МИНИСТЕРСТВО ОБЩЕГО И ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
имени академика С.П. КОРОЛЕВА

Л.С.Зеленко

Т.И.Михеева

Алгоритмические языки и программирование

курс лекций для студентов заочной формы обучения

(1 часть)

Самара 1999

УДК 681. 3

Алгоритмические языки и программирование. 1 часть: Курс лекций / *Л. С. Зеленко, Т. И. Михеева.* Самарский гос. аэрокосмический ун-т. Самара, 1999, 72 с.

ISBN 5-7217-0030-0

Предназначен для студентов заочной формы обучения по специальности 22.02 - "Автоматизированные системы обработки информации и управления" - и содержит описание основных возможностей процедурного языка высокого уровня Turbo Pascal (версия 7.0) для персональных ЭВМ, работающих под управлением операционной системы MS DOS. Излагаются основные приемы программирования, объясняются особенности создания программ, приведено большое количество примеров.

Данный курс лекций будет также полезен и студентам других специальностей, знакомых с основами программирования на алгоритмических языках и обучающихся как по очной, так и заочной форме обучения. Разработан на кафедре информационных систем и технологий.

Печатается по решению радакционно-издательского совета Самарского государственного аэрокосмического университета имени академика С. П. Королева

Рецензент канд. техн. наук, доц. Л. А. Жаринова

ISBN 5-7217-0030-0

©Самарский государственный аэрокосмический университет, 1999

Оглавление

Введение......4

1. Построение программ......7

1.1. Алфавит языка и зарезервированные слова......7
1.2. Правила построения идентификаторов......8
1.3. Общая структура программы......10
1.4. Типы данных......14
1.4.1. Простые типы языка......16
Целочисленные типы......16
Вещественные типы......17
Логический тип......18
Символьный тип......19
1.4.2. Совместимость типов......19
1.4.3. Явное преобразование (приведение) типов......22
1.5 Базовые операции......23
1.5.1. Логические вычисления и операции отношения......24
1.5.2. Математические процедуры и функции......26

2. Управляющие структуры языка......29

2.1. Простые и составные операторы......29
2.2. Условный оператор......30
2.3. Оператор выбора (варианта)......33
2.4. Операторы цикла......35
2.4.1. Итеративный (цикл с шагом)......35
2.4.2. Цикл с предусловием......37
2.4.3. Цикл с постусловием......38

3. Основные структурные типы языка......41

3.1. Массивы......41
3.1.1. Описание массивов......41
3.1.2. Ввод и вывод массивов......46
3.1.3. Форматы вывода......47
3.1.4. Базовые алгоритмы обработки векторов......48
3.1.5. Алгоритмы поиска......49
3.1.6. Базовые алгоритмы обработки матриц......50
3.1.7. Алгоритмы сортировки массивов......53
3.1.8. Совместные алгоритмы обработки матриц и векторов......54
3.2. Множества......55
3.2.1. Описание множеств......55
3.2.2. Операции, применимые к множествам......57
3.2.3. Хранение множеств в памяти компьютера......60

4. Обработка символов и строк......61

4.1. Символьный и строковый типы (Char и String)......61
4.2. Операции над символами......65
4.3. Операции над строками......66
4.4. Процедуры и функции обработки строк......67
4.5. Базовые алгоритмы обработки строк......68

Список рекомендуемой литературы......70

Введение

Система программирования Turbo Pascal (версия 7.0) в состоянии удовлетворить требования, предъявляемые разработчикам прикладного программного обеспечения при работе на персональных ЭВМ в операционной системе MS DOS. Язык Turbo Pascal является структурированным языком высокого уровня, на котором можно написать программу практически неограниченного размера и любого назначения.

Описываемая версия языка представляет полную среду для профессионального программирования, обладающую высокими характеристиками:

Наличие системных библиотечных модулей, являющихся органической составляющей языка (модули System, Crt, Dos, Graph и др.).

Наличие инструментов объектно-ориентированного программирования.

Наличие встроенных процедур и функций.

Наличие встроенного в интегрированную среду программирования компилятора, обладающего высокой скоростью и позволяющего генерировать оптимизированный код, обеспечивающий быстрое выполнение программ.

Возможность создания отдельно компилируемых блоков (модулей), возможность условной компиляции программ.

Поддержка математических сопроцессоров.

Наличие расширенного набора числовых типов с плавающей запятой стандарта IEEE (Institute of Electrical and Electronics Engineers в США) – с одинарной, с двойной , с повышенной точностью – в случае использования сопроцессора.

Эффективный интерфейс с языками Turbo Assembler и Turbo C на уровне объектного кода.

Наличие интегрированного в среду программирования отладчика, позволяющего выполнить полную проверку переменных, структур данных и выражений по шагам или в заданных точках.

Использование оверлеев на основе программных модулей, развитая подсистема управления оверлеями и т.д.

Для понимания общей идеологии построения системы сначала рассмотрим функции ее составных частей.

Пакет программирования Turbo Pascal (в дальнейшем будем называть его "оболочкой") является интерактивной интегрированной средой, сочетающей в себе возможности редактора текстов, компилятора и отладчика. Он поддерживает систему меню, оконный интерфейс, контекстную справочную систему, обеспечивает возможность конфигурирования системы под конкретного пользователя. Сама оболочка активизируется запуском файла Turbo.exe.

Встроенный отладчик позволяет легко выполнять программы по шагам, проверяя или модифицируя при этом переменные или ячейки памяти, устанавливая точки останова и прерывая выполнение программы с помощью специальной комбинации клавиш (двойное нажатие CTRL+Break).

Развитием принципов структурного программирования является введение понятия модуля – части исходного текста программы, которую можно откомпилировать как самостоятельное целое и которая находится в другом файле. Набор модулей можно рассматривать как библиотеку программ, выполняющих стандартные для данного пользователя действия и обеспечивающих интерфейс (сопряжение) по данным и (или) управлению. В Turbo Pascal'е имеются стандартные библиотеки, которые оформлены в виде таких модулей:

System – стандартные и встроенные процедуры языка;

Dos – работа с функциями операционной системы MS DOS;

Crt – работа с клавиатурой и видеомонитором;

Graph – графические процедуры;

Win – быстрая и удобная работа с окнами.

Все эти модули будут более подробно рассмотрены в соответствующих разделах курса.

1. Построение программ

В данном разделе рассматриваются алфавит и ключевые слова языка, правила построения идентификаторов, особенности построения программ на языке Turbo Pascal.

1.1. Алфавит языка и зарезервированные слова

Как и любой язык программирования, Turbo Pascal имеет свой алфавит – набор символов, разрешенных к использованию и воспринимаемых компилятором. В алфавит языка входят:

1. Латинские строчные и прописные буквы	A, B, C . . .Z, a, b ,c, . . . z
2. Цифры	0, 1, …9
3. Символ подчеркивания	“_”
4. Символ «пробел»	Является разделителем (код 32)
5. Символы с кодами ASCII от 0 до 31 (управляющие коды)	Непечатные
6. Специальные символы, участвующие в построении конструкций языка	+ – * / = < > [ ] . , ( ) ; : ^ @ { } $ # ‘
7. Составные символы, рассматриваемые как единое целое (пробелы между ними не допустимы)	<= >= := (* *) (. .) ..

Turbo Pascal имеет большое количество зарезервированных слов, которые не могут быть использованы в качестве идентификаторов, поэтому попытка нарушить этот запрет либо вызовет ошибку при обработке программы компилятором языка, либо семантическую ошибку (при выборе идентификатора, совпадающего с именем стандартных процедур и функций из

библиотечных модулей). Зарезервированные слова выделяются редактором текста белым цветом, после зарезервированного слова символ ";" не ставится (кроме слова End).

Таблица 1. Список зарезервированных слов

ABSOLUTE	EXTERNAL	MOD	SHR
AND	FILE	NIL	STRING
ARRAY	FOR	NOT	THEN
BEGIN	FORWARD	OBJECT	TO
CASE	FUNCTION	OF	TYPE
CONST	GOTO	OR	UNIT
CONSTRUCTOR	IF	PACKED	UNTIL
DESTRUCTOR	IMPLEMENTATION	PROCEDURE	USES
DIV	IN	PROGRAM	VAR
DO	INLINE	RECORD	VIRTUAL
DOWNTO	INTERFACE	REPEAT	WHILE
ELSE	INTERRUPT	SET	WITH
END	LABEL	SHL	XOR

Замечание: Зарезервированное слово PACKED (упакованный) в Turbo Pascal'е игнорируется.

1.2. Правила построения идентификаторов

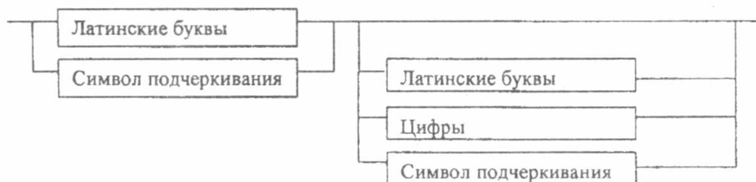
Идентификаторы (ИД) – все имена, которые пользователь использует в программе для обозначения каких-либо понятий. ИД – это имена типов, переменных, констант, процедур, функций, модулей, меток. Для правильного построения ИД необходимо соблюдать следующие правила:

Таблица 2. Правила построения идентификаторов

№ п/п	Правило	Пример написания	
		Правильно	Неправильно
1.	ИД может состоять только из букв латинского алфавита, цифр и специального символа подчеркивания, причем первый символ не может быть цифрой	X Y1 i5 CharVar Task125 My_Int_Type	156_type 666Num
2.	ИД может состоять из произвольного числа символов, но компилятором различаются только первые 63 (остальные игнорируются)		
3.	Все ИД должны быть уникальными		
3.1.	ИД не может совпадать по написанию с зарезервированными словами, в случае необходимости к зарезервированному слову можно добавить символ подчеркивания	End_ _End _End_	End Procedure
3.2.	ИД не должен совпадать по написанию со стандартными операторами и встроенными (библиотечными) процедурами и функциями	Write_ _Write _Window	Write Window
3.3.	Прописные и строчные буквы не различаются	WRITE Write	
4.	ИД должен нести смысловую нагрузку, т.е. при его объявлении необходимо определять дальнейшее использование в программе	OpenWindow Algorithm	
5.	При записи ИД необходимо выбирать английский вариант понятия	Window Count	OKNO Chetchik
6.	При записи составных ИД (ИД, состоящих из нескольких понятий) желательно делать первую букву каждого слова прописной	CheckInFile FileExist	checkinfile fileexist

Правила 1, 2, 3.1. строго контролируются компилятором, поэтому при их нарушении компилятором будет выдано сообщение об ошибке и трансляция программы будет прервана. Остальные правила компилятор не контролирует, но соблюдать их желательно, так как они позволяют улучшить «читабельность» программы и избежать семантических (смысловых) ошибок (правило 3.2.).

Основные правила можно представить следующей диаграммой:



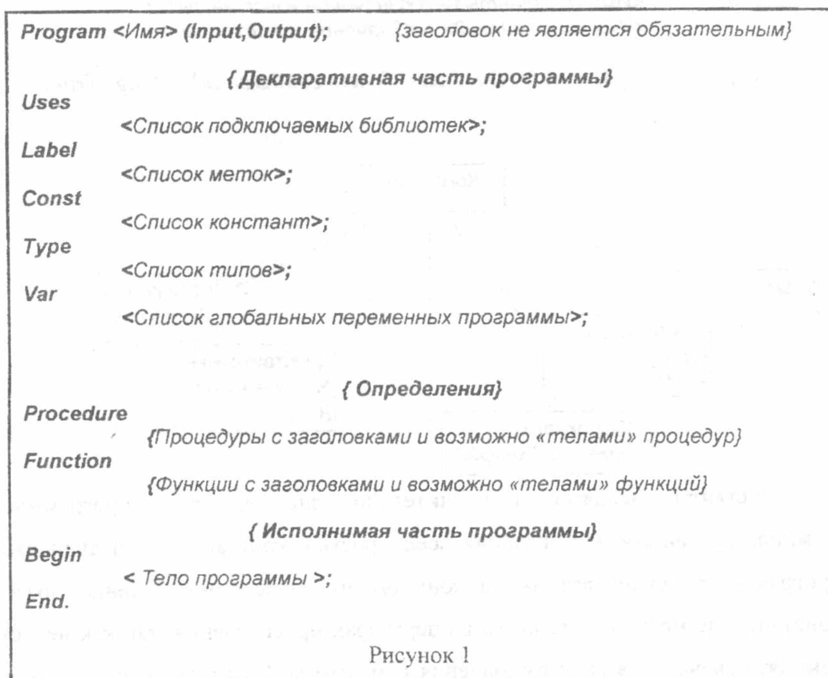
1.3. Общая структура программы

Программы на языке Turbo Pascal состоят из нескольких смысловых блоков, порядок размещения которых в тексте программы может быть достаточно жестким. Написанная по правилам стандарта языка программа будет иметь в своем полном варианте следующую структуру, представленную на рисунке 1.

Директива *Uses* – первый действительно работающий в программе оператор. С ее помощью подключаются библиотечные модули (стандартные и написанные пользователем), расширяя тем самым список процедур, функций, переменных и констант, используемых в программе. У директивы *Uses* есть свое четкое место: она должна появляться в программе перед другими прочими директивами и только один раз. Список библиотек дается через запятую:

Uses Dos, Ctr, My_modul;

Если модули в программе не используются, то директива *Uses* не используется.



```
Program <Имя> (Input,Output);        {заголовок не является обязательным}

                    {Декларативная часть программы}
Uses
        <Список подключаемых библиотек>;
Label
        <Список меток>;
Const
        <Список констант>;
Type
        <Список типов>;
Var
        <Список глобальных переменных программы>;


                    { Определения}
Procedure
          {Процедуры с заголовками и возможно «телами» процедур}
Function
          {Функции с заголовками и возможно «телами» функций}
                    { Исполнимая часть программы}
Begin
        < Тело программы >;
End.
```

Рисунок 1

Блок описания меток ***Label*** содержит перечисленные через запятую метки перехода, которые в программе обозначаются целым числом в диапазоне ***0..9999*** или ИД

Label ***Loop, 1, m1, ProgramExit;***

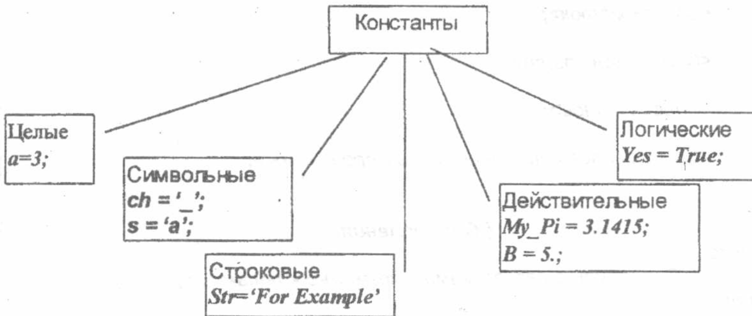
Если в программе метки не используются, то данный блок в программе отсутствует.

Замечание: Практически любая программа может быть написана без использования меток, что определяет уровень мастерства программиста. Студентам рекомендуется не использовать данный блок.

Блок описания ***Const*** может располагаться в любом месте программы, таких блоков может быть несколько или не быть вообще (это замечание относится также к блокам ***Type, Var, Label***).

Const *<Имя константы1> = <Значение константы1>;*
<Имя константы2> = <Значение константы2>;
...

Const ***N_Max=15; OutText='Текст сообщения'; Flag=True;***



Константы вводятся исключительно для удобства программиста. Компилятор запоминает значения всех простых констант и при трансляции программы заменяет все имена констант их значениями. Именно поэтому константы не могут стоять слева в операторах присваивания, так как не могут изменять значения в ходе выполнения программы. Имя и значение константы разделяются знаком равенства «=» (а не оператором присваивания «:=»). После задания значения константы обязательно должна стоять «;». Тип значения не указывается никоим образом, он определяется автоматически при анализе значения константы. В Turbo Pascal'е есть возможность определять значения константы, используя выражения из чисел некоторых функций языка и определенных ранее простых констант. В выражениях допускается использовать все математические операции ***(+, -, /, *, Div, Mod***), поразрядные (битовые) операции, все логические операторы (о них мы будем говорить ниже), операции отношения и т.д. Из стандартных математических функций в выражениях констант могут использоваться только следующие:

Abs(x), Round(x), Trunc(x), Chr(x), Ord(x), Pred(x), Succ(x), Odd(x), SizeOf(x), Length(x), Prt(S,O), Lo(x), Hi(x).

Все выражения вычисляются при компиляции, а затем лишь подставляются всюду вместо идентификаторов констант. Turbo Pascal вводит особый тип констант – типизированных – и позволяет работать с константами сложных типов (кроме файлов). Но такие константы , по существу, являются переменными со стартовыми значениями.

```
Const   Interval : Integer = 158;
        HelpString : String[80] ='Для продолжения работы нажмите'+
        'любую клавишу';
        No : Boolean = false;
        Ok : Char ='Y';
        R : Real = 133.678;
        Rus_Letter: Set of Char = ['А' .. 'р' , 'п' .. 'я'];
```

Раздел описания глобальных переменных содержит список глобальных переменных и их типы, блоков может быть несколько, но переменные в них не должны повторяться.

```
Var         <Имя Переменной1>:<Тип Переменной1>;
            <Имя Переменной2>:<Тип Переменной2>;
            <Имя Переменной3>,<Имя Перемееной4>:<Тип Переменной34>;
            ...
Var    a: Real;
       i,k: byte;
       ch,s,simb: char;
```

Все глобальные переменные хранят свои значения в так называемом сегменте данных. Его максимальный объем теоретически равен 65520 байт (почти 64К) , но практически он меньше на 1..2К. Значения в сегменте данных всегда последовательны по мере перечисления их имен.

a	i	k	ch	s	Simb
+6	+7	+8	+9	+10	+11

Начало отсчета
в байтах

Если в программе используются процедуры и функции, то их определение должно предшествовать основному блоку.

Существуют ограничения на перемещение блоков в программе: программа компилируется последовательно, поэтому все, что в ней вводится, должно быть объявлено до того, как будет использоваться. Компилятор также накладывает ограничения на длину строки: она не может превышать 126 символов, объем файла (текста) программы – 64Кбайт.

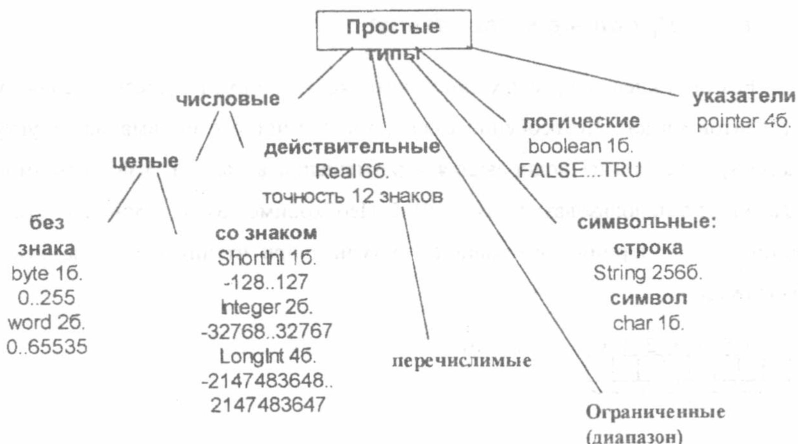
Для того чтобы легче было разобраться в сложных программах, рекомендуется по ходу программы ставить комментарии, в которых указывать на некоторые особенности работы той или иной части программы, расшифровывать назначение переменных и т.д. Количество комментариев в программе должно быть разумным: не нужно комментировать каждый оператор. Признаком начала комментария является наличие фигурных скобок *{ }* или скобок комментариев *(* *)*.

Примеры комментариев.

```
{ Комментарий }
(* Так тоже можно *)
(* И так {тоже} можно *)
{ А (* так } нельзя *)
{ Такое {вложение} не использовать!!! }
```

1.4. Типы данных

Язык Turbo Pascal является языком с сильной системой типизации, поэтому все данные, обрабатываемые программой, должны принадлежать к какому-либо заранее известному типу. В языке предопределено большое количество базовых типов и имеется возможность для объявления типов, необходимых пользователю для конкретных приложений.



Все эти типы могут участвовать в определении сложных типов. Данный список может быть расширен за счет использования математического сопроцессора. Набор сложных типов, определяющих структуры из простых типов, весьма широк:

Массив – ***Array … of …;***

Файлы (три вида) – ***Text, File, File of …;***

Множество – ***Set of …;***

Запись – ***Record ;***

Ссылка - ***^базовый тип;***

Объект – ***Object ;***

Создание нового типа осуществляется при помощи ключевого слова **Type**:

```
Type        <ИмяНовогоТипа1> =      <Имябазовоготипа /
                                     Имя типа, описанного выше>;
```

Пример:

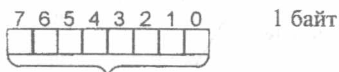
```
Type  Int = Integer;
Var   a: Int; { Integer  - так можно, но не надо; }
```

Так записывать не надо:

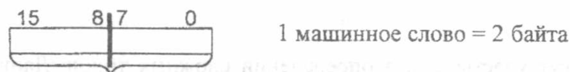
```
Type  Integer = Real;      {результат не определен}
      Real = Integer;
Type  OldType = NewType; {необходимо поменять строки местами}
      NewType = Integer;
```

1.4.1. Простые типы языка

Без обсуждения простых типов невозможен подход к рассмотрению всех элементов языка. Это особенно важно, т.к. без четкого понимания структуры данных, способов их кодирования и размещения в памяти ЭВМ, невозможно эффективно использовать память ЭВМ. Необходимо также помнить о том, что минимальная порция информации, доступная при чтении/записи памяти – 1 байт (8 бит).



1 байт

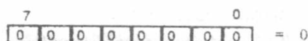


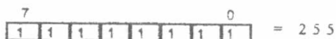
1 машинное слово = 2 байта

Целочисленные типы

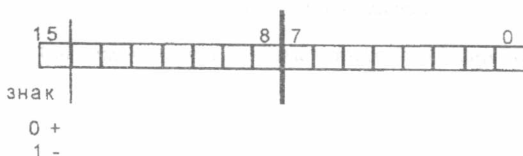
Обилие целочисленных типов позволяет эффективно использовать память и более гибко вводить целочисленные переменные. Рассмотрим на двух примерах принципы кодирования знаковых и беззнаковых типов.

1. ***Byte*** занимает 1 байт памяти, диапазон значений находится в неотрицательной области. Диапазон значений





2. ***Integer*** занимает 2 байта памяти, диапазон значений –32768 до 32767.



15							8	7							0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

знак

0 +
1 -

$-32768 = -2^{15}$

15							8	7							0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

знак

0 +
1 -

$32767 = 2^{14}+2^{13}+2^{12}+2^{11}+2^{10}+2^{9}+$
$2^{8}+2^{7}+2^{6}+2^{5}+2^{4}+2^{3}+2^{2}+2^{1}+2^{0}$

15							8	7							0
1	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1

знак

0 +
1 -

$-32733 = -2^{15}+2^{5}+2^{2}+2^{0}=$
$-32768 + 32 + 2 + 1$

Старший 15-ый бит хранит знак числа, в то же время не теряет своего весового влияния при преобразовании числа к десятичному виду. Биты с0 по 14 кодируют положительную составляющую числа.

Полученные результаты можно записать в шестнадцатеричной системе счисления: -32768 = -$8000; 32767 = $7FFF; -32733 = -$8025. Признаком шестнадцатеричного числа является знак доллара $.

Вещественные типы

Данные типы могут записываться многими способами:

-145.89	0.000564	124.876	.001
6543	-345	0	+65
123E+02	-0.04e+06	6E6	-55E7

Экспоненциальный способ записи (плавающий формат вывода результата) соответствует умножению на заданную степень 10 : -0.004E+06= $-0.004*10^{+06}$

Экспоненциальный способ записи (плавающий формат вывода результата) соответствует умножению на заданную степень 10 : -0.004E+06= - $0.004*10^{+06}$.

Кроме базового типа ***Real*** , который занимает 6 байт памяти, сопроцессор поддерживает еще четыре вещественных типа:

Тип	Диапазон значений	Количество значащих цифр	Размер в байтах
Real	2.9e-39 .. 1.7e+38	11-12	6
Single	1.5E-45 .. 3.4E+38	7-8	4
Double	5.0E-34 .. 1.7E+308	15-16	8
Extended	3.4E-4932 .. 1.1E+4932	19-20	10
Comp	-9.2E+18 .. 9.2E+18	19-20	8

Вещественное число делится на две части: мантиссу (содержит значащие цифры числа) и порядок (экспонента).

Логический тип

Логический тип ***Boolean*** состоит из двух значений: True (истина) и False (ложь). Слова True и False определены в языке и являются, по сути, логическими константами. Регистр в записи не существен: FALSE=False.

Boolean занимает 1 байт памяти, хотя для кодирования двух состояний достаточно было бы 1 бита.

7							0	
0	0	0	0	0	0	0	0	= False

7							0	
0	0	0	0	0	0	0	1	= True

Символьный тип

Символьный тип ***Char*** – это тип данных, состоящий из одного символа (знака, буквы, кода). Он занимает 1 байт памяти и хранит собственно не сам символ, а его кодовое представление (в соответствии со стандартом ASCII). Традиционная запись символьного значения представляет собственно сам символ, заключенный в одиночные кавычки (апострофы). Например, 'A', 'f', '0', 'ж'…

7							0	
0	0	1	1	0	0	0	0	= '0'(код 48)

7							0	
0	1	0	0	0	0	0	1	= 'A' (код 65)

1.4.2. Совместимость типов

Turbo Pascal, являясь языком с сильной системой типов, требует соблюдения правил совместимости типов переменных. Проблем не возникает, если переменные, участвующие в операции присваивания, имеют один или идентичный тип данных. Два типа ***Type1*** и ***Type2*** считаются *идентичными*, если:

– Типы ***Type1*** и ***Type2*** описаны одним и тем же идентификатором простого типа:

```
Type Type1=Real;
     Type2=Real;
```

Для составных типов пользователя это правило не выполняется.

– Типы ***Type1, Type2*** и ***Type3*** описаны как эквивалентные

```
Type Type1=Real;
     Type2=Type1;
     Type3=Type2;
```

Значения типов ***Type1, Type2*** и ***Type3*** будут полностью совместимыми, точно так же, как и переменные ***x1, x2,..x5***, описанные данным типом:

```
Var x1, x2, x3 : Type1; x4: Type2; x5 : Type3;
```

Совместимость типов трактуется следующим образом. Типы считаются совместимыми, если:

- Оба типа являются одинаковыми.
- Оба типа являются вещественными или целочисленными.
- Один тип является поддиапазоном другого.
- Оба типа являются поддиапазоном одного и того же базового типа.
- Оба типа являются множественными типами с совместимыми базовыми типами.
- Один тип является строковым, а другой – строковым или символьным типом.
- Один тип является указателем, а другой – указателем или ссылкой.

Существует еще один вид совместимости: *совместимость по присваиванию*, т.е. правила присваивания значения переменной X1 переменной X2. Они действительны только для операций присваивания и являются более широкими, чем правила совместимости по типам. Об этом будем говорить чуть позже.

Нарушение правил совместимости типов и значений обнаруживается, как правило, на этапе компиляции программы.

С вопросом совместимости типов очень тесно связан вопрос о типе результатов арифметических выражений. Здесь начинают действовать правила внутреннего преобразования типов значений – участников операций:

- В случае бинарной операции, использующей два операнда, оба операнда преобразуются к общему типу перед тем, как над ним совершается действие. Общим является встроенный целочисленный тип с наименьшим диапазоном, включающим все возможные значения обоих типов. Если же один операнд является вещественным, второй – целочисленным, то результат будет только вещественным;
- Выражение справа в операторе присваивания вычисляются независимо от размера переменной слева.

При внутреннем преобразовании типов существуют «подводные» камни: при операциях над целочисленными типами могут возникнуть трудно диагностируемые ошибки в результатах счета программы, а именно: значение результата может быть «обрезано» ее диапазоном.

<u>Пример:</u>

```
Var A, B: Word;
      C, D: Byte;
      F, K: ShortInt;
Begin
      A := 50000; B := A; C := 250;
      D := 2*C; B := B+A;
      F :=125; K:= 2*K;
      Write (D, B, K);
End.
```

При выводе значений переменных D, B и K программа выдаст заведомо неправильное значение, так как полученные результаты не вмещаются в заданные типы. Число

$$2*C = 100\,000 = 1\ \underbrace{1000\ 0110\ 1010\ 0000_2}_{\textit{Word} = 2 \text{ байта}}$$ будет преобразовано в число

$1000\ 0110\ 1010\ 0000_2 = 34464$, что не соответствует полученному результату. Число $B+A = 250+250 = 500 = 1\ \underbrace{1111\ 0100_2}_{\textit{Byte} = 1 \text{ байт}}$ будет преобразовано в число

$1111\ 0100_2 = 244$, которое «укладывается» в диапазон, обеспечиваемый типом ***Byte*** . Число K примет отрицательное значение, так как в старшем бите будет записана 1 (см. § 1.4.1.)=> $125*2 = 250 = 11111010_2$ => -6.

1.4.3. Явное преобразование (приведение) типов

В Turbo Pascal'е имеется очень мощное средство, позволяющее обойти всевозможные ограничения на совместимость типов или значений – приведение типов. Операция приведения типов применима только к переменным и значениям и построена на том, что любая информация в памяти ЭВМ представлена в двоичной системе счисления, интерпретировать ее возможно по-разному, в зависимости от того, какую «маску» (тип данных) наложить на заданную область памяти. Таким образом, с помощью конструкции ***ИмяТипа (Переменная_или_Значение)*** можно преобразовать аргумент к указанному типу. Рассмотрим это на примерах.

```
Var    L: LongInt;        {четырехбайтовое целое со знаком}
       S: ShortInt;       { однобайтовое целое со знаком}
       B: Byte;           {однобайтовое целое без знака}
       W: Word;           {двухбайтовое целое без знака}
Begin
    L := 1234567;  S := -2;
    B := Byte (S);          {B := 254}
```

```
    W := Word (L);          {W := 57920}
    L :=Word (W);           {L := 57920}
End.
```

Приведение типов не переопределяет типы переменных, оно лишь дает возможность нарушить правила совмещения типов при условии, что соответствующие значения совместимы в машинном представлении. Как видно из примера, можно изменять определенные байты общей структуры переменной в памяти независимо от ее типа. Разрешены также другие преобразования:

Integer (' Y') { код символа в формате Integer = 89 и занимает 2 б.}

Boolean (1) { логическое значение True}

Char (56-8) { Символ с кодом ASCII = 48., т.е. '0'}

1.5. Базовые операции

Математические выражения в алгоритмической записи состоят из операций и операндов. Большинство операций являются *бинарными*, т.е. содержат два операнда (*унарные* операции содержат один операнд, например : ***-a***, взятие адреса ***@B***).

Арифметические операции:

+, -, /, *, div (целочисленное деление), ***mod*** (остаток от деления),

Логические: ***not, and, or, xor,***

Операции отношения: >, <, >=, <=, <>, =.

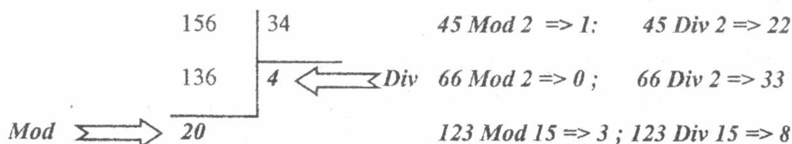
При вычислениях сначала применяются операции наивысшего порядка, затем более низкого в соответствии с таблицей приоритетов. Операции равного приоритета вычисляются слева направо: 4*5 / 6 / 12 = (((4*5) / 6) /12). Применение скобок позволяет явно расставлять приоритеты и менять порядок вычислений.

Замечание: Значение выражения X/Y всегда будет вещественного типа независимо от типа операнда.

Таблица приоритетов			
1	2	3	4
Not @ унарные –, +	and * / div mod	or - +	xor >, <, >=, <=, <>, =

приоритет убывает

Поясним дополнительно, как работают операторы целочисленной арифметики ***Div*** и ***Mod***. Операнды и результат должны быть целого типа. Оператор ***Div*** дает остаток от деления одного числа ***A*** на другое ***B***, а ***Mod*** — частное от деления.



С помощью оператора ***Mod*** можно делать проверку чисел на четность (кратность): если результат =0, то число четное.

1.5.1. Логические вычисления и операции отношения

Наличие типа ***Boolean*** и операций с ним позволяет программировать логические вычисления, в основу которых заложена *Булева алгебра*. Введены четыре логических операции, результат которых всегда имеет тип ***Boolean*** и может иметь только одно из двух значений (***True***≡1 (истинно) или ***False***≡0 (ложно)).

P=A or B			P= A and B			P=A xor B			P= not B	
P	A	B	P	A	B	P	A	B	P	B
0	0	0	0	0	0	0	0	0	1	0
1	0	1	0	0	1	1	0	1	1	0
1	1	0	0	1	0	1	1	0	0	1
1	1	1	1	1	1	0	1	1	0	1

Чтобы не было ошибок, при логических вычислениях лучше расставлять скобки самим. Так, например, запись ***Not A And B*** будет воспринята компилятором как ***(Not A)And B***, а, может быть, необходимо было бы так: ***Not (A And B).***

С порядком логических вычислений тесно связана особая возможность Turbo Pascal'я – поддержка двух различных моделей генерации кода для операций ***Or*** и ***And*** – полное вычисление или вычисление по короткой схеме. При полной схеме предполагается, что каждый член логического выражения будет вычисляться всегда; вычисление по короткой схеме прекращается, как только результат всего выражения становится очевидным (это очень удобно, т.к. она обеспечивает минимальное время выполнения и, как правило, минимальный объем кода). Полная схема необходима лишь тогда, когда один или более операндов представляют собой логические функции с побочными эффектами, которые изменяют смысл программы:

LogFunc1(x) And LogFunc2(x) And LogFunc3(x) . . .

Схема компиляции задается ключом компилятора $B. По умолчанию выставляется состояние ***{$B-}***, соответствующее короткой схеме вычислений.

Операции отношения приведены в таблице. Результат операции отношения – всегда булевское значение, сравнивать можно только совместимые простые значения, указатели, символы, строки.

Название					
Равно	Неравно	Больше	Меньше	Больше либо равно	Меньше либо равно
Обозначение					
=	<>	>	<	>=	<=

Логические выражения и операции отношения обычно используются в операторах управления с условием (If, While, Repeat ... Until), либо при вычислении логических переменных. Например:

If LogFunc1(x) And LogFunc2(x) Then < действие>

Flag := (A>B) And (C<D); *{при A=15, B=5, C=67, D=89 результат= True}*

Замечание: При сравнении вещественных типов нельзя быть уверенным в корректности проверки, т.к. из-за ошибок округления результаты могут отличаться от предполагаемых.

1.5.2. Математические процедуры и функции

Системная библиотека Turbo Pascal'я приведена в таблице:

Математические функции					
Вызов аргумента	*Тип аргумента*	*Тип значения*	*Назначение функции*	*Вызов функции*	*Результат*
Abs(x)	целый/вещ.	как у аргумента	Абсолютное значение X	Y:=Abs(x);	Abs(-6.5) = 6.5
Pi	–	Вещественный	Значение числа «Пи» 3,141592...	Y:=Pi;	-
Тригонометрические функции					
Sin(x)	Вещественный	Вещественный	Синус X радиан	Y:=Sin(x);	Sin(Pi)=0 Sin(-3*pi)=0
Cos(x)	Вещественный	Вещественный	Косинус X радиан	Y:=Cos(x);	Cos(0)=1 Cos(-Pi)=-1
ArcTan(x)	Вещественный	Вещественный	Арктангенс X радиан	Y:=ArcTan(x)	ArcTan(0)=

Sqrt(x)	целый/вещ.	Веществен-ный	Квадратный корень из X, X>0	Y:=Sqrt(x);	Sqrt(4)=2.0 Sqrt(-4)-ошибка
Sqr(x)	целый/вещ.ест.	как у аргумента	Значение квадрата X	Y:=Sqr(x);	Sqr(3.0)=9.0 Sqr(4)=16
Exp(x)	Веществен-ный	Веществен-ный	Значение в степени X	Y:=Exp(x);	Exp(0)=1.0
Ln(x)	Веществен-ный	Веществен-ный	Натуральный логарифм X, X>0	Y:=Ln(x);	Ln(1)=0.0 Ln(0)- ошибка
Trunc(x)	Веществен-ный	Целый	Целая часть значения X	Y:=Trunc(x);	Trunc(3.3)=3 Trunc(3.99)=3 Trunc(-4.5)=-4
Frac(x)	Веществен-ный	Веществен-ный	Дробная часть значения X	Y:=Frac(x);	Frac(6.89)=0.89 Frac(-6.89)=-0.89
Int(x)	Веществен-ный	Веществен-ный	Целая часть значения X	Y:=Int(x);	Int(-8.9)=-8.0 Int(8.9)=8.0
Round(x)	Веществен-ный	LongInt	Правильное округление X до ближайшего целого	Y:=Round(x)	Round(3.45)=3 Round(3.5)=4 Round(-3.45)=-3 Round(-3.51)=-4
Random	–	Веществен-ный	Случайное число (0…1)	Y:=Random;	-
Random(x)	Word	Word	Случайное число (0…X)	Y:=Random(x);	-
Odd(x)	Целый	логический	Возвращает True, если X – нечетное число.	Z:=Odd(x);	Odd(3)=True Odd(-3)=True Odd(4)=False
Математические процедуры					
Randomize			Гарантирует неповторяемость значений в случайной последовательности		
Inc(Var X:целое)			Увеличивает x на 1		
Dec(Var X:целое)			Уменьшает x на 1		
Inc(Var X:целое; N:целое)			Увеличивает x на N		
Dec(Var X:целое; N:целое)			Уменьшает x на N		

Математические функции очень чувствительны к диапазону своих аргументов. Кроме того, возвращаемые значения целых типов должны в них умещаться, иначе возможны фатальные последствия (см. § 1.4.2.). Большинство

функций являются стандартными и не нуждаются в комментариях. Отдельно хотелось бы остановиться на специфических.

Функция PI генерирует число «Пи» с точностью, зависящей от наличия сопроцессора,и содержит от 10 до 14 значащих цифр после запятой, она может использоваться в вычислениях как константа, но не может быть поставлена в вычислимые константы блока ***Const***.

Набор тригонометрических функций ограничен, но дает возможность расширить математическую библиотеку путем введения своих собственных функций, определяющих стандартные математические вычисления. Так, например, можно ввести логарифм с заданным основанием или степенную функцию, используя «базовый» набор.

Примеры:

{Вычисление логарифма x по заданному основанию a}

```
    Function Log(x, a: Real):Real;
    Begin
        Log := Ln(x)/Ln(a);
    End;
```

{Вычисление x^a***, a>0}***

```
    Function Degree(x, a: Real):Real;
    Begin
        Degree := Exp ( a * Ln (x) );
    End;
```

{ Вычисление Tg(x)}

```
    Function Tg(x: Real):Real;
    Begin
        Tg := Sin (x) / Cos(x);
    End;
```

У функции-датчика случайных чисел (ДСЧ) ***Random*** есть одна «неприятная» особенность: при последовательных запусках программ выдавать одни и те же случайные последовательности. Поэтому при моделировании

случайных процессов (величин), когда необходимо набрать статистику, рекомендуется использовать процедуру ***Randomize*** для инициализации ДСЧ, которая записывает случайное число, взятое со встроенного таймера, в так называемую «затравку» случайной последовательности, сбивая тем самым ее на новые числа. Того же самого можно достичь, если использовать системную переменную ***RandSeed*** типа LongInt , записывая каждый раз в нее произвольное, каждый раз новое, значение. На основе ДСЧ можно моделировать случайные последовательности с произвольным (заданным) законом распределения.

2. Управляющие структуры языка

2.1. Простые и составные операторы

Оператор в программе – это единое неделимое предложение, выполняющее какое-либо действие. Пример простого оператора – оператор присваивания, вызов какой-либо процедуры в программе и т.д. Важно, что под оператором всегда подразумевается действие, поэтому блоки описания (типов, констант и т.п.) не являются операторами. Два последовательных оператора разделяются друг от друга «;».Операция присваивания определяется так:

Переменная := Значение;

Оператор присваивания – это составной символ «:=», который читается как «становится равным». В операции присваивания слева всегда стоит имя переменной, а справа то, что представляет собой ее значение (значение как таковое, выражение, вызов процедуры или функции, другая переменная).

Если единое действие выполняется несколькими различными операторами, то говорят о составном операторе. Составной оператор – это последовательность операторов, перед которой стоит слово ***Begin***, а после – слово ***End***. Между операторами должна стоять «;», поэтому между оператором и ***End*** ее можно опустить, т.к. само слово ***End*** не является оператором.

Зарезервированное слово ***Begin*** тоже не является оператором, поэтому после него «;» не ставится.

Пример составного оператора:

```
Begin
.
.          {    } составной оператор (может быть пустой) }
End;
```

Пример:

Begin   A := 56.88; D := Sqtr(A); WriteLn('A=',A, ' D=',D)   End;

Составной оператор – очень важное понятие в структурном программировании, в Turbo Pascal'e все управляющие структуры не различают простой и составной операторы: там где стоит простой оператор, можнс поставить и составной.

2.2. Условный оператор

Условный оператор имеет следующую структуру

```
If <Логическое условие> then <Оператор1>
[else <Оператор2>];
```

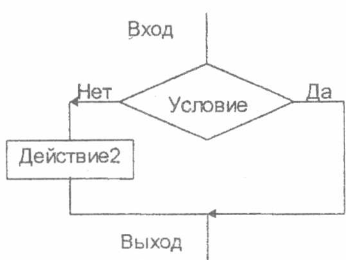
и служит для организации процесса вычислений в зависимости от какого-либо логического условия, которое может быть представлено логической константой, переменной или выражением, например:

If True Then ...; { логическая константа – бесполезное условие }
If LogicalVar Then ...;{ условие - логическая переменная }
If Not LogicalVar Then ...;{ условие - логическое выражение }
Iif x>(A-B)/(c+d) Then ...;{ условие - результат операции сравнения}

Существует несколько различных модификаций условного оператора, рассмотрим их, одновременно проиллюстрируем, как условный оператор будет выглядеть на блок-схеме:

***I f** <Логическое условие> **then** <Действие1>*
[else** <Действие2>**];

***If** <Логическое условие> **then** <Действие1>;*



If** <Логическое условие> **then
***Else** <Действие2>*

> ***Замечание***: «;» в конце всего условного оператора обязательна (после завершения ветви ***Else***), **<u>перед *Else* «;» не ставится</u>** !!!

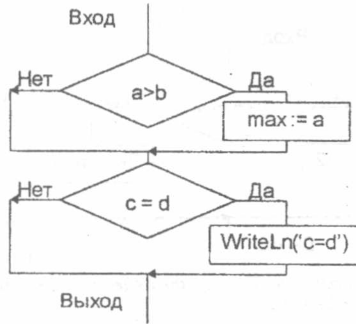
Условные операторы могут быть последовательными (идущими друг за другом) или «вложенными». При вложенных условных операторах самое главное – не запутаться в вариантах сочетаний условий (при этом очень помогает ступенчатая запись текста программы – запись «лесенкой»).

> ***Замечание***: Альтернатива ***Else*** считается принадлежащей ближайшему условному оператору IF , не имеющему ветви ***Else.***

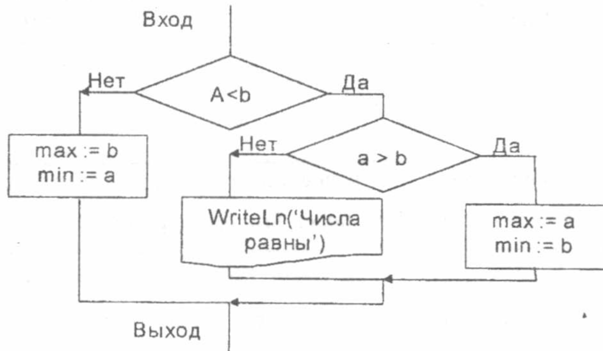
Именно это правило заложено в компилятор, и, как следствие этого, есть риск создать неправильно понимаемые условия.

Примеры:

```
If a > b then max := a;
If c = d then writeln('c=d');
```




```
{Найти минимальное и
максимальное значения среди
двух чисел a и b}
If a<b then
   If a>b then begin
      Max := a;
      Min := b;
   end
   Else writeln('Числа равны')
Else begin
   Max := b;
   Min := a;
End;
```



В следующем примере ошибка, связанная с тем, что **else** принадлежит ближайшему незавершенному **If**.

Неправильный вариант	Правильный вариант

If <У1> then
if <У2> then <Д1>
else <**Д2**>;
	If <У1> then
	if <У2> then <Д1>
	Else
	else <**Д2**>;

При записи условий могут использоваться простые и сложные логические выражения: a > b { **простое логическое выражение** }

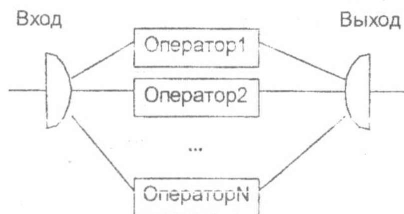
c+d*2 = a { **тоже простое выражение** }

Сложные логические выражения составляются при помощи логических операций **and, or, not, xor** (см.§ 1.5.1.).

2.3. Оператор выбора (варианта)

Оператор варианта необходим в том случае, когда в зависимости от значения какой-либо переменной надо выполнить те или иные операторы, в том числе и составные. Если вариантов два, то можно обойтись и условным оператором, а если их несколько, то лучше использовать ***Case***. Структура оператора варианта имеет вид:

```
Case <Параметр выбора> of
    <1-е значение параметра выбора>: <Оператор1>;
    <2-е значение параметра выбора>: <Оператор2>;
      ...
    <n-е значение параметра выбора>: <Операторn>;
      ...
    <k-ый набор значений параметра выбора>: <Операторk>;
      ...
[else <Альтернативный оператор>;]
end;
```



Параметр выбора (переменной или выражения) должен быть строго перечислимого типа (включая типы ***Char*** и ***Boolean***), диапазоном или целочисленным типом, все прочие типы не будут пропущены компилятором. Набор значений – это конкретные значения управляющей переменной или выражения, при которых необходимо выполнить соответствующий оператор, игнорируя остальные варианты. Если в наборе несколько значений, то они отделяются друг от друга запятой, можно указывать диапазоны значений. В записи оператора допускается необязательная часть ***Else,*** которая будет выполнена, если значение переменной выбора не совпало ни с одним из перечисленных значений.

Примеры:

```
                    { Определить символ ]
Case ch of          { ch: char }
    '.', '!', '?':  WriteLn('Знаки припинания');
    '0'..'9':       WriteLn('Цифры');
    ';':            WriteLn('Точка с запятой');
    'A'..'Z',
    'А'..'Я':       begin
                      WriteLn('Заглавные буквы.');
                      WriteLn('Русские или латинские.');
                    end;
else    WriteLn('Неопознанный символ');
end;
```

{ Вычислить значение функции }

$$y(x) = \begin{cases} ax^2, & 1 <= x < 2; \\ ax^3, & 2 <= x < 3; \\ ax^4, & 3 <= x < 4; \\ ax^5, & 4 <= x < 5; \end{cases}$$

```
N := Trunc(x);
Case N of    { N: byte }
   1:  y := a*Sqr(x);
   2:  y := a*x*x*x;
   3:  y := a*Sqr(Sqr(x));
   4:  y := a*Sqr(Sqr(x)*x;
else   WriteLn('функция не',
'определена по значению');
end;
```

Оператор Case очень удобен и, как правило, более эффективен, чем несколько операторов IF того же назначения. Эффективность его в смысле скорости будет максимальной, если размещать наиболее вероятные значения (или их наборы) первыми в порядке следования.

2.4. Операторы цикла

В практике программирования циклы – выполняющиеся повторения одних и тех же простых или составных операторов – играют очень важную роль. Существует три различных способа организации цикличных вычислений.

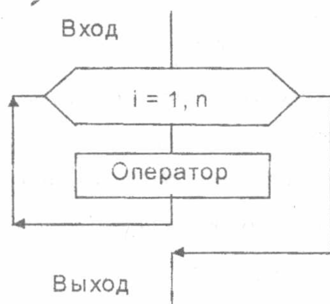
2.4.1. Итеративный (цикл с шагом)

Данный тип цикла является «строгим», т.е. выполняет действие заданное число раз. Синтаксис цикла с параметром следующий:

for** <Параметр цикла> := <Начальное значение> **to** <Конечное значение> **do
<Оператор>; *{ шаг = 1 }*

for** <Параметр цикла> := <Начальное значение> **downto** <Конечное значение> **do
<Оператор>; *{ шаг = -1 }*

На блок-схеме алгоритма цикл обозначается следующей структурой:



Оператор, представляющий тело цикла, может быть простым, составным или пустым (в этом случае после *Do* сразу ставится «;»).

Примеры (сколько раз выполнится цикл?):

```
for i := 1 to 3 do;          { 7 раз       i: Integer, byte, ... }
for ch := 'a' to 'd' do;     { 4 раза      ch: char }
for i := 5 to 5 do;          { 1 раз }
for i := 7 to 1 do;          { ни разу }
for i := 7 downto 1 do;      { 7 раз }
for i := -7 to 7 do;         { 15 раз }
for i := 3.5 to 5.5 do;{ Ошибка!!! Не перечислимый тип }
for i := True to False do;   { ни разу }
for ch := 'A' to 'Я' do;     { ? раз }
for ch := 'a' to 'я' do;     { ? раз }
```

В двух последних строчках число повторений зависит от кодировки, установленной на компьютере.

Далее приведены основные ***правила***, необходимые для использования оператора **for**:

1. Параметр цикла, начальное значение и конечное должны быть перечислимого типа.
2. Количество итераций вычисляется по формуле: <Кол-во операций> = <Конечное значение> - <Начальное значение> +(-) 1
3. Параметр цикла внутри цикла переопределять нельзя.

   For i:= ~~***i+5***~~ ***to n do ;***

   For i:= 1 to n do i := ~~***i+5***~~***;***

4. Параметр цикла за пределами цикла считается неопределенным.
5. Заголовок цикла читается только один раз.

Компилятор несоблюдение правила № 2 «не замечает», но программа с приведенными заголовками не заслуживает никакого доверия.

Итеративные циклы – очень быстрые и генерируют компактный выполнимый код, но правила №1-2 ограничивают их применение.

Циклы ***For*** предполагают вложенность при условии, что никакой из вложенных циклов, наряду с другими операторами, не использует и не модифицирует переменные – параметры внешних циклов.

2.4.2. Цикл с предусловием

Синтаксис цикла с предусловием следующий:

While *<Условие входа в цикл>* ***do*** *<Оператор>;*

На блок-схеме алгоритма цикл обозначается следующей структурой:

Оператор («тело цикла») будет выполняться до тех пор, пока выполняется логическое условие, т.е. пока значение «Условия» равно ***True***. Само условие может быть логической константой (возможен «вечный цикл»), переменной или выражением с логическим результатом. Условие проверяется перед выполнением каждой итерации, поэтому если условие сразу не выполняется, то «тело цикла» игнорируется и оператор (простой или составной) выполняться не будет.

Основные ***правила***, которые необходимо соблюдать для правильного использования цикла ***While***:

1. Перед выполнением цикла всем значениям, участвующим в вычислениях, должны быть присвоены начальные значения (иначе вычисления могут быть выполнены некорректно). Это касается как переменных, участвующих в вычислении условия, так и в действии.

2. Чтобы программа не «зациклилась», действие (оператор) должно влиять на условие выполнения цикла.
3. Переменная цикла должна изменяться внутри цикла принудительно, в явной форме.
4. За пределами цикла переменная цикла сохраняет свое значение.
5. Переменная цикла может иметь любой числовой тип.

Так как цикл ***While*** допускает все что угодно внутри себя, тело цикла может содержать другие вложенные в него циклы (количество уровней вложенности определяется сложностью программы.).

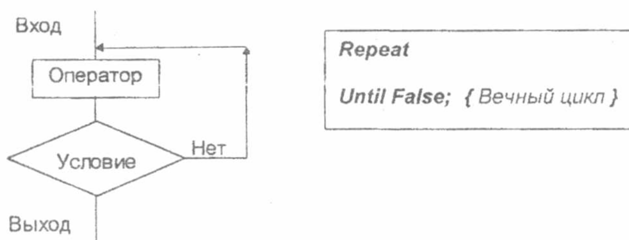
2.4.3. Цикл с постусловием

Синтаксис цикла с постусловием следующий:

```
Repeat
   <Оператор1>;
   <Оператор2>;
   ...
   <ОператорN>;
Until     <Условие выхода из цикла>;
```

Данный цикл используется в том случае, когда по логике алгоритма необходимо проверять условие завершения вычислений *после* очередной итерации, поэтому ***тело цикла выполняется всегда хотя бы один раз***. Это имеет значение лишь на первом шаге вычислений, далее цикл ведет себя точно так же, как и цикл ***While***.

На блок-схеме алгоритма цикл обозначается следующей структурой:



> ***Замечание***: В силу своей гибкости циклы ***While*** и ***Repeat*** используются для контроля ввода переменных.

Примеры:

{Вычисление значения факториала N!= 1*2*3*4*5*…*N}

For

```
Var     n, i : Byte;
        N_Fact:LongInt;
Begin
  Write('Введите n<10');
  ReadLn(n);
  N_Fact:=1;
  For i:=2 to n Do
     N_Fact:=N_Fact*i;
   Write('Факториал=',
         N_Fact);
end.
```

While

```
Var     n, i : Byte;
        N_Fact:LongInt;
Begin
  Write('Введите n<10');
  ReadLn(n);
  N_Fact:=1;
  i:=2;
  While i<= n Do Begin
     N_Fact:= N_Fact*i;
     Inc(i);
  End;
   Write('Факториал=',
         N_Fact);
end.
```

Repeat

```
Var     n, i : Byte;
        N_Fact:LongInt;
Begin
  Write('Введите n<10');
  ReadLn(n);
  N_Fact:=1;
  i:=1;
  Repeat
     N_Fact:= N_Fact*i;
     Inc(i);
  Until i>n;
  Write('Факториал=',
        N_Fact);
end.
```

При вычислениях необходимо помнить о том, чтобы правильно выбрать тип результата. Это особенно касается переменных, хранящих значения произведений (типа факториала), так как значение результата «растет» очень быстро и в результате может произойти переполнение.

В рассмотренных выше примерах вводилось ограничение на диапазон входного значения (n<10). Для того чтобы программа корректно отработала, необходимо, чтобы контроль ввода осуществлялся в явном виде.

Для этого программист должен взять контроль «на себя» и отработать все «критические» ситуации, которые могут привести к аварийному завершению программы и досрочному ее завершению. Компилятор жестко контролирует при выполнении программы границы массивов и соответствие типов. Этим

контролем можно управлять (включать или отключать при необходимости). Достаточно при этом компилятору указать, каким образом ему действовать в той или иной ситуации – задать соответствующую директиву специальным комментарием.

{$R -} – выключить контроль границ массивов;

{$R +} – включить контроль границ массивов;

{$I -} – выключить контроль ввода;

{$I +} – включить контроль ввода.

Для проверки правильности ввода нужно использовать системную переменную***IOResult*** типа Integer (Input/Output Result – результат операции ввода/вывода). IOResult=0, если ввод произведен правильно.

```
Write('Введите значение N<10');
{$I-} Readln(n); {$I+}
While IOResult<>0 do Begin
    Write('неправильный ввод');
    {$I-} Readln(n); {$I+}
End;
```

```
Write('Введите значение N<10');
Repeat
   {$I-} Readln(n); {$I+}
   If IOResult=0 then
    Else Write('неправильный ввод');
Until IOResult=0;
```

Добавим контроль диапазона числа:

```
Write('Введите значение N<10');
{$I-} Readln(n); {$I+}
While (IOResult<>0) Or (N>9) Or
      (N<0) do Begin
    Write('Неправильный ввод');
    {$I-} Readln(n); {$I+}
End;
```

```
Write('Введите значение N<10');
Repeat
   {$I-} Readln(n); {$I+}
   If IOResult=0 then
   Else Write('Неправильный ввод');
Until (IOResult=0) And (n>0)
       And(N<10);
```

3. Основные структурные типы языка

При программировании реальных задач необходимым условием удачного решения является правильный выбор формы представления данных. Каждый тип данных определяет вид действий над ними. От того, насколько программное представление соответствует структуре обрабатываемых данных, зависит размер и эффективность программы.

3.1. Массивы

3.1.1. Описание массивов

Массив - это регулярная структура, расположенная в памяти последовательно и имеющая имя. Для объявления массивов используется специальная конструкция языка, которая имеет следующий вид:

Array [Диапазоны индексов] of ТипКомпонентов

Наиболее* часто массивы используются для хранения дискретных последовательностей чисел (так называемых *векторов* или *матриц*). При объявлении массивов необходимо соблюдать лишь два требования:

1. Если диапазон – числовой, то он обязан «уместиться» в тип Word;
2. Произведение количества компонентов массива на размер компонента в байтах не может превышать 65520 байт (почти 64 Кбайт). Это требование является общим для всех структур данных языка Pascal.

Таким образом, массивы могут быть описаны так

```
Var <Имя массива>:
    array  [<Начальное значение индекса> ..
            <Конечное значение индекса>] of <Тип элементов массива>;
```

Var a: array [1..10] of Real;

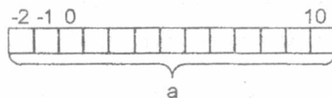
Так хранится в памяти одномерный массив из 10 вещественных чисел.

Примеры объявления массивов:
1).*{ объявление вектора из 10 элементов типа Real }*

```
const N_max = 10;
var a: Array [ 1 .. N_Max ] of Real;
2).        (* такое объявление удобнее *)
const N_max = 10;
Type Vector = Array [ 1 .. N_max ] of Real;
var     a: Vector;
        b: Array [1..n] of Real;
        c,d: Vector;
```

С точки зрения транслятора Pascal **a** и **b** разные, а **c** и **d** одинаковые и можно выполнять групповые присваивания (операции).

```
3). Const   nmax = 10;
            nmin = -2;
Type Vector = Array [ nmin .. nmax ] of
                    Real;
{ 13 элементов }
        VectorChar = array ['a'..'z'] of char;          { 26 элементов }
var     a: Vector;
        b: VectorChar;
        Switch : Array [ Boolean ] of Byte;
```

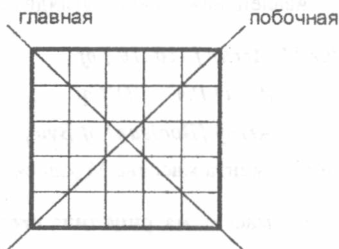


Как видно из третьего примера, ничто не обязывает объявлять диапазон индексов массива числами от 1 до *N_max*, в качестве индексов можно использовать любые перечислимые типы, как встроенные, так и вводимые пользователем. Если диапазон соответствует всему типу, то вместо него можно записать просто имя этого перечислимого типа.

Рассмотренные массивы – одномерные, то есть такие, у которых компоненты – скаляры. Разрешено объявлять массивы массивов (матрицы).

```
const     N_Max = 5;
          M_Max = 6;
Type      Matrix = array [ 1 .. N_Max , 1 .. M_max] of Real;
Type      Matr_Sqr = array [ 1..N_Max, 1 .. N_Max ] of Real;
var       a: Matrix;          {матрица n x m}
```

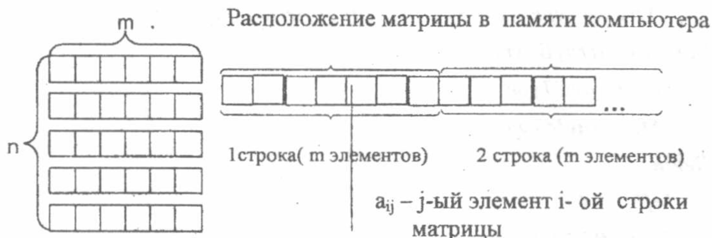
a_{11}	a_{12}	a_{13}	a_{14}	a_{15}	a_{16}
a_{21}					
a_{31}		a_{ij}			
a_{41}					
a_{51}					




```
          b : Matr_Sqr;          {квадратная матрица}
```

Другое описание двумерного массива (вектор векторов) выглядит следующим образом:

```
const n = 5; m = 6; {объявление матрицы размером n x m}
Type      Vector = array [1..m] of Real;
          Matrix = array [1..n] of Vector;
var       a: Vector;
          b: Matrix;
```



Первый вариант описания матрицы более наглядный, так как позволяет сразу определить размеры «таблицы», в которой будут храниться данные.

Замечание: Рекомендуется описывать массивы, используя раздел констант и типов для быстрого изменения не только описания, но и для минимальных исправлений по дальнейшему тексту программы.

В многомерных массивах (количество измерений формально не ограничено) каждое измерение не зависит от остальных, поэтому можно объявлять массивы с разными индексами:

Var M: Array [-10..10, 'A' .. 'D', Boolean] of Byte;

Эквивалентная запись выглядит следующим образом:

```
Var M: Array [ -10..10 ] of
        Array ['A' .. 'D'] of
        Array [Boolean] of Byte;
```

Тип элемента массива M зависит от числа указанных в нем индексов:

M[0] – массив-матрица типа ***Array ['A'..'D', Boolean] of Byte;***

M[0,'A'] – вектор типа ***Array [Boolean] of Byte;***

M[0,'A', True] – значение типа ***Byte***.

При таком описании массивов необходимо помнить о совместимости типов при выполнении операции присваивания.

```
Type ArrayBType = Array [ Boolean ] of Byte;
     ArrayCType = Array ['A' .. 'D' ] of ArrayBType;
     ArrayMType = Array [-10 .. 10] of ArrayCType;
Var B,A: ArrayBType;
     C: ArrayCType;
     M: ArrayMType;
Begin
   M[0] := C;
   B := M[0,'A'];
   A := B;
End.
```

В ***Turbo Pascal'e*** разрешается записывать индексы не только через запятую, но и отдельно:

M[0, 'A', True] *тождественно* *M[0] ['A'] [True]*

В памяти компьютера массивы располагаются последовательно (первая строка, вторая строка и т.д.), быстрее всего изменяется самый «дальний» индекс, если их несколько. Для определения стартового значения массива (в разделе констант) необходимо задавать значения так, как они будут располагаться в памяти (без учета скобок):

```
Type Dim1x5 = Array [ 1 .. 5] of Real;
Dim4x3x2 = Array [ 1 .. 4, 1 .. 3, 1 .. 2] of Byte;
Const D1x5 : Dim1x5 = ( 2.6, 7.9, -67.99, 3.89, 3.89e-06 );
      D4x3x2 : Dim4x3x2 = ( ( (1,2), (3,4), (5,6) ),
                            ( (34,56), (23,89), (24,76) ),
                            ( (12,3), (25,67), (56,45) ),
                            ( (90,56), (125,59), (66,99) ) );
```

При задании структур типа Array of Char, базирующихся на символах, можно не перечислять символы, а слить их в одну строку соответствующей длины

```
Const CharArray: Array[1 .. 10] oh Char ='abcdefghlk';
```

Замечание: Тип можно формировать прямо в описании переменной, но предпочтительнее использовать введенное ранее имя этого типа.

При компиляции программы ***Turbo Pascal*** контролирует принадлежность индекса указанному диапазону (по умолчанию ключ компилятора установлен в положение ***{$R+}***) и в случае нарушения границ диапазона программа прерывается с выдачей сообщения об ошибке 201 (Range Check Error). В режиме компиляции ***{$R-}*** никаких проверок не производится и некорректное значение индекса извлечет какое-нибудь значение, не принадлежащее данному массиву. Поэтому при отладке программы необходимо использовать ключ ***{$R+}***, а при эксплуатации ***{$R-}***, это несколько уменьшает размер Exe-файла и время его выполнения.

Адрес начала массива в памяти соответствует адресу его первого элемента (элемента с минимальными значениями индексов).

Замечание: Работа с элементами массива занимает больше времени, чем со скалярной переменной (необходимо вычислять местоположение элемента в памяти). Поэтому, если какой-то элемент массива используется многократно, то его лучше переписать в скалярную переменную.

3.1.2. Ввод и вывод массивов

Ввод массивов можно организовать по-разному: для одномерных массивов (векторов) – в строку или в столбец; для двумерных (матриц) – по строкам или по столбцам.

```
                {Ввод вектора в столбец }
for i = 1 to n do RealLn(a[i]); { Можно но не нужно! }

             { Лучше ввод организовать так: }
WriteLn('Введите элементы вектора:');
for i := 1 to n do begin
   Write('a[', i, '] = ');
   ReadLn( a[i] );
end;
                   { Вывод вектора: }
Writeln('Результирующий вектор: ');
for i := 1 to n do Write(a[i], '_');
WriteLn;
              {Ввод матрицы по строкам }
for i = 1 to n do Begin
  for j := 1 to m do
    Real(a[i,j]);
  ReadLn;
End;
          { Лучше ввод матрицы организовать так: }
WriteLn('Введите элементы матрицы:');
for i := 1 to n do begin
  WriteLn( 'Введите элементы ',i,'-ой строки матрицы');
  for j := 1 to m do Begin
    Write( 'a[', i, ',' , j,'] =  ');
```

```
      ReadLn(a[i,j]);
    End;
End;
                              {Вывод матрицы по строкам }
Writeln('Результирующая  матрица');
for i := 1 to n do begin
    for j := 1 to m do Write(a[i,j], '_');
    WriteLn;
end;
```

3.1.3. Форматы вывода

Для удобства вывода при выводе данных лучше использовать так называемый «форматный вывод», когда под каждую выводимую переменную отводится определенное количество позиций

Write(a [:F [:n]]); *{ [ ] здесь означают необязательность }*

F - общее число позиций, отводимое переменной (если необходимо вывести больше, то F игнорируется).

n - число знаков после десятичной точки (употребляется только для вывода действительных чисел).

<u>Примеры</u>:

```
a: real;          { a = 354.78 }
Write(a);              { 0.3547800000E+03  =>  0.35478*10^3 }
Write(a:8:4);          { 345.4700 }
I: Integer;            { I = 371 }
Write (I:8);           {_ _ _ _ _371 }
Write (I:-8);          { 371 }
Write (a:6:2);         { a = 15.448 }
                  { _ 15.45 }
Write ( <...> ); { Просто выводит информацию на экран }
WriteLn( <...> );      { Выводит на экран и переводит курсор на следующую
строку }
WriteLn;               { переводит курсор на следующую строку }
Write ('сообщение пользователя': 50);    {можно центрировать по ширине }
```

3.1.4. Базовые алгоритмы обработки векторов

Описательная часть

```
var     a: array [1..n] of <T>;
        min, max : <T>;
        Number :  Byte;
```

1). Сумма элементов массива:

1 вариант

```
S:=0;
For i:=1 to n do S:=S+a[i];
```

2 вариант

```
S:=a[1];
For i:=2 to n do S:=S+a[i];
```

2). Найти произведение четных элементов:

```
p:=1;
for i:=1 to n do
if a[i] mod 2=0 then p:=p*a[i];
```

3). Найти минимальный элемент массива:

```
min:=a[i];
for i:=2 to n do
   if a[i]<min then min:=a[i];
```

4). Найти минимальный элемент и его номер:

```
{1 вариант}
min:=a[1]; number:=1;
for i:=2 to n do
if a[i]<min then begin
   min:=a[i];
   number:=i;
end;
write(min,' ',number);
```

```
{2 вариант}
number:=1;
for i:=2 to n do
if a[i]<a[number] then number :=i;
write(a[number],number);
```

3.1.5. Алгоритмы поиска

1) Поиск заданного элемента в неупорядоченном массиве:

```
i:=1;
while (i<=n) and (a[i]<>x) do inc(i);
if n>i then
  write('Такого элемента нет')
else
write('Это элемент с номером' ,i);
```

```
i := 0;
Repeat
   inc(i);
Until (i >n) or (a[i] = x);
if i > n then
  write('Такого элемента нет')
else
write('Это элемент с номером' ,i);
```

Алгоритм содержит ошибку: выход за границу массива при отсутствии элемента.

```
i := 1;
While a[i] <> x do
   inc(i);
```

```
i := 1;
While (a[i] <> x) and (i <= n) do
   inc(i);
```

При другой реализации программы возможны скрытые ошибки (не рекомендуется изменять переменную цикла внутри цикла).

```
k := 0;
for i := 1 to n do
   if a[i] = x then begin
      k := i;
      i := n;          {     <—   Возможна ошибка }
   end;
```

2) Найти в массиве максимальное четное число:

```
i:=1;             {поиск первого четного элемента вектора}
while (i<=n) and (a[i] mod 2=1) do inc(i);
if i>n then write('Четных элементов нет')
Else begin {четный элемент найден}
   max:=a[i];
   for k:=i+1 to n do
      if (a[k] mod 2=0) and (a[k]>max) then max:=a[k];
      Write('max=', max);
end;
```

3) Алгоритм поиска последнего отрицательного элемента.

Правильный и наиболее оптимальный алгоритм.	Правильный, но не оптимальный.

`i := n+1;`
`repeat`
`  dec(i);`
`until (i < 1) or (a[i] <0);`
	`k := 0;`
	`for i := 1 to n do`
	`  if a[i] = x then k := i;`

4). Инвертирование вектора.

В другой вектор	Сам в себя

`For i : = 1 to n-1 Do`
`B[n-i+1] : = a[ i ];`
	`For j : = 1 to n div 2 Do begin`
	`  v := a[i];`
	`  a[i] := a[n-i+1];`
	`  a[n-i+1] := v;`
	`end;`

3.1.6. Базовые алгоритмы обработки матриц

1) Сумма элементов матрицы.

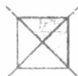
```
S := 0;
for i := 1 to n do
  for j := 1 to m do
    S := S + a[i,j];
```

2) Сумма элементов под главной диагональю (диагональ учитывается).

```
S := 0;
for i := 1 to n do
  for j := 1 to i do
    S := S + a[i,j];
```

3) Сумма элементов над главной диагональю (диагональ учитывается).

```
S := 0;
for i := 1 to n do
  for j := i to n do
    S := S + a[i,j];
```

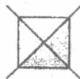


4) Сумма над побочной диагональю.

```
S := 0;
for i := 1 to n do
   for j := 1 to n-i+1 do
      S := S + a[i,j];
```

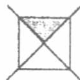
5) Сумма под побочной диагональю.

```
S := 0;
for i := 1 to n do
   for j := n-i+1 to n do
      S := S + a[i,j];
```



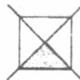
6) Сумма элементов над главной и побочной диагоналями.

```
S := 0;
for i := 1 to (n+1) div 2 do
   for j := 1 to n-i+1 do
      S := S + a[i,j];
```



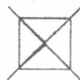
7) Сумма под главной и побочной диагоналями.

```
S := 0;
for i := n div 2 +1 to n do
   for j := n-j+1 to i do
      S := S + a[i,j];
```



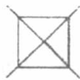
8) Сумма слева от главной и побочной диагоналей.

```
S := 0;
for i := 1 to (n+1) div 2 do
   for j := 1 to n-i+1 do
      S := S + a[i,j];
```



9) Сумма справа от главной и побочной диагоналей.

```
S := 0;
for i := n div 2 +1 to n do
   for j := n-j+1 to i do
      S := S + a[i,j];
```



10) Сумма элементов главной диагонали.

```
S := 0;
for i := 1 to n do S := S + a[i,i];
```

11) Сумма элементов побочной диагонали.

```
S := 0;
for i := 1 to n do S := S + a[i,n-i+1];
```

12) Транспонирование матрицы.

```
var     a: array [1..n,1..m] of <T>; {T – заданный тип}
        b: array [1..m,1..n] of <T>;
        ...
Begin
  for i := 1 to n do
    for j := 1 to m do
      b[i,j] := a[j,i];
End.
```

13) Транспонирование матрицы в самой себе (относительно главной диагонали).

```
var     a: array [1..n,1..n] of <T>;
        V: <T>;
        ...
Begin
  for i := 2 to n do
   for j := 1 to i-1 do begin
      V := a[j,i];
      a[j,i] := a[i,j];
      a[i,j] := V;
    end;
```

14) Умножение матриц ($A(n \times m) \times B(m \times l) = C(n \times l)$)

```
For i:=1 to n do
  For j:=1 to m do begin
    c[i,j]:=0;
    For k:=1 to l do c[i,j]:=c[i,j]+a[i,k]*b[k,j]
    End;
```

15) Поменять местами k и l строки в матрице:

```
Const n_max=10; m_max:=15;
Type
  Matrix=array[1..n_max,1..m_max] of Real;
Var A, B, C: Matrix;
    R: real; i, n, k,l : Byte;

  Begin
    For i:=1 to n do Begin
      R:=a[k,i]; a[k,i]:=a[l,i]; a[l,i]:=R;
    End;
  End.
```

```
Const n_max=10; m_max:=15;
Type vector=array[1.n_max] of Real;
  Matrix=array[1..m_max] of vector;
Var A, B, C: Matrix;
    R: Vector; k, l :Byte;

  Begin
    R:=a[k];
    a[k]:=a[l];
    a[l]:=R;
  End.
```

Замечание: Описание массива влияет на программную реализацию алгоритма.

3.1.7. Алгоритмы сортировки массивов

Для упорядочивания массивов по разным признакам (по возрастанию, по убыванию, по невозрастанию, по неубыванию и т.д.) применяют алгоритмы сортировки, при которых элементы в массиве меняют свое месторасположение в зависимости от условия сортировки. Известно большое количество алгоритмов [1], но в учебных целях используются наиболее простые из них.

1) Обменная поразрядная сортировка (сортировка "пузырьком")

Для вектора из *n* элементов делается *(n-1)* просмотр, в каждом просмотре сравниваются попарно элементы, начиная с первого, если предыдущий элемент больше последующего, их меняют местами. После каждого просмотра очередной наибольший элемент встает на свое место (*сортировка по возрастанию*).

{Основной алгоритм}	{Модифицированный алгоритм}

``
For k:=1 to n-1 do
  For i:=1 to n-k do
   If a[i]>a[i+1] then begin
     R:=a[i];
     a[i]:=a[i+1];
     a[i+1]:=R;
   end;
``
	``
	k:=1;
	Repeat
	  Flag :=False;
	  For i:=1 to n-k do
	   If a[i]>a[i+1] then begin
	     R:=a[i]; a[i]:=a[i+1];
	     a[i+1]:=R;Flag:=True
	   end; Inc(k);
	Until Not Flag Or (k=n)
	``

2) Сортировка посредством выбора

Находится минимальный элемент массива и меняется местами с первым. Среди оставшихся элементов (то есть начиная со второго) опять находится минимальный и меняется местами со вторым и процесс повторяется.

```
For i:=1 to n-1 do Begin
  min:=i;
  for j:=i+1 to n do
    if a[j]<a[min] then min:=j;
  r:=a[min]; a[min]:=a[i]; a[i]:=r;
End;
```

3.1.8. Совместные алгоритмы обработки матриц и векторов

1). По матрице *A(n x m)* получить вектор, элементы которого равны сумме элементов строк матрицы:

```
For i:=1 to n do Begin
  S:=0;
  For i:=1 to m do S:=S+a[i,j];
  b[i]:=S;
End.
```

2) По матрице *A(n x m)* получить вектор, содержащий минимальный элемент соответствующих строк матрицы:

```
For i:=1 to n do Begin
   min:=a[i,1];
   For j:=2 to m do
   If a[i,j]<min then min:=a[i,j];
   b[i]:=min;
End.
```

3.2. Множества

3.2.1.Описание множеств

Множество – это набор значений из некоторого скалярного (перечислимого) типа. Скалярные типы (***Byte*** и ***Char***) вводятся языком, они – перечислимые (все элементы можно поштучно назвать) и на их основе можно построить множество. Если же их станет мало, то всегда можно ввести свой скалярный тип, например:

Type VideoAdapterType = (MDA, CGA, EGA, VGA, Other, NoDetected);

и использовать переменную

Var VideoAdapter : VideoAdapterType;

которая может иметь только перечисленные в задании типа значения. Далее можно ввести переменную – множество из тех же значений.

В описании множества как типа используется конструкция Set of и следующее за ней указание базового типа. Способов задания множеств несколько:

```
Type
   SetOfChar = Set Of Char;           {множество символов}
   SetOfByte = Set Of Byte;           {множество чисел}

   SetOfVideoAdapter = Set of VideoAdapterType;
         { множество из названий видеоадаптеров}
```

SetOfDigit = Set of 0 .. 9; {множество из цифр от 0 до 9}

SetOfDChar = Set of '0' .. '9'; {множество из символов от '0' до '9'}

SetOfVGA = Set Of CGA .. VGA;

{множество из названий видеоадаптеров}

Как видно из примеров, можно в задании типа «урезать» базовый тип, определяя диапазон его значений. В итоге множество может состоять только из элементов, вошедших в этот диапазон.

Разрешено описывать множество сразу в разделе переменных, но это возможно только в случае, если переменная-множество не является параметром процедуры или функции.

Var M1 : Set of ...;

В Turbo Pascal'е разрешено определять множества, состоящие не более, чем из 256 элементов (столько же содержат типы ***Byte*** и ***Char***). Каждый элемент множества имеет сопоставимый номер. Для типа Byte номер равен значению числа, для символов – определяется его ASCII-кодом. Всегда нумерация идет от 0 до 255, поэтому для множеств базовыми не являются типы ShortIn, LongInt, Word, Integer. Множества имеют весьма компактное машинное представление: на один элемент расходуется 1 бит. Поэтому для хранения 256 элементов множества понадобится всего 32 байта. Для меньшего диапазона значений понадобится еще меньше памяти.

Переменная, описанная как множество, подчиняется специальному синтаксису, элементы множества должны заключаться в квадратные скобки []:

SByte := [1..5, 6, 9,89,97]; {целые числа}

LatLetter := ['a' .. 'z', 'A' ..'Z']; {задание латинского алфавита}

Sdigit := ['0' .. '9']; {задание цифр}

RusLetter := ['А' .. 'п', 'р' .. 'я']; {задание русского алфавита}

SChar := ['а', 'е', 'и', 'о', 'у', 'ы', 'ю', 'я', 'э', 'ё']; {гласные буквы}

Empty :=[]; {пустое множество}

Замечание: Порядок следования внутри скобок не имеет значения так же, как не имеет значения число повторений (элемент в множество включается только один раз).

В качестве элементов множества в квадратные скобки могут включаться константы и выражения, если тип результата совпадает с базовым типом множества:

```
Var SetByte : Set of Byte;
        X : Byte;
Begin
   X:= 56;
   SetByte := [1, 7, X];
   SetByte := SetByte + [(X+5) Div 4];
   {и т.п.}
End.
```

3.2.2. Операции, применимые к множествам

Обозна-чение	*Название*	*Форма*	*Комментарий*	*Пример использования*
=	Проверка на равенство	*S1=S2*	Результатом будет *True*, если множества состоят из одинаковых элементов, и *False* – иначе	**S1:=['a', 'b'];** **S2:=['a'];** **S3:=['b', 'a'];** **if S2=S1{False}** **if S3=S1{True}**
<>	Проверка на неравенство	*S1<>S2*	Результатом будет *True*, если множества отличаются хотя бы одним элементом, и *False* – иначе	**S1:=['a', 'b'];** **S2:=['a'];** **S3:=['b', 'a'];** **if S2<>S1{True}** **if S3<>S1{True}**
<=	Проверка на подмножество	*S1<=S2*	Результатом будет *True*, если все элементы множества *S1* содержатся в *S2*, независимо от порядка их следования, и *False* – иначе	*S1:=['a', 'b'];* *S2:=['a'];* *S3:=['a' .. 'z'];* *S4:=[];* *if S1<=S3{True}* *if S1<=S2{False}* *if S4<=S2{True}*

>=	Проверка на надмножество	*S1>=S2*	Результатом будет *True*, если все элементы множества *S2* содержатся в *S1*, независимо от порядка их следования, и *False* – иначе	*S1:=['a', 'b'];* *S2:=['a'];* *S3:=['a' .. 'z'];* *S4:=[];* *if S1>=S3{False}* *if S1>=S2{True}* *if S2+['b']>=S2 {True}*
In	Проверка вхождения элемента в множество	*X in S2* *X in [...]*	Результатом будет *True*, если значение *X* принадлежит базовому типу множества *S2* и входит в множество [...] (*S2*), и *False* – иначе	*X:='9';* *S1:=['0'..'9'];* *S2:=['a'..'z'];* *If X in S1 {True}* *If X in S2 {False}*
+	Объединение (сумма) множеств	*S1+S2*	Результатом будет множество, полученное слиянием элементов этих множеств и исключением дублированных элементов	*S1:=['a '.. 'z'];* *S2:=['a', 'b', 'z '];* *S3:=['0 '. .'9'];* *S4:=S1+S2;* *{ ['a' .. 'z'] }* *S5:=S1+S3;* *{ ['a'..'z', '0'..'9'] }*
-	Разность (дополнение) множеств	*S1-S2*	Результатом будет множество из элементов, входящих в *S1*, но не входящих в *S2*	*S1:=['a '.. 'z'];* *S2:=['a', 'b', 'z '];* *S3:=['0 '. .'9'];* *S4:=S1-S2;* *{ ['c' .. 'y'] }* *S5:=S1-S3;* *{ ['a'..'z'] }*
*	Пересечение множеств	*S1*S2*	Результатом будет множество, состоящее только из элементов, которые содержатся одновременно и в *S1*, и в *S2*	*S1:=['a '.. 'z'];* *S2:=['a', 'b', 'z '];* *S3:=['0 '. .'9'];* *S4:=S1*S2;* *{ ['a', 'b', 'z'] }* *S5:=S1*S3;* *{ [ ] }*

Include (S1,X)	Включение элемента в множество	-	Результатом будет множество, состоящее из элементов, которые содержатся в *S1*, и *X*	*S1:=['a '.. 'z'];* *X := [';']* *Include(S1, X);* *{ ['a' ..'z', ';'] }*
Declude (S1, X)	Исключение элемента из множества	-	Результатом будет множество, состоящее из элементов, которые содержатся в *S1*, за исключением *X*.	*S1:=['a '.. 'z'];* *X := ['a']* *Declude(S1, X);* *{['b' ..'z']}*

> *Замечание:* Операции пересечения и объединения множеств не зависят от мест операндов, но операция дополнения чувствительна к порядку следования операндов в выражении.

Результат вычислений S1-S2 не будет в общем случае равен результату S2-S1,точно так же, как и результат выражения S1-S2-S3 будет зависеть от порядка вычислений (слева направо и наоборот), устанавливаемых компилятором. Обычно принято вычислять слева направо, но лучше не закладывать в программу явные особенности компилятора и вычислять «многоместные» разности через промежуточные переменные, избегая тем самым получение неправильных результатов.

Достоинства множеств очевидны:

- Гибкость представления наборов значений.
- Удобство их анализа и возможность накапливания однотипных значений или их выборок.
- Компактность кодировки.

Недостатки множеств – обратная сторона их достоинств. За компактность приходится платить невозможностью вывода их на экран (хотя содержимое множества можно посмотреть через отладчик); ввод множеств возможен только по элементам.

__Пример:__

{Ввод множества символов до нажатия точки или клавиши *Enter*}

```
Uses Crt;               {для использования функции ReadKey}
Var SChar : Set Of Char;
    Ch : Char;
Begin
  SChar := [];              { «обнуление значений»}
  Repeat
    Ch := ReadKey;          {Ожидание нажатия клавиши}
    If Ch = #0 Then         {если управляющая клавиша, то пропускаем}
    Else SChar := SChar + [Ch];
  Until Ch in ['.', #13];             {ожидание нажатия клавиши '.' или Enter}
  SChar := SChar - [Ch];              {исключение последнего символа}
  End.
```

{ Вывод элементов множества на экран }

```
Var SByte : Set of Byte;
    X : Byte;
Begin
  SByte := [23, 45, 67, 89, 123, 204, 213, 225];
            {Возможно заполнение множества }
            { в режиме диалога с пользователем}
  WriteLn('Содержимое множества Sbyte ');
  For X := 0 To 255
    If X in SByte Then Write(X:4);
End.
```

3.2.3. Хранение множеств в памяти компьютера

Несмотря на недостатки, множества – очень удобный инструмент для обработки данных и оптимальный для некоторых приложений способ хранения данных. Как уже говорилось п.3.2.1 , каждому элементу множества соответствует бит, определяемый по номеру элемента исчисления.

S1:=[1,3,5]

0	0	1	0	1	0	1	0

S2 := [3, 5, 7];

1	0	1	0	1	0	0	0

Пересечение множеств:	Объединение множеств:	Вычитание множеств:
S := S1*S2;	***S:=S1+S2;***	***S:=S2-S1;***
76543210	*76543210*	*76543210*
00101010	*00101010*	*10101000*
and	*or*	*импликация*
10101000	*10101000*	*00101010*
S = 00101000	*S =10101010*	*S =10000000*
[3,5]	*[1,3,5,7]*	*[7]*

```
Var Color:set of (Red, Blue, Green, Black);
{                    0     1     2     3}
Color:=[Red,Green];
```

7	6	5	4	3	2	1	0
0	0	0	0	0 Black	1 Green	0 Blue	1 Red

4. Обработка символов и строк

Язык Turbo Pascal дает возможность создавать программы с развитыми алгоритмами обработки символьной информации.

4.1. Символьный и строковый типы (Char и String)

Язык поддерживает стандартный символьный тип ***Char*** и динамические строки, описываемые типом ***String*** или ***String[n]***.

Значение типа ***Char*** – это непустой символ из алфавита компьютера, заключенный в одиночные кавычки. Кроме этой классической формы записи Turbo Pascal вводит еще два способа:

1) представление символа его кодом ***ASCII (American Standart Code for Interchange Information),*** для этого используется префикс #;

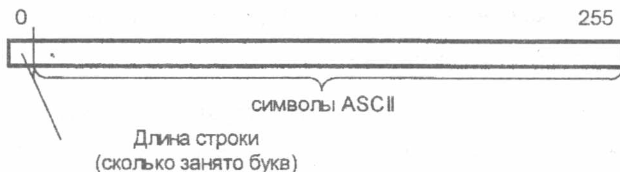
2) представление символа его клавиатурным обозначением (для управляющих символов с кодами в диапазоне от 0 до 31), для этого используется префикс ^;

Символ	*Кодовое представление*	*Клавиатурное представление*	*Комментарий*
‘a’	#97 = Chr (97)	-	Символ ‘a’
‘A’	#65 = Chr (65)	-	Символ ‘A’
‘ ‘	#32 = Chr (32)	-	Символ «пробел»
←	#24 = Chr (27)	^[	Стрелка влево
↑	#26 = Chr (24)	^X	Стрелка вверх
♦	# 4 = Chr (4)	^D	-
-	#7 = Chr (7)	^G	Звонок
-	#10 = Chr (10)	^J	Переход на следующую строку
-	#13 = Chr (13)	^M	Клавиша Enter
-	#0 = Chr (0)	-	Нулевой символ

Максимальная длина строки составляет 255 символов. Строки называются динамическими, поскольку они могут иметь различные длины в пределах объявленных границ. Тип ***String*** является базовым и совместим со всеми производными символьными типами. При попытке записать в строку длиннее, чем объявлено в описании, «лишняя часть» будет отсечена.

Тип ***String*** очень похож на массив символов ***Array [0..255] of char***, но это не совсем так. В нулевом элементе строки хранится ее длина (динамическая «переменная»), которая автоматически меняется в зависимости от наполнения строки символами, чего не скажешь о символьном массиве.

```
Const n=10;
var    a: String;        { В такой строке максимум 255 символов }
       b: String[n];     { В такой строке максимум 10 символов }
       c: String[255];   { то же, что и String }
```



Значением строки может быть любая последовательность символов, заключенная в одинарные кавычки:

'abcdefg….' {Английский алфавит}

'Строка' {произвольная строка}

'' { пустая строка }

'''' {строка из одной одиночной кавычки}

^G'После сигнала нажмите '^J' клавишу пробела'^J^M

{«Звонок» сообщение «переход на следующую строку» сообщение «переход на следующую строку» «клавиша Enter»}

В такой записи не должно быть пробелов вне кавычек. Более общим способом включения в строку любых символов является их запись по ASCII-коду через префикс # или функцию ***Chr***. Механизм показан ниже:

#179' Номер п/п '#179' Ф.И.О. ' +Chr(179)+' Должность '#179

{то же, что и '| Номер п/п | Ф.И.О. | Должность |'}

#7#32#179#32#32#179 {то же, что и **^G' | |'**}

Строки различных длин совместимы между собой в операторах присваивания и сравнения, но «капризно» ведут себя при передаче параметров в процедуры и функции (если тип фактического параметра является производным от строкового, то при вызове он должен совпадать с типом формального параметра (принципы передачи параметров в подпрограммы будут обсуждаться в следующих параграфах)). Поэтому для полной совместимости типов параметры, описываемые на уровне подпрограммы, должны быть описаны исключительно типом ***String*** и компилировать

программу необходимо в режиме {$V-} (режим проверки совместимости формальных и фактических параметров).

К любому символу в строке можно обратиться по его номеру, как к элементу массива, отдельный символ совместим по типу со значением типа ***Char,*** поэтому можно выполнять различные присваивания как на уровне символов строки, так и на уровне произвольных символов.

__Пример:__

```
Var Ch : Char;          I : Byte;       St : String[10];
Begin
   St := 'Example ';                    {длина 8 символов}
   For i := 1 to 4 Do Write (St[i]);    {'Exam'}
   Ch:='1';
   St := St + Ch;                       {'Example 1', длина 9 символов}
   St[9]:='2';   Write(St);             {St = ' Example 2'}
   St[10]:=Ch; Write(St);               {St = ' Example 21'}
End.
```

Как уже говорилось выше, каждая строка «знает», сколько символов в ней содержится в текущий момент времени. Символ ***St[0]*** содержит код, равный числу символов в строке St, т.е. длина строки всегда равна ***Ord(St[0])***. Об этой особенности необходимо помнить при заполнении строки одинаковыми символами. Проблему заполнения решает встроенная процедура ***FillChar*** :

```
Var  St : String;
Begin
   FillChar ( St[1], 50, ' ' );         { заполняем 50 пробелов }
   St[0] := Chr(50);                    { устанавливаем заданную длину строки }
End.
FillChar(St, SizeOf(St), 0);            { Обнуление строки }
```

> ***Замечание:*** Не рекомендуется принудительно изменять длину строки, т.е. символ St[0], за исключением рассмотренной ситуации.

В рассмотренной ситуации принудительное установление длины строки необходимо, так как сама процедура ***FillChar*** не устанавливает заданную длину. При работе с указанной процедурой всегда есть риск «выйти» за

пределы, отводимые заданной строке, и заполнить символом рабочую область памяти. Так как компилятор эту ситуацию не контролирует, то вся ответственность ложится на программиста.

4.2. Операции над символами

Символы можно лишь присваивать друг другу и сравнивать между собой. Сравнение осуществляется на уровне ASCII-кодов (так как в памяти по существу хранятся не сами символы, а их кодовые обозначения). Символы равны, если равны их ASCII-коды, один символ больше другого, если его код имеет большее значение

1) Присваивание.

<Переменная> := <Значение>; ***a := 'a';***

2) Сравнение (проводится традиционным способом по правилам лексикографического упорядочивания (как в словаре)).

>, <, <>, =, >=, <=

'a' > 'A' – *True* *{код 97 > кода 65}*

'a' <> 'A' – *True* *{ код 97 <> коду 65}*

3). Прочие.

Функция : тип	*Назначение*	*Примеры*
CHR(x: byte): char;	Возвращает символ ASCII кода, соответствующий переменной X	***CHR(32) = ' ';*** ***CHR(ORD(' ')) = ' ';***
ORD(ch: char): byte;	Возвращает код из ASCII по символу ***Ch***	***ORD(' ') = 32;*** ***ORD(CHR(32)) = 32;***
PRED(ch: char): char;	Возвращает предыдущий символ для ***Ch*** из ASCII-таблицы	***PRED('c') = 'b';***
SUCC(ch: char): char;	Возвращает следующий символ из ASCII-таблицы	***PRED('c') = 'b';***
UpCase(ch: char): char;	Переводит символ ***Ch*** в верхний регистр. *Действует только для латинских букв,* все остальные символы, в том числе и кириллицу, возвращает в исходном виде	***UpCase('n') = 'N';*** ***UpCase('N') = 'N';*** ***UpCase('m') = 'm';*** ***UpCase('M') = 'M';***

Внимание !!! Следующие значения функций не определены:
PRED(#0); ***SUCC(#255);***

4.3. Операции над строками

Строки можно присваивать, сливать и сравнивать.

1) ***Ввод/Вывод*** строк осуществляется с помощью стандартных операторов ввода/вывода:

ReadLn(S); ***{ ввод строки (максимальное число введенных символов 128 – определяется буфером клавиатуры (128 байт)) }***

Write(S); ***{ вывод строки (курсор остается в текущем положении) }***

WriteLn(S); ***{ вывод строки (курсор переводится в начало следующей строки) }***

Сравнение строк происходит посимвольно, начиная с первого символа в строке. Строки равны, если они равны по длине и посимвольно эквивалентны.

2) ***Присваивание строки.***

S1 := S2;

a := 'a';

Empty := '';

3) ***Сравнение строк*** (если при посимвольном сравнении окажется, что один символ (его код) больше другого, то строка, его содержащая, тоже окажется большей, остаток строк при этом будет игнорироваться и длины строк уже не играют роли)

'abcd' = 'abcd'	***{ True }***
'abcd' = 'ABCD'	***{ False, так как #65 <> #97 'a' <>'A' }***
' abcd' < 'abcd'	***{ True, так как #32 < #65 = 'a' }***
'Яковлев' > 'Я'	***{ True, так как длина первой строки больше }***
' ' <> ''	***{ True, так как #32 > '' }***
'__' = '_'	***{ False, так как длина первой строки больше }***
'__' > '_'	***{ True, так как длина первой строки больше }***
'' = ''	***{ True, так как обе строки пусты}***

4.4. Процедуры и функции обработки строк

Процедуры и функции	Назначение	Пример использования
	Редактирование строк	
Length(S: String): byte;	Функция определяет текущую длину строки	***l := Length(S);*** ***{ S := 'Дом'; l = 3 }*** ***l1 := Length('Дом');*** ***{ l1 = 3 }*** ***l2 := Length(S + 'Дом');*** ***{ l2 = 6 }***
Concat(S1,S2,S3, ... : String): String;	Функция проводит конкатенацию (слияние) строк. Если суммарная длина больше, чем 255 символов, то лишнее будет «обрезано»	***S1 := 'Ура! '; S2 := '6 ';*** ***S3 := 'факультету!';*** ***S := Concat(S1, S2, S3);*** *{ S = 'Ура! 6 факультету!' }* ***S := S1 + S2 + S3;*** { **делает то же, что и** *Concat* }
Copy(S: String; Start, Len: byte): String;	Функция возвращает подстроку из строки S, начиная с позиции **Start** и длиною **Len**. Если позиция находится за пределами строки, то возвращается пустая строка. Если же начальная позиция + длина больше длины строки, то копируется все до конца строки	***S := 'I love you!'*** ***{ l = 11 }*** ***SubS := Copy(S,3,4);*** ***{ SubS = 'love' }*** ***SubS := Copy(S,100,4);*** ***{ SubS = '' }*** ***SubS := Copy(S,3,100);*** **{ SubS = 'love you!' }**
Delete(var S: String; Start, Len: byte);	Процедура удаляет из строки **S** подстроку с позиции **Start** длиной **Len**. ***Замечание***: *изменяет длину строки.* Если **Start** больше длины строки, то строка остается без изменений. Если **Start+Len** больше длины строки, то удаляется подстрока с позиции **Start** до конца строки	***S := '6 факультет';*** ***Delete(S,2,2);*** *{ S = '6 культет' }* ***Delete(S,100,3);*** *{ S := '6 факультет' }* ***Delete(S,3,100);*** *{ S = '6 ' }* ***Delete('Попробуй удали!',3,8);*** *{ **Ошибка!** S не может быть константой }*
Insert(SubS: String; Var S: String; Start: byte);	Процедура вставляет подстроку **SubS** в строку S, начиная с позиции **Start**. ***Замечание***: *изменяет длину строки.* Если новая длина больше	***S := 'Начало - - Конец';*** ***SubS := ' Середина';*** ***Insert(SubS, S, 9);*** *{ S = 'Начало - Середина - Конец' }*

	255, то лишнее будет «обрезано». Если позиция **Start** выходит за длину строки, то вставка будет происходить в позицию за истинной длиной строки. Если строка ограничена, то вставка будет происходить в пределах заданной длины	***S := '123'*** ***Insert('abc', S, 100);*** *{ S = '123abc' }*
Pos(SubS, S: String): byte;	Функция возвращает позицию *первого* вхождения подстроки **SubS** в строку **S**. Если в строке такой подстроки нет, то выдается 0	***S := 'abc1244 a15 bc';*** ***Mask := '15';*** ***P := Pos(Mask,S);*** ***{ P = 10 }*** ***P: = Pos('dab',S);*** ***{ P = 0 }***
Процедуры преобразования		
Str(x [:F[:b]];var S: String); *{ [ ] - необязательность }*	Процедура преобразовывает число **x** в строку **S**. **F** – число символов, **b** – чисел после точки	***Str(3.1415:7:2, S);*** *{ S = '_ _ _ 3.14' }* ***Str(P:7,S);*** ***{ P: Word = 4433;*** ***S = '_ _ _ 4433' }***
Val(S: String; var x; ***Var ErrCode: Integer);***	Преобразовывает строку **S** в число **x** соответствующего типа. **ErrCode** = 0, если преобразование прошло успешно, либо это позиция первого неверного символа при преобразовании	***Write('Введите число: ');*** ***ReadLn(S); { S: String }*** ***val (S, x, ErrCode);*** ***{ x: Word }*** ***if ErrCode = 0 then*** ***Else WriteLn('Ошибка! Повторите ввод.');***

4.5. Базовые алгоритмы обработки строк

Описательная часть: ***Var St : String;***

1) Убрать все пробелы в начале строки:

While S[1]=' ' do Delete(St,1,1);

2) Убрать все пробелы в конце строки:

While St [Length(St)] =' ' do Delete(St, Length(St),1);

{или}

While Copy(St, Length(St),1)=' ' do Delete(St, Length(St),1);

3) Убрать все пробелы из строки:

While Pos(' ',St)>0 do Delete(St, Pos(' ',St), 1);

4) Убрать “лишние” пробелы между словами в строке:

```
While Pos('  ',S) > 0  do  Delete(S, Pos('  ',S),1);
```

5) Ввод чисел с контролем :

```
Var S:String;
      R: Real; Err : Integer;
begin
    .   .    .
    repeat
       Write('Введи число: '); ReadLn(S);
       Val(S, R, Err);
    until Err=0;
  .      .     .
end.
```

6) Ввод строк с контролем допустимых символов:

```
S:='';
repeat
   repeat
      Ch := ReadKey;
      if Ch in ['0'..'9','.','-'] then S := S + Ch;
   until Ch=#13;
   Val(S, R, Err);
until Err=0;
```

7) Вывод чисел на экран в графическом режиме.

```
X:= -37.5;
Str(X,S);  {Преобразуем в строку}
OutText(S);
```

8) Сортировка массива строк (строки сортируются по тем же алгоритмам, что и массивы чисел).

```
for k:=1 to n-1 do
   for i:=1 to n-k do
      if a[i]>a[i+1] then begin
         S := a[i]; a[i]:=a[i+1]; a[i+1]:=S;
end;  {пузырек}
```

Список рекомендуемой литературы

1. Поляков Д.Б., Круглов И.Ю. Программирование в среде Турбо Паскаль. Версия 5.5. М.: Издательство МАИ, А/О «Розвузнаука», 1992.
2. Фаронов В.В. Турбо Паскаль. Версия 7.0. Начальный курс. Учебное пособие. М.: Издательство Нолидж, 1997.
3. Епанишников А.М., Епанишников В.А. Программирование в среде Турбо Паскаль, 7.0. М.: Диалог-МИФИ, 1995.
4. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя. М.: Финансы и статистика, 1989.
5. Керниган Б., Плоджер Ф. Инструментальные средства программирования на языке ПАСКАЛЬ. М.: Радио и связь, 1985.

Учебное издание

АЛГОРИТМИЧЕСКИЕ ЯЗЫКИ И ПРОГРАММИРОВАНИЕ

Курс лекций для студентов заочной формы обучения

Зеленко Лариса Сергеевна

Михеева Татьяна Ивановна

Редактор Н. С. Куприянова
Корректор Т. И. Щелокова

Лицензия ЛР № 020301 от 30.12.96 г.

Подписано в печать18.08.99 г. Формат 60 × 84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л. 4,2. Усл. кр.-отт.4,3. Уч.- изд. л. 4,5.
Тираж 200 экз. Заказ 103

Самарский государственный аэрокосмический университет
имени академика С. П. Королева
443086 Самара, Московское шоссе, 34.

ИПО Самарского аэрокосмического университета.
443001 Самара, ул. Молодогвардейская, 151.
НПЦ "Авиатор" 443001 Самара, ул. Молодогвардейская, 151