HIGH TECH

Delphi

Профессиональное программирование

Дмитрий Осипов

СИМВОЛ®

Санкт-Петербург — Москва
2006

Серия «High tech»

Дмитрий Осипов

Delphi. Профессиональное программирование

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*А. Петухов*
Художник	*В. Гренда*
Корректор	*О. Макарова*
Верстка	*Н. Гриценко*

Осипов Д.
Delphi. Профессиональное программирование. – СПб.: Символ-Плюс, 2006. – 1056 с., ил.
ISBN 5-93286-074-X

Книга Д. Осипова «Delphi. Профессиональное программирование» принципиально отличается от стандартных изданий на эту тему. Это и не скороспелое «полное» руководство по очередной версии Borland® Delphi™, и не рядовой справочник, содержащий перевод файлов помощи к среде программирования. Идея книги в другом. Автор системно и последовательно излагает концепцию Delphi, предоставляя читателю не просто инструмент, а профессиональную методику, позволяющую разрабатывать эффективные приложения для Windows.

Книга рассчитана на подготовленного пользователя ПК, желающего самостоятельно научиться программировать и разрабатывать приложения и базы данных в среде Delphi. Опытные программисты смогут использовать издание как справочник. В тексте подробно описаны более 80 компонентов VCL, функции Object Pascal и Win32 API. В первой части книги излагаются основы языка программирования Delphi, подробно рассматриваются библиотека визуальных компонентов и процесс разработки собственных компонентов, изучаются динамически подключаемые библиотеки, процессы, многопоточные приложения, особенности межпрограммного взаимодействия, программирование на Win32 API, особенности построения сетевого программного обеспечения, технологии COM и OLE-automation. Вторая часть книги посвящена проектированию и созданию реляционных баз данных. Рассматриваются реляционная модель данных и язык SQL, изучаются компоненты доступа к данным и отображения данных, базирующиеся на механизмах BDE, ADO и InterBase.

ISBN 5-93286-074-X

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 30.03.2006. Формат 70x100 1/16 . Печать офсетная.
Объем 66 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Введение

Не многие области науки могут похвастаться таким бурным развитием, какое претерпели за свою сравнительно недолгую историю существования электронно-вычислительная техника и шагающие с ней рука об руку языки программирования. Не так давно самые первые программы писались на языке машинных команд. Это был поистине каторжный труд. Программист тех старозаветных времен не просто знал язык первых машин, он обладал глубокими инженерными знаниями архитектуры электронно-вычислительной машины (ЭВМ), системы команд процессора, организации памяти и многого другого. Такой высококлассный специалист ценился на вес золота, а производительность его работы была до смешного мала. Процесс создания элементарной программы отдаленно напоминал шаманские обряды (кто видел перфоратор, тот меня поймет), а про программистов слагались легенды.

Такое положение вещей мало кого устраивало, посему учеными предпринимались активные попытки хотя бы в какой-то степени «очеловечить» язык машин. Первым успехом в этом направлении было создание компиляторов с языков ассемблера. Язык низкого уровня ассемблер по-прежнему был очень близок к машинным командам, однако в нем уже отдаленно просматривались и человеческие черты. В ассемблере машинным командам соответствовали англоязычные мнемонические коды. Синтаксис языка не отличался особой изысканностью – каждая команда ассемблера могла включать три элемента: поле метки, код операции и поле операндов. Не так густо, но по сравнению с машинными командами это был настоящий прорыв.

Хотя появление ассемблера и соответствующих компиляторов несколько упростило работу программиста, но, по сути, язык мнемокодов все еще значительно отличался от языка общения людей. Однако, без всякого сомнения, можно утверждать, что ассемблер некоторым образом расширил круг программистов и (к сожалению) снизил требования к их инженерной подготовке, скажем, с уровня шамана до уровня вождя племени. Но взамен было получено ощутимое преимущество: разработка программы на ассемблере ускорилась если не на порядок, то, по крайней мере, в разы. Подчеркну еще один немаловажный факт: умение творить на ассемблере не стало анахронизмом и актуально до сих пор, в особенности в области системного программного обеспечения.

Эра превосходства умеющих общаться с ЭВМ шаманов и вождей над обычным людом длилась совсем недолго. Конец неравенству положили новые языки высокого (третьего) уровня. Хотя новые системы программирования по-прежнему представляли собой компромисс между языком машин и людей, но они уже стали доброжелательными, наполнились существенным словарным запасом, плюс ко всему семантика конструкций существенно приблизилась к обычным человеческим фразам. Благодаря всем этим преимуществам, лишь прочитав введение в язык PL/1, ALGOL, ADA, Fortran или во что-нибудь еще, уверенные в своих силах студенты в два счета переводили в состояние ступора ЭВМ любой степени надежности.

Девяностые годы прошлого века ознаменовались рождением языков 4-го поколения – 4GL (fourth generation languages). В них впервые вместо скучных строк кода программист получил удивительную возможность оперировать графическими, интуитивно понятными образами, а для создания элементарного приложения стало достаточно лишь несколько раз щелкнуть кнопкой мыши. В одно время даже раздавались восторженные возгласы о том, что программирование стало доступным для домохозяек...

Не знаю, хорошо это или плохо, но создание профессионального программного продукта и в наши дни по-прежнему требует от человека глубоких и разносторонних знаний, терпения и внимательности, находчивости и сообразительности и, если не таланта, то, по крайней мере, творческой одаренности, потому что программирование уже давно перестало быть просто наукой – это уже и искусство.

В настоящей книге рассматривается один из безусловных лидеров среди современных систем программирования – среда программирования Delphi. Это глубоко продуманный, высокоэффективный и (что немаловажно) весьма удобный программный продукт, позволяющий создавать приложения практически любой сложности, предназначенные для работы под управлением операционных систем Microsoft® Windows® и Linux.

Изначально Delphi специализировалась только на создании программного обеспечения под Windows. Для этого среда снабжена глубоко проработанной и эффективной библиотекой визуальных компонентов (VCL, Visual Components Library), элементы которой не только инкапсулировали в себе функции прикладного программного интерфейса (API, Application Program Interface) Windows, но и внесли существенные усовершенствования. Благодаря этому библиотека VCL успешно конкурирует с библиотекой MFC (Microsoft Foundation Class), разработанной в корпорации Microsoft, и служит фундаментом альтернативным Microsoft Visual Studio средам программирования Borland Delphi и Borland C++.

В условиях жесткой конкуренции фирма Borland постоянно развивает и улучшает возможности среды разработки. Начиная с шестой версии Delphi в состав среды разработки вошел пакет кроссплатформенной разработки CLX (Borland Component Library for Cross-Platform), основанный на идеях, апробированных в VCL. Delphi 2005 впитала идеи создания распределенных программных продуктов, базирующихся на архитектуре Microsoft® .NET Framework.

Особенность пакета CLX в том, что он позволяет строить приложения не только для Windows, но и для набирающей обороты ОС Linux. Тем самым программисты Delphi получили еще одно существенное преимущество – переносимость приложений между разными операционными системами. Однако за универсальность платформы пришлось заплатить – приложения CLX вынуждены отказаться от вызова функций, специфичных для каждой из операционных систем. В связи с этим опора на CLX не столь рациональна в тех случаях, когда вы нацелены только на работу с Windows. Поскольку настоящая книга посвящена программированию для Windows, то мы больше не будем возвращаться к CLX и системе Kylix (дополнение к Delphi для работы с CLX).

Перечисляя заслуги Delphi, стоит упомянуть доступность и интуитивную понятность интерфейса среды, наглядность кодовых конструкций языка Object Pascal, надежную систему выявления ошибок, высокоэффективный компилятор, умение поддерживать самые распространенные форматы баз данных и многое другое.

Соглашения, принятые в книге

Впервые встречающиеся термины выделены полужирным шрифтом, а элементы интерфейса Delphi – шрифтом OfficinaSans.

Моноширинным шрифтом выделены имена файлов, переменных, констант, массивов, записей, методов, классов, свойств, процедур, функций, модулей и библиотек, а также код примеров и синтаксические конструкции. Зарезервированные слова выделены моноширинным полужирным шрифтом.

Для акцентирования внимания читателя на ключевых частях материала текст выделяется следующим образом:

На заметку.
Данный материал – советы, комментарии или замечания – следует принять к сведению.

Внимание!
Текст, отмеченный восклицательным знаком, однозначно описывает действия программиста в той или иной ситуации.

Стоп!
Однозначный запрет; внимание заостряется на характерных ошибках. Короче говоря, никогда так не делайте.

Такой картинкой отмечено описание методов из состава прикладного интерфейса пользователя Windows 32 API. Кроме того, они «переведены» с языка C на язык Pascal.

Подробный предметный указатель позволяет найти в тексте интересующий вас класс, компонент, свойство и метод по их названию.

1

Язык программирования Pascal

Столь популярный сегодня язык программирования Pascal своим рождением обязан профессору Цюрихской высшей технической школы Николаусу Вирту. Язык, названный в честь французского математика Блеза Паскаля, появился на свет в конце 1960-х годов и предназначался для обучения студентов основам структурного программирования.

Уже с самых первых дней своего существования Pascal был обречен на широчайшую популярность благодаря ряду своих неоспоримых достоинств: гибкости и надежности, простоте и наглядности конструкций, способности контроля правильности исходного кода на этапе компиляции, возможности построения новых типов данных и многим другим качествам, с которыми мы познакомимся в этой книге.

*До появления Delphi 7.0 программисты говорили о **«программировании на языке Object Pascal в среде Delphi»** и посмеивались над новичками, употреблявшими словосочетание **«язык Delphi»**. Вот и досмеялись… С официальным выходом Delphi 7.0 человек, сказавший, что он создает программные продукты на «языке Delphi», окажется прав. Действительно, за десятилетие язык Object Pascal был настолько усовершенствован, что Borland официально заговорила о языке Delphi. Вместе с этим, по крайней мере на мой взгляд, термин Object Pascal не стал атавизмом и вполне жизнеспособен. Поэтому эта глава и называется «Язык программирования Pascal».*

Достойным учеником профессора Н. Вирта стал основатель (1983 г.) фирмы Borland International, Inc. Филипп Кан. Совместно с А. Хейльсбергом им были созданы высокоскоростные Турбо-компиляторы для языков Pascal, BASIC, Prolog, C и Assembler.

Наилучшим примером упорного труда программистов Borland по совершенствованию языка Pascal наверное может стать приводимый ниже хронологический перечень:

- 1983 г. – год рождения пакета Turbo Pascal 1.0;
- 1985 г. – выход первой интегрированной среды Turbo Pascal 3.0;

- 1987 г. – Turbo Pascal 4.0, в состав пакета вошли графическая библиотека и средства раздельной компиляции;
- 1988 г. – Turbo Pascal 5.0, дальнейшее совершенствование среды;
- 1989 г. – Turbo Pascal 5.5, промежуточная версия, частично включившая в себя поддержку объектно-ориентированного программирования;
- 1990 г. – Turbo Pascal 6.0, многооконная среда программирования, поддержка мыши, объектно-ориентированная библиотека Turbo Vision;
- 1991 г. – Turbo Pascal for Windows, в состав пакета входит ключевая библиотека `ObjectWindows`;
- 1992 г. – Borland Pascal with Objects 7.0;
- 1995 г. – выход Borland Delphi 1.0 для работы под Microsoft Windows 3.x;
- 1996 г. – Borland Delphi 2.0 (первая 32-разрядная версия);
- 1997 г. – Borland Delphi 3.0;
- 1998 г. – Borland Delphi 4.0;
- 1999 г. – Borland Delphi 5.0;
- 2001 г. – Borland Delphi 6.0;
- 2002 г. – Borland Delphi 7.0;
- 2004 г. – Borland Delphi 8.0.

Простейшая программа на Object Pascal

Как-то раз я услышал неформальное, но, на мой взгляд, очень меткое определение программы: «*Программа – это Идея, которую программист изложил на языке программирования*». Такое определение во главу угла ставит не сотни строк безликого кода, а Ее Величество Идею, то, без чего немыслимо существование творческой личности. Это определение – достойный ответ спорщикам на тему: «Что такое программирование – ремесло или Искусство?»

Эта глава посвящена элементарным составным частям программы на языке Object Pascal. В целом разделы этой главы представляют фундамент, без которого изучение Delphi просто невозможно. Подчеркну, что это лишь предельно сжатый рассказ о возможностях языка Pascal.

При изложении материала, посвященного языку Pascal, мы по возможности абстрагируемся от среды Delphi и ее фундамента – библиотеки визуальных компонентов (VCL). На мой взгляд, первые шаги по изучению языка наиболее эффективны в консольных приложениях, где нет отвлекающих новичка элементов управления и код максимально прост и линеен.

При запуске интегрированной среды разработки Delphi автоматически подготавливается к работе новый проект стандартного приложения для Windows. Но для консольного приложения он не подходит, поэтому закройте созданный по умолчанию проект. Для этого в главном окне Delphi выберите пункт меню File → Close All. Затем найдите пункт File → New → Other... и щелкните по нему. Если все сделано правильно, появится окно New Items с открытой

страницей New. Найдите на этой странице пиктограмму Console Application и щелкните по кнопке OK. За этот каторжный труд Delphi отблагодарит нас заготовкой для самого простейшего приложения – консольного окна Windows.

```
program Project1;
{$APPTYPE CONSOLE}

uses SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Прежде чем я поясню, что содержится в шаблоне кода консольного приложения, научимся сохранять плоды своей деятельности в файл. Для этого выберем пункт меню File → Save и в открывшемся диалоговом окне присвоим своему первому проекту имя FirstPrj.dpr. В завершение нажмем кнопку OK.

Разрабатываемая программа может включать десятки или даже сотни файлов различного типа и назначения. Но как минимум программа состоит из одного файла – главного файла проекта Delphi. Файл такого типа идентифицируется расширением .dpr *(сокращение от Delphi Project).*

Теперь, когда наш проект сохранен на жестком диске компьютера, вновь обратите внимание на первую строку кода. Вместо имени проекта по умолчанию, Project1, после ключевого слова program появится предложенное нами название FirstPrj. Следующее ключевое слово uses применяется для подключения к проекту внешних модулей, как правило, содержащих библиотеки дополнительных подпрограмм. В частности, наш проект будет эксплуатировать наиболее часто используемую библиотеку системных утилит SysUtils. Шаблон завершается составным оператором begin..end, внутри которого размещается выполняемый код. Пока же здесь только текст комментария.

Комментарии в тексте программы

Во всех без исключения языках программирования предусмотрена возможность комментирования строк исходного кода. В комментарии программист в сжатом виде описывает, что делается в этих строках, для чего введена данная переменная, что произойдет после вызова процедуры. Другими словами, в комментариях разработчик кода кратко поясняет смысл рожденных в его голове команд. В результате листинг программы становится более понятным, более читаемым и доступным для изучения.

Для того чтобы при компиляции программы текст комментариев не воспринимался Delphi как исходный код программы и не служил источником ошибок, приняты следующие соглашения. Комментарием считается:

1. Отдельная строка, начинающаяся с двух наклонных черт //:

```
//Одна строка комментария
```

2. Весь текст, заключенный в фигурные скобки {} или в круглые скобки с символами звездочек (**):

```
{Текст комментария}
(*Это также комментарий*)
```

Текст комментария пропускается компилятором и не оказывает влияния на «жизнедеятельность» модуля.

Если внутри фигурных скобок на первой позиции окажется символ $, то это не что иное, как директива компилятора. В шаблоне только что созданного нами приложения такая директива есть:

```
{$APPTYPE CONSOLE}
```

В данном случае это означает, что наш проект является консольным приложением. Начинающему программисту не следует изменять ни содержимое таких строк, ни место их расположения, в противном случае есть риск привести свой проект в негодность.

Компиляция и запуск программы на выполнение

Теперь научимся компилировать программу. **Компилирование** – это процесс, переводящий программу с языка программирования (в нашем случае с языка Pascal) на язык машинных команд. Как-то неинтересно компилировать пустой проект, поэтому давайте научим его чему-нибудь полезному, например здороваться. В следующем листинге предложен пример такой исключительно воспитанной программы. А для того чтобы исходный код был понятнее, он буквально насквозь пропитан комментариями.

```
program FirstPrj;
{это листинг самой короткой и доброжелательной программы на свете}
{$APPTYPE CONSOLE}        //это директива компилятора, которую мы не трогаем

uses SysUtils;  (*строка подключения внешних библиотек подпрограмм, хотя, между нами
говоря, в этой программе внешние модули нам не нужны*)

begin
WriteLn('Hello, World!'); //выводим текст «Привет, Мир!»
ReadLn;                   //ожидаем ввод – нажатие любой клавиши завершит работу
end.
```

Повторив код программы, выберите пункт главного меню Delphi Run → Run. Если все повторено безошибочно, то за считанные доли секунды на экране появятся плоды нашего коллективного творчества – консольное окно со строкой «Hello, World!» Если же вдруг была допущена ошибка, то компилятор просигнализирует о ней, выделив в листинге строку, содержащую предполагаемую ошибку или следующую за ней строку.

Вместо утомительных поисков необходимого элемента в меню у программистов Delphi наибольшей популярностью пользуется быстрая клавиша запуска программы – функциональная клавиша F9. При нажатии этой клавиши осуществляется проверка синтаксиса проекта, его компиляция и запуск исполняемого exe-файла.

Как видите, вся программная логика сосредоточена внутри составного оператора `begin..end` и выполняется линейно в соответствии с очередностью следования строк.

Переменные и константы

Переменная – это хранилище для данных. В самом названии заключен смысл ее применения – переменная предназначена для работы с изменяющимися значениями. В языке Pascal могут быть объявлены переменные различных типов (об основных типах данных мы поговорим чуть позже). Для объявления переменной используется зарезервированное слово var (*сокр.* от variable). Синтаксис выглядит следующим образом:

```
var имя_переменной : тип_данных;
```

Например:

```
var  x : Integer;      //переменная x специализируется на работе с целыми числами
     y, z : real;      //переменные y и z могут хранить действительные числа
     s : char;         //s – символ
     MyVar : Boolean; //MyVar – логическая переменная
```

Как и переменная, константа также является хранилищем для данных, но, в отличие от переменной, константа задается раз и навсегда и не допускает редактирования своего содержимого. Для задания константы применяется зарезервированное слово const (*сокр.* от constant). Синтаксис определения константы следующий:

```
const <имя_константы> [: тип данных] = <значение>;
```

В квадратных скобках может быть отмечен необязательный указатель на тип константы. Приведем пример определения обычных констант:

```
const  A = 100;
       B = -3.1426;
       C = 'Текст';
```

Для задания значения, которое будет содержаться в обычной константе, допускается применение математических выражений и результатов, возвращаемых функциями.

```
const  D = 500;
       E = D+6;
       F = 3/Pi;
```

При определении типизированной константы явным образом указывается тип хранящихся в ней данных:

```
const  H : byte = 255;
       I : Boolean = true;
```

В отличие от обычных, или как их еще иногда называют «истинных», констант, типизированные константы не рекомендуется инициализировать выражениями. В Delphi в качестве констант могут определяться массивы, записи, указатели; кроме того, существуют и экзотические константы, например процедурные.

Важно знать, в каком именно месте листинга допускается объявление переменных и констант. В консольных проектах объявление осуществляется перед составным оператором begin..end.

```
program Project1;

{$APPTYPE CONSOLE}

uses SysUtils;
const X = 10;              //объявление константы
var   Y, Z : integer;      //объявление двух переменных
begin
Z:=X+Y;
//остальной код программы
end.
```

Идентификаторы

Идентификатор – это имя переменной, константы, массива, метода, модуля и всего остального, что должно иметь имя. В Delphi длина идентификатора не ограничена, но значащими являются только первые 255 символов. Идентификатор может содержать любые символы латинского алфавита, цифры и символ нижнего подчеркивания. Первый символ идентификатора обязательно должен быть буквенный. Само собой в роли идентификаторов не допускается применять зарезервированные слова.

В отличие от C, язык программирования Pascal не критичен к регистру символов, поэтому в Delphi следующие названия будут восприниматься как идентичные: `MyValue`, `myvalue`, `MYVALUE`, `mYvALUE`.

Если в рамках одного проекта существует несколько модулей с одинаковыми именами идентификатора, то для обращения к идентификатору требуется уточнить, кому он принадлежит.

```
Form1.Button1.Caption;
Form2.Button1.Caption;
Unit1.MyProcedure;
```

Основные типы данных

Одной из ключевых особенностей языка Object Pascal является жесткая типизация данных. Именно благодаря строгости в подходе к объявлению переменных, процедур и функций, Delphi может похвастаться одним из самых совершенных компиляторов. Что такое типизация? Все достаточно просто. Представьте себе педантичного джентльмена, любящего находить все свои вещи в отведенных им местах: зубную щетку – в шкафчике над умывальником, а смокинг – в платяном шкафу, и никак не наоборот. Если вдруг произойдет обратное, то Delphi потеряет к нам всякий интерес, отправив сообщение об ошибке.

Более того, размеры каждой вещи нашего педанта (другими словами, объем памяти, занимаемый переменной или объектом) соответствуют четко установленным правилам – не больше и не меньше. Однако тип данных не только накладывает ограничения на размер объекта, скажем переменной, но и строго определяет перечень операций, которые можно производить с этим объектом. И это правило весьма логично и последовательно – ведь совсем не

стоит, например, в переменную, предназначенную для хранения целого числа, помещать пусть даже очень хорошую строку «Hello, Word!».

На рис. 1.1. предложен вариант классификации типов данных, применяемых в Delphi.

Рис. 1.1. *Классификация основных типов данных языка Delphi*

Не надо отчаиваться при виде такой паутины (это, кстати, только верхушка айсберга). Нашими общими усилиями узелок за узелком она будет распутана. Итак, каждый тип данных предназначен для хранения информации определенного вида, и в самом общем случае можно говорить о существовании шести основных типов данных языка Delphi:

- простой
- строковый
- структурный
- указательный
- процедурный
- вариантный

Простые типы данных

Самым большим из представленных типов по праву считается простой. Он предназначен для хранения данных в форме чисел или некоторых упорядоченных последовательностей. Этот тип логически разделяется на две ветви: порядковые и действительные типы. К порядковым типам относятся:

- целые числа
- символьные типы
- логические типы
- перечислимые типы
- поддиапазоны

Если первые три группы в языке Delphi описаны самым жестким образом и не допускают каких-либо изменений, то два последних типа (перечисли-

мый и поддиапазон) могут определяться пользователем непосредственно во время процесса разработки программы.

Отношения между элементами любого из порядковых типов складываются вполне ординарно. Все элементы внутри одного типа могут располагаться в виде упорядоченной последовательности (следующий больше предыдущего). Для всех простых типов (за исключением целых чисел) справедливо утверждение, что первый (самый младший) элемент последовательности имеет индекс 0, второй – 1 и т. д. Целые же числа допускают хранение и отрицательных значений, поэтому здесь самый младший элемент последовательности может начинаться не с нуля. К еще одной особенности порядковых типов данных стоит отнести отсутствие знаков после запятой.

Целые числа

В табл. 1.1 представлены порядковые целочисленные значения. В первую очередь типы данных, описывающие целые числа, характеризуются пределом границ диапазона хранимых значений и возможностью описывать отрицательные величины. Чем больше предел допустимых значений, тем больший объем памяти будет занимать переменная этого типа. Как видно из таблицы, самым серьезным из предлагаемых типов является `Int64`, «пожирающий» целых 8 байт (64 бит) ОЗУ. Такой тип способен хранить величины, сопоставимые с количеством звезд во Вселенной. Как правило, для решения «земных» задач программисту достаточно диапазона значений типа `Integer`.

Таблица 1.1. Целые числа

Тип	Диапазон значений	Размер в байтах
`Int64`	$-2^{63} .. 2^{63}-1$	8
`Integer (Longint)`	−2147483648 .. 2147483647	4
`Smallint`	−32768 .. 32767	2
`Shortint`	−128 .. 127	1
`Byte`	0 .. 255	1
`Word`	0 .. 65535	2
`Cardinal (LongWord)`	0 .. 4294967295	4

Символьные типы

Основное назначение символьного типа данных – организация вывода информации на экран компьютера и принтер. В Windows обеспечена поддержка трех наиболее важных наборов символов:

1. OEM – набор символов по умолчанию для MS-DOS.
2. ANSI – набор символов по умолчанию для Windows 9.x.
3. Unicode – набор символов по умолчанию для Windows NT/2000.

Фундаментом наборов символов OEM и ANSI служит код ASCII, в котором каждый символ представлен значением от 0 до 127 (соответственно символ занимает 7 бит памяти). Кодам от 0 до 31 и 127 стандартный 8-битный набор ставит в соответствие управляющие символы (например, символы забоя, та-

буляции, конца строки и возврата каретки); остальные символы могут быть выведены на экран. Исторически сложилось, что оставшиеся символы были закреплены за латинскими буквами.

Вскоре был задействован и восьмой бит кода, что позволило расширить код ASCII до 256 символов («расширенный набор символов»). Этот набор символов был разработан производителями IBM PC и получил название OEM. Здесь коды от 32 до 126 унаследованы от ASCII, а оставшиеся коды включают дополнительные символы, в частности символы псевдографики для программ DOS.

В большинстве случаев Windows и приложения под Win32 используют «набор символов ANSI». Коды данного набора от 32 (0x20) до 127 (0x7F) соответствуют коду ASCII. Сравнительно недавно появилась еще одна кодировка, получившая название UNICODE. Один символ в такой кодировке занимает целых два байта, и благодаря этому он может принимать одно из 65535 значений.

Итак, для работы с отдельными символами Delphi предоставляет следующие типы данных:

Таблица 1.2. Символьные типы

Тип	Кодировка	Размер в байтах
`Char (AnsiChar)`	ANSI	1
`WideChar`	UNICODE	2

Логические (булевы) типы

Логический тип применяется для хранения логических данных, способных принимать только два значения: 1 (`true`/истина) и 0 (`false`/ложь).

Таблица 1.3. Логические типы

Тип	Диапазон значений	Размер в байтах
`Boolean`	0 – `false`; 1 – `true`;	1
`ByteBool`	от 0 до 255, где 0 – `false`, 1..255 – `true`	1
`WordBool`	от 0 до 65535, где 0 – `false`, 1..65535 – `true`	2
`LongBool`	от 0 до 4294967295, где 0 – `false`, 1..4294967295 – `true`	4

Перечислимые типы

Перечислимые типы относятся к типу данных, определяемых программистом. Перечислимый тип данных задается списком имен.

```
type TypeName = (Value1, Value2,..., Value19);
```

Числа, а также логические и символьные константы не могут быть элементами перечислимого типа. В качестве примера представим перечислимый тип, соответствующий дням недели:

```
type TypeWeekDay =(Mon, Tu, We, Th, Fr, Sa, Su);
. . .
var WDay1, WDay2 : TypeWeekDay;
begin
```

```
WDay1 : = Mon;
WDay2 : = Tu;
end;
```

Особенность перечислимого типа в том, что каждому его элементу соответствует порядковый номер, начиная с 0. Наличие порядкового номера позволяет проводить операции сравнения:

```
if WDay1<WDay2 then …
```

Совместно с данными перечислимого типа зачастую используют следующие функции:

```
function Pred(X);   // возвращает предшествующее значение аргумента
function Succ(X);   // возвращает следующее значение аргумента
```

Поддиапазоны

Переменная, входящая в поддиапазон, может принимать значения только в пределах границ диапазона.

```
type SubIntegerRange = 10 .. 100;
type SubCharRange =  'A' .. 'Z';
. . .
var IntValue  : SubIntegerRange;
    CharValue : SubCharRange;
. . .
MyValue : = 50;
CharValue : = 'X';
```

При попытке присвоить переменной `IntValue` значение вне диапазона `SubIntegerRange` компилятор Delphi откажется иметь с нами дело.

Операции с порядковыми типами

Отношения порядка определяют для переменных простого типа перечень простейших допустимых операций.

Таблица 1.4. Допустимые операции

Операция	Содержание
Порядковые функции	
`function Ord(X): Longint;`	Возврат порядкового значения `X`
`function Odd(X: Longint): Boolean;`	Если `X` – нечетное число, то `true`, иначе `false`
`function Succ(X);`	Следующее по порядку значение `X`
`function Pred(X);`	Предыдущее значение `X`
`procedure Inc(var X [ ; N: Longint ] );`	Приращение `X` на `N`, аналог `X:=X + N`
`procedure Dec(var X[ ; N: Longint]);`	Уменьшение `X` на `N`, аналог `X:=X - N`
`function Low(X);`	Минимальное порядковое значение `X`
`function High(X);`	Максимальное порядковое значение `X`
`function Chr(X: Byte): Char;`	Возвращает символ таблицы ASCII, соответствующий порядковому значению `X`

Для всех порядковых типов допустима операция **задания (приведения) типа**. Ее смысл – в приведении преобразования переменной к определенному типу. Для этого саму переменную заключают в круглые скобки, а перед ними ставят название типа, к которому мы хотим привести переменную:

```
var   C: Cardinal;
      I: Integer;
      B: Byte;
begin
…
B:=Byte(C);
B:=Integer(I);
end;
```

Поясню, что происходит при задании типа. Допустим, что мы приводим переменную типа `Integer` к типу данных `Byte`. Если реальное значение, содержащееся в этой переменной, не выходит за границы, допустимые в типе данных `Byte` (0...255), то значение изменениям не подвергается. Но если значение превысит 255 или станет меньше 0, то операция задания типа включит ограничения и не допустит выхода значения за пределы диапазона `Byte`.

```
B:=Byte(-1);    {результат B=255}   B:=Byte(511);  {результат B=255}
B:=Byte(-2);    {результат B=254}   B:=Byte(510);  {результат B=254}
B:=Byte(-255);  {результат B=1}     B:=Byte(257);  {результат B=1}
B:=Byte(-256);  {результат B=0}     B:=Byte(256);  {результат B=0}
```

Читатель, имеющий некоторый опыт программирования, наверное, уже выявил закономерность преобразования числа при операции задания типа. Например, для типа `Byte` действие

```
B:=Byte(I);
```

является аналогом операции

```
B:=I mod 256; //результат = остаток от деления I на 256
```

Действительные типы

Действительные (вещественные) типы данных предназначены для работы со значениями, содержащими не только целую, но и дробную часть.

Таблица 1.5. Действительные числа

Тип	Диапазон значений	Количество знаков	Размер в байтах
`Real48`	$2.9 \times 10^{-39} .. 1.7 \times 10^{38}$	11–12	6
`Single`	$.5 \times 10^{-45} .. 3.4 \times 10^{38}$	7–8	4
`Double (Real)`	$5.0 \times 10^{-324} .. 1.7 \times 10^{308}$	15–16	8
`Extended`	$3.6 \times 10^{-4951} .. 1.1 \times 10^{4932}$	19–20	10
`Comp`	$-2^{63}+1 .. 2^{63}-1$	19–20	8
`Currency`	–922337203685477.5808.. 922337203685477.5807	19–20	8

Если в программе необходимо производить вычисления действительных чисел, то по возможности объявляйте переменные как Single (если, конечно, вас устраивает диапазон данного типа данных).

При осуществлении математических операций с переменными действительного типа будьте готовы к незначительным ошибкам округления.

```
var S : Single; R,F : Real;
begin
S:=1/3;  R:=1/3;
F:=S-R;       //результат F = 0.000000009934107
end;
```

Предложенный листинг демонстрирует ситуацию, когда компьютер «ошибся» в элементарных операциях вычитания. Даже второклассник знает, что 1/3 – 1/3 = 0, а наш кремниевый друг насчитал что-то около 0.000000009934107. На самом деле в ошибке виноваты мы, а не «бестолковый» компьютер. Ведь в программе мы использовали различные типы данных. Переменная S объявлена как Single, поэтому она способна точно хранить лишь 7–8 знаков после запятой, а переменная R объявлена как Real, т. е. способна хранить 15–16 знаков после запятой. В результате имеем S = 0.333333343267441, R = 0.333333333333333.

При проектировании бухгалтерских приложений для обработки денежных величин применяйте переменные типа Currency. *Значение, переданное в переменную этого типа данных, физически хранится в формате* Int64, *при этом Delphi полагает, что четыре последних знака этого значения – знаки после запятой. Таким образом, действительное число 9,9999 обрабатывается как целое 99999, но при выводе на экран и проведении математических операций в тайне от нас Delphi делит его на 10000, тем самым соблюдая статус кво. Вся эта казуистика позволяет избежать ошибок округления, что очень нравится бухгалтерам.*

Строковый тип

Строковый тип данных предназначен для хранения последовательности букв, цифр и других символов. Обычная строка представляет собой не что иное, как массив символьных значений плюс некоторая служебная информация. В Delphi реализовано четыре основных строковых типа (табл. 1.6).

Таблица 1.6. Строковые типы

Тип	Максимальная длина	Размер в памяти
ShortString	255 символов	2 .. 256 байт
String	определяется директивой компилятора $H. Если она включена, то соответствует AnsiString, иначе ShortString	
AnsiString	около 231 символов	4 байт .. 2 Гбайт
WideString	около 230 символов	4 байт .. 2 Гбайт

Исторически в Delphi 1.0 появился тип данных ShortString. В памяти компьютера она представляет собой цепочку байтов, причем в первом байте содер-

жится значение длины текстовой строки, а в остальных – непосредственно информация. Другими словами, если вас зовут «Петя» (что составляет 4 символа), то в служебном байте окажется четверка, а в оставшихся четырех байтах – соответственно символы «П», «е», «т» и «я».

Физический формат строки `AnsiString` значительно сложнее. Официальная информация о том, каким образом создатели Delphi организовали хранение данных в этом типе строки, отсутствует. (Это говорит о том, что Borland оставляет за собой право изменять внутренний формат строки такого типа.) Однако серьезный программист обязан знать о следующих особенностях физического формата `AnsiString`.

Во-первых, это строка, заканчивающаяся нулем, – в самом последнем байте этой строки окажется символ #0. Во-вторых, форматом строки предусмотрена область, хранящая данные о количестве ссылок на эту строку. Благодаря тому что строка завершается нулевым символом, она прекрасно взаимодействует с функциями Windows API, а из-за наличия счетчика ссылок (хранящего данные о том, сколько строковых переменных ссылаются на одно и то же место в памяти) значительно упрощается операция копирования строки из переменной в переменную. В этом случае просто копируется указатель и осуществляется приращение счетчика ссылок.

Строки `WideString` предназначены для работы с 16-битными символами, т. е. здесь на каждый символ отводится два байта. Таким образом, тип данных `WideString` способен работать с символами из таблицы Unicode (UCS-2). Unicode – стандарт, рожденный в недрах Apple и Xerox в 1988 г. Спустя три года для совершенствования и внедрения Unicode был создан консорциум, в состав которого вошли более десятка ключевых компаний, в том числе и Microsoft.

Поскольку на каждый символ отводится два байта, Unicode позволяет кодировать 65 536 символов, что более чем достаточно для работы с любым языком. Поэтому разработчики Unicode решили определить местоположение символов каждого из ключевых мировых языков (табл. 1.7) и расширить набор символов огромным количеством технических символов. На сегодняшний день определено около 35 тысяч кодовых позиций и еще около 30 тысяч позиций свободны.

Таблица 1.7. Unicode

16-битный код	Символы	16-битный код	Символы
0000-007F	ASCII	0300-U36F	Общие диакритические
0080-00FF	Символы Latin 1	0400-04FF	Кириллица
0100-017F	Европейские латинские	0530-058F	Армянский
01 80-01FF	Расширенные латинские	0590-05FF	Еврейский
0250-02AF	Стандартные фонетические	0600-06FF	Арабский
02BO-02FF	Модифицированные литеры	0900-097F	Деванагари

При объявлении переменной строкового типа допускается явным образом ограничить ее длину, для чего в квадратных скобках указывают количество символов в строке:

```
var Name : string[40];
```

Объявленная таким образом переменная занимает в памяти количество байт, равное длине строки + 1 байт.

Структурные типы

Основное назначение структурных типов – совместное хранение множества однотипных или разнотипных значений. Различают следующие разновидности структурных типов:

- массивы
- множества
- классы
- записи
- файлы
- указатели на классы

Первые три типа будут рассмотрены в этой главе, а файлы, классы и указатели на классы – несколько позже.

Массивы

Представим, что перед программистом поставлена задача разработки программного модуля для хранения фамилий студентов или что-нибудь похожее, связанное с обработкой большого количества однотипных данных. Использование обычных переменных для хранения упорядоченных наборов данных одного типа не является эффективным решением подобных задач. Явное неудобство состоит в объявлении десятков, сотен или даже тысяч переменных, а сложность обращения к каждой из них тем более не нуждается в доказательствах. Выходом из такой ситуации является применение массивов (`array`).

Массив, как и переменную, необходимо объявить. Для этого нужно указать размерность массива и тип хранимых данных:

```
var <имя_массива>: array [<нижняя граница> .. <верхняя граница>] of <тип_элементов>;
```

Если в программе будут применяться несколько однотипных массивов, то предварительно стоит определить тип массива, а затем создавать массивы на базе объявленного типа:

```
type TMyArray = Array [0..9] of integer; // массив из 10 элементов
var Array1, Array2 : TMyArray;
```

Для обращения к элементам массива с целью чтения или записи достаточно указать индекс элемента в массиве:

```
NewArray[0]:=199;    // 0-му элементу массива присваивается значение 199
I:=NewArray[9];      // в переменную I записано содержимое 9-го элемента массива
```

При обращении к элементам массива внимательно следите за тем, чтобы передаваемый индекс не выходил за границы массива, в противном случае вы получите сообщение об ошибке.

Иногда полезно задавать массив в виде константы. В следующем примере 12 ячеек массива используются для хранения количества дней в месяцах високосного года:

```
const
DaysInMonth: array [1..12] of byte = (31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
```

Хотя Delphi допускает задание нижней границы массива любым числом (например, массив DaysInMonth*), но хорошей практикой считается начинать нумерацию с нуля. Кстати, методы* Low() *и* High()*, возвращающие границы массива, предназначены для работы только с массивами, начинающимися с нуля, иначе они вернут ошибочные значения.*

Вполне реально объявлять «квадратные» и «кубические» массивы, а также массивы более высоких размерностей. Например, объявление двумерного массива размерностью 10 на 10 ячеек выглядит следующим образом:

```
var MyArray : Array[0..9,0..9] of cardinal;
```

или

```
var MyArray : Array[0..9] of Array[0..9] of cardinal;
```

Но теперь, для того чтобы обратиться к интересующей нас ячейке двумерного массива, потребуется указать 2 индекса:

```
MyArray[3,6]:=56;
```

или

```
MyArray[3][6]:=56;
```

Чем больше элементов включает массив, тем значительнее затраты памяти на хранение его содержимого. Например, двумерный массив типа DWord *размерностью всего 100 на 100 элементов потребует 100×100×8 байт = 80 000 байт оперативной памяти. Но зачастую размерность массива определяется с запасом (на всякий случай), а в ходе выполнения программы последний не заполняется и на 50%. Жаль потерянные байты. В таких случаях применяйте сжатые (packed) массивы:*

```
var MyArray : packed array [0..99,0..99] of cardinal;
```

Такой способ объявления гарантирует более рачительное использование ОЗУ, но несколько снижает скорость доступа к элементам массива.

У рассмотренного выше способа хранения данных есть один существенный недостаток – объявив размерность массива (сделав его **статическим**), мы не сможем выйти за его границы. А что делать, если заранее даже приблизительно неизвестно, сколько элементов может оказаться в массиве? В таких случаях используют **динамические массивы**, которые отличаются от статических тем, что их границы могут изменяться во время работы приложения.

Естественно, объявление динамического массива выглядит несколько иначе:

```
var MyArray: array of INTEGER;
```

Как видите, при объявлении массива мы не определяем его размерность. Но перед заполнением массива нам все-таки придется это сделать с помощью метода SetLength():

```
SetLength(MyArray, 10);     //распределяем память для 10 элементов массива
```

Отсчет элементов динамического массива всегда начинается с нуля. При работе с многомерным динамическим массивом, например следующего вида:

```
var MyArray : Array of Array of Char;
```

все размерности массива можно задавать одновременно:

```
SetLength(MyArray, 10, 5);  //распределили память для 2-мерного массива
```

или последовательно для каждого индекса. Массивы с переменной длиной по разным индексам называют **динамическими разреженными массивами**.

```
SetLength(MyArray,3);       //массив состоит из 3 строк
SetLength(MyArray[0],3);    //в нулевой строке 3 элемента
SetLength(MyArray[1],2);    //в первой строке 2 элемента
SetLength(MyArray[2],10);   //во второй строке 10 элементов
```

Раньше для освобождения памяти, выделенной для динамического массива, применяли процедуру `Finalize()`.

```
Finalize(MyArray);
```

Теперь это делать необязательно, т. к. при завершении работы с массивом Delphi самостоятельно освободит занятые ресурсы.

При работе с однотипными динамическими массивами наиболее эффективным способом копирования данных из одного массива в другой считается вызов функции `Copy()`. Метод умеет копировать как массив целиком, так и некоторые его элементы.

```
var Arr1, Arr2 : array : of integer;
…
SetLength(Arr1,4)
for i:=0 to High(Arr1) do Arr1[i]:=i;   //заполнение массива данными
Arr2:=Copy(Arr1);                       //полное копирование
Arr2:=Copy(Arr1, 0, 2);                 //копирование части массива
```

Заметьте, что мы не задаем длину второго массива, она будет определена автоматически с вызовом метода `Copy()`. Кроме того, динамические массивы понимают механизм ссылок.

```
var Arr1, Arr2 : array : of integer;
…
Arr2:=Arr1;
```

Обратите внимание: при простом копировании массивы хранят данные в разных областях ОЗУ, а при использовании оператора присваивания оба массива будут ссылаться на одну и ту же область памяти. И если теперь нулевому элементу первого массива присвоить значение 10, то это же значение окажется в нулевой ячейке второго массива.

Массив можно передавать и как параметр метода, правда, с некоторыми ограничениями. Если параметр представляет собой статический массив, то массив предварительно должен быть типизирован.

```
type TMyArray = Array [0..5] of Byte;
```

```
procedure Proc1(Arr : TMyArray);
var i : integer;
begin
    for i:=Low(Arr) to High(Arr) do Arr[i]:=0;
end;
```

Объявление процедуры с аргументом в виде динамического массива несколько проще:

```
procedure Proc1(Arr : Array of Byte);
```

Записи

Рассмотрим небольшую задачу. Необходимо организовать учет сотрудников фирмы. Учету подлежат: фамилия работника, его заработная плата и стаж работы на предприятии в годах. На первый взгляд решение лежит на ладони – берем три переменные типа String, Currency и Byte соответственно и с их помощью обрабатываем данные. Но эта задача решается элегантнее с помощью механизма записей. Объявляем тип TPeople:

```
type
    TPeople = Record
    Surname    : String;
    Money      : Currency;
    Experience : Byte;
end;
```

Запись TPeople определяет единый тип данных, содержащий три разнотипных элемента, называемых **полями записи**. Доступ к полям записи осуществляется по их имени. Вот пример заполнения такой записи:

```
var People : TPeople;    //объявление переменной на основе типа данных TPeople
begin
    People.Surname    := 'Петров';
    People.Money      := 1500.55;
    People.Experience := 10;
end;
```

Впрочем, для определения одной единственной переменной-записи совсем необязательно ее типизировать. Приведем пример объявления такой переменной:

```
var  S : record
     Name : string;
     Age  : Integer;
     Experience : byte;
end;
```

Стоит пойти дальше – объединить достоинства массива (хранилища множества однотипных данных) и записи, позволяющей хранить разнотипные данные.

```
var    MyArray : Array[0..99] of TPeople; //объявлен массив записей
begin
    MyArray[0].Family:='Иванов';
```

```
    MyArray[0].Money:=1500.55;
    MyArray[0].Experience:=10;
end;
```

В представленном выше листинге мы объявили массив MyArray, содержащий 100 элементов типа TPeople, и заполнили нулевой элемент массива.

Современный синтаксис позволяет создавать записи с различными вариантами полей. Будем называть такую запись **записью с вариантным полем**. Синтаксис объявления выглядит следующим образом:

```
type <имя_типа_записи> = record
     <поле_1>: <тип_данных_1>;
     <поле_2>: <тип_данных_2>;
       ...
     case <поле_N>: <порядковоый_тип_данных> of
     значение_1: (вариант 1);
       ...
     значение_M: (вариант 2);
end;
```

Отличительная особенность записи с вариантным полем – наличие внутри нее оператора-селектора case. Если этот оператор вам пока незнаком, то отложите рассмотрение этой части материала и вернитесь к ней после того, как прочтете раздел «Операторы», а для всех остальных я продолжу.

Допустим, что мы пишем программу для владельца гостиницы, с помощью которой он собирается вести учет своих постояльцев. Владельцу нужны следующие данные: фамилия (Surname) и имя (FName) постояльца, а также даты въезда (EntryDate) и выезда (ExitDate). Немного подумав, наш привередливый заказчик потребовал, чтобы при заказе номера люкс (VIPRoom) наша программа учитывала домашний адрес постояльца, а для обычных посетителей достаточно запомнить только номер паспорта (Doc). В такой ситуации к концу бессонной ночи мы бы изобрели запись THotelGuest.

```
type
  THotelGuest = record
  SurName, FName : string[30];
  EntryDate, ExitDate: TDate;
  case VIPRooms: Boolean of            //селектор проверяет значение поля VIPRooms
    False: (Doc: string[15]);          //False - только номер паспорта
    True: (City, Street : string[20];  //True - домашний адрес
           HomeNum: smallint);
  end;
```

Оператор case проверяет поле VIPRooms и в зависимости от принимаемого им значения предоставляет для заполнения те или иные данные. Такой подход позволяет значительно рациональнее использовать оперативную память компьютера.

Множества

Множества представляют собой коллекцию однотипных значений. При определении множества можно указать диапазон значений:

```
type TIntSet = set of 1..10;
```

В качестве элементов множества могут использоваться любые порядковые типы данных:

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

Кроме того, вы имеете право определять собственные элементы множества:

```
type TWeekDays = set of (Mo, Tu, We, Th, Fr, St, Su);
```

Нетерпеливый читатель спросит: «Так чем же отличаются множества от изученных ранее поддиапазонов?» Постараюсь объяснить на небольшом примере. Представьте себе, что вы работаете в театре, правда, пока не главным режиссером, а лишь осветителем. Поскольку театр небольшой, то в распоряжении осветителя лишь три прожектора:

```
type TLampsSet = set of (Lamp1, Lamp2, Lamp3);
```

Для освещения сцены осветитель может зажечь один, два или все три прожектора, т. е. выбрать любую понравившуюся комбинацию. Если ассоциировать прожекторы с ячейками памяти компьютера, то это всего-навсего три бита. Единица в ячейке свидетельствует о включении, а ноль – о выключении соответствующего прожектора (рис. 1.2).

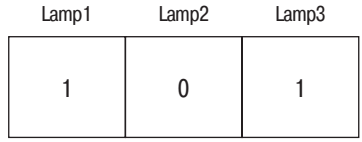

Рис. 1.2. *Представление множества в памяти компьютера*

Если множество содержит всего три элемента, то общее количество возможных комбинаций составляет $2^3 = 8$. Зарезервированное слово set способно определять множество размерностью до 256 элементов, т. е. $2^{256} = 1{,}157920892373161954235709850086 9e+77$ вариантов. На практике такое количество вариантов никогда не понадобится. В частности, разработчики Delphi рекомендуют использовать множество с количеством элементов не более 16.

А теперь рассмотрим несколько строк кода, демонстрирующих работу с множествами:

```
type TLampsSet = set of (Lamp1, Lamp2, Lamp3);
var LampsSet : TLampsSet;
begin
LampsSet:=[];                    //0,0,0 - полностью очистили множество
LampsSet:=[Lamp1];               //1,0,0 - включили первый элемент
LampsSet:=LampsSet+[Lamp3];      //1,0,1 - добавили третий элемент
LampsSet:=LampsSet-[Lamp1];      //0,0,1 - отключили первый элемент
LampsSet:=[Lamp1,Lamp2,Lamp3];   //1,1,1 - включили все три элемента
end.
```

Если определено два однотипных множества, то между ними вполне допустимы операции сложения, вычитания, умножения и сравнения (<=, >=, =, <>).

В операциях сравнения множество X меньше или равно множеству Y (выражение (X <= Y) = True), если каждый элемент множества X является членом

множества Y. Множество X равно множеству Y (выражение (X = Y) = True), если все элементы множества X точно соответствуют элементам Y. Множество X не равно множеству Y (выражение (X <> Y) = True), если хотя бы один элемент множества X отсутствует в множестве Y.

Для того чтобы проверить, включен ли элемент в множество, применяют оператор in.

```
if (Lamp2 in LampsSet) then <операция 1> else <операция 2>;
```

Указатели

А теперь поговорим об указателях. Начнем с напоминания о том, что физически любая переменная представляет собой не что иное, как область памяти, содержащую какие-то данные. Когда мы объявим переменную MyValue : integer, в памяти компьютера для хранения значения этой переменной будет зарезервировано 4 байта. Содержимое переменной MyValue можно просмотреть непосредственно в этой области памяти. Объявив другую переменную, мы заставим операционную систему отвести под эту переменную новые свободные ячейки памяти. При этом значения указателей на MyValue и новую переменную будут различны.

Указатель представляет собой переменную, содержащую *адрес области памяти*. Повторюсь еще раз, т. к. это важно: указатель хранит не содержимое памяти, а адрес ячеек памяти. Поэтому он сам не занимает никакого места, кроме того, которое нужно для хранящегося в нем адреса. На практике это может выглядеть следующим образом:

```
var  MyValue  : integer;
     pMyValue : pointer;
begin
MyValue:=100;
pMyValue:=@MyValue;  // указателю присвоен адрес переменной MyValue
end;
```

Обратите внимание, что в операции присваивания адреса указателю перед названием переменной помещен символ @. Допустим, у нас есть переменная See : Integer и мы хотим через указатель передать ей данные из переменной MyValue. Тогда, дабы увидеть данные, хранящиеся в MyValue(), воспользуемся следующими строками кода:

```
See:=INTEGER(pmyValue^);
```

Эквивалентом оператора @ служит метод Addr(X). *Функция возвращает указатель на объект* X.

```
function Addr(X): Pointer;
```

Тип данных Pointer называют нетипизированным указателем, т. к. он может указывать на переменную любого типа. Чаще применяют так называемые типизированные указатели, которые способны работать с переменными определенного типа. Объявление такого указателя выглядит следующим образом:

```
var pInt:^integer;
```

Рассмотрим еще один небольшой пример:

```
var  pInt    : ^integer;
     MyValue : integer;
begin
MyValue:=100;          //целочисленной переменной присвоено значение 100
Pint  :=Addr(MyValue);  //указатель установлен в область памяти, где хранится MyValue
pInt^ :=123;           //в область памяти записано значение 123
end;
```

Результатом данного упражнения стало изменение значения переменной MyValue без обращения к ней.

Если указатель пуст (ссылается «в никуда»), то он возвращает особое значение nil. *Пустой указатель называют* ***неопределенным****. В языке Object Pascal* nil *– это специальная константа, предназначенная для описания пустых (несуществующих) данных.*

Тип PChar

Это типизированный указатель на строки, завершающиеся нулем (null-terminated strings). Дело в различии между форматом строк, используемых в функциях Windows API, и строками языка Object Pascal. Строки Windows и языка C не имеют определенного размера, и признаком их окончания служит нулевой символ (#0). При необходимости использовать встроенные функции Windows вам придется использовать PChar.

Помимо PChar *в Object Pascal объявлены еще два типа указателей, специализирующихся на работе с текстовыми строками, заканчивающимися нулем. Это указатели* PAnsiChar *и* PWideChar. *Указатель* PAnsiChar *предназначен для работы со строками Ansi, а* PWideChar *– указатель на строку с 16-битными символами из таблицы Unicode.*

```
var Buff : Array[0..12] of Char = 'Hello world!'#0;
    P    : PChar;
begin
p:=@Buff[0];
p:='Hello world!';
end;
```

В этом примере продемонстрированы способы сопоставления текстовых данных с указателем PChar.

Вариантные типы

Универсальный тип данных. В основном он предназначен для работы с результатами заранее не известного типа. Но за универсальность приходится платить: на переменную вариантного типа дополнительно отводится еще два байта памяти (это не считая байт, необходимых для хранения обычной типизированной переменной).

```
var  vUniverse : variant;
     iInt      : integer;
     sStr      : string;
     rR        : real;
```

```
begin
iInt:=1; sStr:='Привет'; rR:=1.987;
vUniverse:=iInt; vUniverse:=sStr; vUniverse:=rR;
end;
```

Ни один из типов данных не позволит таких вольностей, как `variant`. В приведенном примере переменной `vUniverse` по очереди присваиваются значения различного типа.

Вариантный тип переменной полезен при вызове объектов, поддерживающих технологию OLE Automation, хранения значений даты/времени и создания массивов переменной длины.

Операторы и выражения

С помощью операторов выполняются определенные действия (операции) с данными различных типов. Наиболее часто используется оператор присваивания.

Оператор присваивания

С оператором присваивания мы уже познакомились на предыдущих страницах. Он обозначается символами `:=`. При выполнении оператора присваивания результат выражения, стоящего справа от данного оператора, присваивается переменной, находящейся слева от оператора присваивания. В результате операции `X := 1+1` переменной `X` будет присвоено значение 2.

Не забывайте, что Pascal – строго типизированный язык. Тип переменной, стоящей слева от оператора `:=`, должен быть совместим с типом, получаемым в результате выполнения выражения. Delphi не допустит ошибок приведения типов. Для тренировки сообразительности самостоятельно найдите ошибку в следующих строках кода:

```
var  X, Y : real;
     Z : integer;
begin
X:=2; Y:=3;
Z:=X+Y;
end;
```

Порядок операций

В целом порядок выполнения операций в языке Pascal соответствует правилам, принятым в обычной арифметике. Например, для вычисления выражения $2 + 5 \times 10$ необходимо вначале выполнить операцию умножения, а затем сложения. Но в Delphi есть и свои исключения, которые касаются в основном операций, которые не используются в математике.

Таблица 1.8. Приоритеты операций

Приоритет	Операции
1 (высший)	^
2	@, NOT
3	*, /, DIV. MOD, AND, SHL, SHR, AS
4	+, – ,OR
5 (низший)	=, <>, <, <=, >=, IN

Наивысшим приоритетом обладает операция ^, отвечающая за обращение к переменной или полю через указатель. Далее следуют `@` и `not` – операции с одним операндом. Операции `*`, `/`, `div`, `mod`, `and`, `shl`, `shr`, `as` отвечают за умножение, деление и задание типа.

Арифметические операции

Арифметические операции необходимы для математических действий с целыми и вещественными типами данных. Помимо известных еще из курса начальной школы операторов сложения, вычитания, умножения и деления, Delphi обогащен операторами целочисленного деления.

Таблица 1.9. Арифметические операции

Оператор	Операция	Тип данных	Возвращаемый тип	Пример	Результат
+	сложение	integer, real	integer, real	X:=5+5.7;	10.7
-	вычитание	integer, real	integer, real	X:=6-3.5;	2.5
*	умножение	integer, real	integer, real	X:=2*2;	4
/	деление	integer, real	real	X:=3/2;	1.5
Div	целочисленное деление (отбрасывается десятичная часть результата)	integer	integer	X:=3 div 2;	1
Mod	целочисленное деление (отбрасывается целая часть результата)	integer	real	X:=3 mod 2;	0.5

Логические операции

В результате выполнения любой из логических операций мы можем ожидать только одно из двух возможных значений: Да (`True`) или Нет (`False`).

Таблица 1.10. Логические операции

Оператор	Операция	Пример	Результат
	Операции сравнения `var x : INTEGER = 6;`		
=	Сравнение	X=5	FALSE
<>	Неравенство	X<>5	TRUE

Оператор	Операция	Пример	Результат
>	Больше чем	X>5	TRUE
<	Меньше чем	X<5	FALSE
>=	Больше или равно	X>=5	TRUE
<=	Меньше или равно	X<=5	FALSE
	Логические операции `var x : boolean;`		
NOT	Логическое отрицание	x:= NOT True;	FALSE
		x:= NOT False;	TRUE
		x:= NOT Null;	NULL
AND	Логическое умножение (конъюнкция, логическое И) для двух выражений	x:= True AND True;	TRUE
		x:= True AND False;	FALSE
		x:= True AND Null;	NULL
		x:= False AND True;	FALSE
		x:= False AND False;	FALSE
		x:= False AND Null;	FALSE
		x:= Null AND True;	NULL
		x:= Null AND False;	FALSE
		x:= Null AND Null;	NULL
OR	Логическое ИЛИ (сложение) для двух выражений	x:= True OR True;	TRUE
		x:= True OR False;	FALSE
		x:= True OR Null;	TRUE
		x:= False OR True;	TRUE
		x:= False OR False;	FALSE
		x:= False OR Null;	NULL
		x:= Null OR True;	TRUE
		x:= Null OR False;	NULL
		x:= Null OR Null;	NULL
XOR	Исключающее ИЛИ для двух выражений	x:= True XOR True;	FALSE
		x:= True XOR False;	TRUE
		x:= False XOR True;	TRUE
		x:= False XOR False;	FALSE

Побитовые операции

К побитовым операторам относят операторы для работы с отдельными битами переменной.

Таблица 1.11. Побитовые операции

Оператор	Операция	Пример	Результат
SHL	Поразрядный сдвиг влево (после оператора shl указывается количество разрядов)	01101 SHL 1	11010
SHR	Поразрядный сдвиг вправо	11010 SHR 1	01101

Оператор перехода

Воспринимайте оператор перехода GOTO как снегоуборочную машину летом и не более того. Он существовал раньше, живет в настоящем и, возможно, останется в будущем, но не более чем как дань традициям прошлого. Применение оператора GOTO в исходном тексте программы снизит ее читаемость и вынудит компилятор создать далеко не самый эффективный код. Я не знаю ни одного программиста на Delphi, который бы не мог создавать хорошие приложения без этого оператора. Поэтому воспринимайте GOTO просто как недоразумение, оставшееся от древних версий Delphi для обеспечения обратной совместимости.

Оператор вызова

Оператор вызова предназначен для передачи управления внешней процедуре, функции или методу, обработке их кода с возвратом к выполнению оператора, следующего за оператором вызова.

Если вызываемая процедура (функция) требует каких-либо параметров, то передавайте их в круглых скобках после имени процедуры (функции). Подробнее оператор вызова будет рассмотрен в разделе «Процедуры и функции».

Составной оператор begin..end

Зарезервированные слова begin и end означают соответственно начало и конец тела блока. Выражения, заключенные внутри составного оператора, могут рассматриваться Delphi как единый оператор.

Условный оператор if..then

Задачей оператора if..then (если ... тогда) является выполнение действия по условию, причем условие должно быть булевого типа: Да/Нет. Синтаксис условного оператора следующий:

```
if <булево условие> then <оператор>;
```

Например, при выполнении условия X > 5 переменной Y присваивается значение Y+1.

```
if X>5 then Y := Y+1;
```

Условный оператор if..then допускает включение в свою конструкцию ключевого слова else (иначе):

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

Например:

```
if X>5 then Y := Y+1 else Y := Y-1;
```

Оператор-селектор case .. of

Конструкция оператора-селектора case..of следующая:

```
case <селектор> of
     <константа1> : <оператор1>;
     <константа2> : <оператор2>;
     <константа3> : <оператор3>;
else <оператор>;
end;
```

Задачи секции else *внутри селектора* case *аналогичны задачам* else *в операторе* if .. then .. else. *Если селектору не соответствует ни одна из констант, будет выполнен оператор, следующий за словом* else. *Если же в конструкции отсутствует ключевое слово* else*, будет выполнена следующая за оператором строка кода.*

В роли селектора могут выступать переменная, выражение или функция, но они обязательно должны быть порядкового типа. Недопустимы селекторы строкового и действительного типов. Оператор case осуществляет проверку на равенство значений селектора и констант оператора. Если значение селектора совпадает со значением константы, выполняется соответствующий константе оператор.

```
case MyValue of
     1 : X := 1;
     2 : X := Y+3;
else X := 0;
end;
```

Оператор case обладает большей наглядностью, чем группа операторов if..then. Но это не единственное из его достоинств. Еще одним преимуществом селектора case..of считается возможность группировки констант.

```
var Ch:Char;
. . .
case ch of
     'A' .. 'D' : <оператор1>;
     'E'        : <оператор2>;
else <оператор3>;
end;
```

При использовании селектора case *следите за тем, чтобы диапазоны значений констант не пересекались!*

```
case value of
3     : Y:=value-3;
1..10 : Y:= value+1;    //ошибка!!! значение 3 входит в диапазон 1..3
end;
```

Операторы обработки циклов

Циклы предназначены для многократного выполнения оператора (группы операторов), заключенного внутри циклической конструкции. Delphi предлагает три различных типа циклов: `for..do`, `repeat..until` и `while..do`.

Цикл с параметром for..do

Цикл с параметром `for..do` применяется в тех случаях, когда заранее известно количество повторений, необходимых для выполнения каких-то действий. Синтаксическая конструкция выглядит следующим образом:

```
for <параметр цикла>:=<стартовое значение>
to (downto) <конечное значение> do <оператор>;
```

Отличие ключевого слова `to` *от* `downto` *в том, что при использовании в цикле слова* `to` *параметр цикла увеличивает свое значение на 1, а при использовании* `downto` *– уменьшает на 1.*

Параметром цикла может быть любая порядковая переменная. При каждом проходе цикла она получает приращение (уменьшение) на единицу. Цикл продолжается до тех пор, пока значение параметра не достигнет конечного значения. Например, требуется обнулить все элементы массива:

```
MyArray : array[0..99] of integer;
for X:=0 to 99 do MyArray[X]:=0;
```

Если же шаг цикла отличен от 1 или заранее неизвестно количество повторений тела цикла, вместо цикла с параметром следует применять цикл с предусловием или цикл с постусловием.

Ни в коем случае не допускается изменять значения параметра внутри тела цикла `for..do`. *Ведь это не более чем счетчик количества итераций. Если же ваша интуиция и логика программы подсказывают, что необходимо найти решение, в котором параметр цикла будет выступать не только в виде банального счетчика, но и в роли управляемого параметра, то вместо цикла* `for..do` *надо использовать цикл* `while..do` *или* `repeat..until`.

Цикл с предусловием while..do

Особенность цикла с предусловием `while..do` заключается в том, что код внутри тела цикла будет выполняться до тех пор, пока соблюдается условие, описанное в заголовке цикла. Конструкция цикла выглядит так:

```
while <условие - логическое выражение> do <оператор>;
```

Условие цикла задается в виде булевого выражения:

```
X :=0;
while X<=99 do      //выполнять, пока значение X не превышает 99
begin
    MyArray[X]:=0;
    X:=X+1;
end;
```