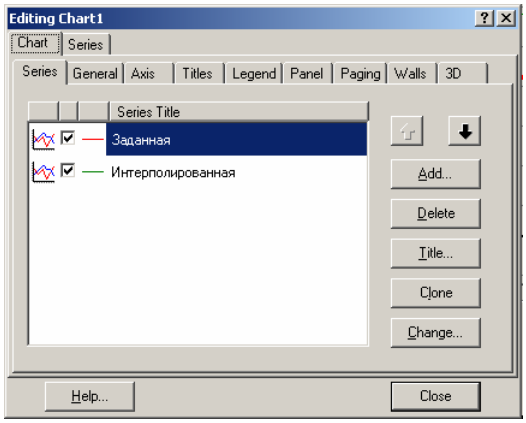


Рис. 2.9.1. Диалоговое окно ввода параметров компонента Chart

4. Два компонента GroupBox «  » (закладка палитры компонентов Standard) предназначены для объединения компонентов соответствующих параметрам и результатам расчета сплайн-функции.
5. Три компонента SpinEdit «  » (закладка палитры компонентов Samples) для ввода целочисленных данных. В нашем случае – количество интервалов интерполяции, число узлов исходной функции, количество циклов расчета функции построения сплайна для определения среднего времени интерполирования.

6. Компонент MainMenu «  » (закладка палитры компонентов Standard). Данный компонент представляет собой верхнее текстовое меню нашей формы. Вход в дизайнер меню осуществляется двойным щелчком левой кнопки мыши. Добавление или удаление элементов меню производится с помощью клавиш Ins и Del.

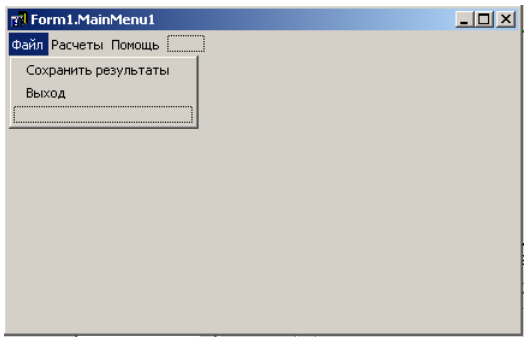


Рис. 2.9.2. Диалоговое окно дизайнера меню

7. Компонент SaveDialog «  » (закладка палитры компонентов Dialogs). Обеспечивает работу диалогового окна для сохранения и указания имени файла.
8. Компонент ProgressBar «  » (закладка палитры компонентов Win32). Этот элемент будет использован для отображения текущего состояния работы программы при анализе быстродействия.
9. Компонент CheckBox «  » (закладка палитры компонентов Standard), по средствам которого будем осуществлять выбор режима анализа быстродействия.
10. Несколько компонентов Label «  » (закладка палитры компонентов Standard). В нашей программе используем их для вывода строки текста на форму.

Внешний вид нашей программы после расстановки всех элементов на форме и указания их параметров показан на рисунке 2.9.3.

Приведенная выше последовательность действий по размещению элементов на форме в большей степени относится к разработке интерфейса нашей программы. Поэтому далее рассмотрим механизм обработки событий вызываемых при обращении к элементам формы, а также рассмотрим, как реализованы алгоритмы построения сплайн-функции и вычисления погрешности интерполирования.

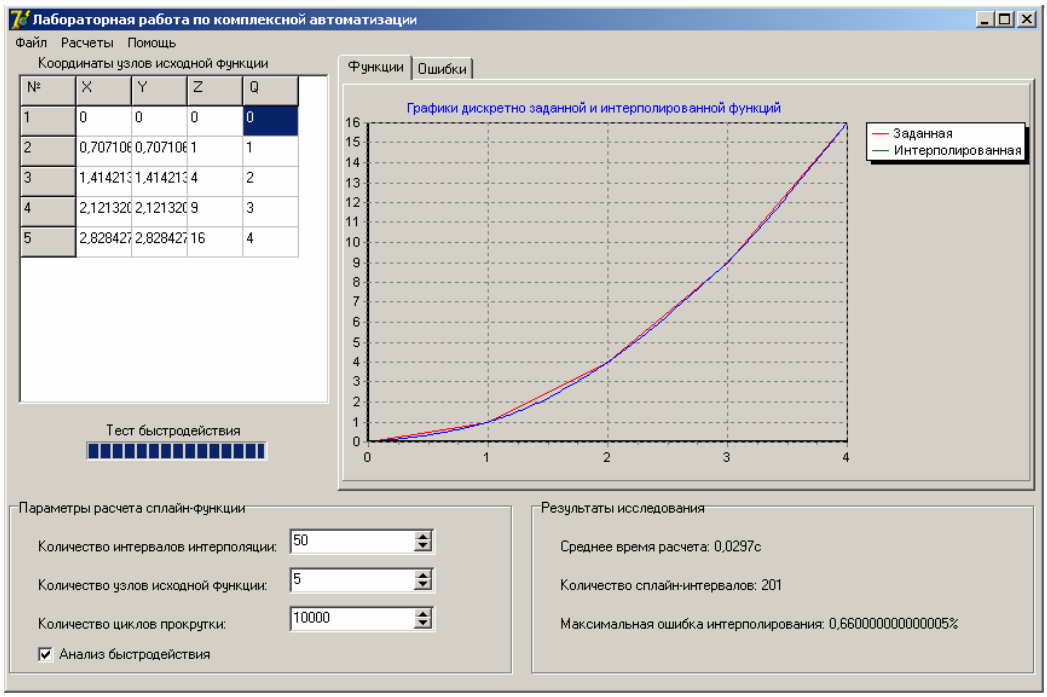


Рис. 2.9.3. Внешний вид разработанной программы

Сохраним текущую версию нашей программы с помощью команды меню File | Save Project As |. На запрос имени модуля (файл *.PAS) вводим Main, на запрос имени проекта (файл *.DPR) введем LaboratoryWork. Имя модуля и проекта не должны совпадать, иначе среда разработчика Delphi выдаст ошибку сохранения.

Поскольку для построения сплайна будем использовать уже написанную ранее функцию, то нам необходимо подключить содержащий ее модуль “SplineAlgorithm.pas”. Для этого откроем наш модуль Main в редакторе кода и допишем его в списке используемых модулей нашей программы:

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
Controls, Forms, Dialogs, Grids, DBGrids, Menus, Chartfx3,
VCFI, AxCtrls, OleCtrls, VCF1, ExtCtrls, TeeProcs, TeEngine,
Chart, StdCtrls, Series, Spin, ComCtrls, TeeFunci, TabNotBk,
about, SplineAlgorithm;
```

Помимо этого нам потребуется добавить в модуль еще две функции. Первая будет определять значение реальной функции (заданной в таблице (4.1.)) в требуемой точке по оси Q. Вторая будет находить соответ-

ствующие значения координат X и Y в зависимости от координаты Q по формуле (2.8.1).

```
implementation

{$R *.dfm}

Function MyFunction(Q:Real):Real;
//  Находим значение реальной функции в точке Q
begin
  Result:=Sqr(Q);
end;

Function DefineXY(Q:Real):Real;
// Находим значение координат X=Y
begin
  Result:=sqrt(sqr(Q)/2);
end;
```

Установим значение числа узловых точек по умолчанию равное 5 в компонент SpinEdit3. Для этого, выделив его левой кнопкой мыши, установим в инспекторе объектов (см. рис. 2.9.4) значение Value=5.

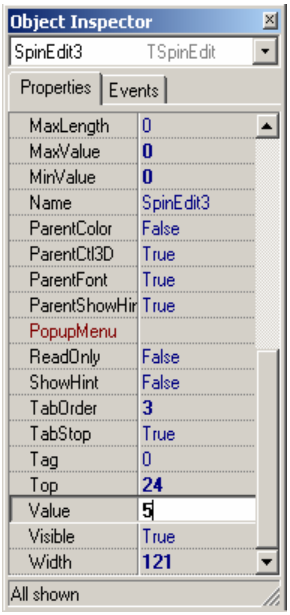


Рис. 2.9.4. Инспектор объектов

Аналогично установим значение компонента SpinEdit2, отвечающего за значение количества циклов прокрутки равное 10000.

Как и в предыдущем случае, установим по умолчанию значение числа узловых точек компонента SpinEdit1 равное 5. Однако помимо

этого следует также учесть, что с изменением числа узловых точек таблица с данными (компонент StringGrid1) должна автоматически устанавливать число строк равное величине SpinEdit1.Value. Для этого нужно войти в процедуру обработки событий компонента SpinEdit1, которая автоматически создается двойным щелчком левой кнопки мыши на данном компоненте, и внести соответствующие изменения:

```
procedure TForm1.SpinEdit1Change(Sender: TObject);
// Установка числа строк в таблице
begin
//Присвоить число строк таблицы равное SpinEdit1.Value+1
  StringGrid1.RowCount:=SpinEdit1.Value+1;
//Пронумеруем первый столбец
  For i:=1 to SpinEdit1.Value+1 do
  StringGrid1.Cells[0,i]:=FloatToStr(i);
end;
```

Далее создадим обработку событий от текстового меню. Рассмотрим пункт меню Файл | Сохранить результаты:

```
procedure TForm1.N4Click(Sender: TObject);
// Запись входного и выходного списка точек в текстовый файл.
begin
// Указать компоненту SaveDialog1 расширение
// сохраняемого файла: XLS.
  Form1.SaveDialog1.Filter:='xls';
// Проверка – расчеты выполнены или нет?
  if Calculated then //Расчеты выполнены
  If  Form1.SaveDialog1.Execute  then  //Нажата  кнопка  сохра-
нить
  begin
// Связать переменную F с указанным файлом.
    AssignFile(F,Form1.SaveDialog1.FileName+'.xls');
{$I-} //Директива компилятора
// Перезаписать файл F, если он есть, или создать новый, ес-
ли нет.
    Rewrite(F);
// Добавить заголовок входного списка точек.
WriteLn(F,'InputPoints');
// Добавить строку в файл с именами столбцов
    WriteLn(F,'X'+'              '+'Y'+'              '+'Z'+'
'+'Q');
// В цикле добавляем в файл все точки дискретно заданной
// функции.
    For i:=0 to Length(InputQZ.X)-1 do
    begin
```

```
      Writeln(F,Floattostr(InputXY.X[i])+'
'+Floattostr(InputXY.Y[i])+'
'+Floattostr(InputQZ.Y[i])+'
'+Floattostr(InputQZ.X[i]));
    end;
// Добавить строку с именем выходного списка точек.
    WriteLn(F,'OutputPoints');
    WriteLn(F,'X'+'              '+'Y'+'              '+'Z'+'
'+'Q'+'             '+'Delta');
// В цикле добавляем в файл все точки интерполированной
// траектории, а также ошибку интерполяции.
    For i:=0 to Length(OutputQZ.X)-1 do
    begin
      Writeln(F,Floattostr(OutputXY.X[i])+'
'+Floattostr(OutputXY.Y[i])+'
'+Floattostr(OutputQZ.Y[i])+'
'+Floattostr(OutputQZ.X[i])+'
'+Floattostr(ErrorValue.Y[i]));
    end;
   CloseFile(F);//Закрываем открытый на запись файл.
{$I+}
  end;
end;
```

Пункт меню Файл | Выход:

```
procedure TForm1.N5Click(Sender: TObject);
// Процедура выхода из программы.
begin
  Close; //Закрыть программу.
end;
```

Пункт меню Расчеты | Рассчитать файл:

```
procedure TForm1.N7Click(Sender: TObject);
// Вызов процедуры расчета сплайна и построения графиков
// на компонентах Chart1 и Chart2.
Var FullTime, Timer,k:Integer;
    MaxFuncVal:Real;
begin
// Устанавливаем длину динамических массивов (поля X и Y)
// записей InputXY и InputQZ.
  SetLength(InputXY.X,SpinEdit1.Value);
  SetLength(InputXY.Y,SpinEdit1.Value);
  SetLength(InputQZ.X,SpinEdit1.Value);
  SetLength(InputQZ.Y,SpinEdit1.Value);
// Очищаем список точек компонентов Chart1 и Chart2
// т.е. очистка графиков.
```

```
  Form1.Chart1.SeriesList[0].Clear;
  Form1.Chart1.SeriesList[1].Clear;
  Form1.Chart2.SeriesList[0].Clear;
  For i:=0 to SpinEdit1.Value-1 do
  begin
// Считывание данных из 3-го столбца таблицы в массив
InputQZ.Y.
    InputQZ.Y[i]:=StrToFloat(StringGrid1.Cells[3,i+1]);
// Считывание данных из 4-го столбца таблицы в массив
InputQZ.X.
    InputQZ.X[i]:=StrToFloat(StringGrid1.Cells[4,i+1]);
// Определение соответствующих точек на осях X и Y.
    InputXY.X[i]:=DefineXY(InputQZ.X[i]);
    InputXY.Y[i]:=InputXY.X[i];
// Заносим данные (координата X) в первый столбец таблицы
    StringGrid1.Cells[1,i+1]:=FloatToStr(InputXY.X[i]);
// Заносим данные (координата Y) во второй столбец таблицы
    StringGrid1.Cells[2,i+1]:=FloatToStr(InputXY.Y[i]);
// Ввод точек дискретно заданной функции в компонент Chart1
// и вывод их на рабочее поле графика.
Form1.Chart1.SeriesList[0].AddXY(InputQZ.X[i],InputQZ.Y[i],'
',clred);
  end;
// Считать данные (число интервалов интерполяции) из компо-
нента
// SpinEdit3 в переменную AmountIntervals.
  AmountIntervals:=SpinEdit3.Value;
// FullTime – переменная, отвечающая за полное время расчета
// при анализе быстродействия
  FullTime:=0;
// Проверка - установлен ли анализ быстродействия.
// Если да, то переменной k присвоить значение количества
// циклов (повторного) пересчета. В противном случае количе-
ство
// пересчетов равно 1.
  if CheckBox1.Checked then k:=SpinEdit2.Value Else k:=1;
// Установить максимальное значение компонента Progressbar1
// равное k. Progressbar1 отвечает за визуальное отображение
// процесса расчета быстродействия.
  Progressbar1.Max:=k;
// Цикл для расчета быстродействия.
  for i:=1 to k do
  begin
// Засечь текущее время.
    Timer:=GetTickCount;
// Если k<>1, то установить текущее положение компонента
// Progressbar1 равное i
    if k<>1 then Progressbar1.Position:=i;
```

```
// Произвести расчет сплайна с помощью вызова функции
CSpline
// из модуля SplineAlgorithm.pas.
    OutputQZ:=CSpline(InputQZ,AmountIntervals);
// Суммировать время расчета сплайна на текущем цикле
// расчета с предыдущим.
  FullTime:=FullTime+(GetTickCount-timer);
  end;
  Form1.Label5.Caption:='Среднее время расчета:
'+FloatToStr(FullTime/k)+'c';//Вывод текста на форму.
  Form1.Label6.Caption:='Количество сплайн-интервалов:
'+FloatToStr(Length(OutputQZ.X));
// Максимальное значение исходной функции.
  MaxFuncVal:=ABS(MyFunction(OutputQZ.X[0]));
// Установить длину динамических массивов (поля X и Y)
// записи OutputXY равную длине соответствующих массивов за-
писи
// OutputQZ.
  SetLength(OutputXY.X,Length(OutputQZ.X));
  SetLength(OutputXY.Y,Length(OutputQZ.X));
  for i:=0 to Length(OutputQZ.X)-1 do
  begin
// Ввод точек интерполированной функции в компонент Chart1
// и вывод их на рабочее поле графика.
Form1.Chart1.SeriesList[1].AddXY(OutputQZ.X[i],OutputQZ.Y[i]
,'',clBlue);
// Определение соответствующих координат точек интерполиро-
ванной
// функции на плоскости XY
    OutputXY.X[i]:=DefineXY(OutputQZ.X[i]);
    OutputXY.Y[i]:=OutputXY.X[i];
// Определение максимальной величины интерполированной функ-
ции.
    if MaxFuncVal<ABS(MyFunction(OutputQZ.X[i])) then
MaxFuncVal:=ABS(MyFunction(OutputQZ.X[i]));
  end;
// Установить длину массивов X и Y записи ErrorValue равной
// длине  OutputQZ соответствующих массивов.
  SetLength(ErrorValue.X,Length(OutputQZ.X));
  SetLength(ErrorValue.Y,Length(OutputQZ.X));
// Присвоить соответствующие значения координат по оси абс-
цисс.
  ErrorValue.X:=OutputQZ.X;
  MaxError:=0; // Максимальная ошибка.
  For i:=0 to Length(OutputQZ.X)-1 do
  begin
// Величина ошибки в текущей точке по оси Q.
```

```
    ErrorValue.Y[i]:=ABS((Myfunction(ErrorValue.X[i]))-
OutputQZ.Y[i]);
// Вывести текущую точку на график.
Form1.Chart2.SeriesList[0].AddXY(ErrorValue.X[i],ErrorValue.
Y[i],'',clBlack);
// Если ранее найденная максимальная ошибка меньше текущей,
то
// присвоить ее значение текущей ошибке.
    if       MaxError<ErrorValue.Y[i]       then       MaxEr-
ror:=ErrorValue.Y[i];
  end;
  MaxError:=MaxError/MaxFuncVal;
// Вывести максимальную ошибку интерполирования на компонент
// формы Label7
  Form1.Label7.Caption:='Максимальная ошибка интерполирова-
ния: '+FloatToStr(MaxError*100)+'%';
// Расчет выполнен.
  Calculated:=True;
end;
```

Для анализа быстродействия в приведенной процедуре используется функция GetTickCount. Поскольку эта функция обладает достаточно большой погрешностью при измерении малых интервалов времени и реальная работа программ в среде Windows происходит с непостоянной скоростью, то предлагается прокрутить выполнение алгоритма расчета сплайна n-ое (в нашем случае n=10000) количество раз, а затем вычислить среднее время расчета.

2.10. Пример реализации виртуального диспетчерского пульта

В качестве примера виртуального диспетчерского пульта рассмотрим программу управления лабораторным стендом «МИКРОКОН», разработанную Научно-производственной фирмой «Ключ-1» [9].

Внешний вид программы представлен на рисунке 2.9.5. На рабочем поле программы представлены два основных визуальных элемента – панель управления и графическая мнемосхема стенда. Панель управления содержит все необходимые компоненты для работы со стендом, такие как выбор формы и уровня задающего сигнала на регулируемый электропривод, нагрузка, сохранение полученных результатов в журнал, настройка порта связи персонального компьютера с контроллером, справка о работе с программой.

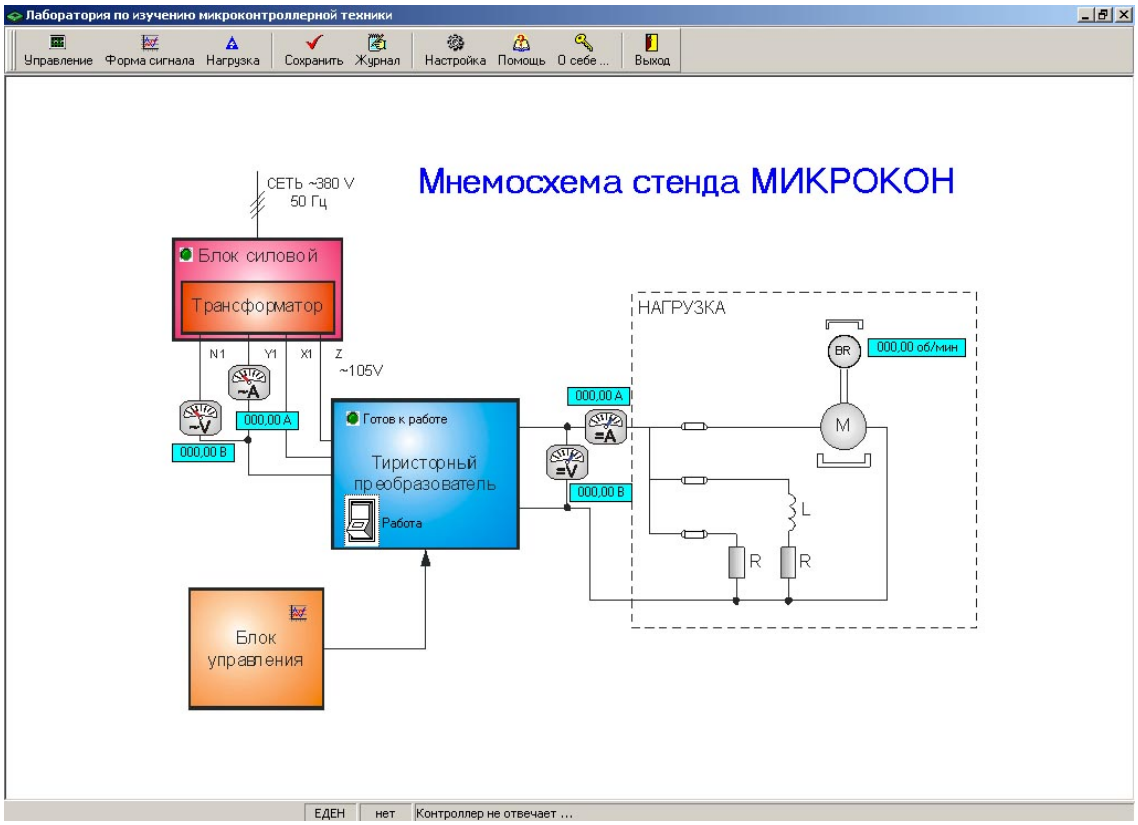


Рис. 2.9.5. Внешний вид программы управления стендом «МИКРОКОН»

Мнемосхема содержит графическое представление элементов лабораторной установки (трансформатор, тиристорный преобразователь, двигатель), строки отображения значений скорости, тока и напряжения в реальном времени. А также позволяет переключать тип нагрузки, что значительно упрощает работу оператора.

Выбор формы задающего сигнала наиболее удобно выбирать, щелкнув левой кнопкой мыши на «Блок управления», расположенный на мнемосхеме. При вызове данной команды вызывается новое окно, содержащее графическую визуализацию формы задающих сигналов (рис. 2.9.6).

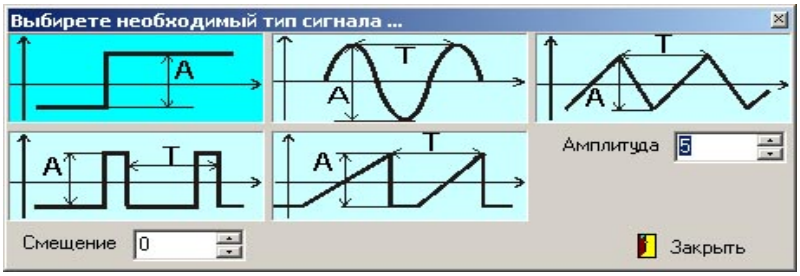


Рис. 2.9.6. Окно выбора задающих сигналов

Результаты работы могут быть сохранены в базу данных для последующей работы с ними. Окно «Журнала записей» представлено на рисунке 2.9.7.

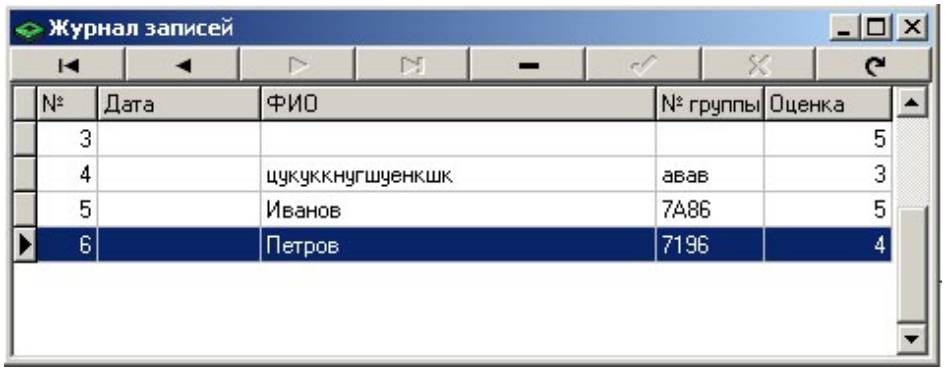


Рис. 2.9.7. Окно журнала записей

Переход к сохраненным данным производится вызовом соответствующей строки «Журнала записей». После открытия требуемой записи происходит вызов окна содержащего сохраненные временные графики скорости, тока и напряжения. Визуальный компонент Delphi, отображающий графики, позволяет также выделить опорные точки, что помогает оценить дискретность считывания информации с датчиков.

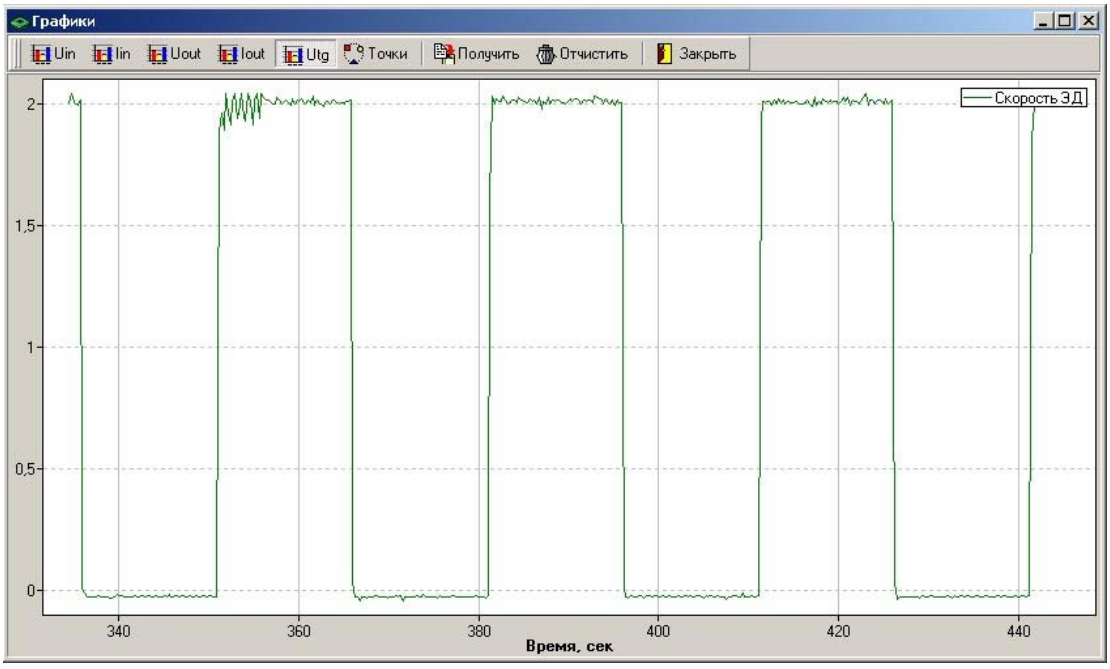


Рис. 2.9.8. Временные графики

Выбор параметров связи с персональным компьютером осуществляется указанием используемого COM-порта (рис 2.9.9).

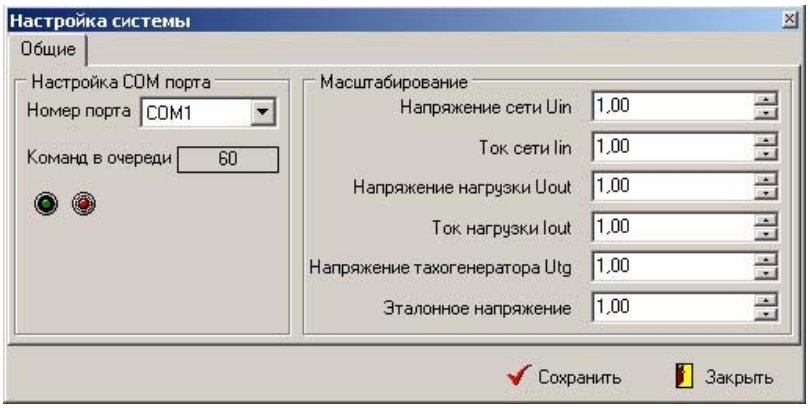


Рис. 2.9.9. Настройки системы

Как видно из приведенного примера, среда разработки программного обеспечения Delphi-6 позволяет создать программный комплекс для управления, сбора, хранения и обработки данных с присущими свойствами всем современным SCADA системам. К преимуществам использования Delphi следует отнести гибкость настройки разрабатываемого программного обеспечения под требования заказчика, наличие мощных средств визуального программирования, простота работы с базами данных. К основным недостаткам применения Delphi следует отнести сложность написания драйверов под нестандартные устройства, а также значительные временные затраты на создание конечной версии рабочей программы.

ПРИЛОЖЕНИЕ 1

Таблица П1

Тригонометрические подпрограммы	
function ArcCos(X: Extended): Extended;	Арккосинус
function ArcSin(X: Extended): Extended;	Арксинус
function ArcTan2(Y, X: Extended): Ex tended;	Вычисляет арктангенс Y/X и возвращает угол в правильном квадранте (функция АгсТап модуля System не учитывает квадрант)
function Cotan(X: Extended): Extended;	Котангенс
function Hypot (X, Y: Extended): Extended;	Корень квадратный из (X** 2 + Y** 2) – гипотенуза прямоугольного треугольника по двум катетам
procedure SinCos (Theta: Extended; va: "'Sin, Cos: Extended);	Возвращает одновременно синус и косинус угла Theta (почти в 2 раза быстрее, чем раздельное получение синуса и косинуса)
function Tan(X: Extended): Extended;	Тангенс
Функции преобразования углов	
function CycleToRad(Cycles: Extended) : Extended;	Radians := Cycles * 2PI
function DegToRad(Degrees: Extended) Extended;	Radians := Degrees * PI / 18
function GradToRad(Grads: Extended): Extended;	Radians := Grads * PI / 200
function RadToDeg(Radians: Extended) Extended;	Degrees := Radians * 180 / PI
function RadToGrad(Radians: Extended) : Extended	Grads := Radians * 200 / PI
function RadToCycle(Radians: Extended) : Extended;	Cycles := Radians / 2PI
Гиперболические функции	
function ArcCosh(X: Extended): Extended;	Гиперболический арккосинус
function ArcSinh(X: Extended): Extended;	Гиперболический арксинус
function ArcTanh(X: Extended): Extended;	Гиперболический арктангенс
function Cosh(X: Extended): Extended;	Гиперболический косинус
function Sinh(X: Extended): Extended;	Гиперболический синус
function Tanh(X: Extended): Extended;	Гиперболический тангенс
Логарифмические функции	
Function LnXP1 (X: Extended) : Exteaded;	Логарифм натуральный от (X+1). Используется, когда X близок к нулю
functios Extended): Extended;	Десятичный логарифм
functior extended): Extended;	Двоичный логарифм

Продолжение таблицы П1

function LogN(Base, X: Extended): Extended;	Логарифм от X при основании Base
Экспоненциальные функции	
Function IntPower(Base: Extended; Expopient: Integer) : Extended;	Возведение Base в целочисленную степень Exponent
Function Power(Base, Exponent: Extended) : Extended;	Возведение Base в вещественную степень Exponent
Подпрограммы разного назначения	
Function Ceil(X: Extended): Integer;	Ближайшее меньшее целое
Function Floor (X: Extended): Integer;	Ближайшее большее целое
procedure Frexp(X: Extended; var Mantissa: Extended; var Exponent: Integer);	Возвращает мантиссу и степень вещественного числа
Function Ldexp(X: Extended; P: Integer) : Extended;	Возвращает X*P*P
Статические программы	
Function Max(A,B: Int64): Int64; overload;	
Function Max (A, B: Integer): Integer; overloads;	Возвращает максимальное из двух чисел
Function Poly(X: Extended; const Coefficients: array of Double): Extended;	Значение полинома A*X"+B*X" '+.. .+Z. Коэффициенты задаются в порядке возрастания степени
function Max(A,B: Single): Single/overload;	
function Max(A,B: Double): Double/overload;	
function Max(A,B: Extended): Extended; overload;	
function MaxIntValue(const Data: array of Integer): Integer;	Возвращает максимальное из набора целых чисел
function MaxValue(const Data: array of Double): Double;	Возвращает максимальное из набора вещественных чисел
function Mean(const Data: array of Double): Extended;	Вычисляет арифметическое среднее для набора вещественных чисел
procedure MeanAndStdDev(const Data: array of Double; var Mean, StdDev: Extended) ;	Вычисляет арифметическое среднее и стандартное отклонение для набора вещественных чисел
function Min(A/B: Integer): Integer/overload/ function Min(A,B: Int64): Int64; overload;	Возвращает минимальное из двух чисел
function Min(A,B: Single): Single/overload; function Min(A,B: Double): Double; overload/ function Min(A,B: Extended): Extended; overload/	

Продолжение таблицы П1

function MinIntValue(const Data: array of Integer): Integer;	Возвращает минимальное из набора целых чисел
function MinValue(const Data: array of Double): Double;	Возвращает минимальное из набора вещественных чисел
procedure MomentSkewKurtosis (const Data: array of Double; var Ml, M2, M3, M4, Skew, Kurtosis: Extended);	Вычисляет статистические моменты порядков с первого по четвертый, а также асимметрию Skew и эксцесс Kurtosis для набора чисел
function Norm(const Data: array of Double): Extended;	Возвращает норму (квадратный корень из суммы квадратов) вещественных чисел
function PopnStdDev(const Data: array of Double): Extended;	Выборочное стандартное отклонение. Отличается от обычного стандартного отклонения тем, что использует выборочное значение дисперсии (см.ниже PopnVariance)
function PopnVariance(const Data: array of Double): Extended;	Выборочная дисперсия. Использует “смещенную” формулу TotalVariance/N (см. ниже TotalVariance)
function RandG(Mean, StdDev: Extended) : Extended;	Генерирует нормально псевдораспределенную последовательность чисел с заданным средним значением Mean и стандартным отклонением StdDev
function StdDev(const Data: array of Double): Extended;	Вычисляет среднеквадратическое отклонение для набора чисел
function Sum(const Data: array"orDouble): Extended register;	Вычисляет сумму чисел
procedure SumsAndSquares(const Data: array of Double;	Одновременное вычисление суммы и суммы квадратов для набора чисел
function Sumint(const Data: array of Integer): Integer register;	Сумма набора целых чисел
function SumOfSquares(const Data: array of Double): Extended;	Сумма квадратов чисел
function TotalVariance(const Data: array of Double): Extended;	Сумма квадратов расстояний всех величин от их среднего арифметического
function Variance(const Data: array of Double): Extended;	Выборочная дисперсия для набора чисел. Использует “несмещенную” формулу TotalVariance/(N–1)
Финансовые функции	
type TPaymentTime = (ptEndOfPeriod, ptStartOfPeriod) ;	Перечисляемый тип, используемый в финансовых функциях
function DoubleDecliningBalance (Cost, Salvage: Extended; Life, Period: Integer): Extended;	Вычисление амортизации методом двойного баланса

Окончание таблицы П1

function FutureValue(Rate: Extended; NPeriods: Integer; Payment, PresentValue: Extended; PaymentTime: TPaymentTime): Extended;	Будущее значение вложения
function InterestPayment(Rate: Extended; Period, NPeriods: Integer; PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended;	Вычисление процентов по ссуде
function InterestRate(NPeriods: Integer; Payment, PresentValue, Future-Value: Extended; PaymentTime: TPaymentTime) : Extended;	Норма прибыли, необходимая для получения заданной суммы
function InternalRateOfReturn (Guess: Extended} const CashFlows: array of Double): Extended;	Вычисление внутренней скорости оборота вложения для ряда последовательных выплат
function NetPresentValue(Rate: Extended; const CashFlows: array of Double; PaymentTime: TPaymentTime): Extended;	Вычисление чистой текущей стоимости вложения для ряда последовательных выплат с учетом процентной ставки
function NumberOfPeriods(Rate, Payment, PresentValue, FutureValue: Extended; PaymentTime: TPaymentTime): Extended/	Количество периодов, за которые вложение достигнет заданной величины
function Payment(Rate: Extended; NPeriods: Integer; PresentValue, Future-Value: Extended; PaymentTime: TPayment-Time) : Extended/	Размер периодической выплаты для погашения ссуды при заданном числе периодов, процентной ставке, а также текущем и будущем значениях ссуды
function PeriodPayment(Rate: Extended; Period, NPeriods: Integer; PresentValue, FutureValu'e: Extended; PaymentTime: TPaymentTime): Extended;	Платежи по процентам за заданный период
function PresentValue(Rate: Extended; NPeriods: Integer; Payment, Future-Value: Extended; PaymentTime: TPayment-Time) : Extended;	Текущее значение вложения
function SLNDepreciation (Cost, Salvage: Extended; Life: Integer): Extended;	Вычисление амортизации методом постоянной нормы
function SYDDepreciation (Cost, Salvage: Extended; Life, Period: Integer) : Extended	Вычисление амортизации методом весовых коэффициентов

ПРИЛОЖЕНИЕ 2

Таблица П2

Function AnsiLowerCase(const S: String): String;	Возвращает исходную строку S, в которой все прописные буквы заменены на строчные в соответствии с национальной кодировкой Windows (т. е. с учетом кириллицы)
Function AnsiUpperCase(const S: String): String;	Возвращает исходную строку s, в которой все строчные буквы заменены на прописные в соответствии с национальной кодировкой Windows
Function Concat(Sl [, S2, ..., SN]: String): String;	Возвращает строку, представляющую собой сцепление строк-параметров S1, S2, ... , SN
Function Copy(St: String; Index, Count: Integer): String;	Копирует из строки St count символов, начиная с символа с номером Index
Procedure Delete(St: String; Index, Count: Integer);	Удаляет count символов из строки St, начиная с символа с номером index
Procedure Insert(SubSt:String; St, Index: Integer);	Вставляет подстроку SubSt в строку St, начиная с символа с номером Index
Function Length(St: String): Integer;	Возвращает текущую длину строки St
Function LowerCase(const S:String): String;	Возвращает исходную строку S, в которой все латинские прописные буквы заменены на строчные
procedure OleStrToStrVar(Source: PWideChar; var Dest:String) ;	Копирует “широкую” (двухбайтную) строку в обычную строку Object Pascal
Function Pos(SubSt, St:String): Integer;	Отыскивает в строке St первое вхождение подстроки SubSt и возвращает номер позиции, с которой она начинается. Если подстрока не найдена, возвращается ноль
Procedure SetLength(St:String; New-Length: Integer);	Устанавливает новую (меньшую) длину NewLength строки St. Если NewLength больше текущей длины строки, обращение к SetLength игнорируется
function StringOfChar(Ch:Char; Count: Integer):String;	Создает строку, состоящую из Count раз повторенного символа ch
function StringToOleStr(const Source: String):PWideChar;	Копирует обычную строку в двухбайтную
function StringToWideChar(const Source: String; Dest:PWideChar; DestSize: Integer) : PWideChar;	Преобразует обычную строку в строку с символами UNICODE
function Uppercase(const S:String): String	Возвращает исходную строку S, в которой все строчные латинские буквы заменены на прописные

Продолжение таблицы П2

Подпрограммы преобразования строк в другие типы	
Function StrToCurr(St: String): Currency;	Преобразует символы строки St в целое число типа Currency. Строка не должна содержать ведущих или ведомых пробелов
Function StrToDate(St: String): TDateTime;	Преобразует символы строки St в дату. Строка должна содержать два или три числа, разделенных правильным для Windows разделителем даты (в русифицированной версии таким разделителем является “.”). Первое число – правильный день, второе – правильный месяц. Если указано третье число, оно должно задавать год в формате XX или XXXX. Если символы года отсутствуют, дата дополняется текущим годом. Например: DateToStr(StrToDate('28.06')) даст строку '28.06.99' (см. ниже пояснения)
Function StrToDateTime(St: String): TDateTime;	Преобразует символы строки St в дату и время. Строка должна содержать правильную дату (см. StrToDate) и правильное время (см. StrToTime), разделенные пробелом, например: StrToDateTime('28.06 18:23')
Function StrToCurr(St: String): Currency;	Преобразует символы строки St в целое число типа Currency. Строка не должна содержать ведущих или ведомых пробелов
Function StrToFloat(St:String): Extended;	Преобразует символы строки St в вещественное число. Строка не должна содержать ведущих или ведомых пробелов
Function StrToInt(St:String): Integer;	Преобразует символы строки St в целое число. Строка не должна содержать ведущих или ведомых пробелов
Function StrToIntDef(St:String; Default: Integer):Integer;	Преобразует символы строки St в целое число. Если строка не содержит правильного представления целого числа, возвращается значение Default

Продолжение таблицы П2

Function StrToIntRange(St:String; Min, Max: Longint):Lomgint;	Преобразует символы строки St в целое число и возбуждает исключение ERangeError, если число выходит из заданного диапазона Min.. .max
Function StrToTime(St:String): TDateTime;	Преобразует символы строки St во время. Строка должна содержать два или три числа, разделенных правильным для Windows разделителем времени (для русифицированной версии таким разделителем является “:”). Числа задают часы, минуты и, возможно, секунды. За последним числом через пробел могут следовать символы “am” или “pm”, указывающие на 12-часовой формат времени
Procedure Val(St: String; var X; Code: Integer);	Преобразует строку символов St во внутреннее представление целой или вещественной переменной x, которое определяется типом этой переменной. Параметр Code содержит ноль, если преобразование прошло успешно, и тогда в x помещается результат преобразования, в противном случае он содержит номер позиции в строке St, где обнаружен ошибочный символ, и в этом случае содержимое x не меняется. В строке St могут быть ведущие и/или ведомые пробелы. Если St содержит символьное представление вещественного числа, разделителем целой и дробной частей должна быть точка, независимо от того, каким символом этот разделитель указан в Windows
Подпрограммы обратного преобразования	
Function DateTimeToStr(Value: TDateTime): String; Procedure DateTime-ToString(var St: String; Format: String;-Value: TData-Time);	Преобразует дату и время из параметра в строку символов. Преобразует дату и время из параметра value в строку St в соответствии со спецификаторами параметра Format (см. пояснения ниже)
Function DateToStr(Value: TDateTime): String;	Преобразует дату из параметра value в строку символов
Function FloatToStr(Value: Extended): String;	Преобразует вещественное значение value в строку символов

Окончание таблицы П2

Function FloatToStrF(Value:Extended; Format: TFloatPormat; Precision, Digits:Integer) : String;	Преобразует вещественное значение Value в строку символов с учетом формата Format и параметров precision и Digits (см. пояснения ниже).
Function Format(const Format: String; const Args: array of const): Strings;	Преобразует произвольное количество аргументов открытого массива Args в строку в соответствии с форматом Format (см. пояснения ниже)
Function FormatDateTime (Format: String; Value:.TDateTime): String;	Преобразует дату и время из параметра value в строку символов в соответствии со спецификаторами параметра Format (см. пояснения ниже)
Function FormatFloat(Format:String; Value: Extended): String;	Преобразует вещественное значение value в строку символов с учетом спецификаторов формата Format (см. пояснения ниже)
function IntToHex(Value: Integer; Digits: Integer):Strings;	Преобразует целое число Value в строку символьного представления шестнадцатеричного формата: Digits – минимальное количество символов в строке
Function IntToStr(Value: Integer) : String;	Преобразует целое значение Value в строку символов
Procedure Str(X [:Width[:Decimals]]; var St:String) ;	Преобразует число x любого вещественного или целого типов в строку символов St; параметры width и Decimals, если они присутствуют, задают формат преобразования: width определяет общую ширину поля, выделенного под соответствующее символьное представление вещественного или целого числа x, а Decimals – количество символов в дробной части (этот параметр имеет смысл только в том случае, когда x – вещественное число)
Function TimeToStr(Value: TDateTime): String;	Преобразует время из параметра Value в строку символов

СПИСОК ЛИТЕРАТУРЫ

1. Архангельский А.Я. Приемы программирования в Delphi. Версии 5–7. – М.: Бином, 2003. –784 с.
2. Архангельский А.Я. Разработка прикладных программ для Windows в Delphi 5. – М.: Бином, 1999. –256 с.
3. Бобровский С. И. Delphi 7: Учебный курс . – СПб.: Питер, 2003. – 736 с.
4. Жуков А. Изучаем Delphi. – СПб.: Питер, 2002. – 352 с.
5. Карпов Б. Delphi: Специальный справочник. – СПб.: Питер, 2001. –688 с.
6. Курченко В. Тонкости программирования на Delphi. –М.: Познавательная книга, 2000. –192 с.
7. Понамарев В. А. Delphi 7 Studio: Самоучитель. – СПб.: БХВ-Петербург, 2003. –502 с.
8. Фаронов В.В. Delphi 6. Учебный курс. – СПб.: Питер, 2002. –512 с.
9. Http://key1.boom.ru.

ОГЛАВЛЕНИЕ

Виктор Григорьевич Букреев
Николай Владимирович Гусев

Delphi-6 – среда разработки программного обеспечения
для систем промышленной автоматизации

Учебное пособие

Редактор О.Н. Свинцова

Подписано к печати 30.07.2004.
Формат 60x84/16. Бумага офсетная.
Печать RISO. Усл. печ. л. 6,16 . Уч.-изд. л. 5,58.
Тираж 100 экз. Заказ . Цена свободная.
Издательство ТПУ. 634050, Томск, пр. Ленина, 30.