Справочник по RemObjects Pascal Script

Автор: Павел Дуборкин Редакция: 31.03.2018

Оглавление

Введение

Скриптовый язык программирования Pascal Script был разработан компанией RemObjects Software. Основное назначение языка – расширение функционала приложений с помощью скриптов, используя интерфейс прикладного программирования (API) приложения. Pascal Script является упрощенной версией языка Object Pascal. Это процедурный язык программирования с поддержкой классов. Т. е. в Pascal Script можно использовать определенные в API классы, но создавать свои нельзя.

В данном справочнике я постарался собрать полную информацию о синтаксисе языка и его особенностях. Большая часть информации была взята с других ресурсов, а именно:

1. Иллюстрированный самоучитель Delphi 6.
2. Программное обеспечение – Паскаль.
3. Turbo Pascal 7.

Материал рассчитан на читателей уже имеющих некоторый опыт программирования.

1. Элементы языка

Элементы языка - это минимальные неделимые ее части, еще 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».

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 идентификаторы 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. Замечу, что 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;
```

чтобы вместо длинных чисел 1048576 (1024*1024) и 1073741824 (1024*1024*1024) писать, соответственно, Mbyte и Gbyte. 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;
```

Здесь 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.

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 (метка). Например:

```
Label
   Loop;
begin
   goto Loop;
   // Программист требует передать управление
   // оператору, помеченному меткой Loop.
   …
   // Эти операторы будут пропущены
   Loop:
   // Оператору, идущему за этой меткой,
   // будет передано управление
   …
end;
```

Подпрограммы - это специальным образом оформленные фрагменты программы. Замечательной 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.

В Object Pascal есть два сорта 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. Если, например, определена функция

```
function MyFunction: Integer;
```

и переменная

```
var X: Integer;
```

то возможен такой оператор присваивания:

```
X := 2 * MyFunction - 1;
```

Имя процедуры нельзя использовать в выражении, т. к. процедура не имеет связанного с нею результата:

```
procedure MyProcedure;
…
X := 2 * MyProcedure - 1; // Ошибка!
```

2. 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).

Паскаль содержит набор ключевых слов, которые имеют определенное назначение. Имена идентификаторов не должны совпадать с ключевыми словами. Список ключевых слов:

and	**for**	**set**
array	**function**	**shl**
as	**goto**	**shr**
begin	**if**	**string**
break	**in**	**then**
case	**is**	**to**
const	**label**	**try**
continue	**mod**	**type**
div	**nil**	**until**
do	**not**	**uses**
downto	**of**	**var**
else	**or**	**while**
end	**procedure**	**with**
except	**program**	**xor**
exit	**record**	
finally	**repeat**	

3. Идентификаторы

Идентификаторы в Object Pascal – это имена констант, переменных, меток, типов, объектов, классов, свойств, процедур, функций, модулей, программ и полей в записях. Идентификаторы могут иметь произвольную длину.

Идентификатор всегда начинается буквой, за которой могут следовать буквы и цифры. Напомню, что буквой считается также символ подчеркивания, поэтому идентификатор может начинаться этим символом и даже состоять только из одного или нескольких символов подчеркивания. Пробелы и специальные символы алфавита не могут входить в идентификатор.

Примеры правильных идентификаторов:

```
a
MyProgramIsBestProgram
ALPHA
date_27_sep_39
_beta
```

Примеры неправильных идентификаторов:

```
1 Program     // начинается цифрой
block#1       // содержит специальный символ
My Prog       // содержит пробел
mod           // зарезервированное слово
```

4. Константы

В качестве констант в Object Pascal могут использоваться целые, вещественные и шестнадцатеричные числа, логические константы, символы и строки символов, конструкторы множеств и признак неопределенного указателя **nil**.

Целые числа записываются со знаком или без него по обычным правилам и могут иметь значение в диапазоне от -2^{63} до $+2^{63}-1$.

Примечание. Следует учесть, что, если целочисленная константа выходит за указанные границы, то ее значением будет 0. Такие константы должны записываться с десятичной точкой, т. е. определяться как

вещественные числа.

Вещественные числа записываются со знаком или без него с использованием десятичной точки и/или экспоненциальной части.

Экспоненциальная часть начинается символом **e** или **E**, за которым могут следовать знаки + или - и десятичный порядок. Символ **e** означает десятичный порядок и имеет смысл «умножить на 10 в степени». Например:

```
3.14E5 – 3,14 умножить на 10 в степени 5;
-17e-2 – минус 17 умножить на 10 в степени минус 2
```

Если в записи вещественного числа присутствует десятичная точка, перед точкой и за ней должно быть хотя бы по одной цифре. Если используется символ экспоненциальной части **e**, за ним должна следовать хотя бы одна цифра десятичного порядка.

Шестнадцатеричное число состоит из шестнадцатеричных цифр, которым предшествует знак доллара $ (код символа 36). Диапазон шестнадцатеричных чисел - от $FFFFFFFFFFFFFFFF до $ $7FFFFFFFFFFFFFFF.

Логическая константа - это либо слово **false** (ложь), либо слово **true** (истина).

Символьная константа - это любой символ ПК, заключенный в апострофы:

```
'z' - символ z
'Ф' - символ Ф
```

Если необходимо записать собственно символ апострофа, он удваивается:

```
'''' - символ ' (апостроф)
```

Допускается использование записи символа путем указания его внутреннего кода, которому предшествует символ # (код 35), например:

```
#97 - символ a
#90 - символ Z
#39 - символ '
#13 - символ возврата каретки CR
```

Строковая константа – любая последовательность символов (кроме символа **CR** - возврат каретки), заключенная в апострофы. Если в строке нужно указать сам символ апострофа, он удваивается, например:

```
'Это - строка символов'
'That''s string'
''
```

Строка символов может быть пустой, т. е. не иметь никаких символов в обрамляющих ее апострофах. Строку можно составлять из кодов нужных символов с предшествующими каждому коду символами #, например:

```
строка #83#121#109#98#111#108 эквивалентна строке «Symbol»
```

Наконец, в строке можно чередовать части, записанные в обрамляющих апострофах, с частями, записанными кодами. Таким способом можно вставлять в строки любые управляющие символы, в том числе и символ **CR** (код 13), например:

```
#7'Ошибка !'#13'Нажмите любую клавишу ...'#7
```

5. Объявление констант

Константы объявляются в разделе **const**:

```
const
   MAX_LEN = 150;
   Msg = 'Привет, МИР!!!';
```

В модуле может быть неограниченное количество разделов **const**. Область видимости констант – от начала объявления до конца модуля.

Примечание. Pascal Script не поддерживает в объявлении констант конструкторы множеств, массивов, неопределенный указатель **nil**, вызовы функций.

6. Объявление переменных

Переменные объявляются в разделе **var**. Переменные делятся на локальные и глобальные. Пример:

```
var
   MyVar: Integer;  // Пример глобальной переменной. Переменная видима в пределах модуля.

procedure MyProc;
var
   s: string;       // Пример локальных переменных.
   i, j: integer;   // Переменные видимы в пределах процедуры.
begin
   …
end;
```

Ключевого слова **var** является началом раздела объявления переменных. Переменные одного типа можно перечислять через запятую. В модуле может быть неограниченное количество разделов **var**. Область видимости переменных – от начала объявления до конца модуля.

7. Выражения

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 более 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:

```
y
21 * (a + b) * c
sin(t)
a > 2
not Flag and (a = b)
nil
```

8. Операции

В Pascal Script определены следующие операции:

- унарные not, @;
- мультипликативные *, /, div, mod, and, shl, 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, причем будут вычисляться только достаточные для однозначного определения результата операции.

Унарная операция @ используется при назначении обработчиков событий, она возвращает адрес точки входа в процедуру. Пример:

```
procedure AfterScrollHandler(Sender: TObject);
begin
end;

procedure Form_Create;
begin
   Self.OnAfterScroll := @AfterScrollHandler;
end;
```

В языке определены следующие логические операции:

- not – логическое НЕ;
- and – логическое И;
- or – логическое ИЛИ;
- xor – исключительное ИЛИ.

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 таблице:

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

К логическим операциям в Паскале 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 j – выражения любого целого типа.

Логические операции над логическими данными дают результат логического типа по правилам, указанным в таблице:

Операнд 1	Операнд 2	not	and	or	xor
True	-	False	-	-	-
False	-	True	-	-	-
False	False	-	False	False	False
False	True	-	False	True	True
True	False	-	False	True	True
True	True	-	True	True	False

Операция отношения **in** применяется к двум операндам. Первым (левым) операндом должно быть выражение любого порядкового типа, вторым - множество, состоящее из элементов того же типа, или идентификатор множественного типа. Результат операции будет True, если левый операнд принадлежит множеству. Пример:

```
if Self.State in [dsInsert, dsEdit] then …
```

Определение типа результата операции

Если в операции участвуют операнды разных типов, то необходимо определить тип достаточный для вмещения результата. К сожалению, Pascal Script не всегда корректно определяет тип результата. Тип результата будет равен типу левого операнда. Пример:

```
var
   b: Byte;
   i: Integer;
begin
   b := 201;
   i := 2;
   Debug(i * b);           // 402
   Debug(b * i);           // 146 = 402 - 256, т. к. тип Byte в пределах 0..255
end;
```

Если в операции участвует целый тип и вещественный, то результат будет вещественного типа независимо от порядка операндов. В следующей таблице приводится несколько примеров того, как Pascal Script определяет тип результата:

Тип левого операнда	Тип правого операнда	Тип результата
Byte	Integer	Byte
Word	Cardinal	Word
Integer	Byte	Integer
Integer	Single	Single
Double	Byte	Double
Byte	Double	Double
Single	Double	Single
Double	Single	Double

Чтобы избежать ошибок в вычислениях, делайте явное приведение к нужному типу. Приведение к типу

похоже на вызов функции, имя функции совпадает с типом:

```
Debug(Integer(b) * i);     // 402
```

Если в выражении используются числовые константы, то их тип зависит от значения: для целых констант минимальный тип **Integer**, для вещественных констант тип всегда **Extended**.

Значение константы	Тип константы
5	Integer
> 2147483647	Cardinal
> 4294967295	Int64
0.1	Extended
1000234234.023	Extended

К сожалению, движок не всегда корректно определяет тип отрицательных констант. Например, -2147483648 будет считаться типом Cardinal, хотя этот тип в принципе не может иметь отрицательные значения, поэтому результат вычислений будет неправильным.

```
var
   i64: Int64;
begin
   Debug(Int64(-2147483648));          // !!! 2147483648 - с константами там не проходит
   i64 := -2147483648;
   Debug(i64);                         // !!! 2147483648 - и даже так
   Debug(-2147483648.0);               // -2147483648 - так правильно
end;
```

Операции деления / и **div** тоже имеют свои особенности. Дело в том, что в Pascal Script эти операции являются идентичными и тип результата определяется по алгоритму указанному в начале параграфа.

```
Debug(5 / 3);              // 1
Debug(5 div 3);            // 1
Debug(5.0 / 3);            // 1,666…
Debug(5.0 div 3);          // 1,666…
```

9. Операторы языка

Оператор является неделимым элементом программы, который дает возможность выполнять определенные алгоритмические действия. Отличием оператора, по отношению к другим элементам, является то, что под ним всегда подразумевается какое-то действие. В языке Паскаль операторы состоят из служебных слов. Операторы, используемые в программе, отделяются между собой и от других элементов программы символом (;). Все операторы языка Паскаль можно условно разбить на две группы:

- простые;
- структурированные.

Простые операторы – это операторы, не содержащие в себе других операторов. К ним относятся:

- оператор присваивания (:=);
- обращение к подпрограмме;
- оператор безусловного перехода (GOTO);

- пустой оператор.

Структурированные операторы – это операторы, которые содержат в себе другие операторы. К ним относятся:

- составной оператор;
- операторы условий (IF, CASE);
- операторы цикла (FOR, WHILE, REPEAT);
- оператор присоединения (WITH).

9.1. Оператор присваивания

С помощью этого оператора переменной или функции присваивается значение выражения. Для этого используется знак присваивания := , слева от которого записывается имя переменной, которой присваивается значение, а справа – выражение, значение которого вычисляется перед присваиванием. Пример:

```
X := Y;
Z := A + B;
Res := (I>0) and (I < 100);
I := Sqr(J) + I * K;
```

9.2. Оператор безусловного перехода

Синтаксис:

```
goto [метка];
```

Goto – зарезервированное слово в языке Паскаль. [метка] – это произвольный идентификатор, который позволяет пометить некий оператор программы и в дальнейшем сослаться на него. Метка располагается перед помеченным оператором и отделяется от него двоеточием (:). Один оператор можно помечать несколькими метками. Они так же отделяются друг от друга (:). Перед тем как использовать метку в разделе оператора ее необходимо описать в разделе **label** (раздел описания). Пример:

```
procedure MyProc;
label
   Lbl1, Lbl2, met1;
   …
begin
   …
Lbl1:
   …
   goto met1;
Lbl2:
   …
   goto Lbl1;
Met1:
   …
end;
```

Действие **goto** передает управление соответствующему помеченному оператору. При использовании меток нужно руководствоваться следующими правилом: метка должна быть описана в разделе

описаний и все метки должны быть использованы.

Оператор **goto** противоречит принципам технологии структурного программирования. Современные языки программирования не имеют в своем составе такого оператора, и в его использовании нет необходимости.

Использовать оператор **goto** следует крайне осторожно. Широкое его применение без особых на то оснований ухудшает понимание логики работы программы. Безусловный переход можно осуществлять далеко не из каждого места программы и не в любое место программы. Так, нельзя с помощью этого оператора перейти из основной программы в подпрограмму или выйти из подпрограммы, не рекомендуется осуществлять переход внутрь структурированного оператора, т. к. он может дать неправильный результат, и т. д.

9.3. Пустой оператор

Пустой оператор не выполняет никакого действия и никак не отображается в программе (за исключением, быть может, метки или точек с запятыми, отделяющих пустой оператор от предыдущих или последующих операторов). Он может потребоваться для осуществления на него безусловного перехода.

9.4. Составной оператор

Составной оператор – это последовательность произвольных операций в программе, заключенная в так называемые операторные скобки (Begin-End). Синтаксис:

```
begin
   [оператор1];
   [оператор2];
   …
   [операторN];
end;
```

Он может потребоваться в тех случаях, когда в соответствии с правилами построения конструкций языка можно использовать один оператор, а выполнить нужно несколько действий. В такой составной оператор входит ряд операторов выполняющих требуемые действия.

В дальнейшем везде, где будет указываться, что можно использовать один оператор, им может быть и составной оператор.

Отдельные операторы внутри составного оператора отделяются друг от друга точкой с запятой.

9.5. Условный оператор IF

Условный оператор IF 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 Паскаль.

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**. Обратите внимание, что перед **else** не должно быть точки с запятой.

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 …
```

Примечание. 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 …
```

С учетом приоритета операций компилятор будет транслировать такую строку как:

```
if a > (b and b) <> 0 then …
```

Часть **else** [оператор2] условного оператора может быть опущена. Тогда при значении **True** условного выражения выполняется [оператор1], в противном случае этот оператор пропускается:

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

В этом примере переменная **Y** всегда будет иметь значение переменной **X**, а в **Max** запоминается максимальное значение **X**.

Вслед за **else** может идти другой оператор **if**. Таким образом, можно составить множественное ветвление:

```
if [условие1] then
   …
else if [условие2] then
   …
else if [условие3] then
   …
```

```
else
   …
```

9.6. Оператор выбора CASE

Оператор реализует множественное ветвление. В определенных ситуациях **case** использовать удобнее, чем **if**. Синтаксис оператора:

```
case [выражение] of
   [выражение_1]:[оператор_1];
   [выражение_2]:[оператор_2];
   …
   [выражение_N]:[оператор_N];
   else [оператор];
end;
```

Выполнение операторов зависит, будет ли [выражение] равно [выражение_1], [выражение_2] или [выражение_N]. Например, [оператор_2] будет выполнен только в случае, когда [выражение]=[выражение_2]. Если ни одно из условий не выполняется, то будет выполнен [оператор], следующий за блоком else. Если блока **else** нет, то оператор **case** просто пропускается. Оператор **case** эквивалентен следующему оператору **if**:

```
if [выражение]=[выражение_1] then [оператор_1]
else if [выражение]=[выражение_2] then [оператор_2]
…
else if [выражение]=[выражение_N] then [оператор_N]
else [оператор]
```

9.7. Операторы цикла

Циклической алгоритмической структурой считается такая структура, в которой некоторые действия выполняются несколько раз. В программировании имеются два вида циклических структур: цикл с параметром и итерационный цикл.

В цикле с параметром всегда имеются так называемые параметры цикла. Иногда цикл с параметром называют регулярным циклом. Характерной чертой является то, что число циклов и повторений можно определить до выполнения цикла.

В итерационном цикле невозможно определить число циклов до его выполнения. Он выполняется до тех пор, пока выполняется условие продолжение цикла.

В языке Паскаль имеются три оператора, реализующих циклические вычислительные структуры:

- счетный оператор **for**. Он предназначен для реализации цикла с параметром и не может быть использован для реализации итерационного цикла;
- оператор цикла с предусловием **while**;
- оператор цикла с постусловием **repeat**.

Последние два ориентированы на реализацию итерационного цикла, однако их можно использовать и для реализации цикла с параметром.

Оператор FOR

Синтаксис оператора:

```
for [параметр] := [начальное значение] to [конечное значение] do [оператор]
```

For, **to**, **do** – служебные слова. Параметр – переменная порядкового типа. Начальное и конечное значения должны быть того же типа, что и параметр.

На первом шаге параметр цикла принимает начальное значение затем осуществляется проверка: параметр цикла меньше или равен конечному значению. Это условие является условием продолжения цикла. Если выполнено, то цикл продолжает свою работу и выполняется [оператор], после чего параметр цикла увеличивается (уменьшается) на единицу. Затем с новым значением параметр цикла, проверяется условие продолжения цикла. Если оно выполняется, то действия повторяются. Если условие не выполняется, то цикл прекращает свою работу.

Оператор **for** существенно отличается от аналогичных операторов в других языках программирования. Отличия следующие:

- Оператор может не выполниться ни разу, поскольку проверка условия продолжения цикла выполняется до тела цикла;
- Шаг изменения параметра цикла постоянный и равен 1;
- Тело цикла в операторе **for** представлено одним оператором. В том случае, если действие тела цикла требует более одного простого оператора, то эти операторы необходимо превратить в один составной оператор посредством операторных скобок (BEGIN-END);
- Параметр цикла может быть только переменой порядкового типа.

Цикл **for** может уменьшать параметр. В этом случае синтаксис следующий:

```
for [параметр] := [конечное значение] downto [начальное значение] do [оператор]
```

Оператор WHILE (цикл с предусловием)

Синтаксис оператора:

```
while [условие] do [оператор]
```

While, **do** – служебные слова, условие – логическое выражение.

Оператор **while** работает следующим образом: вначале работы проверяется результат логического условия. Если результат истина, то выполняется оператор, после которого осуществляется возврат на проверку условия с новым значением параметров в логическом выражении условия. Если результат ложь, то осуществляется завершение цикла.

При работе с **while** надо обратить внимание на его свойства:

- условия, использованные в **while**, являются условием продолжения цикла;
- в теле цикла всегда происходит изменение значения параметра входящего в выражение условия;
- цикл **while** может не выполняться ни разу, поскольку проверка условия в продолжение цикла выполняется до тела цикла.

В примере ниже выполняется обработка записей формы. Пока не достигнуто окончание набора данных формы, выполнять обработку:

```
while Self.EOF = False do
begin
   …
   Self.MoveNext;
end;
```

Оператор REPEAT (цикл с постусловием)

Синтаксис оператора:

```
repeat
   [тело цикла];
until [условие];
```

Оператор **repeat** работает следующим образом: сначала выполняются операторы тела цикла, после чего результат проверяется логического условия. Если результат ложь, то осуществляется возврат к выполнению операторов очередного тела цикла. Если результат истина, то оператор завершает работу.

Оператор **repeat** имеет следующие особенности:

- в **repeat** проверяется условие завершения цикла и если условие выполняется, то цикл прекращает работу;
- тело цикла всегда выполняется хотя бы один раз;
- параметр для проверки условия изменяется в теле цикла;
- операторы тела цикла не надо заключать в операторские скобки (BEGIN-END), при этом роль операторных скобок выполняют **repeat** и **until**.

Прерывание цикла

Любой из циклов можно прервать оператором **break**.

```
while [условие] do
begin
   …
   if [условие выхода из цикла] then Break;
   …
end;
```

10. Процедуры и функции

Процедуры и функции представляют собой относительно самостоятельные фрагменты программы, оформленные особым образом и снабженные именем. Упоминание этого имени в тексте программы называется вызовом процедуры (функции). Отличие функции от процедуры заключается в том, что результатом исполнения операторов, образующих тело функции, всегда является некоторое значение, поэтому обращение к функции можно использовать в соответствующих выражениях наряду с переменными и константами. Обобщенно процедуры и функции будем называть подпрограммами.

Подпрограммы делятся на встроенные и пользовательские. Встроенные подпрограммы входят в состав API DataExpress, пользовательские определяются разработчиком в скрипте. Синтаксис определения процедуры и функции:

```
// Определение процедуры
procedure MyProc([список параметров]);
begin
   …
end;
// Определение функции
function MyFunc([список параметров]): [тип данных];
begin
   …
```

```
end;
```

Верхняя строка процедуры называется заголовком процедуры и состоит из служебного слова **procedure**, названия функции, списка параметров и точки с запятой. Следом идут операторные скобки **begin-end**, называемые телом процедуры. После **end** обязательно ставится точка с запятой. Процедура может не иметь параметров, в этом случае ставить скобки необязательно.

Заголовок функции отличается тем, что вместо слова **procedure** пишется **function**, а после списка параметров или имени (если нет параметров) идет двоеточие и тип результата, который возвращает функция.

Вызов встроенной или пользовательской подпрограммы имеет следующий синтаксис:

```
Proc([список параметров]);
MyFunc([список параметров]);
x := a + MyFunc([список параметров]) – b …
y := MyFunc2 + 1000 - b;
```

Если подпрограмма не имеет параметров, то скобки необязательны.

Имя подпрограммы должно быть уникальным в пределах модуля и других подключаемых модулей и не должно совпадать с именами встроенных.

В подпрограмме могут быть объявлены локальные переменные, видимые только в теле подпрограммы. Раздел объявления переменных находится между заголовком и телом. Пример:

```
procedure MyProc(Param1: Integer);
var
   i, j: Integer;
begin
   i := 1;
   …
end;
```

Имена локальных переменных могут совпадать с именами глобальных переменных. В этом случае использоваться будет локальная переменная.

10.1. Параметры процедур и функций

Синтаксис описания параметров разберем на примере:

```
function MyFunc(x, y: Integer; a: Double; const S, Str: String; var R: String): Boolean;
```

Параметры в заголовке подпрограммы называются формальными параметрами. Параметры, передаваемые при вызове подпрограммы, называются фактическими.

Параметры одного типа отделяются запятой, после двоеточия указывается тип параметра. Параметры разных типов отделяются точкой с запятой. Список и тип формальных и фактических параметров должен совпадать. За этим следит компилятор.

Параметры, передаваемые в подпрограмму, делятся на:

- Параметры-значения. В примере это параметры **x**, **y**, **a**. В качестве фактических параметров могут передаваться константы, произвольные выражения, переменные. Результат вычисления параметра копируется в стек подпрограммы. Если в качестве параметра передается переменная, то изменения производятся над ее копией, а не оригиналом.
- Параметры-переменные. В примере это параметр **R**. Перед параметрами указывается служебное слово **var**. В качестве параметра может передаваться только переменная. Тип

фактического параметра должен строго соответствовать типу формального параметра. В процедуру или функцию передается ссылка на переменную. Все изменения производятся непосредственно над переменной.

- Параметры-константы. В примере это параметры **S**, **Str**. Перед параметрами указывается служебное слово **const**. Параметры-константы похожи на параметры-значения, только в подпрограмму передается ссылка на адрес в памяти, где располагается результат вычисления параметра. Компилятор не позволяет изменять в подпрограмме параметр-константу. Таким образом, не тратится время на копирование параметра в стек подпрограммы, как в случае с параметром-значением.

Пример вызова функции:

```
var
   a: Word;
   MyStr, MyResult: String;
   b: Boolean;
begin
   b := MyFunc(1, a, AnyFunc(20) + 30 / 0.5, MyStr, 'Hello, World', MyResult);
end;
```

В случае с параметрами-значениями и параметрами-константами тип формального и фактического параметра может быть совместимым. Если формальный определен как целое число, то фактический параметр может быть любого целого типа. Только нужно следить, чтобы формальный параметр мог «вместить» фактический параметр. В данном примере тип формального параметра **Y** «вмещает» переменную **a**, т. к. область допустимых значений больше. Целый формальный параметр несовместим с другими типами, в том числе с вещественными. А вот вещественный формальный параметр, совместим с целыми и вещественными.

В случае с параметрами-переменными требуется строгое соответствие типов, даже если формальный и фактический параметры являются родственными, например целыми. Т. е. если формальный параметр имеет тип **Integer**, то фактический параметр тоже должен иметь тип **Integer**, а не **Byte**, **Word** и т. д. Это касается и классов, даже если фактический параметр является потомком класса формального параметра.

10.2. Передача массивов в качестве параметров

Для передачи массива в подпрограмму должен быть определен пользовательский тип:

```
type
   TMyArr = array [1..20] of Integer;
   TMyArr2d = array [1..10] of array [1..5] of Byte;
   TDynaArr = array of String;

procedure MyProc(Arr: TMyArr; var Arr2d: TMyArr2d; DynaArr: TDynaArr);
begin
   …
end;

procedure AnyProc;
var
   A: TMyArr;
   A2d: TMyArr2d;
```

```
   DArr: TDynaArr;
begin
   …
   MyProc(A, A2d, DArr);
end;
```

Фактический параметр должен быть переменной массива.

Примечание. Компилятор не сообщит об ошибке, если объявить параметр-массив таким образом:

```
procedure MyProc(arr: array [1..10] of String);
begin
end;

procedure AnyProc;
var
   A: array [1..10] of String;
begin
   MyProc(a);              // type mismatch
end;
```

Однако во время выполнения вы получите ошибку «type mismatch».

В Паскале есть так называемые открытые массивы. Синтаксис формального параметра открытого массива следующий:

```
procedure MyProc(OpenArr: array of [тип]);
```

Открытый массив представляет собой одномерный массив. Нижняя граница массива всегда равна 0, даже если нижняя граница передаваемого массива отличается. Соответственно, верхняя граница равна длине массива минус 1. Для определения границ массива можно воспользоваться функциями **Low** и **High**. Пример:

```
procedure MyProc(Arr: array of Integer);
begin
   // Low(открытый массив) = 0
   for i := Low(Arr) to High(Arr) do
      …
end;

procedure AnyProc;
var
   arr: array [1..10] of Integer;
   arr2: array [-2..5] of Integer;
begin
   MyProc(arr);
   MyProc(arr2);
   MyProc([1,4,8,9,11,34,23]);
end;
```

Как видно из примера фактическим параметром может быть не только переменная, но и конструктор массива, который представляет собой перечисление элементов через запятую, заключенных в квадратные скобки.

Примечание. В движке не поддерживаются вариантные открытые массивы, хотя некоторые

встроенные подпрограммы имеют параметры такого типа (например, Format).

```
procedure Proc(arr: array of const);          // не поддерживается
```

10.3. Возврат значения функции

Функция имеет предопределенную переменную **Result**, в которую сохраняется возвращаемое значение. Пример:

```
function SumAB(A, B: Double): Double;
begin
   Result := A + B;
end;
```

10.4. Выход из подпрограммы

В любой момент можно выйти из подпрограммы. Для этого предусмотрен оператор **Exit**.

Примечание. В движке не поддерживается передача в EXIT возвращаемого значения.

```
Exit(-1);                          // Компилятор сообщит "internal error(20)"
```

11. Типы данных

Переменные бывают разных типов. Тип переменной определяет набор допустимых значений и операций. Типы делятся на простые, составные, строковые, варианты и классы.

11.1. Простые типы

Простые типы делятся на порядковые, вещественные и дата-время.

Порядковые типы имеют конечное множество возможных значений, их можно определенным образом упорядочить (отсюда и название) и сопоставить некоторое целое число.

Вещественные типы, строго говоря, тоже имеют конечное число значений, которое определяется форматом внутреннего представления вещественного числа. Однако количество возможных значений вещественных типов настолько велико, что сопоставить с каждым из них целое число (его номер) не представляется возможным.

Тип дата-время предназначен для хранения даты и времени. Фактически для этих целей он использует вещественный формат.

Порядковые типы

К порядковым типам относятся целые, логические, символьный и перечисляемый. 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).

К порядковым типам можно также применять функции:

pred(x) - возвращает предыдущее значение порядкового типа, которое соответствует порядковому номеру ord (X) -1, т. е. Ord(Pred(X)) = Ord(X) - 1;

succ(X) - возвращает следующее значение порядкового типа, которое соответствует порядковому номеру ord (X) +1, т. е. Ord(Succ(X)) = Ord(X) + 1.

Например, если в программе определена переменная:

```
var
   c : Char;
begin
   c := '5';
end;
```

то функция Pred(c) вернет символ '4', а функция Succ(C) - символ '6'.

Если представить себе любой порядковый тип как упорядоченное множество значений, возрастающих слева направо и занимающих на числовой оси некоторый отрезок, то функция pred(x) не определена для левого, а succ(X) – для правого конца этого отрезка.

Целые

Ниже приведена таблица поддерживаемых целых типов и диапазон допустимых значений:

Тип данных	Длина, байт	Диапазон допустимых значений
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
Cardinal	4	0 .. 4 294 967 295
LongWord	4	0 .. 4 294 967 295
Int64	8	$-9*10^{18}$ .. $+9*10^{18}$

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.

Логические

Логический тип может принимать 2 значения: **True** или **False**. К логическим типам относят: Boolean, ByteBool, WordBool, LongBool. Первый тип является стандартным типом Паскаля, а последние три введены для совместимости с Windows. В данной реализации языка функция Ord вернет для True – 1, для False – 0.

Примечание. В Pascal Script функция **Ord** для переменной типа **Boolean** также возвращает результат типа **Boolean**.

```
var
   b: Boolean;
   bb: ByteBool;
begin
   b := True;
   bb := True;
   Debug(Ord(b));                  // True
```

```
   Debug(Integer(Ord(b)));        // 1
   Debug(Ord(bb));                // 1
end;
```

Символьный

К символьному типу относится тип **Char**, вмещает в себя 1 байт, символы с кодом 0..255. Переменной этого типа можно присвоить один символ:

```
var
   c: Char;
   n: Byte;
begin
   c := 'A';
   c := #65;          // Код символа A
   n := Ord('E');     // Возвращает код символа
end;
```

Для кодировки символов используется таблица ASCII. Первая половина символов от 0 до 127 является стандартной и приведена в таблице. Вторая половина от 128 до 255 зависит от шрифта.

Код	Символ	Код.	Символ	Код.	Символ	Код	Символ
0	NUL	32	BL	64	@	96	'
1	ЗОН	33	!	65	A	97	a
2	STX	34	“	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	д
8'	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	*ƒ*	76	L	108	1
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	0	111	o

16	DEL	48	0	80	P	112	P
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC 4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	W
24	CAN	56	8	88	x	120	x
25	EM	57	9	89	Y	121	У
26	SUB	58	:	90	Z	.122	z
27	ESC	59	;	91	t	123	{
28	FS	60	<	92	\	124	1
29	GS	61	=	93	]	125	}
30	RS	62	>	94	Л	126	~
31	US	63	*f*	95	_	127	r

Символы с кодами 0..31 относятся к служебным кодам. Если эти коды используются в символьном тексте программы, они считаются пробелами.

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
   TColors = (Red, White, Blue);
var
   Color: TColors;
begin
   Color := Red;
end;
```

Применение перечисляемых типов делает программы нагляднее.

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.

Примечание. В Object Pascal присваивание переменным разных перечисляемых типов значений не своего типа приводит к ошибке несовместимости типов. Например:

```
type
   TColors = (Red, Green, Blue);
   TNum = (One, Two, Three, Four);

procedure Form_Create;
var
   n: TNum;
   c: TColors;
begin
   n := One;
   c := Four;        // Несовместимость типов
   Debug(Ord(c));    // Значение 3
end;
```

Однако компилятор Pascal Script не увидит ошибку и код будет работать. Будьте внимательны.

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. Pascal Script поддерживает следующие вещественные типы:

Тип данных	Длина, байт	Количество значащих цифр	Диапазон допустимых значений
Single	4	7-8	1.5*10e-45 .. 3.4*10e38
Double	8	15-16	5.0*10e324 .. 1.7*10e308
Extended	10	19-20	3.4*10-4951 .. 1.1*10e4932
Currency	8	19-20	+/-922 337 203 685 477,5807

Примечание. В DataExpress числовые поля имеют тип Double.

Сравнение вещественных чисел

Из-за того, что вещественные числа хранят приблизительное значение числа, их нельзя проверять на равенство с числовыми константами. Для сравнения используйте функцию SameValue. Пример:

```
var
   d: Double;
begin
   d := 0.2;
   // Это условие не выполнится
   if d = 0.2 then …
   // Можно так. Это условие выполнится
   if d - 0.2 < 0.001 then …
   // Так правильно сравнивать вещественные числа
```

```
   if SameValue(d, 0.2, 0) then …
end;
```

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. Дата определяется как количество суток, прошедших с 30 декабря 1899 года, а время - как часть суток, прошедших с 0 часов, так что значение 36444,837 соответствует дате 11.10.1999 и времени 20:05. Количество суток может быть и отрицательным, однако значения меньшие -693594 (соответствует дате 00.00.0000 от Рождества Христова) игнорируются функциями преобразования даты к строковому типу.

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. Например, 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. Например, оператор:

```
IbOutput.Caption := DateToStr(Date + 21);
```

поместит в метку IbOutput дату, соответствующую текущей дате плюс 3 недели. Чуть сложнее с исчислением времени. Например, чтобы добавить к текущему времени полтора часа, следует использовать выражение:

```
Time + StrToTime('1:30')
или
Time+1.5/24
```

11.2. Составные типы

К составным типам относятся: массивы, записи и множества. Они 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 (в реализации от RemObjects не проверялось).

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 суть данные одного типа (возможно, структурированного). Эти компоненты можно легко упорядочить и обеспечить доступ к любому из них простым указанием его порядкового номера, например:

```
var
   m: array [0..255] of Single;
   d: array [0..9] of Char;
begin
   …
   d[3] := 'a';
   m[17] := Ord(d[3])/10.0;
```

```
   …
end;
```

Объявление массива имеет следующий синтаксис:

```
<переменная>: array [<начало>..<конец>] of <тип данных>
```

Тип данных массива в свою очередь тоже может быть массивом. Таким образом можно описать n-мерный массив:

```
var
   Matrix2d: array [0..9] of array [0..7] of Integer;
   Matrix3d: array [0..9] of array [0..7] of array [0..10] of Char;
begin
   Matrix2d[0][1] := 10;                // Обращение к элементу
   Matrix3d[0][0][7] := 'J';            // массива.
end;
```

Примечение. Сокращенная запись многомерного массива, а также указание порядкового типа в качестве размерности, в Pascal Script не поддерживается. Пример:

```
var
   Arr2d: array [0..2,0..4] of Integer;        // Не поддерживается
   Arr: array [Byte] of Char;                  // Не поддерживается
begin
   Arr2d[1, 2] := 2;                           // Не поддерживается
end;
```

С помощью оператора присваивания можно копировать элементы массива в другой массив.

```
var
   a, b: array [1..5] of Integer;
   c: array [1..5] of Integer;
   d: array [1..7] of Integer;
begin
   …
   b := a;
   c := a;          // Компилятор пропустит, но во время выполнения произойдет ошибка
   d := a;          // несоответствия типов
   …
end;
```

Для массивов не определены операции отношения, сравнивать массивы можно только поэлементно:

```
for i := 0 to 5 do
   if a[i] <> b[i] then
      …
```

Динамические массивы

В предыдущем разделе были рассмотрены статические массивы, для которых длина задается при объявлении массива. Динамические массивы отличаются тем, что длина задается на этапе выполнения программы и может меняться. Динамические массивы объявляются следующим образом:

```
var
```

```
   a: array of Integer;
   a2: array of array of Double;
   a3: array of array of array of String;
```

Объявление динамического массива отличается тем, что не указываются границы массива в квадратных скобках. Для работы с динамическим массивом его надо инициализировать, т. е. задать длину. Делается это процедурой SetLength.

```
SetLength(a, 10);
```

При инициализации массиву выделяется память, и массив заполняется нулями. Нижняя граница динамического массива всегда равна 0, верхняя будет на единицу меньше его длины (в примере будет 9). Узнать длину массива можно функцией Length:

```
for i := 0 to Length(a) - 1 do
```

Работа с элементами динамического массива ничем не отличается от работы с элементами статического массива. Для работы с многомерным массивом надо задать размер всем его измерениям:

```
SetLength(a2, 10);
for i := 0 to Length(a2) - 1 do
   SetLength(a2[i], 7);

SetLength(a3, 5);
for i := 0 to Length(a3) - 1 do
begin
   SetLength(a3[i], 7);
   for j := 0 to Length(a3[i]) - 1 do
      SetLength(a3[i][j], 3);
end;
```

Так как элемент многомерного динамического массива сам является массивом, то в одном измерении могут быть массивы разной длины:

```
SetLength(a[0], 1);
SetLength(a[1], 2);
SetLength(a[2], 3);
```

Фактически идентификатор динамического массива ссылается на указатель, содержащий адрес первого байта памяти, выделенной для массива. Поэтому присваивание одного массива другому приводит к копированию ссылки на область памяти, занятую массивом, а не самого массива.

```
var
   a, b: array of Integer;
begin
   SetLength(a, 2);
   a[0] := 1;
   a[1] := 2;
   b := a;
   b[2] := 3;        // Фактически произойдет изменение элемента массива a - a[2]=b[2]
end;
```

Для копирования массива вам придется выделять память для нового массива и копировать массив поэлементно.

При изменении длины уже инициированного динамического массива по какому-либо его измерению сначала резервируется нужная для размещения нового массива память, затем элементы старого массива переносятся в новый, после чего освобождается память, выделенная прежнему массиву. Чтобы сократить дополнительные затраты времени, связанные с изменением границ большого динамического массива, следует сразу создать массив максимальной длины.

После окончания работы с массивом нужно освободить память занятую массивом. Для этого длина массива уменьшается до 0.

```
SetLength(a, 0);
// Сначала освобождаем память, занятую массивами-элементами двухмерного массива
for i := 0 to Length(a2) - 1 do
   SetLength(a2[i], 0);
// Теперь можно укоротить сам двухмерный массив.
SetLength(a2, 0);
```

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. Пример использования записи:

```
type
   TMyPoint = record
      X, Y: Integer;
   end;

   TMyData = record
      a, b: Integer;
      Arr: array [1..5] of String;
      Ch: Char;
      S: String;
      Pnt: TMyPoint;
   end;

var
   MyData: TMyData;

procedure Form_Create;
begin
   MyData.a := 2;
   MyData.b := 10;
   MyData.Arr[1] := 'Hello';
   MyData.S := 'World';
   MyData.Pnt.X := 10;
end;
```

Запись объявляется в разделе объявления пользовательских типов **type**. Поля записи могут быть произвольного типа, в том числе другими записями. Обращение к полю записи осуществляется через точку. Ключевое слово **with** упрощает доступ к полям записи:

```
with MyData do
   a := 2;
   b := 10;
   Arr[1] := 'Hello';
   S := 'World';
   Pnt.X := 10;
   // вложенный with
   with Pnt do Y := 8;
end;
// Так тоже можно
with MyData.Pnt do
   X := 5;
```

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 во 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 тогда и только тогда, когда все их элементы одинаковы, причем порядок следования элементов во 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.

Примечание. В Pascal Script допускается составление множеств только из перечисляемого типа.

Примеры использования множеств и операций над множествами:

```
type
   TMyButton = (bnOk, bnCancel, bnYes, bnNo);
   TMyButtons = set of TMyButton;
var
   MyButtons: TMyButtons;
begin
   // Присваиваем множество переменной. Множество представляет собой элементы в квадратных
   // скобках, разделенные запятой
   MyButtons := [bnOk, bnCancel];
   // Включаем во множество элемент. Получаем [bnOk, bnCancel, bnYes]
   Include(MyButtons, bnYes];
   // Исключаем из множеста элемент. Получаем [bnOk, bnYes]
   Exclude(MyButtons, bnCancel);
   // Пересечение множеств. Результатом будет множество, элементы которого есть в обоих
   // множествах. Получаем [bnYes]
   MyButtons := MyButtons * [bnYes, bnNo];
   // Объединение множеств. Первое множество дополняется недостающими элементами из второго
   // множества. Получаем [bnYes, bnNo]
   MyButtons := MyButtons + [bnYes, bnNo];
   // Разность множеств. Результат содержит элементы из первого множества, которые не
   // принадлежат второму. Получаем [bnNo]
```

```
   MyButtons := MyButtons - [bnYes];
   // Проверка эквивалентности. Получаем True, т. к. множества эквиваленты
   if MyButtons = [bnNo] then …
   // Проверка неэквивалентности
   if MyButtons <> [bnYes, bnNo] then …
   // Проверка вхождения. Возвращает True если первое множество включено во второе
   if MyButtons <= [bnYes, bnNo, bnCancel] then …
   // Проверка вхождения. Возвращает True если второе множество включено в первое
   if MyButtons >= [bnYes, bnNo, bnCancel] then …
   // С помощью оператора in определяем принадлежность элемента множеству. Слева от
   // оператора in может быть произвольное выражение
   if bnOk in MyButtons then …
end;
```

11.3. Строковые типы

Строки представлены типами String, AnyString, AnsiString, WideString и PChar. Для всех типов строк, кроме PChar, память выделяется динамически, по мере необходимости. Типы String, AnyString, AnsiString являются синонимами. Тип **String** можно представить, как массив символов. Подобно массиву можно обращаться к отдельным символам строки по их индексу. Первый символ строки имеет индекс 1. Примеры работы со строками:

```
var
   S: String;
begin
   S := 'hello, World';    // Строка в кодировке UTF-8
   S[1] := 'H';            // Меняем h на H. Индекс первого символа равен 1.
   S := S + '!!!';         // Сложение строк.
   S := 'Привет, МИР!!!'; // Строка в кодировке UTF-8
   Debug(S[1]);            // Будет выведен знак «?», потому что для кодирования символов
                           // кириллицы требуется более одного байта.
end;
```

Строка может хранить данные в кодировке ANSI или UTF-8. В кодировке ANSI символ занимает 1 байт. Символ в кодировке UTF-8 может занимать от 1 до 4 байт. Если строка хранит данные в этой кодировке, то обращение к элементам массива строки уместно только, если обрабатываются символы первой половины таблицы ASCII с кодами от 1 до 127. В ином случае нужно применять функции для работы со строками UTF-8.

Примечание. UTF-8 является основной кодировкой, база данных и скрипты используют именно эту кодировку.

К строкам применимы операции сравнения. Сравнение происходит посимвольно.

```
S := 'Hello World';
S2 := 'Hello world';
// Условие ложно, потому что W <> w
if S = S2 then …
// Условие истинно, потому что W < w
if S < S2 then …
```

Тип **WideString** хранит строки в кодировке UTF-16. Каждый символ кодируется двумя байтами. Несмотря на то, что каждый символ занимает 2 байта, вы можете обратиться к символу по индексу, как

если бы для кодирования использовался 1 байт. Строки этого типа используются OLE-объектами и функциями Windows API.

```
var
   S: WideString;
begin
   S := 'Hello, МИР!!!';
   Debug(S[8]);      // Символ «М»
end;
```

Тип **PChar** представляет строки с завершающим нулем. Переменная типа **PChar** содержит указатель на массив строк. Концом строки считается символ с кодом #0. В чистом виде тип **PChar** не используется, т. к. Pascal Script не работает с указателями. Предположительно, он введен только для совместимости с функциями Windows API.

Pascal Script автоматически преобразует строку из одного типа в другой, так что работа со строками разных типов является прозрачной для разработчика.

```
function MessageBoxA(hWnd: Integer; lpText, lpCaption: String; uType: Cardinal): Integer;
external 'MessageBoxA@user32.dll stdcall';
function MessageBoxW(hWnd: Integer; lpText, lpCaption: WideString; uType: Cardinal):
Integer; external 'MessageBoxW@user32.dll stdcall';


var
   S: String;
   WS: WideString;
begin
   S := 'Строка';
   WS := 'Привет, МИР!!!';
   WS := S;
   S := WS;
   MessageBoxA(0, S, 'Сообщение', 0);
   MessageBoxW(0, WS, 'Сообщение', 0);
end;
```

11.4. Варианты

Переменная вариантного типа может принимать хранить значение следующих типов: целый, вещественный, логический, строка, дата-время, OLE-объект. Вариант представляет собой структуру, в которой хранятся сведения о типе и значении переменной. В выражениях и при передаче параметров в процедуры происходит попытка преобразования варианта к нужному типу. Если это не удается, будет ошибка “invalid variant type cast”. Вариант может хранить специальное значение **Null** (пусто, неизвестно). Любая арифметическая операция с **null** даст в результате **null**. Попытка автоматического преобразования **null** к какому-либо типу (кроме Variant) приведет к ошибке “could not convert variant of type (Null) into type (…).

```
var
   V1, V2, V3, V4, V5: Variant;
begin
   V1 := 1;
   V2 := '4';
   V3 := V + V2;           // Результат 5 или '5'
```

```
   V4 := V3 + Null;        // V4 = Null
   V5 := V1 + 'sdfsdf';    // invalid variant type cast
   MsgBox('Сообщение', V4) // Could not convert variant of type (Null) into type (String)
end;
```

Варианты широко используются при работе с OLE-объектами. С помощью специальных функций можно узнать значение какого типа хранится в варианте или преобразовать значение к нужному типу.

11.5. Классы

Класс в Object Pascal – это специальный тип, который содержит поля, методы и свойства. Класс и объект являются ключевыми понятиями в объектно-ориентированном программировании (ООП). Класс определяет структуру и функциональность, а объект является конкретным экземпляром класса, который работает в соответствии с логикой заданной классом. Класс представляет собой относительно самостоятельную часть кода, выполняющую определенную задачу. Например, класс TStringList реализует работу со списками строк, TFileStream – обеспечивает работу с файлами, TdxForm – обеспечивает работу с полями формы, автоматические вычисления и многое другое. Классы позволяют структурировать код, сделать его лучше читаемым, обеспечивают повторное использование кода, облегчают сопровождение программ. Программист может использовать сторонние классы, не вникая в их реализацию. Ему достаточно изучить интерфейсную часть класса: набор методов и свойств. Методы выполняют какие-либо действия, свойства влияют на характеристики и поведение объекта. Например, метод SaveToFile класса TStringList сохраняет список строк в файл на диске, свойство Caption класса TdxButton позволяет узнать или поменять надпись на кнопке. Пример использования класса:

```
var
   SL: TStringList;
begin
   // До создания объекта переменная SL = nil.
   SL := TStringList.Create;
   SL.Delimiter := ' ';
   SL.DelimitedText := 'Один Два Три Четыре Пять';
   Debug(SL.Count);
   SL.SaveToFile('d:\data.txt');
   SL.Free;
   // После удаления объекта переменная SL содержит указатель на несуществующий объект.
   // Попытка обратиться к методу или свойству объекта приведет к ошибке
   // "Access violation". Помещаем в переменную нулевой указатель, чтобы можно было понять
   // существует объект или нет.
   SL := nil;
   if SL <> nil then …
   else …
end;
```

Объект создается с помощью конструктора класса. Как правило, конструктор имеет название **Create**. При вызове конструктора происходит выделение памяти для объекта и другие подготовительные действия, заданные разработчиком класса. В переменной сохраняется ссылку на объект – адрес участка памяти. До вызова конструктора переменная содержит неопределенный указатель, обозначается ключевым словом **nil**. После создания объекта можно обращаться к его методам и свойствам через точку. Закончив работу с объектом, нужно освободить занятую объектом память или, как говорят по-другому, уничтожить объект. Делается это методом **Free**. Стоит обратить внимание, что

при уничтожении объекта переменная по-прежнему содержит ссылку, но уже на несуществующий объект. Попытка обратиться к методу или свойству несуществующего объекта приведет к ошибке “Access violation”. Присваивайте переменной класса значение **nil**, чтобы можно было понять существует объект или нет. Объект можно сравнивать с **nil** или с другим объектом.

Основные принципы ООП

К основным принципам ООП относят: инкапсуляцию, наследование, полиморфизм.

Инкапсуляция – это скрытие реализации класса, предоставление только открытых методов и свойств для работы с ним. Разработчику необязательно вникать во внутренние механизмы класса, ему достаточно знать назначение его открытых методов и свойств. Разработчики классов могут разграничивать доступ к полям, методам и свойствам класса. Есть три уровня доступа: приватный, защищенный и публичный. К приватным данным класса имеет доступ только сам класс. К защищенным данным имеют доступ потомки класса. К публичным данным имеет доступ весь код.

Наследование позволяет создать новый класс на основе существующего класса. Новый класс называют потомком, а наследуемый класс – базовым классом, предком, родительским. Новый класс может расширять функционал базового класса и переопределять существующий функционал. Создаются целые иерархии классов, где базовые классы обладают некоторыми методами и свойствами, которые общие для всех потомков. Взять, к примеру, классы LCL. Элементы графического интерфейса представлены такими классами, как **TEdit** – поле для ввода текста, **TLabel** - метка, **TButton** – кнопка. Каждый из них имеет специфические свойства и методы. Но все они, по сути, являются элементами управления, которые имеют положение, размер, цвет и т. д. И чтобы не повторять для каждого класса одни и те же свойства, а также задать общее для всех поведение, они наследуют общие свойства и методы от базового класса **TControl**.

Полиморфизм – это когда базовый объект «знает», метод какого потомка надо вызвать.

```
procedure RepaintAllControls(Container: TWinControl);
var
   i: Integer;
   C: TControl;
begin
   for i := 0 to Container.ControlCount - 1 do
   begin
      C := Container.Controls[i];
      C.Repaint;                  // Перерисовываем компонент
   end;
end;
```

В данном примере выполняется перерисовка (обновление внешнего вида) компонентов некоторого контейнера. У базового класса всех визуальных компонентов **TControl** есть метод **Repaint**, который переопределяется в потомках, т. е. в каждом потомке реализуется свой алгоритм рисования. Но нам не обязательно вызывать метод **Repaint** конкретного потомка, достаточно вызвать метод базового класса, программа сама разберется метод какого потомка нужно вызвать.

Еще про объекты

Когда создается объект некоторого класса, ссылка на него может быть записана в переменную базового класса. Например:

```
var
```

```
   SL: TStrings;
begin
   SL := TStringList.Create;
   …
end;
```

Переменная базового класса не дает доступа к дополнительным свойствам и методам потомка. Для этого нужно выполнить преобразование типа. Делается это следующим образом:

```
TStringList(SL).Sort;
```

Если переменная содержит ссылку на несуществующий объект, то появится ошибка “cannot cast an object”. Если переменная содержит nil, то появиться ошибка “null pointer exception” или “could not call proc”. Если результат преобразования сохранить в другой переменной, то в случае с **nil** ошибки не будет, другая переменная тоже будет содержать **nil**:

```
var
   SL: TStrings;
   SL2: TStringList;
begin
   SL2 := TStringList(SL);       // Ошибки не будет. SL2 будет равно nil.
end;
```

Часто возникает необходимость определить потомком какого класса является объект. Для этого служит оператор **is**:

```
procedure FieldChange(Sender, Control: TObject; const FieldName: String);
begin
   if Control is TdxEdit then
   begin
      if TdxEdit(Control).ValidateText then …
   end
   else if Control is TCustomDBEditButton then …
end;
```

Для упрощения доступа к элементам класса используется оператор **with**:

```
var
   SL: TStringList;
begin
   SL := TStringList.Create;
   with SL do
   begin
      Add('Один');
      Add('Два');
   end;
   …
end;
```

Создавать объект можно прямо в операторе **with**:

```
with TStringList.Create do
begin
   Add('Один');
   Add('Два');
```

```
  …
  // Не забываем удалить объект
  Free;
end;
```

12. Директивы компилятора

Директивы влияют на процесс компиляции. Директива заключается в фигурные скобки и начинается со знака $. Движок поддерживает несколько директив: $I ($INCLUDE), $DEFINE, $UNDEF, $IFDEF, $IFNDEF, $ELSE, $ENDIF.

12.1. Директива $I: включение кода модуля

Наиболее часто используемая директива, которая включает в текущий модуль код другого модуля.

```
{$I MyUtils}
…
{$I Мой Модуль}
…
procedure Form_Create;
begin
end;
```

В результате включения модулей исходные коды объединяются, и получается один большой модуль. Из этого следует, что в модулях не должны дублироваться имена идентификаторов (переменных, констант, подпрограмм и т. д.).

12.2. Условная компиляция

Остальные директивы позволяют компилировать разные участки кода в зависимости от условия. Таким условием является проверка – определен ли символ условной компиляции или нет. Символ условной компиляции задается директивой $DEFINE:

```
{$DEFINE ABC}
{$DEFINE Проверка}
{$DEFINE 12A}
```

Символ условной компиляции может состоять почти из любых символов. Проверка осуществляется директивами $IFDEF, $IFNDEF:

```
{$IFDEF ABC}
// Код А
{$ELSE}
// Код Б
{$ENDIF}
…
{$IFNDEF 12A}
// Код В
{$NDIF}
```

Код А будет скомпилирован только в том случае, если определен символ ABC, в противном случае будет скомпилирован код Б. Код В будет скомпилирован, если символ 12A не определен.

Директива $UNDEF снимает определение символа, т. е. символ становится неопределенным.

В DataExpress есть два предопределенных символа – WINDOWS и LINUX. Таким образом, можно

компилировать специфические для каждой из систем участки кода.

```
{$IFDEF LINUX}
// Специфичный для Linux код
{$ELSE}
// Специфичный для Windows код
{$ENDIF}
```

13. Обработка исключительных ситуаций

В процессе выполнения скрипта возможно возникновение исключительных ситуаций, ошибок. Если это происходит, то пользователь видит сообщение об ошибке «Опа!...». Вы можете перехватить ошибку и обработать ее особым образом. Для обработки исключительных ситуаций блок кода, в котором они возможны, заключается в составной оператор **try**.

```
try
   FS := TFileStream.Create(MyFileName, fmCreate);
except
   MsgBox('Ошибка', ExceptionToString(ExceptionType, ExceptionParam);
end;
```

Если в блоке **try** происходит исключение, то выполнение кода прерывается и выполняется код в блоке **except**.

Есть другая конструкция – **try..finally:**

```
try
   FS := TFileStream.Create(MyFileName, fmCreate);
   try
      …
   finally
      FS.Free;
   end;
except
   MsgBox('Ошибка', ExceptionToString(ExceptionType, ExceptionParam);
end;
```

Код в блоке **finally** выполняется в любом случае, нормально ли отработал код или произошла ошибка. Если в блоке **try** используется ключевое слово **exit** (выход из процедуры/функции), то код в блоке **finally** также выполняется.

В операторе **try** могут быть использованы оба блока:

```
try
   …
except
   …
finally
   …
end;
```

Вы можете программно сгенерировать исключение, используя процедуру RaiseException:

```
var
   filename, emsg: string;
```

```
begin
   filename = '';
   try
      if filename = '' then
         RaiseException(erCustomError, 'Имя файла не может быть пустым');
      …
   except
      emsg := ExceptionToString(ExceptionType, ExceptionParam);
      …
   end;
end;
```

При обработке исключительных ситуаций могут быть использованы следующие процедуры и функции:

`function ExceptionToString(Err: TIFException; Param: String): String`

Формирует по типу исключения строку сообщения об ошибке.

`function ExceptionType: TIFException`

Возвращает тип возникшего исключения.

`function ExceptionParam: String`

Возвращает текст сообщения об ошибке.

`procedure RaiseException(Err: TIFException; Param: String)`

Генерирует исключительную ситуацию.

`procedure RaiseLastException`

Генерирует последнюю возникшую исключительную ситуацию.

14. Подключение внешних библиотек DLL

Пример подключения функции из библиотеки:

```
function MessageBox(hWnd: Integer; lpText, lpCaption: AnsiString; uType: Cardinal): Integer; external 'MessageBoxA@user32.dll stdcall delayload';
```

До слова **external** идет описание функции: имя и состав параметров. Имена могут быть любыми допустимыми идентификаторами. После слова **external** в кавычках указывается название функции в файле DLL. После символа **@** идет имя самого файла dll. Stdcall – это соглашение о вызове функции. Слово **delayload** необязательное, оно позволяет загружать DLL не сразу при запуске программы, а при первом вызове функции.

Файл DLL должен находиться в папке с программой или в системной папке «system32». Текст подключения может быть в любом модуле. Если функция подключается в пользовательском модуле или модуле **Main**, то вы можете использовать эту функцию и в других модулях.