Содержание

Операторы ветвления

К операторам, позволяющим из нескольких возможных вариантов выполнения программы (ветвей) выбрать только один, относятся `if` и`case`.

Условный оператор if

Оператор `if` выбирает между двумя вариантами развития событий:

```
if <условие>
  then <один_оператор>
  [else <один_оператор>];
```

Обратите внимание, что перед словом `else` (когда оно присутствует, конечно же) символ ";" не ставится - ведь это разорвало бы оператор на две части.

Условный оператор `if` работает следующим образом:

1. Сначала вычисляется значение <условия> - это может быть любое выражение, возвращающее значение *типа boolean*.
2. Затем, если в результате получена "истина" (true), то выполняется оператор, стоящий после ключевого слова then, а если "ложь" (false) - без дополнительных проверок выполняется оператор, стоящий после ключевого слова `else`. Если же else-ветвь отсутствует, то не выполняется ничего.

Что же произойдет, если написать несколько вложенных операторов `if`?

В случае, когда каждый оператор `if` имеет собственную else-ветвь, все будет в порядке. А вот если некоторые из них этой ветви не имеют, может возникнуть ошибка. Компилятор языка *Pascal* всегда считает, что `else` относится к самому ближнему оператору `if`. Таким образом, если написать

```
if i>0 then if s>2
          then s:= 1
             else s:= -1;
```

подразумевая, что else-ветвь относится к внешнему оператору `if`, то компилятор все равно воспримет эту запись как

```
if i>0 then if s>2
         then s:= 1
         else s:= -1
      else;
```

Ясно, что таким образом правильного результата получить не удастся.

Для того чтобы избежать подобных ошибок, стоит всегда (или по крайней мере при наличии нескольких вложенных условных операторов) указывать оба ключевых слова, даже если одна из ветвей будет пустовать. Так вы застрахуетесь от одной из частых "ошибок по невнимательности", которые очень сложно найти в процессе отладки программы.

Итак, исходный вариант нужно переписать следующим образом:

```
if i>0 then if s>2
         then s:=1
         else
      else s:=-1;
```

либо так:

```
if i>0 then begin if s>2
                then s:=1
       end
      else s:=-1;
```

Вообще же, если есть возможность переписать несколько вложенных условных операторов как один оператор ***выбора***, это стоит сделать.

Оператор выбора case

Оператор `case` позволяет сделать выбор между несколькими вариантами:

```
case <переключатель> of
            <список_констант> : <один_оператор>;
            [<список_констант> : <один_оператор>;]
            [<список_констант> : <один_оператор>;]
```

```
            [else <один_оператор>;]
end;
```

Замечание: Обратите внимание, что после `else` двоеточие не ставится.

Существуют дополнительные правила, относящиеся к структуре этого оператора:

1. Переключатель должен относиться только к *порядковому типу* данных, но не к типу longint.
2. Переключатель может быть переменной или выражением.
3. Список констант может задаваться как явным перечислением, так и интервалом или их объединением.
4. Повторение констант не допускается.
5. Тип переключателя и типы всех констант должны быть совместимыми[1].

Пример *оператора выбора*:

```
case symbol(* :char *) of
   'a'..'z', 'A'..'Z' : writeln('Это латинская буква');
   'а'..'я', 'А'..'Я' : writeln('Это русская буква');
   '0'..'9' :           writeln('Это цифра');
   ' ',#10,#13,#26:     writeln('Это пробельный символ');
   else                 writeln('Это служебный символ');
end;
```

Выполнение оператора case происходит следующим образом:

1. вычисляется значение переключателя;
2. полученный результат проверяется на принадлежность к тому или иному списку констант;
3. если такой список найден, то дальнейшие проверки уже не производятся, а выполняется оператор, соответствующий выбранной ветви, после чего управление передается оператору, следующему за ключевым словом `end`, которое закрывает всю конструкцию case.
4. если подходящего списка констант нет, то выполняется оператор, стоящий за ключевым словом `else`. Если else-ветви нет, то не выполняется ничего.

Иллюстрация if и case

В качестве примера, иллюстрирующего использование операторов ***ветвления***, приведем несколько различных реализаций функции sgn(x)[2] - знак числа x. Из математики известно, что эта функция имеет следующие значения:

```
sgn(x) = -1, если  x < 0;
sgn(x) = 0, если  x = 0;
sgn(x) = 1, если  x > 0.
```

Реализовать эту функцию для случая, когда x вещественное, можно следующими способами (при условии, что x:real; sgn: -1..1;):

```
1.  if x=0 then sgn:= 0;
2.  if x<0 then sgn:= -1;
3.  if x>0 then sgn:= 1;
```

Это так называемая реализация "в лоб". Здесь нет никаких хитростей и никаких попыток оптимизации: даже если сработает первый вариант, второй и третий все равно будут проверены, невзирая на то, что результат уже получен.

```
4.  if x=0
5.   then sgn:= 0
6.   else if x<0 then sgn:= -1
7.                  else sgn:= 1;
```

Этот вариант свободен от излишних проверок в случае, если значение переменной не положительно.Эту реализацию следует признать более эффективной, чем предыдущая

```
8.  if x=0
9.   then sgn:=0
10.  else sgn:=x/abs(x);
```

Еще одна попытка сократить текст программы. Здесь используется стандартная функция *abs*(), которая возвращает абсолютное значение аргумента. Проблема в данном случае состоит в том, что "/" - деление дробное, но ведь нам необходим целый, а не вещественный ответ! "Давайте воспользуемся стандартной функцией округления", - скорее всего, скажет внимательный читатель.

```
11.  if x=0
12.   then sgn:=0
13.   else sgn:=round(x/abs(x));
```

И действительно, исправленный вариант будет выдавать верный результат.

```
14.  case x=0 of
15.   true:  sgn:=0;
16.   false:
17.     sgn:=round(x/abs(x));
18.  end;
```

А вот еще один (правда, несколько неестественный) способ с использованием *оператора выбора*. Вся хитрость этого варианта в том, что выбирающий ветви переключатель обязан принадлежать к *перечислимому типу*, именно поэтому пришлось заменить "x" на "x = 0". Напомним, что эта операция сравнения выдает результат *логического типа* boolean, и именно логические константы `true` и `false` фигурируют в качестве меток выбора.

Конечно же, мы перебрали далеко не все возможные способы реализации функции sgn(x) (ведь сколько людей, столько и способов выражать свои мысли - хоть в литературе, хоть в программировании). Однако уже на этом простеньком примере видно, что способов запрограммировать желаемое всегда больше, чем один, и вряд ли самое простое решение будет и оптимальным.

Массивы

Теперь мы приступаем к изучению массива - наиболее широко используемого *структурированного типа данных*, предназначенного для хранения нескольких однотипных элементов.

Описание массива

Для того чтобы задать массив, необходимо в разделе *описания переменных* (var) указать его размеры и тип его компонент.

Общий вид описания (одномерного) массива:

```
array[<тип_индексов> ][1)]  of <тип_компонент>;
```

Чаще всего это трактуется так:

```
array[<левая_граница>..<правая_граница>] of <тип_компонент>;
```

Например, одномерный (***линейный) массив***, состоящий не более чем из 10 целых чисел, можно описать следующим образом:

```
var a1: array [1..10] of integer;
```

Нумерация

Нумерация компонент массива не обязана начинаться с 1 или с 0 - вы можете описывать массив, пронумерованный любыми целыми числами. Необходимо лишь, чтобы номер последней компоненты был больше, чем номер первой:

```
var a1: array [-5..4] of integer;
```

Собственно говоря, нумеровать компоненты массива можно не только целыми числами. Любой *порядковый тип* данных (перечислимый, интервальный, символьный, логический, а также произвольный тип, созданный на их основе) имеет право выступать в роли нумератора. Таким образом, допустимы следующие описания массивов:

`type char = 'a','c'..'z';` (- отсутствует символ "b")

`var a1: array[char] of integer;` - 256 компонент

`a2: array [char] of integer;` - 256 целых компонент

`a3: array [shortint] of real;` - 256 вещественных компонент

Общий размер массива не должен превосходить 65 520 байт. Следовательно, попытка задать массив `a4:array[integer]of byte`; не увенчается успехом, поскольку *тип integer* покрывает 65 535 различных элементов. А про тип `longint` в данном случае лучше и вовсе не вспоминать.

Тип компонент массива может быть любым:

`var a4: array[10..20] of real;` - массив из компонент простого типа

`a5: array[0..100] of record1;` - массив из записей[2)]

`a6: array[-10..10] of ^string;` - массив из указателей[3)] на строки

`a7: array[-1..1] of file;` - массив из имен файловых переменных[4)]

`a8: array[1..100] of array[1..100] of char;` - *двумерный массив* (массив векторов)

Для краткости и удобства ***многомерные массивы*** можно описывать и более простым способом:

`var a9: array[1..10,1..20] of real;` - *двумерный массив* 10 x 20

`a10: array[boolean, -1..1,char, -10..10] of word;` - четырехмерный массив 2 x 3 x 256 x 21

Общее ограничение на размер массива - не более 65 520 байт - сохраняется и для многомерных массивов. Количество компонент многомерного массива вычисляется как произведение всех его "измерений". Таким образом, в массиве a9 содержится 200 компонент, а в массиве a10 - 32 256 компонент.

Описание переменных размерностей

Если ваша программа должна обрабатывать матрицы[5] переменных размерностей (скажем, N по горизонтали и M по вертикали), то вы столкнетесь с проблемой изначального задания массива, ведь в разделе var не допускается использование переменных. Следовательно, самый логичный, казалось бы, вариант

```
var m,n: integer;
    a: array[1..m,1..n] of real;
```

придется отбросить.

Если на этапе написания программы ничего нельзя сказать о предполагаемом размере входных данных, то не остается ничего другого, как воспользоваться техникой динамически распределяемой памяти (см. лекцию 10).

Предположим, однако, что вам известны максимальные границы, в которые могут попасть индексы обрабатываемого массива. Скажем, N и M заведомо не могут превосходить 100. Тогда можно выделить место под наибольший возможный массив, а реально работать только с малой его частью:

```
const nnn=100;
var a: array[1..nnn,1..nnn] of real;
    m,n: integer;
```

Обращение к компонентам массива

Массивы относятся к структурам прямого доступа. Это означает, что возможно напрямую (не перебирая предварительно все предшествующие компоненты) обратиться к любой интересующей нас компоненте массива.

Доступ к компонентам линейного массива осуществляется так[6]:

```
<имя_массива>[<индекс_компоненты>]
```

а многомерного - так:

```
<имя_массива>[<индекс>,_,<индекс>]
```

Правила употребления ***индексов*** при обращении к компонентам массива таковы:

1. Индекс компоненты может быть константой, переменной или выражением, куда входят операции и вызовы функций.
2. Тип каждого индекса должен быть совместим с типом, объявленным в описании массива именно для соответствующего "измерения"; менять индексы местами нельзя.
3. Количество индексов не должно превышать количество "измерений" массива. Попытка обратиться к линейному массиву как к многомерному обязательно вызовет ошибку. А вот обратная ситуация вполне возможна: например, если вы описали N-*мерный массив*, то его можно воспринимать как *линейный массив*, состоящий из (N-1)-мерных массивов.

Примеры использования компонент массива:

```
a2['z']:= a2['z']+1;
a3[-10]:= 2.5;
a3[i+j]:= a9[i,j];
a10[x>0,sgn(x),'!',abs(k*5)]:= 0;
```

Задание массива константой

Для того чтобы не вводить массивы вручную во время отладки программы (особенно если они имеют большую размерность), можно пользоваться не только файлами[7]. Существует и более простой способ, когда входные данные задаются прямо в тексте программы при помощи типизированных констант.

Если массив линейный (вектор), то начальные значения для компонент этого вектора задаются через запятую, а сам вектор заключается в круглые скобки.

Многомерный массив также можно рассматривать как линейный, предполагая, что его компонентами служат другие массивы. Таким образом, длясистемы вложенных векторов действует то же правило задания *типизированной константы*: каждый вектор ограничивается снаружи круглыми скобками.

Исключение составляют только массивы, компонентами которых являются величины *типа char*. Такие массивы можно задавать проще: строкой[8)]символов.

Примеры задания массивов типизированными константами:

```
type mass = array[1..3,1..2] of byte;
const a: array[-1..1] of byte = (0,0,0); {линейный}
b: mass = ((1,2),(3,4),(5,6)); {двумерный}
s: array[0..9] of char = '0123456789';
```

Замечание: Невозможно задать неименованную или нетипизированную константу, относящуюся к типу данных array.

Операторы циклов

Для того чтобы обработать несколько однотипных элементов, совершить несколько одинаковых действий и т.п., разумно воспользоваться *оператором цикла* - любым из четырех, который наилучшим образом подходит к поставленной задаче.

Оператор цикла повторяет некоторую *последовательность операторов* заданное число раз, которое может быть определено и динамически - уже во время работы программы.

Замечание: Алгоритмы, построенные только с использованием циклов, называются итеративными[1] - от слова***итерация***, которое обозначает повторяемую последовательность действий.

for-to и for-downto

В случае когда количество однотипных действий заранее известно (например, необходимо обработать все компоненты массива), стоит отдать предпочтение циклу с параметром (`for`).

Инкрементный цикл с параметром

Общий вид оператора `for-to`:

```
for i:= first to last do <оператор>;
```

Счетчик i (переменная), нижняя граница first (переменная, константа или выражение) и верхняя граница last (переменная, константа или выражение) должны относиться к эквивалентным *порядковым типам* данных. Если тип нижней или верхней границы не эквивалентен типу счетчика, а лишь совместим с ним, то осуществляется неявное приведение: значение границы преобразуется к типу счетчика, в результате чего возможны ошибки.

Цикл `for-to` работает следующим образом:

1. вычисляется значение верхней границы `last;`
2. переменной i присваивается значение *нижней границы* `first;`
3. производится проверка того, что `i<=last;`
4. если это так, то выполняется `<оператор>` ;
5. значение переменной `i` увеличивается на единицу;
6. пункты 3-5, составляющие одну итерацию цикла, выполняются до тех пор, пока `i` не станет строго больше, чем `last;` как только это произошло, выполнение цикла прекращается, а управление передается следующему за ним оператору.

Из этой последовательности действий можно понять, какое количество раз отработает цикл `for-to` в каждом из трех случаев:

- `first < last`: цикл будет работать `last-first+1` раз;
- `first = last`: цикл отработает ровно один раз;
- `first > last`: цикл вообще не будет работать.

После окончания работы цикла переменная-счетчик может потерять свое значение[2]. Таким образом, нельзя с уверенностью утверждать, что после того, как цикл завершил работу, обязательно окажется, что `i=last+1`. Поэтому попытки использовать переменную-счетчик сразу после завершения цикла (без присваивания ей какого-либо нового значения) могут привести к непредсказуемому поведению программы при отладке.

Декрементный цикл с параметром

Существует аналогичный *вариант цикла* for, который позволяет производить обработку не от меньшего к большему, а в противоположном направлении:

```
for i:= first downto last do <оператор>;
```

Счетчик `i` (переменная), верхняя граница first (переменная, константа или выражение) и нижняя граница `last` (переменная, константа или выражение) должны иметь эквивалентные порядковые типы. Если тип нижней или верхней границы не эквивалентен типу счетчика, а лишь совместим с ним, то осуществляется неявное *приведение типов*.

Цикл `for-downto` работает следующим образом:

1. переменной `i` присваивается значение `first` ;
2. производится проверка того, что `i>=last` ;
3. если это так, то выполняется `<оператор>` ;
4. значение переменной `i` уменьшается на единицу;
5. пункты 2-4 выполняются до тех пор, пока `i` не станет меньше, чем `last` ; как только это произошло, выполнение цикла прекращается, а управление передается следующему за ним оператору.

Если при этом

- `first < last`, то цикл вообще не будет работать;
- `first = last`, то цикл отработает один раз;
- `first > last`, то цикл будет работать `first-last+1` раз.

Замечание о неопределенности значения счетчика после окончания работы цикла справедливо и в этом случае.

while и repeat-until

Если заранее неизвестно, сколько раз необходимо выполнить *тело цикла*, то удобнее всего пользоваться ***циклом с предусловием*** (while) или ***циклом с постусловием*** (*repeat*-until).

Общий вид этих операторов таков:

```
while <условие_1> do <оператор>;
repeat <операторы> until <условие_2>;
```

Условие окончания цикла может быть выражено переменной, константой или выражением, имеющим *логический тип*.

Замечание: Обратите внимание, что на каждой итерации циклы for и while выполняют только по одному оператору (либо группу операторов, заключенную в *операторные скобки* begin-end и потому воспринимаемую как единый *составной оператор*). В отличие от них, цикл *`repeat`*-`until`позволяет выполнить сразу несколько операторов: ключевые слова *`repeat`* и `until` сами служат операторными скобками.

Так же, как циклы `for-to` и `for-downto`, циклы `while` и *`repeat`*-`until` можно назвать в некотором смысле противоположными друг другу.

Последовательности действий при выполнении этих циклов таковы:

Для while:	**Для *repeat*-until:**
1. Проверяется, истинно ли <условие_1>.	1. Выполняются <операторы>.
2. Если это так, то выполняется <оператор>.	2. Проверяется, ложно ли <условие_2>
3. Пункты 1 и 2 выполняются до тех пор, пока <условие_1> не станет ложным.	3. Пункты 1 и 2 выполняются до тех пор, пока <условие_2> не станет истинным.

Таким образом, если <условие_1> изначально ложно, то цикл `while` не выполнится ни разу. Если же <условие_2> изначально истинно, то цикл*`repeat`*-`until` выполнится один раз.

break и continue

Существует возможность[3)] прервать выполнение цикла (или одной его итерации), не дождавшись конца его (или ее) работы.

`break` прерывает работу всего цикла и передает управление на следующий за ним оператор.

`continue` прерывает работу текущей итерации цикла и передает управление следующей итерации (цикл *`repeat`*-`until`) или на предшествующую ей проверку (циклы `for-to`, `for-downto`, `while`).

Замечание: При прерывании работы циклов `for-to` и `for-downto` с помощью функции `break` переменная цикла (счетчик) сохраняет свое текущее значение, не "портится".

Оператор безусловного перехода goto

Возвращаясь к сказанному об операторе goto[41], необходимо отметить, что при всей его нежелательности все-таки существует ситуация, когда предпочтительно использовать именно этот оператор - как с точки зрения структурированности текста программы, так и с точки зрения логики ее построения, и уж тем более с точки зрения уменьшения трудозатрат программиста. Эта ситуация - необходимость передачи управления изнутри нескольких *вложенных циклов* на самый верхний уровень.

Дело в том, что процедуры break и continue прерывают только один цикл - тот, в теле которого они содержатся. Поэтому в упомянутой выше ситуации пришлось бы заметно усложнить текст программы, вводя много дополнительных прерываний. А один *оператор goto* способен заменить их все.

Сравните, например, два программно-эквивалентных отрывка:

```
write('Матрица ');
for i:=1 to n do
 begin
  flag:=false;
  for j:=1 to m do
    if a[i,j]>a[i,i]
      then begin flag:=true;
                 write('не ');
                 break;
           end
  if flag then break;
 end;
writeln('обладает свойством
        диагонального
        преобладания.');
```

```
write('Матрица ');
for i:=1 to n do
          for j:=1 to m do
           if a[i,j]>a[i,i]
      then begin
                  write('не ');
                   goto 1;
                 end;
   1: writeln('обладает
               свойством
               диагонального
               преобладания.');
```

Пример использования циклов

Задача. Вычислить *интеграл* в заданных границах a и b для некоторой гладкой функции f от одной переменной (с заданной точностью).

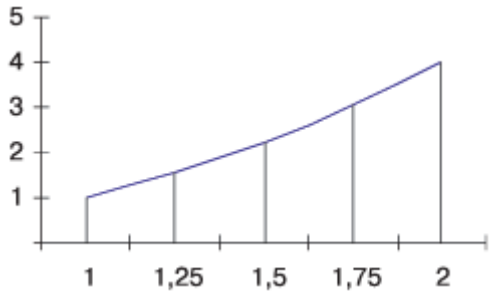
Алгоритм. Метод последовательных приближений, которым мы воспользуемся для решения этой задачи, состоит в многократном вычислении интеграла со все возрастающей точностью, - до тех пор, пока два последовательных результата не станут различаться менее чем на заданное число (скажем,*eps* = 0,001). Количество приближений нам заранее неизвестно (оно зависит от задаваемой точности), поэтому здесь годится только цикл с условием (любой из них).

Вычислять одно текущее значение для интеграла мы будем с помощью *метода прямоугольников*: разобьем отрезок [a,b] на несколько мелких частей, каждую из них дополним (или урежем - в зависимости от наклона *графика функции* на данном участке) до прямоугольника, а затем просуммируем получившиеся площади. Количество шагов нам известно, поэтому здесь удобнее всего воспользоваться циклом с параметром.

На нашем рисунке изображена функция $f(x) = x^2$ (на отрезке [1,2]). Каждая из криволинейных трапеций будет урезана (сверху) до прямоугольника: высотой каждого из них послужит значение функции на левом конце участка. График станет "ступенчатым".

Реализация

```
step:= 1;
h:= b-a;
s_nov:= f(a)*h;
repeat
          s_star:= s_nov;
          s_nov:= 0;
          step:= step*2;
          h:= h/2;
          for i:= 1 to step do
                  s_nov:= s_nov+f(a+(step-1)*h);
          s_nov:= s_nov*h;
   until abs(s_nov - s_star)<= eps;
writeln(s_nov);
```



Вывод массива, удобный для пользователя

Задача. Распечатать *двумерный массив* размерности MxN удобным для пользователя способом. (Известно, что массив содержит только целые числа из промежутка [0..100].)

Алгоритм. Понятно, что если весь массив мы вытянем в одну строчку (или, того хуже, в один столбик), то хороших слов в свой адрес мы от пользователя не дождемся. Именно поэтому нам нужно вывести массив построчно, отражая его структуру.

Реализация

```
for i:= 1 to n do
begin
   for j:= 1 to m do write(a[i,j]:4);
   writeln;
end;
```