Содержание

Символы и строки

Две предыдущие лекции были посвящены произвольным массивам. Перейдем теперь к изучению массивов специального вида - линейных массивов, состоящих только из *символов*, - *строк*. Кроме того, сами *символы* мы тоже не обойдем вниманием.

Описание строк

В разделе `var` *строки* описываются следующим образом[1]:

```
var <имя_строки>: string[[<длина>]] [2]
```

Максимальная длина *строки* - 255 *символов*. Нумеруются ее компоненты начиная с 0, но этот нулевой байт хранит длину *строки*.

Если `<длина>` не указана, то считается, что в *строке* 255 *символов*. Поэтому для экономии памяти следует по возможности точно указывать длину используемых *строк*.

Примеры описаний:

```
var s1: string[10];       (*строка длиной 10 символов*)
   s2: string;     (*строка длиной 255 символов*)
```

Необходимо отметить, что один *символ* и *строка* длиной в один[3] *символ*

```
var c: char;
   s: string[1];
```

совершенно не эквивалентны друг другу. Вне зависимости от своей реальной длины, *строка* относится к конструируемым *структурированным типам данных*, а не к базовым порядковым (см. лекцию 2).

Символ-константа и строка-константа

Неименованные константы

В тексте программы на языке Pascal последовательность любых *символов*, заключенная в *апострофы*, воспринимается как *символ* или *строка*. Например:

```
c:='z';            {c: char}
s:='abc';  {s: string}
```

Константе автоматически присваивается "минимальный" тип данных, достаточный для ее представления: `char` или `string[k]`. Поэтому попытка написать

```
c:='zzz'; {c: char}
```

вызовет ошибку уже на этапе компиляции.

Кроме того, не забывайте, что если константа длиннее той переменной-*строки*, куда ваша программа пытается ее записать, то в момент присваивания произойдет усечение ее до нужной длины.

Пустая *строка* задается двумя последовательными *апострофами*:

```
st:= '';
```

Если же необходимо сделать так, чтобы среди *символов строки* содержался и сам *апостроф*, его нужно удвоить:

```
s:='Don''t worry about the apostrophe!';
```

Если теперь вывести на экран эту *строку*, то получится следующее:

```
Don't worry about the apostrophe!
```

Нетипизированные константы

Все правила задания *символов* и *строк* как неименованных констант остаются в силе и при задании *именованных нетипизированных констант* в специальном разделе `const`. Например:

```
const c3 = '''';  {это один символ - апостроф!}
   s3 = 'This is a string';
```

Типизированные константы

Типизированная константа, которая будет иметь тип `char` или `string`, задается в разделе `const` следующим образом:

```
const c4: char = '''';    {это один символ - апостроф!}
   s4: string[20] = 'This is a string';
```

Действия с символами

Операции

Результатом *унарной операции*
`#<положительная_неименованная_константа_целого_типа>`
является *символ*, номер которого в таблице ASCII соответствует заданному числу. Например,

```
#100  = 'd'
#39   = ''''       {апостроф}
#232  = 'ш'
#1000 = 'ш'        {потому что (1000 mod 256)= 232}
```

Кроме того, к символьным переменным, как и к значениям всех *порядковых типов* данных, применимы операции сравнения `<`, `<>`, `>`, `=`, результат которых также опирается на номера *символов* из таблицы ASCII.

Стандартные функции

Функция *`chr`*`(k:byte):char` "превращает"; номер *символа* в *символ*. Действие этой функции аналогично действию операции `#`. Например:

```
c:= chr(48); {c: char}
{c = '0'}
```

Обратной к функции *`chr()`* является уже изученная нами функция `ord()`. Таким образом, для любого числа `k` и для любого *символа* `c`

`ord(chr(k)) = k` *и* `chr(ord(c)) = c`

Надеемся, читатель помнит, что стандартные процедуры и функции `pred()`, `succ()`, `inc()` и *`dec()`*, определенные для значений любого порядкового типа[1], применимы также и к *символам* (значениям *порядкового типа* данных `char`). Например:

```
pred('[')  = 'Z'
succ('z')  = '{'
inc('a')   = 'b'
inc('c',2) = 'e'
dec('z')   = 'y'
dec(#0,4)  = '№' {#252}
```

Стандартная функция *`upcase`*`(c: char):char` превращает строчную букву в прописную. *Символы*, не являющиеся строчными латинскими буквами, остаются без изменения (к сожалению, в их число попадают и все русские буквы).

Стандартные функции и процедуры обработки строк

Для обработки символьных массивов, которыми являются *строки*, в языке Pascal существуют специальные подпрограммы:

1. Функция `concat(s1, ,sN:string):string` осуществляет слияние (***конкатенацию***) всех перечисленных *строк* или *символов* в указанном порядке. Если длина итоговой *строки* больше 255-ти *символов*, то произойдет *отсечение* "хвоста". Кроме того, даже если результат *конкатенации*не был усечен, но программа пытается сохранить его в переменную заведомо меньшей длины, то усечение все равно состоится:
   ```
   concat('abc','3de',' ','X','yz') = 'abc3de Xyz'
   ```

2. Функция `copy(s:string;i,k:byte):string` вычленяет из *строки* `s` подстроку длиной `k` *символов*, начиная с `i` -го. Если `i` больше длины *строки*, то результатом будет пустая *строка*. Если же `k` больше, чем длина оставшейся части *строки*, то результатом будет только ее "хвост":

```
copy('abc3de Xyz',2,4)  = 'bc3d'
     copy('abc3de Xyz',12,4) = ''
            copy('abc3de Xyz',8,14) = 'Xyz'
```

3. Процедура `delete(s:string;i,k:byte)` удаляет из *строки* `s` подстроку длиной `k` *символов*, начиная с `i` -го. Если `i` больше длины *строки*, то ничего удалено не будет. Если же `k` больше, чем длина оставшейся части *строки*, то удален будет только ее "хвост":

```
{s = 'abc3de Xyz'}                  {s = 'abc3de Xyz'}
     delete(s,2,3);                       delete(s,8,13);
            {s = 'ade Xyz'}                       {s = 'abc3de '}
```

4. Процедура `insert(ss,s:string;i:byte)` вставляет подстроку `ss` в *строку* `s`, начиная с `i` -го *символа*. Если `i` выходит за конец *строки*, то подстрока `ss` припишется в конец *строки* `s` (если результат длиннее, чем допускается для *строки* `s`, произойдет его усечение):

```
{s = 'abc3de Xyz'}                  {s = 'abc3de'}
     insert('xyz',s,2);                   insert('xyz',s,12);
            {s = 'axyzbc3de Xyz'}         {s = 'abc3dexyz'}
```

5. Функция `length(s:string):byte` возвращает длину *строки* `s`:

```
     length('abc3de Xyz') = 10
```

6. Функция *`pos`*`(ss,s:string):byte` определяет позицию, с которой начинается первое (считая слева направо) вхождение подстроки `ss` в *строку* `s`. Если `ss` не встречается в `s` ни разу, функция вернет `0`:

```
     pos('X', 'abc3de Xyz') = 8
```

7. Процедура `str(x[:w[:d]],s:string)` превращает десятичное число `x` (можно указать, что в этом числе `w` цифр, из них `d` дробных) в *строку* `s`. Если число короче указанных величин, то спереди и/или сзади оно будет дополнено пробелами:

```
8.   str(156.4:7:2,s);
            {s = ' 156.4 '}
```

9. Процедура *`val`*`(s:string;i:<арифметический_тип>;`*`err`*`:byte)` превращает *строку* `s` в десятичное число `x` (в случае ошибки в переменную *`err`*будет записан номер первого недопустимого *символа*):

```
10.  {s = '15.47'}
11.         val(s,x,err);
            {x = 15.47}
```

Операции со строками

Сравнения

Строки - это единственный *структурированный тип данных*, для элементов которого определен порядок и, следовательно, возможны операции сравнения (`=`, `>`, `<`).

На *строках* определен так называемый ***лексикографический порядок***: из двух *строк* меньшей считается та, у которой первый различный *символ*меньше. Считается, что пустая *строка* меньше любой другой *строки*.

Таким образом, если начальные *символы* двух сравниваемых *строк* совпадают, то эта совпадающая часть никак не повлияет на *отношение порядка*между *строками*, поэтому ее можно откинуть и сравнивать только первые *символы* оставшихся подстрок. Если одна из *строк* полностью совпадает с началом другой, то после удаления совпадающих частей она превратится в пустую *строку*. Это с очевидностью будет свидетельствовать о том, что начало слова всегда меньше, чем все слово.

Итак,

```
'abc' < 'xyz'
'a' < 'abc'
'1200' < '45'
'Anny' < 'anny'
```

Обращение к компонентам строки

Доступ к `k` -му *символу строки* осуществляется так же, как к `k` -й компоненте массива (жирные скобки являются обязательным элементом синтаксиса):

```
<имя_строки>[<индекс>]
```

Например:

```
{s = '15.47'}
c:= s[3];
{c = '.'}
```

Однако, в отличие от массива, нельзя напрямую заменять *символы* в *строке*, то есть действие

```
s[i]:= 'a';
```

не вызовет ошибки при компиляции, но, скорее всего, не станет работать во время выполнения программы. Для того чтобы изменить *символ* в *строке*, нужно воспользоваться стандартными функциями `length()`, `concat()` и`copy()`. В этом случае простое, казалось бы, действие приходится представлять как последовательность четырех операций:

1. В качестве первой под *строки* взять из *строки* `s` *символы* с `1` -го по (`k-1`)-й:
   ```
   s1:= copy(s,1,k-1);
   ```
2. В качестве второй под *строки* взять новое значение заменяемого *символа*:
   ```
   s2:= new_char;
   ```
3. В качестве третьей подстроки взять оставшуюся часть *строки* `s`:
   ```
   s3:= copy(s,k+1,length(s)-k);
   ```
4. Слить эти *строки* воедино, а результат записать вместо исходной *строки* `s`:
   ```
   s:= concat(s1,s2,s3);
   ```

Или можно объединить все четыре действия в одном операторе:

```
s:= concat(copy(s,1,k-1), new_char, copy(s,k+1,length(s)-k));
```

Конкатенация

Единственная операция, которую разрешается производить с переменными *строкового типа*, - это слияние *строк* или *символов* (*конкатенация*). Она полностью эквивалентна функции `concat()` и записывается при помощи знака " `+` ". Таким образом, предыдущий оператор можно сделать более простым:

```
s:= copy(s,1,k-1) + new_char + copy(s,k+1,length(s)-k);
```

Множества

Еще один *структурированный тип данных* - это *множество* (`set`). В нем может содержаться не более 256 элементов.

Важное отличие множества от остальных структурированных типов состоит в том, что его элементы не являются упорядоченными.

Описание множеств

В разделе `var` множества описываются следующим образом:

```
var <имя_множества>: set of <тип_элементов_множества>;
```

Элементы могут принадлежать к любому *порядковому типу*, размер которого не превышает 1 байт (256 элементов). Например:

```
var s1: set of char;      {множество из 256-ти элементов}
   s2: set of 'a'..'z','A'..'Z';  {множество из 52-х элементов}
   s3: set of 0..10;      {множество из 11-ти элементов}
   s4: set of boolean;    {множество из 2-х элементов}
```

Множество-константа

Неименованная константа

Множество можно задать *неименованной константой* прямо в тексте программы. Для этого необходимо заключить список элементов создаваемого множества в квадратные скобки:

```
[<список_элементов>]
```

Список элементов может быть задан перечислением элементов нового множества через запятую, интервалом или объединением этих двух способов. Элементы и границы интервалов могут быть переменными, константами и выражениями. Если левая граница интервала окажется больше правой, результатом будет пустое *множество*.

Примеры конструирования и использования различных множеств:

```
if c in ['a','e','i','o','u']
   then writeln('Гласная буква');
if set1 < [k*2+1..n,13] then set1:=[];
```

Нетипизированная константа

Множество - это *структурированный тип данных*, поэтому его невозможно задать нетипизированной константой.

Типизированная константа

Задать *множество* как *типизированную константу* можно в разделе `const`:

```
<имя_константы> : set of <тип_элементов> =[<список_элементов>];
```

Например:

```
type       cipher = set of '0'..'9';
const odds: cipher = ['1','3','5','7','9'];
   vowels: set of 'a'..'z' = ['a','o','e','u','i'];
```

Операции с множествами

Все *теоретико-множественные операции* реализованы и в языке Pascal:

1) Пересечение двух множеств `s1` и `s2`:	`s:=s1*s2;`
2) Объединение двух множеств `s1` и `s2`:	`s:=s1+s2;`
3) Разность двух множеств `s1` и `s2` (все элементы, которые принадлежат множеству `s1` и одновременно не принадлежат множеству `s2`)[1]:	`s:=s1-s2;`
4) Проверка принадлежности элемента `el` множеству `s` (результат этой операции имеет тип`boolean`):	`el in s`
5) Обозначение для пустого множества:	`[]`
6) Создание множества из списка элементов:	`s:=[e1,_,eN];`

7) Проверка двух множеств на равенство или строгое включение (результат этих операций имеет тип `boolean`):	`s1 = s2`
	`s1 > s2`
	`s1 < s2`

Не существует никакой процедуры, позволяющей распечатать содержимое множества. Это приходится делать следующим образом:

```
{s: set of type1; k: type1}
for k:= min_type1 to max_type1
   do if k in s then write(k);
```

Представление множеств массивами

Одно из основных неудобств при работе с множествами - это ограничение размера всего лишь 256-ю элементами. Мы приведем здесь два очень похожих способа представления больших множеств массивами. Единственным условием является наличие некоторого внутреннего порядка среди представляемых элементов: без этого невозможно будет их перенумеровать.

Представление множеств линейными массивами

Задав *линейный массив* достаточной длины, можно "вручную" сымитировать *множество* для более широкого, чем 256 элементов, диапазона значений. Например, чтобы работать с *множеством*, содержащим 10 000 элементов, достаточно такого массива:

```
set_arr: array[1..10000] of boolean;
```

При таком способе представления возможно задать *множество* до 65 000 элементов.

Для простоты изложения мы ограничимся только числовыми множествами, однако все сказанное ниже можно применять и к множествам, элементы которых имеют другую природу. Итак, признаком того, что элемент `k` является элементом нашего множества, будет значение `true` в `k` -й ячейке этого массива.

Посмотрим теперь, какими способами мы вынуждены будем имитировать операции над "массивными" множествами.

1. **Проверка множества на пустоту** может быть осуществлена довольно просто:

```
pusto:= true;
for i:= 1 to N do
      if set_arr[i] then begin pusto:= false;
                               break
                         end;
```

2. **Проверка элемента на принадлежность** множеству также не вызовет никаких затруднений, поскольку соответствующая компонента массива содержит ответ на этот вопрос:

```
is_in:= set_arr[element];
```

3. **Добавление элемента в** *множество* нужно записывать так:

```
set_arr[element]:= true;
```

4. **Удаление элемента из множества** записывается аналогичным образом:
```
set_arr[element]:= false;
```
5. **Построение пересечения множеств** реализуется как проверка вхождения каждого элемента в оба множества и последующее добавление удовлетворивших этому условию элементов в результирующее *множество*.
6. **Построение объединения множеств** аналогичным образом базируется на проверке вхождения элемента хотя бы в одно из объединяемых множеств и дальнейшем добавлении элементов в результирующее *множество*.
7. **Построение разности двух множеств** также опирается на проверку вхождения элемента в оба множества, причем добавление элемента в создаваемое *множество* происходит только в том случае, если элемент присутствует в множестве-уменьшаемом и одновременно отсутствует в множестве-вычитаемом.
8. **Проверка двух множеств на равенство** не требует особых пояснений:

```
equal:= true;
for i:=1 to N do
            if set1[i]<> set2[i]
                    then begin equal:= false;
                                                    break
                                                   end;
```

9. **Проверка двух множеств на включение** (`set1<set2`) тоже не потребует больших усилий:

```
subset:= true;
for i:= 1 to N do
            if set1[i]and not set2[i]
                    then begin subset:= false;
                                                    break
                                                   end;
```

Представление множеств битовыми массивами

В случае, если 65 000 элементов недостаточно для задания всех необходимых множеств (например, 10 множеств по 10 000 элементов в каждом), это число можно увеличить в 8 раз, перейдя от байтов к битам. Тогда 1 байт будет хранить информацию не об одном, а сразу о восьми элементах: единичный бит будет означать наличие элемента в множестве, а нулевой бит - отсутствие.

Задавая *битовый массив*, начнем нумерацию его компонент с 0:

```
set_bit: array[0..N-1] of byte;
```

Тогда результатом операции `<номер_элемента> div 8` будет номер той компоненты массива, в которой содержится информация об этом элементе. А вот номер бита, в котором содержится информация об этом элементе, придется вычислять более сложным образом:

```
bit:= <номер_элемента> mod 8;
if bit=0 then bit:= 8;
```

Эти вычисления потребуются нам еще не раз, поэтому запишем их снова, более строго, а затем будем использовать по умолчанию (`element` - это "номер" обрабатываемого элемента в нашем множестве):

```
kmp:= element div 8;      {номер компоненты массива}
bit:= element mod 8;      {номер бита}
if bit=0 then bit:= 8;
```

Перечислим теперь действия, которые потребуются для реализации операций над множествами, заданными*битовыми массивами*.

1. **Проверка множества на пустоту** почти не будет отличаться от аналогичной проверки в случае представления множества не *битовым*, а обычным массивом:

```
pusto:= true;
for i:= 0 to N-1 do
        if set_arr[i]<>0 then begin pusto:= false;
                                    break
                                    end;
```

2. **Проверка элемента на принадлежность множеству** потребует несколько большей изворотливости ведь нам теперь нужно вычленить соответствующий бит:

```
if set_arr[kmp]and(1 shl(bit-1))=0
           then is_in:= false
                   else is_in:= true;
```

   Поясним, что здесь используется операция "побитовое и" (см. лекцию 2), которая работает непосредственно с битами нужной нам компоненты массива и числа, состоящего из семи нулей и единицы на месте с номером `bit`.

3. **Добавление элемента в** *множество* теперь будет записано так:

```
set_arr[kmp]:= set_arr[kmp]or(1 shl(bit-1));
```

   Здесь нельзя использовать обычную операцию сложения (`+`), так как если добавляемый компонент уже содержится в множестве (то есть соответствующий бит уже имеет значение `1`), то в результате сложения `1+1` получится `10`: единица автоматически перенесется в старший бит, а на нужном месте окажется `0`.

4. **Удаление элемента из множества** придется записать более сложным образом:

```
set_arr[kmp]:= set_arr[kmp]and not(1 shl(bit-1));
```

   Операция `not` превратит все `0` в `1` и наоборот, следовательно, теперь в качестве второго операнда для побитового `and` будет фигурировать число, состоящее из семи единиц и нуля на месте с номером `bit`. Единицы сохранят любые значения тех битов, которые должны остаться без изменения, и лишь `0` "уничтожит" значение единственного нужного бита.

5. **Пересечение множеств** реализуется теперь при помощи операции "побитовое и":

```
for i:= 0 to N-1 do set_res[i]:= set1[i] and set2[i];
```

6. **Объединение множеств** реализуется при помощи операции "побитовое или":

```
for i:= 0 to N-1 do
                   set_res[i]:= set1[i] or set2[i];
```

7. **Разность двух множеств** может быть построена так:

```
for i:= 0 to N-1 do
```

```
                    set_res[i]:= (set1[i] or set2[i]) and not set2[i];
```

Поясним, что здесь мы вначале прибавляем содержимое второго множества к первому, чтобы затем быть полностью уверенными в правомерности операции вычитания.

8. **Проверка двух множеств на равенство** по-прежнему не требует особых пояснений:

```
equal:= true;
for i:=0 to N-1 do
      if set1[i]<> set2[i] then begin equal:= false;
                                                    break
                                                 end;
```

9. **Проверка двух множеств на включение** (`set1<set2`) будет производиться по схеме: "Если `(A\B)∪B=A`, то `B⊂A` ", доказательство которой настолько очевидно, что мы не станем на нем задерживаться:

```
subset:= true;
for i:= 0 to N-1 do
    if((set1[i] or set2[i])and not set2[i])
           or set2[i] <> set1[i]
       then begin subset:= false;
                            break
                       end;
```

Замечание. Если предстоит многократно выполнять действия с элементами *битовых массивов*, то используемые для этого значения "единиц" удобнее всего сразу задать в специальном массиве:

```
{ed: array[1..8] of byte;}
ed[1]:=1;
for k:= 2 to 8 do
          ed[k]:= ed[k-1] shl 1;
```

И далее вместо громоздкой конструкции `1 shl(bit-1)` можно использовать просто `ed[bit]`.

Примеры использования символов, строк и множеств

Задача 1. Оставить в *строке* только первое вхождение каждого *символа*, взаимный порядок оставленных *символов* сохранить.

```
program z1;
var s: set of char;
inp, res: string;
i: byte;

begin
   s:=[];
   res:= '';
   for i:= 1 to length(inp) do
                  if not(inp[i] in s)
                    then begin res:= res+inp[i];
                                                        s:= s+[inp[i]];
                                     end;
end.
```

Задача 2 [1] Оставить в *строке* только последнее вхождение каждого *символа*, взаимный порядок оставленных *символов* сохранить.

```
program z2;
var inp, res: string;
i, k: byte;

begin
   res:= '';
   for i:= 1 to length(inp) do
          begin
                  k:= pos(inp[i],res);
                  if k<>0
                          then delete(res,k,1);
                  res:= res+inp[i];
          end;
end.
```

Задача 3. Выдать первые 100 000 натуральных чисел в случайном порядке без повторений.

```
program z3;
var bset: array[0..12499] of byte;   {множество, битовый массив}
 ed: array[1..8] of byte;
 el,k: longint;
 kmp,bin: integer;
begin
        ed[1]:= 1;      {генерация массива битовых единиц}
        for k:= 2 to 8 do ed[k]:= ed[k-1] shl 1;
{-------------------------------------------------------}
        k:=0;
        randomize;      {процедура активизации генератора случайных чисел}
        while k<100000 do
          begin
           el:= 1+random(99999);       {случайное число из диапазона 0..99999}
           kmp:= el div 8;
           bin:= el mod 8;
           if bin=0 then bin:= 8;
           if bset[kmp]and ed[bin]=0   {проверка повторов}
             then begin inc(k);
                        writeln(el);
                        bset[kmp]:= bset[kmp]or ed[bin]
                        end;
          end
end.
```