Object Pascal

1. Основы языка Object Pascal

1.1. Алфавит языка

Основными символами языка Object Pascal являются:

* символы \_ + -
* 26 больших и 26 малых латинских букв A,B, …Y,Z, a,b, …, y,z
* 10 арабских цифр 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
* специальные символы \* / = ^ < > ( ) [ ] { } . , : ; ' # $ @

Буквы русского алфавита не входят в состав алфавита языка. Их использование допустимо только в строковых и символьных значениях.

Нет различий при использовании больших и малых букв в записи имен переменных, процедур, функций и меток. Их максимальная длина ограничена 126 символами.

1.2. Краткие сведения о структуре программы

Программа, написанная в среде Delphi при помощи языка Object Pascal, всегда состоит из нескольких модулей. Как минимум таких модулей должно быть два. Один модуль всегда является головной программой и имеет название program. Прочие модули играют вспомогательную и зависимую от головной программы или от других модулей роль и называются unit. Минимально структурированная программа имеет один модуль program и один модуль unit. Серьезные программы помимо модуля program могут содержать до нескольких десятков авторских модулей unit и большое количество ссылок на фирменные или разработанные как самим автором, так и другими разработчиками модули unit.

Программа всегда начинает работу с модуля program, активизируя функционирование одного или нескольких зависимых модулей unit. Те в свою очередь могут активизировать другие модули unit и т.д.

Исходный программный текст каждого модуля составляется на языке Object Pascal и помещается в отдельный файл, который всегда имеет расширение .pas. Текст модуля program имеет расширение .dpr.

Полный программный текст любого модуля также имеет свою структуру, которая может включать блоки определения констант, внутренних структур описания типов, тексты процедур, функций и др.

1.3. Лексическая структура языка

Строительным материалом для конструирования программного текста модуля являются лексемы – особые языковые конструкции, имеющие самостоятельный смысл. Лексемы строятся при помощи символов алфавита языка. В Object Pascal различают следующие основные классы лексем:

Зарезервированные (служебные) слова. Этот класс состоит из слов, построенных только с помощью букв алфавита. Служебные слова можно использовать только по прямому назначению, т. е. так, как их назначение определил разработчик языка. Ни в каком другом виде, например в качестве имен переменных, их использовать нельзя.

Ниже представлен список таких слов:

|  |  |  |
| --- | --- | --- |
| And  asm  class  destructor  do  end  file  for  if  inherited  interface  library  not  or  procedure  raise  resource  shl  then  try  until  while  with | array  begin  const  dispose  downto  except  finalization  function  implementation  initialization  in  interface  is  mod  object  out  program  record  string  shr  threadvar  type  uses | as  case  constructor  div  else  exports  finally  goto  in  line  label  nil  of  packed  property  repeat  set  string  to  unit  var  xor |

Кроме того, нельзя использовать следующие слова, не принадлежащие к этому классу: **private, protected, public, published, automated, directives, on, virtual**.

Идентификаторы (имена). Идентификаторы или имена предназна-чены для обозначения констант, переменных, типов, процедур, функций, меток. Они формируются из букв, цифр и символа "\_" (подчеркивание). Длина имени может быть произвольной, однако компилятор учитывает име-на по его первым 63 символам. Внутри имени не должно быть пробелов.

Object Pascal в именах не различает больших и малых букв. Так следующие имена будут идентичны:

SaveToFile, SAVETOFILE, savetofile, sAVEtOfILE.

Среди программистов установилось хорошее правило, в соответствии с которым имена формируются таким образом, чтобы одновременно выпол-нять роль комментария, поясняющего назначение имени. Так, в приведенном примере имя переводится с английского как "сохранить в файле". Кроме того, с учетом невозможности вставки внутрь такого имени пробелов, первые буквы слов обычно пишут заглавными, а прочие строчными. Из приведенного примера хорошо видно, что именно такой способ записи наиболее нагляден для визуального восприятия имени. Нередко в качестве заменителя пробела используют символ "\_". Однако это удлиняет и без того длинные имена. Преимущества длинных имен совсем не означают, что нельзя применять короткие имена. Понятно, что проще набрать с клавиатуры и использовать оператор

a := a + 1,

чем идентичный ему оператор

Disk\_C\_DirctoryCounter := Disk\_C\_DirctoryCounter +1.

Следует, однако, с большой осторожностью использовать короткие имена, т. к. это нередко приводит к путанице между глобальными и локальными переменными, обозначенными одинаковыми именами, и, как следствие, к ошибкам в работе программы. Наиболее удобным, безопасным и желательным можно считать локальное использование коротких имен, когда они описаны и использованы внутри какой-нибудь сравнительно небольшой по объему текста процедуры или функции и их действие ограничено пределами только этой алгоритмической единицы. При подозрении на путаницу, действие такой переменной легко проконтролировать визуально.

Изображения. К их числу относятся константы, символьные строки и некоторые другие значения.

Знаки операций формируются из одного или нескольких символов по определению действий, связанных с преобразованием данных.

Разделители используются с целью большего структурирования модуля, с тем чтобы повысить визуальное восприятие длинных текстов. К их числу можно отнести ; := ( .

Комментарии. Эти лексемы используют для пояснения отдельных фрагментов текста программы. Они представляют собой последовательность символов, заключенную в фигурные скобки { } или в разделители (\* и \*), а также последовательность символов, расположенных в строке справа от двух следующих друг за другом символов /.

Примеры комментариев:

{ Функция вычисления количества дней между двумя датами }

(\* Функция вычисления количества дней между двумя датами \*)

// Неправильный ответ

Пробел. Этот символ не имеет видимого изображения и служит для отделения лексем друг от друга в тех случаях, когда это необходимо. Обычно использование одного или нескольких рядом стоящих пробелов не искажает смысл программы.

1.4. Некоторые важные понятия

Остановимся на этих понятиях для того, чтобы коротко определить их для понимания большинства примеров, которыми сопровождается материал. Эти компоненты языка имеют исключительную важность, в последующих разделах они будут описаны более подробно.

Ячейка. Этот несколько устаревший, но весьма удобный термин обозначает фрагмент памяти, который можно представить как некий контейнер для хранения данных определенной структуры. Ячейка всегда имеет свое уникальное имя, которое служит адресом, по которому расположены находящиеся в ней данные. Примером ячейки могут служить любые разрешенные имена, например a1, Imk12, Count и т. д. Термин "ячейка" не является языковым термином Object Pascal и используется здесь только для большей наглядности при описании основ языка.

Значение – это постоянная величина или структурный комплекс постоянных величин, выраженных в явном виде. Значение не имеет имени.

Примеры значений:

-55.455051 { обыкновенное вещественное число},

'Расчет посадки с натягом' {строка символов}.

Константа – это ячейка, в которой всегда хранится одно значение. Константы не могут быть изменены в ходе выполнения программы. В этом смысле константа отвечает общепринятому определению постоянной (неизменяемой) величины. Всякая константа должна быть описана, т. е. должно быть явно указано ее значение. Значение константы неявно определяет ее тип.

Необходимо отметить, что в языке существуют так называемые типизованные константы, которые в ходе прохождения программы могут быть изменены. Тип константы указывается в специальной языковой конструкции, начинающейся словом Type (тип).

Переменная – это ячейка, в которой в каждый момент времени хранится одно значение или не хранится ничего. Переменная в любой момент времени может быть изменена программой. Всякая переменная должна быть описана. т .е. должен быть явно указан ее тип. Тип переменной указывается в специальной языковой конструкции, начинающейся словом Var (от английского variable – постоянная).

Тип – это структура и описание множества значений, которые могут быть присвоены переменной.

Оператор присваивания – это команда, предназначенная для изменения содержимого ячейки. С его помощью происходит изменение значения переменной (или типизованной константы).

Синтаксис оператора присваивания:

x := y; { читается "x присвоить y" }

Здесь x – переменная, y – выражение. Выражением могут быть, в частности, переменная, константа или значение. Последовательность символов ":=" обозначает операцию присваивания, в соответствии с которой сначала вычисляется выражение y, затем получившийся результат в виде значения записывается в переменную x (см. подробнее гл. 9).

Примеры:

d := 5; { значение 5 записывается в переменную D },

h := d + 12.5; { выч. 5+12.5, рез. 17.5 записывается в переменную h }.

2. Система типов

В языке Object Pascal все переменные, т. е. ячейки памяти, предназначенные для записи, чтения и хранения значений, должны быть предварительно описаны. Это означает, что всякая переменная должна быть явно отнесена к какому-либо типу.

Тип – это одновременно структура и описание множества значений, которые могут быть присвоены такой переменной.

Язык Object Pascal имеет множество разнообразных типов. Более того он позволяет самому пользователю конструировать самые разнообразные типы, которые могут быть ему необходимы. Конструирование таких типов производится из сравнительно ограниченного количества стандартных типов.

Типы имеют свою иерархию. На верхнем этаже иерархии расположены следующие типы: простые, составные, ссылочные и процедурные.

3. Стандартные простые типы

Основными типами языка являются стандартные простые типы и стандартные структурные типы.

Простые типы делятся на скалярные и ограниченные типы. Cкалярные типы делятся на стандартные и перечислимые. Стандартные скалярные типы делятся на пять видов:

* целые [Integer],
* вещественные [Real],
* логический (булевский) [Boolean],
* символьные [Char],
* строковые [String].

К ним примыкает особый вариантный тип [Variant].

3.1. Целые типы

Эта группа типов охватывает множество целочисленных значений. Они отличаются друг от друга диапазоном допустимых значений и количеством занимаемой памяти.

Целыми типами являются ShortInt, SmallInt, LongInt, Int64, Byte, Word и LongWord, характеристики которых приведены в табл. 1.

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тип | Диапазон значений | Размер памяти |
| 1.  2.  3.  4.  5.  6.  7. | ShortInt  SmallInt  LongInt  Int64  Byte  Word  LongWord | –128 .. 127  –32768 .. 32767  –2147483648 .. 2147483647  –2^63 .. 2^63–1  0...255  0...65535  0 .. 4294967295 | 1 байт  2 байта  4 байта  8 байтов  1 байт  2 байта  4 байта |

При назначении типа переменной следует исходить из оценки диапазона возможных значений, которые она может принимать в ходе выполнения программы.

Так если значения переменной будут только положительными, то можно ее отнести к одному из типов Byte, Word, LongWord. Если известно также, что ее значения никогда не выйдут за 255 (например, если переменная предназначена для хранения номера месяца текущего года), то лучше использовать тип Byte. При этом память будет расходоваться наиболее экономно.

Не следует, однако, стремиться к излишней экономии памяти на переменных. Нередко экономно описанная переменная может привести к ситуации, когда программа попытается записать в нее такую константу, которая превышает допустимый диапазон значений. Это приведет к немедленному аварийному завершению программы с сообщением "Range check error" (выход за допустимые границы диапазона). Сообщения такого рода могут генерироваться самыми разными операциями и в самых разных местах программы. По этой причине поиск ошибки в программе, особенно если она многомодульна и сложна, может надолго затянуться.

Не следует также злоупотреблять многообъемными типами, т.к. это может привести к излишнему перерасходу, а иногда и нехватке памяти, с одной стороны, и замедлению работы программы – с другой.

Примеры:

Var

A, A\_Par: Integer;

T1, T2, T3: LongInt;

CircleCounter: byte;

Значения целых типов изображаются в обычном десятичном или в шестнадцатеричном видах. Они отличаются тем, что при изображении шестнадцатеричных значений в его начале ставится символ $ и сами значения формируются из шестнадцатеричных цифр 0 .. 9, A ... F.

Максимально допустимый диапазон значений определяется их типом.

Примеры:

0 9877 -56 $F1 ( то же, что 241)

Над целыми значениями можно выполнять четыре обыкновенных арифметических действия: сложение (+), вычитание (-), умножение (\*), деление (/) и два дополнительных действия: деление нацело (div) и взятие остатка от деления (mod). При выполнении деления результатом будет вещественное значение, во всех остальных операциях – целое.

3.2. Вещественные типы

Эта группа типов охватывает вещественные значения.

Вещественные типы не могут быть использованы:

* в качестве индексов массивов;
* в операторах For и Case;
* в качестве базисного типа при определении множеств;
* при определении подтипов.

При описании вместо Real48 можно указывать Real.

Ниже в табл. 2 приведен список типов и их характеристики.

Таблица 2

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № | Тип | Диапазон значений | Значащих цифр в мантиссе | Размер памяти |
| 1.  2.  3.  4.  5.  6. | Real48  Single  Double  Extended  Comp  Currency | 2.9 x 10^–39 ... 1.7 x 10^38  1.5 x 10^–45 ... 3.4 x 10^38  5.0 x 10^–324 ... 1.7 x 10^30  3.6 x 10^–4951 ... 1.1 x 10^4932  -2^63+1 ... 2^63 -1  -922337203685477.5808 ... 922337203685477.5807 | 11 – 12  7 – 8  15 – 16  19 – 20  19 – 20  19 – 20 | 6 байтов  4 байта  8 байтов  10 байтов  8 байтов  8 байтов |

Примеры:

Var

rA, rA\_Par: Real;

T: Integer;

Вещественные значения можно изобразить:

* в форме с фиксированной десятичной точкой;
* в форме с плавающей десятичной точкой.

Первая форма представления вещественного значения представляет привычное число, в котором целая и дробная части разделены десятичной точкой, например

12.455

-988.45

-8.0

Вторая форма предназначена для записи очень больших или очень маленьких по абсолютной величине значений, когда их представление в форме с фиксированной точкой затруднительно или невозможно. Такое значение изображают в виде

<значение с фиксированной точкой > E <порядок>

Примеры:

-45.2E6 ( то же, что -45,2 106)

5.245E-12 ( то же, что 5,24 10-12)

Порядок таких чисел должен быть всегда целым числом.

3.3. Логический (булевский) тип

Логические переменные имеют тип boolean. Такая переменная занимает один байт памяти и может иметь одно из двух возможных значений – True (истина) или False (ложь).

Примеры:

Var

b : boolean;

b1, Ti : boolean;

3.4. Символьный тип

Типы AnsiChar и WideChar описывают множество отдельных символов языка, включая буквы русского алфавита. AnsiChar описывает множество из 256 ASCII-кодов и занимает один байт памяти, WideChar описывает мно-жество Unicode – универсальное множество кодов и занимает два байта памя-ти. Тип AnsiChar эквивалентен базовому типу Char прежних версий языка.

Примеры:

Var

Ch, k : AnsiChar;

Char\_Massivr: array[1..100] of Char;

Символьное значение представляют в виде символа, заключенного с обеих сторон в апострофы. Для изображения самого апострофа его удваивают (последний пример), например:

'h' 'X' '#' '$' ''''

3.5. Строковые типы

Этот тип во многом схож с типом Array of Char, т. е. массивом символов. Отличие состоит в том, что переменная этого типа может иметь динамическое количество символов (от нуля до верхней границы), в то время как массив символов всегда статичен и имеет одинаковое количество символов.

Таблица 3

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тип | Длина строки | Занимаемая память |
| 1.  2.  3. | ShortString  AnsiString  WideString | 0 – 256 символов  0 – 2 Гб символов  0 – 2 Гб символов | (Кол-во символов) х 1 байт  (Кол-во символов) х 1 байт  (Кол-во символов) х 2 байта |

Максимальная длина строковой переменной должна быть указана явно. Размер строки на единицу больше ее объявленной длины, т. к. в ее нулевом байте содержится фактическая длина строки. Длину в нулевом байте можно принудительно менять.

Особо следует выделить тип String. Если длина String-строки не объявлена, то при действии директивы компилятора {**$H+**} или без ее указания такое объявление равносильно AnsiStrig. Если установлена директива {**$H-**}, то тип String равносилен типу ShortString.

Строковое значение изображают в виде последовательности символов, заключенной в апострофы. Пустую строку изображают двойным апострофом.

Примеры значений строковых типов:

'Иванов И.И.' '' 'Газета"ИЗВЕСТИЯ"' 'Строка символов'

Примеры описания переменных строковых типов:

**Var**

s1, s2 : ShortString [12];

st1, st2 : AnsiString [580];

ChMassiv: array [1..15] of String;

3.6. Строковый тип PChar

Для связи с функциями Windows в язык Object Pascal введен новый тип строк – PChar-строки с завершающим нулем. В обычной и привычной для прежних версий языка String-строке нулевой байт отведен для хранения реального количества символов этой строки, а сами символы последовательно располагаются начиная с первого байта. В PChar-строке, наоборот, символы располагаются начиная с нулевого байта, а их последовательность заканчивается завершающим нулем.

Строки PChar можно объявлять как обычные символьные массивы. Например, строку длины 3000 плюс один байт, зарезервированный под завершающий нуль, можно определить следующим образом:

**Var**

s: array[1 .. 3000] of Char;

П р и м е ч а н и е. Без необходимости не используйте PChar-строки. Строковые String-типы и функции для обработки таких строк хорошо отлажены, они легче в использовании, и, как правило, надежнее PChar-строк.

3.7. Динамические PString-строки

Этот тип строк так же, как PChar, введен в язык для обращения к функциям Windows. Подробнее PString-строки описаны далее.

3.8. Перечислимые типы

Этот тип переменных может быть сформирован самим пользователем. Он создается простым перечислением возможных значений переменной.

Примеры перечислимых типов:

**Type**

MaleNames = (Ivan, Peter, Serge);

SwithOpts = (On, Off);

SostTypes = (Active, Passive, Waiting);

Sides = (Left, Right, Top, Down);

В первом примере переменная объявленного типа может принимать значение одного из трех мужских имен. Во втором – одно из двух значений – On (включено) или Off (выключено) и т. д.

Имена из списка перечислимого типа считаются константами соответствующего перечислимого типа и в пределах блока не должны повторяться.

Например, описания вида

Type

Days1 = (Monday, Wednesday, Friday);

Days2 = (Tuesday, Wednesday, Saturday, Sunday);

содержат ошибку, т. к. константа Wednesday используется дважды.

3.9. Ограниченные типы

Этот тип формируется самим пользователем посредством сужения значений ранее определенного или стандартного типов.

Примеры:

Type

Diapason = 1 .. 30;

Letters = 'a' .. 'v';

TList = (t1, t2, t3, t4, t5, t6,t7, t8, t9, t10);

TlistSmall = (t2 .. t8);

3.10. Вариантный тип (Variant)

Тип Variant – особый тип языка Object Pascal. Значение этого типа наперед неизвестно, однако может быть определено через присваиваемое значение одним из следующих типов: все целые, вещественные, строковые, символьные и логические типы, за исключением Int64.

Следующие примеры демонстрируют использование типа Variant и механизм конверсии типов при смешивании его с другими типами. Сопроводи-тельные комментарии поясняют правила, при помощи которых операторы присваивания меняют тип Variant-переменных в зависимости от принятого ими значения.

Var

V1, V2, V3, V4, V5: Variant; {описание Variant-переменных }

I: Integer;

D: Double;

S: string;

...

**begin**

V1 := 1; { integer-значение }

V2 := 1234.5678; { real-значение }

V3 := 'Иванов'; { string-значение }

V4 := '1000'; { string-значение }

V5 := V1 + V2 + V4; { real-значение 2235.5678}

I := V1; { I = 1 (integer-значение) }

D := V2; { D = 1234.5678 (real-значение) }

S := V3; { S = ' Иванов' (string-значение) }

I := V4; { I = 1000 (integer-значение) }

S := V5; { S = '2235.5678' (string-значение) }

**end;**

3.11. Тип "дата – время"

В языке имеется несколько типов, предназначенных для работы с датами и временем. Они имеют вид

**Type**

TDateTime = Double;

TDate = TDateTime;

TTimeStamp = **Record**

Time: Integer; { время в миллисекундах от полуночи }

Date: Integer; { единица + число дней с 01.01.0001 г.}

**end;**

Тип TDateTime предназначен для хранения даты и времени.

Переменная отличается от константы или значения тем, что в процессе работы программы она может менять содержимое своей памяти. Однако в каждый момент времени она хранит только одно значение. Всякая перемен-ная имеет имя, тип и свою область видимости. По сути, переменная явля-ется контейнером для хранения значения идентичного типа. Всякая перемен-ная в блоке описания должна быть представлена только один раз.

Описание переменной или группы переменных начинается словом Var. Область видимости переменной будет подробно описана ниже.

Общий вид описания переменных одного типа:

<переменные> : <тип>;

Пример:

Var

t\_s1, t\_q1: String[255];

rt1, rt2: (Opened, Closed, Unknown);

Re1, Re2, Re3: Real;

i: Integer;

В этом примере переменные t\_s1 и t\_q1 описаны как строковые переменные типа String[255]. При работе программа выделит под каждую из них с учетом нулевого байта по 256 байтов памяти для хранения символьных значений. Переменные rt1, rt2 объявлены как переменные, которые могут принимать в определенный момент времени одно из перечисленных значений: Opened, Closed, Unknown. Переменные Re1, Re2, Re3 объявлены вещественными, а переменная i – целочисленной типа Integer.

Переменными могут быть объявлены не только переменные простых типов. Ниже будут рассмотрены переменные более сложных – структурных – типов. Более того, переменными могут быть объявлены структуры структур, примером которых являются классы. Например:

type

TKdnClass = class(TObject)

…

End;

Var

Ts: Record

A, N: Integer;

End;

Cl: TKdnClass;

5. Описание констант

В Object Pascal различается два вида констант – обыкновенные и типизованные. Описание констант следует после слова Const.

5.1. Обыкновенные константы

Описание константы строится по правилу

<имя константы> = <выражение>;

Примеры:

Const

T\_Par = 12899;

M\_ArrayCount = 16;

Middle\_M\_Array = M\_ArrayCount div 2;

RealMax = 1.7e38;

StarString = '\* \* \* \* \* \* \* \* \* \* \* \* \*';

Log10 = 2.302585;

Log10\_Invert = 1/Log10;

LeftArrayBound = -M\_ArrayCount;

Тип константы определяется автоматически по виду ее значения.

Существует несколько констант, которые заранее предопределены и не требуют описания:

Pi = 3.1415926536E+00 (тип Real)

False, True ( Boolean)

MaxInt = 32767 ( Integer)

MaxLongInt = 2147483647 ( LongInt)

Nil ( Pointer).

Часто константы используют для определения динамических массивов, динамических строк и других динамических структур. Следующие описания демонстрируют пример такого определения.

**Const**

ArrMax = 100;

nSt = 46;

**Var**

IntArr: Array[1..ArrMax] of Integer;

StrArrr: Array[1..ArrMax] of String[nSt];

Такой способ описания позволяет при необходимости легко изменить размер массивов, изменив всего лишь значение константы (в данном случае константы ArrMax). При этом произойдут автоматические изменения определений как в текущем, так и в других модулях, если они содержат определения, опирающиеся на эту константу. Явное определение массивов через значение 100 потребовало бы соответствующих явных изменений этого значения на другое всюду, где такие описания имели бы место. Для больших программ это могло бы стать причиной ошибок, если часть изменений не была бы выполнена по недосмотру.

Типизованные константы

Это специальный тип констант, которые отличаются от обыкновенных констант тем, что при их описании необходимо указывать тип.

Простые типизованные константы. Общий вид константы:

<имя константы> : <тип> = <значение>;

Примеры:

Const

CurrentPosition: Word = 11000;

LastLetter: Char = 'z';

HeadOfModule: String[26] = 'Начало программного модуля';

Типизованные константы нельзя использовать для описания динамических структур. Следующий пример демонстрирует недопустимое описание динамического массива Arr через типизованную константу ArrMax:

Const

ArrMax: Integer = 100;

Var

IntArr: Array [1..ArrMax] of Integer; { Ошибка }

Значение типизованных констант можно изменять в ходе выполнения программы, и они могут быть использованы в качестве Var-параметра процедуры или функции. В этой связи типизованные константы по сути являются переменными с начальным значением.

Типизованные константы типа "массив". Этот тип констант позволяет обозначить постоянной величиной целый массив однотипных значений, например:

Type

tVotes = (Yes, No, UnDef);

tVoteArr = array [tVotes] of String [7];

Const

Votes : tVoteArr = ( 'Да', 'Нет', 'Не знаю' );

Var

V: tVoteArr;

S: String[20];

…

V:= Votes;

S:=V[UnDef];

…

Здесь в секции Type сначала описан перечислимый тип tVote из трех значений Yes, No, Undef. Затем через этот тип определен новый тип tVoteArr как тип-массив из трех элементов, каждый из которых является строкой длины 7. Далее в секции Const определена типизованная константа Votes, которая определена как массив трех строковых значений ('Да', 'Нет', 'Не знаю'). Затем в секции Var описаны переменные V и S разных типов. Предпоследняя и последняя строки являются исполняемыми операторами присваивания. Сначала переменной V присвоено начальное значение – константа Votes. Затем переменной S присвоено значение третьего элемента массива V. В результате значением строковой переменной S будет 'Не знаю'.

Типизованные константы типа "запись". Это комбинированный тип констант, основанный на конструкциях типа Record (см. параграф 7.1), которые состоят из полей. Константы такого типа определяются по правилу "имя поля : значение поля".

Пример:

Type

tDayOfYear = Record {день года}

Week : (Mon,Tue,Wed,Thu,Fri,Sat,Sun); {день недели}

Num : 1 .. 31; {день месяца}

Month : 1 ..12; {месяц }

Year : 1951 .. 2050; {год}

End;

Const

D1 : (Week : Sun; Num : 26; Month : 10; Year : 2001 D1, которая представляет конкретную дату "Воскресенье, 26, октябрь);

Здесь в секции Type описана запись, состоящая из четырех полей и характеризующая день года. Назначения этих полей понятно из сопроводительных комментариев. Затем в секции Const описана типизованная константа (Sun; 26; 12; 2001), т. е. " Воскресенье, 26 октября 2001 года".

Типизованные константы типа "множество". Эти константы могут быть построены как подмножества базовых или производных от них типов.

Примеры:

Type

tChildNameSet = set of ['Таня', 'Валя', 'Володя', 'Гена'];

Const

Girls2Set : tGirlsNamesSet = ['Валя', 'Гена'];

Indexes: set of integer = [300 .. 500, 1777,3700];

Здесь в секции Type описано множество из четырех имен детей. Ниже в секции Const описано две константы, каждая из которых представляет подмножества. Первая – подмножество имен, объявленных ранее в tChildNameSet, вторая – подмножество целых чисел типа Integer.

6. Описание типов

Ранее уже приводились примеры описания переменных, в которых их тип указывался в Var-секции явно или на основе ранее объявленного пользовательского типа.

Описание секции типов начинается словом Type.

В табл. 4 дан пример двух идентичных способов описания переменных t, u, n.

Таблица 4

|  |  |
| --- | --- |
| Явный способ описания  переменных | Описание переменных с предварительным описанием их типа |
| Var  t,u,n:(Mon, Tue,Wed, Thu,Fri,Sat,Sun); | Type  DaysOfWeek = (Mon, Tue,Wed, Thu,Fri,Sat,Sun);  Var t,u,n: DaysOfWeek; |

В тех случаях, когда явный и типизованный способы описания переменных конкурируют, следует всегда отдавать предпочтение способу описания переменных с предварительным объявлением их типа в секции Type. Такой способ позволяет:

а) конкретизировать тип;

б) четко выделить множество переменных этого типа;

в) повысить уровень структурированности программы;

г) снизить вероятность путаницы в типах, когда переменные фактически того же типа объявлены разными способами;

д) уменьшить объем текста за счет возможности быстрой ссылки на ранее определенный тип, особенно в тех ситуациях, когда этот тип используется для порождения новых типов, переменных, функций и пр. в других секциях или модулях.

В этой связи важно подчеркнуть, что даже при совпадении базовых типов различие в пользовательских типах может привести к непредсказуемому поведению программы. Например, в нижеследующей секции Type два производных типа t1 и t2 имеют одинаковый базовый тип byte. Однако объявленные ниже в Var-секции переменные p1 и p2 будут расценены системой как переменные разных типов. Это обстоятельство может послужить причиной недоразумений в ходе составления и/или выполнения программы.

Type

t1 = byte;

t2 = byte;

Var

p1: t1;

p2: t2;

Корректным можно считать следующий аналог:

Type

t1 = byte;

Var

p1,p2: t1;

7. Структурные типы

Структурные типы представляют собой совокупность значений одного или нескольких различных типов. Их наличие позволяет программисту конструировать производные типы практически любой сложности, что резко расширяет возможности языка.

К числу структурных относятся следующие типы:

множественные типы [Set],

регулярные типы (массивы) [Array],

комбинированные типы (записи) [Record],

файловые типы [File],

классы [Class],

классовые ссылки [Class reference],

интерфейсы [Interface].

Ниже будут подробно описаны первых четыре структурных типа – регулярный, комбинированный, множественный и файловый.

Три последних типа будут описаны отдельно в разделах, посвященных объектно-ориентированному программированию.

7.1. Регулярные типы (массивы)

Массив – это структура языка Object Pascal, представляющая собой упорядоченную совокупность элементов одного типа.

Следует различать два вида массивов: массив-тип и массив-переменную.

Массив-тип. Синтаксис маcсива-типа:

<имя массива> = Array [<тип индекса>, <тип индекса>, …, <тип индекса>]

Of <тип элемента>;

Всякий массив имеет размерность. Размерность определяется количеством типов индексов, которые заключены в квадратные скобки [ .. ].

Массив-тип предназначен для описания:

структуры массива как типа;

размерности массива;

типов индексов массива;

типа каждого элемента массива.

Так, в следующем примере

Type

tA1: array [1 .. 10] of Real;

описана структура одномерного массива вещественных элементов (Real), в котором индекс может изменяться в диапазоне целых значений от 1 до 10. Его элементами являются вещественные типы tA1[1], tA1[2], tA1[3], …, tA1[9], tA1[10].

Другой пример:

Type

Color: (Red, Green); { перечислимый тип }

Z: array [1 .. 3, Color ] of Boolean; { массив }

В нем сначала описан простой перечислимый тип Color. Ниже на его основе описан двумерный массив Z логических (Boolean) элементов. Первый индекс массива имеет целый тип, а второй – тип Color. Таким образом, массив состоит из шести элементов – логических типов:

Z [1, Red], Z [1, Green], Z[2, Red], Z[2, Green], Z[3, Red], Z[3, Green].

Массив-переменная. Синтаксис маcсива-переменной:

<имя массива > : ***Array*** [<тип индекса>,<тип индекса>, …, <тип индекса>]

***Of*** <тип элемента>;

Массив-переменная отличается от массива-типа тем, что все его элементы – это отдельные независимые переменные, которые могут содержать различные значения одного типа.

Массив-переменная может быть описан явно или с помощью ранее определенного в секции Type типа.

В следующем примере массивы y, Z описаны идентично, причем y – явно, Z – на основе ранее определенного типа в секции Type, т. е. неявно.

Type

tA1: array [1 .. 10] of Real;

Var

y : array [1 .. 10] of Real; {массив}

Z : tA1; {массив}

Этот пример демонстрирует разные способы описания одинаковых по структуре, но разных по типу массивов. С точки зрения корректного программирования он одновременно является примером того, как не следует описывать идентичные переменные. Причина этого заключается в том, что идентичные структуры во избежание непредсказуемого поведения программы следует описывать одним типом.

В этой связи корректным будет любой из вариантов, приведенных в табл. 5.

Таблица 5

|  |  |
| --- | --- |
| Корректный неявный способ | Корректный явный способ |
| Type tA1: array [1 .. 10] of Real;  Var Y, Z: tA1; | Var y, Z: array [1 .. 10] of Real; |

Многомерные массивы содержат два и более индексов, например:

Var h: array[1 ..3, boolean, -7 .. 7] of Word;

что эквивалентно

Var h: array[1 ..3] of array[boolean] of array[-7 .. 7] of Word;

Для упакованных массивов

Var packed array[Boolean,1..10,TShoeSize] of Char;

что эквивалентно

Var

**packed array**[Boolean] **of packed array**[1..10]

**of packed array**[TShoeSize] **of** Char;

Манипуляции с отдельными элементами массивов. Обращение к отдельному элементу массива возможно через его индексы. В следующем примере в секции Var описаны простая переменная i и два одномерных массива A и V как целые переменные типа Integer. Затем в блоке begin … end расположены три вычислительных оператора.

Var

i: Integer;

A, V: array[1..100] of Integer;

…

begin

i:= 5;

V[8]:= i+9;

A[45]:= V[i+3]\*2;

end;

При выполнении первого из них переменная i примет значение 5. При выполнении второго – восьмой элемент массива V примет значение 14. В третьем операторе сначала будет вычислен индекс i + 3 = 8, затем значение восьмого элемента массива V (значение 14) будет умножено на 2 и полученный результат – значение 28 – будет присвоено 45-му элементу массива A.

Манипуляции с массивами. Язык допускает с помощью одного оператора присваивания выполнить операцию над массивом в целом. Пусть, например, массивы A, V объявлены как квадратные матрицы. Тогда оператор

V:= A;

выполнит копирование значений всех элементов массива A в массив V, а после выполнения оператора

V:= A \* V;

будет выполнено умножение матрицы А на матрицу V и результат будет помещен в матрицу V с предварительным затиранием старых значений.

Упакованные массивы. Элементы упакованного массива хранятся в памяти максимально плотно. При записи он предварительно упаковывается с целью экономии памяти. При чтении, наоборот, распаковывается. Операции упаковки и распаковки требуют дополнительного времени. Поэтому использование упакованных массивов несколько замедляет работу программы. От обычного массива Array описание упакованного массива отличается тем, что перед этим словом добавляется слово Pаcked, например:

Var W: packed array [1..100] of Integer;

7.2. Комбинированные типы (записи)

Запись – это объединение элементов разных типов. Как и в массивах, следует различать запись-тип и запись-переменную. Один элемент записи называется полем.

Запись-тип. Синтаксис записи-типа:

<имя записи> = ***Record***

<имя поля 1> : <тип>;

<имя поля 2> : <тип>;

...

<имя поля N> : <тип>;

<вариантная часть >

***End***;

Записи очень удобны для описания и хранения разнотипных данных о каких-либо однотипных структурах.

Примером могут служить сведения о студентах. Сведения о любом из них могут включать поля: Фамилия, Имя, Отчество, Год рождения, Группа, Год поступления в вуз, Курс. Такие структуры являются однотипными и могут быть описаны следующим типом:

Type

TStud = Record { Сведения о студенте как запись }

Fio : String[40]; { ФИО как строка из 40 символов }

Name : String[20]; { Имя как строка из 20 символов }

Otch : String[30]; { Отчество как строка из 30 символов }

BirthYear : Word; { Год рождения как целое типа Word }

Group : String[8]; { Группа как строка из 8 символов }

ElectYear : Word; { Год поступления как целое типа Word }

Curs : Byte; { Курс как целое типа Byte }

***End;***

В этом примере типы полей записи типа tStud назначены с учетом максимально возможных значений этих полей. Так, структура может хранить фамилию из не более чем 40 символов. Полю Curs назначен тип Byte, который имеет диапазон значений 0 .. 255, т. к. значением этого поля может быть одно из значений 1 .. 6, которые полностью охватываются диапазоном типа Byte. Этому полю можно было бы назначить любой другой целый тип, например Word. Однако, в целях экономии памяти, повышения скорости чтения и записи данных, следует назначить именно тип Byte, который занимает всего 1 байт памяти, а не тип Word, который требует 2 байта памяти. В то же время, например, для поля ElectYear (год поступления) тип Byte непригоден, т. к. имеет недостаточный диапазон значений.

Записи с вариантами. Синтаксис записи допускает вариантность описания полей. Вариантная часть содержит несколько альтернатив, в каждой из которых в круглых скобках задается список полей, присущих своему варианту. Примером могут служить записи о пенсионерах:

Type

tPensioner = ***Record*** { пенсионер }

FioIO : String[100]; { Фамилия, имя, отчество одной строкой }

Age : Byte; { Возраст }

**Case** Citizen: boolean **of** {Горожанин ли ?}

TRUE : (Town : String[30];) {Город, в котором проживает}

FALSE: (Address : String[100]; {Полный адрес одной строкой}

Transport : String[200];) {Транспорт, которым можно добраться до города}

End;

В этом примере запись tPensioner содержит понятные поля FioIO и Age, а также поле нового вида – логическое поле Citizen вариантного типа. От значения этого поля зависит появление и непоявление некоторых потенциальных полей записи. Так если значение этого поля TRUE (истина), то в записи появляется (становится доступным) поле Town (город), при значении FALSE (ложь) – поля Address и Transport.

При использовании вариантных полей в записях следует подчиняться следующим правилам синтаксиса:

Вариантная часть должна начинаться со строки, в начале которой располагается слово Case, а в ее конце – слово Of. Между ними располагается поле-признак.

Запись должна содержать только один вариант, который должен располагаться в конце всех описанных полей непосредствено перед словом End.

Имена полей во всех вариантах должны быть разными. Они должны также отличаться от имен полей фиксированной части.

Для некоторых возможных значений поля-признака вариант может отсутствовать. В этом случае после двоеточия, соответствующего значению варианта, следует поставить пустой список ( ) либо не указывать этот вариант вообще (включая значение, двоеточие и пустое поле).

Запись-переменная. Синтаксис записи-переменной:

<имя записи> : ***Record***

<имя поля 1> : <тип>;

<имя поля 2> : <тип>;

...

<имя поля N> : <тип>;

<вариантная часть >

***End;***

т.е. синтаксисы переменной и типа отличаются одним символом (":" и "=").

Пример:

Type

tMass : Array [1 .. 2, 1 .. 50] of Real;

tRec: **Record**

Name : String [10];

Mass2: tMass;

**End;**

Var

J: Integer;

S: String[70];

F,Gri : Record

a,b,c: Integer;

k: Array [1..10] of String [60];

z: tMass;

r: tRec;

**End;**

В секции Var описаны две простые переменные J и S и две записи F и Gri, имеющих одинаковую, но достаточно сложную структуру:

первых три поля записей F и Gri имеют имена a,b,c и тип Integer;

поле k представляет собой одномерный строковый массив из 10 элементов;

поле z имеет тип, описанный в секции Type как двумерный вещественный массив, в котором первый индекс может изменяться в диапазоне 1 .. 2, а второй индекс – в диапазоне 1 .. 50;

поле r в свою очередь само является записью, поля которой описаны типом tRec в секции Type.

Доступ к полям записей. Переменная, представляющая поле, конструируется из имени записи и поля, отделенного друг от друга десятичной точкой. Такая составная переменная называется квалификационной.

Примеры квалификационных полей вышеприведенных записей:

F.a Gri.a F.k[6] Gri.z [2, 34] F.r.Name F.r.Mass2[1, 50]

Примеры операторов присваивания с участием полей записей:

S := 'Иванов Иван Петрович';

J := 123;

F.a := J + 9;

Gri.a := ( F.a + J ) \* ( F.c + F.b - Gri.c);

Gri.a := ( F.a + J ) \* ( F.c + F.b - Gri.c);

F.k [1] := F.z [2,30];

Gri.r.Name := 'Студент ' + F.k [8];

Gri.a := 12 \* (Gri.a + Gri.b + Gri.c);

Доступ к полям записей с помощью оператора With. Для упрощения обращения к полям одной и той же записи можно использовать оператор With.

Пример:

With Gri do

Begin

a:= 12 \* (a + b + c + F.a);

b:= 64 \* ( b - c);

End;

Эти операторы выполняют те же операции, что и операторы

Gri.a:= 12 \* (Gri.a + Gri.b + Gri.c + F.a);

Gri.b:= 64 \* (Gri.b - Gri.c);

7.3. Множественные типы

Множество – это совокупность однотипных элементов. Во многом оно похоже на типизованную константу, однако имеет от него принципиальное отличие. Это отличие состоит в том, что значениями множества являются все его допустимые подмножества.

Как и в массивах, следует различать множество-тип и множество-переменную.

Множество-тип. Синтаксис множества-типа:

<имя множества> = Set of <базовый тип >;

Пример:

Type

TSomeInts = 1..250;

TIntSet = set of TSomeInts;

создает тип множества с именем TIintSet, которое содержит множество целых чисел в диапазоне от 1 до 250. Это же множество могло быть описано явно:

type TIntSet = set of 1..250;

Множество-переменная. Синтаксис множества-переменной:

<имя множества> : Set of <базовый тип >;

В соответствии с вышеописанными типами можно объявить множества:

Var Set1, Set2: TIntSet;

а затем в операторной части задать эти множества:

...

Set1 := [1, 3, 5, 7, 9];

Set2 := [2, 4, 6, 8, 10];

Можно объявить множество явно, перечислив его элементы:

Var

MySet1 : set of 'a' .. 'z';

MySet2 : set of Byte

MySet3 : set of (Club, Diamond, Heart, Spade)

MySet4 : set of Char;

...

MySet 1:= ['a','b','c']; {оператор определения множества}

Операции над множествами. Допустимые операции над множествами приведены в следующей табл. 6:

Таблица 6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Опера-ция | Наименование операции | Тип операндов | Тип результата | Пример |
| +  –  \*  <=  >=  =  <>  in | Объединение  Вычитание  Пересечение  Не меньше  Не больше  Равенство  Неравенство  Принадлежание | set  set  set  set  set  set  set  элемент set | set  set  set  boolean  boolean  boolean  boolean  boolean | Set1 + Set2  S - T  S \* T  Q <= MySet  S1 >= S2  S2 = MySet  MySet <> S1  A in Set1 |

Объединение, вычитание и пересечение множеств.

Результатом любой из операций будет также множество.

Пример:

Var

S1, S2,S3 : set of Byte;

...

S1:= [1, 2 , 3, 4]; {оператор определения множества}

S2:= [3, 4, 5, 6, 78]; {оператор определения множества}

S3:= S1 + S2; {объединение множеств}

{результат S3 = [1, 2, 3, 4, 5, 6, 78] }

S3:= S2 - S1; {вычитание множеств}

{результат S3 = [1, 2, 5, 6, 78] }

S3:= S2 \* S1; {пересечение множеств}

{результат S3 = [3, 4] }

Операции сравнения множеств.

Результатом любой из операций будет логическая константа True (истина) или False (ложь).

Пример:

Var

S1, S2, S3 : set of Byte;

B: boolean;

...

S1:= [3, 4]; {оператор определения множества}

S2:= [1, 3, 4]; {оператор определения множества}

S3:= [3, 4, 5, 6, 78]; {оператор определения множества}

B:= S1 <= S3; {True, т. к. S1 является подмножеством S3}

B:= S3 >= S2; {False, т. к. S2 не является подмножеством S2}

B:= S3 = S2; {False, т. к. мн-ва S2 и S3 не равны друг другу }

B:= S3 <> S2; {True, т. к. мн-ва S2 и S3 не равны друг другу }

Проверка вхождения элемента во множество. Результатом операции in будет логическая константа True (истина) или False (ложь). Пример:

Var

S1 : set of Integer;

B: boolean;

...

S1:= [3, 4, 18 .. 178, 3101, 4427]; {оператор определения множества}

B:= ( 4 in S1); {True, т. к. 4 является элементом множества S1}

B:= (200 in S1); {False, т. к. 200 не является элементом S1}

7.4. Файловые типы

В языке Object Pascal есть три типа файлов:

текстовые файлы,

файлы с типом,

файлы без типа.

Связь с файлом может быть установлена через файловую переменную, которая после описания, но до использования в программе должна быть связана с внешним файлом с помощью процедуры AssignFile.

Текстовой файл – это последовательность символьных строк перемен-ной длины. Всякая такая строка завершается маркером конца строки CR/LF. Текстовые файлы можно обрабатывать только последовательно. Ввод и вывод нельзя производить для открытого файла, используя одну файловую переменную. Текстовой файл имеет тип TextFile, или просто Text. Пример описания файловой переменной текстового типа:

Var Fi: TextFile;

Файлы без типа состоят из компонент одинакового размера, структура которых не известна или не имеет значения. Допустим прямой доступ к любой компоненте файла. Пример объявления файловой переменной файла без типа:

Var F: File;

Файлы c типом состоят из однотипных компонент известной структуры. Допустим прямой доступ к любой компоненте файла. Пример объявления файловых переменных для файлов с типом:

Type

TRec = **Record**

A: Real;

B: Integer;

C: Char;

**End;**

Var

F : File of Real;

Q : File of String[100];

Fr: File of TRec;

В этом примере F объявлена как файловая переменная вещественного типа. Это означает, что компонентами файла могут быть только вещественные значения. Файловая переменная Q предназначена для доступа к файлам, которые состоят из символьных строк длины 100. Файловая переменная Fr предназначена для работы с файлами, которые состоят из записей типа TRec, объявленного в секции Type.

8. Совместимость типов

Необходимым условием корректного вычисления выражений или выполнения операторов присваивания является совместимость типов входящих в них компонент.

Суть совместимости типов удобнее пояснить на примере простейших выражений, которые состоят из одного (для одноместных операций) или двух (для двухместных операций) компонент (операндов) и одной операции.

8.1. Совместимость по вычислению

Вычисление выражений возможно только при соблюдении следующих условий.

Типы операций и операндов эквивалентны.

Например, нельзя применять арифметические операции к логическим переменным и, наоборот, логические операции – к арифметическим переменным:

Type

R1, R2 : Real;

L1, L2 : Integer;

B1, B2: boolean;

...

**Not** (R1 + R2) B1 + B2 'Иванов' + ' ' + 'Петр' {недопустимые выражения}

Типы операндов эквивалентны.

Типы операндов целые или вещественные, например:

R1 + R2 L1 + R2 L2 / R1 / L1 {допустимые выражения}

Один тип является базовым, а второй – ограниченным типом этого же базового типа.

Type

L11, L12 : Integer;

K : -199 .. 199;

Типы являются множествами, причем их базовые типы совместимы.

**Type**

L : set of 21 .. 2141;

K : set of -199 .. 199;

Один тип является строковым, а другой – также строковым либо символьным.

Type

L : String [34]; Q : String [23]; K: Char;

Один тип является ссылочным, а другой – также ссылочным либо безтиповым указателем.

Оба типа являются упакованными символьными массивами с одинаковым числом элементов.

Один тип является строковым, а другой – также строковым типом, либо упакованным символьным массивом, либо символьным типом.

Один имеет тип Variant, а другой – тип integer, real, string, character или Boolean.

Оба типа операндов являются процедурными типами с одинаковым количеством параметров и идентичными порядковыми типами этих параметров. Для функций должны быть идентичны и типы результата.

8.2. Совместимость по присваиванию

Оператор присваивания считается корректным, если тип переменной, расположенной в его левой части, совместим с типом выражения, располо-женного в правой части. Выражение T2 может быть присвоено переменной T1, если будут соблюдены следующие условия.

Оба типа T1 и Т2 идентичны. Недопустимо присваивание файловых типов или структур, содержащих файловые типы (о файловых типах подробнее см. гл. 17).

T1 и Т2 имеют совместимые простые типы.

T1 и Т2 имеют вещественные типы.

T1 и Т2 имеют целые типы.

T1 и Т2 имеет тип PChar или другой строковый тип, и выражение представляет строковую константу.

T1 и Т2 имеют строковые типы.

T1 имеет строковый тип, а Т2 – символ или упакованная строка.

T1 – длинная строка, а Т2 имеет тип PChar.

T1 и Т2 имеют совместимые типы упакованных строк.

.T1 и Т2 имеют совместимые множественные типы.

.T1 и Т2 имеют совместимые Pointer-типы.

.T1 имеет тип PChar или PWideChar, а T2 есть символьный массив вида array[0 .. n] of Char.

.T1 и T2 имеют совместимые процедурные типы.

.T1 имеет тип Variant, а T2 – один из типов integer, real, string, character или Boolean.

.T1 имеет тип integer, real, string, character или Boolean, а Т2 – тип Variant.

9. Выражения

Вычислительная система выполняет вычислительные и управляющие операции по командам, которые представлены в программе с помощью операторов. Большинство таких операторов строится с использованием выражений, которые в практике программирования играют большую роль, определяя способ и порядок преобразования данных. Выражения состоят из операндов (значений, констант, переменных, функций), соединенных с помощью операций. Для изменения порядка выполнения операций могут быть использованы круглые скобки. Наиболее важную роль играют арифметические, логические и строковые выражения.

9.1. Арифметические выражения

При описании арифметических выражений для простоты типом Integer обозначен любой целый тип, а типом Real – любой вещественный тип. Выражение строится с помощью арифметических значений, констант, переменных, функций, арифметических операций. В выражениях можно применять круглые открывающие и закрывающие скобки. При этом количество открывающих скобок должно быть равно количеству закрывающих скобок.

При вычислении выражения операции выполняются в строго определенной последовательности в соответствии с их приоритетом. Порядок выполнения операций можно изменить применением блоков, включающих подвыражения, заключенные в круглые скобки.

В языке Object Pascal существует шесть арифметических операций. Учитывая, что арифметические операции образуют подмножество множества всех операций языка, в табл. 7 показано абсолютное значение приоритета каждой операции.

Таблица 7

|  |  |  |
| --- | --- | --- |
| Операция | Наименование | Приоритет |
| +  -  \*  /  div  mod | Сложение  Вычитание  Умножение  Деление  Деление нацело  Остаток от деления | 2  2  1  1  1  1 |

При вычислении выражения его тип определяется типами операндов. Операндом называется любая компонента, к которой применяется операция. Операндом может быть, например, значение, константа, переменная или выражение, заключенное в скобки. Типы элементарных выражений показаны в табл. 8.

Таблица 8

|  |  |  |  |
| --- | --- | --- | --- |
| Операция | Тип операндов | Тип результата | Пример |
| +  –  \*  /  div  mod | Integer, real  Integer, real  Integer, real  Integer, real  Integer  Integer | integer, real  integer, real  integer, real  real  integer  integer | X + Y  Result - 1  P \* InterestRate  X / 2  Total div UnitSize  Y mod 6 |

Примеры:

4\*5 = 20, 6/5 = 1.2, 8+7 = 15, 7-3 = 4, 16 div 5 = 3, 16 mod 5 = 2.

Порядок выполнения операций определяется приоритетом операций и расположением внутренних выражений, заключенных в круглые скобки. Все операции в арифметическом выражении выполняются слева направо.

Пример:

|  |  |
| --- | --- |
| Выражение: | 15 \* ((25/5-5\*9 + (j-8) \* 7.55) / 8.67) |
| Порядок выполнения операций: | 8 2 5 3 6 1 4 7 |

9.2. Логические выражения

Результатом вычисления логического выражения может быть одно из двух логических значений: True (истина ) или False (ложь).

Логическое выражение строится с помощью других выражений, (арифметических, строковых и др.), значений, констант, переменных, функций, логических операций и логических отношений.

В языке существует четыре логических операций. Приоритет операций показан в табл. 9.

Таблица 9

|  |  |  |
| --- | --- | --- |
| Операция | Наименование | Приоритет |
| Not  And  Or  Xor | Отрицание  Конъюнкция  Дизъюнкция  Спец. дизъюнкция | 3  4  5  6 |

Значения элементарных логических выражений, поясняющих назначение этих операций, приведены в табл. 10.

Таблица 10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A | B | not A | A and B | A or B | A xor B |
| True  True  False  False | True  False  True  False | False  False  True  True | True  False  False  False | True  True  True  False | False  True  True  False |

В табл. 11 представлены логические отношения.

Таблица 11

|  |  |
| --- | --- |
| Отношение | Наименование |
| >  <  >=  <=  =  <> | Больше  Меньше  Больше или равно  Меньше или равно  Равно  Не равно |

Все отношения равноприоритетны.

Порядок выполнения операций при вычислении логического выражения следующий:

сначала вычисляются арифметические выражения;

затем – отношения;

в последнюю очередь вычисляются логические операции.

Примеры (для x=12, z = 0, y=1):

Таблица 12

|  |  |
| --- | --- |
| Выражение | Результат |
| 5 > 8  (5 < 3) and (z = 0)  ((4 + 8) < 0) or not (y = 0)  not ((x < y) and (z > y))  ('ab' = 'ac') and (x=z)  (4 in [ 2 .. 23 ]) | True  False  True  True  False  True |

Пример, демонстрирующий порядок выполнения операций при вычислении логического выражения:

|  |  |
| --- | --- |
| Выражение: | **Not** ((x > 6 + 8 \* 2) **and** (y < 7) **or** (z > 7)) **and** (x <> y) |
| Порядок: | 9 3 2 1 6 4 7 5 10 8 |

9.3. Строковые выражения

Строковые выражения, частными случаями которых могут быть пустой символ '' или одиночный символ (например 'A'), строятся из строковых или символьных значений, констант, переменных и строковых функций при помощи строковой операции конкатенации (присоединения). Эта операция обозначена символом + (плюс). Скобки в строковых выражениях не применяются.

Пример:

|  |  |
| --- | --- |
| Выражение: | 'Object '+'Pascal '+' для Delphi' |
| Результат: | 'Object Pascal для Delphi' |

10. Операторы

Оператор – языковая конструкция, представляющая описание команды или комплекса команд по обработке и преобразованию данных.

Все операторы делятся на две части – простые операторы и структурные операторы.

11. Простые операторы

К их числу относятся: оператор присваивания, оператор безусловного перехода, составной оператор, оператор процедур, специальные операторы.

11.1. Оператор присваивания

В параграфе 1.4 было дано краткое определение этого оператора – одного самых простых и наиболее часто используемых операторов. Напомним, что его синтаксис имеет вид

x := y;

где x – имя переменной или функции; y – совместимое по типу выражение (о совместимости типов см. гл. 8). Символы ":=" обозначают операцию присваивания, в соответствии с которой вычисленное значение выражения y присваивается переменной x.

Примеры операторов присваивания (комментарии показывают присвоенные значения):

Var

Ch: Char;

S: String[5];

Q: String[18];

L, J: Integer;

P: Byte;

R: Real;

B: Boolean;

Rec: **Record**

A: Word;

B: String[20];

**End;**

…

Таблица 13

|  |  |
| --- | --- |
| Оператор присваивания | Значение |
| Q:= 'd:\Dir1\Word\a.txt';  S:= Q;  Q:= S+'r\';  Ch:=Q[2];  L:= 450;  P:= L;  J:= 100;  R:= -L / J;  J:=-L / J;  J:=-L - 200;  B:= J > L;  B:= (J < L) and (Q[5] = 'i');  Rec.A:= J-20;  Rec.B:= 20;  Rec.B:= S[1]+S[3]+'\d'; | 'd:\Dir1\Word\a.txt'  'd:\Di'  'd:\Dir\'  ':'  450  Ошибка, т. к. max P = 255  100  -4.5  Ошибка. Несоответствие типов  250  False  True  230  Ошибка. Несоответствие типов  'd\\d' |

11.2. Оператор безусловного перехода

Этот оператор выполняет передачу управления оператору, которому предшествует метка. Синтаксис оператора:

Goto Метка;

Язык допускает в качестве меток использовать имя или значение целого типа из диапазона 1 .. 9999.

При использовании операторов перехода необходимо придерживаться следующих правил:

Все метки, используемые в блоке, должны быть описаны словом Label.

Пример оператора описания меток:

Label 1, 2, Met1, Met2, Met3;

Метка должна располагаться в том же блоке, что и оператор, который ею помечен.

Недопустим переход к метке внутрь структурного оператора (например, внутрь цикла, минуя его заголовок). Компилятор не реагирует на эту ситуацию, однако поведение программы может быть непредсказуемо. Недопустимы также вход или выход по метке в процедурах или функциях.

П р и м е ч а н и е. Не используйте меток, если в этом нет особой необходимости. В технике современного программирования использование меток считается правилом плохого тона, т. к. их применение часто ведет к составлению плохо структурированных модулей, усложняющих чтение, сопровождение и отладку программ.

11.3. Оператор обращения к процедуре

Этот оператор вызывает активизацию операторов, расположенных в теле процедуры (см. параграф 15.1). После выполнения процедуры управление передается к оператору, расположенному вслед за оператором процедуры.

При вызове процедуры её формальным параметрам должны строго соответствовать по совместимости типов и количеству фактические параметры.

Примеры обращения к процедурам:

Val (s, r, Code);

Sort (a, n \* 2);

SaveParameters;

11.4. Обращение к функции

Следует подчеркнуть, что не существует специального оператора обращения к функции (см. параграф 15.1). Обычно такое обращение производится посредством другого оператора, часто оператора присваивания.

Обращение к функции активизирует ее внутренние операторы точно так же, как это происходит в процедурах. Принципиальное отличие между выполнением процедуры и функции состоит в следующем:

после выполнения оператора процедуры управление передается к следующему за ним оператору;

после выполнения функции управление вновь передается в оператор, который содержит обращение к этой функции, с целью передачи в него вычисленного значения функции и для завершения вычислений внутри этого оператора.

Поясним это на примере оператора присваивания, содержащего обращение к функции Func8:

G:= 2 \* Pi \* Func8(m, n, a) / Sqr (z);

При выполнении этого оператора сначала происходит обращение к функции Func8. После выполнения операторов, составляющих тело этой функции, вычисленное значение возвращается в этот оператор, где оно используется для выполнения дальнейших вычислений внутри оператора.

12. Стандартные процедуры и функции

Язык содержит ряд процедур и функций, которые в практике программирования ипользуются наиболее часто. Расширенный список процедур и функций, которые могут найти применение в практике программирования, приведен в приложении.

12.1. Строковые процедуры и функции

**Function** Length (St): LongInt;

Возвращает длину строки символов St, т. е. количество символов в ней (не путать с размером строки).

St:= '1234abc';

L:= Length(St); {L= 7}

**Procedure** Delete (St, Pos, Num);

Удаляет Num символов из строки St начиная с позиции Pos, если Pos<Length (St).

St:= '1234abc';

Delete(St, 4, 2); { St= '123bc'}

Delete(St, 3, 120); { St= '12'}

**Procedure** Insert (Obj, Target, Pos);

Вставляет строку Obj в строку Target начиная с позиции Pos. Если Pos>Length (Target), то результат Target + Obj.

St1:= '\*\*\*';

St2:= '1234abc';

Insert (St1, St2, 3) { St2= '12\*\*\*34abc'}

**Procedure** Str (Value, St);

Преобразует значение Value числового типа в строку символов St. Value может сопровождаться форматом.

L:=19;

Str (L, g); {g= '19'}

R:= 2.123155;

Str (R: 8:3, h); {h= ' 2.123' (длина 8, в дробной части 3}

**Procedure** Val (St, Vr, Code);

Преобразует строку символов St в числовую величину Vr целого или вещественного типа. Code = 0, если преобразование прошло успешно, иначе в Code будет записан номер первого ошибочного символа конвертируемой строки, при этом значение Vr не определено.

St:='319';

Val (St, k, Cod); {k= 319, Cod = 0}

St:='81yy9';

Val (St, k, Cod); {k= ?, Cod = 3}

**Function** Copy (St, Pos, Num): String;

Выделяет из строки St подстроку символов длиной Num начиная с позиции Pos. Если Pos>Length, то возвращает пустую строку.

St1:='АБВГДЕ';

St2:= Copy(St1, 2, 3); {St2= 'БВГ'}

St2:= Copy(St1, 2, 27); {St2= 'БВГДЕ'}

St2:= Copy(St1, 44, 2); {возвращает пустую строку St2= ''}

**Function** Concat (St1, St2{, …, StN}): String;

Объединяет строки в одну строку.

St:='abc';

St1:=Concat( 'sss', St, '1234'); {St1= 'sssabc1234'}

St1:=Concat( St, '123'); {St1= 'abc123'}

**Function** Pos (Obj, Target): Integer;

Возвращает номер символа, начиная с которого строка Obj первый раз входит в строку Target. Если строка Obj отсутствует в строке Target, то Pos = 0.

Q:= 'Иванов Сергей Петрович';

H:= Pos ('Сергей', Q); {H= 7}

H:= Pos ('Игорь', Q); {H= 0}

**Function** SizeOf (Obj): Integer;

Возвращает размер переменной Obj.

**Function** FormatFloat(const Format: string; Value: Extended): string;

Возвращает форматированное вещественное значение в виде строки. Format – формат числа, Value – число. В табл. 14 даны форматы функции FormatFloat.

Таблица 14

|  |  |
| --- | --- |
| Формат | Описание |
| 0  #  .  ,  E+, E-    ; | Поле для цифры. Недостающие позиции заменяются нулями  Поле для цифры. Если в позиции имеется значащая цифра, то оно выводится. Нули не выводятся  Разделитель целой и дробной частей  Поле разделителя тысяч, миллионов  Формат представления чисел с плавающей точкой. Если "+" указан, то перед порядком выводится знак. Если указан "-", то минус выводится только для отрицательных порядков  Разделитель форматов |

Примеры действия форматов при конвертации числа в строку представлены в табл. 15.

Таблица 15

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Формат | Число 1 | Число 2 | Число 3 | Число 4 |
| 0  0.00  #.##  #,##0.00  #,##0.00;(#,##0.00)  #,##0.00;;Zero  0.000E+00  #.###E-0 | 1234  1234  1234.00  1234  1,234.00  1,234.00  1,234.00  .234E+03  1.234E3 | -1234  -1234  -1234.00  -1234  -1,234.00  (1,234.00)  -1,234.00  -1.234E+03  -1.234E3 | 0.5  1  0.50  .5  0.50  0.50  0.50  5.000E-01  5E-1 | 0  0  0.00  0.00  0.00  Zero  0.000E+00  0E0 |

12.2. Стандартные функции

***Function*** Char (X: byte): Char;

Возвращает символ с номером X.

Ch:= Char(74); {Ch= 'J'}

**Function** Ord (X): LongInt;

Возвращает порядковый номер скалярного аргумента.

j:= Ord('J'); {j= 74}

**Function** Round (X: Real): LongInt;

Возвращает округленное до целого значение вещественного аргумента.

j:= Round(12.8235); {j= 13}

**Function** Trunc (X: Real): LongInt;

Возвращает целое путем отбрасывания дробной части вещественного аргумента.

j:= Round(12.8235); {j= 12}

12.3. Арифметические процедуры и функции

**Function** Frac (X: Extended): Extended;

Возвращает дробную часть аргумента, например:

r:= Frac(-12.82); {r = -0.82, Frac(12.82)=0.82 }

**Function** Abs (X: Extended): Extended;

Возвращает абсолютное значение аргумента, например:

r:= Abs(-12.82); {r = 12.82}

**Functio**n ArcTan (X: Extended): Extended;

Возвращает арктангенс аргумента.

**Function** Cos (X: Extended): Extended;

Возвращает косинус аргумента.

**Function** Sin (X: Real): Real;

Возвращает синус аргумента.

**Function** ArcCos(X: Extended): Extended;

Возвращает арккосинус аргумента, значение которого должно принадле-жать отрезку [-1, 1]. Возвращает значение из отрезка [0, Pi].

**Function** ArcSin(X: Extended): Extended;

Возвращает арксинус аргумента, значение которого должно принадле-жать отрезку [-1, 1]. Возвращает значение из отрезка [-Pi/2, Pi/2].

**Function** ArcTan2(Y, X: Extended): Extended;

Возвращает арктангенс аргументов, вычисляя ArcTan(Y/X) в соответ-ствии с квадрантами координатной плоскости xOy. Возвращает значение из отрезка [-Pi, Pi].

**Function** Exp (X: Real): Real;

Возвращает экспоненту аргумента.

**Function** Sinh(X: Extended): Extended;

Возвращает гиперболический синус аргумента.

**Function** Cosh(X: Extended): Extended;

Возвращает гиперболический косинус аргумента.

**Function** Tanh(X: Extended): Extended;

Возвращает гиперболический тангенс аргумента.

**Function** ArcSinh(X: Extended): Extended;

Возвращает гиперболический арксинус аргумента.

**Function** ArcCosh(X: Extended): Extended;

Возвращает гиперболический арккосинус аргумента.

**Function** ArcTanh(X: Extended): Extended;

Возвращает гиперболический арктангенс аргумента.

**Function** Ln (X: Real): Real;

Возвращает натуральный логарифм аргумента.

**Function** Sqr (X: Real): Real;

Возвращает квадрат аргумента.

**Function** Sqrt (X: Real): Real;

Возвращает квадратный корень аргумента.

**Function**Ceil(X: Extended):Integer;

Возвращает наибольшее целое аргумента.

Сeil(-2.8) = -2

Ceil(2.8) = 3

Ceil(-1.0) = -1

**Function** Floor(X: Extended): Integer;

Возвращает наименьшее целое аргумента.

Ceil(-2.8) = -3

Ceil(2.8) = 2

Ceil(-1.0) = -1

**Function** Dec (X, [n]: LongInt): LongInt;

Уменьшает значение аргумента на величину второго параметра. Если он отсутствует, то уменьшает на 1.

J:=67;

K:=Dec(J); {j= 66}

K:=Dec(J, 13); {j= 53}

**Function** Inc (X, [n]: LongInt): LongInt;

Увеличивает значение аргумента на величину второго параметра. Если он отсутствует, то увеличивает на 1.

J:=67;

K:=Inc(J); {j= 68}

K:=Inc(J, 13); {j= 81}

12.4. Скалярные функции

**Function** Odd (X: LongInt): Boolean;

Возвращает True, если аргумент четный.

J:=67;

K:=Odd(J); {K= False}

**Function** Pred (X);

Возвращает предшествующее значение типа аргумента.

**Function** Succ (X);

Возвращает последующее значение типа аргумента.

12.5. Процедуры завершения

**Procedure** Exit; Выход из процедуры.

**Procedure** Halt([Code:Word]);

Выход в операционную систему с кодом возврата, если он указан.

12.6. Процедуры и функции для работы с типами "дата/время"

Типы TDateTime и TTimeStamp, а также производные от них типы предназначены для хранения даты и времени. Эти типы используются в ряде весьма полезных и необходимых процедур и функций для работы с датами и временем.

**Function** Now: TDateTime;

Возвращает текущую дату и время.

**Function** Date: TDateTime;

Возвращает текущую дату.

**Function** Time: TDateTime;

Возвращает текущее время.

**Function** DateToStr (D: TDateTime): String;

Преобразует дату в строку символов, например:

S:= DateTimeToStr(Date); {текущая дата '26.10.99'}

**Function** TimeToStr(T: TDateTime): String;

Преобразует время в строку символов, например:

S:= TimeToStr(Time); { текущее время '13.58.13'}

**Function** DateTimeToStr(DateTime: TDateTime): String;

Преобразует дату/время в строку символов, например:

S:= DateTimeToStr(Now); { текущие дата и время '26.10.99 14.01.51'}

**Function** DateTimeToTimeStamp(DateTime: TDateTime): TTimeStamp;

Конвертирует TDateTime в TTimeStamp, например:

TS:= DateTimeToTimeStamp(Now); {type TS = TTimeStamp}

s:= IntToStr(Ts.Date) + ' ' + IntToStr(Ts.Time); {'730053 51095810' – (прошло дней с 00.00.0000 г. и миллисекунд от полуночи текущего дня}

**Function**TimeStampToDateTime(const TimeStamp: TTimeStamp): TDateTime;

Конвертирует TTimeStamp в TDateTime.

**Procedure** DecodeDate(Date: TDateTime; var Year, Month, Day: Word);

Раскладывет дату Date на год, месяц и день, например:

DecodeDate(Now, Y, M, D);

s:= IntToStr(Y) + ' ' + IntToStr(M) + ' ' + IntToStr(M); {'1999 10 26'}

.**Procedure** DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);

Раскладывет время Time на час, минуты, секунды и миллисекунды, например:

DecodeTime(Now, H, M, S, MS);

ss:= IntToStr(H) + ' ' + IntToStr(M) + ' ' + IntToStr(S) + ' ' + IntToStr(MS);

{'14 22 34 567', т. е. 14 ч 22 мин 34 с 567 мс}

.Function EncodeDate(Year, Month, Day: Word): TDateTime;

Противоположна DecodeDate;

.**Function** EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;

Противоположна DecodeTime;

.**Function** FormatDateTime(const Frmt: string; DateTime: TDateTime): string;

Преобразует DateTime в строку с заданным форматом. Если формат пуст, то функция возвращает строку в формате "c". Следующий оператор присвоит строковой переменной s значение 'Встреча состоится: пятница, 6 Ноябрь, 1999, в 10:30 AM'.

s:= FormatDateTime('"Встреча состоится:" dddd, mmmm d, yyyy, ' +'"в" hh:mm AM/PM', StrToDateTime('6.11.99 10:30am'));

Виды форматов даны в табл. 16.

12.7. Прочие процедуры и функции

**Function** Hi(X): byte;

Возвращает старший байт своего целочисленного аргумента.

**Function** Lo(X) : byte;

Возвращает младший байт своего целочисленного аргумента.

**Procedure** Swap(X);

Меняет старший и младший байты целочисленного аргумента местами.

**Procedure** Randomize;

Инициализирует генератор случайных чисел.

**Function** Random(N: Integer): Integer;

Возвращает случайное число из интервала (0, N).

**Function** SizeOf(X) : Integer;

Возвращает число байт, занимаемых аргументом.

**Procedur**e Move(Var Source, Dest; Count: Integer);

Копирует Count байт из переменной Source в переменную Dest. В случае перекрытия областей памяти пересылка в перекрывающуюся область не производится.

**Function** ParamCount: Word;

Возвращает число параметров, переданных в командной строке.

**Procedure** Break;

Оператор безусловного завершения цикла, процедуры или функции.

. **Procedure** Continue;

Оператор, используемый в цикле для передачи управления в его начало.

Таблица 16

|  |  |
| --- | --- |
| Формат | Описание |
| C    D  Dd  Ddd  Dddd  Ddddd  Dddddd  M  Mm  Mmm  Mmmm  Yy  Yyyy  h  hh  n  nn  s  ss  t  tt  am/pm  ampm  a/p  /  : | Показывает сначала дату в формате дд.мм.гг, затем время в формате чч.мм.сс. Не показывает время, если дробная часть DateTime равна нулю  Показывает день без лидирующего нуля (1 – 31)  Показывает день с лидирующим нулём (01 – 31)  Показывает день недели в укороченном формате (Вос – Суб)  Показывает день недели в полном формате (Воскресенье – Суббота)  Показывает дату в формате дд.мм.гг.  Показывает дату в формате д Месяц год  Показывает месяц без лидирующего нуля (1 – 12)  Показывает месяц с лидирующим нулём (01 – 12)  Показывает месяц в сокращенном виде (Янв – Дек)  Показывает месяц в полном формате (Январь – Декабрь)  Показывает год в виде двух последних цифр (00 – 99)  Показывает год в виде четырех цифр (00000 – 9999)  Показывает час без лидирующего нуля (0 – 23)  Показывает час с лидирующим нулем (00 – 23)  Показывает минуту без лидирующего нуля (0 – 59)  Показывает минуту с лидирующим нулем (00 – 59)  Показывает секунду без лидирующего нуля (0 – 59)  Показывает секунду с лидирующим нулем (00 – 59)  Показывает время в формате чч:мм  Показывает время в формате чч:мм:сс  Показывает время в 12-часовом формате (am – до полудня, pm – после полудня)  Показывает время в 12-часовом формате без указателя до/после полудня  Использует Windows-разделитель даты.  Использует Windows-разделитель времени |

. **procedure** Abort;

Используется в контексте с другим оператором; отменяет "задним числом" оператор в случае его аварийного завершения, блокирует выдачу сообщения об ошибке, удобен к использованию в блоке try … finally.

13. Структурные операторы

К их числу относятся:

* составной оператор,
* условный оператор If,
* оператор варианта Case,
* оператор цикла For – Do,
* оператор цикла While – Do,
* оператор цикла Repeat – Until,
* оператор записи With,
* оператор Try – Except – End,
* оператор Try – Finally – End,
* оператор On – Do,
* оператор Try – Finally – End.

13.1. Составной оператор

Это простая структура следующих друг за другом операторов, заключенных в операторные скобки begin … end.

Синтаксис составного оператора:

Begin

Оператор1

Оператор2

…

ОператорN

End;

Составной оператор применяется в тех случаях, когда какое-либо действие необходимо применить не к одному, а сразу к нескольким операторам.

Пример:

**Begin**

R:= X;

X:= Y;

Y:= R;

**End;**

13.2. Условный оператор If

Синтаксис допускает два вида оператора:

if логическое выражение then оператор1 else оператор2;

и его усеченный вариант:

if логическое выражение then оператор1;

Оператор работает следующим образом. Сначала вычисляется логичес-кое выражение. Если оно истинно, то выполняется оператор1, иначе – оператор2. Усеченный оператор выполняет оператор1 только в случае истинного значения логического выражения и не производит никаких действий в случае его ложности.

Примеры:

if (x < 10.7) then a[4]:= 5 else a[4]:= 6;

if (x < 10.7) then a[4]:= 5;

Допустима вложенность условных операторов внутри составного условного оператора. Например, оператору

if L1 then if L2 then St1 else St2 else St3;

эквивалентен оператор

if L1 then

begin

if L2 then St1 else St2;

end

else St3;

В этом операторе для повышения структурированности использованы операторные скобки begin … end. При конструировании сложного условного оператора во избежание логических ошибок следует отдавать предпочтение структурному способу записи такого оператора.

13.3. Оператор варианта Case

Синтаксис оператора:

Case Selector of

Const1: Оператор1;

Const2: Оператор2;

…

ConstN: ОператорN

[else Оператор];

End;

Selector может быть любой простой тип кроме Real. Каждая из Const1 … ConstN может быть значение, несколько перечисленных через запятую значений или отрезок типа. Оператор Else, как видно из описания, может отсутствовать. В том случае, если он присутствует, то действует общее правило: перед словом Else не должно быть символа ";" (точка с запятой). Поясним работу оператора Case на примере:

Case i of

0 : x := 0;

1,3 : x := 1;

10 .. 15: x := 2

**else** x := 3;

End;

При выполнении оператора Case управление будет передано тому оператору, который помечен константой, являющейся значением переменной i. Например, если к моменту выполнения Case-оператора i = 0, то будет выполнен оператор x := 0. Иначе, если i = 1 или i = 3, то будет выполнен оператор x := 1; иначе, если значение i в диапазоне 10 .. 15, то будет выполнен оператор x := 2; наконец, если i не равно ни одной из вышеперечисленных констант, то будет выполнен оператор x := 3, следующий за словом else (иначе).

13.4. Оператор цикла For – Do

Синтаксис оператора имеет две разновидности:

For счетчик цикла:=нач.знач. To конеч.знач. Do оператор

For счетчик цикла:=нач.знач. Downto конеч.знач. Do оператор

Здесь конструкция For .. Do называется заголовком цикла, оператор – телом цикла.

Для циклов должны соблюдаться следующие правила и ограничения:

начальное и конечное значения счетчика цикла должны быть одинаковых простых типов, кроме Real;

в теле цикла счетчик не должен менять значение;

вход в цикл минуя заголовок запрещен;

для первой разновидности начальное значение не должно быть больше конечного;

для второй разновидности начальное значение не должно быть меньше конечного.

Первая разновидность оператора цикла For работает следующим образом. Сначала счетчик цикла принимает нач.знач. и выполняется оператор, расположенный вслед за словом Do. Затем значение счетчика будет увеличено на шаг счетчика 1 и вновь будет выполнен оператор и т. д., пока счетчик не переберет все значения от нач.знач. до конеч.знач.

Пример.

s:= 0;

**For** i:=1 **to** 44 **do** s:= s + z[i];

В результате в переменной s будет накоплена сумма первых 44 элементов массива z.

Другая разновидность оператора For отличается лишь отрицательным шагом –1 счетчика.

Пример.

s:= 0;

**For** i:=44 **Downto** 1 **do** s:= s + z[i];

Будет получен тот же результат.

13.5. Оператор цикла While – Do

Синтаксис оператора:

While логическое выражение Do оператор;

Цикл выполняет оператор, расположенный вслед за словом Do до тех пор, пока истинно логическое выражение, расположенное за словом While ("выполняй, пока истинно").

Пример.

x:= 0;

i:=0;

**While** (x < 101.667) do

**Begin**

Inc (i);

X:= X + 5.617;

Y[i]:= Func (i + 6, 9 \* i, X);

**End**;

В этом примере цикл будет выполняться до тех пор, пока не выполнится условие x < 101.667. В теле цикла переменная X с каждым шагом цикла увеличивает свое значение на 5.617 так, что на определенном шаге условие x < 101.667 впервые не будет выполнено. В этот момент без входа в тело цикл закончит работу.

13.6. Оператор цикла Repeat – Until

Синтаксис оператора:

Repeat

Оператор1;

Оператор2;

…

ОператорN;

Until логическое выражение;

Цикл работает, пока логическое выражение ложно ("повторяй, пока не выполнится").

Пример.

s:= 0;

i:=0;

***Repeat***

Inc (i);

s:= s + z[i];

***Until*** (i = 44);

В этом примере цикл будет выполняться до тех пор, пока не выполнится условие i = 44. Результат будет тот же, что в примере для For-цикла.

13.7. Операторы Break и Continue

Оператор Break может быть размещен в теле цикла. При его выполнении цикл прекращает работу и считается выполненным.

Пример.

s:= 0;

i:=0;

**Repeat**

Inc (i);

s:= s + z[i];

if (s > 14) then Break;

**Until** (i = 44);

В этом примере цикл будет выполняться до тех пор, пока не выполнится условие i = 44 или если в операторе if переменная s превысит значение 14.

Оператор Continue также может быть размещен в теле цикла. При его выполнении управление независимо от того, где он расположен, сразу передается в начало цикла для выполнения следующего шага.

Пример.

s:= 0;

i:=0;

***Repeat***

Inc (i);

s:= s + z[i];

**if** (s > 20) **then** Continue;

**if** (s > 14) **then** Break;

***Until*** (i = 44);

В этом примере если в первом операторе if выполнится условие s > 20, то сработает оператор Continue. Он сразу передаст управление на первый оператор в теле цикла – Inc (i), предотвратив тем самым выполнение ниже-следующих операторов – второго if и Until.

13.8. Вложенные циклы

В теле оператора цикла могут быть размещены другие операторы цикла. Такие структуры называются вложенными циклами. Язык допускает любую глубину вложенности циклов. При использовании вложенных циклов необходимо иметь в виду следующее:

все вложенные циклы For – Do должны иметь различные счетчики (иначе это противоречило бы требованию на запрет изменения значения счетчика внутри цикла);

нет никаких ограничений на досрочный выход из внутреннего цикла наружу;

недопустим вход во внутренний цикл For – Do, минуя его заголовок, что соответствует общему требованию о корректном входе в цикл.

Вложенные циклы используются в ситуациях, когда на каждом шаге наружного цикла необходимо полностью выполнить внутренний цикл.

Пример.

Const

n = 15;

m = 24;

Var

i,j: Byte;

R,Tau,s: Real;

z: array[1..n, 1..m] of Real;

…

{заполнение массива z с использованием вложенных циклов}

Tau:= Pi/m;

For i:=1 to n do begin

R:=4.0\*Pi\*Sin(i\*Tau); {первый оператор в теле цикла по i}

For j:=1 to m do z[i, j] := R+j; {второй оператор в теле цикла по i}

end {i};

{вычисление суммы положительных элементов массива z с использованием вложенных циклов }

s:=0;

For i:=1 to n do

For j:=1 to m do

if ( z[i, j] > 0) then s:= s + z [i, j];

Приведенный пример содержит две структуры вложенных циклов. Первая структура предназначена для заполнения элементов двумерного массива z с помощью математической формулы



Наружный цикл со счетчиком i в теле цикла содержит два оператора – оператор присваивания (вычисление значения вспомогательной переменной R с целью сокращения времени вычислений) и оператор внутреннего цикла со счетчиком j. Поскольку наружный цикл в своем теле содержит несколько операторов, то они заключены в операторные скобки begin … end.

Эта структура работает следующим образом. После входа в наружный цикл переменная i (счетчик этого цикла) примет значение 1. Далее будет вычислено значение переменной R при i = 1. После этого будет выполнен внутренний цикл со счетчиком j, где j на каждом шаге будет последовательно принимать значения 1, 2, 3, … m (i при этом остается неизменным и равным 1). В результате будут вычислены элементы z11, z12, …,z1m первой строки массива. Затем будет выполнен возврат к заголовку наружного цикла, где значение счетчика i будет увеличено на 1 (т. е. i станет равно 2) и вновь будет выполнены операторы, расположенные в его теле. В результате будут определены элементы z21,z22, …,z2m второй строки массива и т.д.

Вторая структура вложенных циклов предназначена для вычисления суммы положительных элементов массива z. Для этого сначала переменной s будет присвоено значение 0, а затем во вложенных циклах будет накоплена требуемая сумма в ячейку s.

13.9. Оператор записи With

В ранних версиях языка оператор использовался для более удобного доступа к полям записи.

Пример:

Var

Student : ***Record***

Fam: String[30];

Name: String[20];

Age: Word;

***End;***

…

Student.Fam:= 'Колокольников';

Student.Name:= 'Павел';

S:=Student.Fam + ' '+Student.Name;

{предыдущих три оператора эквивалентны следующим}

***With*** Student do

**Begin**

Fam:= 'Колокольников';

Name:= 'Павел';

S:= Fam + ' '+ Name;

***End;***

13.10. Оператор Try – Except – End

Это новейшее средство языка. Блок Try – Except – End используется для предотвращения исключительных ситуаций (ИС), которые могут возникнуть при выполнении программы. К их числу относятся сбои в работе аппаратуры, ошибки вычислений (например деление на нуль), попытки присвоить значение, выходящее за пределы типа и т. д.

Синтаксис:

***Try***

{операторы, способные генерировать ИС}

***Except***

{операторы, обрабатывающие генерированные ИС}

***end*;**

Блок Try – Except – End работает следующим образом. Выполнение начинается с операторов, расположенных в блоке Try – Except. Если в каком-либо операторе возникает ИС, то она подавляется и затем выполняются все операторы, расположенные в блоке Except – End. В результате предотвращается аварийное прерывание программы. Использование блока Try – Except – End открывает возможность программного контроля за ИС.

Пример.

i:= 0;

n:= 8;

Try

i:= n div i; {Деление на нуль. Оператор генерирует ИС}

n:= i + 9;

Except

ShowMessage('Ошибка. Деление на нуль в операторе i := n / i');

End;

Результатом выполнения блока операторов будет появление на экране формы с сообщением "Ошибка. Деление на нуль в операторе i := n / i", после чего программа продолжит работу с оператора, следующего за словом End, а не с оператора n := i + 9.

Если бы оператор i := n div i не был защищен блоком Try – Except – End, то возникшая при его выполнении ИС привела бы к нежелательному аварийному завершению программы.

13.11. Оператор On – End

При возникновении ИС язык позволяет не только предотвратить прерывание программы, но и определить, какого именно вида была ИС. Для этого в блоке Except – End можно использовать оператор On –Do.

Пример

i:= 0; n:= 8;

***Try***

i:= n div i; {Деление на нуль. Оператор генерирует ИС}

n:= i + 9;

***Except***

On Ex: EdivByZero do ShowMessage('Деление на нуль');

***End;***

В этом примере сообщение о возникновении ИС будет выдано только в случае, когда ИС будет только деление на нуль (EdivByZero). Во всех остальных случаях ИС будет предотвращена, однако никакого сообщения о ее возникновении выдано не будет. Объявленная в блоке Except – End переменная Ex может быть любым именем (здесь Ex используется только для примера).

13.12. Оператор Try – Finally – End

Блок Try – Finally – End также используется для предотвращения ИС, которые могут возникнуть при выполнении программы. В отличие от блока Try – Except – End блок Try – Finally – End используется для освобождения ресурсов памяти, закрытия файлов и пр. в случае возникновения ИС.

Синтаксис:

***Try***

{операторы, способные генерировать ИС}

***Finally***

{операторы освобождения ресурсов памяти }

***end*;**

Блок Try – Finally – End работает следующим образом. Выполнение начинается с операторов блока Try – Finally, которые в правильно написанной программе должны содержать операторы выделения ресурсов памяти. Если в каком-либо операторе возникает ИС, то управление сразу передается к операторам блока Finally – End, где производится освобождение памяти, закрытие файлов и пр. В результате, с одной стороны, предотвращается аварийное прерывание программы и, во вторых, корректно освобождается ранее зарезервированная память, выполняется ряд других необходимых операций.

Отметим, что блок Finally – End выполняется всегда вне зависимости от того, была или не была сгенерирована ИС.

Пример.

i:= 0;

n:= 8;

Try

GetMem (p, 8000); {выделение памяти}

i:= n div i; {Деление на нуль. Оператор генерирует ИС}

n:= i + 9;

Finally

FreeMem (p, 8000); {освобождение памяти}

End;

14. Указатели

В языке есть средство, разрешающее запрашивать память динамически, т. е. по необходимости. Это позволяет уменьшить объем кода программы и экономно расходовать оперативную память. Такое средство представляет собой специальный тип, называемый указателем. Имеется два типа ука-зателей: указатель на объект некоторого типа и указатель без типа.

Тип Pointer образует указатель без типа. Указатель на тип имеет синтаксис:

^ Имя типа

Примеры объявления указателей:

Type

tDinArr = Array[1 .. 1000, 100] of String[255]; {обычный тип}

tDinArrPtr = ^tDinArr; {указатель на тип tDinArr}

tRecPtr = ^tRec; {указатель на тип записи, который описан ниже}

tTRec = **Record** {обычный тип-запись}

A: Integer;

B: Real;

C: String[255];

**End;**

Var

DinArr: tDinArr; {обычная запись}

DinArrPtr: tDinArrPtr; {указатель на тип}

RecPtr: tRecPtr; {указатель на тип-запись}

Pn1, Pn2: Pointer; {указатели без типа}

Модули System и SysUtils содержат большое количество типов для работы с указателями. Эти типы могут быть использованы для повышения эффективности пользовательских программ, в которых используются указатели. К их числу относятся: PAnsiString, PString, PByteArray, PCurrency, PExtended и ряд других указателей. Впрочем, эти типы могут быть легко заменены стандартными типами. Например PString эквивалентен ^String и т.д.

14.1. Операции с указателями

Для указателей допустимы операции присваивания и сравнения. Указателю можно присваивать:

содержимое указателя такого же типа;

константу Nil (пустой указатель);

адрес объекта с помощью функции Addr;

адрес с помощью оператора @;

адрес, построенный функцией Ptr.

Пример:

TrPt:= Nil;

Klo1Ptr:= Klo2Ptr;

P1:=@Pp; {эквивалентно P1:= Addr(Pp);}

P2:= Ptr($B701);

14.2. Стандартные процедуры и функции для работы с указателями

**Procedure** GetMem(Var: P: Pointer; Size: Word);

Выделяет блок памяти размера Size и присваивает адрес начала блока указателю P.

**Procedure** FreeMem(Var: P: Pointer; Size: Word);

Освобождает блок памяти размера Size, адресованный указателем P.

Ниже приведен подробный пример, демонстрирующий экономный процесс копирования текстового файла 't1.txt' в файл 't2.txt' с использованием указателя Buffer.

Var

F1, F2: file; {объявление файловых переменных}

Buffer: PChar; {объявление указателя на строку }

begin

AssignFile(F1, 't1.txt'); {связывание F1 с файлом 't1.txt'}

Reset(F1, 1); {файл открыт для ввода/вывода}

AssignFile(F2, 't2.txt'); {связывание F2 с файлом 'text.txt'}

Rewrite(F2, 1); {файл открыт для вывода}

try

Size := FileSize(F1); {вычисления размера файла}

GetMem(Buffer, Size); {выделение памяти под чтение файла}

try

BlockRead(F1, Buffer^, Size); {считывание всего файла 't1.txt'}

BlockWrite(F2, Buffer^, Size); {запись в файл 't2.txt'}

finally

FreeMem(Buffer); {освобождение памяти}

end;

finally

CloseFile(F1); {закрытие файла F1}

CloseFile(F2); {закрытие файла F2}

end;

end;

В этом примере объявлен указатель на строку Buffer с завершающим нулем, которая будет использована для копирования файла 't1.txt' в файл 't2.txt'. Для этого оператором GetMem для переменной Buffer^ будет динамически выделен блок памяти размером, равным размеру файла. Далее оператором BlockRead файл 't1.txt', связанный файловой переменной F1, будет считан в Buffer^ и затем оператором BlockWrite переменная Buffer^ будет записана в файл 't2.txt', связанный с файловой переменной F2. Для предотвращения исключительных ситуаций пример содержит два вложенных блока try – finally – end. Внутренний блок обслуживает возможный сбой в ситуации, когда по какой-либо причине файл не удалось прочитать или записать операторами BlockRead или BlockWrite. Такой способ гарантирует освобождение памяти оператором FreeMem как в случае успешного копирования, так и в случае возможного сбоя. Внешний блок обслуживает ситуацию, когда у системы возможно нет того объема памяти, который запрашивает оператор GetMem. В любых вариантах – при успешном или безуспешном копировании файла – следующие за последним finally операторы CloseFile закроют открытые операторами Reset и Rewrite файлы F1 и F2 и позволяет программе продолжить работу.

**Procedure** New(Var: P: Pointer);

Создает новую динамическую переменную того типа, на который ссылается указатель. Эквивалентна оператору GetMem(P, SizeOf(P^));

**Procedure** Dispose(Var: P: Pointer);

Уничтожает динамическую переменную, на которую указывает P. Эквивалентна оператору FreeMem(P, SizeOf(P^));

**Procedure** ReallocMem(var P: Pointer; Size: Integer);

Процедура работает следующим образом:

если P= Nil и Size = 0, то оператор не выполнит никаких действий;

если P= Nil и Size > 0, то оператор сработает аналогично GetMem;

если P <> Nil и Size = 0, то оператор сработает аналогично FreeMem.

**Function** Addr(X): Pointer;

Адрес указанного имени.

14.3. Прочие процедуры и функции для работы с указателями

В модулях System и SysUtils объявлены процедуры и функции, которые могут найти применение в пользовательских программах. Ниже дано описание некоторых функций и процедур.

**Function** GetHeapStatus: THeapStatus;

Расположена в модуле System. Дает сведение о состоянии распределен-ной и доступной программе памяти. Тип функции имеет вид

THeapStatus = record

TotalAddrSpace: Cardinal;

TotalUncommitted: Cardinal;

TotalCommitted: Cardinal;

TotalAllocated: Cardinal;

TotalFree: Cardinal;

FreeSmall: Cardinal;

FreeBig: Cardinal;

Unused: Cardinal;

Overhead: Cardinal;

HeapErrorCode: Cardinal;

end;

Function AllocMem(Size: Cardinal): Pointer;

Выделяет блок памяти и устанавливает каждый байт "в нуль". Освобо-ждение памяти можно выполнить с помощью процедуры FreeMem.

Procedure GetMemoryManager(var MemMgr: TMemoryManager);

Дает текущее состояние менеджера памяти – специальной записи с типом:

TMemoryManager = record

GetMem: function(Size: Integer): Pointer;

FreeMem: function(P: Pointer): Integer;

ReallocMem: function(P: Pointer; Size: Integer): Pointer;

end;

**Procedure** SetMemoryManager(var MemMgr: TMemoryManager);

Устанавливает менеджер памяти – выполняет операции выделения и освобождения памяти в соответствии с предварительно установленными в менеджере памяти значениями.

14.4. Глобальные переменные AllocMemCount и AllocMemSize

В модуле System объявлены две глобальные переменные, значения которых могут быть использованы при контроле за динамически создаваемыми и разрушаемыми пользовательскими переменными.

AllocMemCount – количество блоков выделенной памяти.

AllocMemSize – размер блоков выделенной памяти.

***15. Подпрограммы***

Подпрограмма – это законченная алгоритмическая единица, которая предназначена для выполнения некоторого ограниченного по отношению к использующей ее программе круга операций.

В языке Object Pascal есть два вида подпрограмм – процедуры и функции. Структура всякой подпрограммы во многом напоминает структуру исходного модуля. Каждая такая подпрограмма перед ее использованием должна быть описана. Описанием подпрограммы называется ее исходный код, а обращением к подпрограмме является оператор или его часть, которые содержат код вызова такой подпрограммы. Таким образом, описание – это технология, а обращение – это действия по предписанной технологии.

Всякая подпрограмма может иметь локальные и глобальные по отношению к ней параметры. Локальным является параметр, действие которого ограничено только подпрограммой, в которой он описан. Всякий другой параметр будет глобальным. Все глобальные параметры всегда описаны за пределами подпрограммы, в которой они используются.

15.1. Процедуры

Всякая процедура имеет заголовок и тело. Тело процедуры состоит из операторов, предназначенных для описания имен и действий над данными. Синтаксис процедуры имеет вид

Procedure procedureName(parameterList); directives;

localDeclarations;

begin

statements;

end;

Здесь

Name – имя процедуры,

parameterList – список формальных параметров,

directives – директивы,

localDeclarations – внутренние описания,

statements – операторы тела процедуры.

procedureName – имя процедуры. Именем процедуры может быть любое имя, не совпадающее ни с каким другим описанным в том же, что и процедура, блоке, именем.

parameterList – список формальных параметров может быть либо пуст (в этом случае скобки можно не использовать), либо должен содержать последовательность входных и/или выходных величин. Отдельная величина в описании заголовка может быть:

объявленной с помощью слова var переменной с типом или без типа;

константой;

выходной величиной (т. н. out-параметром).

Пример описания процедуры.

procedure ByRef(var X: Integer; L, K: Integer);

begin

X := X \* 2 \* L; {правильно}

K := 2 + L; {ошибка}

end;

Процедура c именем ByRef содержит три параметра – переменную X и две константы L и K. Тело процедуры состоит из операторов, заключенных в операторных скобках begin – end. Переменные L, K являются только входными и не могут быть изменены в теле процедуры. По этой причине оператор K:= 2 + L не даст результата и значение К останется таким же, каким оно было до обращения к процедуре. Напротив, переменная X, объявленная с помощью слова var, будет вычислена и на выходе будет иметь то значение, которое будет вычислено внутри процедуры ByRef.

Out-параметр являет прямую противоположность константе, он должен быть только выходной, определяемой внутри процедуры, величиной. Например, в следующем коде

procedure GetInfo(out Info: SomeRecordType);

var MyRecord: SomeRecordType;

...

Proc1(MyRecord);

…

переменная MyRecord не может передавать данные в процедуру Proc1. Напротив, только Proc1 может сформировать данные и передать их в MyRecord.

Тип параметра может быть любым. Однако нельзя объявлять новый тип прямо в заголовке подпрограммы. Такой тип должен быть объявлен раньше и только после этого может быть использован в заголовке подпрограммы.

localDeclarations – внутренние описания могут содержать описание локальных имен типов или переменных. В предыдущем примере переменная MyRecord типа SomeRecordType, объявленная внутри процедуры GetInfo, является образцом такого описания.

directives – директивы используются для того, чтобы дать компилятору некоторые дополнительные указания об описываемой процедуре. Директивы описаны в параграфе 15.3.

Для вызова процедуры, т. е. для обращения к заложенному в ней алгоритму, следует записать оператор, в котором на место формальных параметров должны быть подставлены фактические параметры. Следующий пример содержит как описание процедуры MatrMult, так и обращение к ней.

Program {начало программы}

…

Const n = 10; {размер матриц}

Type tMatr = array[1.. n, 1.. n] of Real; {тип матриц}

Var Q1, Q2, Q3: tMatr; {описание матриц}

Procedure MatrMult(M1, M2: tMatr; Var M3: tMatr); {заголовок}

Var i, j, k: Integer;

Begin

For i:= 1 to n do

For j:= 1 to n do

Begin {блок определения одного элемента матрицы M3}

M3[i, j]:=0;

For k:=1 to n do

M3[i, j]:= M3[i, j] + M1[i, k] \* M2[k, j];

End;

End; {конец описания процедуры MatrMult}

Procedure Prim; {заголовок процедуры Prim}

Var i, j: Integer;

Begin

For i:= 1 to n do {блок задания элементов матриц Q1, Q2}

For j:= 1 to n do

Begin

Q1[i, j]:=i + j;

Q2[i, j]:=i - j;

End;

MatrMult(Q1, Q2, Q3); {оператор обращения к процедуре MatrMult}

End; {конец описания процедуры Prim}

… {текст головной программы}

Prim; {обращение к процедуре Prim}

Q3[2, 3]:= 1;

…

Приведенный в данном примере текст программы можно разделить на четыре части:

описание глобальных констант и переменных;

текст процедуры MatrMult;

текст процедуры Prim;

текст головной программы.

В верхней части описана глобальная константа n = 10, которая используется во всех следующих за этим структурах. Так, в секции Type объявлен тип вещественного квадратного массива tMatr. Далее в секции Var объявлены три переменные – матрицы Q1, Q2, Q3 типа tMatr.

Далее дан текст процедуры MatrMult, которая представляет алгоритм перемножения матриц M1 и M2 с записью результата в переменную M3. Матрицы M1, M2 в процедуре не меняются, поэтому их необязательно объявлять переменными (с позиций процедуры MatrMult переменные M1, M2 выступают в качестве констант). Напротив, матрица M3 получается как результат, определяемый внутри процедуры MatrMult, поэтому она объявлена в заголовке словом Var. Необъявление M3 как переменной привело бы к ошибке: измененные внутри процедуры значения этой матрицы не были бы зафиксированы. Переменные i, j, k, объявленные внутри процедуры, являются локальными и действуют только в пределах этой процедуры.

Описание процедур заканчивается процедурой Prim. В теле этой процедуры объявлено две локальные переменные i, j, выполняющих вспомогательную роль счетчиков циклов. Далее расположено два вложенных друг в друга цикла, с помощью которых определяются элементы матриц Q1 и Q2. По отношению к процедуре Prim эти матрицы являются глобальными переменными и, следовательно, они доступны не только во внешнем блоке, но и внутри процедуры Prim. И наконец, в нижней части расположен оператор обращения к процедуре MatrMult, который предназначен для вызова алгоритма перемножения матриц.

В нижней части программы дан фрагмент текста головной программы, содержащей вызов процедуры Prim.

Опишем механизм вычислений, который запускается приведенной программой. Сначала будет выполнена процедура Prim (обращение к ней содержится в самой нижней части текста примера). Эта процедура без параметров, поэтому управление будет сразу передано в тело процедуры, где начнется последовательное выполнение содержащихся в нем операторов. Сначала будут выполнены два вложенных цикла For, где элементы матриц Q1, Q2 будут заполнены значениями (например Q1[1, 1] = 2, Q2[1, 1] = 0 и т. д.). Далее уже внутри Prim произойдет обращение к процедуре MatrMult. При этом сначала произойдет подстановка фактических параметров Q1, Q2, Q3 на место соответствующих формальных параметров M1, M2, M3. Далее управление будет передано внутрь процедуры MatrMult, где аналогично последовательным выполнением ее операторов произойдет перемножение матриц. После выполнения процедуры MatrMult управление будет передано в ту же точку, с которой производился ее вызов. Поскольку вызов MatrMult производился из Prim, то управление будет вновь возвращено в процедуру Prim к оператору, расположенному вслед за оператором вызова MatrMult. Поскольку в Prim больше нет невыполненных операторов, то она также заканчивает свою работу и управление передается в ту алгоритмическую единицу, которая и вызывала Prim, а именно в головную программу, к оператору Q3[2, 3] := 1.

15.2. Функции

В отличие от процедуры функция предназначена для вычисления одного значения любого типа. Тип функции указывается в конце ее заголовка. Тип возвращаемого значения отделяется от списка формальных параметров символом ":" (двоеточие). Кроме того, в теле функции, по крайней мере, один раз должно быть определено значение функции. Это значение присваивается имени функции или переменной Result.

Синтаксис функции имеет вид

function functionName(parameterList): returnType; directives;

localDeclarations;

begin

statements;

end;

Здесь functionName – имя функции; ParameterList, directives, localDeclarations, statements имеют тот же смысл, что и в процедуре; ReturnType – тип возвращаемого результата.

Пример.

Function Fact(n: Word): LongInt; {заголовок функции Fact}

Var i: Word; j: LongInt;

Begin

j:=1;

if (n > 1) then

For i:= 2 to n do j:= j \* i;

Result:= j;

End; {конец описания функции Fact }

… {текст головной программы}

Var r: Real;

…

r:= 3.4 \* Fact(3) / 2.5 / (122 - Fact(5)); {обращение к функции Fact}

…

В этом примере описана функция с именем Fact вычисления факториала n! неотрицательного целого числа. Тип функции определен как LongInt. В теле функции размещен оператор Result:= j, который определяет возвращаемый функцией результат. Способ обращения к функции демонстрирует последний оператор примера. Видно, что способ обращения к функции имеет существенное отличие от способа обращения к процедуре. Так, в этом операторе обращение к функции Fact производится дважды – один раз с фактическим параметром 3, другой – с параметром 5. Далее возвращенные результаты (соответственно, 6 и 120) будут подставлены в выражение правой части оператора, после чего последний будет вычислен и переменная r получит вещественное (Real) значение 4.08.

15.3. Параметры без типа

Это особый вид параметров, который может быть использован только в заголовках имен процедур и функций. Каждый параметр без типа должен быть описан как var, const или out-параметр. Например:

procedure TakeAnything(const C);

описывает С как константу без типа.

Параметр без типа компилятор расценивает как параметр потенциально любого типа. Это означает, что в тело подпрограммы передается только адрес параметра и компилятор не контролирует правильность выполнения операций над таким параметром.

Однако имеется одно исключение: при обращении к подпрограммам, содержащим параметры без типа, нельзя использовать числовые значения или нетипизированные числовые константы.

Следующий пример использует параметры без типа в функции Compare, которая сравнивает размеры двух переменных V1 и V2 и возвращает ответ в виде константы –1, если размер V1 меньше размера V2, нуль – если размеры одинаковы, 1 – если размер V1 меньше размера V2.

function Compare(var V1, V2): ShortInt;

Var i, j: LongInt;

Begin

I:=SizeOf(V1);

J:=SizeOf(V2);

If (I < J) then Result:= -1

Else

If (I > J) then Result:= 1

Else Result:= 0;

End;

Примеры обращений к функции Compare:

type

TVector = array[1..10] of Integer;

TPoint = record

X, Y: Integer;

end;

var

Vec1, Vec2: TVector;

N,i: Integer;

P: TPoint;

…

i:= Compare(Vec1, Vec2) ; {0, размеры одинаковы}

Vec[1]:= Compare(i, Vec1); {-1, размер i меньше размера Vec1}

P.X:= Compare(Vec1, P); {1, размер Vec1 больше размера P}

P.Y:= Compare(i, P); {-1, размер i меньше размера P}

Vec2[8]:= Compare(i, P.X); {0, размеры i и поля P.X одинаковы}

…

15.4. Декларации процедур и функций

Заголовок процедуры или функции может содержать декларацию. Декларация указывается сразу вслед за списком параметров в процедуре или за типом возвращаемого результата в функции.

1. Декларации о вызове по соглашению (calling convention). К их числу относятся декларации register, pascal, cdecl, stdcall и safecall, например:

function MyFunction(X, Y: Real): Real; cdecl;

Этот вид деклараций предназначен для задания способа передачи параметров в процедуру. Декларации register, pascal передают параметры слева направо, cdecl, stdcall и safecall – наоборот, справа налево.

Директивы near, far и export предназначены для разграничения способов обращения в 16-разрядных приложениях. Для современных 32-разрядных приложений они не имеют значения.

Отметим, что упомянутые декларации используются в сложных, весьма тонких ситуациях и для начинающего программиста не представляют большого интереса.

2. Директива Forward указывает на то, что заголовок процедуры или функции объявлен раньше, чем описана сама подпрограмма, например:

**function** Calculate(X, Y: Integer): Real; forward;

…

**function** Calculate;

...

**begin**

...

**end;**

3. Директива External указывает на то, что текст процедуры содержится в отдельном объектном (откомпилированном) модуле. Такой способ позволяет присоединить объектный код к компилируемой программе из указанного модуля. Для этого необходимо указать директиву компилятора со ссылкой на модуль, в котором содержится объектный код декларируемой процедуры, например:

{$L BLOCK.OBJ}

…

procedure MoveWord(var Source, Dest; Count: Integer); external;

procedure FillWord(var Dest; Data: Integer; Count: Integer); external;

Этот пример показывает, что при компиляции коды процедур MoveWord и FillWord следует искать в объектном коде BLOCK.OBJ.

4. Директива OverLoad позволяет использовать одно имя для нескольких подпрограмм, например:

function Divide(X, Y: Real): Real; overload;

begin

Result := X/Y;

***end*;**

function Divide(X, Y: Integer): Integer; overload;

begin

Result := X div Y;

end;

В этом примере описаны две функции с одним именем Divide. При обращении к функции с таким именем вызвана будет та функция, фактические параметры которой соответствуют формальным параметрам. Так, при обращении в виде Divide (6.8, 3.2) будет вызвана первая функция, т. к. ее формальные параметры также вещественны, а при обращении Divide(6, 8) будет вызвана вторая функция.

Директива Overload разрешена для подпрограмм, в которых могут различаться только типы параметров, поэтому недопустимы описания вида

function Cap(S: string): string; overload;

procedure Cap(var Str: string); overload;

15.5. Процедурные типы

Процедурные типы допускают использование процедур и функций в виде значений, которые могут быть присвоены переменным или переданы в другие процедуры или функции. Например, в нижеследующем примере определена функция Calc с двумя целочисленными формальными параметрами X, Y, возвращающая целый тип:

**function** Calc(X,Y: Integer): Integer;

Эта функция может быть определена как тип для переменной F:

var F: function(X,Y: Integer): Integer;

и связана с этой переменной оператором присваивания:

F := Calc;

Точно так же можно определить любой другой новый процедурный тип и переменную:

Type {объявление процедурных типов}

TIntegerFunction = function: Integer;

TProcedure = procedure;

TStrProc = procedure(const S: string);

TMathFunc = function(X: Double): Double;

Var {объявление процедурных переменных}

F: TIntegerFunction; {F функция без параметров, возвращающая целое}

Proc: TProcedure; {Proc – процедура без параметров}

SP: TStrProc;

M: TMathFunc;

При использовании операторов над процедурными типами и процедурами или функциями необходимо различать связывание процедурной переменной и обращение к процедуре или функции. Так в нижеследующем примере объявляются переменная F типа функции, а переменная I – простого целочисленного типа, затем следует текст процедуры SomeFunction:

F: function(X: Integer): Integer;

I: Integer;

function SomeFunction(X: Integer): Integer;

...

В операторной части первый оператор связывает F с конкретной функцией, не производя над ней никаких действий:

F := SomeFunction;

напротив, оператор

I := F(4);

вызывает эту функцию (запускает ее алгоритм) и после обработки возвращает результат вычислений переменной I.

15.6. Формальные и фактические параметры

В Object Pascal есть понятия формального и фактического параметров. Формальным называется параметр, который содержится в заголовке описания подпрограммы, а фактическим – параметр в обращении к подпрограмме. Так, в вышеприведенном примере параметр X является формальным, а значение 4 – фактическим.

При вызове подпрограмм необходимо иметь в виду следующее:

фактические значения или константы должны быть совместимы по типу с объявленными формальными параметрами;

фактические var или out-параметры должны быть идентичны по типу объявленным формальным параметрам, исключением являются только нетипизированные параметры;

формальным параметрам без типа не могут соответствовать такие фактические параметры, как числовые значения и нетипизированные число-вые константы.

Приведем еще один пример, демонстрирующий способы обращения к подпрограммам.

Const

IntCount = 1200;

Type

TFunc12 = Function(c1, c2: Integer): Integer;

Function Func12\_1(k1, k2: Integer): Integer;

Begin

Result:= k1 + k2;

End;

Function Func12\_2(g1, g2: Integer): Integer;

Begin

Result:= g1 - g2;

End;

Procedure AnyPro(u1: Real; Var u2: Real; Var u3; Const u4: Integer; F: tFunc12);

Begin

u2:= u1 + u4 + F(u4, Round(u4 \* 3.14));

u3:= u1 - u4 - F(u4, Round(u4 \* 3.14));

End;

Var

k: Integer;

v1, v2: Real;

ss: String;

…

{примеры обращения к процедуре AnyPro:}

AnyPro(v1, v2, v1, v2, Func12\_1);

AnyPro(v1, v2, ss, v1, Func12\_2);

AnyPro(k, v1, ss, v2, Func12\_1);

AnyPro(k + 8, v2, ss, IntCount, Func12\_1);

AnyPro(8, v2, ss, v1+6.7, Func12\_2);

Параметры u1, u2, u3, u4, F в заголовке процедуры AnyPro, являются формальными параметрами: u1 – константа типа Real; u2 – переменная типа Real; u3 – переменная без типа; u4 – константа типа Integer; F – параметр-функция типа TFunc12, который объявлен выше в секции Type.

Параметры, заключенные в скобки в примерах обращения к процедуре AnyPro, являются фактическими параметрами. Такие параметры могут быть значениями (8), константами (IntCount), переменными (v1), выражениями (k + 8), именами процедур или функций (Func12\_1) и др.

15.7. Область действия имен

В подпрограммах часть параметров может быть объявлена прямо в заголовке или ее теле. Такие параметры действуют только внутри этой подпрограммы и поэтому называются локальными параметрами. В то же время в подпрограмме можно использовать параметры, которые описаны за пределами подпрограммы. Такие параметры называются глобальными параметрами.

Глобальные параметры могут быть описаны в том же модуле, который содержит использующую их подпрограмму, или в другом модуле, на который имеется ссылка в списке uses. Если два параметра имеют одинаковое имя и один из них описан внутри подпрограммы, а другой – вне ее, то действует тот параметр, который описан в подпрограмме. Аналогично определяется область доступности параметров описанных в разных модулях. Таким образом, при описании имен действует следующий принцип: более позднее объявление отменяет облаcть действия ранее описанных имен. Внутри одной подпрограммы нельзя объявлять двух и более одинаковых имен.

Поясним область действия имен на примере следующего модуля

Unit Mod4;

interface

uses Mod1, Mod2, Mod3;

….

Type

Vr = Integer; {допустимо}

…

Var

Vr: Real; {недопустимо}

…

implementation

Var Vr: Char; {недопустимо}

…

procedure Pro1; {не содержит внутреннего объявления имени Vr}

…

procedure Pro2; { содержит внутреннее объявление имени Vr}

Var

Vr: String; {допустимо}

Vr: Real; {недопустимо}

…

В приведенном тексте модуля Mod4 содержится несколько описаний имени Vr, часть которых допустима, другая часть ошибочна. Недопустимо описание этого имени:

в var-секции в разделе interface, так как оно уже использовано в этом же разделе выше – в секции type;

в var-переменной в разделе implementation, так как оно уже использовано в этом же модуле в разделе interface;

как переменной типа Real в теле процедуры Pro2, т. к. оно уже использовано в этой же процедуре при описании String-переменной.

Более позднее объявление отменяет действие ранее описанного имени. Так, внутри процедуры Pro2 имя Vr представляет переменную типа String, а внутри процедуры Pro1 имя Vr действует как глобальный тип Integer, объявленный выше – в секции type.

Если бы это имя вообще не было описано в модуле Mod4, но было бы объявлено в одном или нескольких модулях, указанных в ссылочном списке uses, то оно могло бы быть использовано как глобальный параметр внутри этого модуля (Mod4). При этом действовало бы то имя, которое объявлено в разделе interface самого последнего содержащего его модуля списка uses. Например, если имеется описание имени Vr в модулях Mod1 и Mod2, то действовало бы описание из Mod2. Если в списке uses поменять Mod1 и Mod2 местами, то будет действовать описание, которое выполнено для этого имени в модуле Mod1.

Следует проявлять особую осторожность при использовании глобальных переменных в подпрограммах. Нижеприведенный пример демонстрирует непредсказуемое поведение программы, использующей функцию Deccy и глобальный по отношению к ней параметр d:

Function Deccy(x: Integer) : Integer;

Begin

d:= d - x;

Deccy:= Sqr(x);

End;

…

d:= 3; a:= Deccy(3) \* Deccy(d); {a= 0, d= 0}

d:= 3; a:= Deccy(d) \* Deccy(3); {a= 81, d= -3}

Пример показывает, что два, казалось бы, корректных способа обраще-ния к функции дают тем не менее разные результаты вычислений.

15.8. Рекурсивные процедуры и функции

В Object Pascal допустимо обращение подпрограммы к самой себе (рекурсивное обращение). При таком обращении параметры, которые использует подпрограмма, заносятся в стек и сохраняются там до конца работы подпрограммы. Рекурсивные подпрограммы являются исключительно удобным, нередко незаменимым инструментом построения эффективных алгоритмов. Оборотной стороной рекурсивных процедур является опасность переполнения стека, что часто ограничивает возможность написания таких алгоритмов.

В качестве иллюстрации приведем пример простой и чрезвычайно эффективной процедуры сортировки (расстановки элементов в порядке неубывания) фрагмента целочисленного одномерного массива A:

procedure QuickSortPart(var A: array of Integer; iLo, iHi: Integer);

var

Lo, Hi, Mid, T: Integer;

begin

Lo := iLo;

Hi := iHi;

Mid := A[(Lo + Hi) div 2]; {средний элемент фрагмента}

repeat {деление фрагмента на левую и правую части}

while A[Lo] < Mid do Inc(Lo);

while A[Hi] > Mid do Dec(Hi);

if Lo <= Hi then

begin

T := A[Lo];

A[Lo] := A[Hi];

A[Hi] := T;

Inc(Lo);

Dec(Hi);

end;

until Lo > Hi;

if Hi > iLo then QuickSortPart(A, iLo, Hi); {сортировка левой части}

if Lo < iHi then QuickSortPart (A, Lo, iHi); {сортировка правой части}

end;

Процедура QuickSortPart сортирует фрагмент одномерного массива A, начинающийся индексом iLo и заканчивающийся индексом iHi. Процедура основана на методе половинного деления. В соответствии с этим методом сначала выбирается элемент, расположенный в середине сортируемого фрагмента, затем элементы меньшие его отправляются в левую часть фрагмента, прочие – в правую часть. Далее сортируются левая и правая части разделенного массива как отдельные фрагменты по той же схеме, т. е. к каждой из них применяется та же процедура QuickSortPart. Именно обращение процедуры к самой себе и делает ее рекурсивной.

Ниже приведена обычная (нерекурсивная) процедура QuickSort сортировки всех элементов массива, которая выполняется обращением к рекурсивной процедуре QuickSortPart, где фрагмент – весь массив A.

procedure QuickSort (var A: array of Integer);

begin

QuickSortPart(A, Low(A), High(A));

end;

15.9. Параметры и конструкторы открытых массивов

Открытые массивы допускают передачу массивов различного размера в качестве параметров в процедурах и функциях. В этом случае можно объявить массив в виде

array of type (предпочтительнее array[X .. Y] of type)

Например, операторы

procedure NullChar(A: array of Char);

begin

for i:= Low(A) to High (A) do A[i]:= '0';

end;

объявляют процедуру NullChar, которая содержит один параметр – открытый символьный массив А любого размера. В теле процедуры используется оператор цикла, который заполняет каждый элемент массива символом '0'. Для определения нижней границы индекса использована стандартная функция Low, для верхней – High.

Если к такой процедуре обратиться оператором NullChar(z), где тип переменной z = array[5 .. 55] of Char, то весь массив z будет заполнен символами "нуль".

Конструкторы открытых массивов допускают конструирование значений таких массивов прямо внутри оператора обращения к подпрограмме.

Пример:

var I, J: Integer;

**procedure** Add (A: array of Integer);

В этом случае можно обратиться к процедуре Add, например, так:

Add ([5, 7, I, I + J]);

16. Структура программы

В среде Delphi программа как единое целое представляется в виде проекта. В новой версии языка Object Pascal для представления проекта используется пять основных типов файлов:

dpr-файл головной программы;

текстовые pas-файлы;

откомпилированные dcu-файлы;

res-файлы ресурсов;

dfm-файлы ресурсов экранных форм;

готовые к использованию программные exe-файлы.

Исходная программа, написанная в среде Delphi на языке Object Pascal всегда состоит из нескольких модулей, каждый из которых размещается в отдельном текстовом файле. Один модуль является головной программой. Он начинается словом Program и размещается в файле с расширением .dpr. Все остальные модули являются подчиненными и начинаются словом Unit. Такие модули размещаются в файлах с расширением .pas. Все модули заканчиваются оператором End, после которого ставится символ "точка".

Всякий модуль может использовать другие модули, к числу которых могут относиться текстовые файлы, res- и dfm-файлы ресурсов или откомпилированные файлы Unit-модулей. Сcылка на необходимые к использованию модули содержится в секциях Uses. Текстовые или скомпилированные файлы обычно содержат необходимые для использующего их модуля величины – константы, типы, переменные, процедуры и функции. Файлы ресурсов необходимы для подключения констант, описывающих используемые внешние ресурсы.

Вышеперечисленные модули, размещенные в \*.pas-, \*.dcu-, \*.res-, \*.dfm-файлах, играют вспомогательную роль: они предназначены для компиляции и последующей сборки в полноценный программный модуль – exe-файл, готовый к исполнению на компьютере.

Ниже приведен пример исходных текстов головной программы KdnBread и одного подчиненного (используемого) ею модуля Main.

Program KdnBread; {начало текста головной программы}

{текст содержится в файле 'c:\Borland\Projects\KdnBread.pas'}

***uses*** {ссылки на модули типа unit }

Forms, {ссылка на модуль Forms }

main in 'main.pas' {Form1}; {ссылка на модуль main }

{$R \*.RES}

**begin**

Application.Initialize;

Application.CreateForm(TForm1, Form1);

Application.Run;

end. {конец текста головной программы}

unit Main; {начало текста модуля Main}

{ текст модуля содержится в файле 'c:\Borland\Projects\Main.pas' }

***interface*** {начало интерфейсной части модуля}

***uses***

Windows, Messages, SysUtils, {ссылки на другие модули }

Graphics, Controls, Forms, StdCtrls;

***Type*** {описание типов}

TForm1 = class(TForm)

Button1: TButton;

L1: TLabel;

**procedure** Button1Click(Sender: TObject);

**private**

{ Private declarations }

**public**

{ Public declarations }

**end;**

***Var*** {описание переменных}

Form1: TForm1;

b: boolean;

i: Integer;

IterationPar: Word;

**functio**n OneSymbString(c: Char; d: byte): String; {заголовок функции}

implementation {начало процедурного блока модуля}

{$R \*.DFM}

**procedure** TForm1.Button1Click(Sender: TObject); {заголовок процедуры}

**begin**

if (i > 12) and b then

L1.Caption:='Студент:'+AnsiUpperCase ('Иванов Владимир Иванович');

**end**; {конец процедуры}

**function** OneSymbString(c: Char; d: byte): String; {заголовок функции}

**begin**

Result:=CharStr(c, d);

**end;** {конец функции}

**initialization**

IterationPar:= 0;

end. {конец текста модуля Main}

Выполнение программы всегда начинается с модуля Program, т. е. с головной программы. Program активизирует выполнение процедур и функций в используемых ею модулях Unit.

16.1. Структура модуля

Модуль имеет следующую структуру:

Unit <имя>;

interface

<интерфейсная часть>

implementation

<выполняемая часть>

initialization

<блок инициирования>

finalization

<блок завершения>

end.

16.2. Раздел Interface

Раздел Interface модуля Unit предназначен для описания внешних компонент: используемых модулей, типов, констант, переменных, заголовков процедур и функций. Так, в вышеприведенном примере в разделе Interface содержатся:

в списке Uses – ссылки на модули Windows, Messages, SysUtils, Graphics, Controls, Forms, StdCtrls;

в секции Type – описание типа экранной формы – класс TForm1;

в секции Var – описание переменных Form1, b, i и описание заголов-ка функции OneSymbStr, предназначенной для создания строки повторяю-щихся d раз символов Ch.

16.3. Раздел Implementation

Раздел Implementation модуля Unit предназначен для описания внутренних, т.е. доступных к использованию только внутри данного Unit, компонент: типов, констант, переменных, процедур и функций. Так, в вышеприведенном примере в разделе Implementation содержится описание процедуры TForm1.Button1Click(Sender: TObject) и функции OneSymbStr.

16.4. Инициирование и завершение модуля

Всякий модуль может содержать блок инициирования и блок завершения. Эти блоки располагаются в нижней части модуля, непосредственно примыкая к последнему оператору end. Первый блок начинается словом initialization, второй – словом finalization.

Блок инициирования initialization заканчивается последним оператором end модуля либо, при наличии блока завершения, продолжается до слова finalization.

Обычно в блоке инициирования располагаются операторы определения начальных значений каких-либо переменных, выделения ресурсов памяти, открытия файлов и пр., т. е. все то, что необходимо инициализировать в модуле до передачи управления в использующие его модули.

Блок завершения может быть использован только в том случае, если модуль имеет блок инициирования. Этот блок, в противоположность блоку инициирования, предназначен для размещения операторов завершения, т. е. операторов освобождения ресурсов памяти, закрытия ранее открытых в блоке инициирования файлов и пр.

Например, модуль может заканчиваться следующими операторами:

…

Initialization {инициирование}

Ga:= 0;

AssignFile(f, 'c:\Projects\BreadPro\First.dat');

Reset(f, SizeOf(Rec1));

New(AppMem);

Finalization {завершение}

Dispose(AppMem);

CloseFile(f);

End. {последний оператор модуля}

Если несколько модулей имеют блоки инициирования, то они выполняются в том порядке, в котором имена модулей располагаются в списке Uses головной программы. Если несколько модулей содержат блоки завершения, то они выполняются в порядке, противоположном порядку перечисления модулей в списке uses головной программы.

17. Файлы

Файлом называется область данных на внешнем носителе – жестком диске, дискете и пр. Всякий файл имеет имя, представляющее собой строку символов. Различают обычное имя (или просто имя) и полное имя. Под полным именем понимается абсолютный адрес файла, состоящий из пути и имени файла. Например, строка 'C:\Program Files\Folder1\Students.dat' является полным именем. Оно состоит из пути 'C:\Program Files\Folder1' к файлу и собственно имени файла 'Students.dat'. Это означает, что файл 'Students.dat' расположен на диске C в папке (директории) Program Files непосредственно в подпапке (субдиректории) Folder1.

Ранее упоминалось, что в языке Object Pascal существует три типа файлов:

<имя> = TextFile; {текстовые файлы}

<имя> = File; {файлы без типа}

<имя> = File of <тип даных>; {файлы с типом}

17.1. Файловая переменная

Для того чтобы получить доступ к файлу, его необходимо сначала открыть. Открытие файла выполняется посредством связывания файла с особой переменной, называемой файловой переменной. Именно файловая переменная и характеризует тип файла. Связывание файла с файловой переменной еще не означает открытия этого файла. Открытие файла производится специальными процедурами, о которых будет упомянуто ниже.

Связывание файла с файловой переменной производится с помощью стандартной процедуры AssignFile, которая имеет заголовок:

AssignFile(<файловая переменная >, <имя файла>);

Например, фрагмент

Var

f1: TextFile;

FullPath: String[60];

…

FullPath:= 'a:\a1.res';

AssignFile(f1, FullPath);

cодержит объявление файловой переменной f1 текстового типа и строки FullPath, которые затем используются в исполнительной части для указания полного имени файла и связывания его с файловой переменной f1.

17.2. Текстовые файлы

Текстовой файл – это последовательность символьных строк перемен-ной длины. Всякая строка завершается маркером конца строки, всякий текстовой файл завершается маркером конца файла. Такие файлы можно обрабатывать только последовательно. Один и тот же текстовой файл не может быть открыт одновременно для ввода и вывода. Файловая переменная этого файла имеет тип TextFile или просто Text.

Для текстовых файлов есть две стандартные файловые переменные – Input и Output, которые не нужно объявлять отдельно.

17.2.1. Процедуры и функции для работы с текстовым файлом

**Procedure** AssignFile(f: TextFile; FileName: String);

Связывает файловую переменную f с дисковым файлом FileName.

**Procedure** Append(f: TextFile);

Открывает существующий файл для добавления строк в конец файла. При отсутствии файла возникает ошибка ввода/вывода.

**Procedure** Rewrite(f: TextFile);

Создает новый файл и открывает его для вывода. Если файл существует, то он уничтожается и создается как новый. Когда новый текстовой файл закрывается, к нему автоматически добавляется маркер конца файла.

**Procedure** Reset(f: TextFile);

Открывает существующий файл для чтения и устанавливает указатель на первую строку файла. При его отсутствии возникает ошибка ввода/вывода.

**Procedure** Read( f: TextFile[; v1, v2, …,vN]);

Читает данные из файла и заносит их в переменные v1, v2, …, vN. Переменные могут иметь символьный, строчный или арифметические типы.

**Procedure** Readln( f: TextFile[; v1, v2, …,vN]);

Читает данные из файла целыми строками и заносит их в переменные v1, v2, …, vN. Если список переменных пуст, то происходит перемещение указателя на следующую строку.

**Procedure** Write( f: TextFile[; v1, v2, …,vN]);

Записывает данные из переменных v1, v2, …, vN в файл в символьном виде.

**Procedure** SetTextBuf ( f: TextFile; Var Buf[; Size: Integer]);

Устанавливает буфер чтения текстового файла. Процедура должна быть вызвана после AssignFile, но до первого вызова процедур чтения. Буфер используется для чтения больших фрагментов файла, включая символы конца строк. Если размер буфера не указан, то по умолчанию он принимается равным 128.

**Procedure** CloseFile( f: TextFile);

Закрывает текстовой файл.

**Procedure** Flush( f: TextFile);

Выводит содержимое внутреннего буфера в файл.

**Function**Eof( f: TextFile): boolean;

Возвращает True, если достигнут конец файла.

**Function** Eoln( f: TextFile): boolean;

Возвращает True, если достигнут конец текущей строки.

**Function** SeekEof( f: TextFile): boolean;

Возвращает статус конца файла.

**Function** SeekEoln( f: TextFile): boolean;

Возвращает статус конца строки.

Пример:

Var

F1, F2: TextFile;

Ch: Char;

St: String[255];

Buf: array[1..4096] of Char; { текстовой буфер размером 4K}

begin

AssignFile(F1, 'T1.TXT');

SetTextBuf(F1, Buf); { большой буфер для ускорения чтения}

Reset(F1); {F1 открыт для чтения}

AssignFile(F2, 'WOOF.DOG');

Rewrite(F2); {F2 создан как новый для вывода }

while not Eof(F1) do {пока не достигнут конец файла – выполнять}

begin

Read(F1, Ch); {читает один символ из файла F1}

Write(F2, Ch); {пишет один символ в файл F2}

end;

CloseFile(F1); {файл F1 закрыт}

CloseFile(F2); {файл F2 закрыт}

Reset(F1); {F1 снова открыт для чтения}

Rewrite(F2); {F2 снова создан для вывода }

while not Eof(F1) do {пока не достигнут конец файла – выполнять}

begin

Readln(F1, St); {читает строку из файла F1}

Write(F2, St); {пишет строку в файл F2}

end;

CloseFile(F1); {файл F1 закрыт}

CloseFile(F2); {файл F2 закрыт}

end;

Приведенный фрагмент модуля является демонстрационным и предназначен для копирования файла 'T1.TXT' в файл 'WOOF.DOG' . В первом цикле While – do копирование ведется посимвольно, во втором цикле – построчно.

Пример процедуры, записывающей в конец текстового файла строку символов:

Procedure AddStrToTextFile(nF, St:String);

***Var*** f: Text;

Begin

AssignFile(f, nF);

If not FileExists(nF) then Rewrite(f) {не существует, создать и открыть}

Else {иначе}

Begin

Reset(f); {существует, открыть }

***While*** ***not*** Eof(f) ***do*** Readln(f); {передвинуть указатель в конец файла}

End;

Writeln(f, St); {записать строку }

CloseFile(f); {закрыть файл}

End;

К процедуре можно обратиться, например, так:

**Var**

S1: String[58];

S2: String[189];

…

AddStrToTextFile('c:\Files\ring.txt', 'Строка символов');

AddStrToTextFile('ring.txt', S1);

AddStrToTextFile('ring.txt', S2);

17.3. Файлы с типом

Файл состоит из любых однотипных компонент. Доступ к данным осуществляется через файловую переменную. В отличие от текстового файла в таком файле допустим прямой доступ к любой записи, причем в рамках открытого файла допустимо как записывать, так и читать записи.

Примеры объявления файловой переменной для файлов с типом:

**Var**

F1: File of String[45];

F2: File of Real;

F3: File of tRecord24;

После каждого чтения или вывода записи указатель автоматически устанавливается на следующую запись.

17.3.1. Процедуры и функции для работы с типизированным файлом

**Procedure** AssignFile( f: File of Type; FileName: String);

Связывает файловую переменную f с дисковым файлом FileName.

**Procedure** Rewrite( f: File of Type);

Создает новый файл и открывает его. Если файл существует, то он уничтожается и создается как новый.

**Procedure** Reset( f: File of Type);

Открывает существующий файл и устанавливает указатель на первую запись. При отсутствии файла возникает ошибка ввода/вывода.

Procedure Read( f: File of Type[; v1, v2, …,vN]);

Читает записи из файла и заносит их в переменные v1, v2, …, vN. Чтение начинается с той записи, на которую установлен указатель. Типы файла и переменных должны быть одинаковы.

**Procedure** Write( f: File of Type[; v1, v2, …,vN])***;***

Записывает данные из переменных v1, v2, …, vN в файл. Вывод данных начинается с той записи, на которую установлен указатель. Если указатель установлен на существующую запись, то при выводе она будет замещена новой записью. Если одновременно выводится несколько записей, то будет замещено такое же количество существующих записей. Типы файла и переменных должны быть одинаковы.

**Procedure** Seek( f: File of Type; N: LongInt);

Перемещает указатель на запись с номером N. Первая запись имеет порядковый номер 0.

**Function** FilePos( f: File of Type): LongInt;

Возвращает номер записи, на которую установлен указатель.

**Procedure** CloseFile( f: File of Type);

Закрывает файл.

**Function** Eof(f: File of Type): boolean;

Возвращает True, если достигнут конец файла.

**Function** FileSize(f: File of Type): LongInt;

Возвращает количество записей в файле. Например, Seek(f, FileSize(f)) установит указатель в конец файла (после последней записи).

**Procedure** Truncate(f: File of Type);

Уничтожает (отрубает) конец файла начиная с записи, на которую установлен указатель.

17.4. Файлы без типа

Файл состоит из компонент одинакового размера. Тип данных не имеет значения. Доступ к данным осуществляется через файловую переменную. Как и в файлах с типом, в таком файле допустим прямой доступ к любой записи, причем в рамках открытого файла допустимо как писать, так и читать записи.

Файловая переменная может быть объявлена так:

Var F: File;

После каждого чтения или вывода записи указатель автоматически устанавливается на следующую запись.

Отсутствие типа записи позволяет выполнять обработку файлов различных типов с помощью универсальных процедур и функций.

17.4.1. Процедуры и функции для работы с файлом без типа

**Procedure** AssignFile( f: File; FileName: String);

Связывает файловую переменную f с дисковым файлом FileName.

**Procedure** Rewrite( f: File);

Создает новый файл и открывает его. Если файл существует, то он уничтожается и создается как новый.

**Procedure** Reset( f: File[; Size: Word]);

Открывает существующий файл и устанавливает указатель на первую запись. При отсутствии файла возникает ошибка ввода/вывода. Параметр Size указывает размер записи открываемого файла. При его отсутствии размер записи по умолчанию равен 1.

**Procedure** BlockRead( f: File; Var Buf; Count: Word[; Var Result: Word]);

Читает из файла Count записей в переменную Buf. Result – реально прочитанное количество записей.

**Procedure** BlockWrite( f: File; Var Buf; Count: Word[; Var Result: Word]);

Пишет в файл первых Count записей из переменной Buf. Result – реально записанное количество записей.

**Procedure** Seek( f: File; N: LongInt);

Перемещает указатель на запись с номером N. Первая запись имеет порядковый номер 0.

**Functio**n FilePos( f: File): LongInt;

Возвращает номер записи, на которую установлен указатель.

**Procedure** CloseFile( f: File);

Закрывает файл.

**Function** Eof(f: File): boolean;

Возвращает True, если достигнут конец файла.

**Function** FileSize(f: File): LongInt;

Возвращает количество записей в файле. Например, Seek(f, FileSize(f)) установит указатель в конец файла (после последней записи).

**Procedure** Truncate(f: File of Type);

Уничтожает (отрубает) конец файла начиная с записи, на которую установлен указатель.

Язык Object Pascal не накладывает никаких ограничений на длину записи (теоретически она может иметь размер до 2 Гб).

Пример описания и обращения к функции ReadFromFile, читающей из файла nF в позиции Pos запись r размером Sz.

function ReadFromFile(nF: String; Pos: Word; Var r; Sz: Word): boolean;

**Var**

g: File;

Recs, ReadReal: Integer;

RecRead: boolean;

Begin

Assign(g, nF);

Recs:= FileSize(g) div Sz; {количество записей в файле}

RecRead:= (Pos < Recs); {запись с номером Pos есть ?}

if RecRead then begin {если запись есть}

Reset(g, Sz); {открыть файл}

try

Seek(g, Pos); {установить указатель на запись}

BlockRead(g, r, 1, ReadReal); {прочитать запись}

RecRead:= (ReadReal = 1); {прочитано успешно ?}

finally

Close(g); {закрыть файл}

end;

end;

Result:= RecRead;

end {ReadFromFile};

…

Type

tStud = **Record**

Fio: String [60];

Curs: byte;

Stipendiya, Room: boolean;

**End;**

Var Stud: tStud;

…

**if** ReadFromFile('base2.ff1', 12, Stud, SizeOf(Stud))

**then** Writeln('Запись из 12-й позиции прочитана');

Приведем еще пример. В директории 'c:\Bases\SdudBase' находится файл 'AllStuds.bs', в котором хранятся данные о студентах в виде записей типа

Type

TStud = **Record** {студент}

Fio: String[50]; {'Фамилия Имя Отчество'}

Born: byte; {Год рождения, например, 1979}

Faculty: String[4]; {Факультет, например, 'МТФ'}

Group: String[8]; {Группа, например, 'МТ 17-2'}

End;

Ниже приведена универсальная процедура, которая копирует из этого файла в другой файл данные только о тех студентах, которые имеют заданный год рождения:

Procedure StudsCopy(nF1, nF2: ShortString; BornYear: byte;

**Var** Count: Word; **Var**: Ind: ShortInt);

{nF1 – файл-источник, nF2 – файл-приёмник,

BornYear – требуемый год рождения,

Count – скопировано записей,

Ind – индикатор контроля:

0 – нормально, 1 – было неверное чтение, была неверная запись}

Var

g: tStud;

K, Sz, i,j: Word;

f1, f2: File;

Begin

Count:= 0; {инициализация счетчика}

Ind:=0; {изначально предполагаем нормальный процесс, иначе Ind изменим}

Sz:= SizeOf(g); {размер одной записи}

K:= KdnFileSize(nF1, Sz); {количество записей в файле-источнике}

If (K > 0) then {если в файле-источнике есть записи }

Begin

Assign(f1, nF1); {файл-источник связываем переменной f1}

Reset(f,Sz); {открываем файл-источник с записями размера Sz}

Assign(f2, nF2); {файл-приёмник связываем переменной f2 }

Rewrite(f2,Sz); {создаем новый файл-приёмник под записи размера Sz}

try

For j:=1 to K do

Begin

BlockRead(f1, g, 1, i); {чтение записи}

Case i of

1: {запись прочитана}

if (g.Born = BornYear) then { студент имеет требуемый год рождения}

begin

BlockWrite(f2, g, 1, i); {запись в файл-приёмник}

If (i > 0) then Inc(Count) {если записано правильно}

else

begin Ind:= 1; Break; End; {записано неверно, сразу выход из цикла}

end; {if}

0: begin Ind:= -1; Break; end; {запись не прочитана, сразу выход из цикла}

end; {Case}

end; {цикла For}

finally

CloseFile(f1); {закрываем файл-источник}

CloseFile(f2); {закрываем файл-приёмник}

end; {блока try – finally – end}

End {If };

End {StudsCopy};

Операторы, реализующие копирование требуемых данных в файл '1979.bs':

StudsCopy ('AllStuds.bs', '1979.bs', 1979, Count1979, Ind1979);

**Case** Ind1979 **of**

-1: Writeln('Зафиксирована ошибка чтения');

1: Writeln('Зафиксирована ошибка записи');

0: Writeln('Процесс прошел нормально');

**end;** {Case}

Writeln('Скопировано записей: ' + IntToStr(Count1979));

В этом примере использована внешняя процедура KdnFileSize {количество записей в файле }. Приведем ее текст:

function KdnFileSize(nF: ShortString, Siz: Word): LongInt;

{nF – имя файла, Siz – размер одной записи }

Var

F: File;

L: LongInt;

Begin

L:=0;

If FileExists(nF) then

begin

Assign(f, nF);

Reset(f,1);

L:= SizeOf(f);

If not (L mod Siz = 0) then Writeln('Файл ' + nF + имеет другой тип');

L:= L div Siz;

CloseFile(f);

End;

Result:= L;

End;

17.5. Процедуры и функции для работы с файлами

Эти подпрограммы предназначены для работы с файлами, папками (директориями) и дисками.

**Procedure** ChDir(Dir: String);

Делает папку Dir текущей. Пример: ChDir('c:\');

**Procedure** GetDir(D: Byte; Var Dir: String);

Возвращает текущую папку на заданном устройстве. (D= 0 – текущий диск, 1 – диск А, 2 – диск B и т.д.). Пример: GetDir(0, s);

**Procedure** RmDir(Dir: String);

Уничтожает заданную папку. Папка не должна содержать вложенных папок или файлов. Пример: RmDir('Folder66');

**Procedure** Erase(f);

Удаляет файл, связанный с файловой переменной f. Файл должен быть закрыт.

**Procedure** Rename(f, FileName: String);

Переименовывает файл, связанный с файловой переменной f. Файл должен быть закрыт. Пример: Rename(g, 'studs.txt');

**Function** DiskFree(D: byte): LongInt;

Возвращает количество свободной памяти в байтах на устройстве D. Код драйвера задается так же, как в процедуре GetDir. Если код указан неверно, то возвращает -1.

**Function** DiskSize(D: byte): LongInt;

Возвращает количество свободной памяти в байтах на устройстве D. Код драйвера задается так же, как в процедуре GetDir. Если код указан неверно, то возвращает -1.

**Function**FindFirst(const Path: string; Attr: Integer;

var F: TSearchRec): Integer;

Находит имя первого файла с заданными атрибутами Attr в папке Path. Результат поиска выводит в переменную F. Если поиск успешен, то функция вернет 0, иначе вернет код ошибки Widows. К FindFirst можно обращаться не только как к функции, но и как к процедуре.

Атрибуты файла приведены в табл. 17.

Таблица 17

|  |  |
| --- | --- |
| Атрибут | Описание файлов |
| faReadOnly  faHidden  faSysFile  faVolumeID  faDirectory  faArchive  faAnyFile | Файлы "Только для чтения"  Скрытые файлы  Системные файлы  Файл ID-значений  Папки (директории)  Архивы (файлы)  Все файлы |

Тип, характеризующий найденный файл, представляет запись вида :

type

TSearchRec = Record

Time: Integer; {время}

Size: Integer; {размер файла в байтах}

Attr: Integer; {атрибуты файла}

Name: TFileName; {DOS-путь файла}

ExcludeAttr: Integer;

FindHandle: THandle;

FindData: TWin32FindData; {дополнительная информация о файле}

end;

Пример:

Var

SR: TSearchRec;

S: String;

…

FindFirst('c:\Program Files\delphi4\bin\\*.\*', faAnyFile, SR);

if (SR.Attr = faArchive) then

S:= 'Файл ' + SR.Name + ' имеет размер ' + IntToStr(SR.Size) + ' байт';

В данном примере процедура FindFirst ищет первый файл по маске '\*.\*' (все файлы) в папке 'c:\Program Files\delphi4\bin'. Атрибут faAnyFile означает, что поиск производится по всем видам файлов, под которыми понимаются папки (директории), '.', '..' – ссылки на текущую и родительскую папку, внутренние папки и собственно файлы. Последние в терминологии файловой атрибутики называются архивами. Далее, если найденный файл есть архив, т е. файл в общепринятой терминологии, то в строку S будет помещено сообщение. Например, если найденный файл имеет имя Ig.ttg и его размер равен 15899, то S= 'Файл Ig.ttg имеет размер 15889 байтов'.

**Function** FindNext(var F: TSearchRec): Integer;

Находит следующий файл, атрибуты которого указаны в FindFirst.

**Procedure** FindClose(var F: TSearchRec);

Закрывает действие FindFirst/FindNext.

**Function** DeleteFile(const FileName: string): Boolean;

Удаляет файл по имени. Если файл не может быть удален или не существует – возвращает False.

**Function** CreateDir(const Dir: string): Boolean;

Создает новую папку.

**Function**GetCurrentDir: string;

Возвращает текущую папку.

**Function** GetCurrentDir: string;

Возвращает текущую папку.

**Function** SetCurrentDir(const Dir: string): Boolean;

Установка новой текущей папки.

**Function** RemoveDir(const Dir: string): Boolean;

Удаление папки. Перед удалением папка должна быть пустой.

**Function** ExtractFileDir(const FileName: string): string;

Выделяет из полного имени файла FileName папку, в которой содержится это файл.

**Function** ExtractFilePath(const FileName: string): string;

Выделяет из полного имени файла FileName путь до файла.

**Function** ExtractFileExt(const FileName: string): string;

Возвращает расширение файла FileName.

**Function** ExtractFileName(const FileName: string): string;

Возвращает имя файла FileName (без расширения).

**Function** DirectoryExists(Dir: string): boolean;

Проверяет существование директории. Пример:

if DirectoryExists('C:\APPS\SALES\LOCAL') then ;

**Function** FileExists(FileName: string): boolean;

Проверяет существование файла. Примеры:

B:= FileExists('C:\APPS\SALES\LOCAL\Fort.pas'); {полное имя}

B:= FileExists('Fort.pas'); {указано усеченное имя файла, проверка его существования только в текущей директории}

**Procedure** ForceDirectories(Dir: string);

Создает новую директорию.

**Procedure** ForceDirectories(C:\APPS\SALES\LOCAL).

П р и м е ч а н и е. К моменту обращения к процедуре директории APPS и SALES должны существовать.

Пример процедуры удаления данных из текущей директории, включая файлы и вложенные папки.

Procedure DelInsideDir(FullDir: tPathStr);

Var

L: Integer;

Sr: TSearchRec;

dr, q: tPathStr;

begin

if ExistDir(FullDir) then {такая директория есть}

begin

GetDir(0,dr); {запомнить текущую директорию}

ChDir(FullDir); {текущей становится удаляемая директория}

L:=FindFirst(Slash(FullDir)+'\*.\*',faAnyFile,Sr);{поиск первого файла}

try

While (L = 0) do begin {пока файлы находятся}

Case Sr.Attr of

faDirectory:{найденный файл – внутренняя директория}

if (Sr.Name<>'.') and (Sr.Name<>'..') then {это не ссылка, директория}

begin

{удаление внутреннего содержимого директории}

DelInsideDir(Slash(FullDir)+Sr.Name);

q:= Slash(FullDir)+Sr.Name;

ChDir(ExtractFilePath(q));

{удаление самой директории (можно, т. к. она теперь пуста)}

if NotEmpStr(ExtractFileName(q)) then RmDir(ExtractFileName(q));

end;

faArchive: DeleteFile(Sr.Name); {это файл, удаляется}

end; {Конец Case-оператора}

L:= FindNext(Sr); {следующий файл директории}

end; {цикла While}

finally

FindClose(Sr); {закрыть поиск файлов}

end; {try – finally – end}

ChDir(dr); {вернуться в текущую директорию}

end; {if}

end;{процедуры}

Например, если необходимо стереть данные с дискеты, то это можно сделать с помощью оператора: DelInsideDir('A:\');

18. Классы и объекты

В Object Pascal классами называются специальные типы, которые содержат поля, методы и свойства. Предшественником класса является устаревший ныне тип языка Turbo Pascal, называемый объектом. Объект был введен в Turbo Pascal до создания Delphi. С появлением Delphi в новой версии языка Object Pascal объекты, для совместимости со старым программным продуктом, сохранены. Однако ныне использование объектов не актуально.

Класс представляет собой указатель. Однако в отличие от традиционных указателей это указатель особого типа: в нем нельзя использовать символ "^" при обращении к классу.

18.1. Инкаспуляция, наследование и полиморфизм

Класс, объединяя в себе поля, методы и свойства в единое целое, является законченной структурной единицей, предназначенной для решения отдельной задачи. Обычно такой задачей является задача разрешения некоторого круга сопутствующих проблем. Так, класс TRichEdit представляет собой мощный текстовой редактор rtf-файлов (файлов в формате Rich Text Format), который предназначен для организации просмотра и редактирования файла, сохранения и изменения размеров и типов шрифтов, поиска строк символов и многого другого. Такое объединение полей, методов и свойств в единое целое называется инкаспуляцией.

В языке существует множество классов (около 300), которые созданы разработчиками языка Object Pascal – сотрудниками фирмы Inprise International – для программистов, использующих среду Delphi. Такие классы можно назвать фирменными.

Программист, составляя программу, всегда создает свои пользовательские классы. Эти классы создаются либо неявно, когда программист конструирует программу визуальными средствами Delphi, а текст классов при этом составляет сама Delphi, либо явно, когда программист пишет код класса средствами языка Object Pascal.

Новый класс строится на основе другого, более простого, класса. Для этого в заголовке класса указывается его класс-родитель. Синтаксис заголовка нового класса имеет вид

type className = class (ancestorClass)

Здесь className – имя нового класса; ancestorClass – имя класса-родителя. Новый класс автоматически наследует поля, методы и свойства своего родителя и может пополниться своими полями, методами и свойствами. Это свойство классов называется наследованием. Возможность наследования позволяет, следуя методу от простого к сложному, создавать классы какой угодно степени сложности. Простейшим классом является класс TObject, который не содержит полей и свойств, однако имеет некоторое множество методов, обеспечивающих создание, уничтожение и обслуживание этого класса и необходимых для нормального функционирования программы. Именно на основе этого общего для всех классов прародителя строится дерево наследования классов. Например:

type TPersistent = class (TObject),

type TComponent = class (TPersistent),

type TControl = class (TComponent).

Нередко методы, описанные в классе-родителе, оказываются по каким-либо причинам неудовлетворительными для класса-потомка. В этом случае в классе-потомке можно создать метод с тем же именем, что и в классе-родителе. При этом окажется, что в обоих классах будут действовать разные методы с одним и тем же именем. Полиморфизм и есть такое свойство родственных классов, которое состоит в допустимости объявления в них одноименных методов.

18.2. Синтаксис класса

Синтаксис всякого класса имеет вид

***type*** className = class (ancestorClass)

memberList

***end*;**

Здесь className – имя класса; class – ключевое слово; ancestorClass – тип класса-родителя; memberList – список полей, методов и свойств. Ниже приведен текст модуля main, содержащий класс TForm1.

unit main;

interface

uses

Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms;

type

TForm1 = class(TForm) {объявление класса TForm1}

Button1: TButton; {поле}

L1: TLabel; {поле}

L2: TLabel; {поле}

Button2: TButton; {поле}

procedure Button1Click(Sender: TObject); {метод}

procedure FormActivate(Sender: TObject); {метод}

end;

Var i: Integer;

implementation

{$R \*.DFM}

procedure TForm1.Button1Click(Sender: TObject); {описание метода}

begin

L1.Caption:= DateTimeToStr(Date);

L2.Caption:= TimeToStr(Time);

end;

procedure TForm1.FormActivate(Sender: TObject); {описание метода}

begin

i:=125;

end;

end.

18.3. Поля класса

Полем может быть любой инкаспулированный в класс тип или другой класс, например:

type

TKdnClass = class(TObject)

i, j: integer;

s: String;

TKdn1: TKdn0;

End;

Если потомком является TObject, то в заголовке его можно опустить.

Класс-потомок имеет доступ ко всем полям своих предков, но не может их переопределять, т. к. он станет недоступен. Пример:

type

TPredok = class {объявление класса-предка}

Value: Integer;

end;

TPotomok = class(TPredok) {объявление класса-потомка}

Value: string; {перекрытие наследуемого поля}

end;

var

My1: TPredok; {объявление переменной класса}

My2: TPotomok; {объявление переменной-класса}

begin

My1 := TPotomok.Create; {создает класс типа TPredok !}

My2 := TPotomok.Create; {создает класс типа TPotomok}

My1.Value := 'Hello!'; {ошибка, не тот тип поля TPredok}

My2.Value := 'Hello!'; {правильно, работает поле Value: String}

My2.Value := 8; {ошибка: поле Value: Integer перекрыто}

end;

В этом примере описано два класса: TPredok – предок и TPotomok – потомок. Каждый из классов содержит одноименные поля Value разных типов.

Далее в var-секции объявлены две различные переменные My1 и My2 типа class. На первый взгляд может показаться, что оператор-конструктор объекта My1:= TPotomok.Create создаст объект My1 (выделит под него память) типа TPotomok. Однако это не так, поскольку My1 имеет другой тип. По этой причине конструктор создаст объект родительского типа, т. е. объект типа TPredok. Теперь становится понятен источник ошибок, которые имеют место в нескольких операторах приведенного примера.

18.4. Методы класса

Методом класса является инкаспулированная процедура или функция. Эти подрограммы объявляются так же, как обычные подпрограммы. Метод должен быть объявлен в описании класса в виде отдельного заголовка, а код метода – описан в секции implementation с указанием через символ "." принадлежности метода к своему классу, например:

type

TMyClass = class(TObject){объявление класса}

...

procedure DoSomething; {объявление метода DoSomething}

...

end;

Описание для DoSomething должно быть приведено позже в секции implementation модуля:

procedure TMyClass.DoSomething;{вид заголовка класс.метод}

begin

...

end;

При обращении к методу возможно использование составного имени либо оператора With, например:

Var KdnClass: TKdnClass;

…

KdnClass.MyProc1; // два примера обращения к методам

X:= KdnClass.MyFunc2; // с помощью составных имен

…

With KdnClass do // те же обращения

Begin // с помощью оператора With

MyProc1;

X:=MyFunc2;

End;

Одноименные методы могут перекрываться в потомках точно так, как это показано в примере перекрытия полей. Такое перекрытие называется статическим.

Для расширения возможностей чаще используется динамическое перекрытие. Для этого родительский метод должен иметь директиву dinamic (динамический метод) или virtual (виртуальный метод), а перекрывающий метод – директиву override. Пример:

type

TFigure = class

procedure Draw; virtual; {виртуальный метод}

end;

TRectangle = class(TFigure)

procedure Draw; override; {перекрывающий метод}

end;

TEllipse = class(TFigure)

procedure Draw; override; {перекрывающий метод}

end;

В этом примере объявлен виртуальный метод Draw родительского класса TFigure и два одноименных метода в классах-потомках TRectangle и TEllipse. Последние объявлены перекрывающими (override).

Такое объявление позволяет перекрывать методы с целью достижения нужных целей:

var

Figure: TFigure;

begin

Figure := TRectangle.Create; //создание класса

Figure.Draw; // вызов TRectangle.Draw

Figure.Destroy; // уничтожение класса

Figure := TEllipse.Create; //создание класса

Figure.Draw; // вызов TEllipse.Draw

Figure.Destroy; // уничтожение класса

end;

Семантически виртуальный и динамический методы работают одинаково. Разница состоит в том, что виртуальный метод оптимизирует скорость вычислений, а динамический метод оптимизирует размер соответствующего программного кода.

В классе метод может быть объявлен абстрактным с помощью директивы adstract. Такой метод является виртуальным или динамическим, однако, в отличие от других методов, может не иметь в секции implementation своего кода. Класс, имеющий абстрактные методы, называется абстрактным. Такие классы и методы могут ничего не делать, инкаспулируя таким способом доступ к методам потомков, например:

procedure DoSomething; virtual; abstract;

Обращение к неперекрываемому абстрактному методу вызывает ошибку времени выполнения (run time error), например:

Type

TClass2 = class(TClass0)

…

procedure Paint; virtual; abstract;

end;

TClass1 = class(TClass0)

…

procedure Paint; override;

end;

var

jClass1: TClass1;

jClass2: TClass2;

begin

jClass1.Paint; // правильно

jClass2.Paint; // неправильно: обращение к абстрактному методу

…

end;

Каждый класс имеет два особых метода – конструктор и деструктор. Конструктор предназначен для создания класса, т. е. для выделения под него динамической памяти. Деструктор, наоборот, предназначен для уничтожения класса, т. е. для освобождения участка памяти, занятого этим классом. В классе TObject имеются стандартные методы Create (создать) и Destroy (уничтожить). В этом классе объявлен также метод Free, который сначала проверяет корректность адреса и только потом вызывает метод Destroy. В этой связи предпочтительнее использовать метод Free вместо метода Destroy. Всякий класс по умолчанию содержит переменную Self, в которую после выделения динамической памяти помещается адрес класса. Прежде чем выполнить обращение к методам класса, его нужно создать. Хотя конструктор и деструктор являются процедурами, они объявляются специальными словами. Конструктор объявляется словом Constructor, деструктор – словом Destructor. Часто для обеспечения доступа к полям предка в конструкторе необходимо предварительно создать класс-предок. Это можно сделать c помощью слова Inherited.

Пример:

type

TShape = class(TGraphicControl)

Private {внутренние объявления}

FPen: TPen;

FBrush: TBrush;

procedure PenChanged(Sender: TObject);

procedure BrushChanged(Sender: TObject);

public {внешние объявления}

constructor Create(Owner: TComponent); override;

destructor Destroy; override;

...

end;

constructor TShape.Create(Owner: TComponent);

begin

inherited Create(Owner); // создание класса-предка TGraphicControl

Width := 65; // изменение наследуемых свойств TGraphicControl

Height := 65;

FPen := TPen.Create; // создание отдельного поля TPen типа class

FPen.OnChange := PenChanged;

FBrush := TBrush.Create; // создание отдельного поля TBrush типа class

FBrush.OnChange := BrushChanged;

end;

Некоторые простые классы могут быть созданы и уничтожены без объявления конструкторов и деструкторов. Например, если класс является потомком TObject, то в нем явно Constructor и Destructor в некоторых случаях объявлять нет нужды:

Type TClassy = class;

..

var Classy: TClassy;

…

Classy:= TClassy.Create; {создание класса}

…

Classy:= TClassy.Free; {уничтожение класса}

В языке имеется возможность объявлять в пределах одного класса несколько методов с одним и тем же именем. При этом всякий такой метод должен быть перезагружаемым (директива overload). Компилятор такие методы идентифицирует по своим уникальным наборам формальных параметров. Для того чтобы отменить реакцию компилятора Delphi на появление метода с тем же именем, каждый такой метод нужно пометить директивой reintroduce. Далее в секции implementation необходимо привести коды всех таких методов.

Пример:

Type TClassy = class;

Procedure HH(i, j: byte; var s: String); reintroduce; overload;

Procedure HH(q: String); reintroduce; overload;

Procedure HH(a: array oh Integer); reintroduce; overload;

…

implementation

…

Procedure TClassy.HH(i, j: byte; var s: String);

**Begin**

S:=IntToStr(i + j);

End;

Procedure TClassy.HH(q: String);

**Begin**

L2.Cattion:= q;

End;

Procedure TClassy.HH(a: array oh Integer);

Begin

L1.Cattion:= IntToStr(a[6] + a[4]);

End;

…

Теперь, после обращения к методу по имени TClassy.HH, программа вызовет именно тот метод, формальные параметры которого соответствуют фактическим параметрам в обращении.

18.5. Свойства класса

Свойства, как и поля, являются атрибутами класса. Свойства объявляются с помощью слов property, read и write. Слова read и write конкретизируют назначение свойства. Синтаксис свойства таков:

property propertyName[indexes]: type index integerConstant specifiers;

где propertyName – имя свойства; [indexes] – параметры-имена в форме имя1, имя2, ... , имяN: type; index – целая константа; read, write, stored, default (или nodefault) и implements – спецификации. Всякое объявление свойства должно иметь одну из спецификаций read или write или обе вместе.

Примеры:

property Objects[Index: Integer]: TObject read GetObject write SetObject;

property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;

property Values[const Name: string]: string read GetValue write SetValue;

property ErrorCount: Integer read GetErrorCount;

property NativeError: Longint read FNativeError;

Неиндексированные свойства похожи на обычные поля, а индексированные свойства напоминают поля-массивы. В программе свойства ведут себя почти так же, как обычные поля. Разница в том, что свойство имеет более ответственное назначение. Например, оно может активизировать некоторые методы для придания объектам требуемого свойства. Так если изменено свойство шрифта какого-либо визуального класса, то смена свойства шрифта повлечет за собой перерисовку текста и выполнение ряда сопутствующих операций, которые обеспечат классу именно такое свойство.

Каждое свойство может иметь спецификацию read или write или оба вместе в форме

read fieldOrMethod

write fieldOrMethod

где fieldOrMethod – имя поля или метода, объявленного в классе, или свойство класса-предка.

Если fieldOrMethod объявлено в классе, то оно должно быть определено в том же классе. Если оно объявлено в классе-предке, то оно должно быть видимо из потомка, т. е. не должно быть частным полем или методом класса-предка. Если свойство есть поле, то оно должно иметь тип. Если fieldOrMethod есть read-спецификация, то оно должно быть функцией без параметров, тип которой совпадает с типом свойства. Если fieldOrMethod есть write-спецификация и метод, то оно должно быть процедурой, возвращающей простое значение или константу того же типа, что тип свойства. Например, если свойство объявлено:

property Color: TColor read GetColor write SetColor;

тогда метод GetColor должен быть описан как

function GetColor: TColor;

и метод SetColor должен быть описан как

procedure SetColor(Value: TColor);

или

procedure SetColor(const Value: TColor);

Если свойство имеет спецификацию read, то оно имеет атрибут "read only" (только для чтения). Если свойство имеет спецификацию write, то оно имеет атрибут "write only" (только для чтения).

18.6. Структура класса

Всякий класс имеет структуру, которая состоит из секций. Каждая секция объявляется специальным зарезервированным словом. К их числу относятся: published (декларированные), private (частные), protected (защищенные), public (доступные), automated (автоматизированные). Внутри каждой секции сначала объявляются поля, затем – свойства и методы.

Пример:

type

TMyClass = class(TControl)

private

... { частные объявления здесь}

protected

... { защищенные объявления здесь }

public

... { доступные объявления здесь }

published

... { декларированные объявления здесь }

end;

Секции определяют области видимости компонент класса:

Private – компоненты класса доступны только внутри этого класса;

Public – компоненты класса доступны в текущем и любом другом модуле, который содержит ссылку в списке uses на модуль, в котором объявлен класс;

Published – то же, что Public, однако в ней должны быть перечислены свойства, которые доступны не только на этапе выполнения программы, но и на этапе ее визуального конструирования средствами Delphi;

Protected – cекция доступна только методам текущего класса и методам классов-предков;

Automated – секция используется для объявления свойств и методов обработки OLE-контейнеров в рамках OLE-технологии.

Порядок следования секций произволен. Любая из секций может быть как пустой, так и объявлена несколько раз в рамках одного класса.

18.7. Операции над классами

Над классами разрешено выполнять две операции – is и as.

1. Операция is. Синтаксис выражения, содержащего операцию is, имеет вид

object **is** class

Это выражение имеет логический тип (boolean) и возвращает True, если переменная object имеет тип class класса, иначе – False.

Пример:

if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;

В этом примере: если класс ActiveControl имеет тип TEdit, то будет выполнен метод TEdit(ActiveControl).SelectAll.

2. Операция as. Синтаксис выражения, содержащего операцию as:

object as class

Результатом вычисления этого выражения является ссылка на объект того же типа, что и тип класса class. При выполнении программы object может иметь тот же тип, или тип класса-потомка, или nil.

Примеры:

with Sender as TButton do // если Sender имеет тип TButton

begin // или тип-потомок от TButton

Caption := '&Ok';

OnClick := OkClick;

end;

(Sender as TButton).Caption := '&Ok'; //свойству Caption переменной

// Sender типа TButton или его потомка присваивается значение '&Ok'

Приложение

Перечень

отлаженных процедур и функций,

написанных автором

Ниже использованы глобальные типы и переменные:

Type

CompareType = (Less, Equal, Greater);

Var

Lon, Lon2: LongInt;

Serv: String[255];

**Procedure** Delay(MilliSec: LongInt);

{задержка времени на MilliSec миллисекунд}

Var k: LongInt;

begin

k:=GetTickCount; {в модуле Windows.pas}

While GetTickCount<(MilliSec+k) do ;

end;

**Function** Ctrl\_ Shift\_Down(i: byte): boolean;

{Нажата ли одна из этих клавиш Ctrl – 1, Shift – 2}

var

ShiftState: TShiftState;

j: LongInt;

begin

Result:=false;

Case i of

1: j:= VK\_CONTROL;

2: j:= VK\_SHIFT;

end;

ShiftState := KeyDataToShiftState(j);

Case i of

1: Result:= (ssCtrl in ShiftState);

2: Result:= (ssShift in ShiftState);

end;

end;

**Function** CtrlDown: boolean;

{нажата ли клавиша Ctrl}

begin

Result:=Ctrl\_ Shift\_Down(1);

end;

**Function** ShiftDown: boolean;

{нажата ли клавиша Shift}

begin

Result:=Ctrl\_Shift\_Down(2);

end;

**Function** Profit(Expend, Price: Real): Real;

{рентабельность=(цена - затраты)/затраты\*100}

begin

if (Expend<>0) then Result:= (Price/Expend-1.0)\*100.0

else Result:= 1.e5;

end;

**Procedure** Warn1(S: Variant);

{Окно с Variant-значением, например Warn1('Процесс закончен')}

begin

MessageDlg(S, mtInformation, [mbOk], 0);

Screen.ActiveForm.Refresh;

End;

**Procedure** Warn4(s1,s2,s3,s4: String);

{то же , что Warn1, но в 4 строки}

var i,j: byte;

begin

i:=Length(s1); j:=i;

i:=Length(s2);

if (i>j) then j:=i;

i:=Length(s3);

if (i>j) then j:=i;

i:=Length(s4);

if (i>j) then j:=i;

Warn1(Center(s1,j)+''#13#10+''+Center(s2,j)

+''#13#10''+Center(s3,j)+''#13#10+''+Center(s4,j));

end;

**Function** DaNet(S: String): boolean;

{Окно. Предназначено для вопроса, на который можно ответить, щелкнув по одной из кнопок "Да" или "Нет"}

begin

DaNet:=MessageDlg(S, mtConfirmation, [mbYes, mbNo], 0)=mrYes;

Screen.ActiveForm.Refresh;

end;

**Function** DaNet4(s1,s2,s3,s4: String): boolean;

{Окно. То же, что DaNet, только в 4 строки}

begin

DaNet4:=MessageDlg(Trim(s1)+''#13#10+''+Trim(s2)+''#13#10''+Trim(s3)

+''#13#10+''+Trim(s4),mtConfirmation,[mbYes, mbNo], 0)=mrYes;

Screen.ActiveForm.Refresh;

end;

**Function** InOtrReal(i,a,b: real): boolean;

{Если i в орезке [a, b], то возвращает True}

begin

Result:=(i>=a) and (i<=b);

end;

**Function** ExitK: boolean;

{стандартный вопрос о выходе}

begin

ExitK:=DaNet('Выход ?');

end;

**Function** Pos2(SubS, S: String; StartPos: byte): boolean;

{входит ли SubS в S начиная с StartPos}

begin

Lon:=Pos(SubS,S);

Result:= (Lon > 0) and (StartPos = Lon);

end;

**Function** ChStr(Ch: Char; d: Word): String;

{создает строку из символа Ch, повторенного d раз}

begin

if d>0 then

begin

SetLength(Result,d);

FillChar(Result[1],d,Ch);

end;

end;

**Function** Prop(d: Word): String;

{создает строку из d пробелов}

begin

Result:=ChStr(' ',d);

end;

**Function** Pad(s: String; d: Word): String;

{вставляет справа от строки пробелы, добирая ее до длины d}

begin

Serv:=s;

Lon:=Length(s);

If (d>Lon) then Serv:=s+Prop(d-Lon);

Result:=Serv;

end;

**Function** PadCopy(s: String; n,d: Word): String;

{копирует из s начиная с позиции n строку длины d. В случае меньшей строки добирает ее до длины d}

begin

Serv:=Copy(s,n,d);

if Length(Serv) < d then Serv:=Pad(Serv,d);

Result:=Serv;

end;

**Function** LeftPad(s: String; d: Word): String;

{вставляет слева от строки пробелы, добирая ее до длины d}

begin

Serv:=s;

Lon:=Length(s);

if (d>Lon) then Serv:=Prop(d-Lon)+s;

Result:=Serv;

end;

**Function** Center(s: String; d: Word): String;

{вставляет слева и справа от строки поровну пробелы, добирая ее до длины d}

begin

Serv:=s;

Lon:=Length(s);

Lon2:=Round(0.5\*(d-Lon));

if (d>Lon) then Serv:=Prop(Lon2)+s+Prop(d-Lon2);

Result:=Serv;

end;

**Functio**n CompStrings(s1,s2: String): CompareType;

{сравнение строк: s1<s2 - Less, s1=s2 - Equal, s1>s2 - Greater}

begin

if (s1<s2) then CompStrings:=Less

else

if (s1>s2) then CompStrings:=Greater

else

CompStrings:=Equal;

end;

**Function** CompReal(r1,r2: Real): CompareType;

{сравнение вещественных чисел}

begin

if (r1<r2) then Result:=Less

else

if (r1>r2) then Result:=Greater

else

Result:=Equal;

end;

**Procedure** IncRe(Var r: Real; h: real);

begin

r:=r+h;

end;

**Function** LongToStr(L: LongInt; d: byte): String;

{конвертирует целое в строку длины d}

begin

Str(L,Serv);

Result:=LPad(Serv,d);

end;

**Function** Long2Str(L: LongInt): String;

{конвертирует целое в строку}

begin

Str(L,Serv);

Result:=Serv;

end;

**Function** StrLong(st: String): LongInt;

{конвертирует строку в целое }

begin

Val(Trim(st),Lon,Code);

Result:=Lon; end;

**Function** Str2Long(st: String; Var L: LongInt): boolean;

{конвертирует строку в целое. Возвращает True в случае успеха}

begin

Val(Trim(st),L,Code);

Result:=(Code=0);

end;

**Function** RealToStr(R: Real; Posle: byte): String;

{Конвертирует Real в строку, Posle – количество символов в дробной части R}

begin

Str(R:20:Posle,Serv);

RealToStr:=Trim(Serv);

end;

**Function** Slash(Dir: String): String;

{ставит в конец пути символ '\'}

begin

Serv:=Trim(Dir);

if (Serv[Length(Serv)]<>'\') then Result:=Serv+'\'

else Result:=Serv;

end;

**Function** ChWinDos(Ch: Char): Char;

{преобразует русский Windows-символ в русский DOS-символ}

Var i,j: byte;

begin

i:=Ord(Ch);

Case i of

168: {Ё} j:=240;

184: {ё} j:=241;

192..255: if (i>239) then j:=i-16 else j:=i-64

else j:=i;

end;

Result:=Char(j);

end;

**Function** ChDosWin(Ch: Char): Char;

{преобразует русский DOS-символ в русский Windows-символ}

Var i,j: byte;

begin

i:=Ord(Ch);

Case i of

240: {Ё} j:=168;

241: {ё} j:=184;

128..175: j:=i+64;

224..239: j:=i+16

else j:=i;

end;

Result:=Char(j);

end;

**Function** StrWinDos(st: String): String;

{преобразует русскую Windows-строку в русскую DOS-строку}

Var

n, i: byte;

s: ^String;

begin

New(s);

n:=Length(st);

s^:= '';

if (n>0) then

for i:=1 to n do

s^:= s^+ChWinDos(st[i]);

Result:=s^;

Dispose(s);

end;

**Function** StrDosWin(s: String): String;

{преобразует русскую DOS-строку в русскую Windows-строку}

Var

n,i: byte;

s: ^String;

begin

New(s);

n:=Length(st);

s^:= '';

if (n>0) then

for i:=1 to n do

s^:= s^+ChDosWin(st[i]);

Result:=s^;

end;

**Functio**n InputStr(const Prompt: String; Var s: String; IsParol: byte): boolean;

{ввод строки. Prompt – пояснение, s – вводимая строка,

isParol=1, если засекреченный ввод, иначе видимый}

begin

Result:=

KdnInputQuery('Ввод строки', Prompt, s, clBlack, (IsParol=1));

end;

**Function** ParolControl(RealParol: String): boolean;

{возвращает True, если введенная строка совпадает с RealParol}

var

b,h: boolean;

i: byte;

begin

St:='';

i:=0;

b:=false;

Repeat

Inc(i);

h:=InputStr('Введите пароль ...',St,1);

if h then b:= (St=RealParol);

if not b and h then Warn1('Ошибка');

Until b or (i=3) or (not h);

Result:=b;

end;

**Function** ExistSubDir(SubDir:String; Dir: tPathStr):boolean;

{устанавливает наличие субдиректории SubDir внутри директории Dir. Например, в D:\DIR0001 субдиректории BAR }

begin

Result:=DirectoryExists(Slash(SubDir)+Dir);

end;

**Function** GetFileSize(const FileName: string): LongInt;

{размер файла}

var Sr: TSearchRec;

begin

if FindFirst(ExpandFileName(FileName), faAnyFile, Sr) = 0 then

Result := Sr.Size

else Result := -1;

end;

**Function** FileDateTime(const FileName: string): System.TDateTime;

{время создания файла FileName, например:

s:= DateTimeToStr(FileDateTime('c:\KdnBread\Bread.exe'))}

begin

Result := FileDateToDateTime(FileAge(FileName));

end;

**Function** HasAttr(const FileName: string; Attr: Word): Boolean;

{имеет ли файл FileName атрибут Attr}

begin

Result := (FileGetAttr(FileName) and Attr) = Attr;

end;

**Procedure** AppendText(Var f: Text; nF: String);

{открывает текстовой файл для добавления строк}

begin

Assign(f,nF);

if KdnFS(nF,1)>0 then Append(f) else Rewrite(f);

end;

**Procedure** AppendToText(nF,s: String);

{добавляет строку в конец текстового файла}

Var f: TextFile;

begin

AppendText(f, nF);

Writeln(f,s);

CloseFile(f);

end;

**Procedure** KdnExec(Command: String);

{запуск другого приложения, например 'c:\KdnBreadDir\KdnBread.exe'}

begin

Serv:=Command+#0;

If WinExec(@Serv[1], SW\_SHOWNORMAL)<32

then Warn2('Ошибочное завершение WinExec');

end;