АЛГОРИТМЫ И ПРОГРАММЫ

Автор: ФЕДОРОВ АЛЕКСАНДР МИХАЙЛОВИЧ - учитель информатики Кюкяйской средней общеобразовательной школы Сунтарского улуса Республики Саха, 2010 год

КОМАНДЫ ДЛЯ КОМПЬЮТЕРА

Компьютерная программа представляет собой список команд, которые указывают компьютеру , что он должен делать.Некоторые программы – системные – управляют основными действиями компьютера. Такие программы постоянно хранятся в ПЗУ, встроенном в компьютер.

Другие программы – прикладные – подробно указывают компьютеру, что нужно делать при выполнении определенной работы, например при обработке текста. Такие программы обычно записываются на диске и загружаются в компьютер, когда нужно. Все программы должны быть написаны очень тщательно, потому что ошибки в них ведут к неправильной работе компьютера.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРА

Каждая программа предназначена для практического использования:играть, сочинять музыку, производить вычисления или рисовать . Программы такого типа называются прикладными.

Прикладная программа обычно хранится на магнитной дискете или на CD. Эта программа представляет собой набор команд, вводимых в память компьютера.

Существует и другой тип программы: программа, которая управляет информацией, хранящейся на диске, называется дисковой операционной системой – ДОС(DOS). ПЗУ содержит , как правило, лишь необходимый минимум информации, а остальная часть ОС может быть считана с системного диска.

Оперативная память – это ЗУ, предназначенное для информации, непосредственно участвующей в процессе выполнения операций, выполняемых процессором.

Процессор и память – это главные компоненты компьютера.

Компьютер, обработав информацию из оперативной памяти, записывает ее во внешнюю память и “ листает книжку” дальше, считывая в оперативную память очередную порцию информации.

Для ПК имеются внешние носители информации: жесткие, гибкие, лазерные, магнитооптические диски, магнитные ленты и т. д.

ПЕРИФЕРИЙНЫЕ УСТРОЙСТВА(УСТРОЙСТВА ВВОДА – ВЫВОДА)

Клавиатура или манипулятор мышь - для ввода задания ПК
Сканер – ввод графического изображения
Дисковод – для получения информации из внешней памяти
Монитор – для вывода итогов работы ПК на экран
Принтер – для вывода на бумагу
Проектор – для вывода на экран слайдов
Модем – “ модулятор – демодулятор ” преобразует цифровые сигналы “ своего компьютера в аналоговые и посылает их в телефонную сеть. На конце телефонной линии другой модем преобразует аналоговые сигналы в цифровые.

Основы алгоритмического управления

Совокупность всех команд, которые может выполнить конкретный БИ, называется системой команд этого исполнителя. А совокупность всех действий , которые он может выполнить в ответ на эти команды, называется системой допустимых действий исполнителя.

Алгоритм – это организованная последовательность допустимых для некоторого исполнителя действий, приводящая к определенному результату, а программа – это запись алгоритма на языке конкретного БИ.

Как правило, алгоритмы пишутся для человека. При этом можно применять какие- то сокращения слов или заменять одни слова другими. Важно только , чтобы за этими словами стояли действия, допустимые для данного исполнителя.

Примеры: 1. Как открыть дверь.

2. Как приготовить хороший чай.

3. Как приготовить пельмени.

БИ – бездумный исполнитель(прикладные программы, любой компьютер со своей специфичной системой команд).

Основной задачей программирования является разработка проекта до такого уровня , чтобы в нем остались только стандартные конструкции и операторы из системы команд БИ, для которого пишется программа.

АЛГОРИТМ ПОЛУЧЕНИЯ КИПЯТКА

Исправьте алгоритм , чтобы предотвратить несчастный случай

Налить в чайник воду.

Открыть кран газовой горелки.

Поставить чайник на плиту.

Ждать,пока вода не закипит.

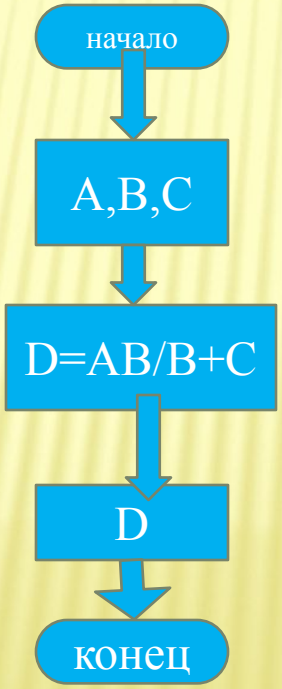
Поднести спичку к горелке.

Зажечь спичку.

Выключить газ.

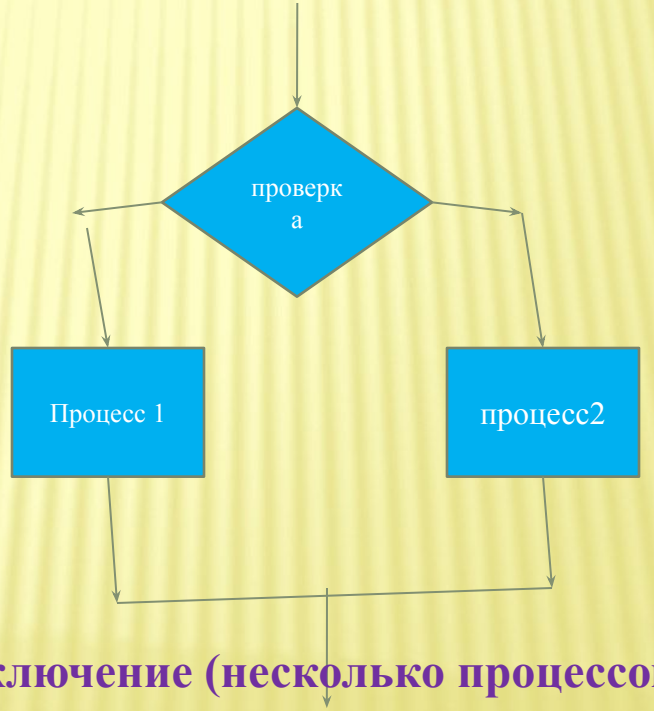
Алгоритм и его свойства

Алгоритмы – это организованная последовательность действий, допустимых для некоторого БИ. Алгоритм не роскошь, а средство достижения цели.
Линейный алгоритм – набор команд, выполняемых последовательно во времени друг за другом.



Разветвляющийся алгоритм

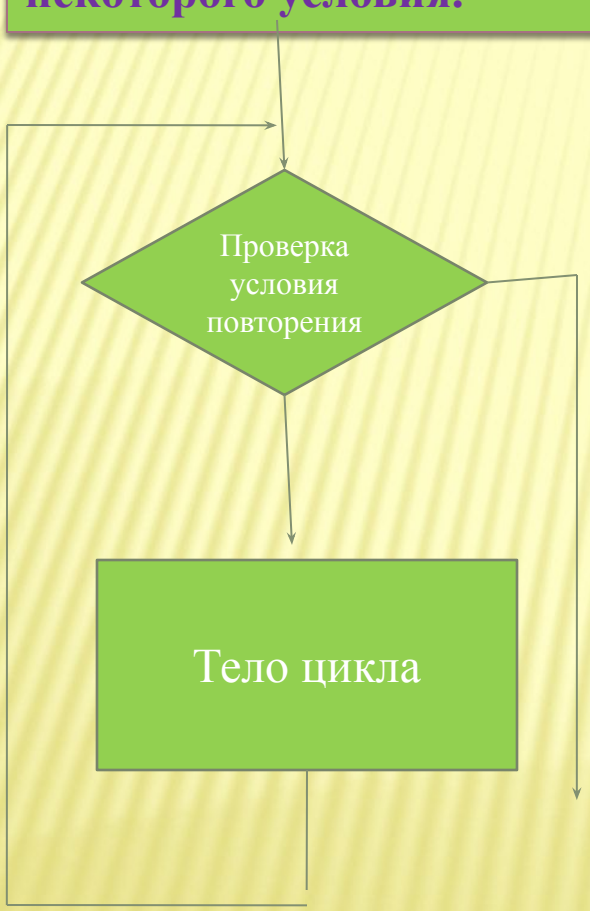
Разветвляющийся алгоритм– алгоритм, содержащий хотя бы одно условие, в результате проверки которого БИ переходит на один из двух возможных шагов.



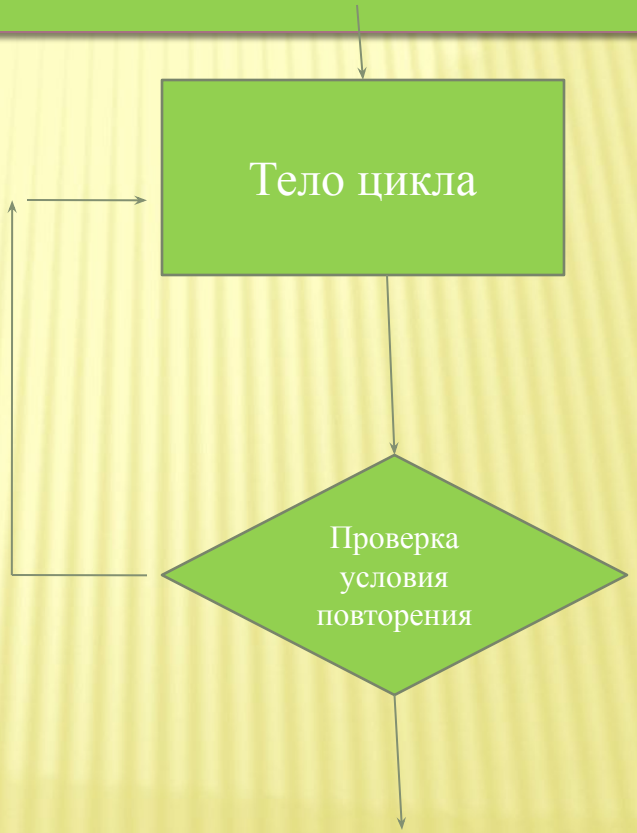
ответвление, раздвоение, переключение (несколько процессов)

Циклический алгоритм

Многократное повторение одного и того же действия над новыми исходными данными. Цикл программы – последовательность команд(серия, тело цикла) , которая которая может выполняться до удовлетворения некоторого условия.



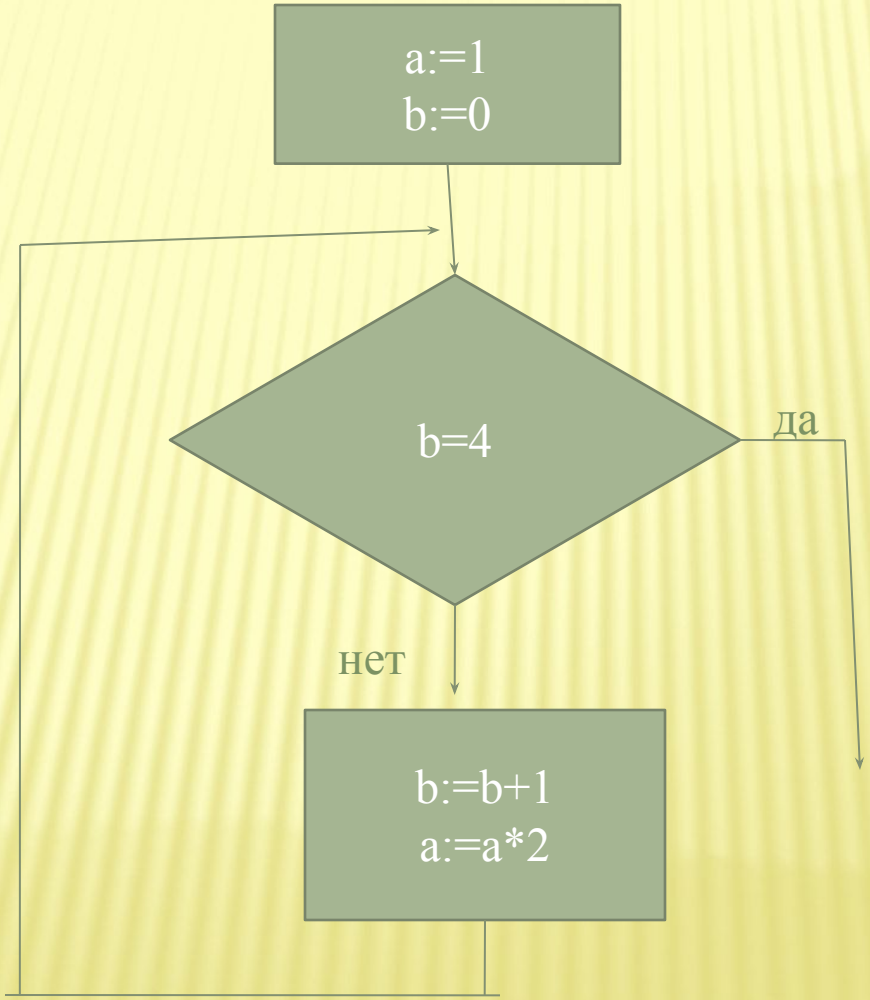
Проверка в начале цикла



Проверка в конце цикла

БЛОК-СХЕМА программы покупки арахиса

Определите значение переменной a после выполнения фрагмента алгоритма



* Умножение

:= присваивание

Алгоритмический язык

АЯ – это система обозначений формальной записи алгоритмов, предназначенных для анализа их исполнителем – человеком,а затем и для реализации на машине.

В общем виде имеются операторы (описание однородных этапов изучаемого процесса) четырех типов:

1. Действующие; определяют изменение в объекте.
2. Логические; описывают условия, от которых зависит направление процесса.
3. Варьирующие;определяют изменения вспомогательных величин, называемых *параметрами.*
4. Операторы перехода; определяют порядок выполнения операторов первых трех типов. Кроме операторов, применяются знаки *начала и конца* процесса.

Оператор предписывает исполнителю алгоритма **команду. Команды** обозначают одним полным или сокращенным словом,а также определенным значком. Эти обозначения команд назыв. *служебными словами и символами.*

Алгоритм вычисления суммы пяти чисел

```
алг  сумма ( цел S, цел N, таб M[1:5])
       арг  N
       рез S
            S:=0            - действующий
            N:=5            - действующий
нач  цел   i                - знак начала процесса
             i:=0           - действующий
          пока  i <=N       -логический
          нц                - перехода
               i:=i+1       - действующий
               S:=S+M[i]    - действующий
           кц               -перехода
 кон                        - знак конца процесса
```

БИБЛИОТЕКА АЛГОРИТМОВ

А1. Алгоритм вычисления модуля действительного числа

```
алг  МОД  ( вещ x, вещ y )
        арг x
        рез y
нач
        если  x>=0
                то y:=x
                иначе  y:=-x
        все
кон
```

А2. Алгоритм поиска большего из двух чисел a и b

```
алг  БИД  ( вещ  a,b, вещ y)
        арг a,b
        рез y
нач
        если  a>=b
                то  y:=a
                иначе  y:=b
        все
кон
```

МНОГОКРАТНОЕ СУММИРОВАНИЕ (ПРОИЗВЕДЕНИЕ)

1. $\sum_{k=1}^{n} k$ =1+2+3+…+n;

2. $\prod_{k=1}^{n} k$ =1*2*3*…*n =n!(факториал)

3. $\prod_{k=1}^{n} x$ =x*x*x*…*x= x^n

```
алг  задача1 ( цел n,s )
            дано n
            надо s
   нач  цел k
        s:=0
        нц  для  k  от 1  до n
        s:=s+k
         кц
    кон
```

k - называется параметром суммирования(умножения)
k изменяется по закону k:=k+1
Суммирование:
S:=0 {начальное значение суммы}
нц для k от a до b
S:=S+f_k
кц

Умножение:
P:=1 { начальное значение произведения }
нц для k от a до b
P:=P*f_k
кц

a – начальное значение параметра k, b- конечное значение k
f_k – k-й член (k- е слагаемое(сомножитель)).

```
алг  задача2(цел n, fact)
        дано n
        надо fact
нач  цел   k
         fact :=1
        нц  для  k  от  1 до n
          fact:= fact*k
         кц
кон
```

Для задачи3 составить алгоритм самим

Язык программирования ПАСКАЛЬ

1.**Начальные операторы** Program < имя >;
begin
end

2. **Описание числовых переменных** Var < список переменных > :
integer ; - целочисленные переменные
Var < список переменных >:
real; - вещественные

3. **Оператор условия** IF (< условие>) THEN
begin
< оператор >;
…
end
ELSE
begin
< оператор>
…
end;

Конструкция ИНАЧЕ (ELSE) может отсутствовать. В языке ПАСКАЛЬ необходимо после последнего оператора END поставить точку с запятой.

4. Оператор цикла по условию WHILE (< условие>) DO
begin
< оператор >
…
end;

5. Оператор цикла с параметром FOR < переменная > := < арифм. выражение >
TO
begin
< оператор >
…
end;

6. Работа с подпрограммами Procedure < имя >
(< список параметров с указанием их типов >)
begin
< оператор >
…
end;

Для возврата в главную программу служит оператор EXIT;

7. Операторы ввода и вывода информации

READLN(<имя переменной>);
WRITE(< сообщение >,<имя переменной>);

Алгоритм вычисления члена ряда Фибоначчи

```
PROGRAM Fib;
VAR  A1,A2,N,R,I: INTEGER;
BEGIN
      READLN(N);
      A1:=0;
       A2:=1;
       FOR I:=1 TO N DO
         BEGIN
                R:= A1+A2;
                A1:=A2;
                A2:=R;
           END;
         WRITE (N, '-й член равен', A2)
END.
```

СИМВОЛЬНОЕ КОДИРОВАНИЕ

Бит – 0 или 1 ; Байт – последовательность битов (8 или 16 битов) ,один символ соответствует 1 байту. 1 килобайт = 1024 байта, 1 мегабайт = 1024 кбайт, 1 гигабайт = 1024 мегабайт.

Считая, что каждый символ кодируется 16 – ю битами , оцените информационный объем следующей пушкинской фразы в кодировке Unicode:
Привычка свыше нам дана: Замена счастию она.

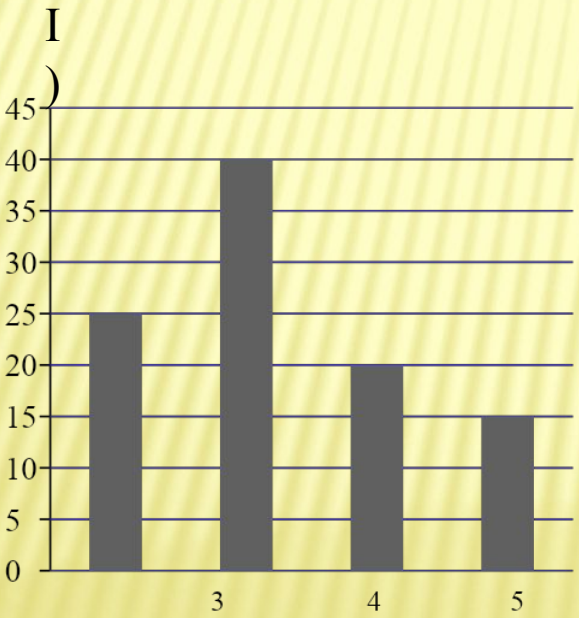
Световое табло состоит из лампочек, каждая из которых может находиться в двух состояниях (“вкл” и “выкл”) . Какое наименьшее количество лампочек должно находиться на табло, чтобы с его помощью можно было передать 50 различных сигналов?

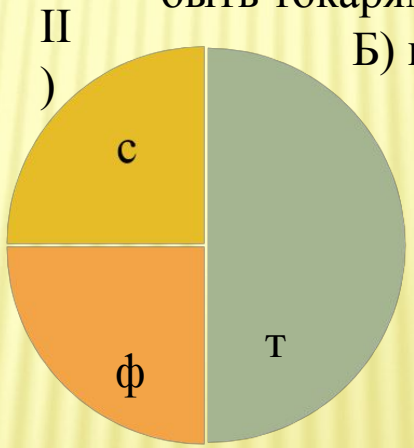
Сколько единиц в двоичной записи числа 195?

Значение выражения 16+8*2 в двоичной системе счисления равно:
----------?

Диаграммы

В цехе трудятся рабочие трех специальностей – токари(Т), слесари (С) и фрезеровщики . Каждый рабочий имеет разряд не меньший второго и не больший пятого. На диаграмме I отражено количество рабочих с различными разрядами, а на диаграмме II – распределение рабочих по специальностям.
Каждый рабочий имеет только одну специальность и один разряд.





Имеются четыре утверждения:

А) все рабочие третьего разряда могут быть токарями

Б) все рабочие третьего разряда могут быть фрезеровщиками

В) все слесари могут быть пятого разряда

Г) все токари могут быть четвертого разряда

АЛГОРИТМИЧЕСКИЙ ЯЗЫК

- Ада
  - В память дочери поэта лорда Байрона первой программистки
  - 1980 год
- Алгол
  - Один из первых универсальных языков
  - Алгол – 60 , Алгол – 68 и др
- Бейсик
  - Самый распространенный в мире
  - Имеет много диалектов

Кобол, Паскаль , ПЛ -1, Си, Фортран, Ассемблер и т. д.

Язык программирования

Язык Паскаль был разработан Никлаусом Виртом в 1970 г. как язык со строгой типизацией и интуитивно понятным синтаксисом. В 80-е годы наиболее известной реализацией стал компилятор Turbo Pascal фирмы Borland, в 90-е ему на смену пришла среда программирования Delphi, которая стала одной из лучших сред для быстрого создания приложений под Windows. Delphi ввела в язык Паскаль ряд удачных объектно-ориентированных расширений, обновленный язык получил название Object Pascal

PascalABC.NET - достаточно зрелая среда. Ее прототип - учебная система Pascal ABC - появилась в 2002 году и достаточно активно используется в российских школах.
PascalABC.NET - развивающаяся среда. Ведутся разработки новых языковых возможностей, новых библиотек и новых языков программирования, которые будут интегрированы в общую среду

Структура программы: обзор

Программа на языке **PascalABC.NET** имеет следующий вид:
program имя программы;
раздел **uses**
раздел описаний
begin
  операторы
end.
Первая строка называется ***заголовком программы*** и не является обязательной.
Раздел **uses** начинается с ключевого слова **uses**, за которым следует список имен модулей и пространств имен .NET, перечисляемых через запятую.
Раздел описаний может включать разделы описания переменных, констант, меток, типов, процедур и функций, которые следуют друг за другом в произвольном порядке.
Далее следует блок **begin**/**end**, внутри которого находятся операторы, отделяемые один от другого символом "точка с запятой".
Раздел **uses** и раздел описаний могут отсутствовать.
Например:

```
program MyProgram;
var
  a,b: integer;
  r: real;
begin
  readln(a,b);
  x := a/b;
  writeln(x);
end;
```

Описание переменных

Переменные могут быть описаны в разделе описаний, а также непосредственно внутри любого блока **begin**/**end**.
Раздел описания переменных начинается со служебного слова **var**, после которого следуют элементы описания вида
список имен: тип; или
имя: тип := выражение; или
имя := выражение; Имена в списке перечисляются через запятую. Например:

```
var
  a,b,c: integer;
  d: real := 3.7;
  s := 'Pascal forever';
  al := new ArrayList;
  p1 := 1;
```

В последних трех случаях тип переменной определяется по типу правой части.
Переменные могут описываться непосредственно внутри блока. ***Внутриблочные описания переменных*** имеют тот же вид, что и в разделе описаний, с тем исключением, что в каждой секции **var** может быть лишь один элемент описания:

```
begin
  var a1,a2,a3: integer;
  var s := '';
  ...
end.
```

Кроме того, переменные-параметры цикла могут описываться в заголовке операторов **for** и **foreach**

Описание констант

Раздел описания именованных констант начинается со служебного слова **const**, после которого следуют элементы описания вида
имя константы = значение; или
имя константы : тип = значение; Например:

```
const
  Pi = 3.14;
  Count = 10;
  Name = 'Mike';
  DigitsSet = ['0'..'9'];
  Arr: array [1..5] of integer = (1,3,5,7,9);
  Rec: record name: string; age: integer end = (name: 'Иванов'; age: 23);
  Arr2: array [1..2,1..2] of real = ((1,2),(3,4));
```

Описание меток

Раздел описания меток начинается с зарезервированного слова **label**, после которого следует список меток, перечисляемых через запятую. В качестве меток могут быть использованы идентификаторы и положительные целые числа:

label a1,l2,777777;

Описание типов

Раздел описания типов начинается со служебного слова **type**, после которого следуют строки вида имя типа = тип; Например,

```
type
  myint = integer;
  arr10 = array [1..10] of integer;
  pinteger = ^integer;
  A = class
    i: integer;
    constructor Create(ii: integer);
    begin
      i:=ii;
    end;
  end;
```

При описании рекурсивных структур данных указатель на тип может фигурировать раньше описания самого типа в определении другого типа:

```
type
  PNode = ^TNode;
  TNode = record
    data: integer;
    next: PNode;
  end;
```

При этом важно, чтобы определения обоих типов находились в одном разделе **type**.

В отличие от Delphi Object Pascal следующее рекурсивное описание верно:

```
type
  TNode = record
    data: integer;
    next: ^TNode;
  end;
```

Отметим, что для ссылочных типов (классов) разрешается описание поля с типом, совпадающим с типом текущего класса:

```
type
  Node = class
    data: integer;
    next: Node;
  end;
```

Область действия идентификатора

Любой используемый в блоке идентификатор должен быть предварительно описан. Идентификаторы описываются в разделе описаний. Идентификаторы для переменных могут также описываться внутри блока.

Основная программа, подпрограмма, блок, модуль, класс образуют так называемое ***пространство имен*** - область в программе, в которой имя должно иметь единственное описание. Таким образом, в одном пространстве имен не может быть описано двух одинаковых имен (исключение составляют перегруженные имена подпрограмм). Кроме того, в сборках .NET имеются явные определения пространств имен.

Область действия идентификатора (т.е. место, где он может быть использован) простирается от момента описания до конца блока, в котором он описан. Область действия глобального идентификатора, описанного в модуле, простирается на весь модуль, а также на основную программу, к которой данный модуль подключен в разделе **uses**.

Кроме этого, имеются переменные, определенные в блоке и связанные с некоторыми конструкциями (**for**, **foreach**). В этом случае действие переменной i простирается до конца соответствующей конструкции.

Так, следующий код корректен:

```
var a: array of integer := (3,5,7);
for i: integer := 1 to 9 do
  write(a[i]);
foreach i: integer in a do
  write(i);
```

Идентификатор с тем же именем**, определенный во вложенном пространстве** имен, *скрывает* идентификатор, определенный во внешнем пространстве имен.

Например, в коде

```
var i: integer;
procedure p;
var i: integer;
begin
  i := 5;
end;
```

значение 5 будет присвоено переменной i, описанной в процедуре p; внутри же процедуры p сослаться на глобальную переменную i невозможно.

Переменные, описанные внутри блока, не могут иметь те же имена, что и переменные из раздела описаний этого блока. Например, следующая программа ошибочна:

```
var i: integer;
begin
  var i: integer; // ошибка
end.
```

В производных классах, напротив, можно определять члены с теми же именами, что и в базовых классах, при этом их имена скрывают соответствующие имена в базовых классах. Для обращения к одноименному члену базового класса из метода производного класса используется ключевое слово inherited:

```
type
  A=class
    i: integer;
    procedure p;
    begin
      i := 5;
    end;
  end;
  B=class(A)
    i: integer;
    procedure p;
    begin
      i := 5;
      inherited p;
    end;
  end;
```

Алгоритм поиска имени в классе следующий: вначале имя ищется в текущем классе, затем в его базовых классах, а если не найдено, то в глобальной области видимости.
Алгоритм поиска имени в глобальной области видимости при наличии нескольких подключенных модулей следующий: вначале имя ищется в текущем модуле, затем, если не найдено, по цепочке подключенных модулей в порядке справа налево. Например, в программе

```
uses unit1,unit2;
begin
  id := 2;
end.
```

описание переменной id будет искаться вначале в основной программе, затем в модуле unit2, затем в модуле unit1. При этом в разных модулях могут быть описаны разные переменные id. Данная ситуация означает, что unit1 образует внешнее пространство имен, пространство имен unit2 в него непосредственно вложено, а пространство имен основной программы вложено в unit2.
Если в последнем примере оба модуля - unit1 и unit2 - определяют переменные id, то рекомендуется уточнять имя переменной именем модуля, используя конструкцию *ИмяМодуля.Имя*:

```
uses unit1,unit2;
begin
  unit1.id := 2;
end.
```

Обзор типов

Типы в **PascalABC.NET** подразделяются на простые, строковые, структурированные, типы указателей, процедурные и классовые типы.
К ***простым*** относятся целые и вещественные типы, логический, символьный, перечислимый и диапазонный тип.
Структурированные типы образованы массивами, записями, множествами и файлами.
Все простые типы, кроме вещественного, называются ***порядковыми***. Только значения этих типов могут быть индексами массивов с фиксированной размерностью и параметрами цикла **for**. Кроме того, для порядковых типов используются функции Ord, Pred и Succ, а также процедуры Inc и Dec.
Все типы, кроме типов указателей, являются производными от типа Object (как в .NET).
Каждый тип в **PascalABC.NET** имеет отображение на тип .NET. Тип указателя принадлежит к неуправляемому коду и моделируется типом void*.
Все типы подразделяются на две большие группы: ***размерные*** и ***ссылочные***. Данные размерных типов располагаются на стеке, данные ссылочных типов представляют собой указатели, хранящие адрес динамической памяти, где расположены собственно данные указанного типа. К размерным относятся все простые типы, указатели, записи, статические массивы. К ссылочным типам относятся множества, классы, динамические массивы, строки, файлы и процедурный тип.

Строковый тип

Строки имеют тип string, состоят из набора последовательно расположенных символов char и используются для представления текста.
Строки могут иметь произвольную длину. К символам в строке можно обращаться, используя индекс: s[i] обозначает i-тый символ в строке, нумерация начинается с единицы. Если индекс i выходит за пределы длины строки, то генерируется исключение.
Операция + для строк означает конкатенацию (слияние) строк. Например: 'Петя'+'Маша' = 'ПетяМаша'.
Операция += для строк добавляет в конец строки - левого операнда строку - правый операнд. Например:

```
var s: string := 'Петя';
s += 'Маша'; // s = 'ПетяМаша'
```

Строки реализуются типом System.String платформы .NET и представляют собой ссылочный тип. Таким образом, все операции над строками унаследованы от типа System.String. Однако, в отличие от .NET - строк, строки в **PascalABC.NET** изменяемы. Например, можно изменить s[i] (в .NET нельзя). Более того, строки string в **PascalABC.NET** ведут себя как размерные: после

```
var s2 := 'Hello';
var s1 := s2;
s1[2] := 'a';
```

строка s1 не изменится. Аналогично при передаче строки по значению в подпрограмму создается копия строки, т.е. обеспечивается поведение, характерное для Delphi Object Pascal, а не для .NET.
Однако, строке можно присвоить nil, что необходимо для работы с NET-кодом.
Кроме того, в **PascalABC.NET** реализованы размерные строки. Для их описания используется тип string[n], где n - константа целого типа, указывающая длину строки. В отличие от обычных строк, содержимое размерных строк при присваивании и передаче по значению копируется. Кроме того, размерные строки, в отличие от обычных, можно использовать как компоненты типизированных файлов. Для совместимости с Delphi Object Pascal в стандартном модуле описан тип shortstring=string[255].
Стандартные подпрограммы работы со строками представлены здесь.
В .NET каждый тип является классом. Члены класса string приведены здесь. Отметим, что в методах класса string считается, что строки индексируются с нуля.

Указатели

Указатель - это ячейка памяти, хранящая адрес. В **PascalABC.NET** указатели делятся на ***типизированные*** (содержат адрес ячейки памяти данного типа) и ***бестиповые*** (содержат адрес оперативной памяти, не связанный с данными какого-либо определенного типа).
Тип указателя на тип T имеет форму ^T, например:

```
type pinteger = ^integer;
var p: ^record r,i: real end;
```

Бестиповой указатель описывается с помощью слова pointer.
Для доступа к ячейке памяти, адрес которой хранит типизированный указатель, используется ***операция разыменования*** ^:

```
var
  i: integer;
  pi: ^integer;
...
pi := @i; // указателю присвоили адрес переменной i
pi^ := 5; // переменной i присвоили 5
```

Операция разыменования не может быть применена к бестиповому указателю.
Типизированный указатель может быть неявно преобразован к бестиповому:

```
var
  p: pointer;
  pr: ^real;
...
p := pr;
```

Обратное преобразование также может быть выполнено неявно:

```
pr := p;
pr^ := 3.14;
```

Указатели можно сравнивать на равенство (=) и неравенство (<>). Для того чтобы отметить тот факт, что указатель никуда не указывает, используется стандартная константа **nil** (нулевой указатель) : p := **nil**.
Внимание! Ввиду особенностей платформы .NET тип T типизированного указателя не должен быть ссылочным или содержать ссылочные типы на каком-то уровне (например, запрещены указатели на записи, у которых одно из полей имеет ссылочный тип).
Причина такого ограничения проста: указатели реализуются неуправляемым кодом, который не управляется сборщиком мусора. Если в памяти, на которую указывает указатель, содержатся ссылки на управляемые переменные, то они становятся недействительными после очередной сборки мусора. Исключение составляют динамические массивы и строки, обрабатываемые особым образом. То есть, можно делать Указатели на записи, содержащие в качестве полей строки и динамические массивы.

Процедурный тип

Переменные, предназначенные для хранения процедур и функций, называются ***процедурными***. Тип процедурной переменной представляет собой заголовок процедуры или функции без имени. Например:

```
type
  procI = procedure(i: integer);
  funI = function: integer;
```

Процедурной переменной можно присвоить процедуру или функцию с совместимым типом:

```
procedure my(i: integer);
begin
  ...
end;
function f: integer;
begin

end;
...
var
  p1: procI;
  f1: funI;
...
p1 := my;
f1 := f;
```

После этого можно вызвать процедуру или функцию через эту процедурную переменную, пользуясь обычным синтаксисом вызова:

```
p1(5);
write(f1);
```

Для процедурных переменных принята структурная эквивалентность типов: можно присваивать друг другу и передавать в качестве параметров процедурные переменные, совпадающие по структуре (типы и количество параметров, тип возвращаемого значения).

Обычно процедурные переменные передаются как параметры для реализации *обратного вызова*:

```
procedure forall(var a: array of real; p: procedure(var r: real));
begin
  for var i := 0 to a.Length-1 do
    p(a[i]);
end;
procedure mult2(var r: real);
begin
  r := 2*r;
end;
procedure add3(var r: real);
begin
  r := r + 3;
end;
procedure print(var r: real);
begin
  write(r,' ');
end;
...
forall(a,mult2); // умножение элементов массива на 2
forall(a,add3);  // увеличение элементов массива на 3
forall(a,print); // вывод элементов массива
```

Процедурная переменная может хранить нулевое значение, которое задается константой nil. Вызов подпрограммы через нулевую процедурную переменную приводит к ошибке.

Процедурные переменные реализуются через делегаты .NET. Это означает, что они могут хранить несколько подпрограмм. Для добавления/отсоединения подпрограмм используются операторы += и -=:

```
p1 += mult2;
p1 += add3;
forall(a,p1);
```

Подпрограммы в этом случае вызываются в порядке прикрепления: вначале умножение, потом сложение.

Отсоединение неприкрепленных подпрограмм не выполняет никаких действий:

```
p1 -= print;
```

Кроме того, к процедурной переменной можно прикреплять/откреплять статические и экземплярные методы классов.

Пример.

```
type
  A = class
  private
    x: integer;
  public
    constructor Create(xx: integer);
    begin
      x := xx;
    end;
    procedure pp;
    begin
      write(x);
    end;
    procedure ppstatic; static;
    begin
      write(1);
    end;
  end;
begin
  var p: procedure;
  var a1: A := new A(5);
  p += a1.pp;
  p += A.ppstatic;
  p;
end.
```

В результате запуска данной программы на экран будет выведено:

51

Целые типы

Ниже приводится таблица целых типов, содержащая также их размер и диапазон допустимых значений.

Тип	Размер, байт	Диапазон значений
`shortint`	1	-128..127
`smallint`	2	-32768..32767
`integer`, `longint`	4	-2147483648..2147483647
`int64`	8	-9223372036854775808..9223372036854775807
`byte`	1	0..255
`word`	2	0..65535
`longword`, `cardinal`	4	0..4294967295
`uint64`	8	0..18446744073709551615

Типы `integer` и `longint`, а также `uint64` и `cardinal` являются синонимами.

Максимальные значения для каждого целого типа определены как внешние стандартные константы: `MaxInt64`, `MaxInt`, `MaxSmallInt`, `MaxShortInt`, `MaxUInt64`, `MaxLongWord`, `MaxWord`, `MaxByte`.

Для каждого целого типа `T` определены также следующие константы как члены класса

`T.MinValue` - константа, представляющая минимальное значение типа `T`;

`T.MaxValue` - константа, представляющая максимальное значение типа `T`;

Константы модуля PABCSystem

```
MaxShortInt = shortint.MaxValue;
```
Максимальное значение типа shortint

```
MaxByte = byte.MaxValue;
```
Максимальное значение типа byte

```
MaxSmallInt = smallint.MaxValue;
```
Максимальное значение типа smallint

```
MaxWord = word.MaxValue;
```
Максимальное значение типа word

```
MaxInt = integer.MaxValue;
```
Максимальное значение типа integer

```
MaxLongWord = longword.MaxValue;
```
Максимальное значение типа longword

```
MaxInt64 = int64.MaxValue;
```
Максимальное значение типа int64

```
MaxUInt64 = uint64.MaxValue;
```
Максимальное значение типа uint64

```
MaxDouble = real.MaxValue;
```
Максимальное значение типа double

```
MinDouble = real.Epsilon;
```
Минимальное положительное значение типа double

```
MaxReal = real.MaxValue;
```
Максимальное значение типа real

```
MinReal = real.Epsilon;
```
Минимальное положительное значение типа real

```
MaxSingle = single.MaxValue;
```
Максимальное значение типа single

```
MinSingle = single.Epsilon;
```
Минимальное положительное значение типа single

```
Pi = 3.141592653589793;
```
Константа Pi

```
E = 2.718281828459045;
```
Константа E

Вещественные типы

Ниже приводится таблица вещественных типов, содержащая их размер, количество значащих цифр и диапазон допустимых значений:

Тип	Размер, байт	Количество значащих цифр	Диапазон значений
real	8	15-16	$-1.8 \cdot 10^{308} .. 1.8 \cdot 10^{308}$
double	8	15-16	$-1.8 \cdot 10^{308} .. 1.8 \cdot 10^{308}$
single	4	7-8	$-3.4 \cdot 10^{38} .. 3.4 \cdot 10^{38}$

Типы real и double являются синонимами. Самое маленькое положительное число типа real приблизительно равно $5.0 \cdot 10^{-324}$, для типа single оно составляет приблизительно $1.4 \cdot 10^{-45}$.

Максимальные значения для каждого вещественного типа определены как внешние стандартные константы: MaxReal, MaxDouble и MaxSingle.

Для каждого вещественного типа R определены также следующие константы как члены класса:

R.MinValue - константа, представляющая минимальное значение типа R;

R.MaxValue - константа, представляющая максимальное значение типа R;

R.Epsilon - константа, представляющая самое маленькое положительное число типа R;

R.NaN - константа, представляющая не число (возникает, например, при делении 0/0);

R.NegativeInfinity - константа, представляющая отрицательную бесконечность (возникает, например, при делении -2/0);

R.PositiveInfinity - константа, представляющая положительную бесконечность (возникает, например, при делении 2/0).

Для каждого вещественного типа R определены следующие статические функции:

R.IsNaN(r) - возвращает True, если в r хранится значение R.NaN, и False в противном случае;

R.IsInfinity(r) - возвращает True, если в r хранится значение R.PositiveInfinity или R.NegativeInfinity, и False в противном случае;

R.IsPositiveInfinity(r) - возвращает True, если в r хранится значение R.PositiveInfinity, и False в противном случае;

R.IsNegativeInfinity(r) - возвращает True, если в r хранится значение R.NegativeInfinity, и False в противном случае;

`R.Parse(s)` - функция, конвертирующая строковое представление числа в значение типа `R`. Если преобразование невозможно, то генерируется исключение;

`R.TryParse(s,res)` функция, конвертирующая строковое представление числа в значение типа `R` и записывающая его в переменную `res`. Если преобразование возможно, то возвращается значение `True`, в противном случае - `False`.

Кроме того, определена экземплярная функция `ToString`, возвращающая строковое представление переменной типа `R`.

Вещественные константы можно записывать как в форме с плавающей точкой, так и в экспоненциальной форме:

`1.7` `0.013` `2.5e3` (2500) `1.4e-1` (0.14)

Логический тип

Значения логического типа boolean занимают 1 байт и принимают одно из двух значений, задаваемых предопределенными константами True (истина) и False (ложь).

Для логического типа определены статические функции:

boolean.Parse(s) - функция, конвертирующая строковое представление числа в значение типа boolean. Если преобразование невозможно, то генерируется исключение;

boolean.TryParse(s,res) - функция, конвертирующая строковое представление числа в значение типа boolean и записывающая его в переменную res. Если преобразование возможно, то возвращается значение True, в противном случае - False.

Кроме этого, определена экземплярная функция ToString, возвращающая строковое представление переменной типа boolean.

Логический тип является порядковым. В частности, False<True, Ord(False)=0, Ord(True)=1.

Символьный тип

Символьный тип char занимает 2 байта и хранит Unicode-символ. Символы реализуются типом System.Char платформы .NET.
Стандартные подпрограммы работы с символами представлены здесь.
Члены класса char приведены здесь.
Для преобразования между символами и их кодами в кодировке Windows (CP1251) используются стандартные функции Chr и Ord:
Chr(n) - функция, возвращающая символ с кодом n в кодировке Windows;
Ord(c) - функция, возвращающая значение типа byte, представляющее собой код символа c в кодировке Windows.
Для преобразования между символами и их кодами в кодировке Unicode используются стандартные функции ChrUnicode и OrdUnicode:
ChrUnicode(w) - возвращает символ с кодом w в кодировке Unicode;
OrdUnicode(c) - возвращает значение типа word, представляющее собой код символа c в кодировке Unicode.
Кроме того, выражение *#число* возвращает Unicode-символ с кодом *число* (число должно находиться в диапазоне от 0 до 65535).
Аналогичную роль играют явные преобразования типов:
char(w) возвращает символ с кодом w в кодировке Unicode;
word(c) возвращает код символа c в кодировке Unicode.

Перечислимый и диапазонный типы

Перечислимый тип определяется упорядоченным набором идентификаторов.

type typeName = (value1, value2, ..., valuen);

Значения перечислимого типа занимают 4 байта. Каждое значение value представляет собой константу типа typeName, попадающую в текущее пространство имен.

Например:

```
type
  Season = (Winter,Spring,Summer,Autumn);
  DayOfWeek = (Mon,Tue,Wed,Thi,Thr,Sat,Sun);
```

К константе перечислимого типа можно обращаться непосредственно по имени, а можно использовать запись typeName.value, в которой имя константы уточняется именем перечислимого типа, к которому она принадлежит:

```
var a: DayOfWeek;
a := Mon;
a := DayOfWeek.Wed;
```

Для значений перечислимого типа можно использовать функции Ord, Pred и Succ, а также процедуры Inc и Dec. Функция Ord возвращает порядковый номер значения в списке констант соответствующего перечислимого типа, нумерация при этом начинается с нуля.

Для перечислимого типа определена экземплярная функция ToString, возвращающая строковое представление переменной перечислимого типа. При выводе значения перечислимого типа с помощью процедуры write также выводится строковое представление значения перечислимого типа.

Например:

```
type Season = (Winter,Spring,Summer,Autumn);
var s: Season;
begin
  s := Summer;
  writeln(s.ToString); // Summer
  writeln(s); // Summer
end.
```

Диапазонный тип представляет собой подмножество значений целого, символьного или перечислимого типа и описывается в виде a..b, где a - нижняя, b - верхняя граница интервального типа, a<b:

```
var
  intI: 0..10;
  intC: 'a'..'z';
  intE: Mon..Thr;
```

Тип, на основе которого строится диапазонный тип, называется *базовым* для этого диапазонного типа. Значения диапазонного типа занимают в памяти столько же, сколько и значения соответствующего базового типа.

Класс string

Тип string в PascalABC.NET является классом и содержит ряд свойств, статических и экземплярных методов.

Свойства класса String

Свойство	Описание
`s[i]`	Индексное свойство. Возвращает или позволяет изменить i-тый символ строки s. Строки в PascalABC.NET индексируются от 1.
`Length: integer`	Возвращает длину строки

Статические методы класса String

Метод	Описание
`String.Compare(s1,s2: string): integer`	Сравнивает строки s1 и s2. Возвращает число <0 если s1<s2, =0 если s1=s2 и >0 если s1>s2
`String.Compare(s1,s2: string; ignorecase: boolean): integer`	То же. Если ignorecase=True, то строки сравниваются без учета регистра букв
`String.Format(fmtstr: string, params arr: array of object): string;`	Форматирует параметры arr согласно форматной строке fmtstr
`String.Join(ss: array of string; delim: string): string`	Возвращает строку, полученную слиянием строк ss с использованием delim в качестве разделителя

Экземплярные методы класса String

Отметим, что все экземплярные методы не меняют строку, как это может показаться на первый взгляд, а при необходимости возвращают измененную строку. Кроме того, считается, что символы в сроке индексируются с нуля.

Метод	Описание
`Contains(s: string): boolean`	Возвращает True, если текущая строка содержит s, и False в противном случае
`EndsWith(s: string): boolean`	Возвращает True, если текущая строка заканчивается на s, и False в противном случае

`String.Format(fmtstr: string, params arr: array of object): string;`	Форматирует параметры arr согласно форматной строке fmtstr
`String.Join(ss: array of string; delim: string): string`	Возвращает строку, полученную слиянием строк ss с использованием delim в качестве разделителя

Экземплярные методы класса String

Отметим, что все экземплярные методы не меняют строку, как это может показаться на первый взгляд, а при необходимости возвращают измененную строку. Кроме того, считается, что символы в сроке индексируются с нуля.

Метод	Описание
`Contains(s: string): boolean`	Возвращает True, если текущая строка содержит s, и False в противном случае
`EndsWith(s: string): boolean`	Возвращает True, если текущая строка заканчивается на s, и False в противном случае
`IndexOf(s: string): integer`	Возвращает индекс первого вхождения подстроки s в текущую строку
`IndexOfAny(cc: array of char): integer`	Возвращает индекс первого вхождения любого символа из массива cc
`Insert(from: integer; s: string): string`	Возвращает строку, полученную из исходной строки вставкой подстроки s в позицию from
`LastIndexOf(s: string): integer`	Возвращает индекс последнего вхождения подстроки s в текущую строку
`LastIndexOfAny(a: array of char): integer`	Возвращает индекс последнего вхождения любого символа из массива cc
`PadLeft(n: integer): string`	Возвращает строку, полученную из исходной строки выравниванием по правому краю с заполнением пробелами слева до длины n
`PadRight(n: integer): string`	Возвращает строку, полученную из исходной строки выравниванием по левому краю с заполнением пробелами справа до длины n
`Remove(from,len: integer): string`	Возвращает строку, полученную

`Remove(from,len: integer): string`	Возвращает строку, полученную из исходной строки удалением len симолов с позиции from
`Replace(s1,s2: string): string`	Возвращает строку, полученную из исходной строки заменой всех вхождений подстроки s1 на строку s2
`Split(delim: char := ' '): array of string`	Возвращает массив строк, полученный расщеплением исходной строки на слова, при этом в качестве разделителей используется символ delim
`StartsWith(s: string): boolean`	Возвращает True, если текущая строка начинается на s, и False в противном случае
`Substring(from,len: integer): string`	Возвращает подстроку исходной строки с позиции from длины len
`ToCharArray: array of char`	Возвращает динамический массив символов исходной строки
`ToLower: string`	Возвращает строку, приведенную к нижнему регистру
`ToUpper: string`	Возвращает строку, приведенную к верхнему регистру
`Trim: string`	Возвращает строку, полученную из исходной удалением лидирующих и завершающих пробелов
`TrimEnd(params cc: array of char): string`	Возвращает строку, полученную из исходной удалением завершающих символов из массива cc
`TrimStart(params cc: array of char): string`	Возвращает строку, полученную из исходной удалением лидирующих символов из массива cc

Массивы

Массив представляет собой набор элементов одного типа, каждый из которых имеет свой номер, называемый *индексом* (индексов может быть несколько, тогда массив называется *многомерным*).

Массивы в **PascalABC.NET** делятся на статические и динамические.

При выходе за границы изменения индекса в **PascalABC.NET** всегда генерируется исключение.

Статические массивы

Тип статического массива конструируется следующим образом:
array [тип индекса1, ..., тип индексаN] **of** базовый тип

Тип индекса должен быть порядковым. Обычно тип индекса является диапазонным и представляется в виде a..b, где a и b - константные выражения целого, символьного или перечислимого типа. Например:

```
type
  MyEnum = (w1,w2,w3,w4,w5);
  Arr = array [1..10] of integer;
var
  a1,a2: Arr;
  b: array ['a'..'z',w2..w4] of string;
  c: array [1..3] of array [1..4] of real;
```

При описании можно также задавать инициализацию массива значениями:

```
var
  a: Arr := (1,2,3,4,5,6,7,8,9,0);
  cc: array [1..3,1..4] of real := ((1,2,3,4), (5,6,7,8), (9,0,1,2));
```

Статические массивы одного типа можно присваивать друг другу, при этом будет производиться копирование содержимого одного массива в другой:

```
a1:=a2;
```

При передаче статического массива в подпрограмму по значению также производится копирование содержимого массива - фактического параметра в массив - формальный параметр:

```
procedure p(a: Arr);
...
p(a1);
```

Как правило, в этой ситуации копирование не требуется, поэтому статический массив рекомендуется передавать по ссылке:

```
procedure p1(var a: Arr);
...
r(a1);
```

Динамические массивы

Тип динамического массива конструируется следующим образом:

array of *тип элементов* (одномерный массив)

array [] **of** *тип элементов* (одномерный массив)

array [,] **of** *тип элементов* (двумерный массив)

и т.д.

Переменная типа динамический массив представляет собой ссылку. Поэтому динамический массив нуждается в инициализации (выделении памяти под элементы).

Для выделения памяти под динамический массив используется два способа.

Первый способ использует операцию new в стиле вызова конструктора класса:

```
var
  a: array of integer;
  b: array [,] of real;
begin
  a := new integer[5];
  b := new real[4,3];
end.
```

В этом случае можно также задавать инициализацию массива значениями:

```
a := new integer[3](1,2,3);
b := new real[4,3] ((1,2,3),(4,5,6),(7,8,9),(0,1,2));
```

Второй способ использует стандартную процедуру SetLength:

```
SetLength(a,10);
SetLength(b,5,3);
```

Процедура SetLength обладает тем преимуществом, что при ее повторном вызове старое содержимое массива сохраняется.

Если объявлен массив массивов

```
var c: array of array of integer;
```

то его инициализацию можно провести только с помощью SetLength:

```
SetLength(c,5);
for i := 0 to 4 do
  SetLength(c[i],3);
```

Для инициализации такого массива с помощью new следует ввести имя типа для **array of** integer:

```
type IntArray = array of integer;
var c: array of IntArray;
...
c := new IntArray[5];
for i := 0 to 4 do
  c[i] := new integer[3];
```

При описании инициализацию динамического массива можно проводить в сокращенной форме:

```
var
  a: array of integer := (1,2,3);
  b: array [,] of real := ((1,2,3),(4,5,6),(7,8,9),(0,1,2));
  c: array of array of integer := ((1,2,3),(4,5),(6,7,8));
```

При этом происходит выделение памяти под указанное справа количество элементов. При таком способе в инициализаторе одномерного массива должно содержаться не менее 2 элементов.

Динамические массивы одного типа можно присваивать друг другу, при этом обе переменные-ссылки будут указывать на одну память:

```
var a1: array of integer;
var a2: array of integer;
a1 := a2;
```

Следует обратить внимание, что для динамических массивов принята структурная эквивалентность типов: можно присваивать друг другу и передавать в качестве параметров подпрограмм динамические массивы, совпадающие по структуре.

Чтобы одному динамическому массиву присвоить копию другого массива, следует воспользоваться стандартной функцией Copy:

```
a1 := Copy(a2);
```

Длина массива (количество элементов в нем) возвращается стандартной функцией Length или свойством Length:

```
l := Length(a);
l := a.Length;
```

Для многомерных массивов длина по каждой размерности возвращается стандартной функцией Length с двумя параметрами или методом GetLength(i):

```
l := Length(a,0);
l := a.GetLength(0);
```

Указатели

Указатель - это ячейка памяти, хранящая адрес. В **PascalABC.NET** указатели делятся на ***типизированные*** (содержат адрес ячейки памяти данного типа) и ***бестиповые*** (содержат адрес оперативной памяти, не связанный с данными какого-либо определенного типа).

Тип указателя на тип T имеет форму ^T, например:

```
type pinteger = ^integer;
var p: ^record r,i: real end;
```

Бестиповой указатель описывается с помощью слова pointer.

Для доступа к ячейке памяти, адрес которой хранит типизированный указатель, используется ***операция разыменования*** ^:

```
var
  i: integer;
  pi: ^integer;
...
pi := @i; // указателю присвоили адрес переменной i
pi^ := 5; // переменной i присвоили 5
```

Операция разыменования не может быть применена к бестиповому указателю.

Типизированный указатель может быть неявно преобразован к бестиповому:

```
var
  p: pointer;
  pr: ^real;
...
p := pr;
```

Обратное преобразование также может быть выполнено неявно:

```
pr := p;
pr^ := 3.14;
```

Указатели можно сравнивать на равенство (=) и неравенство (<>). Для того чтобы отметить тот факт, что указатель никуда не указывает, используется стандартная константа **nil** (нулевой указатель) : p := **nil**.

Внимание! Ввиду особенностей платформы .NET тип T типизированного указателя не должен быть ссылочным или содержать ссылочные типы на каком-то уровне (например, запрещены указатели на записи, у которых одно из полей имеет ссылочный тип). Причина такого ограничения проста: указатели реализуются неуправляемым кодом, который не управляется сборщиком мусора. Если в памяти, на которую указывает указатель, содержатся ссылки на управляемые переменные, то они становятся недействительными после очередной сборки мусора. Исключение составляют динамические массивы и строки, обрабатываемые особым образом. То есть, можно делать указатели на записи, содержащие в качестве полей строки и динамические массивы.

Записи

Запись представляет собой набор элементов разных типов, каждый из которых имеет свое имя и называется полем записи. Тип записи конструируется следующим образом:

record
  список полей
  список методов
end

Приведем пример записи, содержащей только поля:

```
type
  SexType = (male, female);
  Person = record
    Name: string;
    Age, Weight: integer;
    Sex: SexType;
  end;
```

При описании переменной или константы типа запись можно использовать инициализатор записи (как и в Object Pascal):

```
const p: Person = (Name: 'Петрова'; Age: 18; Weight: 55; Sex: female);
var p: Person := (Name: 'Иванов'; Age: 20; Weight: 80; Sex: male);
```

Присваивание записей копирует содержимое полей одной переменной-записи в другую:

```
d2 := d1;
```

Для записей принята именная эквивалентность типов: можно присваивать друг другу и передавать в качестве параметров подпрограмм записи, совпадающие только по имени.

В отличие от Object Pascal, в **PascalABC.NET** отсутствуют записи с вариантами.

Помимо полей, внутри записей могут содержаться также методы и свойства. Таким образом, записи очень близки к классам. Список отличий между записями и классами приводятся ниже:

1. Запись представляет собой размерный тип (переменные типа запись располагаются на стеке).

2.Все члены записей – публичные, модификаторы доступа внутри записей запрещены.

3.Записи нельзя наследовать; от записей также нельзя наследовать (отметим, что записи, тем не менее, могут реализовывать интерфейсы). В .NET тип записи неявно предполагается наследником типа System.ValueType и реализуется struct-типом.

Конструкторы для записей имеют тот же синтаксис, что и для классов. Однако, в отличие от классов, вызов конструктора записи не создает новый объект в динамической памяти, а только инициализирует поля записи.

По умолчанию процедура write для переменной типа запись выводит ее тип. Чтобы изменить это поведение, в записи следует переопределить метод ToString класса Object.

Например:

```
type
  SexType = (male, female);
  Person = record
    Name: string;
    Age, Weight: integer;
    Sex: SexType;
    constructor Create(Name: string; Age, Weight: integer; Sex: SexType);
    begin
      Self.Name := Name;
      Self.Age := Age;
      Self.Weight := Weight;
      Self.Sex := Sex;
    end;
    function ToString: string; override;
    begin
      Result := string.Format('Имя: {0} Возраст: {1} Вес: {2} Пол: {3}',
                        Name, Age, Weight, Sex);
    end
  end;
  ...
var p: Person := new Person('Иванов',20,70,Sex);
writeln(p);
```

Множества

Множество представляет собой набор элементов одного типа. Элементы множества считаются неупорядоченными; каждый элемент может входить во множество не более одного раза. Тип множества описывается следующим образом:

set of базовый тип

В качестве базового может быть ***любой*** тип, в том числе строковый и классовый.

Например:

```
type
  ByteSet = set of byte;
  StringSet = set of string;
  Digits = set of '0'..'9';
  SeasonSet = set of (Winter,Spring,Summer,Autumn);
  PersonSet = set of Person;
```

Элементы базового типа сравниваются на равенство следующим образом: у простых типов, строк и указателей сравниваются значения, у структурированных и у классов - значения всех элементов или полей. Однако, если поля относятся к ссылочному типу, то сравниваются только их адреса (неглубокое сравнение).

Переменная типа множество может содержать несколько значений базового типа. Чтобы сконструировать значение типа множество, используется конструкция вида

[список значений]

где в списке могут перечисляться через запятую либо выражения базового типа, либо (для порядковых типов) их диапазоны в виде a..b, где a и b - выражения базового типа. Например:

```
var
  bs: ByteSet := [1,3,5,20..25];
  fios: StringSet := ['Иванов','Петров','Сидорова'];
```

Значения в списке могут отсутствовать, тогда множество является пустым:

```
bs:=[];
```

Пустое множество совместимо по присваиванию с множеством любого типа.

Для множеств имеет место структурная эквивалентность типов.

Множества целых и множества на базе типа и его диапазонного подтипа или на базе двух диапазонных типов одного базового типа неявно преобразуются друг к другу. Если при присваивании s := s1 во множестве s1 содержатся элементы, которые не входят в диапазон значений базового типа для множества s, то они отсекаются.

Например:

```
var st: set of 3..9;
...
st := [1..5,8,10,12]; // в st попадут значения [3..5,8]
```

Операция **in** проверяет принадлежность элемента множеству:

```
if Wed in bestdays then ...
```

Для множеств определены операции + (объединение), - (разность), * (пересечение), = (равенство), <> (неравенство), <= (нестрогое вложение), < (строгое вложение), >= (нестрого содержит) и > (строго содержит).

Процедура write при выводе множества выводит все его элементы:

```
write(['Иванов','Петров','Сидорова']);
```

выведет ['Иванов','Петров','Сидорова'], при этом данные, если это возможно, будут отсортированы по возрастанию.

Для перебора всех элементов множества можно использовать цикл **foreach**, данные перебираются в некотором внутреннем порядке:

```
foreach s: string in fios do
  write(s,' ');
```

Для добавления элемента x к множеству s используется конструкция s += [x] или стандартная процедура Include: Include(s,x). Для удаления элемента x из множества s используется конструкция s -= [x] или стандартная процедура Exclude: Exclude(s,x).

Процедурный тип

Переменные, предназначенные для хранения процедур и функций, называются ***процедурными***. Тип процедурной переменной представляет собой заголовок процедуры или функции без имени. Например:

```
type
  procI = procedure(i: integer);
  funI = function: integer;
```

Процедурной переменной можно присвоить процедуру или функцию с совместимым типом:

```
procedure my(i: integer);
begin
  ...
end;
function f: integer;
begin

end;
...
var
  p1: procI;
  f1: funI;
...
p1 := my;
f1 := f;
```

После этого можно вызвать процедуру или функцию через эту процедурную переменную, пользуясь обычным синтаксисом вызова:

```
p1(5);
write(f1);
```

Для процедурных переменных принята структурная эквивалентность типов: можно присваивать друг другу и передавать в качестве параметров процедурные переменные, совпадающие по структуре (типы и количество параметров, тип возвращаемого значения).

Обычно процедурные переменные передаются как параметры для реализации *обратного вызова*:

```
procedure forall(var a: array of real; p: procedure(var r: real));
begin
  for var i := 0 to a.Length-1 do
    p(a[i]);
end;
procedure mult2(var r: real);
begin
  r := 2*r;
end;
procedure add3(var r: real);
begin
  r := r + 3;
end;
procedure print(var r: real);
begin
  write(r,' ');
end;
...
forall(a,mult2); // умножение элементов массива на 2
forall(a,add3);  // увеличение элементов массива на 3
forall(a,print); // вывод элементов массива
```

Процедурная переменная может хранить нулевое значение, которое задается константой nil. Вызов подпрограммы через нулевую процедурную переменную приводит к ошибке.

Процедурные переменные реализуются через делегаты .NET. Это означает, что они могут хранить несколько подпрограмм. Для добавления/отсоединения подпрограмм используются операторы += и -=:

```
p1 += mult2;
p1 += add3;
forall(a,p1);
```

Подпрограммы в этом случае вызываются в порядке прикрепления: вначале умножение, потом сложение.

Отсоединение неприкрепленных подпрограмм не выполняет никаких действий:

```
p1 -= print;
```

Кроме того, к процедурной переменной можно прикреплять/откреплять статические и экземплярные методы классов.

Файловые типы

Файл представляет собой последовательность элементов одного типа, хранящихся на диске. В **PascalABC.NET** имеется два типа файлов - *двоичные* и *текстовые*. Текстовые файлы хранят символы, разделенные на строки символами #13#10 (Windows) и символом #10 (Linux). Двоичные файлы в свою очередь делятся на типизированные и бестиповые.

Для описания текстового файла используется стандартное имя типа text, бестиповые файлы имеют тип **file**, а для описания типизированного файла используется конструкция **file of** тип элементов:

```
var
  f1: file of real;
  f2: text;
  f3: file;
```

В качества типа элементов в типизированном файле не могут фигурировать указатели, ссылочные типы, а также тип записи, содержащий ссылочные поля или указатели.

Стандартные файловые процедуры и функции описываются в пункте Процедуры и функции для работы с файлами.

Кроме того, в .NET имеется ряд классов, связанных с работой с файлами.

Эквивалентность и совместимость типов

Совпадение типов

Говорят, что типы T1 и T2 совпадают, если они имеют одно имя либо же определены в секции **type** в виде T1 = T2. Таким образом, в описаниях

```
type
  IntArray = array [1..10] of integer;
  IntArrayCopy = IntArray;
var
  a1: IntArray;
  a2: IntArrayCopy;
  b1,c1: array [1..15] of integer;
  b2: array [1..15] of integer;
```

переменные a1 и a2 и переменные b1 и c1 имеют один и тот же тип, а переменные b1 и b2 - разные типы.

Эквивалентность типов

Говорят, что типы T1 и T2 эквивалентны, если выполняется одно из следующих условий:

1. T1 и T2 совпадают
2. T1 и T2 - динамические массивы с совпадающими типами элементов
3. T1 и T2 - указатели с совпадающими базовыми типами
4. T1 и T2 - множества с совпадающими базовыми типами
5. T1 и T2 - процедурные типы с совпадающим списком формальных параметров (и типом возвращаемого значения - для функций)

Если типы эквивалентны только если их имена совпадают, то говорят, что имеет место ***именная эквивалентность типов***.

Если типы эквивалентны если они совпадают по структуре, то говорят, что имеет место ***структурная эквивалентность типов***. Таким образом, в **PascalABC.NET** имеет место именная эквивалентность для всех типов, кроме динамических массивов, множеств, типизированных указателей и процедурных типов, для которых имеет место структурная эквивалентность типов.

Только если типы T1 и T2 эквивалентны, фактический параметр типа T1 может быть подставлен вместо формального параметра-переменной типа T2.

Совместимость типов

Говорят, что типы T1 и T2 совместимы, если выполняется одно из следующих условий:

1. T1 и T2 эквивалентны
2. T1 и T2 принадлежат к целым типам
3. T1 и T2 принадлежат к вещественным типам
4. Один из типов - поддиапазон другого или оба - поддиапазоны некоторого типа
5. T1 и T2 - множества с совместимыми базовыми типами

Совместимость типов по присваиванию

Говорят, что значение типа T2 можно присвоить переменной типа T1 или тип T2 совместим по присваиванию с типом T1, если выполняется одно из следующих условий:

T1 и T2 совместимы

T1 - вещественного типа, T2 - целого

T1 - строкового типа, T2 - символьного

T1 - pointer, T2 - типизированный указатель

T1 - указатель или процедурная переменная, T2=nil

T1 - процедурная переменная, T2 - имя процедуры или функции с соответствующим списком параметров

T1, T2 - классовые типы, один из них - наследник другого. Поскольку в **PascalABC.NET** все типы кроме указателей являются потомками типа Object, то значение любого типа (кроме указателей) можно присвоить переменной типа Object

T1 - тип интерфейса, T2 - тип класса, реализующего этот интерфейс

Если тип T2 совместим по присваиванию с типом T1, то говорят также, что тип T2 ***неявно приводится*** к типу T1.

// **НеРеализовано** Если при приведении типа происходит выход за диапазон значений типа-результата, то при отключенной директиве компилятора #rangecheck off (по умолчанию) значения приводимого типа усекаются до значений типа-результата, при установленной директиве компилятора #rangecheck on генерируется исключение.

Отображение на типы .NET

Стандартные типы **PascalABC.NET** реализуются типами библиотеки классов .NET. Далее приводится таблица соответствий стандартных типов **PascalABC.NET** и типов .NET.

Тип PascalABC.NET	Тип .NET
int64	System.Int64
uint64	System.UInt64
integer, longint	System.Int32
longword, cardinal	System.UInt32
smallint	System.Int16
word	System.UInt16
shortint	System.SByte
byte	System.Byte
boolean	System.Boolean
real	System.Double
double	System.Double
char	System.Char
string	System.String
object	System.Object
array of T	T[]
record	struct

Арифметические операции

К *арифметическим* относятся бинарные операции +, -, *, /, +=, -=, *=, /= для вещественных и целых чисел, бинарные операции div и mod для целых чисел и унарные операции + и - для вещественных и целых чисел. Тип выражения `x op y`, где *op* - знак бинарной операции +, - или *, определяется из следующей таблицы:

	shortint	byte	smallint	word	integer	longword	int64	uint64	single	real
shortint	integer	integer	integer	integer	integer	int64	int64	uint64	single	real
byte	integer	integer	integer	integer	integer	longword	int64	uint64	single	real
smallint	integer	integer	integer	integer	integer	int64	int64	uint64	single	real
word	integer	integer	integer	integer	integer	longword	int64	uint64	single	real
integer	integer	integer	integer	integer	integer	int64	int64	uint64	single	real
longword	int64	longword	int64	longword	int64	longword	uint64	uint64	single	real
int64	int64	int64	int64	int64	int64	uint64	int64	uint64	single	real
uint64	uint64	uint64	uint64	uint64	uint64	uint64	uint64	uint64	single	real
single	single	single	single	single	single	single	single	single	single	real
real	real	real	real	real	real	real	real	real	real	real

То есть, если операнды - целые, то результатом является самый короткий целый тип, требуемый для представления всех получаемых значений.

При выполнении бинарной операции с `uint64` и знаковым целым результирующим типом будет `uint64`, при этом может произойти переполнение, не вызывающее исключения.

Для операции / данная таблица исправляется следующим образом: результат деления любого целого на целое имеет тип real.
Для операций **div** и **mod** выполняются эти же правила, но операнды могут быть только целыми. Правила вычисления операций **div** и **mod** - следующие:
x **div** y - результат целочисленного деления x на y. Точнее, x **div** y = x / y, округленное до ближайшего целого по направлению к 0;
x **mod** y - остаток от целочисленного деления x на y. Точнее, x **mod** y = x - (x **div** y) * y.
Унарная арифметическая операция + для любого целого типа возвращает этот тип. Унарная арифметическая операция - возвращает для целых типов, меньших или равных integer, значение типа integer, для longword и int64 - значение типа int64, к uint64 унарная операция - не применима, для типов single и real - соответственно типы single и real. То есть так же результатом является самый короткий тип, требуемый для представления всех получаемых значений.
Бинарные операции +=, -=, *=, /= не возвращают результат. Они используются в операторах присваивания и имеют следующий смысл: a #= b означает a := a # b, где # - знак операции +, -, *, /.
Например:
a += 3; // увеличить a на 3
b *= 2; // увеличить b в 2 раза
Операция /= неприменима, если левый операнд - целый.
Операции +=, -=, *=, /= могут также использоваться со свойствами классов соответствующих типов в левой части.

Логические операции

К *логическим* относятся бинарные операции **and**, **or** и **xor**, а также унарная операция **not**, имеющие операнды типа boolean и возвращающие значение типа boolean. эти операции подчиняются стандартным правилам логики: a **and** b истинно только тогда, когда истинны a и b, a **or** b истинно только тогда, когда истинно либо a, либо b, a **xor** b истинно только тогда, когда только одно из a и b истинно, **not** a истинно только тогда, когда a ложно.

Выражения с **and** и **or** вычисляются по ***короткой схеме***:

в выражении x **and** y если x ложно, то все выражение ложно, и y не вычисляется;

в выражении x **or** y если x истинно, то все выражение истинно, и y не вычисляется.

Побитовые операции

К *побитовым* относятся бинарные операции **and**, **or**, **not**, **xor**, **shl**, **shr**. Они производят побитовые манипуляции с операндами целого типа. Результирующий тип для **and**, **or**, **xor** будет наименьшим целым, включающим все возможные значения обоих типов операндов. Для **shl**, **shr** результирующий тип совпадает с типом левого операнда, для **not** - с типом операнда.
Побитовые операции осуществляются следующим образом: с каждым битом (0 принимается за False, 1 - за True) производится соответствующая логическая операция. Например:
00010101 **and** 00011001 = 00010001
00010101 **or** 00011001 = 00011101
00010101 **xor** 00011001 = 00001100
not 00010101 = 11101010
(операнды и результат представлены в двоичной форме).

Операции сравнения

Операции сравнения <, >, <=, >=, =, <> возвращают значение типа boolean и применяются к операндам простого типа.
Операции = и <> также применяются ко всем типам (!). Для размерных типов по умолчанию сравниваются значения, для ссылочных типов - адреса. Можно переопределить это поведение, перегрузив операции = и <>. Аналогично можно перегрузить все операции сравнения для типов записей и классов, вводимых пользователем.

Строковые операции

К строкам применимы все операции сравнения <, >, <=, >=, =, <>. Кроме этого, к строкам и символам применима операция конкатенации (слияния) +, ее результат имеет строковый тип.
Например, 'a'+'b'='ab'.
К строкам также применима операция +=:
s += s1;
означает
s := s + s1;

Операции с указателями

Ко всем указателям применимы операции сравнения = и <>.
К типизированным указателям применима ***операция разыменования*** ^: если p является указателем на тип T, то p^ - элемент типа T, на который указывает p.
Указатели pointer разыменовывать нельзя.

Операции с множествами

К множествам с базовыми элементами одного типа применимы операции + (объединение), - (разность) и * (пересечение), а также +=, -= и *=:

```
var s1,s2,s: set of byte;
begin
  s1 := [1..4];
  s2 := [2..5];
  s := s1 + s2; // s = [1..5]
  s := s1 - s2; // s = [1]
  s := s1 * s2; // s = [2..4]
  s += [3..6];  // s = [2..6]
  s -= [3];     // s = [2,4..6]
  s *= [1..5];  // s = [2,4..5]
end.
```

К множествам с базовыми элементами одного типа применимы также операции сравнения = (равенство), <> (неравенство), <= (нестрого вложено), < (строго вложено), >= (нестрого содержит) и > (строго содержит):

```
[1..3] = [1,2,3]
['a'..'z'] <> ['0'..'9']
[2..4] < [1..5]
[1..5] <= [1..5]
[1..5] > [2..4]
[1..5] >= [1..5]
```

Но неверно, что [1..5] < [1..5].

Наконец, операция **in** определяет, принадлежит ли элемент множеству: 3 **in** [2..5] - верно, 1 **in** [2..5] неверно.

Операция @

Операция @ применяется к переменной и возвращает ее адрес. Тип результата представляет собой типизированный указатель на тип переменной. Например:

```
var
  r: real;
  pr: ^real := @r;
```

Операции is и as

Операция **is** предназначена для проверки того, имеет ли классовая переменная указанный динамический тип. Операция **as** позволяет безопасно преобразовать переменную одного классового типа к другому классовому типу (в отличие от явного приведения классового типа).

Операция **is** имеет вид:

a **is** *ClassType*

и возвращает True если a принадлежит к классу *ClassType* или одному из его потомков.

Например, если Base и Derived - классы, причем, Derived - потомок Base, переменные b и d имеют соответственно типы Base и Derived, то выражения b **is** Base и d **is** Base возвращают True, а b **is** Derived - False.

Операция **as** имеет вид:

a **as** *ClassType*

и возвращает ссылку на объект типа *ClassType* если преобразование возможно, в противном случае возвращает **nil**.

Например, в программе

```
type
  Base = class
  end;
  Derived = class(Base)
    procedure p;
    begin
    end;
  end;
var b: Base;
begin
  b := new Base;
  writeln(b is Derived);
  b := new Derived;
  writeln(b is Derived);
end.
```

первый раз выводится False, второй - True.

Операции **is** и **as** используются для работы с переменной базового класса, содержащей объект производного класса.

1 способ.

```
if b is Derived then
  Derived(b).p;
```

2 способ.

```
var d: Derived := b as Derived;
d.p;
```

Операция new

Операция **new** имеет вид:

new *ИмяКласса*(*ПараметрыКонструктора*)

Она вызывает конструктор класса *ИмяКласса* и возвращает созданный объект.

Например:

```
type
  My = class
    constructor Create(i: integer);
    begin
    end;
  end;
var m: My := new My(5);
```

Эквивалентным способом создания объекта является вызов конструктора в стиле Object Pacal:

```
var m: My := My.Create(5);
```

Создание объекта класса при инициализации переменной проще проводить, используя ***автоопределение типа***:

```
var m := new My(5);
```

Операции typeof и sizeof

Операция **sizeof**(*имя типа*) возвращает для этого типа его размер в байтах.
Операция **typeof**(*имя типа*) возвращает для этого типа объект класса System.Type. Приведем пример использования typeof:

```
type
  Base = class ... end;
  Derived = class(Base) ... end;
var b: Base := new Derived;
begin
  writeln(b.GetType = typeof(Derived));
end.
```

Операция явного приведения типов

Операция явного приведения типов имеет вид
ИмяТипа(*выражение*)
и позволяет преобразовать выражение к типу *ИмяТипа*. Тип выражения и тип с именем *ИмяТипа* должны оба принадлежать либо к порядковому типу, либо к типу указателя, либо один тип должен быть наследником другого, либо тип выражения должен поддерживать интерфейс с именем *ИмяТипа*. В случае указателей запрещено преобразовывать типизированный указатель к типу указателя на другой тип.

Пример.

```
type
  pinteger = ^integer;
  Season = (Winter,Spring,Summer,Autumn);
var i: integer;
    b: byte;
    p: pointer := @i;
    s: Season;
begin
  i := integer('z');
  b := byte(i);
  i := pinteger(p);
  s := Season(1);
end.
```

// **НеРеализовано** Как и для неявного приведения типов, если при приведении типа происходит выход за диапазон значений типа-результата, то при отключенной директиве компилятора #rangecheck off (по умолчанию) значения приводимого типа усекаются до значений типа-результата, при установленной директиве компилятора #rangecheck on генерируется исключение.

Приоритет операций

Приоритет определяет порядок выполнения операций в выражении. Первыми выполняются операции, имеющие высший приоритет. Операции, имеющие одинаковый приоритет, выполняются слева направо.

Таблица приоритетов операций

@, **not**, ^, +, - (унарные), new	1 (наивысший)
*, /, **div**, **mod**, **and**, **shl**, **shr**, **as**	2
+, - (бинарные), **or**, **xor**	3
=, <>, <, >, <=, >=, **in**, **is**, +=, -=, *=, /=	4 (низший)

Для изменения порядка выполнения операций в выражениях используются скобки.

Оператор присваивания

Оператор присваивания имеет вид:
переменная := выражение
В качестве переменной может быть простая переменная, разыменованный указатель, переменная с индексами или компонент переменной типа запись.
Символ := называется значком присваивания. Выражение должно быть совместимо по присваиванию с переменной.
Оператор присваивания заменяет текущее значение переменной значением выражения.
Например:
i := i + 1; // увеличивает значение переменной i на 1
В **PascalABC.NET** определены также операторы присваивания со значками +=, -=, *=, /=, которые трактуются как значки операций. Данные операции не возвращают значений, но изменяют переменную - левый операнд. Их действие для процедурных переменных описано здесь. Для числовых типов действие данных операций описано здесь. Кроме того, использование операции += для строк описано здесь и операций +=, -= и *= для множеств - здесь.

Составной оператор (блок)

Составной оператор предназначен для объединения нескольких операторов в один. Он имеет вид:

```
begin
  операторы
end
```

В **PascalABC.NET** составной оператор также называется ***блоком***. (традиционно в Паскале блоком называется раздел описаний, после которого идет составной оператор; в **PascalABC.NET** принято другое решение, поскольку можно описывать переменные непосредственно внутри составного оператора). Операторы отделяются один от другого символом ";". Служебные слова **begin** и **end**, окаймляющие операторы, называются ***операторными скобками***.

Например:

```
s:=0; p:=1;
for i:=1 to 10 do
begin
  p:=p*i;
  s:=s+p
end
```

Перед **end** также может ставиться ";". В этом случае считается, что последним оператором перед **end** является пустой оператор, не выполняющий никаких действий.

Помимо операторов, в блоке могут быть внутриблочные описания переменных:

```
program MyProgram;
begin
  var a,b: integer;
  var r: real;
  readln(a,b);
  x := a/b;
  writeln(x);
end;
```

Пустой оператор

Пустой оператор не включает никаких символов, не выполняет никаких действий и используется в двух случаях: 1. Для использования символа ";" после последнего оператора в блоке:

```
begin
  a := 1;
  b := a;
end
```

Поскольку в языке Паскаль символ ";" разделяет операторы, то в приведенном выше коде считается, что после последней ";" находится пустой оператор. Таким образом, ";" перед end в блоке можно либо ставить, либо нет.

1. Для пометки места, следующего за последним оператором в блоке::

```
label a;
begin
  goto a;
  x := 1;
a:
end
```

Условный оператор

Условный оператор имеет *полную* и *краткую* формы.
Полная форма условного оператора выглядит следующим образом:
if условие **then** оператор1
else оператор2
В качестве условия указывается некоторое логическое выражение. Если условие оказывается истинным, то выполняется оператор1, в противном случае выполняется оператор2.
Краткая форма условного оператора имеет вид:
if условие **then** оператор
Если условие оказывается истинным, то выполняется оператор, в противном случае происходит переход к следующему оператору программы.
В случае конструкции вида
if условие1 **then**
  if условие2 **then** оператор1
  else оператор2
else всегда относится к ближайшему предыдущему оператору **if**, для которого ветка **else** еще не указана. Если в предыдущем примере требуется, чтобы **else** относилась к первому оператору **if**, то необходимо использовать составной оператор:

```
if условие1 then
begin
  if условие2 then оператор1
end
else оператор2
```

Например:

```
if a<b then
  min := a
else min := b;
```

Оператор выбора

Оператор выбора выполняет одно действие из нескольких в зависимости от значения некоторого выражения, называемого *переключателем*. Он имеет следующий вид:

```
case переключатель of
  список выбора 1: оператор1;
  ...
  список выбора N: операторN;
  else оператор0
end;
```

Переключатель представляет собой выражение порядкового типа, а списки выбора содержат константы совместимого по присваиванию типа. Как и в операторе **if**, ветка **else** может отсутствовать.

Оператор **case** работает следующим образом. Если в одном из списков выбора найдено текущее значение переключателя, то выполняется оператор, соответствующий данному списку. Если же значение переключателя не найдено ни в одном списке, то выполняется оператор по ветке **else** или, если ветка **else** отсутствует, оператор **case** не выполняет никаких действий.

Список выбора состоит либо из одной константы, либо из диапазона значений вида a..b (константа a должна быть меньше константы b); можно также перечислить несколько констант или диапазонов через запятую:

```
case DayOfWeek of
  1..5: writeln('Будний день');
  6,7: writeln('Выходной день');
end;
```

Списки выбора не должны пересекаться. Например, следующий фрагмент

```
case i of
  2,5: write(1);
  4..6: write(2);
end;
```

приведет к ошибке компиляции.

Оператор цикла for

Оператор цикла **for** имеет одну из двух форм:
for переменная := начальное значение **to** конечное значение **do**
  оператор
или
for переменная := начальное значение **downto** конечное значение **do**
  оператор
Кроме того, переменную можно описать непосредственно в заголовке цикла:
for переменная: тип := начальное значение **to** или **downto** конечное значение **do**
  оператор
или
for var переменная := начальное значение **to** или **downto** конечное значение **do**
  оператор
В последнем случае используется автоопределение типа переменной по типу начального значения. В двух последних случаях область действия объявленной переменной распространяется до конца тела цикла, которое в данном случае образует неявный блок.
Текст от слова **for** до слова **do** включительно называется ***заголовком цикла***, а оператор после **do** - ***телом цикла***.
Переменная после слова **for** называется ***параметром цикла***. Для первой формы цикла с ключевым словом **to** параметр цикла меняется от начального значения до конечного значения, увеличиваясь всякий раз на единицу, а для второй формы ключевым словом **downto** - уменьшаясь на единицу. Для каждого значения переменной-параметра выполняется тело цикла. Однократное повторение тела цикла называется ***итерацией цикла***. Значение параметра цикла после завершения цикла считается неопределенным.
Переменная-параметр цикла может иметь любой порядковый тип. При этом начальное и конечное значения должны быть совместимы по присваиванию с переменной-параметром цикла.
Например:

```
var en: (red,green,blue,white);
...
for en := red to blue do
  write(Ord(en):2);
for var c := 'a' to 'z' do
  write(c);
```

Если для цикла **for** ... **to** начальное значение переменной цикла больше конечного значения или для цикла **for** ... **downto** начальное значение переменной цикла меньше конечного значения, то тело цикла не выполнится ни разу.

Если цикл используется в подпрограмме, то переменная-параметр цикла должна быть описана как локальная.

Изменение переменной-параметра цикла внутри цикла является логической ошибкой. Например, следующий фрагмент со вложенным оператором **for** является ошибочным:

```
for i := 1 to 10 do
  for i := 1 to 5 do
    write(i);
```

Оператор цикла foreach

Оператор цикла **foreach** имеет одну из двух форм:

foreach переменная **in** контейнер **do**
  оператор

или

foreach переменная: тип **in** контейнер **do**
  оператор

В качестве контейнера может фигурировать динамический массив, строка, множество, а также любой контейнер, удовлетворяющий интерфейсу IEnumerator (например, List<T>, Dictionary<T1,T2> и т.д.). Переменная цикла должна иметь тип, совместимый с элементами контейнера.

Переменная цикла пробегает все значения элементов контейнера, для каждого значения переменной цикла выполняется тело цикла. Изменение переменной цикла внутри тела цикла не меняет элементы контейнера, т.е. они доступны только на чтение.

Например:

```
var
  ss: set of string := ['Иванов','Петров','Сидоров'];
  a: array of integer := (3,4,5);
  b: array [1..5] of integer := (1,3,5,7,9);
  l := new List<real>;
begin
  foreach s: string in ss do
    write(s,' ');
  writeln;
  foreach x: integer in a do
    write(x,' ');
  writeln;
  foreach x: integer in b do
    write(x,' ');
  writeln;
  foreach r: real in l do
    write(r,' ');
end.
```

СПАСИБО ЗА ВНИМАНИЕ!