15. Структурированные типы данных. Массивы

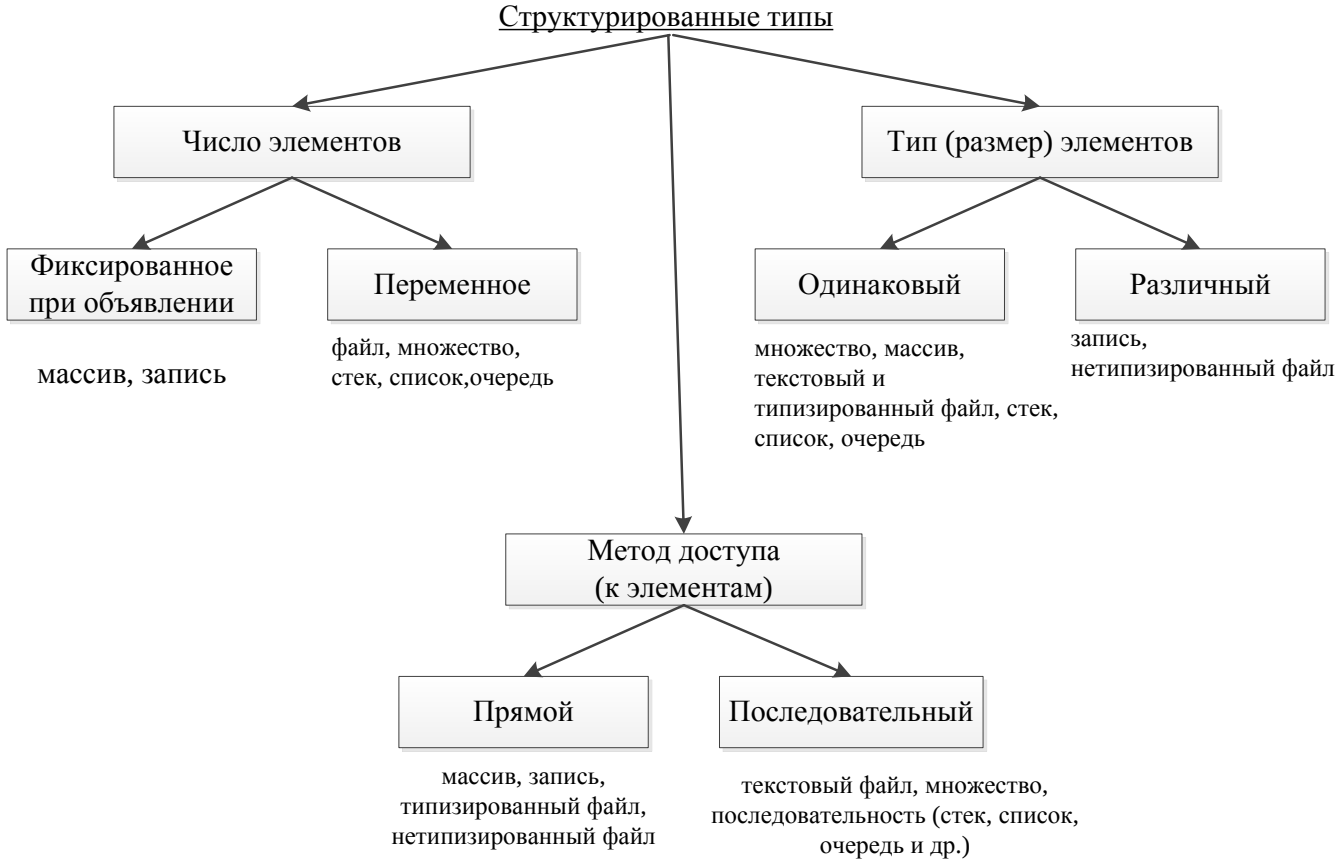
15.1 Классификация (особенности) структурированных типов данных

Практически все типы, рассмотренные до этого, кроме строкового, были **скалярными**, которые обладали следующими особенностями:

1. С именем скалярной переменной обычно было связано только одно значение (в памяти).
2. Неделимость этого значения (без специальных ухищрений, таких как маски и поразрядные операции, в целом значении нельзя обратиться напрямую к отдельному биту).

В Турбо Паскале кроме скалярных типов данных имеются **структурированные типы** данных, для которых под одним именем объединяются сразу несколько значений и вы можете работать с каждым из этих значений, используя обобщенное (родовое) имя структурированной переменной.

Классифицируются структурированные типы по следующим признакам:



Метод доступа показывает, в каком порядке можно обращаться к компонентам объекта. При **прямом** методе доступа можно произвольно (в любом порядке) обращаться к любой компоненте (части) переменной независимо от того, к какой компоненте было предыдущее обращение. При **последовательном** методе доступа можно обращаться к компоненте следующим образом: обратиться к i-ой компоненте можно только после обращения к (i-1)-ой, причём движение от компоненты к компоненте может происходить лишь в одном направлении (от (i-1)-ой к i-ой, от i-ой к (i+1)-ой и т.д.).

15. 2. Определение массива

С учётом приведённой выше классификации массив - это такой структурированный тип данных, в котором число элементов фиксировано при объявлении, тип компонентов массива одинаков, и имеется возможность прямого доступа к каждой отдельной компоненте объекта через индекс (порядковый номер).

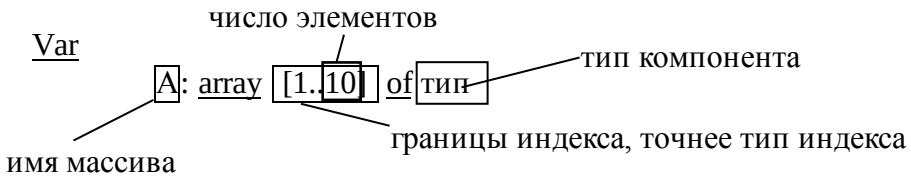
Каждый компонент имеет номер (индекс). Обращение к каждой отдельной компоненте массива происходит с использованием обобщенного имени и индекса. К сожалению, в тех редакторах, которые используют в записи программ привычные формы записи индекса массива не проходят:

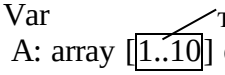
нельзя написать Ai (i рассматривается как часть имени, а не как индекс).

A_i - так тоже нельзя (при записи программ не используются нижние индексы).

На Турбо Паскале для записи индекса используются квадратные скобки: A[i]

15.3 Объявление массива на Турбо Паскале





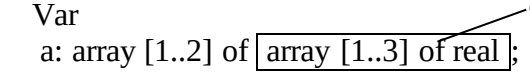
На Паскале **тип компонента массива** может быть любой, в том числе: **массив, запись, файл**. На Паскале в квадратных скобках после ключевого слова array на самом деле записываются не границы изменения индекса, а **тип индекса**. Эти типом может быть любой порядковый тип, кроме Longint и Integer, из-за ограничения на размер массива.

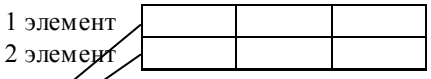
```
Var
 b: array [byte] of real;
 c: array [boolean] of real;
```

В том случае, если для типа индекса используется не тип-диапазон, а какой-то другой тип из числа разрешенных, то в этом случае число элементов массива определяется числом различных возможных значений того типа, который вы избрали для типа индекса. Для верхнего примера - число 256, в нижнем примере - 2: true, false - boolean.

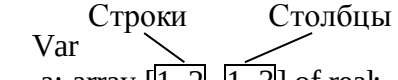
2

Если компонентой массива является массив, то будут получаться следующие длинные записи:





В данном случае объявляется массив из 2 элементов, где каждый элемент представляет собой массив (строку) из 3 элементов. Такую запись на Паскале сокращают следующим образом:



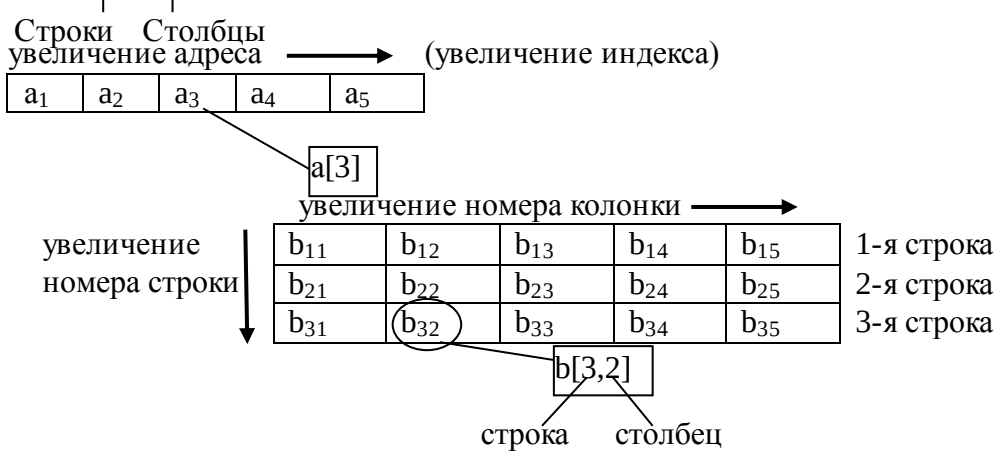
- многомерный (двухмерный) массив.

Для таких многомерных массивов при их описании отдельно описываются в квадратных скобках через запятую типы индексов, необходимые для доступа по каждой размерности массива (по каждой координате массива). Число индексов определяет размерность массива: одномерные, двумерные, и т.д.

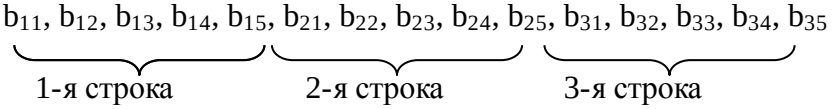
15.4 Хранение элементов массива. Доступ к элементам и частям массива

О доступе к элементам массива:

```
Var
 a: array [1..5] of byte;
 b: array [1..3, 1..5] of byte;
```



В памяти элементы массива b занимают непрерывную область и располагаются в следующем порядке:



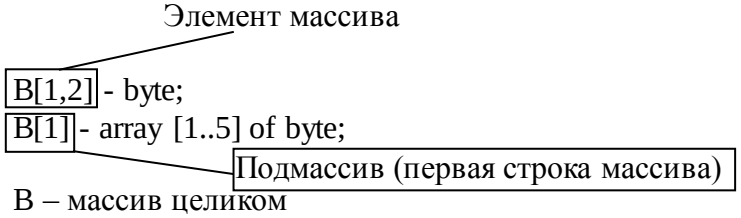
Как видно из рисунка сверху, в памяти элементы массива хранятся по строкам (эту последовательность надо знать, когда массив надо описать как типизированную константу).

Доступ к элементом массива в общем случае выполняется с использованием имени массива и индексов, записанных в квадратных скобках.

Индекс - порядковый номер элемента массива. Если одномерный - индекс - порядковый номер абсолютный. Если двумерный - первый индекс относится к строке, второй - к столбцу.

В общем случае индексом может быть выражение, значение которого должно быть совместимо по присваиванию с типом индекса, указанным при объявлении.

Правило относительно **определения типа элемента массива**: тип элемента массива определяется количеством индексов, которые указываются для этого элемента после имени массива.



B – массив целиком

Особенностями хранения массива в памяти является следующее:

1. Все компоненты массива располагаются в памяти в порядке возрастания индексов.
2. Компоненты в памяти располагаются так, что занимают в памяти непрерывную область.
3. В случае многомерных массивов их элементы располагаются в памяти таким образом, что при движении по элементам массива в направлении возрастания адресов наиболее быстро увеличивается самый правый индекс (элементы массива хранятся по строкам).

Другими словами: массивы хранятся в памяти **по строкам**. Об этом нужно помнить при инициализации массивов как **типизированных констант.**

```
Const
 a: array [1..2, 1..3] of byte = ((1,2), (3,4), (5,6)); - неправильно
 b: array [1..2, 1..3] of byte = ((1, 2, 3), (4, 5, 6)); - правильно
```

При задании начального значения массива как типизированной константы, начальное значение заключается в круглые скобки (внешние), причем внутри внешних круглых скобок может быть еще несколько пар круглых скобок. Самая внешняя пара круглых скобок соответствует массиву в целом. Внутри этой самой внешней пары должно быть столько пар круглых скобок (со значениями), сколько имеется элементов по самой левой размерности. Внутри каждой пары внутренних (после внешних) скобок должно быть еще столько пар скобок (со значениями), сколько элементов имеется по более правой размерности, и т. д. Число элементов в самых внутренних скобках должно быть равно числу элементов по самой правой размерности.

```
c: array [1..2, 1..3, 1..4] of byte = (((1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)),
                                       ((1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)))
```

2 тройки, в каждой тройке по 4 элемента

3 четверки

15.5 Уточнение команд обработки массива

Последовательность - структура данных, в которой для каждого элемента действует отношение следования (для каждого кроме последнего имеется следующий, а для каждого элемента кроме первого имеется предыдущий). Число компонентов последовательности не ограничено. Конец последовательности определяется по специальному признаку - признаку конца последовательности. Например для последовательности положительных целых чисел признаком конца последовательности может служить -1. Метод доступа к элементам - последовательный.

Замечание: массив и **последовательность** похожи друг на друга (элементы массива так же располагаются в памяти последовательно один за другим - по возрастанию индекса). Поэтому часто последовательность значений (например, последовательность промежуточных значений переменной во времени) хранится и обрабатывается как массив (массив выступает как носитель последовательности). Однако не всегда в программе надо использовать массив, если в алгоритме встречается следующая последовательность действий по обработке последовательности:

Обработать x_1
Обработать x_2
Обработать x_3
……
Обработать x_n

Правило: если элементы x_i вводятся и обрабатываются **последовательно** и их хранение (для последующей обработки) не требуется, то массив может не понадобиться. Это важно потому что, к сожалению, общий размер всех данных в памяти компьютера для программ на Паскале не может превышать 64 Кбайт. По этой же причине типом индекса не может быть Longint и Integer.

ПРИМЕР. Нахождение среднего значения в **последовательности** целых **положительных** чисел.

```
Var
        i, {вводимое число}
        j, {счетчик}
        s : Integer; {сумма}
        Продолжать: boolean;
begin
       i := 0;
        s := 0;
       j := 0;
       Продолжать := True;
        while Продолжать do
         begin
           write('Введите эл-т послед-ти: ');
           readln(i);
           if (i <> -1) {признак конца }
            then
              if i > 0
                 then
                   begin
                     s := s+i;
                     j := j+1
                   end
                 else ;
            else Продолжать := False;
          end;
       if j > 0 then s := round(s/j);
end.
```

```
Var
        i, {вводимое число}
        j, {счетчик}
        s : Integer; {сумма}
begin                    если известно, что чисел не более 10
        s := 0;
        for j := 1 to 10 do
          begin
              write('Введите эл-т послед-ти: ');
              readln(i);
              if (i <> -1)
               then
                 if i > 0
                    then
                        s := s+i;
                  else
                    break;
                end;
          end;
       if s > 0
         then
            s := round(s/j);
end.
```

Замечание: мы рассмотрели две ситуации, когда массивы не нужны для решения задачи обработки последовательности.

15. 6 Действия над массивами

15.6.1 Можно **присваивать** содержимое одного массива другому, но не всегда.

```
Type
  t = array [1..10] of byte;
Var
  a, b: array [1..10] of byte;
  c: array [1..10] of byte
  e, f: t;
begin
  a := b;
  e := f;
```

Правильно

```
  a := c;
  e := a;
```

Неправильно

Правило, по которому неправильное отделяется от правильного, очень простое: вы можете присвоить содержимое только таких массивов, у которых **эквивалентны типы** согласно объявлению.

Тип C неэквивалентен типам A и B. Если бы **нужно было** сделать тип C эквивалентным типам A и B, то нужно:

```
a, b, c: array [1..10] of byte;
```

Замечание: хотя можно присваивать содержимое одного массива другому, вы нельзя их сравнивать.

```
if a = b then ... - нельзя.
```

Поэтому, если нужно сравнить 2 массива, то их сравнивать придется посимвольно (кроме символьных массивов, которые в Паскале эквивалентны строкам).

15.6.2 Инициализация массивов.

Замечание: в отличие от языка Си, где есть старт языка т согласно этому стандарту все глобальные переменные, в том числе массивы, в начале выполнения программы инициализируются нулями, для Паскаля (Турбо-Паскаля) нет стандарта (этот язык - внутренняя разработка фирмы Борланд) и массивы (в зависимости от реализации) могут не инициализироваться (инициализироваться всяким мусором - не обязаны инициализироваться обязательно 0). Поэтому для определенности необходимо перед использованием массивов задать им начальное значение. Для этого есть три пути:

1. Использование типизированных констант (уже рассмотрели выше)
2. Использование цикла (по элементам)
3. Использование специализированной функции Fillchar().

```
Const
  a: array[1..2,  1..3]  of byte =
((1,2,3),  (4,5,6));
```

с ее помощью удобно обнулять массив

Использование цикла (по элементам)

```
Var
 a: array [1..5] of byte;
 i: byte;
begin
 randomize;
 for i := 1 to 5 do
  a[i] := i;                    // элементы массива инициализируются значениями параметра цикла
 или
  a[i] := random(100);          // элементы массива инициализируются псевдослучайными числами
        или
  begin                         // элементы массива инициализируются в диалоге с пользователем
    write('a[',  i,  '] = ');
    readln(a[i])
  end;
```

Число байтов, а не элементов массива. Причем проверка на выход за границы массива не будет выполняться.

Использование процедуры Fillchar(): типа char

Fillchar(имя массива, число байт (не элементов), значение) - заполняет побайтно всю область памяти под массив заданным значением {fillchar (a, 5 ,#0);}, причём проверка на выход за границы массива не выполняется.

Пользоваться этой процедурой необходимо аккуратно: если тип элемента массива будет не byte, а integer, то fillchar (a,5,#1) забьет массив значением 257, а не 1. Кроме того, если бы тип элементов был integer, надо было бы писать 5*sizeof(integer) вместо 5.

15.6.3 Элементы массива можно использовать везде, где можно использовать обычные скалярные переменные (в операторах присваивания и процедурах ввода-вывода и т.п.)

15.6.4 Обмен значениями между подмассивами и между массивом и подмассивами

15.6.4.1 Обмен между подмассивами

Подмассив – часть массива, к которой можно обратиться, используя число индексов, меньшее чем число размерностей массива. Более общее название подмассива – сечение. В языке PL/I: A(1,*) – 1-я строка, A(*,1) – 1-я колонка массива. В Паскале сечения возможны только для строк: A[1] – первая строка n-мерного массива.

Правило (выше было дано): тип элемента массива определяется тем, сколько индексов указано после имени массива.

Рассмотрим пример:

```
Const N=3;
Var
  A: array[1..2, 1..N] of byte;
  B: array[1..2, 1..N] of byte;
Begin
  Fillchar(a, sizeof(a) div 2, 1);      ---------- заполнение 1-ой строки единицами (правильно)
  Fillchar(a[2,1], sizeof(a) div 2, 2); ---------- заполнение 2-ой строки двойками (неправильно)
```

можно использовать sizeof(тип элемента)*число_ячеек

можно и так: round(sizeof(a)/2) -- в отличие от DIV деление с помощью '/' всегда дает вещественный результат

Замечание. Так указывать параметр fillchar ***нельзя*** (и указатель использовать тоже нельзя)
Надо (можно) один из двух вариантов:

```
1) Fillchar(a, sizeof(a), 2);          //весь массив a заполняем двойками
   Fillchar(a, sizeof(a) div 2, 0);    //первую строку очищаем от двоек
2) Fillchar(a[2], sizeof(a) div 2, 2); //вторую строкузаполняем двойками
```

```
  a[1] := a[2]; -------- можно ------- обмен между 1 и 2 строками одного массива
  b[1] := a[2]; -------- нельзя (типы не эквивалентны)
end.
```

Правило: Обмениваться могут подмассивы только эквивалентных типов.

Однако это правило *можно обойти* следующим образом. Пусть надо b[1] := a[2].

```
Move( a[2],          -- 1 аргумент (откуда брать)
      b[1],          -- 2 аргумент (куда помещать)
      sizeof(a) div 2); -- 3 аргумент (сколько переслать байтов)
```

Эта процедура просто перемещает байты из одного места в памяти в другое (проверку типа этих байтов она не выполняет).

15.6.4.2 Обмен между подмассивом и массивом

Рассмотрим сразу Пример:

```
Type
  t=array[1..3] of byte;
Var
  A: array[1..2, 1..3] of byte;
  B: t;
  C: array[1..2] of t;
Begin
  A[2] := b;     ---------- нельзя (типы не эквивалентны)
  C[2] := b;     ---------- можно (тип c[2] = t = тип b)
  Move(b, a[2], sizeof(b)); ------- можно

  B := a[2];     ----------нельзя
  B := t(a[2]);  ----------можно
  Move(a[2], b, sizeof(b)); ------- можно
  B := c[2];     ----------  можно
End.
```

тип A[2] = array[1..3] of byte (≠ t)

тип b = t

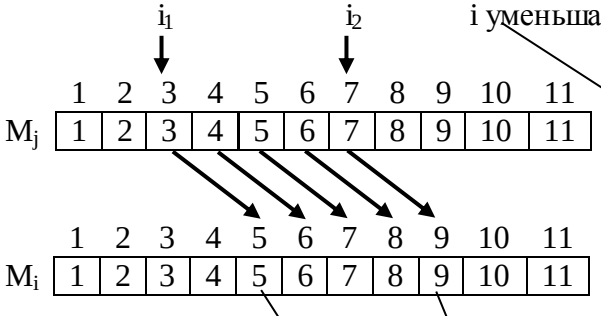
15.6.5 Сдвиг значений внутри массива

15.6.5.1 Сдвиг элементов одномерного массива вправо

Задача: необходимо выполнить сдвиг заданной части массива M (начиная с позиции i_1 (пусть i_1=3) и заканчивая позицией i_2 (пусть i_2=7) на заданное число позиций i_3 вправо (пусть i_3 = 2).

Изображение постановки задачи имеет вид:

Var M: array[1..11] of byte;



Используем разработку цикла восходящим методом и получим следующий линейный алгоритм:

```
m[9]:=m[7]
m[8]:=m[6]
m[7]:=m[5]
m[6]:=m[4]
m[5]:=m[3]
```

(i) (j)

Обобщенная запись:
M[i] := M[j] = M[a*i+b]

Массив одномерный и для прохода по нему достаточно одного индекса – i (а у нас в обобщенной записи – два индекса). Чтобы избавиться от второго индекса (j), надо в общем случае выразить зависимость j от i. В общем виде эта зависимость линейная (можно по точкам построить зависимость j от i для проверки) и имеет вид: j := a*i + b

Для нахождения a и b надо составить систему уравнений (из линейного алгоритма):

7 = a*9 + b
6 = a*8 + b
(j, i)
j = a*i+b

→ a=1; b=-2; → j=i-2 → Обобщенная запись для сдвига вправо: M[i]:=M[i-2];

кстати для сдвига влево:
M[i]:=M[i+2];

Теперь можно записать цикл. **НО:**

Нельзя

(глядя на рисунок выше написать)

$i_1 + i_3$ $i_2 + i_3$-1

```
For i:= 5 to 9 do
  M[i] := M[i-2];
```

Здесь новые значения преждевременно будут затирать старые

```
M[5]:=M[3];
M[6]:=M[4];
M[7]:=M[5];
M[8]:=M[6];
M[9]:=M[7];
```

Надо

число байт

```
For i:=9 downto 5 do
  m[i] := m[i-2];
```

т.е. двигаться по массиву при сдвиге вправо надо в направлении, противоположном направлению сдвига, т.е. влево.

```
Move[a[5], a[3], 5); // сдвиг влево
Move[a[3], a[5], 5); // сдвиг вправо
```

для произвольного типа надо писать так:

$(i_2 - i_1 + 1)$ * (sizeof(тип_элемента))

число пересылаемых элементов

число байт на один элемент

Процедура Move всегда **правильно** выполняется (она сама в правильном порядке расставляет операторы присваивания)

Самостоятельно: разобрать сдвиг массива влево.

Мы рассмотрели ситуацию, когда поиск выполняется только по одному критерию (признаку). Поиск по нескольким критериям может быть выполнен:

см. Однопроходные алгоритмы (индуктивные функции) в Кушниренко, Лебедев «Программирование для математиков»

а) За один проход по массиву (файлу, списку, последовательности) - может быть выполнен, если удовлетворение запроса не связано с накоплением статистики и ее последующим анализом (накопленной статистики).

Например: выбрать из всех студентов только мужского пола ростом выше 1.8м и не старше 20 лет.

Здесь можно записать обобщенный критерий поиска в виде логического выражения

П1	и	П2	и	П3
Пол = мужской		рост >= 180 см		возраст <=20

Задача поиска сводится к однократному проходу по массиву и отбору в ходе прохода тех студентов, характеристики которых обращают приведенное логическое выражение в истину (TRUE):

```
Repeat
 if  (П1 and П2 and П3)
      then отобрать
      else искать дальше
  перейти к следующему
until закончить
```

б) За несколько проходов по массиву– если удовлетворение запроса связано с накоплением и анализом статистики.

Например: надо выбрать из всех студентов только 3 студентов моложе 20 лет и ростом не ниже 180см которые **чаще** всего звонят (пишут) мне ночью.

Слово **чаще** в условии говорит о том, что необходимо накопить статистику перед выбором.

Здесь уже надо рассматривать не один критерий, а два (второй позволит выбрать из тех, кого отобрали по первому критерию):

- критерий, по которому отбирается (накапливается) статистика;
- критерий отбора из того, что накопили.

Критерий для накопления статистики (сколько раз звонил) должен включать признаки:

- информация о поле студента;
- информация о возрасте студента;
- информация о росте студента;
- информация о времени звонка студента

и будет иметь вид, подобный рассмотренному выше

П1	и	П2	и	П3	и	П4
Пол = мужской		рост >= 180 см		возраст <=20		(время >= 22) и (время от 0 до 9)

Решить поставленную задачу можно по крайней мере так:

Вначале за первый проход выбираем (запоминаем число звонков) для всех тех студентов, которые подходят по критерию, сформулированному выше. Затем за второй проход среди отобранных кандидатов проверяем (или вычисляем, если при первом проходе было лень вычислять) число звонков и выбираем тех, у кого это число больше.

15.6.6 Поиск элемента (одномерного) массива

Поиск – это нахождение индекса(ов) элемента(ов), значение которого удовлетворяет некоторому установленному правилу (критерию). Обычно в качестве критерия принимают равенство эталону.

Поиск

Среди неупорядоченных элементов

Простейший из методов - Линейный поиск:
Последовательный перебор в цикле всех элементов массива и сравнение их с "эталоном" до тех пор, пока не будет совпадения.
Нельзя проверить до завершения поиска, есть или нет в массиве искомый эталон.

Среднее число шагов = N/2

```
Var
  A: array[1..10] of integer;
  эталон: integer; {то, что ищем}
  i: integer; {индекс массива}
  совпадение: Boolean; {признак того, что нашли}
  rez: integer; {индекс искомого элемента}
Begin
  {Ввод значений элементов массива}
  .....
  совпадение := ложь;
  Rez := 0;
  эталон := ....;
  for i:=1 to 10 do
    IF A[i] = эталон
      then
        begin
          Совпадение := истина;
          Rez := i;
          break;
        end;
  IF совпадение = TRUE
    then
      writeln('Нашли в ', Rez, '-ом элементе.')
    else
      writeln(«Совпадений не найдено.»)
end.
```

перебор и анализ (разработка цикла нисходящим методом)

если надо найти первый (а не последний)

В самом начале двоичного поиска имеем 2 ситуации:

Ситуация	Действие
Эталон в массиве есть	Найти
Эталона в массиве нет (A[Низ] > эталон) OR (A[Верх] < эталон)	Сообщение

цикл, где на каждом повторе вычисляем новую середину и анализируем 3 ситуации:

Ситуация	Действие
Эталон в левой половине (A[Середина] > эталон)	Верх := Середина
Эталон в правой половине (A[Середина] < эталон)	Низ := Середина
Эталон ровно в середине (A[Середина] = эталон)	Нашли := True Результат := Середина

выход из цикла := ((Нашли = True) OR (Низ >= Верх))

Среди упорядоченных элементов

ascend - по возрастанию
descend – по убыванию

Можно (и нужно) проверить до начала поиска, есть или нет в массиве искомый эталон.
Простейший из методов - двоичный поиск.
Пусть требуется найти в массиве A (элементы в нём отсортированы по возрастанию) из N элементов элемент, совпадающий с "эталоном".
Вначале мы массив из N элементов делим на 2 половины и определяем, в какой из половин находится эталон. Эта половина принимается за новый интервал и снова делится и т.д.

Среднее число шагов = $\log_2 N$

```
label
  уходим;
Const
  L=1;
  H=10;
Var
  A: array[L..H] of integer;
  эталон: integer;
  Результат: Longint;
  Нашли: Boolean;
  Середина, Низ, Верх: Longint; //текущие границы интервала
Begin
  {Ввод значений элементов массива}
  .....
  Низ := L; {левая граница интервала}
  Верх := H; {правая граница интервала }
  Нашли := FALSE; {признак того, что нашли}
  Середина:= 0; {индекс середины интервала между Верх и Низ}
  Результат := 0; {индекс искомого элемента}
  эталон := ....; {то, что ищем}
  {Проверка - есть ли искомый эталон в массиве?}
  IF (A[Низ] > эталон) OR (A[Верх] < эталон)
    then
      begin
        writeln('Совпадение искать бесполезно');
        goto уходим;
      end;
  Repeat
    Середина := round((Верх+Низ)/2);
    IF A[Середина] > эталон
      then Верх := Середина //эталон в левой половине
      else
        if A[Середина] < эталон
          then Низ := Середина //эталон в правой половине
          else
            Нашли := TRUE; // значит A[Середина] = эталон
  until (Нашли=TRUE) OR (Низ >= Верх))
  IF Нашли = TRUE
    then
      Результат := Середина {в Результат - индекс искомого элемента}
    else
      writeln(«Совпадений не найдено.»)
  уходим:;
```

вариант цикла ***без break***:

```
i:=0'
repeat
  i:=i+1;
  if a[i]=эталон
    then
      begin
        совпадение:=True;  Rez:=i;
      end;
until (совпадение = true) or (i=10);
```

15.6.7 Сортировка элементов (одномерного) массива (для ускорения поиска в массиве)

Сортировка - расположение (переупорядочивание) элементов по определенному критерию. Обычно - по убыванию (descend) или возрастанию (ascend). Существует много методов сортировки. Простейший, но не самый худший (лучший) - сортировка выбором. Рассмотрим, как выполнить сортировку по убыванию. Для этого надо:

- сначала последовательно рассматривать кандидатов на место самого первого элемента (самого большого), после чего наибольший из кандидатов становится первым (обменивается с первым);

- затем среди оставшихся (N-1) элементов массива последовательно рассматриваются кандидаты на место 2-го элемента и наибольший из кандидатов при просмотре становится вторым (обменивается со вторым элементом);

- и т.д.

Очевидно, что число просматриваемых элементов с каждым шагом уменьшается.

Введем обозначения:

i – индекс элемента, на место которого ищем кандидата

j – индекс элемента, который мы просматриваем в качестве кандидата

```
i=1   i=2   i=3   i=4   i=5
1     3     4     2     5--- Исходное состояние массива
3     1     4     2     5----После итерации для j=2----------i = 1  j = 2
4     1     3     2     5----После итерации для j=3----------i = 1  j = 3     i = 1
4     1     3     2     5----После итерации для j=4----------i = 1  j = 4
5     1     3     2     4----После итерации для j=5----------i = 1  j = 5
-----------------------------------------------------------------
5     3     1     2     4----После итерации для j=3----------i = 2  j = 3
5     3     1     2     4----После итерации для j=4----------i = 2  j = 4     i = 2
5     4     1     2     3----После итерации для j=5----------i = 2  j = 5
-----------------------------------------------------------------
5     4     2     1     3----После итерации для j=4----------i = 3  j = 4     i = 3
5     4     3     1     2----После итерации для j=5----------i = 3  j = 5
-----------------------------------------------------------------
5     4     3     2     1---- После итерации для j=5----------i = 4  j = 5     i = 4
```

Можно заметить:

- при поиске каждого нового кандидата число просматриваемых элементов уменьшается на единицу;

- при поиске кандидата на I-ое место I фиксируется, а J пробегает значения от I+1 до N.

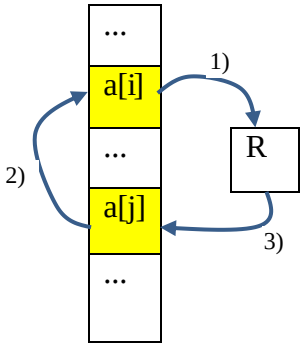
Очевидно, что требуется два цикла:

- внешний по элементам, для которых ищем кандидата (цикл по параметру I, где I изменяется от 1 до N-1);

- внутренний по элементам-кандидатам (цикл по параметру J, где J изменяется от I+1 до N).

В итоге получим следующий фрагмент программы:

```
for i:= 1 to N-1 do
 for j:=i+1 to N do
                          ему ищем замену
  if ( a[j] > a[i] ) // если кандидат больше по значению, чем текущий
                кандидат
    then
      begin
       1)   R := a[i];  //чтобы не потерять значение переменной a[i]
       2)   a[i] = a[j];
       3)   a[j] = R;
      end;
```



Замечания: по поводу реализации массивов в Паскале:

1. Число размерностей массива ничем не ограничено, однако размер массива как любой статической переменной в памяти ограничен 64 Кб (нельзя объявить тип размером >64 Кб).
2. Индексы могут быть как положительными, так и отрицательными.
3. Тип индексов может быть любым порядковым, кроме типа longint.
4. В Турбо-Паскале имеется два режима компиляции, имеющие отношение к массивам:

{$R+} и {$R-} - выполняется или нет проверка, не выходит ли индекс за объявленные границы диапазона:

- если проверка выполняется, то при нарушении границ диапазона программа аварийно завершается с кодом 201 (Range Check Error);
- если проверка не выполняется, то при нарушении границ диапазона программа аварийно не завершается и никаких сообщений не выдается (но при этом программа сама себя портит).

Примечание:

Если отключить контроль диапазонов, можно создать массив с переменными границами (свободный массив):

```
{$R-}
Var a: array [1..1] of integer:
```

Такое объявление указывает компилятору, что мы хотим иметь переменную *a* с индексом.

Если по адресу *a* динамически выделить память, то получим массив такого размера, какого пожелаем (мы вернемся к этому, когда будем рассматривать тему «Указатели»).

5. При передаче массивов как параметров в подпрограммы не рекомендуется определять тип массива непосредственно в заголовке подпрограммы (из-за невозможности при этом обеспечить эквивалентность типов фактического и формального параметров - массивов).

Замечание. Записать-то так при объявлении подпрограммы можно (сама запись не есть ошибка), но делать так не надо из-за проблем с совместимостью дальше по тексту программы..

6. В качестве формального параметра подпрограммы можно использовать массив без границ (открытый массив).

```
Var
 procedure p(a: array of char); .....        high(a)
```

Для работы с ними используются функции High (возвращает верхнюю границу индекса - обычно = число элементов массива минус 1; не путать с Hi(x) для целый аргументов) и Low (возвращает нижнюю границу индекса, обычно = 0; не путать с Lo(x) для целых аргументов).

Как можно было бы сделать, если использовать массив?

1. Ввести значения и заполнить ими массив.
2. Найти среднее значение в массиве.

```
Var
  a: array[1..10] of integer;
  s: integer;
  i:byte;
begin
  s:=0;
  for i:=1 to 10 do
    begin
        wtite('введите -->');
        readln(a[i]);
        s:=s+a[i];
    end;
end.
```

Прежде, чем рассматривать далее вопросы уточнения команд обработки массивов **(в учетом их регулярной структуры), рассмотрим дополнительно теорию - Правила** разработки циклов.

16. Правила разработки цикла

Известно по крайней мере два класса **задач, которые приводят к появлению цикла**:

1. **обработка или формирование массивов** - при переносе формулировки задачи типа

<обработать массив > //типа <вывести на экран>

или

<сформировать массив> //типа инициализировать/ввести массив>

с объекта на каждый его компонент получаем повторяющиеся подзадачи. Эту последовательность подзадач сокращаем, записывая формулировку подзадачи один раз (в обобщенной форме) и повторяя в цикле. При решении этого класса задач циклы разрабатываются **сверху вниз.** Число повторений обычно определяется числом элементов массива, поэтому используются циклы for (со счетчиком).

2. **итерационные алгоритмы** – алгоритмы, основанные на использовании рекуррентных формул. При решении задач этого (второго) класса циклы разрабатываются **снизу-вверх.** Рекуррентные формулы обычно используются потому, что для снижения сложности вычислений (замена умножения - сложением, а возведения в степень - умножением и т.п.) на каждом шаге при каждом новом повторении цикла каждый новый результат получается с использованием старого (ранее полученного) результата. Можно выделить 2 разновидности итерационных алгоритмов:

2.1 Алгоритмы, связанные с реализацией т.н. n-арных операций:

$S = \sum_{i:=1}^{n} x_i = x_1+x_2+x_3+...+x_n$ $S = \prod_{i:=1}^{n} x_i = x_1 \cdot x_2 \cdot x_3 \cdot ... \cdot x_n$ $S = X! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot ... \cdot X$ и т.п.

Реализация n-арных операций требует преобразования их в последовательность двухместных (бинарных) операций, что приводит к появлению повторяющихся действий и необходимости сворачивания этих повторяющихся действий в цикл.

2.2 Алгоритмы приближенных вычислений - алгоритмы, разрабатываемые для поэтапного (пошагового) приближенного нахождения решения задач (например, поиск корня уравнения), для которых неизвестно точное решение.

Обе задачи второго класса приводят к появлению циклов, которые разрабатываются методом **"снизу вверх".** При этом первая разновидность задач второго класса реализуется с помощью **циклов с известным числом повторений** (цикла for - для каждой n-арной операции, зная *n*, можно определить число повторений), а вторая разновидность - с помощью циклов с **неизвестным числом повторений.**

Известно, что структура цикла (с известным числом повторений) включает в себя три компонента: подготовка цикла, заголовок цикла, тело цикла (те действия, которые необходимо многократно повторить). Заголовок цикла задает правила изменения параметра цикла (тело цикла повторяется для каждого значения параметра цикла).

Как правило, очередное повторение тела цикла строится с использованием результата, полученного при предыдущем повторении тела цикла, например, как в данном случае

$x_i = x_{i-1} + h$

При этом подготовка цикла практически имитирует такое выполнение тела цикла, которого еще не было. Это делается затем, чтобы настроить тело цикла на первое правильное выполнение.

Однако, если цикл организован таким образом, что очередное повторение тела цикла не использует результат предыдущего повторения, как в данном случае

$x_i = (i-1)*h$

то в этом случае подготовка цикла не нужна.

Порядок разработки циклов восходящим методом:

1. Записываем линейный алгоритм (в котором, например, многоместная операция реализуется с помощью последовательности большого количества бинарных операций).

2. Находим рекуррентную формулу (замечаем повторяемость) для определенной последовательности шагов.

3. Те повторяющиеся действия, для которых удалось получить обобщенную формулу, сворачиваем в цикл. При этом в тело цикла попадают те действия, которые записаны в виде рекуррентной формулу; в заголовке цикла записываем правило, с помощью которого устанавливается, повторять далее цикл или нет; в подготовку цикла включаются те действия,

которые не удалось обобщить.

4. Если подготовка цикла не простая, то необходимо выполнить ее упрощение.

5. Если попытка упрощения подготовки цикла удалась, то необходимо скорректировать заголовок так, чтобы число повторений стало на 1 больше.

6. Необходимо выполнить операцию отбрасывания лишних индексов.

Теперь о том, когда **циклы разрабатываются методом "сверху вниз"**. Вернемся к уточнению команд обработки цикла.

При появлении команд:

сформировать массив

или

обработать массив

Пример:

ввести массив
или
вывести массив
или
присвоить массив массиву

} не связано с вычислением и накоплением значений

Но не:

Найти сумму элементов массива
Найти минимальный или максимальный элемент массива

} связано с накоплением значений

Эти команды (из верхних примеров) уточняются следующим образом: в силу того, что объект (массив) обладает однородной внутренней структурой, мы можем воспользоваться стандартным правилом уточнения команд и перенести исходную задачу на каждый элемент массива (компонент объекта).

В результате получается большое количество формулировок команд следующего вида:

обработка массива → обработать первую компоненту
обработать вторую компоненту
.....
обработать n-ую компоненту

уточнение

Как известно, наличие большого количества повторяющихся формулировок команд вызывает необходимость сворачивания их в цикл, в отличие от разработки цикла восходящим методом. В данном случае **появление циклов можно предвидеть**, при этом нет необходимости вначале записывать линейный алгоритм, находить обобщенную запись и т. д. В данном случае **цикл разрабатывается методом "сверху вниз"**, и такая разработка включает в себя следующие шаги:

1. Сразу можно записать тело цикла, имеющее одну из следующих 2 форм:

сформировать i-ю компоненту

или

обработать i-ю компоненту

2. Можно сразу записать заголовок цикла, в котором в качестве параметра цикла можно использовать индекс (номер) элементов массива. Закон изменения параметра цикла, или количество повторений цикла, будет определяться числом компонент, которые необходимо обработать.

Например:

For i := <нач. знач.> to <кон. знач.> do …

где i – параметр цикла.

3. Согласно ранее сформулированному правилу с учетом того, что тело цикла обычно не использует результат предыдущего, подготовка цикла обычно не нужна. Выше было показано, что подготовка цикла нужна только в случае формулировок задач вида

- Найти сумму элементов массива
- Найти минимальный или максимальный элемент массива

и т.п.