Министерство образования Российской Федерации
НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра “Прикладная математика”

Элементы алгоритмизации и основы программирования на языке Турбо Паскаль

Методическая разработка по курсу “Информатика”
для студентов дневного и вечернего обучения

Нижний Новгород 2001

Составители : **В. Ф. Билюба, Е. А. Маслова, С. П. Никитенкова**

УДК 651.3.06

Элементы алгоритмизации и основы программирования на языке Турбо Паскаль: Метод. разработка по курсу “Информатика” /НГТУ;
Сост.: В. Ф. Билюба и др. Н. Новгород, 2001. 43 с.

Представлены основные элементы алгоритмизации и программирования процессов обработки различных типов данных средствами алгоритмического языка ТУРБО ПАСКАЛЬ. Приведены примеры составления различных видов алгоритмов и программ для решения на компьютере типовых задач математического, экономического и инженерного содержания.

Научный редактор С. Н. Митяков
Редактор И. И. Морозова

Компьютерный набор : В. Ф. Билюба
Е. А. Маслова
С. П. Никитенкова

Подп. 04.06.2001. Формат 60x84 1/16. Бумага газетная. Печать офсетная.
Печ.л. 2,75. Уч.-изд. л. 2,6 . Тираж 500 экз. Заказ 430 .

Нижегородский государственный технический университет.
Типография НГТУ. 603600, Н. Новгород, ул. Минина, 24.

Оглавление

1. ВВЕДЕНИЕ

Язык программирования **Pascal** был первым языком, в котором реализована концепция структурного программирования, определенная Дейэкстрой и Хоором. Pascal был разработан Н.Виртом в Швейцарском техническом институте в Цюрихе. Язык **Pascal** представляет собой значительный шаг в развитии языков программирования. Разветвленная структура типов данных позволяет программисту работать с разнообразными данными в необходимом ему абстрактном виде. Система “сильной” типизации обеспечивает надежность создаваемых программ. Программные конструкции языка **Pascal** позволяют легко отразить в структуре программы структуру лежащего в ее основе алгоритма. Разработка фирмой *Borland International* интегрированной системы программирования **Turbo Pascal** со встроенным отладчиком позволяет сократить время отладки, использовать многооконный режим работы, мышь, возможность использования при написании программ языка низкого уровня **Ассемблер**, а также возможность создавать объектно-ориентированные программы. Все это предопределяет использование языка **Pascal** для создания программ различного назначения, а также его преимущества перед другими языками в качестве средства обучения программированию. В настоящее время разработано множество диалектов языка Pascal. Одним из наиболее популярных является **Turbo Pascal (TP)**, краткое изложение которого в версии 7.0 приводится ниже.

2. НАЧАЛА АЛГОРИТМИЧЕСКОГО ЯЗЫКА ПАСКАЛЬ

2.1. Алфавит языка, идентификаторы, зарезервированные слова

Алфавит - совокупность допустимых в языке символов. В языке **TP** все элементы (числа, слова, предложения, операторы) формируются из множества символов международного стандарта ASCII, используемого в современных ПК, а также специфических для данной страны литер. Каждому символу алфавита согласно этому соглашению соответствует свой числовой код от 0 до 255. Символы с кодами от 0 до 127 составляют основную таблицу стандартных кодов. Вторая половина кодов от 128 до 255 используется для обозначения букв русского алфавита и иных.

Идентификатором является последовательность букв, цифр и символов подчеркивания, которая начинается с буквы или символа подчеркивания и не содержит пробелов. Идентификаторы используются в качестве имен различных компонент языка при программировании, а именно: констант, типов, переменных, процедур, функций, модулей, программ, меток и полей в записях. Для составления идентификаторов можно использовать латинские буквы, арабские цифры от 0 до 9, символ подчеркивания __. Заметим, что в идентификаторах

можно использовать строчные вместо прописных букв и наоборот (**компилятор их не различает!**). Пробел является разделителем и не может стоять внутри идентификатора. Длина идентификатора не ограничена, однако значимыми являются только первые 63 символа. В языке **TP** есть две разновидности идентификаторов: стандартные и пользовательские.

Стандартными идентификаторами являются имена всех встроенных в язык процедур и функций (**Read, Write, Sin** и др.), типов (**Integer, Real, Char** и др.) и т.д. Стандартные идентификаторы, как правило, не переопределяются.

Пользовательские идентификаторы придумывает программист как имена переменных, функций, процедур и других элементов создаваемой им программы.

Зарезервированные (ключевые) слова в языке **TP** имеют строго определенное назначение, которое не может быть изменено. Поэтому, пользовательские идентификаторы не должны совпадать с зарезервированными словами, например, такими как: **Program, Begin, End, Var, Type, Record, Function, Uses, Label, Array, Goto, If, Else, Then, Do, For, Repeat, Until, And, Or** и др.

2.2. Общая структура программ и типов данных

Любую программу, написанную на Паскале, можно условно разделить на три основные части: раздел объявлений, раздел текстов процедур и функций, раздел основного блока программы .

Более подробно структуру программы можно представить следующей таблицей:

```
 { РАЗДЕЛ ОБЪЯВЛЕНИЙ }
Program     {Начало заголовока программы }
{$...}      { Начало глобальных директив компилятора}
Uses        { Начало подключение библиотеки модулей }
Label       { Начало подраздела объявлений глобальных меток}
Const       { Начало подраздела объявлений глобальных констант }
Type        { Начало подраздела объявления глобальных типов}
Var         { Начало подраздела объявления глобальных переменных }
 {РАЗДЕЛ ТЕКСТОВ ПРОЦЕДУР И ФУНКЦИЙ }
Procedure   { Начало представления (описания) процедуры }
{Другой вариант – Function для начала представления функции }
Label       {Подраздел объявления локальных меток }
Const       {Подраздел объявления локальных констант }
Type        {Подраздел объявления локальных типов}
Var         {Подраздел объявления локальных переменных}
Begin
...         {Основной блок процедуры (или функции) }
End ;       {Точка с запятой в конце процедуры обязательна!}
```

```
... ;        {Другие процедуры имеют аналогичную структуру}

  {РАЗДЕЛ ОСНОВНОГО БЛОКА ПРОГРАММЫ}
BEGIN
...  {Основной блок программы содержит вызов для исполнения
      представленных в предыдущем разделе процедур и функций}
END .        {Точка в конце программы обязательна!}
```

Структура программы на языке ***Turbo Pascal***

В разделе объявлений программист сообщает компилятору, какими идентификаторами (именами) он обозначает данные (например константы и переменные), а также определяет типы данных, которые он намерен использовать во всей программе. Тип определяет множество допустимых значений, а также множество допустимых операций, которые применимы к значениям этого типа. Кроме того, тип данных определяет формат внутреннего представления данных в памяти компьютера.

Язык **TP** содержит разветвленную структуру типов данных, к числу которых относятся:

1. ***Простые типы***	***Обозначения в программе***
- целочисленные ;	**integer**, byte, shortint, word, longint ;
- вещественные ;	**real**, single, double, extended, comp ;
- логический (булевский);	**boolean** ;
- символьный ;	**char** ;
- перечисляемый ;	
- интервальный (тип- диапазон).	
2. ***Структурированные типы***	
- массивы;	**array**;
- строки;	**string**;
- записи;	**record**;
- файлы;	**file, text**;
- множества;	**set**.
3. ***Указатели***	^ .
4. ***Объекты***	**object** .

Раздел текстов процедур и функций предназначен для написания подпрограмм пользователя, вызываемых для выполнения в основном блоке программы или из других процедур (функций) пользователя. Стандартные процедуры либо функции и модули, записанные в библиотечных файлах, объявленных в подразделе **Uses**, также можно вызывать для исполнения в основном блоке программы, а также из других процедур и функций.

В тексте программы могут присутствовать комментарии. Они нужны для документирования (пояснения) программы или ее подразделов. Это могут быть пояснения назначения переменных, операторов, частей программы, всей программы в произвольной форме, необходимо только заключить текст комментария с обеих сторон в фигурные скобки { } или в символы (* *).

Многие программы разрабатываются по принципу: ввод исходных данных - обработка данных - вывод результатов обработки. В качестве примера рассмотрим следующую программу, складывающую два целых числа, вводимых с клавиатуры.

```
Program Slogenie_celih;                    {Заголовок программы}
Var   Number_1, Number_2, Sum : integer;
BEGIN
Write  (‘ Введите первое число = ’)  ;
ReadLn   (Number_1) ;
Write   (‘ Введите второе число = ’)  ;
ReadLn  (Number_2) ;
Sum := Number_1 + Number_2 ;
WriteLn (‘ Сумма введенных чисел равна: ’ , Sum)
END . { Точка в конце программы – обязательна!}
```

Замечание. Символы «Ln» в стандартных процедурах **ReadLn** и **WriteLn** означают, что после выполнения оператора ввода/вывода курсор текста переходит в начало следующей экранной строки. Процедуры **Write** в начале программы используются для вывода на экран подсказки пользователю о необходимости ввода числа в той же строке. Процедура (оператор) **WriteLn** в конце программы обеспечивает вывод результата.

2.3. Простые типы данных. Константы, переменные, операции, выражения. Стандартные функции

- *Целочисленные типы* предназначены для хранения и обработки целых чисел со знаком (**integer** (чаще используется), shortint, longint) и без знака (byte, word), занимающие в памяти компьютера место различной длины и имеющие различный диапазон изменения соответственно. *Арифметические операции*, допустимые для данного типа: **+** (сложение), **-** (вычитание), * (умножение), **div** (частное от целочисленного деления), **mod** (остаток от целочисленного деления).

- *Вещественные типы*. Предназначены для хранения и обработки чисел с дробной частью. Заметим, что вещественный тип данных допускает ввод данных с десятичной точкой и без нее. Действительные числа не могут храниться в целочисленных переменных. Основной вещественный тип **real** определяет точность 11-12 десятичных знаков и занимает в памяти 6 байтов. Другие вещественные типы (single, double, extended и comp) имеют точность от 7-8 десятичных знаков до 19-20 и занимают в памяти от 4 до 10 байт. Над данными

вещественного типа допустимы известные *арифметические операции*: **+** (сложение), **-** (вычитание), ***** (умножение), / (деление).

- *Логический* (*булевский*) *тип*. Подразумевает два значения: true (**истина**) и false (**ложь**). Булевские переменные предназначены для хранения значений логических выражений. Переменные булевского типа описываются как **boolean**. Значение *true* соответствует единице (**1**), а *false* - нулю (**0**). Допустимые *логические операции* над данными булевского типа: **and** (логическое **и**); **or** (логическое **или**); **xor** (исключающее **или**), **not** (логическое **не**). Например, логическое выражение **(x>=0) and (x<=1)** определяет принадлежность вещественной переменной **x** отрезку [0; 1] и принимает значение true, если **x** принадлежит отрезку и значение false, если **|x| ≥ 1**.

Над данными целого и вещественного типов можно осуществлять следующие *операции отношения*: > (**больше**); < (**меньше**); >= (**больше или равно**); <= (**меньше или равно**); = (**равно**); < > (**не равно**).

- *Символьный тип*. Тип **char** предназначен для хранения одного символа (буквы, цифры, знака и т.д,) из 256 символов кода ASCII и занимает в памяти 1 байт. Для присвоения символьной переменной значения необходимо взять присваиваемый символ в апострофы или перед кодом этого символа поставить знак # . Например, **s2 := #65** (символьной переменной **s2** присваивается символ с кодом прописной латинской буквы **A**), **s3:= #66** (символьной переменной **s3** присваивается символ **B**) и т. д.. Для того чтобы узнать код символьного аргумента, используется стандартная функция **Ord**, например **Ord(s2)** принимает значение **65, Ord(s3) – 66**.

Замечание. К символьными данным применимы *операции отношения*, при этом сравниваются коды символов.

-*Перечисляемый тип* определяется как упорядоченный набор идентификаторов, заданный перечислением. Например:

```
Type
    Week = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
```

- *Тип-диапазон* используется для представления подмножества данных уже определенного типа, за исключением вещественного. Например, объявление

```
Type
      number_row = 1..10;
Var
      n1 : number_row;
```

задает следующее ограничение на величину n1: n1 должно быть целочисленным и находиться в интервале от 1 до 10.

Константа - это имя (идентификатор), обозначающее в программе заданную неизменяемую величину определенного типа. При объявлении констант используется зарезервированное слово **Const** . Например:

```
Const  e    =  2.71828 ;
       letter = ’a’;
        row  = ‘язык ’;
```

Переменная - это имя (идентификатор) ячейки памяти компьютера, т. е. элемент программы, который предназначен для хранения, изменения и передачи данных внутри программы. При объявлении переменных используется зарезервированное слово **Var** (от **Var**iable - переменная). Например:

```
Var
        i,  j,  k       : integer;
        x, a, b, c, z : real;
        s1,s2          : char;
        str             : string;
```

Оператор присваивания позволяет присвоить определенное значение переменной выбранного типа. Форма записи:

Переменная **:=** *Выражение*

Если *Переменная* и *Выражение* одного типа, то операция присваивания выполнится всегда. Выражение целого типа можно использовать для присваивания переменной вещественного типа. Ниже приводится пример программы, в которой используются операторы присваивания при вычислении значения переменной ***x*** и выводится результат последнего на экран.

```
Program  Example;
Var    x : real;
BEGIN
x := 2;                 {   Переменной   x   присваивается значение  2 }
x := x + 5;             {   Величина    x       увеличивается на  5  }
Write (‘ x = ‘, x ); {   На экран выводится текст    x=7   }
END .
```

В выражении целого типа используются значения, константы и переменные только целого типа. При вычислении целого или вещественного выражения арифметические операции производятся слева направо, но, как и в математике, умножение и деление имеют более высокий приоритет, чем сложение и вычитание. Этот порядок можно изменить, расставив соответствующим образом круглые скобки (). В записи выражения целого или вещественного типа можно

использовать *стандартные математические функции и процедуры* с аргументами целого или вещественного типа:

abs (x) - абсолютная величина (модуль) числа;
arctan (x) - арктангенс числа;
cos (x) - косинус угла, заданного в радианах;
exp (x) - экспонента числа;
frac (x) - дробная часть числа;
int (x) - целая часть от числа;
ln (x) - натуральный логарифм числа;
pi - число **π**;
random(x) - целое случайное число от 0 до x;
round (x) - округленное целое, т.е. **int**(x + 0.5);
sin (x) - синус угла, заданного в радианах;
sqr (x) - квадрат числа;
sqrt (x) - корень квадратный из числа.

Для вычисления функций, отсутствующих в данном списке, следует выразить их через известные стандартные для Паскаля функции :

a^x = exp(x * ln(a));

$\sqrt[n]{x}$ = exp(ln(x)/n), *где n - целое значение;*

tg x =sin(x)/cos(x) ;

$\arcsin x = \operatorname{arctg}\frac{x}{\sqrt{1-x^2}} = \operatorname{Arc}\tan(x / \operatorname{sqrt}(1 - x * x));$

$\arccos x = \operatorname{arctg}\frac{\sqrt{1-x^2}}{x} = \operatorname{Arc}\tan(\operatorname{sqrt}(1 - x * x) / x);$

lg x = ln(x) / ln(10) .

Пример записи выражения из данных вещественного и целого типов

$$\frac{2a + \sin^2 x - |a - x|}{\sqrt[3]{|a + x|} + \arcsin x - 3\ln a}$$

в форме языка **Паскаль**:

```
( 2 * a + Sqr ( Sin (x )) - Abs(a-x)) / (Exp(Ln (Abs (a+x)/3)) +Arctan(x/Sqrt(1-Sqr(x))) - 3 *  Ln (a))
```

Стандартные функции **TP** облегчают обработку данных не только целого и вещественного типов, но и *других типов*, например:

Chr *(x)* - преобразование кода ASCII целого типа в символьный;
Ord (x) - преобразование любого порядкового типа в целый, например символа в его код;
Odd (x) - проверка целого числа на нечетность;
Round (x) - округление вещественного числа до ближайшего целого;
Pred (x) - определение предыдущей величины ;
Succ (x) - определение последующей величины .

2.4. Стандартрые процедуры ввода и вывода. Программирование линейного алгоритма

Для *вывода* информации в **TP** используется стандартная процедура с произвольным списком параметров **Write** или**WriteLn**. Она предназначена для вывода данных из оперативной памяти ПК на внешние носители информации, в частности экран, бумагу, в файл на диске.Форма записи:

Write (*Список параметров*) ,

где в качестве параметров могут использоваться переменные, строки символов и константы.

Для *ввода* информации с клавиатуры, чтения ее из файла в память ПК аналогично используется стандартная прцедура **Read** (или **ReadLn**). Окончание **Ln** переводит курсор на начало новой строки после завершения работы процедуры ввода/вывода. При выполнении процедур *ввода/вывода* данные вводятся или выводятся в порядке очередности в списке. Ниже приводится пример программы, рассчитывающей величину подоходного налога для зарплаты, которая вводится с клавиатуры пользователем в ответ на подсказку.

```
Program     Payment ;
Var   Pay:  Real;  { зарплата   }
      Tax:  Real;  { налог }
BEGIN
      Write (‘  Введите зарплату =   ‘) ;
      Readln ( Pay ) ;
      Tax : = 0.12 *  Pay ;
      Writeln(‘Для зарплаты‘,Pay:8:2, ‘ рублей налог= ‘ , Tax : 7 : 2, ‘ руб.’)
END .
```

При выполнении этой программы первым выполнится процедура **Write**, и на экране появится подсказка

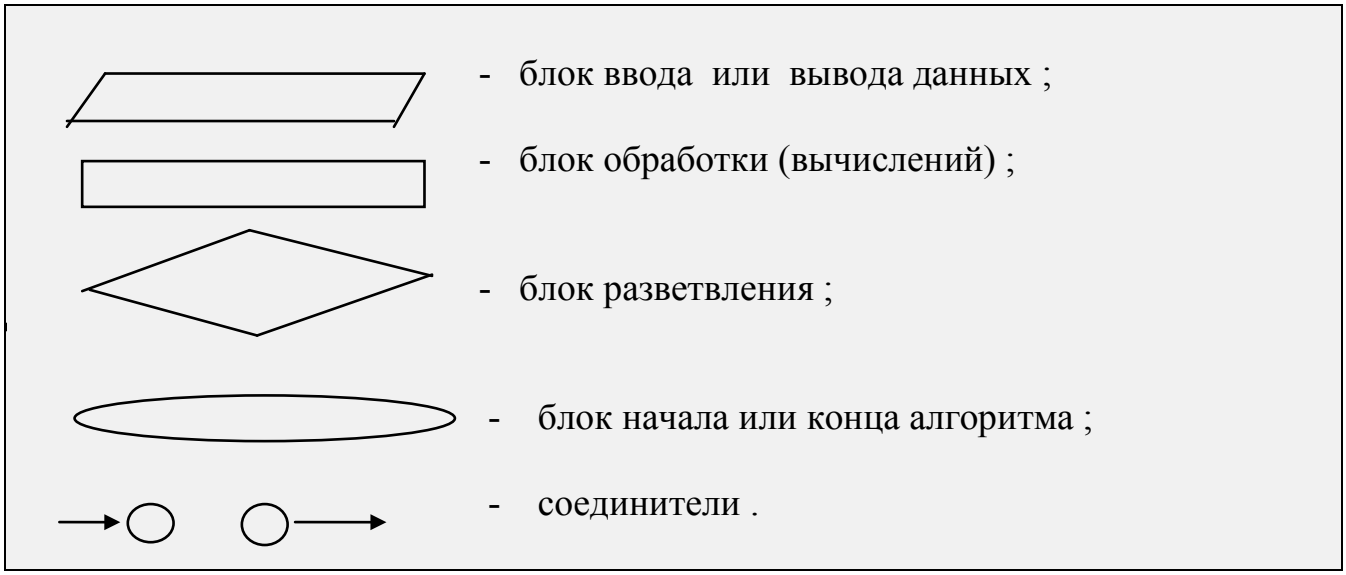
Введите зарплату =

В ответ на подсказку пользователь должен набрать число, например 1000, и нажать клавишу **Enter**. При этом выполнится процедура **Readln**. В результате работы программы на экране будет напечатан следующий текст

Для зарплаты 1000.00 рублей налог= 120.00 руб.

Заметим, что в списке вывода могут присутствовать не только выводимые переменные, константы и выражения, но и их ***форматы*** : первое целое число после двоеточия указывает ширину поля вывода элемента списка (в знакоместах), а следующее целое - сколько цифр предусмотреть в дробной части вещественного числа. Для целых и символьных типов данных в формате задается только ширина отводимого поля.

Алгоритм - последовательность действий, приводящая к намеченному результату. Алгоритм можно записать в виде текста на обычном языке или в виде программы на любом из алгоритмических языков, а также изобразить в графическом виде как блок-схему, используя следующие типовые блоки :



Алгоритм называется ***линейным***, если его блок-схему можно представить в виде однонаправленной линии блоков. Блоки выполняются последовательно друг за другом. Рассмотренная программа для вычисления подоходного налога является примером алгоритма линейной структуры.

3. ПРОГРАММИРОВАНИЕ ТИПОВЫХ АЛГОРИТМОВ РАЗВЕТВЛЕННОЙ СТРУКТУРЫ. ОПЕРАТОРЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

3.1. Операторы управления

Операторы, записанные в основном блоке программы или процедуры, выполняются в том порядке, как они следуют в программе, начиная с первого за ключевым словом **Begin** оператора до тех пор, пока не встретится *оператор* передачи *управления* очередностью решения задачи. К их числу относятся: *оператор безусловной передачи управления* **Goto**, *оператор условной передачи управления* **If ... Then ... Else**, *оператор выбора* **Case**, а также *операторы цикла* **Repeat ... Until, While, For** и другие. Кроме того, пользователь при программировании алгоритма может объединять ряд последовательно записанных операторов в один *составной оператор (блок)*. Для этого достаточно группу операторов заключить в операторные скобки **Begin ... End**. Все операторы, как предложения языка **TP**, отделяются друг от друга символом **;** .

3.2. Оператор безусловной передачи управления Goto (идти)

Форма записи оператора безусловного перехода:

```
Goto метка ;
```

где **Goto** - зарезервированное слово, *метка* - идентификатор или номер оператора, которому программист желает передать управление. В случае необходимости приписывания оператору программы метки требуется объявить все существующие метки в разделе объявлений **Label**. Метка отделяется от оператора, на который передается управление, двоеточием **:** . Заметим, что стандартная процедура **Halt** завершает выполнение программы, а процедура **Exit** осуществляет выход из цикла либо из процедуры и функции в вызывающий ее блок (на уровень выше). Язык **TP** имеет богатый набор конструкций управления, поэтому оператор **Goto** редко используется при написании программ, т.к. его использование затрудняет прослеживание выполнения алгоритма в программе, ее прочтение.

3.3. Оператор условной передачи управления If (если)

Оператор условного перехода имеет вид:

```
If   условие
  Then
    Begin
    оператор 1 ветви Then
    оператор 2 ветви Then
        ..................…                    { блок    Then }
    оператор n ветви Then
    End
   Else
    Begin
    оператор 1 ветви Else
    оператор 2 ветви Else
       .....                                   { блок    Else }
     оператор m ветви Else
     End;
```

Выполнение оператора **If** начинается с проверки условия. Если условие выполняется (значение выражения *условие* = true), то выполняется группа операторов блока **Then** (*затем*), а группа операторов блока **Else** (*иначе*) опускается. В противном случае, т.е. если условие не выполняется (значение выражения *условие* = false), наоборот, выполняется группа операторов блока **Else**, а группа операторов блока **Then** игнорируется. Блок **Else** может отсутствовать, тогда, если условие не выполняется (значение false), управление

передается оператору, следующему в программе за оператором **If**. Заметим, что если составной оператор Begin .. End содержит только один оператор, то операторные скобки Begin и End можно опустить.

Заметим, что операторы **If** допускают любую степень вложенности, т.е. операторы блоков **Then** и **Else** в свою очередь могут сами являться операторами **If** .

3.4. Оператор выбора Case

Форма записи оператора выбора **Case**:

```
Case  переменная     Of
  значение_1: Begin
                оператор_1_1 ;
                оператор_1_2 ;
                  ..........…                  {     Блок  1     }
                 оператор_1_ m ;
                 End;
  значение_2: Begin
                оператор_2_1  ;
                оператор_2_2  ;
                  .....                        {     Блок  2     }
                оператор_2_ m ;
                End;
              .....
  значение_n:  Begin
                оператор_n_1 ;
                оператор_n_2 ;
                 .....                         {     Блок  n     }
                 оператор_n_m ;
                 End;
       Else
            Begin
            оператор_n+1_1 ;
            оператор_n+1_2 ;
             .....                             {     Блок  n+1   }
            оператор_n+1_m  ;
            End;
End ;
```

Оператор выбора **Case** предназначен для выполнения одного из многих (в данном случае из n+1-го) вариантов программных блоков в зависимости от

значения управляющей переменной оператора **Case**, которая должна иметь порядковый тип. Если перед выполнением оператора **Case** *переменная* имела *значение*_2, то выполняется только Блок 2 операторов (остальные блоки не выполняются, управление после выполнения Блока 2 передается оператору, следующему за оператором **Case**. Аналогично выполняется оператор **Case** в случае, когда значение *переменной* равно одному из значений *значение*_1 ... *значение*_n. Если значение *переменной* не совпадает ни с одним из указанных значений, выполняется блок ветви **Else**. Заметим, что, как и в случае оператора **If**, ветвь **Else** может отсутствовать. Если при этом значение *переменной* не совпадает ни с одним из значений списка *значение*_1 ... *значение_n*, то управление передается оператору, следующему за оператором **Case,** причем не выполняется ни один из *n* блоков.

3.5. Программирование типовых алгоритмов разветвленной структуры

Пример 1. Пусть требуется по данной зарплате *zp* вычислить величину подоходного налога *pn* для каждого работника предприятия по прогрессивной шкале налогообложения:

$$Pn = \begin{cases} f1(zp) = 0, & \text{если} \quad zp \le 3mz \\ f2(zp) = 0.12(zp - 3mz), & \text{если } 3mz \le zp \le 10mz \\ f3(zp) = 0.2(zp - 10mz) + 0.12 * 7 * mz, & \text{если} \quad zp \ge 10mz \end{cases},$$

Здесь введены обозначения: mz - минимальная зарплата (83,49 руб.); коэффициент 0.12 соответствует 12% налога при зарплате в интервале от 3mz до 10mz; коэффициент 0.2 соответствует 20% налога на превышение зарплаты над 10mz. Для расчета размеров налога для каждого сотрудника введем его порядковый номер (i=1,2,3,...). Для прекращения обработки зарплат сотрудников условимся вводить i=0. Один из возможных вариантов решения задачи представлен программой, ориентированной на сокращенный оператор **If** .

```
Program Nalog;          { заголовок программы  }
Const  mz = 83.49;      { раздел объявления констант }
          p1 = 0.12;
          p2 = 0.2;
Var  i: integer;            { раздел объявления переменных}
       zp, pn : real;
BEGIN                              { основной блок программы }
write   (‘Введите номер и зарплату сотрудника= ’);
readln (i,zp);              { ввод с клавиатуры через пробел   i   и  zp }
if  (   i = 0  )
  then
     begin
      writeln    (‘ конец работы    ’);
      halt;
```

```
    end;
  else
      if    zp  <=   3 * mz
        then  pn :=  0;
      if  ( 3 *  mz <= zp ) and  (  zp <= 10 * mz )
        then  pn := p1* ( zp - 3 *  mz ) ;
      if  zp >10 *  mz
        then   pn := p2 * (  zp - 10 * mz ) + 7 * p1 *  mz;
writeln ('Для номера =' , i , 'зарплата =', zp:10:2, ' и подоходный налог='   ,  pn : 10
: 2 );
END.
```

Заметим, что если изменится минимальная зарплата или процентные ставки, то в программе достаточно изменить данные в разделе объявления констант.

4. ПРОГРАММИРОВАНИЕ ТИПОВЫХ АЛГОРИТМОВ ЦИКЛИЧЕСКОЙ СТРУКТУРЫ. ОПЕРАТОРЫ ЦИКЛА

4.1. Простой цикл

Алгоритм имеет *циклическую* структуру, если содержит повторяющуюся последовательность действий (цикл). Для записи цикла можно использовать оператор цикла с предусловием **While … Do** или оператор цикла с послеусловием **Repeat … Until** или оператор цикла с параметром **For….Do**. Структура записи первых двух операторов имеет вид:

```
Repeat
  оператор 1        { начало тела цикла  }
  оператор 2
  .....................
   оператор m       { конец тела цикла    }
Until условие   ;
```

или

```
While  условие Do
   Begin
   оператор 1    { начало тела цикла }
   оператор 2
    .....................
   оператор m  { конец тела цикла }
   End;      .
```

В случае цикла типа **While** (*пока)* проверка производится *перед* выполнением тела цикла и цикл выполнятся, пока указанное после ключевого слова **While** условие выполняется (логическое выражение имеет значение true - **истина**).

В случае цикла **Repeat ... Until (***повторять до выполнения условия*) проверка условия, записанного после ключевого слова **Until**, осуществляется *после* выполнения тела цикла и его **повторение** продолжается, пока условие принимает значение false (**ложь**). Выход из цикла осуществляется, как только условие примет значение true (**истина**). Заметим , что в случае использования цикла **While** тело цикла может НЕ исполниться НИ РАЗУ (если при входе в цикл проверяемое условие примет значение false - **ложь**), а в случае цикла **Repeat** тело цикла исполнится ПО КРАЙНЕЙ МЕРЕ ОДИН РАЗ, даже если проверяемое после тела цикла условие сразу примет значение true (**истина**) и цикл будет завершен.

Если тело цикла **While** содержит только один оператор, операторные скобки Begin ... End можно опустить. Заметим, что в случае цикла **Repeat ... Until** операторные скобки Begin ... End вовсе не используются. Их роль играют ключевые слова Repeat ... Until, обрамляющие тело цикла.

Рассмотрим пример программы вычисления таблицы функции $y=\sin(x)$ на промежутке *[xn;xk]* с шагом *hx*.

```
Program    Cycl;
Var    x, xn, xk, hx, y : Real;
       i, n :  Integer;                    {i -номер строки таблицы }
BEGIN                                          {n+1 -число строк таблицы}
write (‘    Введите xn, xk, n = ’)  ;
readln (xn, xk, n);
hx := (xk - xn) / n;
x := xn; i := 0;
writeln (‘  ТАБЛИЦА ФУНКЦИИ’:20); { вывод на экран заголовка }
writeln;                                          { вывод пустой строки }
writeln(‘ Номер ’:10,’ X ’: 10,’ Y ’:10);  {заголовки колонок таблицы }
Repeat
  y := sin (x);
  writeln (i:10, x:10:2; y:10:2);
  x := x + hx; i := i + 1;
Until x > xk;
END .
```

Та же программа может быть записана с использованием оператора цикла **While**. Заменим в программе *Cycle* фрагмент, заключенный между ключевыми словами Repeat ... Until на следующий:

```
While  x <= xk  Do
```

```
  Begin
  y := sin (x);
   writeln (i,  x : 10 : 2 ,  y : 10 :2 );
   x := x + hx;
   i := i + 1   { или Inc ( i ) ; }
   End ;
```

В случае, когда управляющая переменная цикла является переменной целого типа (например **integer**), изменяющейся с шагом 1 или -1, например как номер i, удобнее воспользоваться циклом **For** (*для*):

```
For  i := in To ik  Do     {шаг управляющей переменной цикла = +1 }
  Begin
  оператор_1
  оператор_2
   .............
  оператор_m
  End ;
```

или

```
For i := ik Downto in Do  {шаг управляющей переменной цикла= -1 }
   Begin
   оператор_1
   оператор_2
    .............
    оператор_m
    End;                          .
```

В первом случае управляющая переменная цикла i на каждом шаге цикла автоматически увеличивается на единицу от начального **in** до конечного **ik**, во втором случае управляющая переменная цикла автоматически уменьшается на единицу.

В качестве примера применения цикла **For** запишем для программы *Cycle* вместо оператора **Repeat …. Until** оператор **For …. To…Do**:

```
For i :=0 To n Do
 Begin
 y := sin(x);
 writeln(i, x:10:2, y:10:2);
 x := x + hx;
 End;
```

4.2. Вложенные циклы

Часто решение практических задач приводит к алгоритмам, в которых в качестве тела цикла по одной переменной необходимо использовать цикл по другой переменной. Такие циклы называются вложенными.

Типовая задача на вложенные циклы. Пусть показатель качества *P* объекта проектирования зависит от величин *x* и *a* по формуле $P = f(x,a)$. Величины *x* и *a* можно изменять в данных промежутках [*xn, xk*] и [*an, ak*] с данными шагами *hx, ha* соответственно. Требуется вычислить таблицу значений *P* для ***всех вариантов*** изменения значений *x* и *a*.

Идея алгоритма перебора всех вариантов изменения значений двух переменных *x* и *a* состоит в следующем. Переменной *a* присвоим начальное значение *a:=an* и будем вычислять и печатать первый блок таблицы значений *(P, x, a)* при изменении переменной *x* по простому циклу от *xn* до *xk* с шагом *hx*. Далее увеличим *a* на шаг *a:=a+ha* и снова повторим цикл по переменной *x*, вычислив и распечатав второй блок искомой таблицы, и так до тех пор, пока не будет распечатана вся таблица при увеличении *a* от начального *an* до конечного *ak*. Переменная *a* называется переменной внешнего цикла, а переменная *x* - переменной внутреннего цикла. Тогда, используя оператор **Repeat …. Until ,** программу можно записать:

```
Program      Cycle_in_cycle;
Var  a, an, ak, ha    : real;
       x, xn, xk, hx, P : real;
BEGIN
write (‘Введите an, ak, ha=’);
readln (an, ak, ha);
write (‘Введите xn, xk, hx=’);
readln (xn, xk, hx);
writeln (‘Таблица функции 2-х переменных’:30);writeln;
writeln(‘X’:5,’A’:15,’P’:25);writeln;
a := an;
Repeat
  x := xn;
  Repeat
     P := sin (x + a);
`    write (x : 10, a : 10, P : 10);
     x := x +hx;
  Until x > xk;       writeln;
  a := a + ha;
Until a > ak;
END .
```

В случае использования оператора цикла **While** программа имеет вид:

```
a := an;
While a <= ak Do
  Begin
   x := xn;
   While x <= xk Do
       Begin
        P := sin (x + a);
        writeln ( x:10, a:10, P:10);
        x := x + hx;
       End;            writeln;
        a := a + ha;
  End;
```

5. СТРУКТУРИРОВАННЫЕ ТИПЫ ДАННЫХ В ВИДЕ МАССИВОВ. ПРОГРАММИРОВАНИЕ ТИПОВЫХ АЛГОРИТМОВ

Массив - это упорядоченный набор однотипных данных, объединенных одним именем, но отличающихся одним или несколькими номерами (индексами), изменяющимися от начального до конечного значения. Адрес каждого элемента массива в памяти задается именем массива с указанием индексов, т.е. индексированной переменной.

5.1. Одномерные массивы. Типовые задачи

В информатике массив называется *одномерным*, если положение каждого элемента в массиве указывается одним индексом.

Объявление массива в *разделе объявлений* программы на **TP** можно выполнить в разделе **Type** или в разделе **Var**. Формат оператора описания переменной **Var** для случая объявления массива следующий:

Var *имя _массива* **: Array** [*нач. индекс* **..** *кон. индекс*] **Of** *тип_данных*.

Здесь *нач. индекс* и *кон. индекс* - граничные значения изменения индекса массива, это переменные или константы любого целого типа, разделенные знаком двоеточие **. .** .

Например, массив, содержащий величины вкладов в банке для 1000 клиентов, может быть описан в разделе объявлений:

```
Const n=10;
Var
       Vklad : array [1..1000] of real;          {Массив вкладов}
       X : array [1..3] of  real;                {Массив X }
       Y:  array [1..n] of real;
       Name : array [1..1000} of string[25];
```

Задать оператор обработки каждого элемента массива можно, указав конкретное значение индекса в квадратных скобках вслед за именем массива. Например, оператор Vklad[10] := 1000 позволяет присвоить десятому элементу массива Vklad значение 1000. Обработку всех элементов массива можно задать, используя переменный индекс i , изменяя его в нужных пределах оператором **For... To ... Do**. Например, для того чтобы очистить массив Vklad, т.е. занулить все его элементы, достаточно в основном разделе программы (в ее исполняемой части) записать :

```
For  i := 1 To 1000  Do  Vklad[i] := 0
```

В **ТП** 7.0 существуют операции, позволяющие производить некоторые действия с массивом в целом. Ниже приведен фрагмент программы, которая создает одномерный массив из целых чисел, состоящий из 11 элементов, величины которых есть случайные числа от нуля до ста (для их задания мы воспользовались стандартной функцией Паскаля **random**). Формирование массива **a** осуществляется посредством оператора **For ...To...Do**, причем к каждому элементу массива обращение производится по имени массива с указанием вслед за именем в квадратных скобках индекса элемента. После того как сформирован массив **a** , мы воспользовались одной из немногих матричных операций **ТП 7.0**, скопировав массив **a** в массив **b** .

```
Const m = 10;
Type
        matrix: Array [0..m] Of  integer;
Var     i: integer;
        a, b: matrix;
BEGIN
  for i := 0 to  m  do  a [i] := Random(100);
  b := a;
END.
```

Заметим, что этот матричный оператор **b := a**; эквивалентен следующему циклу на Паскале, где присвоение значений производится поэлементно:

```
for   i := 0   to   m   do    b[i] := a[i];       .
```

Типовая задача 1 (Суммирование массива). Необходимо составить программу на языке **ТП 7.0**, которая должна предусматривать ввод 1000 чисел-значений массива *Vklad* для вкладов 1000 клиентов банка, контрольный вывод массива, суммирование денег всех вкладчиков банка и вывод результата.

```
Program Sum_Vklad;
Const n = 1000;
```

```
Var    Vklad : array [1 .. n] of real;
       Sum`: real;
       i: integer;
BEGIN
for i := 1 to n do
   begin
      write (‘Введите ‘,i : 4,’-й вклад’); readln (Vklad[i]);
  end;
writeln (‘КОНТРОЛЬНЫЙ  ВЫВОД ВКЛАДОВ’ : 30);
for i := 1 to n do
  writeln (‘Номер=’, i : 4, ‘ Вклад=’, Vklad[i]);
Sum := 0;      { начальное присвоение  значения суммы }
for i := 1 to n do            { блок суммирования }
  Sum := Sum + Vklad[i];
writeln (‘СУММА ВКЛАДОВ БАНКА =’, Sum);
END.
```

Типовая задача 2 (Вычисление массивов). В налоговой инспекции зарегистрированы 500 деклараций, в которых указаны величина годового дохода *doh* и уплаченный за год подоходный налог *nalog_upl*. Требуется начислить подоходный налог согласно следующему законодательству по формуле

$$nalog\_zakon = \begin{cases} 0.12 * doh, & \text{если } doh \le 50\,000, \\ 0.3 * doh, & \text{если } doh > 50\,000, \end{cases}$$

а также коррекцию к уплате налога по формуле

$$nalog\_cor = nalog\_treb - nalog\_upl.$$

```
Program    Dohodi;
Const n = 500;
Type          Vector = array [ 1 .. n ] of real;
Var    doh, nalog_upl, nalog_treb, nalog_cor : Vector;
                i: integer;
BEGIN        { Начало основного блока программы }
for i := 1 to n do     { начало блока ввода исходных данных }
   begin
   write (‘Введите’, i:4,‘-й доход и уплаченный налог через пробел’);
   readln(doh [i], nalog_upl [i]);
   end;         { конец ввода }
for i := 1 to n do
  begin
   if doh [i] <= 50 000
```

```
      then
nalog_treb := 0.12 * doh [i]
      else
        nalog_treb := 0.3   * doh [i];
  nalog_cor [i] := nalog_treb [i] - nalog_upl [i]
  end;{ конец блока вычислений}
writeln (‘ВЕДОМОСТЬ’ : 30);
writeln (‘ Номер ’ : 5, ‘ Доход ‘: 10,’ Налог упл.’ : 10,’ Налог начисл.’ :10,
‘Коррекция’ : 10);
for i := 1 to n do
writeln(i:5, doh[i]: 10, nalog_upl [i]: 10, nalog_treb [i]: 10, nalog_cor [i]: 10);
END .{ конец программы }
```

Типовая задача 3 (***Последовательный поиск в символьном массиве***). Необходимо разработать программу, которая обеспечит ввод с клавиатуры 80-ти символов и сохранение их в одномерном массиве *symbols*. Затем программа должна запросить ввод еще одного символа *symbol*, проверить, содержит ли ранее введенный массив *symbols* символ *symbol* и сколько раз, а также выдать соответствующее сообщение.

```
Program      Poisk;
Const n = 80;
Var    i: integer;     { i - рабочая переменная цикла }
       k: integer;    {k-счетчик количества вхождений символа в массив }
       symbol:  char;
       symbols: array [1 .. n] of char;
BEGIN
for i := 1 to n do
  begin
  write (i, ‘ - й элемент массива=’);
  readln (symbols [i] )
   end;
writeln (‘ МАССИВ : ‘);
for i := 1 to n do { контрольный вывод введенного массива - строки}
   write(symbols [i] );      writeln ;
write (‘ СИМВОЛ для поиска в массиве=’);        readln (symbol);
 k := 0;        { начало поиска }
 for i := 1 to n do
   if symbols [i] = symbol
      then
        begin
        k := k + 1;
```

```
        writeln (‘обнаружено’, k,’-е вхождение литеры ‘, symbol,’ в ‘, i ,’-й позиции’);
        end ;
if k <> 0       { если k не равно нулю }
  then   { то }
     writeln (‘ИТОГО :  СИМВОЛ ’, symbol, ‘ ВСТРЕЧАЕТСЯ ‘, k, ‘РАЗ’)
  else { иначе }
     writeln (‘СИМВОЛ ‘,symbol, ‘ НЕ ВСТРЕЧАЕТСЯ НИ РАЗУ’);
END.
```

5.2. Многомерные массивы. Матрицы

Массив называется *многомерным*, если положение каждого элемента в массиве указывается более чем одним индексом. В случае индексации элемента в массиве двумя индексами массив называется *двумерным*. Примером двумерного массива чисел является *матрица* чисел **a** размером 2 на 3:

$$a = \begin{pmatrix} 7 & 3 & -2 \\ 4 & 5 & 7 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

В примере используются принятые в математике индексированные переменные вида $\mathbf{a_{i,j}}$, где **i**-текущий номер строки, **j**- текущий номер столбца матрицы, на пересечении которых находится переменная. Объявление массива **a** [2 x 3] выгдядит следующим образом:

```
Var
        a : array [1..2, 1..3] of integer;
```

Это описание эквивалентно следующему:

```
Const n = 2;  m = 3;
Type
  row  = array [1..m ] of integer;
  Matrix = array [1..n ]  of row;
Var
  a: Matrix;
```

К индексированным переменным можно обращаться следующим образом: **a** [1, 1], **a**[i,j], **a**[2*i+1,j-1]. Для использования индексированной переменной в программе необходимо предусмотреть, чтобы к моменту обращения к ней индексы были определены или вычислены.

Для поэлементной обработки всей матрицы в прогамме необходимо предусмотреть использование переменных индексов **i** и **j** , а также их изменение от начальных значений до конечных. В случае выбора очередности обработки матрицы *‘по строкам’* в программе используется алгоритм вложенных циклов , у которого переменной внешнего цикла является номер строки **i** , а переменной

внутреннего цикла - номер столбца **j** .В случае необходимости обработки матрицы по*‘столбцам‘*- вложенность циклов обратная: внешний цикл по **j** , а внутренний цикл по **i** .

Пример построчного ввода матрицы:

```
Const  n  = 2;          { Количество строк матрицы }
       m = 3; { Количество столбцов матрицы }
Var    i, j: integer;
       a : array[1..n, 1..m] of real;
Begin
writeLn (‘ВВЕДИТЕ МАТРИЦУ размером 2 на 3:’);
for i := 1 to n do
  begin
  for j := 1 to m do
    read ( a [ i, j ] );
  writedLn   {Для перевода курсора на начало новой строки }
  end;
End;
```

Пример построчного вывода элементов матрицы:

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

Типовая задача 4 (Вычисление и суммирование матриц). Малое предприятие в своей структуре имеет 4 отдела по 10 работников в каждом. Зарплаты всех сотрудников занесены в таблицу - матрицу размером 4 строки по 10 чисел-зарплат в каждой строке.

Требуется начислить матрицу налогов, взымаемых со всех работников предприятия, матрицу зарплат к выдаче за вычетом налогов, а также сумму к выдаче по итогам месяца, если прогрессивный подоходный налог исчисляется от зарплаты по следующей формуле:

$$pn = \begin{cases} f1(zp) = 0, & \text{если } zp \le 3mz; \\ f2(zp) = 0.12 * (zp - 3 * mz), & \text{если } 3mz \le zp \le 10mz; \\ f3(zp) = 0.2 * (zp-10*mz)+7*0.12*mz, & \text{если } zp \ge 10mz ; \end{cases}$$

Введем обозначения - идентификаторы:

mz - минимальная зарплата;

zp - имя массива начисленных зарплат;

pn - имя массива подоходных налогов;

zpv - имя массива зарплат к выдаче за вычетом налога, каждый элемент которого вычисляется по формуле: zpv[i,j] = zp[i,j] - pn[i,j];

Sum - сумма денег к выдаче ,

где i - номер отдела, индекс строки матрицы,

j - табельный номер в отделах, индекс столбца матрицы.

```
Program    Zarplata;
Const m = 4; n  = 10;    mz = 83.49;
Var    zp, zpv, pn: array [1..m,1..n] of real;
       i,j: integer;
       Sum: real;
BEGIN
for i := 1 to m do            { блок ввода зарплат }
  for j := 1 to n do
    begin
    write (‘Введите зарплату:‘, i:2,‘-й отдел, табельный номер’,j:3,’=‘);
    readln (zp[i,j]);
    end;
Sum := 0; { блок вычислений  }
for i := 1 to m do
   for j := 1 to n do
       begin
 if zp [i, j] <= 3 * mz
            then
               pn [i , j] := 0
            else
                if zp [i, j] <= 10 * mz
                   then
                      pn[i, j]:=0.12*(zp [i, j]-3*mz)
                         else
                      pn[i,j]:=0.2*(zp[i,j]-10*mz)+0.84*mz ;
               zpv [i, j] := zp [i, j] - pn [i, j] ;  { Вычитание матриц}
               sum := sum + zpv [i, j] ;  { Суммирование массива}
               end ;
writeln (‘ВЕДОМОСТЬ ‘ : 40 ); writeln ;     { Блок вывода ведомости }
writeln (‘ Отдел ’,’ Номер ’,’ Зарплата’ : 10,’ Налог’ : 10, ‘ Зарплата к    выдаче’ :16
, ‘ Подпись’:10) ; writeln ;
```

```
for i:=1 to m do
        for j:=1 to n do
        writeln(i:5, j:6, zp[i,j]:10, pn[i,j]:10, zpv[i,j]:16, ’          ‘:10) ;
writeln (‘ ИТОГО К ВЫДАЧЕ : ‘, Sum : 14) ;
END.
```

5.3. Типовые программные блоки обработки матриц

Рассмотрим программирование других типовых операций обработки матриц на примере матрицы чисел **a** [2 x 3]. Не останавливаясь на других разделах программы (объявления, ввод матрицы, вывод результатов), запишем следующие программные блоки.

Поиск максимального элемента матрицы amax и его расположения (**imax, jmax**) выполняется путем поэлементного перебора матрицы. Перед началом процесса поиска переменной **amax** присваивается в качестве начального значения заведомо *меньшая* величина. На каждом шаге очередной элемент матрицы сравнивается с искомой величиной **amax**. Если очередное значение **a**[i,j] превышает **amax**, это значение пересылается в **amax** , в противном случае просто переходим к очередному элементу матрицы и снова сравниваем **amax** и **a** [i,j]. Поскольку вначале мы присвоили **amax** заведомо меньшее значение, то на первом же шаге перебора элементов матрицы элемент **a**[1,1] пересылается на место старого значения **amax** . Таким образом, в переменной **amax** после окончания перебора будет находиться максимальный элемент матрицы. Следующий фрагмент программы реализует выше описанный алгоритм:

```
amax:=-1E+30;   {начальное присвоение amax заведомо малого значения }
for i := 1 to 2 do   {блок поиска максимального элемента матрицы}
        for j := 1 to 3 do
          if a [i, j] > amax
             then    begin  amax := a [i, j];imax:=i ; jmax :=j  end;
```

Заметим, что в случае поиска минимума в матрице, в этом фрагменте потребуется заменить два оператора:

1) amax := -1E+30 заменим на amin := +1E+30 (заведомо большое число);

2) в операторе **If** поменяется знак неравенства:

```
  if   a [i, j] < amin    then     amin := a [i,j] ;
```

.

Переставить первую и вторую строки матрицы **a[2 x 3]:**

```
for j := 1 to 3 do              {блок обмена данных в двух строках}
  begin
  r := a [1, j];                {r- рабочая ячейка (переменная) памяти}
  a [1, j] := a [2, j];
```

```
  a [2, j] := r
  end;
```

Перестановка столбцов матрицы.

Переставить 2-й и 3-й столбцы матрицы **a** [2, 3].

```
for i := 1 to 2 do          {Блок перестановки столбцов матрицы }
  begin
  r := a [i,2];
  a [i, 2] := a [i, 3];
  a [i, 3] := r
  end;
```

***Просуммировать матрицу* a[2x3] *по строкам*, вычислив массив b[2] :**

```
for  i  := 1 to 2 do        { блок суммирования по строкам }
  begin
  s := 0;                   {s - рабочая ячейка для сумм}
  for j := 1 to 3 do
     s := s + a [i, j];
  b [i] := s
  end;
```

***Просуммировать матрицу* a [2 x 3] *по столбцам, вычислив массив* c[3]** :

```
for j := 1 to 3 do          { блок суммирования по столбцам  }
  begin
  s := 0;
  for  i  := 1 to 2 do
     s := s + a [i, j];
  c[j] := s;
  end;
```

6. ПРОЦЕДУРЫ И ФУНКЦИИ

6.1. Стандартные процедуры и функции

Малые программы пользователь записывает обычно в виде единой программы. Более сложные программы, насчитывающие сотни и тысячи операторов, составляются с использованием процедур и функций. При этом общая задача разделяется на ряд частных задач, каждая из которых может быть записана в виде подпрограммы, называемой в **ТР**: **процедура** или **функция**. Сокращение объема всей программы может производиться за счет многократного вызова одной процедуры при разных данных. Вызов процедуры или функции может занимать в программе одну строку, в то время как сама подпрограмма

содержать тысячи строк. Если процедура или функция используется многократно в разных программах, то такие процедуры или функции могут быть помещены в библиотеки стандартных модулей. Указание в программе об использовании файлов стандартных модулей пользователь делает в подразделе объявлений **Users**. Например, для очистки экрана используется стандартная процедура **ClrScr**, хранящаяся в модуле **Crt** ,о чем объявляется в программе,например:

```
Program   Primer;
Uses  Crt;
BEGIN
  ClrScr;      { Вызов стандартной процедуры}
END.
```

Другим примером использования стандартных процедур и функций являются известные процедуры ввода/вывода **Read** и **Write**, а также библиотечные функции **sin(x),ln(x),exp(x)** и др. Эти стандартные модули хранятся в разделе **System** библиотечных файлов объектных модулей , который автоматически используется компилятором без его объявления в программе в подразделе **Users.**

Процедуры и функции могут содержать или нет *аргументы-параметры*, записываемые в скобках после имени процедуры.Так, в процедуре **Read**(*a,x*) предусмотрены в качестве параметров переменные *a* и *x* для ввода в них данных. В процедуре **Write(** ‘ОТВЕТ : x=’, x, ‘ y=’ , y) параметрами являются :

‘ОТВЕТ : x=’ { строка символов }
x {переменная }
‘ y=’ {строка символов }
y {переменная }.

Процедуры , записанные в форме **WriteLn , ClrScr , Pi, –** параметров не содержат. *Параметры-аргументы* у функций используются для передачи исходных данных для вычисления функции, например, *x* для **sin(*x*)** . Параметры (‘ ОТВЕТ : x=’, x, ‘ y=’ , y) процедуры **Write** (‘ ОТВЕТ : x=’, x, ‘ y=’ , y) - предназначаются пользователем для вывода результатов, причем количество параметров у процедуры **Write** пользователь может задать для ее вызова произвольно. У других процедур параметры могут быть установлены однозначно и по их количеству, и по их назначению, и по порядку записи. Часть из них используется для передачи данных в память, связанную с процедурой , другая часть предназначается для передачи результатов из процедуры в память основной программы.

6.2. Нестандартные процедуры и функции

Нестандартная (*пользовательская*) процедура или функция представляет собой относительно самостоятельный фрагмент программы пользователя, предназначенный для решения определенной части общей задачи программы.

Каждая процедура или функция существует в программе в единственном экземпляре, в то время как обращаться к ней можно из разных точек программы многократно при разных исходных данных. Вызов для исполнения нестандартной процедуры в основной программе записывается в виде оператора вызова процедуры:

Имя процедуры (*Список фактических параметров*);

т.е. так же, как и вызов стандартной процедуры. Здесь параметры называются *фактическими,* так как их значения должны быть определены конкретными значениями либо перед вызовом процедуры (для параметров- исходных данных) либо после выполнения процедуры (для параметров- результатов выполнения процедуры).

Каждая процедура записывается в разделе объявлений основной программы по той же структуре, что и основная программа. Структура описания процедуры выглядит следующим образом:

```
Procedure  Имя процедуры  (  Список формальных параметров);
Label                ... ;     {объявление локальных  меток}
Const                ... ;     { объявление локальных  констант}
Type                 ... ;     { объявление локальных  типов данных}
Var                ... ;  { объявление локальных  переменных}
Procedure  .....;  { описание внутренних процедур или }
                     {описание внутренних функций Function }
Begin
  .....
  .....            { операторы тела процедуры }
 End;
```

Переменная, объявленная внутри процедуры или функции, называется ***локальной.*** С ней можно оперировать только внутри этой процедуры или функции. Переменная, объявленная в разделе объявлений основной программы, называется ***глобальной,*** и с ней можно оперировать как и внутри процедур и функций, используемых в программе, так и вне их, в любом месте программы. С помощью значений глобальных переменных можно передавать информацию из процедуры в программу и обратно ***без использования параметров процедуры.*** Локальные и глобальные переменные могут иметь одинаковые имена, однако это разные переменные. Любое обращение к таким переменным в теле процедуры или функции трактуется как обращение к локальным переменным, т.е. глобальные переменные в этом случае попросту недоступны

Описание функций похоже на описание процедуры. Только необходимо заменить

Procedure **Имя процедуры** (*список формальных параметров*);

на

Function **Имя функции**(*список формальных параметров*):**тип результата**;

Отличие функции от процедуры заключается и в том, что результатом исполнения операторов, образующих тело функции, всегда является некоторое единственное значение простого, сторокового типа или указателя. Поэтому обращение к функции можно использовать в соответствующих выражениях наряду с переменными и константами. Среди операторов, составляющих тело функции, должен присутствовать оператор присваивания идентификатору функции результата вычислений, который будет возвращаться как значение функции при выходе из нее.

Описание вызова процедуры или функции осуществляется простым упоминанием имени процедуры или функции в основном блоке программы или в другой процедуре с указанием фактических параметров на месте формальных.

При выполнении программы на этапе вызова процедуры или функции начинают выполнятся входящие в нее операторы. После выполнения последнего из них управление возвращается в основную программу и выполняются операторы, стоящие непосредственно за оператором вызова процедуры или функции.

Задача 5. Дан одномерный массив-вектор размерностью *n,* элементами которого являются действительные числа. Найти сумму элементов вектора. Программа, использующая для решения **процедуру**, имеет следующий вид:

```
Program Summirovanie;
Const   n =  20;     { длина массива }
Type  tvector = array [1 .. n] of real;
Var    Vector: tvector;
                i: integer;
Procedure  Summa(vec: tvector; m: integer; name: string);
Var  i: integer; s: real;
Begin
s:= 0;
for  i :=  1 to m do
  s := s + vec [i];
writeln ('Сумма элементов вектора ', name, ' = ', S: 7: 2);
End;          { конец процедуры }
BEGIN              { начало программы}
writeln (' Введите, чередуя через пробел, элементы массива = ');
for i := 1 to n do
  read (Vector [i]);
readln;
Summa(Vector, n, 'Vector' );{Вызов процедуры -отдельным оператором}
```

```
END.
```

Эта же программа, использующая **процедуру–функцию**:

```
Program  Summirovanie_Fun;
Const    n =  20; { длина массива }
Type   tvector = array [1 .. n] of real;
Var    Vector: tvector;
                i: integer; Sum: real;
Function Summa(vec: tvector; m: integer): real ;
Var  i: integer; s: real;
Begin
s:= 0;
for  i :=  1 to m do
   s := s + vec [i];
Summa := s{ Присваивание имени функции значения }
End; { конец процедуры-функции }
BEGIN        { начало основной программы}
writeln (‘ Введите данные массива vector [n]: ‘);
for i := 1 to n do
        read (vector [i]);
readln;
Sum :=  Summa ( vector, n  );    {Вызов процедуры - отдельным оператором}
writeln (‘ Сумма элементов вектора vector =‘, Sum : 7 : 2);  { или }
writeln (‘Сумма элементов вектора vector = ‘, Summa ( Vector, n ) );
END.
```

6.3. Формальные и фактические параметры. Способы передачи данных в процедуры и функции

Параметры, указываемые в заголовке процедуры или функции при ее описании, называются *формальными параметрами***.** Список формальных параметров необязателен и может отсутствовать. Если же он есть, то все переменные из этого списка могут использоваться в любых выражениях внутри процедуры или функции. При вызове процедуры или функции ей нужно указать уже конкретные параметры. При передаче необходимо следить, чтобы в операторе вызова были указаны все параметры, описанные в заголовке процедуры или функции. Параметры, указываемые при вызове процедуры или функции, называются *фактическими параметрами***.**

Корректность передачи параметров основывается на их порядке перечисления в заголовке процедуры и совместимости по присваиванию между соответствующими фактическими и формальными параметрами. Задача программиста - убедиться, что параметры, которые он указывает при вызове

(фактические параметры), соответствуют по очередности формальным параметрам. Компилятор может проверить только очевидные случаи - неправильное число параметров или несовместимость типов.

Формальные параметры могут быть *параметрами-переменными* и *параметрами-значениями.*

Заголовок процедуры с описанными *параметрами-значениями*:

```
Procedure  MyProc (Par1, Par2  :type1; Par3, Par4 : type2);
```

Если формальный параметр есть параметр-значение, соответствующий фактический параметр может быть выражением.

Если формальный параметр есть результат работы процедуры, то такой параметр называется *параметром-переменной.* При описании заголовков процедур перед идентификаторами параметров-переменных *ставится ключевое слово* **Var.** Заголовок процедуры с объявленными параметрами-переменными Par3,Par4 и параметрами-значениями Par1,Par2 :

```
Procedure    MyProc ( Par1, Par2  :type1;   Var    Par3, Par4 : Type2)
```

В качестве фактического параметра-переменной могут использоваться переменные любых типов, включая файловые, зато использование констант не допускается. Использование для объявления параметров-переменных исключает возможность вызова процедуры или функции с фактическими параметрами в виде выражений, что делает программу менее компактной. Не рекомендуется использовать параметры-переменные в заголовке функции.

Задача 6. Вернемся к задаче о вычислении матрицы налогов и других матриц для формирования ведомости к выдаче зарплат для работников малого предприятия (***Типовая Задача 4*** параграфа 5.2). Программу *Zarplata* запишем в другой форме , используя две процедуры : **vvod_matric** и **vich_matric**, которые записываются в подразделе текстов процедур в разделе объявлений программы. Тогда запись основного блока программы сократится и упростится за счет вызова этих процедур.

Первую процедуру **vvod_matric**, предназначенную для ввода матрицы зарплат, сделаем универсальной для ввода матриц в других программах. Поэтому оформим ее в локальных (формальных) переменных , объявляемых в самой процедуре:

x[i,j] - индексированная переменная матрицы зарплат **x**, где

i – номер отдела предприятия (**i**=1,2,…m1);

j – табельный номер в любом отделе (**j** =1,2 …n1);

Здесь константы m1 и n1 относятся к размерам предприятия по административной структуре.

Вторую процедуру **vich_matric** используем для сокращения записи основного блока программы и не будем делать ее универсальной. Поэтому для

ее записи воспользуемся глобальными переменными, определяемыми в разделе **Var** основной программы.

```
Program Zarplat_Procedur;
Const   m =  4; n  = 10;  mz = 83.49;
Type   t1 = array[1..m,1..n] of  real;
Var    zp, zpv, pn: t1;  i,j: integer; sum: real;
Procedure vvod_matric(m1, n1: integer; Var x: t1);
Begin
writeln ('Начнем ввод зарплат');
for i := 1 to m do            { блок ввода зарплат }
   begin
   writeln ('Отдел  ' , i );
   for j := 1 to n do
     begin
     write ('Для ',j:2,'-го работника  зарплата=');
     readln (x[i,j]);
     end;
   end;
End;
Procedure    vich_matric; {В процедуре используются только глобальные}
Begin                                                          {переменные}

Sum := 0;
for i := 1 to m do            { блок вычислений  }
  for j := 1 to n do
begin
if zp [i, j] <= 3 * mz
 then
          pn [i , j] := 0
 else
           if zp [i, j] <= 10 * mz
      then
              pn[i, j]:=0.12*(zp [i, j]-3*mz)
                   else
               pn[i,j]:=0.2*(zp[i,j]-10*mz)+0.84*mz ;
            zpv [i, j] := zp [i, j] - pn [i, j] ;        { Вычитание матриц}
            sum := sum + zpv [i, j] ;                    { Суммирование массива}
            end ;
End ;
BEGIN                                        {Начало основного блока     }
vvod_matric   ( m,n, zp  );                              { Вызов 1-й процедуры }
```

```
vich_matric;                                    { вызов 2-й процедуры }
writeln (‘ВЕДОМОСТЬ ‘ : 40 ); writeln ;     { Блок вывода ведомости }
writeln (‘ Отдел ’,’ Номер ’,’ Зарплата’ : 10,’ Налог’ : 10, ‘ Зарплата к      выдаче’
:16 , ‘ Подпись’:10) ; writeln ;
for i:=1 to m do
  for j:=1 to n do
        writeln(i:5, j:6, zp[i,j]:10, pn[i,j]:10, zpv[i,j]:16, ’          ‘:10) ;
writeln (‘ ИТОГО К ВЫДАЧЕ : ‘, Sum : 14) ;
END.
```

7. СТРОКИ СИМВОЛОВ

Для обработки текстов широко используются данные типа **строка** (**string**). Этот тип данных можно рассматривать как обобщение понятия ОДНОМЕРНЫЙ МАССИВ СИМВОЛОВ (array[1..n] of char), где *n* - максимальная длина (количество символов) строки. В случае объявления в программе:

VAR st1 : **string** ; st2 : **array**[1..13] **of char** ;

в переменной *st*1 можно запомнить текстовую строку любой допустимой длины до 255 символов; а в переменной *st*2 - только до 13 символов. Для того, чтобы экономно использовать память ПК можно сократить размер обрабатываемой строки, например, до 25 символов путем объявления переменной st3 в виде:

Var st3: **string**[25] ;

Над длиной строки можно осуществлять необходимые действия и таким способом изменять ее длину. Размер текущей строки хранится в самом первом байте строки, имеющем индекс 0. Первый значащий символ строки занимает второй байт и имеет индекс 1. Доступ к каждому элементу строки, например st3, можно осуществить указанием индекса (номера) символа строки. Например,

```
Program      Primer_strok_simvolov ;
Var   Name : string[12] ;
BEGIN
Name := ‘ МАША ‘ ;
Name[2] := ‘ И ‘ ;
writeln (Name); { напечается на экране МИША }
Name := Name + ‘ НОВИКОВ‘ ;
writeln (Name) ;{ напечатается МИША НОВИКОВ}
writeln (ord (Name[0])) ;    {будет напечат размер строки, т.е 12 }
writeln (length (Name)) ;    { напечатается также 12 }
END .
```

Заметим, что в этой программе используются операции:

- присваивание строке *Name* начального значения “МАША”;
- присваивание *второму элементу строки Name* символа “И” вместо “А”;

- ***конкатенация*** (слияние, используя знак операции +) нескольких строк символов в одну строку с увеличением ее размера до 12;
- вызов стандартной функции **ord** (Name[0]) или **length** (Name) для получения длины строки.

Для обработки всех символов строки поочередно используют переменный индекс, например **i**, изменяя его от 1 до N. Например, для распечатки строки символов из 26 прописных букв латинского алфавита можно поступить следующим образом : сформировать строку ST, содержащую необходимые символы, а затем распечать.

```
Program      Alfavit;
Var st : string[26];
         i : integer;
BEGIN
for i:=1 to 26 do
   st:=st+chr(ord('A')+i-1) ;
writeln(st) ;  {Распечатается : ABCD…Z}
END .
```

Для обработки строк можно использовать следующие основные стандартные

Процедуры и функции:

concat (S1,S2,…SN) - формирует строку путем сцепления строк S1,S2,..SN ;
copy (ST,I,N) - функция копирует из строки ST цепочку символов из N символов, начиная с символа с номером I ;
delete (ST,I,N) - удаляет N символов из строки ST, начиная с символа с номером I ;
insert (SUBST,ST,I) - процедура вставляет подстроку SUBST в строку ST, начиная с символа с номером I ;
length (ST) - функция типа integer. Возвращает длину строки ST.
upcase (CH) - функция типа char. Возвращает для символьного выражения CH вместо строчных латинских букв соответствующие прописные буквы. Если значением CH является любой другой символ, функция возвращает его без изменений.

В качестве примера использования массива строк символов рассмотрим массив NAME фамилий работников малого предпрятия для распечатки ведомости к выдаче зарплат (см. п.6.3.). В разделе объявления переменных программы *Program Zarplat_Procedur* необходимо довавить:

```
Var    Name : array [1..m,1..n] of string [25] ;
```

В основном блоке для ввода фамилий предусмотрим :

```
writeln (‘Введите фамилии’) ;
for i := 1 to m
    begin
    writeln(‘отдел’, i) ;
    for j := 1 to n do
      begin
      write(‘работник’, j, ‘фамилия ? ’ ) ;
      readln (Name[i,j]) ;
      end ;
    end ;
```

Для вывода ведомости с учетом фамилии в каждой строке необходимо предусмотреть в заголовке ведомости раздел Ф.И.О. и в соответствующем списке вывода оператора **writeln** в теле цикла по i и по j предусмотреть индексированную переменную NAME[i,j].

Упражнение. Составить новую программу для распечатки ведомости с учетом фамилий работников.

8. ЗАПИСИ

Для хранения информации об экономическом или промышленном субъекте приходится использовать структурированные данные различных типов. Совокупность разных, в том числе разнотипных данных об объекте или субъекте, удобно описывать с помощью типа данных, называемого **записью (Record).** Элементы, из которых составляется запись, называются **полями записи.** Например, каждая запись из записной книжки может содержать следующие поля : порядковый номер, фамилию, имя, адрес, телефон. Переменная *zap_knigi* может быть обьявлена как

```
Var  zap_knigi = Record
       Nomer : integer ;
       Familia: string[20] ;
       Imja : string[20
       Adress : string [40] ;
       Telefon : integer ;
        End ;    {Конец записи}
```

Записи такого рода можно рассматривать как структуру данных на карточке в каталоге или строку в книге (журнале) регистрации товаров. При обращении к какому-либо полю записи для обработки (для присваивания, ввода, вывода и др.) необходимо указывать имя записи и имя поля **через точку**, например :

```
zap_knigi.Familia := ‘Петрухин’ ;
```

```
zap_knigi.Imja     := ‘Николай’ ;
zap_knigi.Telefon :=  366397 ;
```

В том случае, когда полей в записи содержится много и все их нужно обработать (например ввести данные для каждого поля записи), имя записи можно объявить только один раз для всего блока обработки полей записи, что сократит текст программы. Чтобы упростить доступ к полям записи, используется оператор **with… do**, благодаря которому отпадает необходимость повторно писать несколько раз имя записи:

```
With  {имя записи} do
                Begin
 {оператор 1 с указанием только имени 1-го поля записи}
 {оператор 2 для второго поля}
        {оператор n для n-го поля }
 End ;
```

Например, ввод одной записи в записную книжку может выглядеть следующим образом :

```
With zap_knigi do { Блок ввода записи}
 Begin
 write(‘Номер:’)     ; readln (Nomer) ;
 write(‘Фамилия:’) ; readln (Familia) ;
 write(‘Имя:’)         ; readln (Imja) ;
 write(‘Адрес:’)      ; readln (Adress) ;
 write(‘Телефон:’)  ; readln (Telefon) ;
End ;   .
```

9. ФАЙЛЫ

9. 1. Типизированные файлы

Переменная **файл данных** понимается либо как именованная область внешней памяти ПК (жесткого диска, гибкой дискеты, электронного виртуального диска), либо логическое устройство - потенциальный приемник или источник информации.

Любой **файл** имеет три характерные особенности. Во-первых, у него есть имя, что дает программе возможность работать одновременно с несколькими файлами. Во-вторых, он содержит компоненты одного типа. Типом компонентов может быть любой тип **Turbo Pascal**, кроме файлов. *Имя файла* - это любое выражение строкового типа, которое строится по правилам определения в операционной системе ПК. Перед именем может ставиться так называемый путь к файлу: имя диска и/или имя текущего каталога и имена каталогов вышестоящих уровней: например A:\ CATALOG\ PODKATALOG\ test.dat.

В программе на языке **TurboPascal** с файлами связана *файловая переменная*, которая должна быть объявлена наряду с другими переменными в разделе **Var**.

Типизированный файл может быть объявлен следующим образом:

Var{ *имя_переменной_файла* } : **File Of** { *Тип_данных_элемента* }

Идентификатор { имя_переменной_файла } переменной типа **File** выбирается произвольным образом, например *f1*. *Тип данных* элемента переменной типа **File** может быть любым из допустимых в **TurboPascal 7.0** (простым или структурированным). На практике в качестве составных элементов типизированных файлов часто выступают записи.

Любой файл становится доступным программе только после особой процедуры открытия файла. Эта процедура заключается в *связывании* объявленной ранее файловой переменной с именем существующего или вновь создаваемого файла. Файловая переменная связывается с именем файла в результате обращения к стандартной процедуре **assign** (*назначить*):

Assign (*имя_переменной_файла в программе, имя файла в MS DOS*);

Для обработки файловых переменных в программе используются другие стандартные процедуры и функции.

Процедура **Reset** открывает для чтения уже существующий файл, имя которого было связано перед этим с файловой переменной процедурой **Assign** . Например, при выполнении операторов

```
Assign (f1,'test.dat') ;
Reset (f1) ;
```

уже существующий файл *test.dat* подготавливается к чтению информации .

Процедура **Rewrite** инициирует запись информации в файл, имя которого уже было связано перед этим с файловой переменной процедурой **Assign** . Например, пара операторов

```
Assign (f1,'test.dat') ;
Rewrite (f1) ;
```

– открывает существующий файл *test.dat*, чистит его и подготавливает к приему информации .

Процедура **Close** закрывает открытый ранее файл, связанный с указываемой в качестве параметра переменной

```
Close (f1) ;
```

Во избежание ошибок все открытые файлы в конце программы должны быть закрыты.

Функция **EOF** (**E**nd **O**f **F**ile - конец файла). Логическая функция, тестирующая конец файла. С помощью ее можно проверить достигнут ли конец файла при его прочтении.Например, цикл

```
While Not EOF (f1) Do
   Read (f1, zap_rnigi)
```

будет считывать записи из файла, связанного с файловой переменной *f1*, хранящего данные записной книжки до тех пор, пока *файловый указатель* **не** (**Not**) достигнет конца файла. Другие процедуры менее часто используются для обработки файлов. О них можно прочитать в [1].

В качестве примера программы, использующей указанные процедуры, рассмотрим задачу заполнения файла ‘ A:\ kn.dat’ , открытого на дискете A:, записями типа *Person* для запоминания информации из записной книжки (см.п.8), контрольную распечатку файла и перезапись его в другой файл.

```
Program   Create_file ;
Uses Crt ; {Использование модуля Crt для работы с экраном}
Type
  Person = Record
  Nomer : integer;
  Familia: string[20];
  Imja : string[20];
  Adress : string [40];
  Telefon : integer ;
              End ;
Var
  zap_knigi : Person ;
  namef : string[20];
  f1, f2 : File Of Person ; { файловые  f2 ,f1 - из записей}
Procedure   Vvod_zapisi ;      { Процедура  ввода записи    }
              Begin
              With zap_knigi Do
   Begin
   Write(‘Номер:’); readln (nomer);
   Write(‘Фамилия:’); readln (familia);
   Write(‘Имя:’); readln (imja);
   Write(‘Адрес:’); readln (adress);
   Write(‘Телефон:’); readln (telefon);
                End
End ;
Function Cycle : Boolean ;{Функция  определяет: вводить ли запись}
Var  simbol: char;
```

```
Begin
Writeln ;
Write('Будете вводить еще записи  ? Ответ: Y – Да,  N – Нет') ;
InKey(symbol) ;
If UpCase(symbol)='Y'
  then
    Cycle := true
  else
    Cycle :=false ;
End ; { конец  функции Cycle}
BEGIN                                    {основной блок программы}
Clrscr ; {очистка экрана}
Write('Под каким именем сохранить файл на дискете?('A:\zkn.dat?)') ;
Readln (namef) ; { Ввод с клавиатуры имени файла}
Assign (f1, namef) ;
Rewrite (f1) ;
While Cycle Do              {Ввод записей в файл с переменной f1}
  Begin
                FillChar (Zapis_knigi, SizeOf(zapis_knigi),' ');{очистка памяти под
               запись}
  Writeln ;
                Vvod_zapisi ;{ввод записи с клавиатуры}
                With Zapis_knigi Do
     Begin
       if  Length (familia) > 0 {если указана фамилия}
         then Write (f1,zapis_knigi) ;{то данные записать из памяти в файл f1}
      End
   End ;
Writeln(Lst) ;
Writeln (Lst, 'Контрольная печать записной книжки из файла') ;
Writeln (Lst) ;
Reset (f1) ;
While Not EOF(f1) Do{пока не будет прочитан конец файла f1}
  Begin
  Read (f1,zapis_knigi) ;{читать очередную запись}
                With zapis_knigi do    {и распечатывать ее}
     begin
     Writeln(Lst,'Номер:',nomer,'Фамимлия:',familia,'Имя:',imja,
                  'Адрес:',adress,'Телефон:',telefon) ;
      end ;
   End ;
Assign (f2,'new.dat') ;{назначить новый файл для  перезаписи}
```

```
Rewrite (f2) ;        {начальные установки}
Reset (f1) ;          {для файлов f1 и f2     }
While Not  EOF(f1) Do          { Перезапись}
   Begin
   Read(f1,zapis_knigi) ;{ чтение текущей записи в память из файла f1}
   Write(f2,zapis_knigi) ;{ запись текущей записи из памяти в файл f2}
   End;
Close(f1) ;                    {закрытие файла f1}
Close(f2) ;                    { закрытие файла f2}
END.          {конец программы}
```

Заметим, что для вывода на принтер в процедуре **Write** использовано стандартное имя файла-листинга: **Lst** , не требующее объявления в программе. Перед чтением файла **f1** для его контрольной печати или для перезаписи в файл **f2** необходимо переустановить **указатель** элементов (записей) файла f1 на начальное значение 0.

Операции обработки файлов включают:

– ввод\вывод записей данных;

– сортировку записей в порядке алфавита по какому-нибудь полю записей файла;

– выборка записей на экран по запросу(ключу), совпадающему ,например,со вторым полем записи типа *Person* , т.е. по фамилии и др.

9.2. Текстовые файлы

Стандартные процедуры и функции применимы для обработки не только типизированных файлов, но и файлов, элементами которых являются строки символов, т.е. текстовых файлов. Текстовые файлы связываются с файловыми переменными, принадлежащими стандартному типу **text:**

```
Var    {имя файловой переменной} : text ;
```

Для примера рассмотрим программу вычисления площади треугольника по трем сторонам *a,b,c* по формуле Герона с записью результатов в файл, который затем будет присоединен при распечатке к файлу, содержащему текст отчета.

```
Program Geron ;
Var a,b,c,S,p : string ;
      f1 : text ;
      namef : string[20] ;
BEGIN
write ('Введите размеры треугольника a,b,c:') ;
readln (a,b,c) ;
p := (a+b+c)/2 ;
```

```
S := sqrt (p*(p-a)*(p-b)*(p-c)) ;
write('Введите имя файла результатов, например  lab1.dat') ;
readln(namef) ;
assign (fl, namef) ;
rewrite(fl) ;
writeln(fl,'Исходные данные:','a=',a:6:2,' b=',b:6:2,' c=',c:6:2) ;
writeln (fl,' Результат : площадь треугольника равна',S:10:2) ;
writeln(fl,'Выполнил студент гр. 2013-ЭПА  Билюба Дима') ;
close(f1) ;
END.
```

Кроме процедур **Reset** и **Rewrite** к текстовым файлам может быть применена еще одна стандартная процедура **Append**. Эта процедура инициирует запись строк символов в ранее существовавший текстовый файл для его расширения, перед записью указатель файла устанавливается на его конец.

10. ЛИТЕРАТУРА

1. Турбо Паскаль 7.0-К.: Торгово-издательское бюро BHV, 1996. 448 с: ил.
2. Епанишников А. М. Епанишников В. А. Программирование в среде TURBO PASCAL 7.0.-3-е изд. -М. : ” ДИАЛОГ-МИФИ “ , 1996. 288 с .
3. Шаньгин В. Ф.,Поддубная А. М. Программирование на языке ПАСКАЛЬ. -М. : Высш. шк., 1991.
4. Пильщиков В. Н. Сборник упражнений по языку Паскаль: Учеб. пособие для вузов. -М.: Наука, 1989. 160 с.
5. Абрамов С. А. , Зима Е. В. Начала программирования на языке Паскаль.-М.: Наука,1987. 112 с.
6. Borland Pascal with Objects, Vertions 7.0. Language Guide. Borland International INC,1992. 371 p.
7. Фигурнов В. Э. IBM PC для пользователя. Краткий курс. 7-е изд. М. : ИНФРА, 1997. 478 с.
8. Левин А. Самоучитель работы на компьютере . 3-е изд. М.: 1997. 367 с.
9. Начальные сведения о персональной ЭВМ типа IBM PC: Метод. указания./ НГТУ; Сост.: В.Ф. Билюба, И. В Зороастрова и др. Н. Новгород, 1996. 31 с.
10. Компьютерно-програмное моделирование процессов решения вычислительных и функциональных задач: Метод. разработка /НГТУ; Сост.: В. Ф. Билюба, С. Н. Митяков и др. Н. Новгород. 1997. 41 с.
11. Фаронов В. В. Турбо Паскаль 7. 0. Начальный курс: Учеб. пособие.-М.: «Нолидж», 1998. 616 с.: ил.
12. Фаронов В. В. Турбо Паскаль 7. 0. Расширение возможностей. М.: «Нолидж», 1998. 616 с.: ил.
13. Основы алгоритмизации и программирования на языке Турбо Паскаль: Метод. разработка по курсу “Информатика” /НГТУ; Сост.: В.Ф. Билюба, Е. А. Маслова и др. Н. Новгород, 1998. 43 с.
14. Элементы численных методов и основы технологии работы в Windows-приложении MathCAD: Метод. разработка по курсу “Информатика” /НГТУ; Сост.: В.Ф.Билюба,Т.В. Моругина, и др. Н. Новгород, 2001.