В В Е Д Е Н И Е

Язык программирования Паскаль, разработанный в 1970 г. профессором Швейцарской высшей политехнической школы Никлаусом Виртом специально для целей обучения студентов, быстро завоевал широкую популярность благодаря своей простоте, логичности языковых конструкций и максимальному соответствию современным стандартам программирования. Однако основной успех в профессиональной среде ему был обеспечен с появлением диалектов под названием Турбо Паскаль, созданных американской фирмой Борланд. Последовательно сменяющие друг друга версии языка Турбо Паскаль, начиная с 3.0 (1985 г.), существенно увеличили функциональные возможности языка, сохраняя его первоначальную четкость и логичность построения. С помощью последних версий языка (7.0, Borland Pascal, Delphi) уже могут быть созданы программные комплексы практически любого объема и степени сложности.

В предлагаемом учебном пособии основное внимание уделяется алгоритмам и методике решения различного класса задач, но в тесной связи с соответствующими конструкциями языка Турбо Паскаль как инструментального средства программной реализации алгоритмов. При этом везде, где это возможно, дается не только формальное описание синтаксиса языка, но и смысловое определение языковых конструкций и соответствующих программных действий.

Практически во всех разделах пособия теоретические сведения проиллюстрированы примерами решения различных задач или их фрагментов. В приложении приведено свыше 100 программ с детальным их описанием, дано также около 200 задач по программированию для самостоятельного решения. Все программы, использованные в пособии, проверены на компьютере.

Учебное пособие представляет собой начальный курс программирования и не предполагает в обучаемых какого-либо опыта работы с компьютером, хотя наличие такого опыта по крайней мере в объеме курса информатики средней школы, безусловно, полезно. В пособии не рассматриваются вопросы организации и использования операционной системы MS DOS, операционной оболочки Norton Commander, интегрированной среды компилятора Turbo Pascal. Эти вопросы достаточно подробно изложены в литературе, их освоение является необходимым условием работы с компьютером.

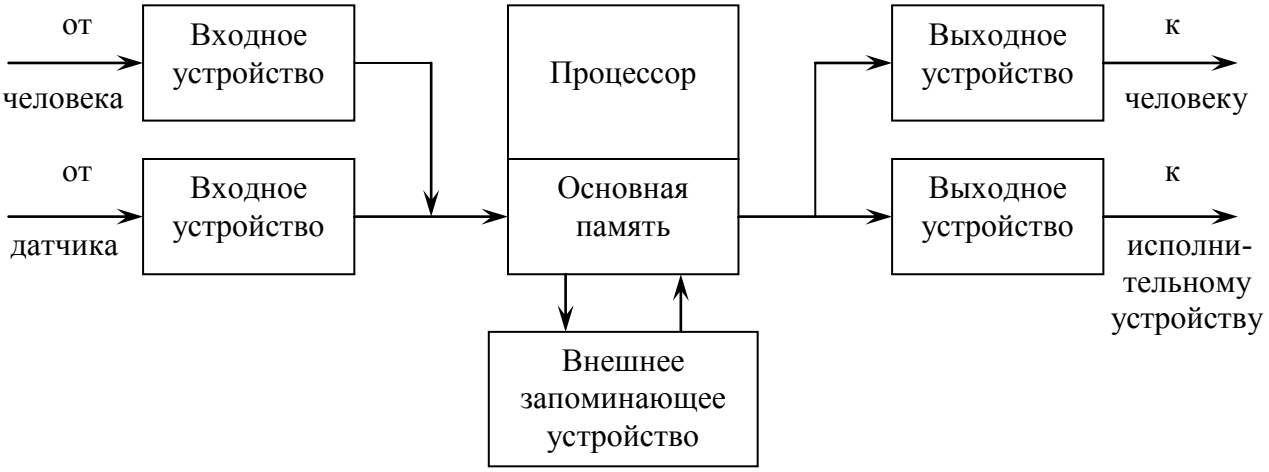
Язык Турбо Паскаль обладает богатыми функциональными возможностями, но далеко не все они необходимы при освоении начального курса программирования. Цель учебного пособия – обучение основам алгоритмизации и программирования, используя достаточный, с точки зрения автора, набор конструкций языка Турбо Паскаль. В связи с этим в описании языка не рассматривается целый ряд его элементов, не применяются многие стандартные процедуры и функции. Например, в составе разделителей не указывается символ табуляции, в разделах описания Паскаль-программы не упоминается предложение exports, не используются процедуры Mark, Release и др.. Все эти конструкции описаны в литературе и могут быть освоены студентом в дальнейшем самостоятельно.

В учебном пособии не затрагиваются также два больших раздела: машинная графика и объектно-ориентированное программирование. Вывод на экран графических примитивов не представляет сложности, но требует изложения большого количества деталей программной реализации, что значительно увеличило бы объем пособия. Серьезная же машинная графика (трехмерные объекты, мультипликация и т.п.) – это уже следующий этап профессиональной подготовки программиста. То же самое относится к объектно-ориентированному программированию, идеология которого является качественным развитием методов процедурного программирования, рассматриваемого в учебном пособии.

В заключение необходимо подчеркнуть, что успешное овладение начальными навыками программирования возможно лишь при внимательном изучении материала пособия с обязательным решением задач, приведенных в приложении, причем по крайней мере треть из них должна быть "проиграна" на компьютере.

СТРУКТУРНАЯ СХЕМА ЭВМ

Электронная вычислительная машина - это устройство для автоматической обработки информации. При этом источник и форма представления входной информации, адресат и форма представления выходной информации могут быть самыми различными. В укрупненном виде структуру ЭВМ можно изобразить следующим образом:



Входная информация может быть представлена на различных внешних носителях: на перфокартах, перфоленте, магнитной ленте, в виде сигналов от телефонных и телеграфных линий связи и т.п. В частности, на персональном компьютере основным устройством для ввода входной информации является его клавиатура.

Информация всегда вводится в основную память ЭВМ. Процессор производит непосредственную ее обработку. Промежуточные и конечные результаты вычислений записываются в память.

Основная память имеет сравнительно небольшой объем, но высокое быстродействие. При отключении питания ЭВМ информация в основной памяти, как правило, не сохраняется. Для длительного хранения больших объемов информации используются внешние запоминающие устройства: накопители на магнитной ленте (НМЛ), накопители на магнитном диске (НМД) и др. В персональных компьютерах для этой цели, как правило, используют накопители на гибких и жестких магнитных дисках.

Выходная информация может быть отпечатана на бумаге, представлена в виде рисунков и графиков на графопостроителе, выдана в виде электрических сигналов для исполнительных устройств автоматики и т.п. На ПЭВМ в качестве выходных устройств используются, как правило, экран дисплея и принтер.

ЭВМ может быть применена для непосредственного управления промышленными объектами. В этом случае оперативная информация поступает от датчиков, сигнализирующих о параметрах объекта (температура в различных узлах, давление, скорость и др.) и обрабатывается процессором по заданной предварительно программе, а полученная при такой обработке управляющая информация поступает от ЭВМ на исполнительные органы объекта (двигатели, электромагниты, пневматические устройства и т.п.).

Информация в ЭВМ содержится в виде последовательности двоичных цифр 0 и 1. Ее обработка осуществляется путем выполнения элементарных машинных команд (сложение, вычитание, умножение и т.п.). Последовательность таких команд называется машинной программой.

Например, вычисление выражения

$$z = \frac{a\ x + b\ y}{c - d}$$

на условном машинном языке выглядело бы примерно следующим образом:

```
1)  ( - )   < c >   < d >   R1
```

Из содержимого ячейки памяти, где находится число *c,* вычесть содержимое ячейки памяти, в которой находится число *d,* и результат записать в ячейку *R1*. Здесь `1)` – номер (адрес) машинной команды; `(-)` – условный код операции; `<c>` – адрес первого операнда, т.е. уменьшаемого *c*; `<d>` – адрес второго операнда, т.е. вычитаемого *d*; `R1` – адрес результата, т.е. номер ячейки памяти, в которую требуется записать результат вычитания $c - d$.

```
2)  ( * )   < a >   < x >   R2
3)  ( * )   < b >   < y >   R3
4)  ( + )     R2      R3    R2
5)  ( : )     R2      R1    < z >
```

В этой программе 5 команд. В каждой команде указывается код операции, адреса первого и второго операндов и адрес результата. Все составные части машинной команды - это также последовательности цифр 0 и 1.

Примечание. Операнд - это обобщенный термин для обозначения величин, над которыми выполняется та или иная операция, т.е. аргумент операции (например, для операции деления операндами являются делимое и делитель).

Приведенный выше фрагмент программы на машинном языке может иметь следующий вид:

Адрес команды	Машинная команда
10100011	01010111100110111001110010100000
10100100	01011000100110011001110110100001
10100101	01011000100110101001111010100010
10100110	01010110101000011010001010100001
10100111	01011001101000011010000010011111

Здесь в каждой машинной команде первые 8 двоичных цифр – код операции, остальные группы по 8 цифр – адреса соответственно первого операнда, второго операнда и результата.

Программа, как и обрабатываемые числа, записывается в память ЭВМ. Выполнение команд программы производится, как правило, последовательно, начиная с адреса первой команды (пускового адреса программы). Принцип работы ЭВМ, при котором обрабатываемая информация и обрабатывающая программа одновременно находятся в памяти ЭВМ, называют принципом программного управления.

Машинный язык для человека трудно воспринимаем. Программирование на машинном языке характеризуется очень низкой производительностью. Поэтому для разработки программ обработки данных используют различные алгоритмические языки (Фортран, Бейсик, Паскаль и др.), которые в значительной степени приближены к привычной для человека форме чтения и записи информации. Программа на алгоритмическом языке записывается в память ЭВМ, а затем транслируется (переводится) на машинный язык. Этот перевод осуществляется автоматически специальной программой, которая называется транслятором (или компилятором). ЭВМ обрабатывает информацию только на машинном языке.

СИСТЕМЫ СЧИСЛЕНИЯ

Системы счисления разделяются на позиционные и непозиционные. Примером непозиционной системы счисления является общеизвестная римская система. Например, число 1997 в этой системе имеет вид M C M X C V II .

В качестве цифр в римской системе счисления используются прописные буквы латинского алфавита:

M = 1000; D = 500; C = 100; L = 50; X = 10; V = 5; I = 1.

Число получается сложением значений цифр, если после большей цифры идет меньшая или такая же цифра, и вычитанием, если имеет место обратный порядок. Для приведенного выше примера:

1000 - 100 + 1000 - 10 + 100 + 5 + 1 + 1 = 1997

Значение цифры в римской системе не зависит от ее положения (позиции) в числе. Поэтому такая система счисления называется непозиционной.

Древнеславянская нумерация сходна с римской тем, что здесь для записи цифр также используют буквы алфавита. Например, число 333 в этой системе имеет вид Т Л Г , где Т = 300, Л = 30, Г = 3 (дополнительно над числом записывается также знак "титл", напоминающий перевернутую букву Z; титл - это признак числа). Славянская система счисления также является непозиционной.

Непозиционные системы практически непригодны для выполнения арифметических операций.

Примером позиционной системы счисления является привычная для нас десятичная система.

Рассмотрим произвольное число 83887. Это число можно записать как сумму значений отдельных его цифр: 80000 + 3000 + 800 + 80 + 7 . Как видно из этой записи, значение любой цифры в числе, в частности цифры 8, зависит от ее позиции.

Пусть число, записанное в произвольной позиционной системе счисления, имеет $n+1$ цифру в целой части и m цифр в своей дробной части:

$$a_n a_{n-1} a_{n-2} \ldots a_2 a_1 a_0, b_1 b_2 \ldots b_m ,$$

где a_i - i-ая цифра целой части, b_j - j-ая цифра дробной части числа.

Тогда значение этого числа можно представить в виде

$$a_n \cdot q^n + a_{n-1} \cdot q^{n-1} + \ldots + a_1 \cdot q^1 + a_0 \cdot q^0 + b_1 \cdot q^{-1} + b_2 \cdot q^{-2} + \ldots + b_m \cdot q^{-m} ,$$

где q - основание системы счисления.

Например, для десятичного числа ($q = 10$) имеем

$$83887{,}45 = 8 \cdot 10^4 + 3 \cdot 10^3 + 8 \cdot 10^2 + 8 \cdot 10^1 + 7 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2} .$$

В ЭВМ наиболее часто применяются двоичная, восьмеричная и шестнадцатеричная системы счисления (2, 8 и 16 с/с). При этом восьмеричная и шестнадцатеричная системы используются, как правило, для компактной записи двоичных чисел.

1. Двоичная система счисления.

Количество цифр, используемых для изображения числа в любой позиционной системе счисления, равно основанию этой системы. В 2 с/с ($q = 2$) для этой цели применяются цифры 0 и 1.

Рассмотрим произвольное двоичное число

$$11010{,}101 = 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} =$$

$$= 16 + 8 + 2 + \frac{1}{2} + \frac{1}{8} = 26\frac{5}{8} = 26{,}625 \quad (10 \text{ с/с}).$$

Представленная здесь схема разложения двоичного числа является одновременно схемой перевода из 2 с/с в 10 с/с ($2 \to 10$).

Рассмотрим теперь перевод $10 \to 2$. Такой перевод производится отдельно для целой и дробной частей числа.

Пример 1. $43{,}375_{10} = 101011{,}011_2$

43	| 2				
42	21	| 2			
1	20	10	| 2		
	1	10	5	| 2	
		0	4	2	| 2
			1	2	1
				0	

Деление выполняется до тех пор, пока не будет получено частное, меньшее делителя. После этого цифры частного и остатков записываются в обратном порядке. В данном случае получим 101011.

В самом деле, мы 5 раз разделили исходное число на 2. Следовательно, в нем 1 раз содержится 2^5. Кроме этого, полученные остатки указывают, что в числе содержатся дополнительно 0 раз 2^4, 1 раз 2^3, 0 раз 2^2, 1 раз 2^1 и 1 раз 2^0.

Тогда можно записать:

$$43 = 1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 ,$$

т.е. мы получили разложение числа по степеням основания 2.

Для перевода дробной части рассмотрим вначале произвольную десятичную дробь. Будем последовательно умножать на 10 исходное число и дробные части получаемых в процессе преобразования чисел до тех пор, пока дробная часть очередного числа не станет равной нулю.

$$\begin{array}{l} 0,\,|\,935\cdot 10 \\ 9,\,|\,35\cdot 10 \\ 3,\,|\,5\cdot 10 \\ 5,\,|\,0 \end{array}$$

Эта схема показывает, что в исходном числе содержится $9\cdot 10^{-1}$ (после первого умножения) $+ 3\cdot 10^{-2}$ (после второго умножения) $+ 5\cdot 10^{-3}$ (после третьего умножения).

Если вместо умножения на 10 мы будем умножать на число q, то получим перевод дробной части числа в систему счисления с основанием q. Для исходного числа, приведенного в примере 1, получим:

$$\begin{array}{l} 0,\,|\,375\cdot 2 \\ 0,\,|\,75\cdot 2 \\ 1,\,|\,5\cdot 2 \\ 1,\,|\,0 \end{array}$$

Следовательно, $0{,}375_{10} = 0{,}011_2$.

Конечная десятичная дробь не всегда образует конечную двоичную дробь. Например, для числа 0,4 имеем:

$$
\begin{array}{l|l}
0, & 4 \cdot 2 \\
0, & 8 \cdot 2 \\
1, & 6 \cdot 2 \\
1, & 2 \cdot 2 \\
0, & 4 \cdot 2 \\
0, & 8 \cdot 2 \\
1, & 6 \cdot 2 \\
\multicolumn{2}{c}{\ldots\ldots\ldots\ldots}
\end{array}
$$

$0,4_{10} = 0,011001100110\ldots_2 = 0,(0110)_2$.

Естественно, в этом случае полученное двоичное число округляют.

2. Восьмеричная система счисления.

Основанием системы является число 8. Для изображения произвольного числа используются 8 цифр: 0, 1, 2, 3, 4, 5, 6, 7.

Перевод $8 \to 10$:

$35174,6_8 = 3 \cdot 8^4 + 5 \cdot 8^3 + 1 \cdot 8^2 + 7 \cdot 8^1 + 4 \cdot 8^0 + 6 \cdot 8^{-1} = 3 \cdot 4096 + 5 \cdot 512 +$

$+1 \cdot 64 + 7 \cdot 8 + 4 + 6/8 = 12288 + 2560 + 64 + 56 + 4 + 3/4 = 14972,75_{10}$.

Пример 2. Перевод $10 \to 8$. Схема перевода такая же, как и для 2 с/с.

$$
\begin{array}{lll}
 & & 397,2_{10} = 615,15_8 \\
397 & \underline{\mid 8} & 0, \mid 2 \cdot 8 \\
\underline{32} & 49 \quad \underline{\mid 8} & 1, \mid 6 \cdot 8 \\
77 & \underline{48} \quad 6 & 4, \mid 8 \cdot 8 \\
\underline{72} & 1 & 6, \mid 4 \cdot 8 \\
5 & & 3, \mid 2 \cdot 8 \\
 & & 1, \mid 6 \cdot 8 \\
 & & \ldots\ldots\ldots\ldots
\end{array}
$$

$0,2_{10} = 0,146314631\ldots = 0,(1463)_8$

Полученная восьмеричная дробь числа (615,15) округлена до двух цифр.

Правило округления: чтобы округлить дробное число до m цифр, нужно к $(m+1)$-ой цифре добавить половину цены разряда для данной системы счисления, после чего отбросить все дробные цифры, начиная с $(m+1)$-ой. Для 8 с/с половина цены разряда равна 4, для 2 с/с - 1, для 16 с/с - 8.

В рассмотренном выше примере имеем (для $m = 2$):

$$
\begin{array}{rl}
 & 0,1463146\,3 \\
+ & \quad\;\; 4 \\
\hline
 & \underline{\mid 0,15\mid} 23146\,3
\end{array}
$$

Примечание. Здесь сложение выполнено в восьмеричной системе счисления.

Перевод $8 \to 2$. Для перевода восьмеричного числа в 2 с/с нужно каждую восьмеричную цифру записать в виде двоичной триады, т.е. трех двоичных цифр.

Восьмеричное число	Двоичное число	Двоичная триада
0	0	000
1	1	001
2	10	010
3	11	011
4	100	100
5	101	101
6	110	110
7	111	111

Например, $3763{,}24_8$ = 011 111 110 011 , 010 100 $_2$ = $11111110011{,}0101_2$ (отброшены незначащие нули).

Приведенное выше правило перевода $8 \to 2$ связано с тем, что $8 = 2^3$.

В самом деле,

$3763{,}24 = 3\cdot 8^3 + 7\cdot 8^2 + 6\cdot 8^1 + 3\cdot 8^0 + 2\cdot 8^{-1} + 4\cdot 8^{-2} = (0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0)\cdot 2^9 + (1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0)\cdot 2^6 + (1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0)\cdot 2^3 + (0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0)\cdot 2^0 + (0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0)\cdot 2^{-3} + (1\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0)\cdot 2^{-6} = 0\cdot 2^{11} + 1\cdot 2^{10} + 1\cdot 2^9 + 1\cdot 2^8 + 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4} + 0\cdot 2^{-5} + 0\cdot 2^{-6} =$

$= 011\ 111\ 110\ 011, 010\ 100_2$.

Для перевода $2 \to 8$ следует разделить двоичное число влево и вправо от запятой на триады, а затем заменить каждую триаду одной восьмеричной цифрой. Если первая триада в целой части или последняя триада в дробной части числа получаются неполными, то нужно дополнить их незначащими нулями.

Пример 3.

$1\,|\,011\,|\,110\,|\,111{,}101\,|\,101\,|\,01_2 = 001\,|\,011\,|\,110\,|\,111{,}101\,|\,101\,|\,010 = 1367{,}552_8$

В связи с тем, что перевод $10 \to 8$ или $8 \to 10$ выполняется быстрее, чем перевод $10 \to 2$ или $2 \to 10$, то перевод $10 \to 2$, как правило, производят по схеме $10 \to 8 \to 2$, а вместо $2 \to 10$ соответственно $2 \to 8 \to 10$.

3. Шестнадцатеричная система счисления.

Основанием системы является число 16. Для изображения произвольного числа нужно использовать 16 цифр. Так как цифр 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 недостаточно, то дополнительно применяют первые буквы латинского алфавита: A (цифра 10), B (цифра 11), C (цифра 12), D(цифра 13), E (цифра 14), F (цифра 15).

Пример 4. Перевод $16 \to 10$:

$A8B7{,}E_{16} = 10\cdot 16^3 + 8\cdot 16^2 + 11\cdot 16^1 + 7\cdot 16^0 + 14\cdot 16^{-1} = 10\cdot 4096 + 8\cdot 256 +$

$+ 11\cdot 16 + 7 + 14/16 = 43191{,}875_{10}$.

Пример 5. Перевод $10 \to 16$.

$7643{,}4_{10} = 1EDB{,}66_{16}$

```
7643 | 16                          0, | 4 · 16
64     477 | 16                   6, | 4 · 16
124    32    29 | 16              6, | 4 · 16
112    157   16   1               6, | 4 · 16
 123   144   14                   ……
 112    13
  11
```

Перевод $16 \to 2$. Так как $16 = 2^4$, то в этом случае каждая шестнадцатеричная цифра должна быть представлена двоичной тетрадой.

Шестнадцате-ричное число	Двоичное число	Двоичная тетрада	Шестнадцате-ричное число	Двоичное число	Двоичная тетрада
0	0	0000	8	1000	1000
1	1	0001	9	1001	1001
2	10	0010	A	1010	1010
3	11	0011	B	1011	1011
4	100	0100	C	1100	1100
5	101	0101	D	1101	1101
6	110	0110	E	1110	1110
7	111	0111	F	1111	1111

Пример 6.
4AD,B8 = 0100 1010 1101, 1011 1000 = 10010101101,10111.

Перевод $2 \to 16$. Исходное число разделяют влево и вправо от запятой на тетрады, а затем каждую тетраду записывают одной шестнадцатеричной цифрой.

Пример 7.
$11 \mid 1011 \mid 1110 \mid 0001 \mid ,0110 \mid 11_2 = 0011 \mid 1011 \mid 1110 \mid 0001 \mid ,0110 \mid 1100 = 3BE1,6C_{16}$

Для сокращения вычислений вместо $10 \to 2$ выполняют $10 \to 16 \to 2$, а вместо $2 \to 10$ выполняют $2 \to 16 \to 10$.

Примечание. Очевидно, что переводы из 16 с/с в 8 с/с и наоборот выполняют с применением 2 с/с как буферной. Например, перевод $16 \to 8$ производится по схеме $16 \to 2 \to 8$.

4. Сложение и вычитание в 2, 8 и 16 с/с.

Методика выполнения операций сложения и вычитания в этих системах аналогична выполнению соответствующих операций в 10 с/с. Ниже приведены примеры указанных операций.

Пример 8. Двоичная система счисления.

```
  1 0 1 1 0 1 1, 1 1 1      1 0 1 1 0 1 1, 1 1 1        Проверка вычитания
+ 1 0 1 1 0 1, 0 1 0      - 1 0 1 1 0 1, 0 1 0        1 0 1 1 0 1, 0 1 0
-----------------         -----------------         + 1 0 1 1 1 0, 1 0 1
1 0 0 0 1 0 0 1, 0 0 1      1 0 1 1 1 0, 1 0 1        -----------------
                                                      1 0 1 1 0 1 1, 1 1 1
```

Пример 9. Восьмеричная система счисления.

		Проверка вычитания
7 3 0 4 6, 3 5	7 3 0 4 6, 3 5	6 4 2 5 2, 6 7
+ 6 5 7 3, 4 6	- 6 5 7 3, 4 6	+ 6 5 7 3, 4 6
1 0 1 6 4 2, 0 3	6 4 2 5 2, 6 7	7 3 0 4 6, 3 5

Пример 10. Шестнадцатеричная система счисления.

		Проверка вычитания
8 A 7 B E, D 3	8 A 7 B E, D 3	7 B 6 C A, 6 7
+ F 0 E 4, 6 C	- F 0 E 4, 6 C	F 0 E 4, 6 C
9 9 8 A 3, 3 F	7 B 6 C A, 6 7	8 A 7 B E, D 3

Определенная трудность у начинающего программиста возникает при выполнении операции вычитания в 2, 8 или 16 с/с. Ссылка на то, что методика получения результата аналогична выполнению вычитания в 10 с/с, здесь не всегда помогает. Дело в том, что многие манипуляции, в том числе арифметические действия, мы выполняем автоматически, не задумываясь над последовательностью своих действий. Формализация же этих действий и составляет основу алгоритма решения поставленной задачи. В связи с этим обсудим в деталях порядок выполнения операции вычитания на конкретном примере.

Пример 11. 8 A 0 0 3 E 7 B - 1 9 D 5 6 4 A A = 7 0 2 A D 9 D 1

1)

$\bullet_{10}$
8 A 0 0 3 E 7 B
- 1 9 D 5 6 4 A A
. . . 9 D 1

Обозначим цифры уменьшаемого через a_i, вычитаемого - через b_i, разницы - через c_i (если в вычитаемом меньше цифр, чем в уменьшаемом, то вычитаемое нужно дополнить слева незначащими нулями). При нумерации цифр справа налево индекс i в данном примере изменяется от 0 до 7.

Если $a_i \geq b_i$, то результат получаем обычным образом. Для $i = 0$ имеем

$$(\text{B} - \text{A})_{16} = (11 - 10)_{10} = 1 .$$

Для $i = 1$ имеет место $a_1 < b_1$. Поэтому ищем ближайшую слева значащую цифру (это a_2), отнимаем от нее единицу и переносим эту единицу в первый разряд. То, что в a_2 изъята единица, обозначено точкой, а то, что это значение добавлено в первый разряд, записано над цифрой 7 ($10_{16} = 16_{10}$). Тогда

$a_1 = (10 + 7)_{16} = 17_{16} = 23_{10}$; $c_1 = (23 - 10)_{10} = 13_{10} = \text{D}_{16}$;

$c_2 = \text{D} - 4 = 9$ (от значения цифры E ранее была отнята 1).

2)

$\bullet\ \bullet$
$\bullet_{10}\ 10\ 10\ \bullet_{10}$
8 A 0 0 3 E 7 B
- 1 9 D 5 6 4 A A
7 0 2 A D 9 D 1

$a_3 < b_3$. Ближайшая слева значащая цифра – это $a_6 =$ A. Производим последовательный перенос единицы в a_5, a_4, a_3. Эти действия обозначены на схеме операции.

$$c_3 = (10 + 3 - 6)_{16} = (19 - 6)_{10} = 13_{10} = D_{16};$$

$$c_4 = (F - 5)_{16} = (15 - 5)_{10} = 10_{10} = A_{16};$$

$$c_5 = (F - D)_{16} = (15 - 13)_{10} = 2;$$

$$c_6 = 9 - 9 = 0;$$

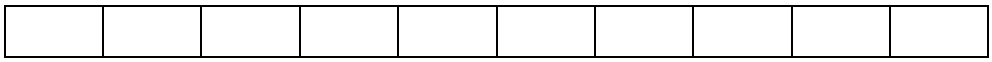
$$c_7 = 8 - 1 = 7.$$

Правильность выполнения операции вычитания целесообразно проверить сложением вычитаемого и разницы.

ЧИСЛА С ФИКСИРОВАННОЙ И ПЛАВАЮЩЕЙ ЗАПЯТОЙ

Каждая ЭВМ использует для представления чисел фиксированное количество двоичных разрядов. Их называют обычно разрядной сеткой ЭВМ.

Представим, что в условной ЭВМ (например, на калькуляторе) разрядная сетка содержит 10 десятичных разрядов:



Произвольное число в общем случае имеет целую и дробную части. Следовательно, в разрядной сетке нужно установить границу между этими частями.

Поставим разделяющую запятую, например, между шестым и седьмым разрядами. Тогда первые 6 разрядов сетки представляют целую часть числа, а последние 4 разряда - его дробную часть. Максимальное значение числа в этом случае равно 999999,9999; минимальное - 0,0001. Следовательно, при такой разрядной сетке обработка чисел может быть организована лишь в диапазоне 0,0001 .. 1000000, что явно недостаточно.

Если в разрядной сетке машины запятая, разделяющая целую и дробную части числа, фиксирована в заранее определенной позиции, то получаемые в этом случае числа называют числами с фиксированной запятой.

В ЭВМ, как правило, применяют один из двух способов представления чисел с фиксированной запятой:

1) запятая фиксирована перед старшим разрядом; в этом случае число имеет только дробную часть и не имеет целой части;

2) запятая фиксирована после младшего разряда; в этом случае число имеет только целую часть и не имеет дробной части.

Наиболее часто применяется второй способ.

Числа с плавающей запятой имеют следующую форму представления:

$$u \cdot 10^p,$$

где u - мантисса; p - порядок; 10 - основание системы счисления, записанное в этой же системе ($10_2 = 2$; $10_8 = 8$; $10_{16} = 16$). Это так называемая экспоненциальная (или показательная) форма записи числа.

Например, число 358,5 можно записать в виде

$$0{,}3585 \cdot 10^3 = 358{,}5 \cdot 10^0 = 3585{,}0 \cdot 10^{-1} = 0{,}003585 \cdot 10^5 .$$

Чтобы обеспечить единственность представления числа, на мантиссу накладывается следующее ограничение:

$$0{,}1 \le u < 1 \text{ (в данной с/с).}$$

Следовательно, мантисса - это дробь, в которой первая цифра должна быть значащей, т.е. не равной нулю. Такое представление числа с плавающей запятой называется нормализованным. В рассмотренном выше примере нормализованным является первый вариант представления заданного числа.

В разрядной сетке машины часть разрядов выделяется для мантиссы, а часть - для порядка числа.

Предположим, что в приведенной ранее разрядной сетке для мантиссы отведено 8 разрядов, а для порядка - 2 разряда (знак числа и знак порядка временно не рассматриваются). Тогда максимальное значение мантиссы составляет 0,99999999 , что примерно равно 1; максимальное значение порядка равно 99; число имеет максимальное значение $1 \cdot 10^{99}$, что достаточно для любых практических применений.

Сравним выполнение операции сложения для чисел с фиксированной запятой (целых чисел) и чисел с плавающей запятой.

Пусть $x_1 = 78535$, $x_2 = 416$. В формате целых чисел они имеют вид $x_1 =$ 0000078535, $x_2 =$ 0000000416. Их сумма получается по обычному правилу сложения:

```
  0 0 0 0 0 7 8 5 3 5
+ 0 0 0 0 0 0 0 4 1 6
_____________________
  0 0 0 0 0 7 8 9 5 1
```

В формате с плавающей запятой заданные числа имеют вид

$$x_1 = 0{,}78535 \cdot 10^5 , \quad x_2 = 0{,}416 \cdot 10^3 ,$$

или в машинном представлении $x_1 =$ 7853500005, $x_2 =$ 4160000003.

Непосредственно складывать мантиссы, если слагаемые имеют разные порядки, нельзя. В этом случае сложение должно выполняться в следующей последовательности:

1) Определяется разница порядков слагаемых $\Delta p = p_1 - p_2$; здесь имеем $\Delta p = 2$.

2) Если $\Delta p > 0$, то u_2 сдвигается вправо на Δp разрядов; если $\Delta p < 0$, сдвигу подвергается u_1; при $\Delta p = 0$ сдвиг не производится.

В данном примере после сдвига получим:

$$u_1 = 0{,}78535000; \quad u_2 = 0{,}00416000$$

3) Выполняется сложение мантисс

$$u = u_1 + u_2 = 0{,}78951000$$

4) Результату приписывается максимальный из порядков слагаемых

$$p = \max(p_1, p_2) = 5$$

В результате получаем

$$y = x_1 + x_2 = 7895100005$$

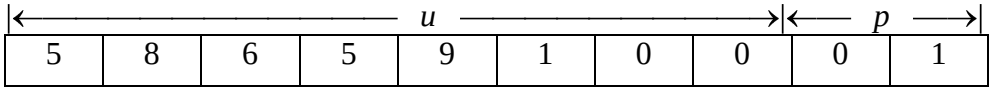
При сложении мантисс может иметь место случай $u > 1$ (например, для $x_1 = 0{,}8 \cdot 10^3$, $x_2 = 0{,}7 \cdot 10^3$) . Тогда получаемый результат нужно нормализовать, т.е. суммарную мантиссу сдвинуть на один разряд вправо, а к порядку добавить 1.

Из примера видно, что способ сложения чисел с плавающей запятой более сложный по сравнению со сложением целых чисел. Следовательно, для выполнения арифметических

операций над числами с плавающей запятой требуется больше машинного времени процессора, чем для выполнения таких же операций по отношению к целым числам. В связи с этим практически в любой ЭВМ числа представлены как в формате с фиксированной запятой, так и в формате с плавающей запятой. При программировании рекомендуется применять в первую очередь целые числа, если это возможно по условиям задачи. Обычно целочисленные переменные в программе определяют объекты, которые могут принимать только целые значения (количество предметов, порядковый номер предмета и т.п.). Числа с плавающей запятой отображают вещественные числа, которые в технических задачах определяют, как правило, результаты измерения каких-либо параметров (длина, вес, площадь, время и т.п.).

В заключение рассмотрим вопрос о происхождении названия "число с плавающей запятой".

Пусть в разрядной сетке условной машины содержится число



т.е. число $x = 0{,}586591 \cdot 10^1 = 5{,}86591$.

Если рассматривать его как число с фиксированной запятой, то разделяющая запятая должна быть поставлена после первого разряда ($p = 1$).

Для числа

5	8	6	5	9	1	0	0	0	2

запятая будет установлена после второго разряда ($x = 58{,}6591$; $p = 2$).

Для числа

5	8	6	5	9	1	0	0	0	3

запятая устанавливается после третьего разряда ($x = 586{,}591$) и т.д.

При этом создается впечатление, что запятая "плавает" по разрядной сетке при изменении порядка числа.

ПРЕДСТАВЛЕНИЕ ИНФОРМАЦИИ В ПЭВМ

Языки высокого уровня (Паскаль, Си, Бейсик и т.п.) не ориентированы на конкретный тип ЭВМ. Это означает, что программа, написанная, например, на языке Паскаль, должна решаться без изменений на ЭВМ любого типа, имеющей транслятор с данного языка. В связи с этим программиста практически не интересуют конкретные технические сведения о конструкции и особенностях функционирования отдельных устройств ЭВМ. Тем не менее для создания эффективной программы необходимо обладать некоторым минимальным объемом знаний о технических параметрах ЭВМ, на которой будет реализована эта программа, в частности о способах представления числовой и нечисловой информации.

В ПЭВМ, как и во многих машинах другого типа, минимальной единицей обрабатываемой информации является байт (byte). Байт состоит из 8 двоичных разрядов, или бит (bit, от слов BInary digiT - двоичная цифра). Биты нумеруются 0, 1, 2, 3, 4, 5, 6, 7. Биты 0..3 и 4..7 образуют два полубайта - левый и правый, представляемые как двоичные тетрады. При записи содержимого байта каждый полубайт обозначают одной шестнадцатеричной цифрой.

Возможные значения байта:

0000 0000 = 00
0000 0001 = 01
0000 0010 = 02
.....
1111 1111 = FF_{16} = 255_{10}.

Следовательно, байт может принимать 256 различных значений.

Соответствие между кодовыми комбинациями байта и символами, реализуемыми на ПЭВМ, отображается в кодовой таблице ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией). В этой таблице представлены латинские и русские буквы, цифры, знаки операций и др. Например, цифре 6 соответствует кодовая комбинация 00110110 (36_{16}), букве K - код 01001011 ($4B_{16}$) и т.д.

Каждый байт в памяти ЭВМ имеет свой номер (адрес). Адреса изменяются последовательно от 0 до некоторого максимального значения, определяемого объемом памяти ЭВМ. Объем памяти измеряют в килобайтах, мегабайтах, гигабайтах.

1 Кбайт = 2^{10} байт = 1024 байта;
1 Мбайт = 2^{10} Кбайт = 2^{20} байт;
1 Гбайт = 2^{10} Мбайт = 2^{30} байт.

Для ПЭВМ, имеющей объем памяти 1 Мбайт, максимальный адрес равен FFFFF (FFFFF = 100000_{16} - 1 = 16^5 - 1 = 2^{20} - 1).

Байты могут обрабатываться каждый отдельно или полями. Поле - это группа последовательных байтов. Длина поля равна количеству содержащихся в нем байтов. Адресом поля является адрес его крайнего левого байта. Некоторые поля имеют отдельные наименования: слово (поле длиной 2 байта), двойное слово (поле длиной 4 байта).

Все, что обрабатывает ЭВМ, обобщенно называют данными. К ним, в частности, относятся целые и вещественные числа, а также так называемые логические данные.

1. **Целые числа (числа с фиксированной запятой).**

ПЭВМ имеет несколько типов целых чисел, различающихся между собой количеством содержащихся в них разрядов. Здесь будет рассмотрен тип *integer*.

Для чисел типа *integer* отводится поле длиной 2 байта. Биты этого поля нумеруются в последовательности 0, 1, ... , 15. Нулевой бит, т.е. бит с нулевым порядковым номером содержит знак числа ("0" - это знак "+", "1" - знак "-").

Например, число 3104_{10} = $C20_{16}$ в формате *integer* имеет вид

0C20 = 0000 1100 0010 0000

Здесь 16 с/с используется для компактного изображения двоичного числа.

Примечание. Поскольку число в формате *integer* имеет размер 2 байта, то шестнадцатеричная запись числа, применяемая для его компактного представления, должна всегда содержать 4 цифры. С этой целью в приведенном выше примере к шестнадцатеричной записи числа добавлен незначащий нуль.

Отрицательные числа с фиксированной запятой представлены в так называемом дополнительном коде.

Дополнительный код отрицательного числа - это его дополнение до такого числа, которое в этой же системе счисления представлено единицей и столькими нулями, сколько цифр имеет исходное число.

Пример 1.

10 с/с	16 с/с	2 с/с
1 0 0 0 0	1 0 0 0 0	1 0 0 0 0
- 2 8 9 3	- A 5 4 C	- 1 0 1 0
8 1 0 7	5 A B 4	0 1 1 0

Существует более простое правило формирования дополнительного кода отрицательного числа: чтобы получить такой код, необходимо каждую цифру исходного отрицательного числа заменить ее дополнением до максимальной цифры в данной системе счисления, а затем добавить единицу к младшему разряду образовавшегося числа.

Для 10 с/с максимальная цифра - это 9, для 16 с/с - F, для 2 с/с - 1. Отметим, что для 2 с/с первый этап указанного преобразования сводится к инвертированию числа, т.е. замене нулей на единицы, а единиц - на нули.

Для числа -3104_{10} в формате *integer* имеем F3E0. Для числа $-58_{10} = -003A_{16}$ получим FFC6.

Дополнительный код позволяет заменить операцию вычитания операцией сложения с дополнительным кодом отрицательного числа.

Пример 2. Пусть нам требуется выполнить в 10 с/с операцию

```
  6 5 8 6 1
- 4 8 2 7 3
-----------
  1 7 5 8 8
```

Вместо вычитания числа 48273 выполним сложение с его дополнительным кодом:

```
  6 5 8 6 1
+ 5 1 7 2 7
-----------
|1|1 7 5 8 8
```

Обведенная единица переноса выходит за пределы разрядной сетки и теряется. Следовательно, в обоих случаях мы получили одинаковые результаты.

Использование дополнительного кода позволяет отказаться от установки в процессоре блока вычитания, что упрощает его конструкцию.

Максимальное значение числа формата *integer*:

x_{max} = 0111 1111 1111 1111 = 1000 0000 0000 0000 - 1 = 2^{15} - 1 = 32767

Минимальное значение определяется кодом 1000 0000 0000 0000, которому соответствует число -32768.

Рассмотрим формирование отрицательного минимума для формата *integer*.

8000_{16} = $1000\ 0000\ 0000\ 0000_2$

1111 1111 1111 1111 - доп.код числа -1

\+ 1000 0000 0000 0001 - доп.код числа -111 1111 1111 1111 = - (2^{15} - 1)

---

1 1000 0000 0000 0000

Единица переноса из старших складываемых разрядов отбрасывается.

Результат: $8000_{16} = -(2^{15} - 1) - 1 = -2^{15} = -32768$.

При анализе содержимого полей памяти ПЭВМ нередко возникает обратная задача: по дополнительному коду определить прямой код отрицательного числа. Здесь можно выполнить действия, обратные по отношению к описанным выше: вычесть единицу из младшего разряда, а затем заменить каждую цифру ее дополнением до максимальной цифры данной системы счисления. Однако проще воспользоваться следующим правилом: дополнительный код дополнительного кода есть прямой код числа. В этом случае исчезает необходимость выполнять операцию вычитания.

Пример 3 (для 16 с/с):

исходное число	- 5B9538B6
дополнительный код	A46AC74A
дополнительный код дополнительного кода	- 5B9538B6

2. Вещественные числа (числа с плавающей запятой).

ПЭВМ имеет несколько типов вещественных чисел, различающихся количеством разрядов, отведенных для мантиссы и порядка числа. Мы будем рассматривать тип *real*.

Тип *real* - это двоичное число, занимающее 6 байт памяти. Нумерация двоичных разрядов 0, 1, 2, ... , 47. Первые пять байтов занимает мантисса числа, шестой байт - характеристика. Нулевой бит отведен для знака числа. Отрицательные числа изображаются в прямом коде.

Характеристика - это преобразованный определенным образом порядок числа.

Порядок вещественного числа может быть положительным или отрицательным. Для знака порядка, как и для знака числа, необходимо отвести один разряд. В этом случае отрицательный порядок, представляющий собой целое однобайтное число, нужно изображать в дополнительном коде, а при обработке порядка выполнять преобразование из прямого кода в дополнительный и обратно. Чтобы ускорить процесс обработки числа, в старший разряд порядка добавляют единицу. Тогда получим

$$r = p + 1000\,0000_2 = p + 80_{16} = p + 128_{10},$$

где r - характеристика числа, p - его порядок.

Это приводит к смещению значения порядка на 128, в связи с чем характеристику называют также смещенным порядком.

Характеристика всегда положительная.

Если $-128 \le p < 0$, то $0 \le r < 128$;

если $p = 0$, то $r = 128$;

если $0 < p \le 127$, то $128 < r \le 255$.

Максимальной характеристике $r = 255$ соответствует порядок $p = 127$, минимальной характеристике $r = 0$ - порядок $p = -128$. При этом получаем соответственно максимальное и минимальное значения вещественного числа:

$$x_{\max} = 1 \cdot 2^{127} \approx 1.7 \cdot 10^{38}; \quad x_{\min} = 1 \cdot 2^{-128} \approx 2.95 \cdot 10^{-39}.$$

Примечание. Если в программе задать $x > 1.7 \cdot 10^{38}$, то при трансляции будет выведено на экран сообщение «Const out of range» (Константа вне допустимых границ), если задать $x < 2.95 \cdot 10^{-39}$, то переменной *x* будет присвоено нулевое значение.

В нормализованной двоичной мантиссе первая цифра всегда равна единице. Для экономии памяти мантисса числа формата *real* в процессоре перед записью ее в основную память изображается сдвинутой на один разряд влево. При этом первая цифра мантиссы, перешедшая в ее целую часть, не включается в код числа. При чтении вещественного числа из памяти в процессор его мантисса восстанавливается.

Пример 4.

$687{,}25_{10} = 2AF{,}4_{16} = 0{,}2AF4 \cdot 16^3 = 0,\ 0010\ 1010\ 1111\ 0100 \cdot 2^{12} =$

$= 0,\ 1010\ 1011\ 1101 \cdot 2^{10}$ (после нормализации мантиссы) =

= 0010 1011 1101 0000 0000 0000 0000 0000 0000 0000 1000 1010

(мантисса сдвинута на два разряда влево, ее первая единица отброшена, биты числа, включая знаковый, разделены на тетрады).

Здесь порядок $p = 10$, характеристика $r = 10 + 128 = A + 80_{16} = 8A_{16}$.

В формате *real* имеем 2B D0 00 00 00 8A .

Для отрицательного числа достаточно записать единицу в нулевой разряд, что эквивалентно добавлению значения 8 к первой цифре положительного числа. В этом случае для числа 687,25 получим AB D0 00 00 00 8A (первая цифра A = 2 + 8 = 0010 + 1000 = 1010).

3. Логические данные.

Логические переменные и логические операции относятся к области математики, которая называется алгеброй логики. В алгебре логики рассматриваются высказывания, в отношении которых имеет смысл говорить об их истинности или ложности. Например, "снег белый", "сегодня - пятница", "$x > 0$", "$a + b < z$" и т.д. Истинность высказывания может принимать одно из двух значений: 0 (высказывание ложное) или 1 (высказывание истинное). Алгебра логики широко применяется при проектировании и анализе работы устройств ЭВМ, поскольку элементы, входящие в состав таких устройств, являются бинарными (двоичными) и могут находиться лишь в одном из двух возможных состояний, которые обозначаются соответственно 0 и 1. В программировании операции алгебры логики используются при вычислении логических выражений.

Алгебра логики определяет 16 логических операций. Наиболее важными из них являются три: отрицание, логическое умножение и логическое сложение.

а) Отрицание (операция НЕ). Таблица операции:

$\overline{0} = 1$
$\overline{1} = 0$

Читается: "Не нуль есть единица".

б) Логическое умножение (конъюнкция, операция И). Таблица операции:

$0 \wedge 0 = 0$
$0 \wedge 1 = 0$
$1 \wedge 0 = 0$
$1 \wedge 1 = 1$

Читается: "Нуль и нуль есть нуль".

в) Логическое сложение (дизъюнкция, операция ИЛИ). Таблица операции:

$0 \vee 0 = 0$
$0 \vee 1 = 1$
$1 \vee 0 = 1$
$1 \vee 1 = 1$

Читается: "Нуль или нуль есть нуль".

Логические операции являются битовыми. Если их применяют для полей битов, то эти операции выполняются отдельно для каждой пары бит.

Пример 5. Пусть мы имеем два поля *X* и *Y* длиной 4 байта:

X = F570 1A8B ; Y = 37E4 90CD .

Здесь шестнадцатеричная запись используется только с целью компактности представления содержимого поля памяти.

Тогда

$\overline{X}$ = 0 A 8 F E 5 7 4

$X \wedge Y$ = 3 5 6 0 1 0 8 9

$X \vee Y$ = F 7 F 4 9 A C F

Примечание. Чтобы выполнить записанные выше операции, следует исходные числа (содержимое полей памяти) *X* и *Y* представить в виде последовательности двоичных тетрад, применить заданные логические операции отдельно для каждой пары бит, а затем каждую тетраду полученного числа записать одной шестнадцатеричной цифрой.

ЛЕКСЕМЫ И РАЗДЕЛИТЕЛИ

Программа на Паскале состоит из лексем и разделителей. Лексема - это минимальная значимая единица языка, состоящая из одного или нескольких последовательных символов.

Лексемами являются следующие элементы:
- специальные символы;
- зарезервированные слова;
- числа;
- строки символов;
- идентификаторы;
- метки.

Разделителями в Паскале являются:
- пробел;
- конец строки (разделитель строк);
- примечание (комментарий).

Пробел - это пустое место в тексте на бумаге или на экране дисплея.

Представим, что мы записываем в память ЭВМ произвольный текст. Слова в тексте разделены пробелами. Память ЭВМ - это линейная последовательность байтов. В каждом

байте содержится один символ текста. Пробел, как и буквы в словах, отображается вполне конкретным содержимым байта.

Соответствие между символом и его кодом представлено в кодовых таблицах. Для ПЭВМ обычно используется таблица ASCII (American Standard Code for Information Interchange). В этой таблице записаны 256 символов с порядковыми номерами 0 .. 255 (цифры, прописные и строчные буквы, знаки операций, знаки препинания и др.). Порядковый номер символа в таблице ASCII - это код данного символа.

Первые 32 символа ASCII - управляющие символы, которые используются при обращении к внешним устройствам. Эти символы имеют названия, но не имеют графического обозначения. Обычно в программе их задают порядковым номером в таблице ASCII. Например, #7, #12 (символ "#" - это знак номера). Порядковым номером можно задавать и другие символы. Например, #32 (пробел), #158 (буква "Ю"). Содержимое байта для пробела будет иметь значение $32_{10} = 20_{16} = 00100000_2$, для буквы "Ю" – $158_{10} = 9E_{16} = 10011110_2$.

Текст "Programming in Pascal" будет изображен следующей последовательностью байтов:

50726F6772616D6D696E6720696E2050617363616C

Память ЭВМ можно представить в виде длинной ленты, разделенной на ячейки (байты). Такая лента представляет собой одномерный объект. В отличие от плоскости (листа бумаги), являющейся двумерным объектом, на ленте нельзя записать строки текста друг под другом. Следовательно, необходимо использовать какой-либо символ для обозначения строк записанного текста. Таким символом является "Конец строки", используемый для разделения текста в памяти ЭВМ на отдельные строки, длина которых может быть различной. Этому символу соответствуют два последовательных байта с кодами #13 и #10.

Комментарий в программе определяется парой символов, обозначающих его начало и окончание. В качестве таких символов могут использоваться "{" и "}" или "(*" и "*)". Текст комментария, заключенный в фигурные скобки, может содержать любые символы ASCII, кроме символов начала и конца комментария. Символы "(*" и "*)" имеют более высокий приоритет; между ними могут быть символы "{" и "}". Комментарий не влияет на работу программы и предназначен для лучшего понимания программы человеком. При трансляции все комментарии удаляются из программы.

Лексемы в Паскаль-программе обязательно разделяются одним или несколькими символами-разделителями. Обычно это пробелы.

При конструировании лексем используются буквы и цифры. К буквам относятся прописные и строчные латинские буквы (но не буквы национального алфавита, в частности русского): A B C D ... X Y Z a b c d ... x y z . Цифры: 0 1 2 3 4 5 6 7 8 9 .

1. Специальные символы (их количество равно 27):

+ - * / . , : ; ' ^ = <> > < >= <= ( ) [ ] { } := .. @ $ #

Примечание. Символом "<>" в Паскале обозначается операция отношения "не равно" ("≠"), символом ">=" - операция "больше или равно" ("≥"), символом "<=" - операция "меньше или равно" ("≤"). Символ «^» (тильда) в дальнейшем используется для обозначения динамических переменных.

2. Зарезервированные слова.

Эти слова имеют четко установленный смысл в Паскаль-программе. Примеры таких слов*: Begin, End, If, And, Array, For* .

Зарезервированные слова нельзя применять в программе для обозначения переменных, массивов и других объектов.

Прописные и строчные буквы в зарезервированных словах эквивалентны.

Например, *BEGIN ≡ Begin ≡ begin.*

Примечание. В программах, приводимых в учебном пособии, зарезервированные слова выделены полужирным шрифтом.

3. Числа. В Паскаль-программе используются целые десятичные, целые шестнадцатеричные и вещественные десятичные числа.

Целые десятичные числа записываются обычным образом и должны находиться в диапазоне от -2 477 483 648 до 2 147 483 647 (от -2^{31} до $2^{31}-1$).

Пример 1. 0 21 -456 3897653 -987321123 .

Для обозначения целых шестнадцатеричных чисел используется префикс $, который ставится перед числом. Эти числа определены в диапазоне от $00000000 до $FFFFFFFF. В качестве шестнадцатеричных цифр могут использоваться как прописные, так и строчные буквы латинского алфавита, но предпочтительно все же использовать прописные буквы.

Пример 2. $0 $A5F $E45D07B9 $ab7f.

Шестнадцатеричные числа применяются главным образом для указания содержимого поля памяти или задания адреса переменной.

Вещественные числа могут быть представлены в двух различных формах записи: обычной и показательной. В обычной форме число записывается в виде целой и дробной частей, разделенных точкой; в показательной - в виде мантиссы и порядка с основанием 10, при этом в качестве признака основания ставится прописная буква "E" или строчная буква “e”.

Пример 3. 0.6 -32.648 6.0E-1 0.6E0 0.06E1 0.06E+1 -3.2648E1 -5.6e-12.

4. Строка символов - это последовательность символов таблицы ASCII, заключенная в апострофы. Апостроф определяет границы строки.

Если внутри строки нужно поставить апостроф, то он ставится дважды.

Пример 4.

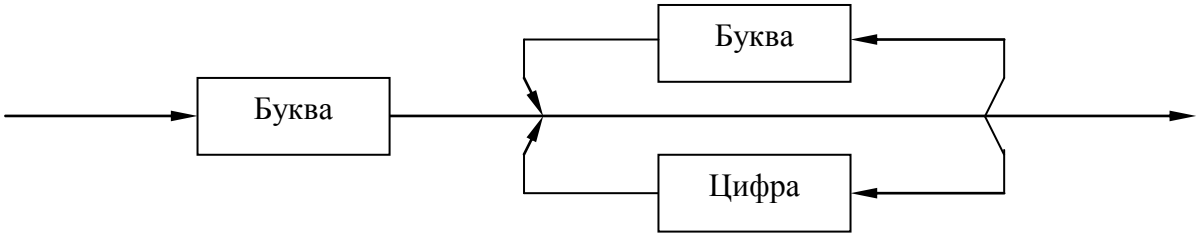
'A'; 'a+b=c'; 'This string has 30 characters'; ' Символ '' - это апостроф'

В строке прописные и строчные буквы считаются различными, так как они имеют различные номера в таблице ASCII. Поэтому

'PASCAL' ≠ 'Pascal'; 'ПРОГРАММА' ≠ 'программа'

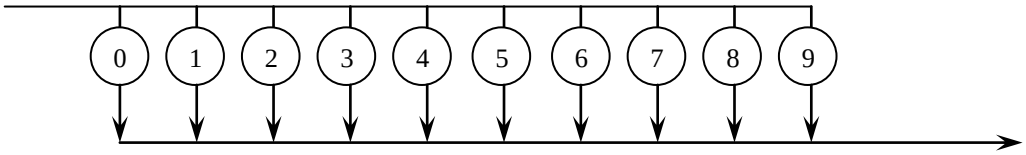
5. Идентификатор (имя). Это последовательность букв и цифр, начинающаяся с буквы.

Конструкцию грамматических объектов алгоритмического языка можно наглядно и точно изобразить на синтаксической диаграмме, которая в данном случае имеет вид:



В прямоугольнике синтаксической диаграммы находится имя, указывающее на другую диаграмму, в которой это имя определено.

Например, для цифры имеем



В кружки или в овалы помещают терминальные символы, т.е. символы, которые в грамматике языка не могут быть описаны элементами более низкого уровня. Это буквы, цифры, специальные символы, зарезервированные слова. При этом буквами в Паскале считаются лишь буквы латинского алфавита, а также символ подчеркивания.

Примеры идентификаторов:

X A8 alpha Massiv z52d9 eps Res_52_a ___75

Прописные и строчные буквы в идентификаторах считаются эквивалентными. Поэтому PASCAL, Pascal, pascal - это один и тот же идентификатор.

Длина идентификатора в Турбо Паскале не ограничивается, но значимыми считаются лишь первые 63 символа.

6. Метка.

Метка в Паскале представляет собой целое без знака и используется для маркировки операторов программы. Метка может принимать значения от 0 до 9999. В Турбо Паскале в качестве метки можно применять также идентификатор. Внешним признаком метки в программе является двоеточие, которое записывается перед маркируемым оператором.

ПРОСТЫЕ ТИПЫ ДАННЫХ

Данные - это общее понятие для всего того, с чем оперирует ЭВМ.

В машине все данные представлены как последовательность двоичных цифр и могут иметь различный смысл в зависимости от того, как они обрабатываются.

Для примера рассмотрим в ПЭВМ два поля памяти *A1* и *A2* длиной 4 байта:

Содержимое полей *A1* и *A2* - это последовательность двоичных цифр 0 и 1, которая ничего не означает, если не применить к ней конкретную машинную операцию.

Четырехбайтное поле в ПЭВМ может интерпретироваться как целое число длиной 4 байта (тип *longint*, условно формат *L*), как число с плавающей запятой длиной 4 байта (тип *single,* условно формат *S*), как строка символов, как битовая последовательность и др. Если для полей *A1* и *A2* мы применим операцию сложения для формата *L* “(+) L A1 A2” , то тогда содержимое полей *A1* и *A2* будет рассматриваться как целые двоичные числа типа *longint.*

Для операции “(+) S A1 A2” содержимое полей *A1* и *A2* интерпретируется как числа с плавающей запятой типа *single.*

При использовании логической операции “(\/) A1 A2” содержимое этих же полей памяти интерпретируется как битовые последовательности длиной 32 бита.

Пусть содержимое *A1* в шестнадцатеричной записи имеет вид 50 6A 37 B1. Рассмотрим его интерпретацию при обработке различными машинными операциями.

а) Тип *longint.*

$$5\cdot16^7 + 0\cdot16^6 + 6\cdot16^5 + 10\cdot16^4 + 3\cdot16^3 + 7\cdot16^2 + 11\cdot16^1 + 1\cdot16^0 = 1\ 349\ 088\ 353.$$

б) Тип *single* (в предположении, что подобно типу *real* первые три байта поля A1 - мантисса, а четвертый байт - характеристика вещественного числа).

$$r = \text{B1};\quad p = r - 80_{16} = \text{B1} - 80 = 31_{16} = 49_{10}\ 0{,}101\ 0000\ 0110\ 1010\ 0011\ 0111 \cdot 2^{49} =$$

$$0{,}1101\ 0000\ 0110\ 1010\ 0011\ 0111 \cdot 2^{49} =$$

$$= 0{,}\text{D06A37} \cdot 2^{49} = \text{D06A37} \cdot 2^{25} = 13\ 658\ 679 \cdot 33\ 554\ 432 = 458\ 309\ 215\ 275\ 328$$

в) Битовая последовательность: 0101 0000 0110 1010 0011 0111 1011 0001

г) Строка символов (по таблице ASCII): 'Pj7Б' .

При программировании на Ассемблере, который фактически представляет собой условную запись машинного языка, программист может по-разному использовать одно и то же поле памяти. Эта свобода действий одновременно таит в себе опасность неверной трактовки содержимого поля памяти (например, целое число может ошибочно обрабатываться как число с плавающей запятой). Ограничение такой свободы и возложено на тип данных, основным назначением которого является контроль корректности использования переменных.

Языки высокого уровня, в том числе и Паскаль, позволяют абстрагироваться от деталей конкретного представления данных за счет введения концепции типов данных.

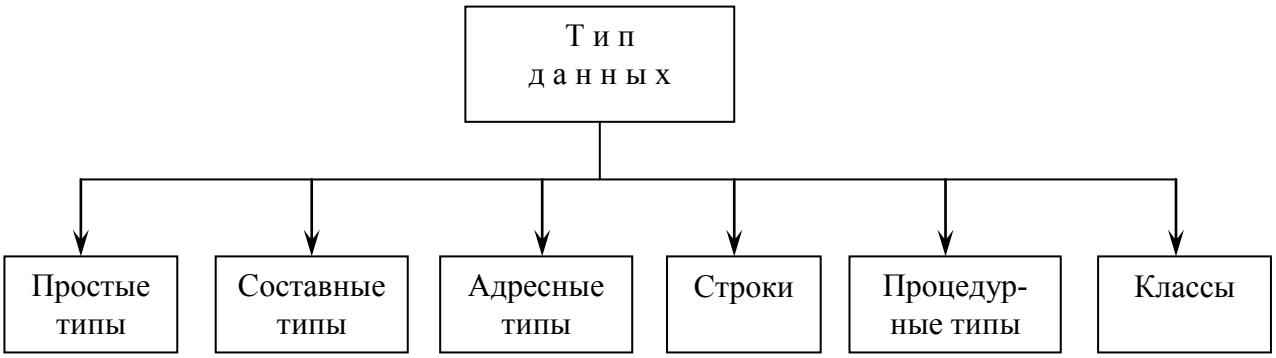
Тип данных определяет:

- множество значений, которые имеет право принимать переменная заданного типа;
- множество операций, которые допустимы при обработке этой переменной;
- формат внутреннего представления данных в памяти ЭВМ, в том числе объем памяти, выделяемой для размещения переменной.

С каждой переменной в программе может быть сопоставлен только один тип.

Классификация типов данных в Турбо Паскале представлена на приведенной ниже схеме.

В учебном пособии будут рассмотрены все перечисленные здесь типы данных, кроме классов, которые являются основой в объектно-ориентированном программировании.



Классификация простых типов данных представлена на следующей схеме. Здесь термин "ординальный" означает "порядковый" (ordinal - порядковое числительное).

Ординальный тип описывает конечное и упорядоченное множество значений, для каждого элемента которого однозначно определяется предыдущий и следующий за ним элементы. В частности, последовательность целых чисел является ординальным типом данных. Например, для элемента 5 существует единственное следующее значение (6) и единственное предыдущее значение (4). Символы алфавита также представляют собой ординальный тип данных. Например, в латинском алфавите для символа "M" следующим является символ "N", а предыдущим - "L". В то же время вещественные числа нельзя отнести к ординальному типу. Так, для числа 15.8 существует бесконечное множество следующих и предыдущих значений (15.81, 15.801, 15.8001 и т.д.).

Каждый ординальный тип имеет минимальное и максимальное значения. Для всех значений, кроме минимального, существует предшествующее значение; для всех значений, кроме максимального, существует последующее значение.

Предописанные функции *succ, pred* и *ord* воспринимают аргументы любого из ординальных типов.

Функция *succ*(*x*) (*succeed* - следующий) дает следующее после *x* ординальное значение; функция *pred*(*x*) (*predecessed* - предшествующий) - предшествующее значение; функция *ord*(*x*) дает ординальный (порядковый) номер для значения *x* .

Значения переменных ординального типа однозначно отображаются на последовательность порядковых номеров 0, 1, 2,... Исключение сделано только для целых чисел, которые отображаются сами на себя.

Пример 1.

```
succ(8) = 9 ;      succ('d') = 'e' ;
pred(8) = 7 ;      pred('d') = 'c' ;
ord(' ') = 32;      ord('Ю') = 158 ;
ord(8) = 8 ;         ord(-8) = -8 .
```

Для всех ординальных типов существуют операции отношения "=", "<>", "<", "<=", ">", ">=". При этом предполагается, что оба операнда в операции отношения имеют одинаковый тип. Отношение определяется с помощью порядковых номеров, присущих операндам. Например, проверка истинности отношения 'A' > 'Z' заменяется проверкой отношения *ord*('*A*') > *ord*('*Z*').

Термин "предописанный" (или "предопределенный"), в отличие от термина "зарезервированный", означает, что в принципе предописанные слова можно переназначать в программе, например, использовать их как имена переменных. Однако такое переназначение делать не рекомендуется, поскольку это ухудшает понимание программы.

1. Логический тип (тип ***boolean*****).**

Логическое значение - это одно из двух значений истинности, которые обозначают предописанными именами *false* и *true* (*false* эквивалентно значению 0, *true* - значению 1 в алгебре логики; использование имен *false* и *true* вызвано стремлением отличать в программе логические значения от числовых значений 0 и 1).

Логический тип определен так, что *false* < *true*, что соответствует машинному представлению логических констант *false* и *true*.

Операции отношения всегда дают логический результат. Например, отношение $x+1<y$ при $x = 0, y = 0$ дает *false*; при $x = 0, y = 10$ - значение *true*.

Для переменных логического типа в Турбо Паскале определены четыре логические операции:

1) отрицание ***not;***
2) логическое умножение ***and;***
3) логическое сложение ***or;***
4) исключающее ИЛИ (отрицание равнозначности, сложение по модулю 2) ***xor.***

Исключающее ИЛИ определяется следующей таблицей операций:

$0 \oplus 0 = 0$
$0 \oplus 1 = 1$
$1 \oplus 0 = 1$
$1 \oplus 1 = 0$

Как было ранее указано в разделе «Представление информации в ПЭВМ», в алгебре логики основными операциями являются отрицание ***not,*** логическое умножение ***and*** и логическое сложение ***or***. Все остальные операции могут быть выражены через эти базовые операции. Например, *a* ***xor*** *b* эквивалентно

(a ***or*** *b)* ***and*** *(****not*** *a* ***or*** ***not*** *b).*

Обозначения в символах алгебры логики:

$$a \oplus b = (a \vee b) \wedge (\bar{a} \vee \bar{b}) .$$

Справедливо также выражение

$$a \oplus b = (\bar{a} \wedge b) \vee (a \wedge \bar{b}) .$$

Примечание.

Если переменные *a, b, c* имеют тип boolean, то оператор

```
c:=a xor b
```

эквивалентен оператору

```
c:=a<>b.
```

В самом деле, из таблицы операций видно, что если переменные *a* и *b* имеют одинаковые значения, то переменная *c* получает значение 0, в протвном случае – значение 1.

Пример логического выражения:

$$((x + 1 < y)\ \textbf{\textit{and not}}\ (x > 5))\ \textbf{\textit{or}}\ ((y > 0)\ \textbf{\textit{and}}\ \textit{true}) .$$

Вычислим значение этого выражения при $x = 5, y = 5$:

$((6 < 1)\ \textbf{\textit{and not}}\ (5 > 5))\ \textbf{\textit{or}}\ ((5 > 0)\ \textbf{\textit{and}}\ \textit{true}) \Rightarrow (\ \textit{false}\ \textbf{\textit{and not}}\ \textit{false}\ )\ \textbf{\textit{or}}$
$(\ \textit{true}\ \textbf{\textit{and}}\ \textit{true}\ ) \Rightarrow (\ \textit{false}\ \textbf{\textit{and}}\ \textit{true}\ )\ \textbf{\textit{or}}\ (\ \textit{true}\ \textbf{\textit{and}}\ \textit{true}\ ) \Rightarrow \textit{false}\ \textbf{\textit{or}}\ \textit{true} \Rightarrow \textit{true}$.

В символике алгебры логики:

$$(0 \wedge \bar{0}) \vee (1 \wedge 1) = (0 \wedge 1) \vee (1 \wedge 1) = 0 \vee 1 = 1 .$$

Для переменной типа *boolean* отводится один байт памяти. Машинное представление значения *true* имеет вид 00000001, значения *false* - 00000000, т.е. *ord*(*false)* = 0, *ord*(*true*) =1.

Существуют несколько предописанных логических функций, т.е. функций, которые дают логический результат. В частности, при обработке целочисленных переменных применяется функция *odd*(*x*), которая дает значение *true*, если целое *x* - нечетное, и значение *false*, если *x* - четное. Другие логические функции будут рассмотрены позже.

2. Целые типы данных.

Значениями переменных целого типа являются элементы ограниченного подмножества целых чисел.

В Турбо Паскале имеется пять целочисленных типов данных: *shortint, byte, integer, word* и *longint.*

Тип *shortint* (*short integer* – короткое целое) обозначает целочисленную переменную длиной один байт со знаком. Биты однобайтного поля памяти нумеруются в последовательности 0, 1, 2, 3, 4, 5, 6, 7. Нулевой бит – это знак числа: 0 – «+», 1 – «-». Следовательно, значение числа определяется семью двоичными цифрами. Число типа *shortint* изменяется в пределах -128 .. 127 (минимальное значение $10000000_2 = -128_{10}$, см. раздел «Представление информации в ПЭВМ»).

Тип *byte* – это короткое целое длиной один байт без знака (все восемь бит - двоичные цифры), пределы изменения 0 .. 255.

Тип *integer* определяет целое число длиной два байта со знаком. Пределы изменения – от –32768 до 32767. Этому типу соответствует предописанная константа *MaxInt,* равная максимальному значению числа: $MaxInt = 2^{15} - 1 = 32767$.

Тип *word* – это целое длиной два байта без знака, пределы изменения 0 .. 65535.

Тип *longint* - длинное целое размером 4 байта со знаком, пределы изменения

- 2 147 483 648 .. 2 147 483 647.

Для типа *longint* предопределена константа *MaxLongint*, равная $2^{31} - 1 = 2147483647$.

Для целых значений допустимы следующие арифметические операции:

+ сложение;
- вычитание;
* умножение;
div деление нацело (целая часть от деления);
mod остаток от деления.

Пример 2.
19 ***div*** 5 = 3; 19 ***mod*** 5 = 4
Очевидно, что a ***mod*** $b = a - (a$ ***div*** $b) \cdot b$

В арифметическом выражении не могут стоять рядом два знака операции. Например, нельзя писать *a* * -*b*. Здесь должно быть *a* * (-*b*).

Старшинство операций:
1) выражения в скобках;
2) *, ***div, mod*** (операции типа умножения);
3) +, - (операции типа сложения).
Операции одинакового старшинства выполняются слева направо.

Пример 3.

5 + 3 * 7 ***div*** 4 - 17 ***mod*** 3 = 5 + 5 - 2 = 8

Примечание. На машинном уровне для целочисленных переменных может быть использована лишь одна операция деления, но ее результат записывается в два различных регистра (регистр – это быстродействующая ячейка памяти): в один регистр – частное, во второй – остаток от деления. Тогда в Паскаль-программе содержимое первого регистра воспринимается как результат операции ***div*** , а второго – как результат операции ***mod.***

Операции возведения в степень в Паскале нет. Для целых показателей степени эта операция может быть заменена многократным умножением. Более эффективный способ программной реализации операции возведения в степень рассмотрен в разделе «Вычисление степенной функции».

Целый результат дают следующие предописанные функции:
1) *abs*(*i*) - абсолютное значение целого аргумента *i*;
2) *sqr*(*i*) - квадрат значения целого аргумента *i*;
3) *trunc*(*R*) - целая часть вещественного значения *R*;
4) *round*(*R*) - целое значение, ближайшее к вещественному значению *R*.

Пример 4.

```
trunc(3.3) = 3;      round(3.3) = 3;
trunc(3.5) = 3;      round(3.5) = 4;
trunc(3.8) = 3;      round(3.8) = 4;
trunc(-3.3) = -3;    round(-3.3) = -3;
trunc(-3.8) = -3;    round(-3.8) = -4.
```

Следовательно,

$$trunc(x) = sign(x) \cdot trunc(abs(x)); \quad round(x) = sign(x) \cdot round(abs(x)),$$

где $sign(x) = 1$ при $x > 0$, $sign(x) = 0$ при $x = 0$ и $sign(x) = -1$ при $x < 0$.

Если i - целое, то $succ(i) = i + 1$; $pred(i) = i - 1$; $ord(i) = i$.

Целочисленным переменным можно присваивать значения не только десятичных, но и шестнадцатеричных констант. Отрицательные значения можно задавать, используя знак "-" или дополнительный код числа. В этом смысле приведенные ниже операторы для переменных i, k и l эквивалентны.

```
i:=-$5D;          i:=$A3;
k:=-$7C0F;        k:=$83F1;
l:=-$2BF01400;    l:=$C40FEC00 .
```

Целочисленные переменные могут рассматриваться как битовые последовательности. В связи с этим к ним применимы следующие операции:

Shl k (Shift Left) – сдвиг влево на k разрядов;
Shr k (Shift Right) – сдвиг вправо на k разрядов;
Not – отрицание;
And – логическое умножение;
Or – логическое сложение;
Xor – исключающее ИЛИ.

При сдвиге влево разряды, выходящие за пределы разрядной сетки, теряются, а справа в число добавляется k нулевых бит. Аналогичная работа выполняется при сдвиге вправо.

Для положительных чисел операции «**shl** k» и «**shr** k» эквивалентны соответственно умножению и делению на 2^k .

Пример 5

```
Var   m,n,p : integer;
.................
  p:=100;  m:=p shl 2;  n:=p shr 2;
```

Получим: `m = 400;  n = 25.`

В двоичном представлении:

```
p = 0000 0000 0110 0100;
m = 0000 0001 1001 0000;   n = 0000 0000 0001 1001.
```

При компиляции программы выражения типа $p*2^k$ и p **div** 2^k автоматически заменяются операциями сдвига.

Логические операции **not, and, or, xor** выполняются поразрядно (см. пример 5 в разделе «Представление информации в ПЭВМ»).

Рассмотрим теперь возможные способы определенич четности целочисленных переменных.

Вполне очевидно что целочисленная переменная k четная, если

$$k \textbf{ mod } 2 = 0,$$

и нечетная, если

$$k \textbf{ mod } 2 = 1.$$

В то же время несколько выше было указано, что для определения четности может быть использована логическая функция odd(k).

Запишем последовательно в двоичном представлении числа 5, 6, 7, 8, 9:

```
0101
0110
0111
1000
1001
```

Из этой записи видно, что четность двоичного числа определяется значением последнего разряда (0 – четное, 1 – нечетное число). Следовательно, функция odd(k) анализирует лишь последний разряд числа. Это эквивалентно вычислению следующего выражения:

$$\text{odd(k)} \equiv \text{k } \textbf{and } 1 \equiv \text{k } \textbf{and } 00000001$$

Операция **and** – это логическая операция, **mod** – арифметическая, которая выполняется на порядок дольше по сравнению с логической операцией. Следовательно, использование функции odd(k) при определении четности повышает быстродействие программы (микрооптимизация программы).

3. Символьный тип данных (тип ***char*).** Обозначение *char* - это сокращение слова character (символ, 'kæriktə).

Значениями переменных символьного типа являются элементы конечного и упорядоченного множества символов. Это значение обозначается одним символом, заключенным в апострофы.

Пример 6. 'A', 'a', '8', '''' (апостроф как символ пишется дважды).

Вне зависимости от реализации, т.е. от конструкции конкретного транслятора, для символьного типа справедливы следующие допущения.

1) Десятичные цифры от '0' до '9' упорядочены в соответствии с их значениями и записаны одна за другой. Следовательно,

$$succ('5') = '6';\ pred('5') = '4'.$$

2) Имеются все прописные буквы латинского алфавита от 'A' до 'Z'. Это множество упорядочено по алфавиту, но не обязательно связно. Следовательно, в любой реализации должно выполняться 'I' < 'J', но может не выполняться *succ*('I') = 'J'.

3) Могут быть строчные буквы латинского алфавита от 'a' до 'z'. Если это так, то это множество букв упорядочено по алфавиту, но не обязательно связно.

В таблице ASCII заданы как большие, так и малые латинские буквы, причем их последовательность непрерывная.

Для символьного типа определены две взаимно обратные функции преобразования *ord* и *chr:*

$k = ord(ch)$ - порядковый номер символа *ch*;

$ch = chr(k)$ - символ с порядковым номером *k*.

Очевидно, что

$$chr\,(\,ord\,(\,ch\,)\,) = ch\,; \qquad ord\,(\,chr\,(\,k\,)\,) = k\,.$$

В последнем случае должно быть $k = 0\ ..\ 255$.

Рассмотрим содержательный смысл функций *ord* и *chr* на машинном уровне. Как уже отмечалось, внутреннее представление символа в байте памяти - это его порядковый номер по таблице ASCII. Например, для буквы "Ю" имеем #158, тогда содержимое байта равно $9E_{16} = 10011110_2$.

Примечание. «#» - это порядковый номер символа в таблице ASCII.

Если байт, содержащий значение 10011110, используется в программе при выводе результатов как символ, то печатается буква "Ю"; если этот байт используется как число, то печатается значение 158. Следовательно, функции *ord* и *chr* никакого преобразования со-

держимого байта не производят; они лишь разрешают программе по-разному трактовать это содержимое (как символ или как число).

Для символьного типа определены все операции отношения.

Считается, что *ch1* < *ch2*, если *ord*(*ch1*) < *ord*(*ch2*), где *ch1, ch2* - переменные символьного типа.

4. Вещественные типы.

Вещественные числа - это числа с плавающей запятой. Широкое их использование характерно для инженерно-технических задач. Тем не менее процессоры 8086, 8088 и другие, на которых были построены первые ПЭВМ малой производительности, не имеют в своем составе операций над числами с плавающей запятой. Такие операции выполняются программным путем. Это означает, что транслятор включает в программу пользователя подпрограммы арифметических операций над числами с плавающей запятой. Тогда вместо выполнения, например, одной машинной команды сложения производится обращение к подпрограмме и выполняется целая группа машинных команд.

Для повышения производительности компьютера при обработке вещественных чисел к нему добавляют сопроцессор. Это небольшая плата, на которой аппаратно реализованы операции с плавающей запятой. При этом скорость выполнения таких операций увеличивается в несколько раз.

При отсутствии сопроцессора реализуется только один вещественный тип - тип *real;* при наличии сопроцессора реализуются также типы *single, double, extended, comp*. Вещественные типы отличаются друг от друга количеством разрядов, отводимых для представления мантиссы и порядка. В частности, переменная типа *single* имеет длину 4 байта, из них три байта - мантисса и один байт - характеристика; для переменной типа *double* отводится 8 байт памяти и т.д. В дальнейшем при разработке программ в качестве вещественного типа будет рассматриваться только тип *real.* Информация о других вещественных типах представлена в разделе «Использование сопроцессора».

Значение числа типа *real* в десятичном представлении может изменяться в диапазоне от $2{,}95 \cdot 10^{-39}$ до $1{,}7 \cdot 10^{38}$. Размер мантиссы такого числа обеспечивает получение 11 – 12 значащих десятичных цифр.

Для вещественных типов определены четыре арифметические операции:

+ сложение ;
- вычитание ;
* умножение ;
/ деление .

Результатом операций "+", "-", "*" является вещественное значение, если хотя бы один из операндов имеет вещественный тип. Операция "/" дает вещественное значение и в том случае, когда оба ее операнда относятся к целочисленным типам.

Например, 12 / 5 = 2.4 , в то время как 12 *div* 5 = 2 .

Стандартные функции *abs*(*x*) и *sqr*(*x*) дают вещественный результат, если их аргумент *x* имеет тип *real*. Вне зависимости от типа аргумента следующие стандартные функции всегда дают вещественный результат: *sin*(*x*), *cos*(*x*), *ln*(*x*), *exp*(*x*), *arctan*(*x*), *sqrt*(*x*) (корень квадратный).

Вещественный результат при вещественном аргументе дают также функции:

Int(*x*) - целая часть вещественного значения *x;*

Frac(*x*) - дробная часть вещественного значения *x* (*Frac*(*x*) = *x* - *Int*(*x*)).

Отсутствующие в Турбо Паскале функции *arcsin*(*x*), *arccos*(*x*), *lg*(*x*) могут быть определены следующим образом:

$$\text{а)}\ \arcsin(x) = \begin{cases} \dfrac{\pi}{2}, & \text{если } |x| = 1 \\ arctg\ \dfrac{x}{\sqrt{1-x^2}}, & \text{если } |x| \neq 1 \end{cases}$$

Запись на Паскале:

```
If abs(x)=1 then
  arcsin:=pi/2
Else
  arcsin:=arctan(x/sqrt(1-sqr(x)));
```

$$\text{б)}\ \arccos(x) = \begin{cases} \dfrac{\pi}{2}, & \text{если } |x| = 0 \\ arctg\ \dfrac{\sqrt{1-x^2}}{x}, & \text{если } x > 0 \\ arctg\ \dfrac{\sqrt{1-x^2}}{x} + \pi, & \text{если } x < 0 \end{cases}$$

На языке Паскаль:

```
If x=0 then
  arccos:=pi/2
Else
  arccos:=arctan(sqrt(1-sqr(x))/x)+pi*byte(x<0);
```

В выражении *byte*(*x*<0) используется аппарат приведения типов, который будет рассмотрен позже. В данном случае

byte(*x*<0) = 1, если $x < 0$; *byte*(*x*<0) = 0, если $x \geq 0$.

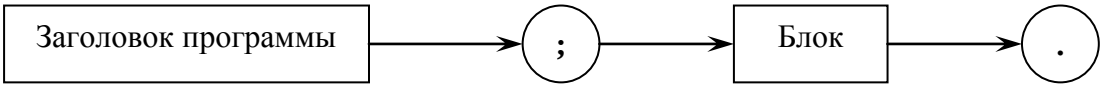
в) *lg*(*x*): *ln*(*x*)/*ln*(10).

Степенная функция для вещественного показателя степени вычисляется следующим образом:

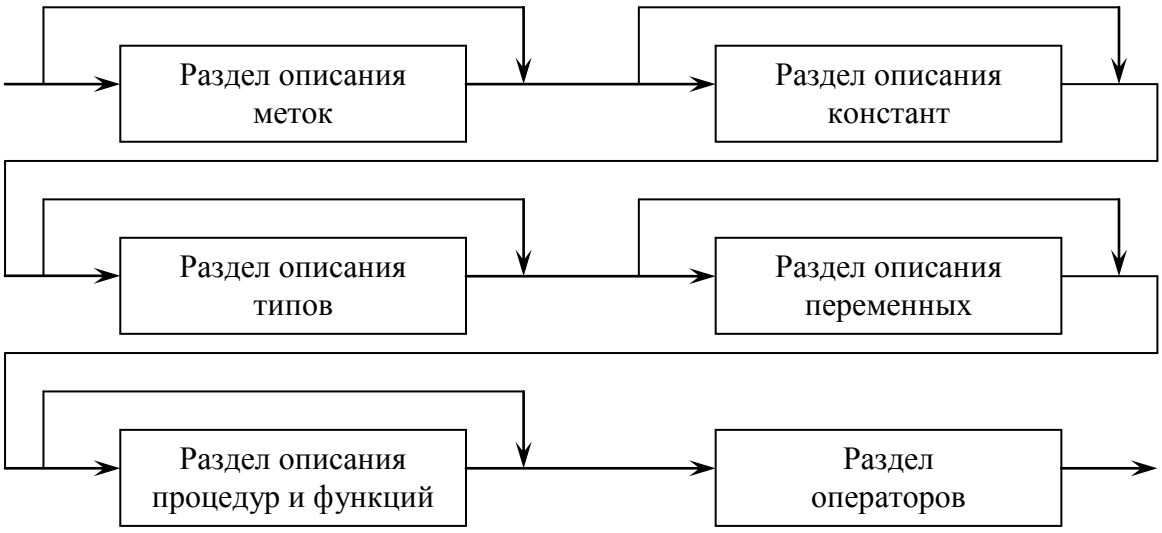
$$a^b = e^{b \cdot \ln a} \Rightarrow \exp(b * \ln(a))$$

СТРУКТУРА ПАСКАЛЬ-ПРОГРАММЫ

Паскаль-программа состоит из заголовка и блока:



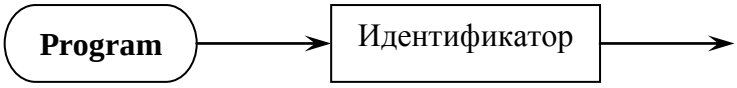
Блок содержит разделы описаний, в которых определяются все локальные по отношению к данной программе объекты, и раздел операторов, где заданы действия, которые необходимо выполнить над этими объектами.



Приведенная синтаксическая диаграмма наглядно показывает, что в блоке может отсутствовать любой из разделов, кроме раздела операторов.

Последовательность описаний, приведенная на синтаксической диаграмме, регламентирована лишь для стандартного Паскаля. В Турбо Паскале она может быть произвольной, любой из разделов описаний может также повторяться несколько раз.

В заголовке программы указывается имя программы:

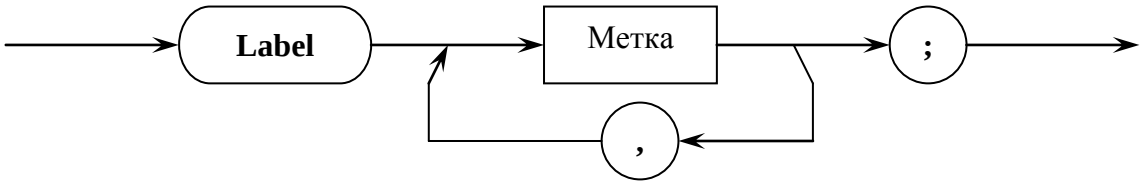


Пример 1.

```
Program  Example;
```

1. Раздел описания меток.

Любой оператор в программе может быть промаркирован меткой. Метка ставится перед оператором и отделяется от него двоеточием. Все метки должны быть описаны в разделе описания меток.

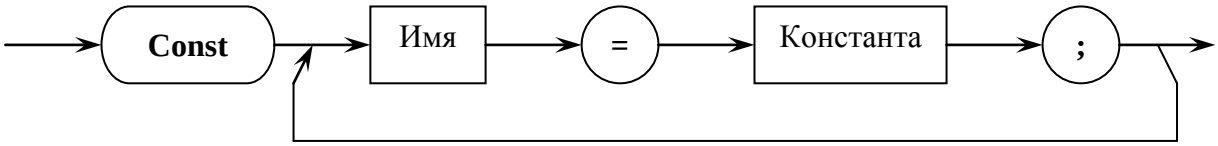


Пример 2.

```
Label  10, 20, Met15;
```

2. Раздел описания констант.

Этот раздел определяет некоторые идентификаторы как синонимы констант.



Пример 3.

```
Const  g = 9.81;
       Nmax = 100;
       TextString = 'Нажмите клавишу Enter';
```

Константа *pi* является предописанной и равна *pi* = 3.1415926536 .

В разделе описания констант можно использовать выражения, в состав которых входят константы, знаки операций и некоторые функции (*abs, odd, ord* и др.). Значения таких выражений вычисляются при компиляции программы.

Пример 4.

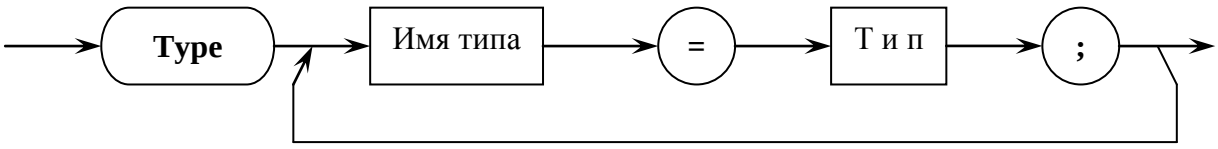
```
Const  pi4 = pi/4;
       MaxSize = MaxInt div SizeOf(real);
```

В операторах программы рекомендуется, как правило, вместо значений констант использовать имена констант. Это улучшает читаемость программы и облегчает ее модификацию.

Предположим, что константа Nmax = 100 в примере 3 определяет максимальный размер массива. Если в операторах программы в различных местах непосредственно записано значение 100, то при необходимости изменения размера массива нужно везде заменить это значение другим, например, числом 200. При этом не исключено, что где-нибудь в программе будет пропущена такая замена или же, наоборот, изменено число 100, которое не является размером массива. В то же время, если в программе везде записано Nmax, то достаточно изменить лишь одно число в разделе констант.

3. Раздел описания типов.

Типы данных *real, integer, boolean, char* являются предопределенными и используются в разделе описания переменных. Если программисту требуется ввести новый тип данных, то этот тип нужно описать в разделе описания типов.



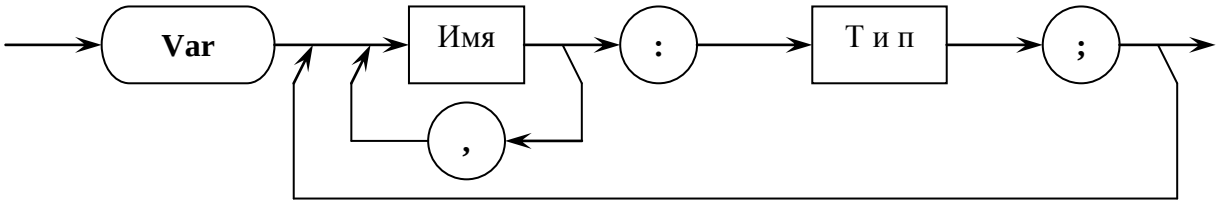
Пример 5.

```
Type  Interval = 10..50;
      Ar = array[1..100] of real;
```

Здесь *Interval* - диапазонный тип, *Ar* - тип массива с вещественными компонентами.

4. Раздел описания переменных.

Каждое имя переменной (простой, составной или другого типа) должно быть приведено в разделе описания переменных.



Пример 6.

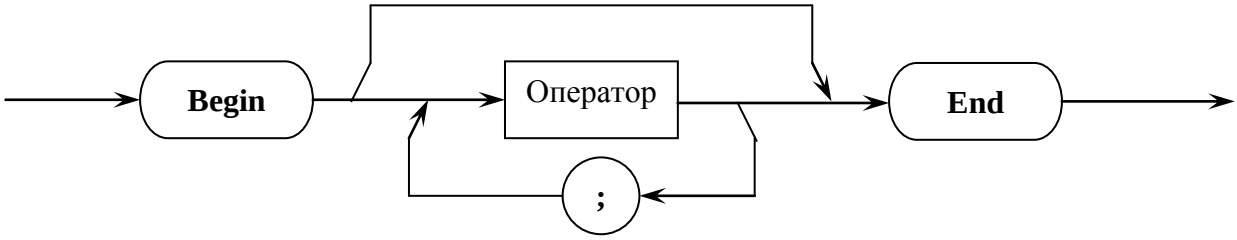
```
Var  i, j, k : integer;
     a, b : real;
     X : Ar;
     d : Interval;
```

Примечание. **Var** – это сокращение слова variable (переменная, 'vɛriəbl).

5. Раздел описания процедур и функций. Назначение и структура раздела будут изложены в дальнейшем при рассмотрении процедур и функций.

6. Раздел операторов.

Раздел содержит в себе операторы, реализующие обработку информации в программе. Его синтаксическая диаграмма :



Раздел операторов - это частный случай составного оператора, который включает в себя один или несколько операторов, заключенных в "операторные скобки" **begin .. end**. Разделителем между операторами является точка с запятой.

Операторы Паскаль-программы - это оператор присваивания, условный оператор, оператор перехода и др.

Из синтаксической диаграммы следует, что между словами ***Begin*** и ***End*** могут отсутствовать какие-либо операторы, может быть одна или несколько точек с запятой. Это возможные примеры пустого оператора, который будет рассмотрен несколько позже.

7. Последовательность разделов.

Как уже было отмечено, разделы описаний в Турбо Паскале могут следовать в произвольном порядке, при этом любой из них может повторяться несколько раз.

Пример 7.

```
Const Nmax = 100;
Type  Ar = array[1..Nmax] of integer;
Var   X : Ar;
Const  g = 9.81;
Type  Ars = array[0..Nmax+1] of real;
Var   i,n : byte;
      Y,Z : Ars;
```

При этом должны выполняться два основополагающих правила:

1) ни одно имя не должно быть описано дважды;

2) любое имя может быть использовано лишь после его описания (например, имя *Nmax* используется в разделе ***Type*** после описания этого имени в разделе ***Const***).

Тем не менее рекомендуется без необходимости не изменять порядок следования описаний, принятый в стандартном языке Паскаль: ***Label, Const, Type, Var***, процедуры и функции.

АЛГОРИТМ И СПОСОБЫ ЕГО ПРЕДСТАВЛЕНИЯ

Обработка информации в ЭВМ производится на основании алгоритма. Под алгоритмом понимают конечное упорядоченное множество правил, четко и однозначно определяющих последовательность операций для решения задачи или класса задач.

Для формализации записи алгоритмов используют различные изобразительные средства. К основным из них относятся формульно-словесный, блок-схемный и запись с помощью алгоритмических языков.

1. Формульно-словесный способ.

Здесь для записи алгоритма используется естественный язык с привлечением, если это необходимо, математических формул и обозначений.

Пример 1. Найти $y = \max(a, b, c)$.

Шаг 1. Положить y равным a.

Шаг 2. Если $b > y$, то положить y равным b.

Шаг 3. Если $c > y$, то положить y равным c. Конец.

Пример 2. Найти наибольший общий делитель двух целых чисел $m \geq n \geq 0$.

Для решения этой задачи обычно используют разложение значений m и n на простые множители, после чего произведение множителей, общих для m и n, определяет их наибольший общий делитель.

Например, для $m = 420$ и $n = 90$ имеем

$$420 = 2 \cdot 2 \cdot 3 \cdot 5 \cdot 7; \quad 90 = 2 \cdot 3 \cdot 3 \cdot 5.$$

Наибольший общий делитель в этом случае равен $2 \cdot 3 \cdot 5 = 30$.

В принципе этот способ можно использовать для формулировки рассматриваемого алгоритма, однако вначале потребуется разработать алгоритм разложения числа на простые множители, что является нетривиальной задачей.

Более просто поставленная задача решается с помощью так называемого алгоритма Евклида.

Обозначим наибольший общий делитель через $d(m, n)$. Вполне очевидно, что $d(m, 0) = m$.

Тогда алгоритм Евклида можно описать следующим образом.

Шаг 1. Если $n = 0$, то принять $d = m$ и закончить вычисления, иначе перейти к шагу 2.

Шаг 2. Вычислить $q = [m / n]$ и $r = m - q n$.

Шаг 3. Заменить значение m на значение n, а значение n - на значение r. Перейти к шагу 1.

Здесь q - целая часть от деления m на n; r - остаток от деления.

При $m = 420$, $n = 90$ имеем:

Шаг 1. $n = 90 \neq 0$;

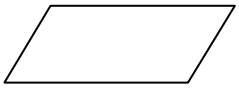
Шаг 2. q = [420/90] = 4; r = 420 - 4 · 90 = 60;
Шаг 3. m = 90; n = 60;
Шаг 1. n = 60 ≠ 0;
Шаг 2. q = [90/60] = 1; r = 90 – 1 · 60 = 30;
Шаг 3. m = 60; n = 30;
Шаг 1. n = 30 ≠ 0;
Шаг 2. q = [60/30] = 2; r = 60 – 2 · 30 = 0;
Шаг 3. m = 30; n = 0;
Шаг 1. n = 0 $\Rightarrow$ d = 30 .

Основным недостатком формульно-словесной записи алгоритма является то, что здесь используется естественный язык, для которого органически присуща неоднозначность слов. К недостаткам данного способа относят также ненаглядность записи алгоритма.

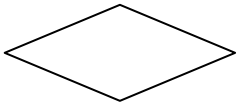
2. Блок-схемный способ.

Блок-схема - это графическое изображение логической структуры алгоритма. Здесь каждый этап вычислительного процесса изображается в виде определенной геометрической фигуры, называемой блоком. Основными блоками являются:


- начало или конец алгоритма (овал);


- ввод или вывод данных (параллелепипед);


- обработка данных, т.е. вычислительный процесс (прямоугольник);


- логический блок (ромб).

Связь между блоками изображают горизонтальными или вертикальными линиями. Излом линии связи делается только под прямым углом. При направлении линии связи слева направо или сверху вниз стрелку на линии можно не ставить; при направлении справа налево, снизу вверх или при наличии излома стрелка обязательна.

3. Алгоритмические языки применяются для записи алгоритмов в виде программ, предназначенных для реализации на ЭВМ.

ОПЕРАТОР ПРИСВАИВАНИЯ

В блок-схемах обычная математическая запись $y = a\,x + b$, означающая, что переменная y принимает значение выражения $a\,x + b$, представляется в виде $y := a\,x + b$ (читается "y становится равным $a\,x + b$"). При выполнении этого оператора переменная y теряет свое предыдущее значение и получает новое значение.

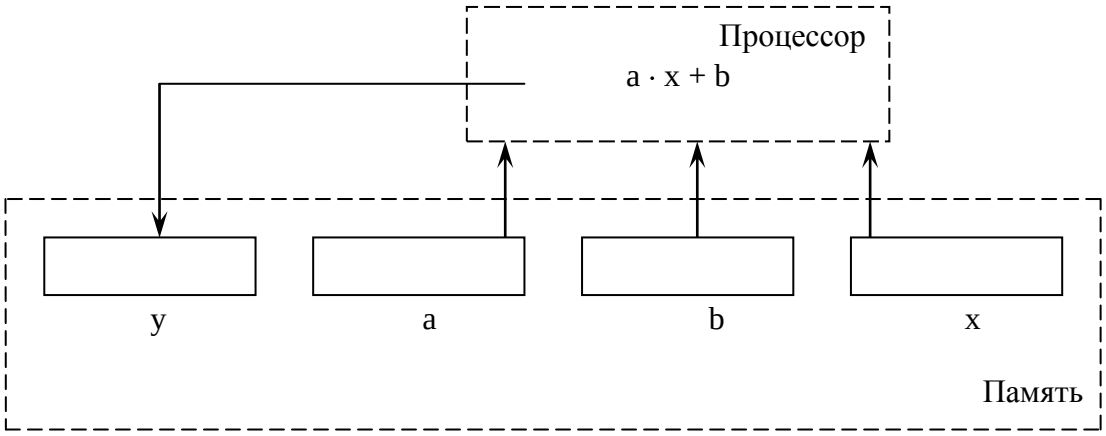
Каждое имя в программе (в том числе в блок-схеме) – это косвенное обозначение адреса поля памяти. Следовательно, при выполнении оператора $y := a\,x + b$ в оперативной памяти будут использованы поля с именами a, b, x, y.

Выполнение оператора присваивания производится в четкой последовательности во времени:

1) из памяти в процессор читаются значения переменных a, x, b (содержимое полей памяти с адресами a, x, b);

2) в процессоре вычисляется значение выражения $a \cdot x + b$;

3) результат вычислений записывается в память в поле с именем y.



В обычном математическом представлении знак "=" имеет два смысла. Запись $y = a\,x + b$ можно интерпретировать следующим образом:

1) вычислить значение выражения $a \cdot x + b$ и присвоить его переменной y;

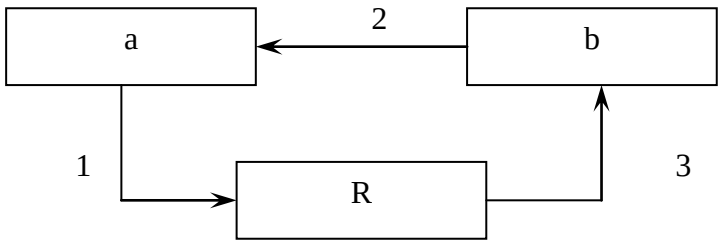
2) проверить, равно ли значение переменной y значению выражения $a \cdot x + b$.

Поскольку в программе двусмысленность недопустима, то в первом случае используется символ ":=" (операция присваивания), а во втором случае - символ "=" (операция отношения).

Использование операции присваивания позволяет записывать такие конструкции, которые в обычной математической интерпретации бессмысленны. Например, $i := i + 1$. Это означает, что к текущему значению переменной i добавляется 1, после чего полученное значение присваивается той же переменной i. Следовательно, значение переменной i увеличивается на единицу.

Пример 1. Обменять значения переменных a и b.

Если мы перешлем содержимое поля a в поле b, т.е. выполним оператор $b := a$, то прежнее значение переменной b будет уничтожено. При выполнении оператора $a := b$ будет уничтожено текущее содержимое поля a. Поэтому обмен содержимого двух полей памяти необходимо делать с помощью дополнительного буферного поля. Это может иметь следующий вид:



Числа 1, 2, 3 указывают последовательность пересылок содержимого полей памяти a, b и R.

Представленную выше схему можно реализовать такими операторами:

$$R := a;\ \ a := b;\ \ b := R\ .$$

Подчеркнем еще раз смысл оператора присваивания. Если в правой части оператора записана переменная, например,

$$a := b,$$

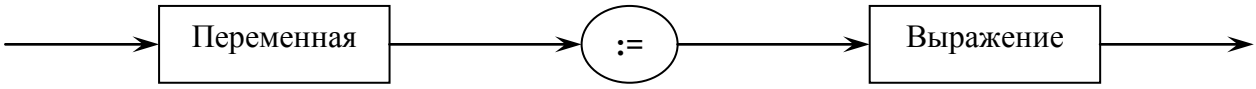
то это означает, что содержимое поля памяти с именем (адресом) b пересылается в поле a. Разумеется, такая пересылка осуществляется через процессор.

Если в правой части оператора присваивания стоит выражение, например,

$$a := b + c - 1,$$

то вначале в процессор читаются значения операндов (содержимое поля b, содержимое поля c и содержимое поля памяти, в котором записана константа 1), вычисляется значение выражения, а полученный результат записывается в поле памяти, имя которого указано в левой части оператора присваивания (в данном случае в поле a).

В Паскале оператор присваивания имеет такое же обозначение, как и на блок-схеме. Его синтаксическая диаграмма:



Выражение состоит из операндов, знаков операций и круглых скобок. Операндом может быть константа, переменная или функция. Выражение задает порядок вычисления значения, основанный на обычном правиле вычисления слева направо и старшинстве операций.

Выражения могут быть арифметические, логические и строковые.

Старшинство операций:

1) ***not;*** (отрицание)
2) *, /, ***div, mod, and;*** (операции типа умножения)
3) +, -, ***or, xor;*** (операции типа сложения)
4) =, <>, <, <=, >, >=; (операции отношения)
5) ***in.*** (операция принадлежности)

Примечание. Операция ***in*** будет определена при рассмотрении множеств.

Пример 2.

$$(25 - 3*3)/(3 + 1) * 7 - 4 + 2*3 = (25 - 9)/4 * 7 - 4 + 6 = 16/4 * 7 + 2 =$$
$$4.0 * 7 + 2 = 28.0 + 2 = 30.0$$

Некоторую особенность имеет вычисление логических операций ***and*** и ***or***, для которых в Турбо Паскале применяется так называемая короткая схема вычислений. По этой схеме вначале вычисляется первый операнд. Если для операции ***and*** получено при этом значение *false*, то второй операнд не вычисляется, так как результат операции все равно будет иметь значение *false.* Аналогично, если первый операнд операции ***or*** имеет значение *true,* то второй операнд не вычисляется. Например, в выражении `(x<0) and (y>x-1)`

при x = 1 будет вычислен лишь первый операнд и получено общее значение *false* вне зависимости от значения, которое имеет переменная *y*.

Оператор присваивания выполним, если переменная и выражение совместимы по присваиванию. Основные варианты совместимости по присваиванию:

1) переменная и выражение относятся к одному типу.

2) переменная относится к типу *real*, а значение выражения - к одному из целых типов.

Пример 3.

```
Var  a,b : real;
     m,n : integer;
Begin
  a:=55.85; m:=15;
  b:=m; n:=a;
```

В операторе $n := a$ не выполняется требование совместимости по присваиванию. Если допустить такой оператор, то имела бы место потеря точности, так как переменной *n* можно присвоить лишь целую часть значения переменной *a* (при присваивании $b:=m$ потери точности не происходит). Если в программе действительно нужно присвоить переменной *n* значение переменной *a*, то это можно сделать следующим образом:

$$n := Trunc(a) \quad \text{или} \quad n := Round(a)\ .$$

Другие варианты совместимости по присваиванию будут рассмотрены позже.

УСЛОВНЫЙ ОПЕРАТОР

С помощью условного оператора реализуются разветвляющиеся вычислительные процессы.

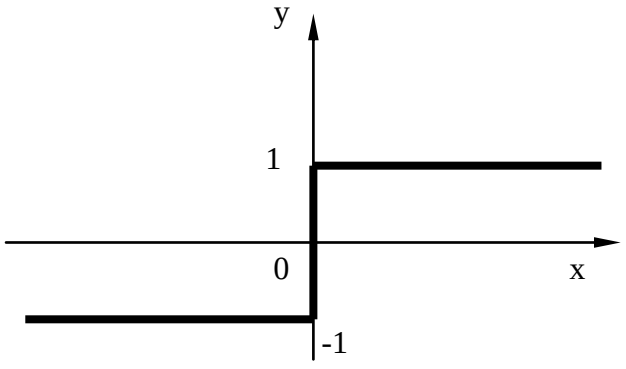
Разветвляющимися процессами называются такие, в которых в зависимости от значения некоторого признака обработка данных производится по одному из возможных направлений (по одной из ветвей).

В условном операторе в зависимости от выполнения проверяемого условия вычисление производится по одной из двух возможных ветвей (если количество ветвей вычислительного процесса свыше двух, то в этом случае обычно используется оператор Case, который в дальнейшем рассматривается в разделе «Оператор варианта»).

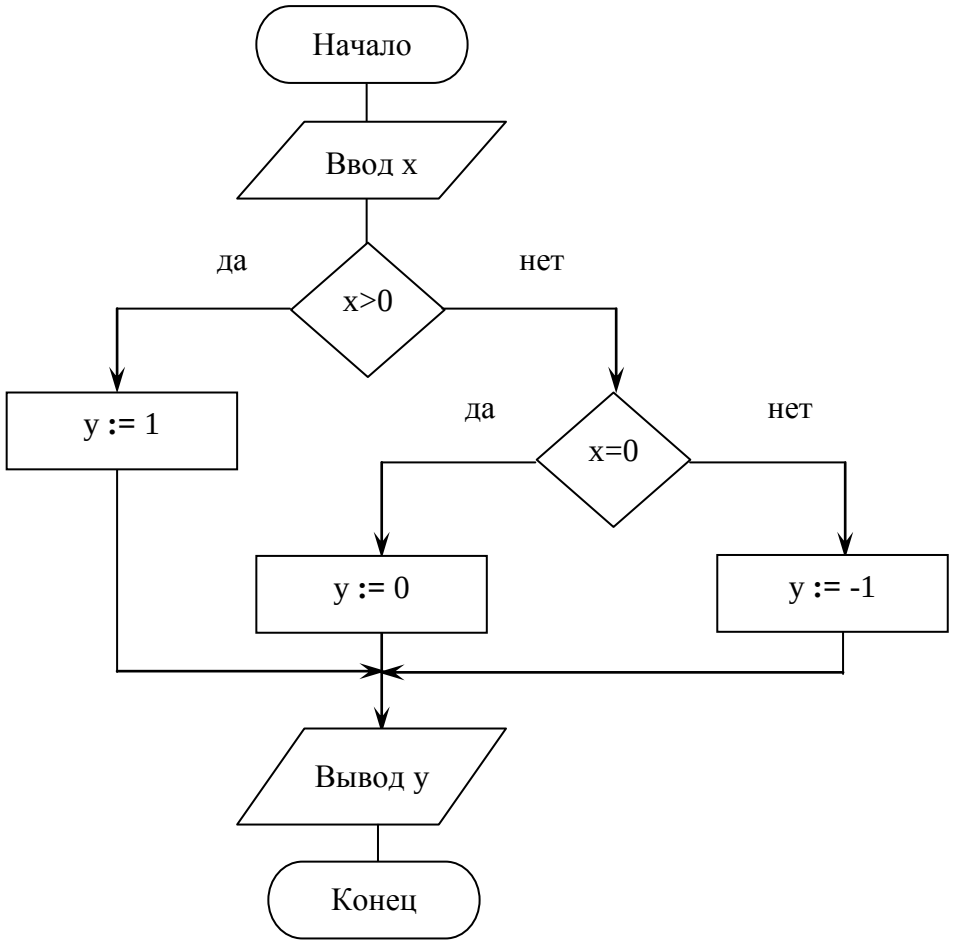
Пример 1.

$$y = \begin{cases} 1, & \textit{если } x > 0 \\ 0, & \textit{если } x = 0 \\ -1, & \textit{если } x < 0 \end{cases}$$

Это сигнатура, функция знака $y = \text{sign}(x)$. График функции:

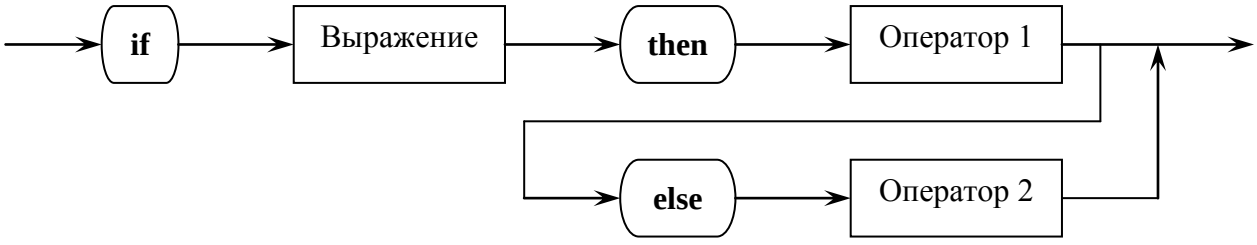


Блок-схема вычисления функции знака:



Здесь производятся две проверки. В первой из них ($x > 0$) определяется первая ветвь вычислений, но остаются неопределенными вторая и третья ветви в формуле вычисления функции знака. Разделение этих ветвей выполняет вторая проверка ($x = 0$). В последнем случае знак «=» - это операция отношения, но не присваивание переменной x нулевого значения.

Синтаксическая диаграмма условного оператора:



Выражение между ***if*** и ***then*** должно иметь тип *boolean*, т.е. это логическое выражение. Если значение выражения равно *true,* то выполняется оператор 1; в противном случае - оператор 2 или не выполняется никаких действий (при отсутствии альтернативы ***else***).

Ранее указывалось, что операторы в Паскаль-программе разделяются между собой точкой с запятой. В связи с этим внутри любого структурного оператора, в том числе опе-

ратора ***if***, символ ";" не должен встречаться, иначе все, что стоит после него, будет считаться уже другим оператором.

Пример 2.

```
If x>0 then
  y:=sqrt(x);
Else
  y:=sqr(x);
```

В этом примере точка с запятой, стоящая перед словом ***else***, заканчивает текст условного оператора. А это приводит к ошибке, поскольку оператора, начинающегося с зарезервированного слова ***else,*** в Паскале нет.

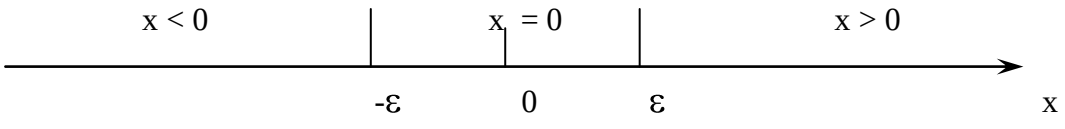
Следовательно, перед словом ***else*** в условном операторе не должно быть точки с запятой, в противном случае это рассматривается как ошибка.

Паскаль-программа для функции знака:

```
Program Sign1;
Var  x : integer;
     y : shortint;
Begin
  Ввод x
  If x>0 then
    y:=1
  Else
    If x<0 then
      y:=-1
    Else
      y:=0;
  Печать  y
End.
```

Если x - вещественная переменная, то сравнение в большинстве случаев нужно выполнять по ε, где ε - достаточно малое число (погрешность задания исходных данных). Как уже ранее указывалось, вещественные числа, как правило, определяют результаты измерений, производимых с определенной точностью. В этом случае значение $x = 0.001$ с точки зрения математики является положительным числом. Если же переменная x означает вес груза, то значение $x = 0.001$ *кг* должно считаться эквивалентным нулю, если точность измерения веса $\varepsilon = 0.1$ *кг*.

На числовой оси вещественный нуль - это ε-окрестность точки 0. Другими словами, если $|x| \le \varepsilon$, то x считается равным нулю.



Следовательно, для вещественных переменных

$$x = 0, \quad \text{если } |x| \le \varepsilon;$$
$$x > 0, \quad \text{если } x > \varepsilon;$$
$$x < 0, \quad \text{если } x < -\varepsilon$$

Аналогично, $a > b$, *если* $a - b > \varepsilon$ и т.п.

Реализация функции знака для вещественного аргумента:

```
Program Sign2;
Const  eps = 0.000001;
Var  x : real;
     y : shortint;
Begin
  Ввод x
  If x>eps then
    y:=1
  Else
    If x<-eps then
      y:=-1
    Else
      y:=0;
  Печать  y
End.
```

В реальных программах вычисление *sign*(*x*) оформляется в виде функции, обращение к которой аналогично, например, обращению к функции *sin*(*x*).

Вне зависимости от физического смысла переменных операции отношения для вещественных значений в большинстве случаев должны выполняться по ε.

Предположим, что в программе требуется вычислить значение корня нечетной степени:

$$y = \sqrt[n]{x} = x^{\frac{1}{n}} = e^{\ln(x)/n}$$

Поскольку областью определения корня нечетной степени является вся числовая ось, то в программе должны анализироваться альтернативные варианты $x < 0$, $x = 0$ и $x > 0$. Учитывая, что *ln*(*x*), используемый при вычислении степенной функции, определен лишь при $x > 0$, рассматриваемую задачу можно реализовать следующим образом:

```
If abs(x)<eps then
  y:=0
Else
  y:=sign(x)*exp(ln(abs(x))/3);
```

То же, но без использования функции знака:

```
If abs(x)<eps then
  y:=0
Else
  Begin
    y:=exp(ln(abs(x))/3);
    If x<0 then
      y:=-y;
  End;
```

Здесь рекомендуется установить *eps* <= 1E-30 (при $x = 10^{-30}$ и $n = 3$ получим $y = \sqrt[3]{10^{-30}} = 10^{-10}$).

Пример 3.

$$y = \frac{a\sin(x)}{\sqrt{1-\cos^2(x)}}\ ;\ \text{сравнить}\ \ a\ \ \text{и}\ \ y\ .$$

```
Var  a,x,y : real;
     b : boolean;
Begin
  a:=5; x:=0.88;
  y:=a*sin(x)/sqrt(1-sqr(cos(x));
  b:=y=a;
```

Значения $\sin(x)$ и $\sqrt{1-\cos^2(x)}$ равны лишь при точных вычислениях. При вычислении на ЭВМ мы имеем дело с приближенными значениями вещественных переменных.

Здесь мы получим b = *false*, так как $y - a = 1.5 \cdot 10^{-11}$ (точное значение $y - a = 0$).

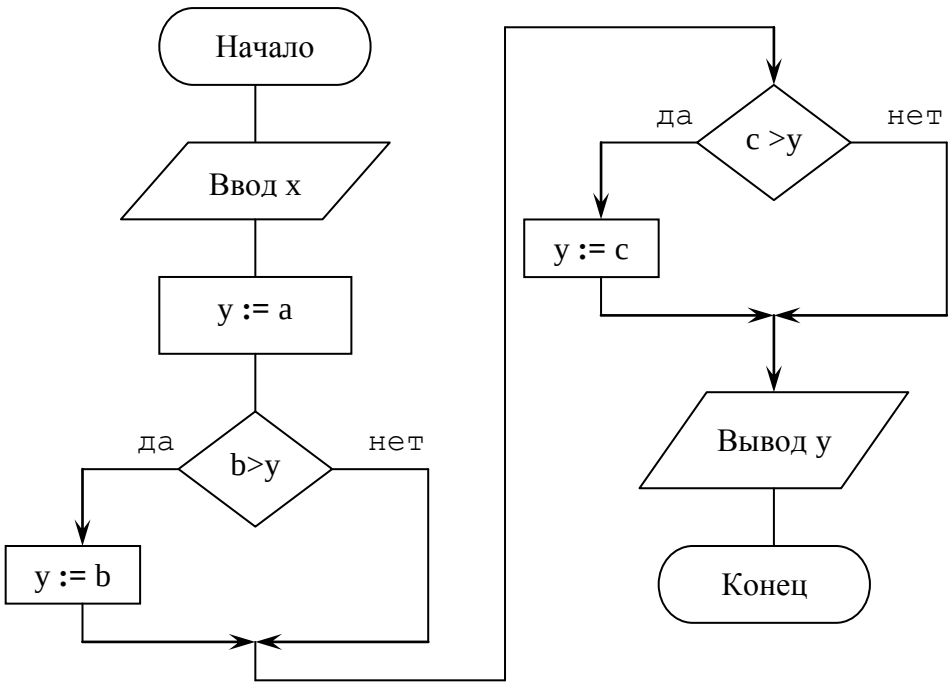
Правильная реализация:

```
Const  eps = 0.000001;
Var  a,x,y : real;
     b : boolean;
Begin
  a:=5; x:=0.88;
  y:=a*sin(x)/sqrt(1-sqr(cos(x));
  b:=abs(y-a)<eps;
```

Здесь имеем b = *true.*

Пример 4. $y = \max(a,b,c)$.

Блок-схема для примера 4.



Ниже приведена программная реализация для примера 4.

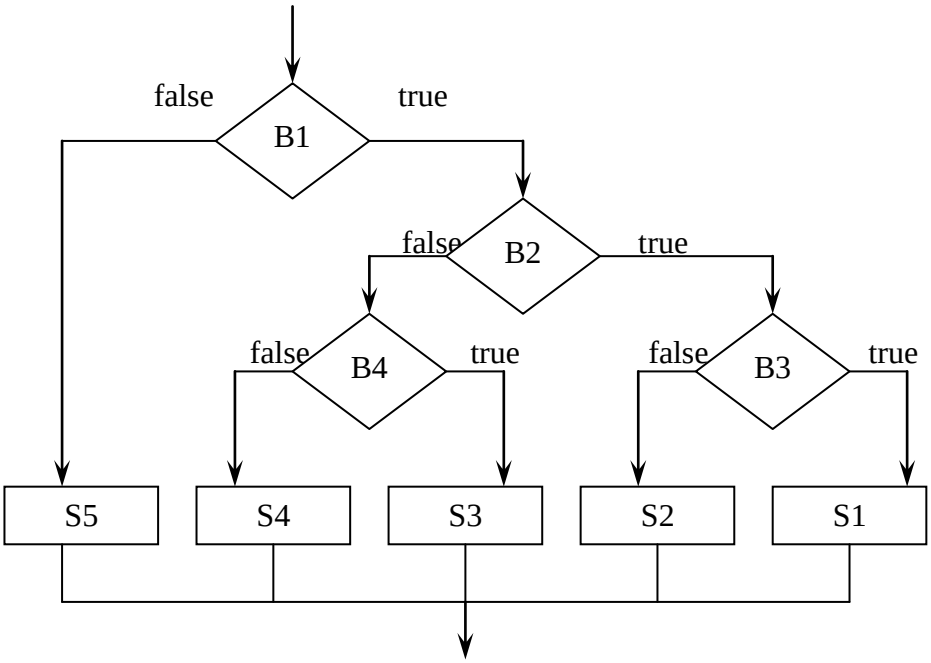
```
Program Max;
Var  y,a,b,c : real;
Begin
  Ввод a,b,c
  y:=a;
  If b>y then
    y:=b;
  If c>y then
    y:=c;
  Печать  y
End.
```

Примечание. Здесь, в отличие от предыдущих примеров, нет необходимости проверять отношения между значениями переменных с учетом погрешности ε.

Операторы 1 и 2, изображенные на синтаксической диаграмме условного оператора, в свою очередь могут быть условными операторами.

Пример 5.

Многоступенчатая схема вложенности условных операторов:



Здесь `B1,B2,B3,B4` – некоторые логические выражения, принимающие значения *true* или *false*; `S1,S2,S3,S4,S5` – операторы (например, операторы присваивания).

```
If  B1  then
  If  B2  then
    If  B3  then
      S1
    Else
      S2
  Else
    If  B4  then
      S3
    Else
```

```
      S4
Else
  S5 .
```

В программе каждое слово "***else***" в условном операторе относится к ближайшему слову "***if*", не связанному с каким-либо словом "*else*".**

Условные операторы, многократно вложенные друг в друга, трудно понимать и в программах их применять не рекомендуется.

Пример 6. Сгруппировать переменные a, b, c в порядке возрастания.

Содержание алгоритма:

1) Если $a > b$, то обменять значения a и b ($a \leftrightarrow b$).
2) Если $a > c$, то $a \leftrightarrow c$.
3) Если $b > c$, то $b \leftrightarrow c$.

```
Program Group;
Var  a,b,c,R : real;
Begin
  Ввод  a,b,c
  If a>b then
    Begin
      R:=a; a:=b; b:=R;
    End;
  If a>c then
    Begin
      R:=a; a:=c; c:=R;
    End;
  If b>c then
    Begin
      R:=b; b:=c; c:=R;
    End;
  Печать  a,b,c
End.
```

ПРОЦЕДУРЫ ВВОДА-ВЫВОДА

Для ввода-вывода данных в Паскаль-программе применяются предописанные процедуры ввода-вывода *Read, Readln, Write, Writeln* (*Readln* - сокращение слов *Read Line* - читать с новой строки; *Write Line* - записать строку). Каждая из этих процедур представляет собой некоторую подпрограмму, которая активизируется оператором процедуры. Процедуры ввода-вывода часто называют просто операторами ввода-вывода.

Мы будем рассматривать только ввод с клавиатуры и вывод на экран дисплея. Процедуры ввода-вывода для произвольных файлов будут рассмотрены позже.

Основной процедурой ввода является *Read.* С ее помощью можно прочитать любое количество исходных данных, указав в скобках идентификаторы переменных, которым нужно присвоить конкретные значения.

Структура обращения к процедуре ввода:

```
Read( список-ввода ),
```

где список-ввода - это имена переменных, разделенные между собой запятыми.

Элементами списка ввода являются простые переменные, т.е. это не могут быть массивы, константы, функции или выражения.

Если в программе встречается процедура *Read*(*x*), то решение прерывается, и программа ожидает от пользователя набора на клавиатуре значения переменной *x* и нажатия клавиши Enter.

При наборе с клавиатуры вводимые данные накапливаются в буфере ввода, который представляет собой специально выделенную для этого область памяти. Максимальная длина буфера ввода при работе с клавиатурой составляет 127 байтов. Одновременно набираемые данные высвечиваются на экране дисплея. После нажатия клавиши Enter происходит чтение данных из буфера ввода и присваивание их значений переменным, имена которых записаны в списке ввода. Числа, набираемые с клавиатуры в буфер ввода, должны представлять собой правильные константы и отделяться друг от друга одним или несколькими пробелами.

Пусть в буфер ввода набраны следующие данные:

15.8 -4 73.7 12.9

и после этого нажата клавиша Enter.

Тогда процедура *Read*(*a,b,c,d*), где *a, b, c, d* - переменные типа *real,* прочтет четыре числа из буфера ввода и присвоит их значения этим переменным.

Написанная выше процедура ввода эквивалентна следующим:

```
Read(a); Read(b); Read(c); Read(d).
```

При отработке процедуры *Read* каждое введенное значение удаляется из буфера.

Предположим, что в буфере ввода после нажатия клавиши Enter содержатся числа

15.8 -4 73.7

Поскольку процедура *Read*(*a,b,c,d*) эквивалентна четырем процедурам

```
Read(a); Read(b); Read(c); Read(d),
```

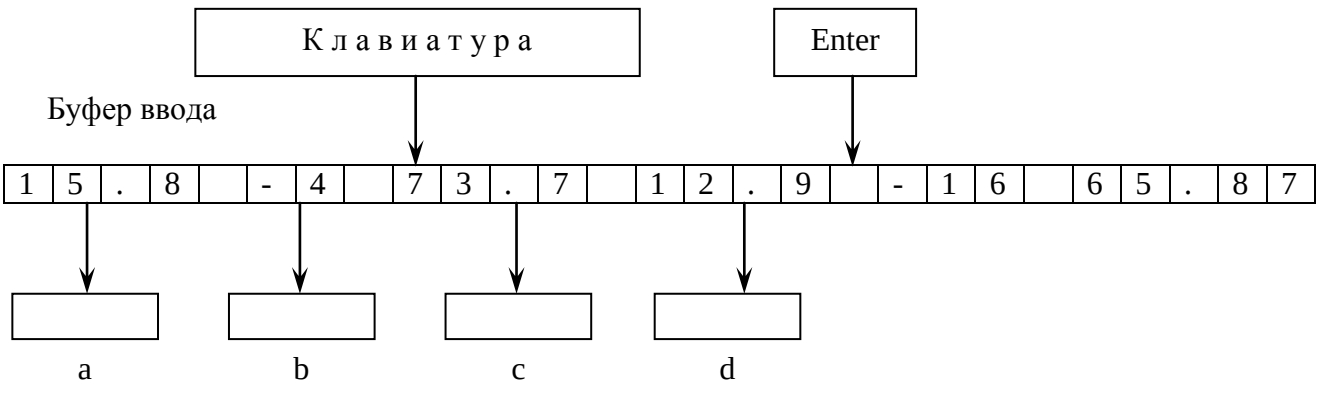
то будут выполнены только первые три из них. Программа снова приостанавливает свою работу в ожидании нового заполнения буфера ввода и нажатия клавиши Enter.

Пусть в аналогичной ситуации буфер ввода имеет вид:

15.8 -4 73.7 12.9 -16 65.87

Тогда из буфера будут введены первые 4 числа и программа продолжит свою работу. Если дальше в программе будут встречаться процедуры *Read,* то оставшиеся в буфере два числа будут введены этими процедурами.

Рассмотренный пример можно проиллюстрировать следующей схемой:



Каждое из полей *a, b, c, d* имеет размер 6 байтов (*a,b,c,d : real*). При передаче процедурой *Read* значения переменной из буфера ввода в поле памяти производится преобразование формы ее представления (из литерного формата, в котором каждый символ занимает один байт, в формат *real*).

Процедура *Readln* отличается от *Read* тем, что после ее выполнения производится переход на новое содержимое буфера ввода. При этом текущее содержимое буфера стирается.

Пусть буфер ввода имеет вид:

15.8 -4 73.7 12.9 -16 65.87

При выполнении процедур

```
Readln(a,b); Readln(c,d);
```

из буфера будут введены первые два числа, содержимое буфера стирается, а работа программы приостанавливается в ожидании нового заполнения буфера.

Процедура *Readln* без параметров производит только сброс буфера ввода и переход к его новому содержимому. В частности, *Readln*(*a,b,c,d*) эквивалентна следующим процедурам:

```
Read(a); Read(b); Read(c); Read(d); Readln.
```

Для вывода данных используются процедуры *Write* и *Writeln.* Процедура *Writeln* отличается от процедуры *Write* тем, что после ее выполнения происходит переход на новую строку экрана.

Элементами списка вывода, в отличие от списка ввода, являются выражения. Частным случаем выражения являются константа, имя переменной, функция. В свою очередь частным случаем константы является текстовая константа, т.е. последовательность символов, заключенная в апострофы.

Пусть заданы две процедуры:

```
Write('  Переменная x = '); Write('  Переменная y = ');
```

При их выполнении на экране будет отпечатано:

```
  Переменная x =   Переменная y =
```

Для процедур

```
  Writeln('  Переменная x = '); Writeln('  Переменная y =');
```

получим:

```
  Переменная x =
  Переменная y =
```

Пример вывода выражений:

```
Write(a+b,'  ',5,'  ',2*sin(x)-1).
```

Выводом можно управлять с помощью форматов. Элемент списка вывода может быть записан в форме

v
или v:w
или v:w:d .

Здесь *v* - выражение, *w* и *d* - форматы.

Если указано только выражение *v*, то количество позиций в строке экрана, отводимое для печати значения *v*, зависит от типа этого значения.

Для целых переменных отводится столько позиций, сколько цифр содержит значение переменной. Для знака "-" отводится одна позиция перед первой цифрой числа. Для знака "+" никакой позиции не отводится.

Пример 1.

```
Var  i : integer;
     l : longint;
```

```
Begin
  i:=555; l:=123456789;
  Write('i=',i); Writeln('   l=',l);
  i:=-555; l:=-123456789;
  Write('i=',i); Writeln('  l=',l);
```

Будет отпечатано:

```
i=555   l=123456789
i=-555  l=-123456789
```

Для вещественного значения отводится 17 позиций. Число печатается в виде мантиссы и порядка. Длина мантиссы - 11 цифр, из них одна цифра целой части числа.

Пример 2.

```
Var  R : real;
Begin
  R:=555; Write('R=',R);
  R:=-555; Writeln('   R=',R);
```

Будет отпечатано:

```
R= 5.5500000000E+02   R=-5.5500000000E+02
```

Здесь *E* -признак десятичного порядка.

Формат *w* указывает ширину поля (количество позиций, отводимых для изображения значения *v*).

Пусть *n* - количество цифр целой переменной, включая знак "-", если он имеется.

Если $w > n$, то значение *v* печатается в правой части поля *w*; если $w < n$, то для печати значения *v* отводится *n* позиций (отсекание "лишних" цифр не производится).

Пример 3.

```
Var  i : integer;
     l : longint;
Begin
  i:=5555; l:=5555;
  Write('i=',i:8); Writeln('   l=',l:2);
  i:=-5555; l:=-5555;
  Write('i=',i:8); Writeln('   l=',l:2);
```

Будет отпечатано:

```
i=    5555   l=5555
i=   -5555   l=-5555
```

Если для вещественной переменной задано значение $w > 17$, то слева от переменной печатается соответствующее количество пробелов; если $w < 17$, то соответственно уменьшается количество цифр мантиссы. Минимальное количество цифр мантиссы равно 2, в этом случае значение $w < 8$ воспринимается как $w = 8$. Значение мантиссы при печати округляется по последней печатаемой цифре.

Пример 4.

```
Var  R : real;
Begin
  R:=555;
  Write('R=',R:20); Writeln('   R=',R:17);
  Write('R=',R:12); Writeln('   R=',R:8);
  Writeln('R=',R:5);
```

Будет отпечатано:

```
R=    5.5500000000E+02   R= 5.5500000000E+02
R= 5.55000E+02   R= 5.6E+02
R= 5.6E+02
```

Формат *d* применяется только для вещественных значений и означает, что число должно печататься в виде целой и дробной частей, разделенных точкой; при этом значение *d* указывает, сколько позиций из общего количества *w* должно быть отведено для печати дробной части числа. Если *d* = 0, то печатается только целая часть без разделяющей точки. Если поля *w* недостаточно для печати всех цифр числа, то к значению *w* добавляется необходимое количество позиций.

Пример 5.

```
Var  R : real;
Begin
  R:=123.456789;
  Write('R=',R:12:5); Writeln('   R=',R:8:5);
  Write('R=',R:8:1); Writeln('   R=',R:8:0);
  Writeln('R=',R:8:10);
  R:=-123.456789;
  Write('R=',R:12:5); Writeln('   R=',R:8:5);
  Write('R=',R:8:1); Writeln('   R=',R:8:0);
  Writeln('R=',R:8:10);
```

Будет отпечатано:

```
R=   123.45679   R=123.45679
R=    123.5   R=      123
R=123.4567890000
R=  -123.45679   R=-123.45679
R=   -123.5   R=     -123
R=-123.4567890000
```

Если *w* < 0, то значения числовых переменных, а также переменных типов *char*, *string* и *boolean* записываются в левые позиции поля *w*. При этом, вне зависимости от значения *w*, для переменной отводится лишь столько позиций, сколько символов содержит ее значение. Если вещественная переменная записывается в виде мантиссы и порядка, то мантисса округляется до двух цифр.

Пример 6.

```
Var   k : integer;  R : real;
      b : boolean;
Begin
  k:=125; r:=-32.63; b:=true;
  Writeln(k:10,' 1');
  Writeln(k:-10,' 1');
  Writeln(k:-1,' 1');
  Writeln(r:18:4,' 1');
  Writeln(r:-18:4,' 1');
  Writeln(r:-1:4,' 1');
  Writeln(r:28,' 1');
  Writeln(r:-0,' 1');
  Writeln('0123456789':20,' 1');
  Writeln('0123456789':-20,' 1');
  Writeln(b:20,' 1');
```

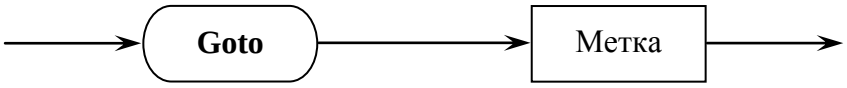
```
  Writeln(b:-1,' 1');
```
Будет напечатано:
```
       125 1
125 1
125 1
            -32.6300 1
-32.6300 1
-32.6300 1
          -3.2630000000E+01 1
-3.3E+01 1
         0123456789 1
0123456789 1
               TRUE 1
TRUE 1
```

Примечание. В список ввода могут входить лишь вещественные и целочисленные переменные, а также переменные типа *char* и *string*. Следовательно, переменные других типов (массивы, множества и др.) не могут быть непосредственно введены процедурой *Read*. Косвенная методика ввода таких переменных рассматривается дальше в соответствующих разделах.

В список вывода, кроме типов, перечисленных в списке ввода, могут входить также переменные типа *boolean*.

ОПЕРАТОР ПЕРЕХОДА

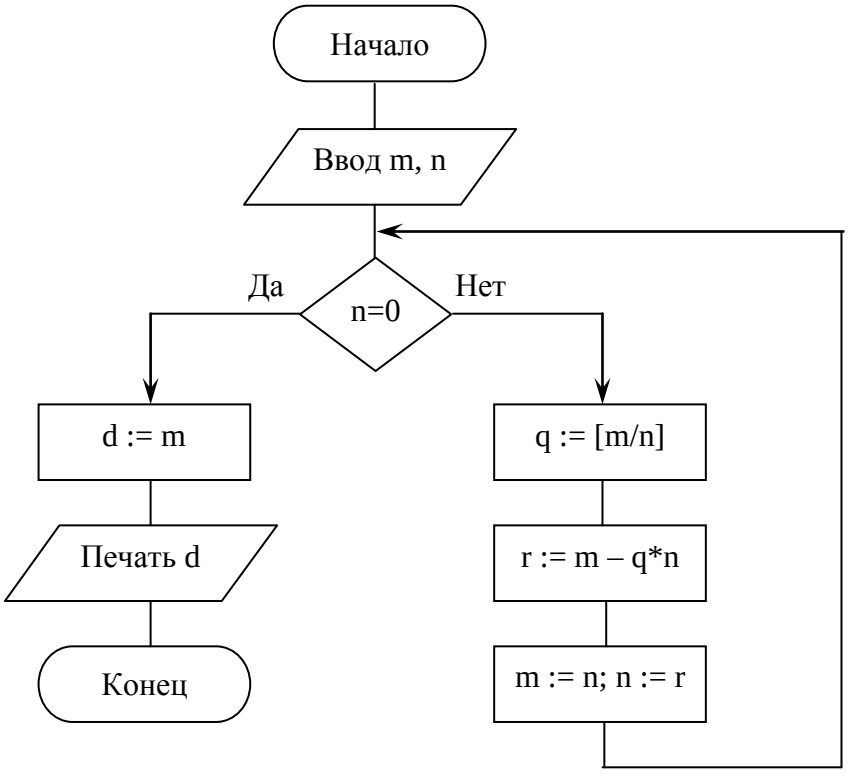
Синтаксическая диаграмма:



Оператор перехода указывает, что дальнейшая работа должна продолжаться в другой части программы, а именно с того места, где находится метка.

Переход внутрь сложного оператора запрещен (такими операторами являются составной и условный операторы, операторы цикла, варианта, присоединения).

Пример. Алгоритм Евклида.



```
Program Evklid1;
Label  10;
Var  m,n,d,q,r : word;
Begin
  Read(m,n); Writeln('m = ',m,'  n = ',n);
  10:
  If n>0 then
    Begin
      q:=m div n; r:=m mod n;
      m:=n; n:=r;
      Goto 10
    End
  d:=m; Writeln('d=',d)
End.
```

Примечание. Оператор *q := m* ***div*** *n* является избыточным, поскольку вычисленное значение *q* в программе Evklid1 не используется.

Рассмотрим вкратце, что собой представляет метка в программе.

При трансляции Паскаль-программы каждый ее оператор преобразуется в одну или несколько машинных команд. Эти команды размещаются в последовательных полях памяти. Каждая команда имеет конкретный адрес, определяемый адресом крайнего левого байта поля памяти, которое занимает команда. Команды, как правило, выполняются последовательно, в порядке их записи в памяти. Однако если встречается команда перехода, которой соответствует оператор ***Goto*** в Паскаль-программе, то дальше будет выполняться не следующая по порядку машинная команда, а команда, адрес которой указан в команде перехода. Этот адрес перехода и соответствует метке в Паскаль-программе.

Оператор ***Goto*** рекомендуется применять лишь в исключительных случаях, когда сложный фрагмент программы трудно реализовать без этого оператора или когда оператор перехода заметно повышает эффективность программы. В общем случае наличие в программе операторов ***Goto*** ухудшает понимание программы. Особенно недопустимым считается передача управления снизу вверх, как это сделано в программе Evklid1.

Ниже перечисляются случаи, когда целесообразно применять оператор ***Goto:***

- выход из блока в вызывающую программу (переход с помощью оператора ***Goto*** на конец блока);
- переход на конец тела цикла;
- принудительный выход из цикла;
- переход к удаленному фрагменту программы.

Однако для того, чтобы явно не применять оператор ***Goto,*** в Турбо Паскале для первых трех случаев предусмотрены специальные операторы, а именно ***Exit, Continue*** и ***Break.***

*Примечание****.*** Операторы ***Continue*** и ***Break*** введены в версии 7.0 языка Турбо Паскаль.

В частности, алгоритм Евклида легко реализовать без операторов ***Goto*** с помощью операторов цикла ***While*** или ***Repeat.***

ПУСТОЙ ОПЕРАТОР

В Паскаль-программе точка с запятой используется как разделитель между операторами, т.е. она не входит в состав оператора. Поэтому если в программе поставить подряд две точки с запятой, то будет считаться, что между ними находится пустой оператор. Тот же эффект мы будем иметь, если перед словом ***end*** поставить точку с запятой, если устано-

вить точку с запятой непосредственно после слова ***else***, если нет ни одного оператора между альтернативами ***then*** и ***else*** и др. Пустой оператор не выполняет никаких действий, но может иметь метку. Эта метка может быть использована, например, для перехода с помощью оператора ***Goto*** на конец составного оператора ***begin*** … ***end***.

Пример.

```
Program Empty;
Label  10,20;
Var  x,y,z : real;
Begin
  x:=-15.3; ;
  y:=exp(x)-1;
  10: ;
  z:=x+y;
  20:
End.
```

Здесь три пустых оператора.

ОПЕРАТОР ЦИКЛА С ПРЕДУСЛОВИЕМ

Циклом называется многократно выполняемый участок программы.

Обычно для организации циклических программ используют операторы цикла. В Паскале определены три таких оператора: оператор цикла с предусловием, оператор цикла с постусловием и оператор цикла с параметром.

Синтаксическая диаграмма оператора цикла с предусловием:

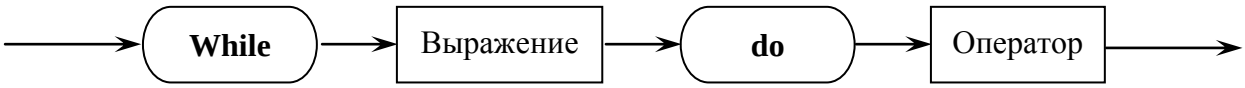
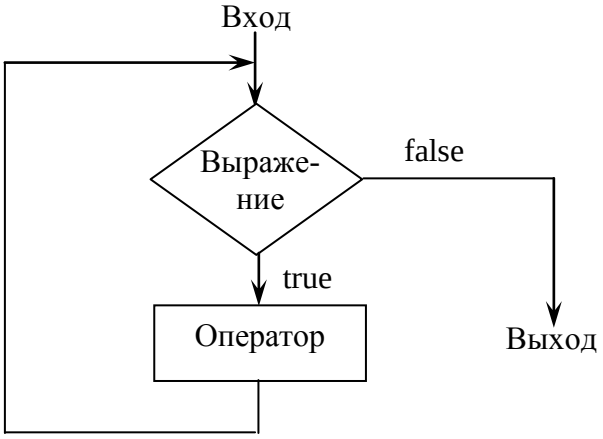


Схема выполнения оператора ***While:***



Значение выражения должно быть типа *boolean.* Оператор после слова ***do*** выполняется нуль или более раз. Чтобы цикл не был бесконечным (случай "зацикливания"), значение выражения в операторе должно изменяться таким образом, чтобы оно стало равным *false.*

Пример 1. Зацикливание программы.

```
x:=10;
```

```
  While x>0 do
    Begin
      y:=x+1; z:=y-2;
    End;
```

Пример 2. Алгоритм Евклида.

```
Program Evklid2;
Var  m,n,d,r : word;
Begin
  Read(m,n); Writeln('m = ',m,'  n = ',n);
  If m<n then
    Begin                      { Обмен значений m и n, }
      r:=m; m:=n; n:=r;        { если  m < n           }
    End;
  While n>0 do
    Begin
      r:=m mod n; m:=n; n:=r;
    End;
  d:=m; Writeln('d = ',d)
End.
```

Если в данном примере введено $m = 10$, $n = 0$, то цикл *While* ни разу не выполняется и в результате получаем $d = 10$.

Пример 3. Модификация алгоритма Евклида.

```
Program Evklid2a;
Var  m,n,d : word;
Begin
  Read(m,n); Writeln('m = ',m,'  n = ',n);
  While (m>0) and (n>0) do
    If m>n then
      m:=m mod n
    Else
      n:=n mod m;
  d:=m+n; Writeln('d = ',d)
End.
```

Программы Evklid2 (пример 2) и Evklid2a (пример 3) можно считать сопоставимыми по быстродействию. Компьютерный эксперимент показывает, что программа Evklid2a работает примерно на 9% быстрее по сравнению с программой Evklid2. Это достигается за счет того, что здесь, в отличие от программы Evklid2, в каждом цикле не выполняются операторы m := n и n := r.

Работа цикла **While** в примере 3 заканчивается, когда переменная *m* или переменная *n* становятся равными нулю.

Оператор цикла с предусловием наиболее часто применяют при реализации итерационных процессов.

ПРОГРАММИРОВАНИЕ ИТЕРАЦИОННЫХ ЦИКЛОВ

Итерационным называется такой вычислительный процесс, для которого заранее неизвестно количество выполнений цикла; это количество зависит от конкретных значений исходных данных. Критерием окончания циклического процесса является выполнение некоторого заранее заданного условия.

Частным случаем итерационного вычислительного процесса является так называемый метод последовательных приближений, в котором для определения очередного, более точного значения функции используется ее предыдущее значение.

Как правило, признаком окончания вычислений является достижение заданной точности:

$$\left| y_{n+1} - y_n \right| \le \varepsilon,$$

где ε - максимально допустимая погрешность вычисления функции y;

y_n, y_{n+1} - предыдущее и очередное значения функции y.

Пример 1. Итерационная формула Ньютона для функции $y = \sqrt[m]{x}$:

$$y_{n+1} = \frac{1}{m}[(m-1)y_n + \frac{x}{y_n^{m-1}}]; \quad y_0 = x; \quad x \ge 0 \ ,$$

где y_0 - начальное приближение. В частности, для $y = \sqrt{x}$ $(m = 2)$ получим:

$$y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n}); \quad y_0 = x \ .$$

Рассмотрим последовательность вычисления функции $y = \sqrt{x}$ для $x = 16;\ \varepsilon = 0.01$.

y_0 = 16 ;

y_1 = 0.5 * (16 + 16/16) = 8.5 ;

y_2 = 0.5 * (8.5 + 16/8.5) = 5.191176 ;

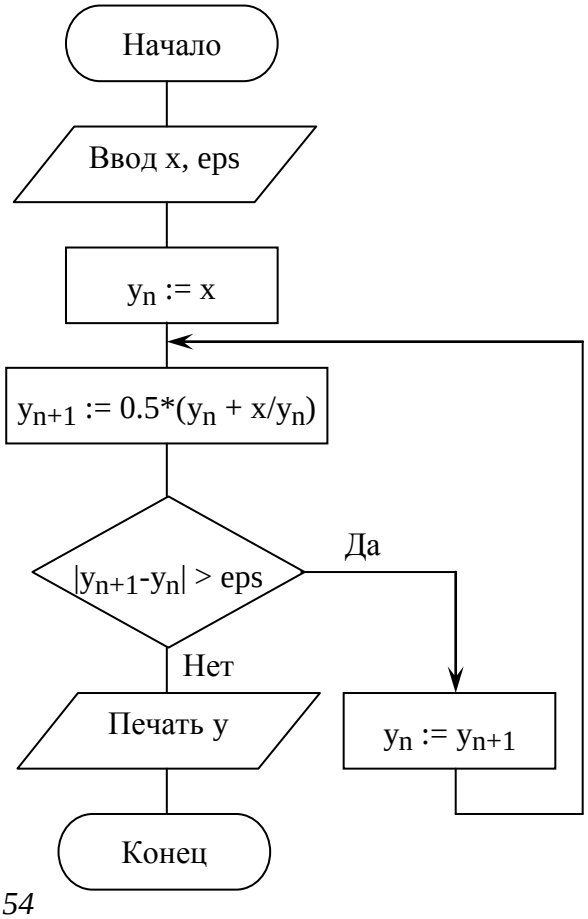
y_3 = 0.5 * (5.191176 + 16/5.191176) = 4.136664 ;

y_4 = 0.5 * (4.136664 + 16/4.136664) = 4.002257 ;

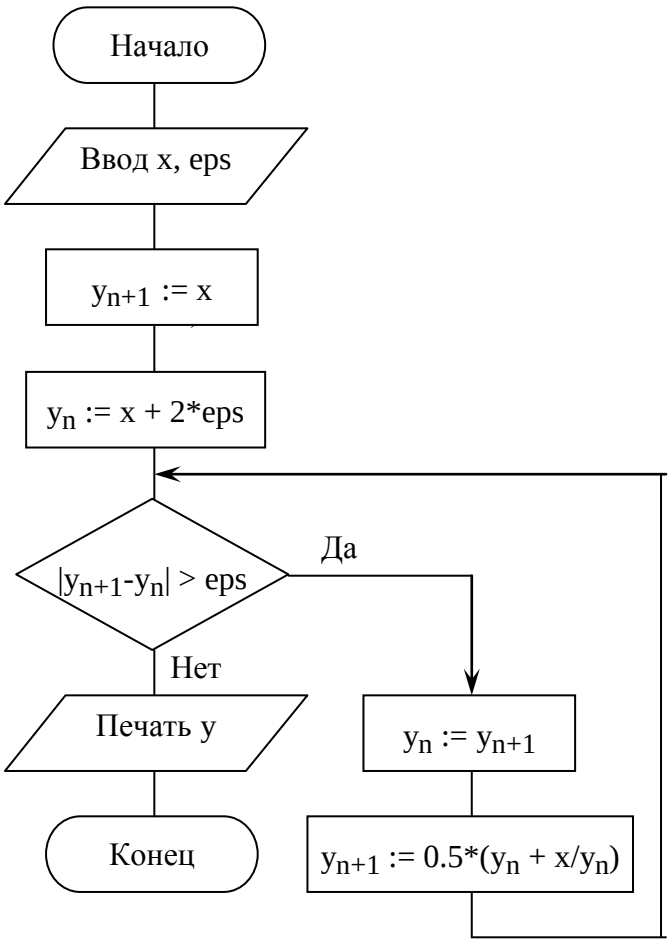
y_5 = 0.5 * (4.002257 + 16/4.002257) = 4.000005 .

Условие $\left| y_{n+1} - y_n \right| \le \varepsilon$ выполняется при $n = 4$. Следовательно, значение y = 4.000005 есть значение искомой функции при x = 16 с погрешностью не более ε = 0.01 .

а)



б)



В блок-схеме варианта а) нельзя применить оператор цикла с предусловием, так как в операторе ***While*** проверка условия выполнения цикла должна производиться в самом начале цикла. В связи с этим внесены изменения в схему реализации итерационной формулы Ньютона (вариант б)).

Следует отметить, что здесь переменные y_n и y_{n+1} не являются элементами массива, это два последовательных во времени значения функции y (предыдущее и текущее значения).

В программе вместо исходных переменных y_{n+1}, y_n будут использованы переменные *y, yn.*

```
Program Newton1;
Const  eps = 0.00001;
Var  x,y,yn : real;
Begin
  Ввод и печать x
  y:=x; yn:=x+2*eps;
  If x>0 then
    While abs(y-yn)>eps do
      Begin
        yn:=y;
        y:=0.5*(yn+x/yn);
      End;
  Печать  y
End.
```

Оператор "***If*** $x > 0$ ***then***" в программе Newton1 исключает ее прерывание из-за деления на нуль при $x = 0$ (если $x = 0$, то цикл **While** не выполняется и результатом является значение $y = 0$).

Пример 2. Вычисление функции по ее разложению в ряд.

В Паскале, как и в других языках программирования высокого уровня, определены стандартные функции *sin*(*x*), *cos*(*x*) и др. Вполне очевидно, что вычисление этих функций производится по некоторым формулам, а не по таблицам значений. Одним из методов вычисления функций является разложение их в ряд, в частности, в так называемый ряд Маклорена (существуют и другие ряды, например, ряд Лорана, ряд Фурье, разложение по полиномам Чебышева и пр.). Программная реализация таких рядов сводится к организации итерационных циклов.

Рассмотрим методику вычисления по формулам разложения в ряд на примере функции $y = \sin(x)$.

$$y = \sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots \qquad (1)$$

Как известно, $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$. Например, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$.

В частности, считают $1! = 1; \quad 0! = 1$.

Тогда

$$y = \sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \ldots$$

Является очевидным, что при малых значениях аргумента x (например, $x < 0.1$), значение функции $\sin(x)$ примерно равно первому члену ее разложения в ряд, т.е. значению самого аргумента x (в связи с этим в учебнике тригонометрии записывается $\sin x \approx x$ при малых значениях переменной x).

Пусть $x = \frac{1}{2}$. Тогда

$$y = \sin(\frac{1}{2}) = \frac{1}{2} - \frac{1}{8 \cdot 6} + \frac{1}{32 \cdot 120} - \frac{1}{128 \cdot 5040} + ... = 0.5\text{-}0.020833\text{+}0.0002604\text{-}$$
$$\text{- } 0.00000155\text{+}...$$

При $\varepsilon = 0.001$ достаточно взять первые три члена разложения функции $sin(x)$ в ряд. Тогда $y = sin(0.5) = 0.4794$.

Как видно из примера, элементы разложения функции $\sin(x)$ очень быстро убывают по абсолютному значению, стремясь в бесконечности к нулю.

Обозначим члены ряда через $a_0, a_1, a_2, ..., a_n$. Тогда общий член ряда

$$a_n = (-1)^n \frac{x^{2n+1}}{(2n+1)!} \qquad (2)$$

Частная сумма, определяющая с некоторой погрешностью значение искомой функции y , есть

$$\begin{aligned} S_0 &= a_0 \\ S_1 &= a_0 + a_1 = S_0 + a_1 \\ S_2 &= a_0 + a_1 + a_2 = S_1 + a_2 \\ &\text{……..} \\ S_{n+1} &= S_n + a_{n+1} \end{aligned} \qquad (3)$$

Условие окончания вычислений (окончания итерационного цикла):

$$\left| S_{n+1} - S_n \right| \le \varepsilon$$

или, используя (3),

$$\left| a_{n+1} \right| \le \varepsilon \ .$$

Каждый очередной член ряда (1) можно вычислять непосредственно по формуле (2), задав соответствующее значение n . Однако при этом значительная часть вычислений будет дублировать друг друга.

Например, при вычислении a_2 нужно выполнить действия

$$\frac{x \cdot x \cdot x \cdot x \cdot x}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5} \ ,$$

для вычисления a_3 - действия

$$-\frac{x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x}{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7} \ .$$

Однако при вычислении a_2 уже были получены значения x^5 и 5! Поэтому при вычислении a_3 целесообразно использовать уже имеющееся значение a_2:

$$a_3 = -\frac{a_2 \cdot x \cdot x}{6 \cdot 7}$$

Выведем формулу для общего случая вычисления a_{n+1}.

Из формулы (2), подставив вместо n значение $n+1$, получим

$$a_{n+1} = (-1)^{n+1} \frac{x^{2n+3}}{(2n+3)!} .$$

Тогда

$$\frac{a_{n+1}}{a_n} = -\frac{x^{2n+3}(2n+1)!}{(2n+3)!\ x^{2n+1}}$$

$$a_{n+1} = -\frac{x^2}{(2n+2)(2n+3)} a_n \qquad (4)$$

Здесь используется равенство

$$(2n+3)! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (2n)(2n+1)(2n+2)(2n+3) = (2n+1)!(2n+2)(2n+3)$$

Формула типа (4), в которой для вычисления очередного члена a_{n+1} числовой последовательности $a_0, a_1, a_2, ..., a_n, a_{n+1}, ...$ используется значение ее предыдущего члена a_n, называется ***рекуррентной***. Слово «рекуррентный» оначает «возвращающийся». В данном случае это элемент последовательности, который определяется через предыдущие и для его вычисления нужно возвращаться к ним.

Например, для арифметической прогрессии мы имеем

$$a_n = a_{n-1} + d ;$$

аналогично для геометрической прогрессии

$$b_n = b_{n-1} \cdot q.$$

Для вычисления ряда (1) имеем следующие рабочие формулы:

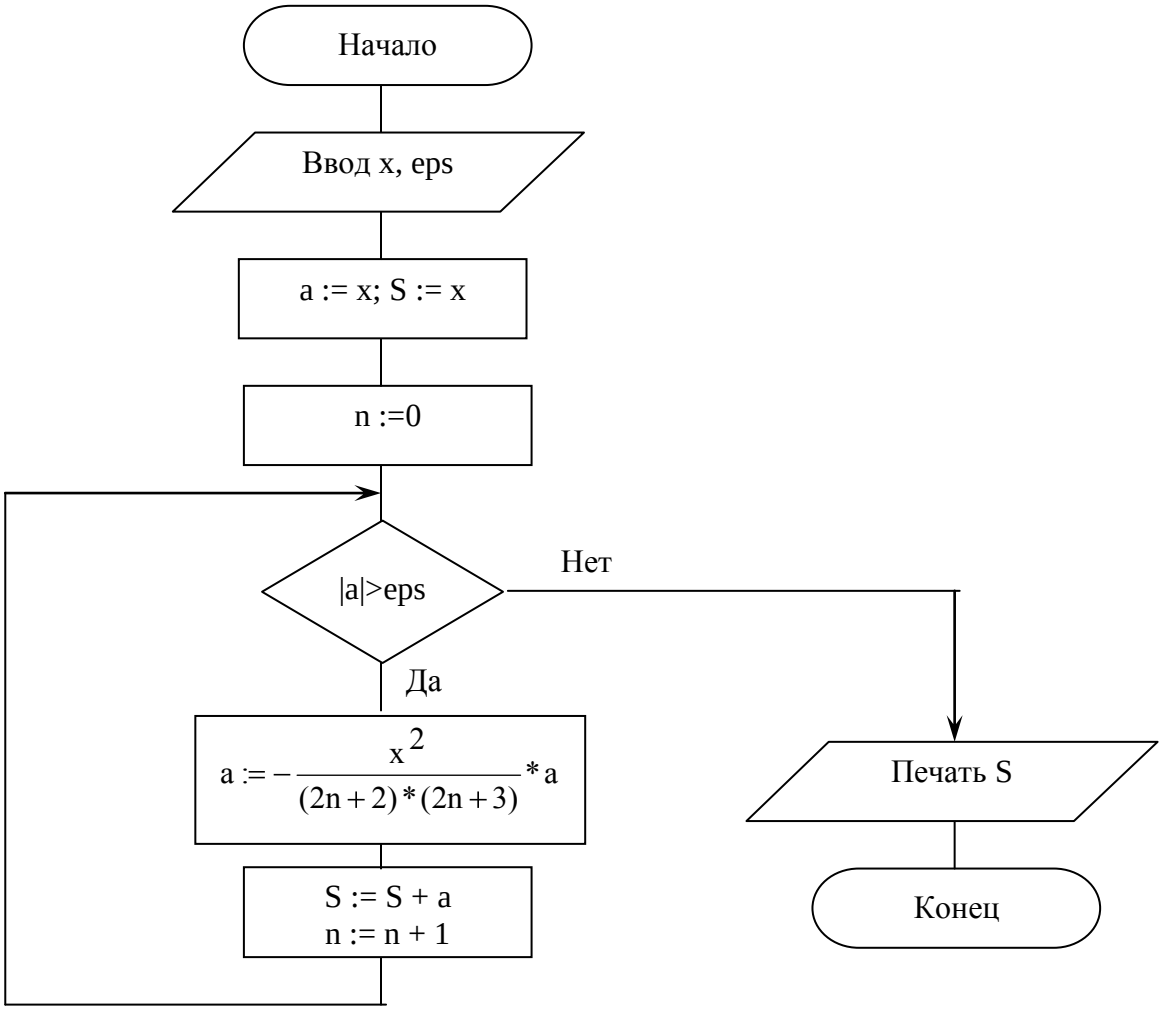
$$a_0 = x ; \quad S_0 = x ; \quad a_{n+1} = -\frac{x^2}{(2n+2)(2n+3)} a_n$$

$$S_{n+1} = S_n + a_{n+1} ; \quad |a_{n+1}| \le \varepsilon$$

```
Program MacLoren;
Const  eps = 0.00001;
Var  x,a,S,n  :  real;
Begin
  Read(x); Writeln('x = ',x:8:3);
  a:=x; S:=x; n:=0;
  While abs(a)>eps do
    Begin
      a:=-sqr(x)*a/((2*n+2)*(2*n+3));
      S:=S+a; n:=n+1
    End;
  Writeln('S,n = ',S:8:3,'    ',n:3:0);
End.
```

Значение целочисленной переменной n непосредственно входит в формулу для вычисления элемента a. Чтобы исключить многочисленные преобразования *integer* $\rightarrow$ *real*, в программе переменная n объявлена вещественной. Тогда в процедуре Writeln формат n:3:0 указывает, что значение переменной n следует печатать без дробной части и точки, разделяющей ее целую и дробную части.

Блок-схема вычислений:



Пример 3. Требуется найти корень функции $y = f(x)$ на интервале $[a, b]$ с погрешностью, не превышающей заданного значения ε. При этом предполагается, что на интервале $[a, b]$ имеется лишь один корень этой функции.

Из существующих методов уточнения корня функции будем использовать наиболее простой - метод половинного деления.

Вычислим $F_a = f(a)$ и $F_b = f(b)$. Поскольку на заданном интервале расположен лишь один корень функции, то $sign(F_a) \neq sign(F_b)$.

Разделим исходный интервал пополам и определим

$$c = \frac{a+b}{2} \quad \text{и} \quad F_c = f(c).$$

Если $sign(F_a) = sign(F_c)$, то передвинем левую границу к середине интервала, т.е. примем $a = c$ и $F_a = F_c$. В противном случае к середине интервала сдвигаем правую границу: $b = c$, $F_b = F_c$. Деление интервала пополам будем продолжать до тех пор, пока не будет выполнено условие $(b - a) \leq \varepsilon$.

Определение корня функции методом половинного деления реализовано ниже в программе Root на примере функции

$$y = \frac{1}{4}\sqrt{x} + \sin(x)$$

на интервале $\left[\frac{\pi}{2}, \frac{3}{2}\pi\right]$.

```
Program Root;
Const  eps = 0.001;
Var  n : byte;
     a,b,c,Fa,Fb,Fc : real;
Begin
  a:=0.5*pi; b:=1.5*pi;
  Fa:=0.25*sqrt(a)+sin(a); Fb:=0.25*sqrt(b)+sin(b);
  n:=0;
  While (b-a)>eps do
    Begin
      n:=n+1;
      c:=0.5*(a+b); Fc:=0.25*sqrt(c)+sin(c);
      If Fa*Fc>0 then      { если Fa*Fc>0, то переменные }
        Begin              { Fa и Fc имеют одинаковый }
          a:=c; Fa:=Fc     { знак }
        End
      Else
        Begin
          b:=c; Fb:=Fc
        End
    End;
  c:=0.5*(a+b); Fc:=0.25*sqrt(c)+sin(c);
  Writeln('n = ',n,'  c = ',c:9:6,'  Fc = ',Fc:12);
End.
```

Результаты вычислений по программе Root приведены ниже в таблице.

ε	n	c	F_c
0.01	9	3.641670	-2.41473E-03
0.001	12	3.638986	-2.33136E-04
0.0001	15	3.638746	-3.81965E-05
0.00001	19	3.638702	-1.63049E-06
0.000001	22	3.638699	-1.07303E-07

Примечание. Рассмотренный ранее алгоритм Евклида также является итерационным. Здесь количество циклов заранее неизвестно и определяется значениями исходных данных m и n. Условием окончания цикла является выполнение отношения $n = 0$.

ВЫЧИСЛЕНИЕ СТЕПЕННОЙ ФУНКЦИИ

Рассмотрим вычисление функции $y = x^n$, где n - целое неотрицательное значение (переменная x может быть вещественного или целого типа).

Если $n < 0$, то вычисляется $z = x^{|n|}$, после чего $y = \frac{1}{z}$. При $n = 0$ имеем $y = 1$.

Пусть $y = x^{165}$. Для получения значения y в виде

$$y = x \cdot x \cdot x \cdot \ldots \cdot x \cdot x$$

требуется выполнить 164 умножения. Однако возможна и другая схема вычислений:

$$\begin{array}{l|l} y_0 = x & \text{1-я степень} \\ y_1 = y_0 \cdot y_0 & \text{2-я степень} \\ y_2 = y_1 \cdot y_1 & \text{4-я степень} \\ y_3 = y_2 \cdot y_2 & \text{8-я степень} \\ y_4 = y_3 \cdot y_3 & \text{16-я степень} \\ y_5 = y_4 \cdot y_4 & \text{32-я степень} \\ y_6 = y_5 \cdot y_5 & \text{64-я степень} \\ y_7 = y_6 \cdot y_6 & \text{128-я степень} \end{array}$$

$$y = y_7 \cdot y_5 \cdot y_2 \cdot y_0 = x^{128} \cdot x^{32} \cdot x^4 \cdot x^1$$

Здесь требуется только 10 умножений.

В приведенной схеме используется разложение показателя n по степеням числа 2:

$$165 = 128 + 32 + 4 + 1 .$$

Обозначим через m текущее значение показателя степени (исходное значение n нельзя изменять в процессе вычисления функции y), через b - сомножитель, включаемый в состав произведения y.

Начальные значения переменных y, m, b:

$$y := 1; \quad m := n; \quad b := x.$$

Описание алгоритма:

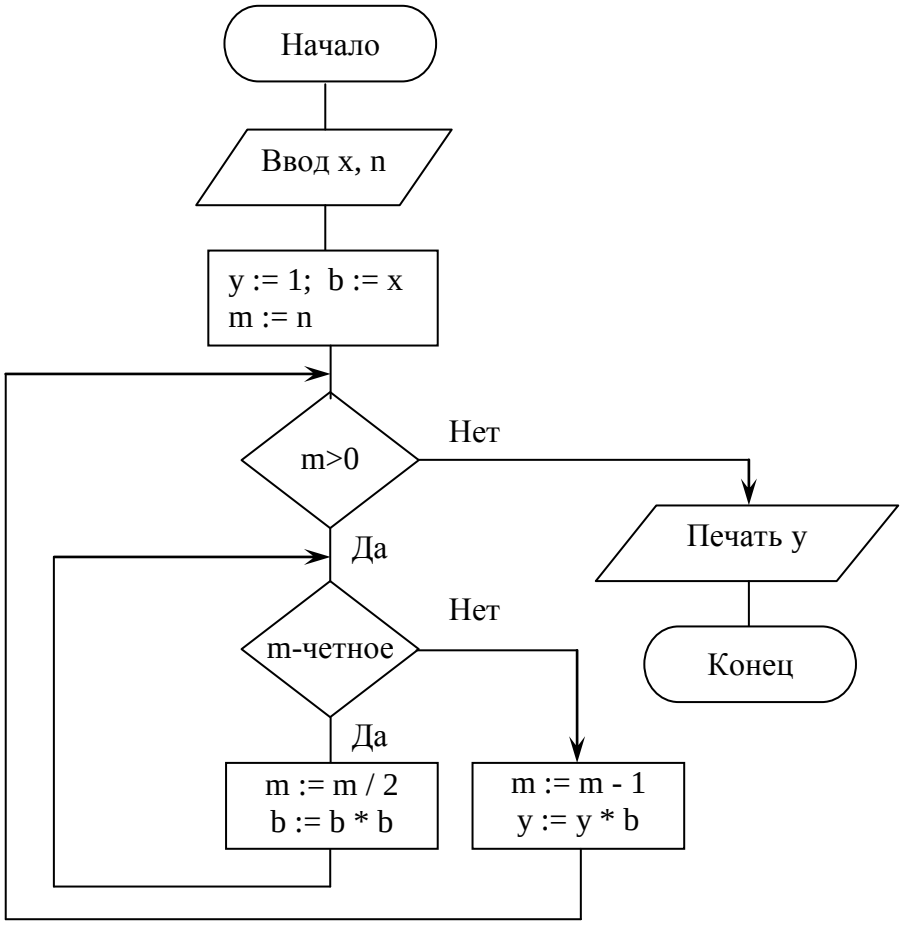
1) если m нечетное, то значение b включается в состав произведения y, после чего показатель степени m уменьшается на 1;
2) если m четное, то значение b возводится в квадрат, после чего показатель степени m уменьшается в 2 раза;
3) работа алгоритма продолжается до тех пор, пока $m > 0$.

Проиллюстрируем работу алгоритма при рассмотренном выше значении $m = 165$:

1) m нечетное $\Rightarrow y := y \cdot b = x^1; \quad m := m - 1 = 164;$
2) m четное $\Rightarrow b := b \cdot b = x^2; \quad m := m / 2 = 82;$
3) m четное $\Rightarrow b := b \cdot b = x^4; \quad m := m / 2 = 41;$
4) m нечетное $\Rightarrow y := y \cdot b = x^1 \cdot x^4; \quad m := m - 1 = 40;$
5) m четное $\Rightarrow b := b \cdot b = x^8; \quad m := m / 2 = 20;$
6) m четное $\Rightarrow b := b \cdot b = x^{16}; \quad m := m / 2 = 10;$
7) m четное $\Rightarrow b := b \cdot b = x^{32}; \quad m := m / 2 = 5;$
8) m нечетное $\Rightarrow y := y \cdot b = x^1 \cdot x^4 \cdot x^{32}; \quad m := m - 1 = 4;$
9) m четное $\Rightarrow b := b \cdot b = x^{64}; \quad m := m / 2 = 2;$
10) m четное $\Rightarrow b := b \cdot b = x^{128}; \quad m := m / 2 = 1;$
11) m нечетное $\Rightarrow y := y \cdot b = x^1 \cdot x^4 \cdot x^{32} \cdot x^{128}; \quad m := m - 1 = 0.$

Примечание. Деление на 2 осуществляется путем сдвига целочисленной двоичной переменной на один разряд вправо. Следовательно, здесь мы имеем 11 умножений, 7 операций сдвига и 3 вычитания, на что затрачивается значительно меньше машинного времени по сравнению с многократным умножением (164 операции умножения)..

Блок-схема программы:



Легко убедиться, что эта программа дает правильный результат при любом $n \geq 0$.

```
Program Exponent;
Var  m,n : word;
     x,y,b : real;
Begin
  Read(x,n);
  y:=1; b:=x; m:=n;
  While m>0 do
    Begin
      While not odd(m) do
        Begin
          m:=m div 2; b:=sqr(b);
        End;
      m:=m-1; y:=y*b;
    End;
  Writeln('y = ',y:12);
End.
```

Комментарии к программе.

1. Проверка четности целой переменной *m* возможна двумя способами:

а) с помощью функции *odd*(*m*);

б) путем вычисления логического выражения *m* ***mod*** 2 = 1.

Первый способ более эффективен по скорости выполнения и размеру объектного кода.

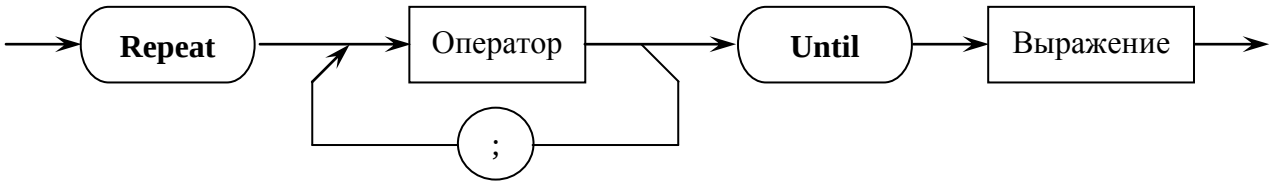
Четность двоичного числа определяется значением его последнего разряда (0 или 1). Работа функции *odd*(*m*) сводится к логическому умножению значения *m* на целочисленную константу 0000 0000 0000 0001 и анализу полученного результата (0 или 1, что соответствует машинному представлению значений *false* и *true*). Другими словами, здесь имеет место $odd(m) \equiv m\ and\ 1$. Такие действия выполняются быстрее, чем операция деления ***mod.***

2. Процедура *Writeln*(*'y=',y:12*). Печать вещественных значений в общем случае целесообразно выполнять в форме целой и дробной частей, разделенных точкой. Изображение вещественного числа в виде мантиссы и порядка предпочтительнее в тех случаях, когда диапазон возможных изменений соответствующей переменной очень большой, что имеет место в данной программе (здесь для печати всего числа отводится 12 позиций, при этом мантисса будет иметь 12 - 6 = 6 цифр).

3. В программе Exponent используется итерационный цикл. Количество его повторений зависит от значения показателя степени *n.*

ОПЕРАТОР ЦИКЛА С ПОСТУСЛОВИЕМ

Синтаксическая диаграмма:

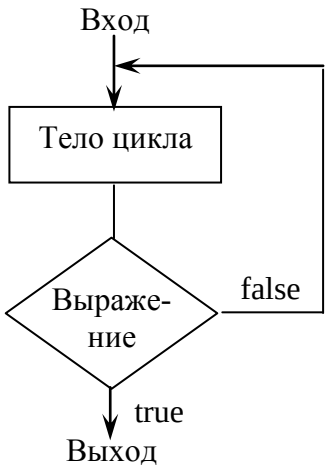


Между зарезервированными словами ***Repeat*** и ***Until*** может быть произвольное количество операторов, разделенных точкой с запятой. В частном случае здесь может быть лишь один пустой оператор. В отличие от цикла ***While*** группу операторов в теле цикла ***Repeat*** не нужно заключать в операторные скобки ***Begin … End.***

Оператор ***Repeat*** выполняется до тех пор, пока не станет истинным логическое выражение, записанное после слова ***Until***. При этом тело цикла (операторы между ***Repeat*** и ***Until***) выполняется по крайней мере один раз. Выражение после ***Until*** должно иметь тип *boolean.*

В теле цикла ***Repeat*** должно изменяться значение выражения, записанного после слова ***Until,*** таким образом, чтобы это выражение в конечном счете приняло значение *true,* в противном случае будет иметь место зацикливание программы.

Схема выполнения:



Пример 1. Алгоритм Евклида.

```
Program Evklid3;
Var  m,n,d,r : word;
Begin
  Read(m,n); Writeln('m = ',m,'  n = ',n);
  If m<n then                  { обмен значений m и n, }
    Begin                      { если  n > m           }
      r:=m; m:=n; n:=r;
    End;
  If n>0 then
    Repeat
      r:=m mod n;
      m:=n; n:=r;
    Until n=0;
  d:=m; Writeln('d = ',d)
End.
```

Если бы в программе не была предусмотрена проверка $n > 0$, то оператор ***Repeat*** выполнялся бы при любых значениях исходных данных, в том числе и при $n = 0$. В последнем случае в теле цикла имело бы место деление на нуль, что привело бы к прерыванию работы программы.

Оператор цикла с постусловием, как и оператор цикла с предусловием, в основном применяется для реализации итерационных процессов.

Пример 2. Итерационная формула Ньютона для $y = \sqrt{x}$.

```
Program Newton2;
Const  eps = 0.00001;
Var  x,y,yn : real;
Begin
  Ввод и печать x
  y:=x;
  If x>0 then
    Repeat
      yn:=y;
      y:=0.5*(yn+x/yn);
    Until abs(y-yn)<eps;
  Печать  y
End.
```

Здесь в отличие от программы Newton1, использующей оператор ***While,*** не требуется до начала цикла искусственно создавать разницу между переменными *y* и *yn*, превышающую значение *eps.*

Пример 3. Определить значение и порядковый номер наибольшего числа Фибоначчи, не превышающего заданного значения *b.*

Числа Фибоначчи получаются с помощью рекуррентного отношения

$$f_{i+2} = f_{i+1} + f_i; \quad i \geq 0; \quad f_0 = 0; \quad f_1 = 1 .$$

Имеем последовательность чисел:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

```
Program Fibonacci;
Var k,b,fi,fi1,fi2 : word;
Begin
  Read(b); Writeln('b = ',b);
  fi:=0; fi1:=1; k:=1;
  Repeat
    fi2:=fi1+fi;
    fi:=fi1; fi1:=fi2;
    k:=k+1;
  Until  fi2 > b;
  Writeln('k = ',k,'  fi = ',fi);
End.
```

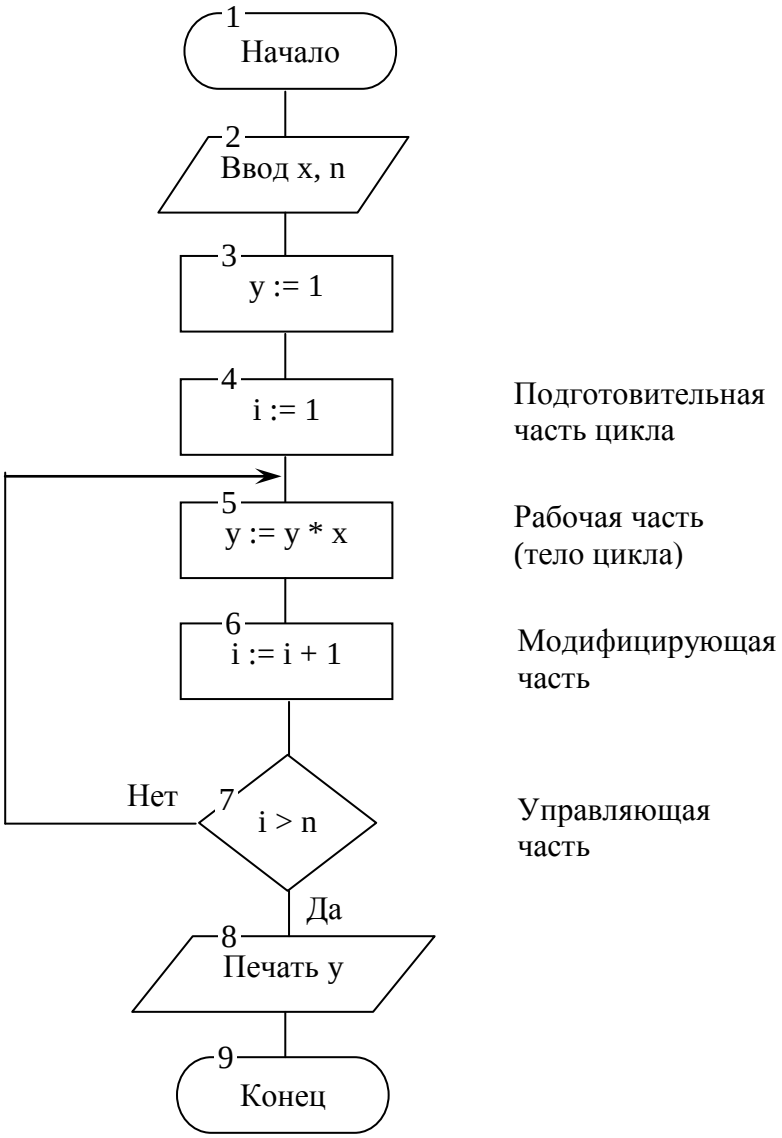
Примечание. Поскольку после окончания цикла ***While*** будет получено $fi2 > b$, а в самом теле цикла произведен перенос $fi1 \to fi$, $fi2 \to fi1$, то ближайшим наибольшим числом Фибоначчи, не превышающим заданную величину b, является значение переменной *fi.*

ОПЕРАТОР ЦИКЛА С ПАРАМЕТРОМ

С помощью такого оператора реализуются циклы, количество повторений которых заранее известно или может быть легко определено.

Пример 1. Вычислить $y = x^n$ путем многократного умножения (n - целое положительное число).

В приведенной ниже блок-схеме программы i – параметр цикла, т.е. переменная, которая управляет количеством его повторений. В подготовительном блоке параметру цикла назначается определенное начальное значение, модифицирующий блок изменяет значение параметра, управляющий блок проверяет условие окончания цикла.



Пример 2. Вычислить и отпечатать значения функции $y = e^{\sqrt{x}} \sin^2 x$ для x, изменяющегося от начального значения x_n до конечного значения x_k с шагом h_x.

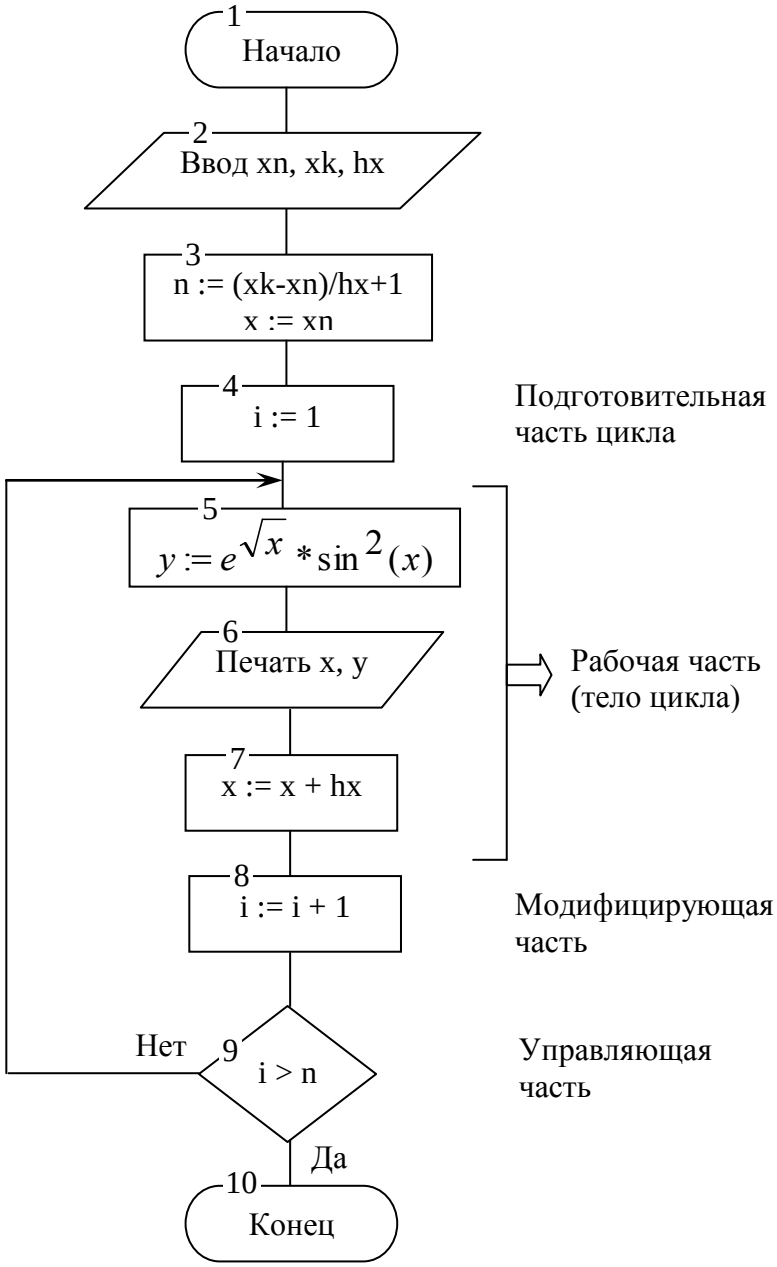
Количество вычислений функции y :

$$n = \left[\frac{x_k - x_n}{h_x} \right] + 1 ,$$

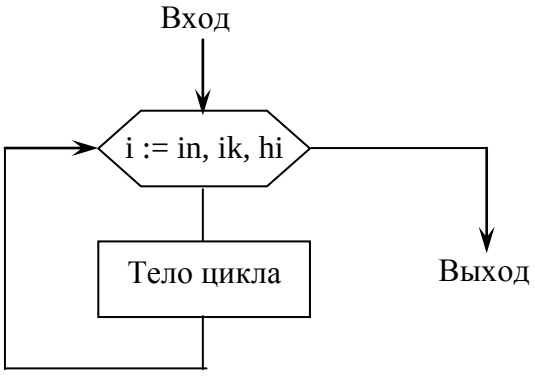
где квадратные скобки определяют целую часть вещественного числа.

В этом примере количество выполнений цикла заранее не указывается, но легко определяется по значениям исходных данных.

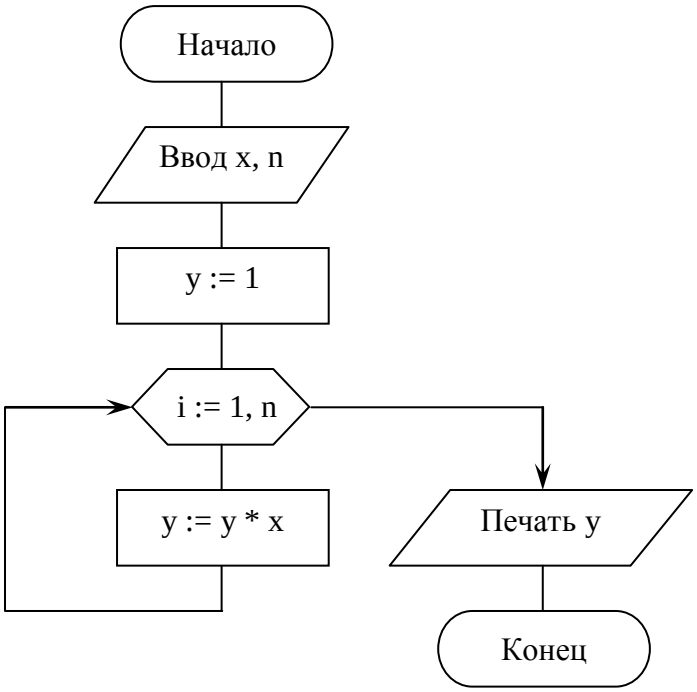
Блок-схема решения задачи:



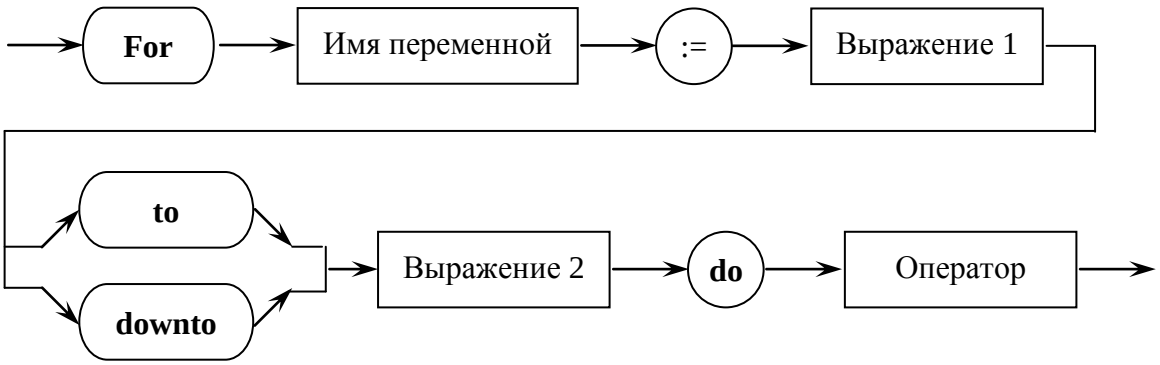
Для компактности изображения блок-схемы подготовительную, модифицирующую и управляющую части цикла рисуют одним блоком, который называют оператором цикла. На представленной ниже блок-схеме i_n, i_k, h_i - соответственно начальное значение, конечное значение и шаг изменения параметра цикла. Если $h_i = 1$, то его можно не показывать на блок-схеме.



Блок-схема для примера 1 будет иметь вид:



Синтаксическая диаграмма оператора цикла с параметром:



Имя переменной на диаграмме - это параметр цикла. Он должен иметь ординальный тип и должен быть описан в том же блоке, где находится сам оператор цикла.

Выражение 1 - начальное значение параметра, выражение 2 - его конечное значение. Эти выражения должны быть ординального типа, совместимого с типом параметра цикла.

Если в операторе цикла используется зарезервированное слово ***to,*** то шаг изменения параметра цикла равен +1; для ***downto*** этот шаг равен -1.

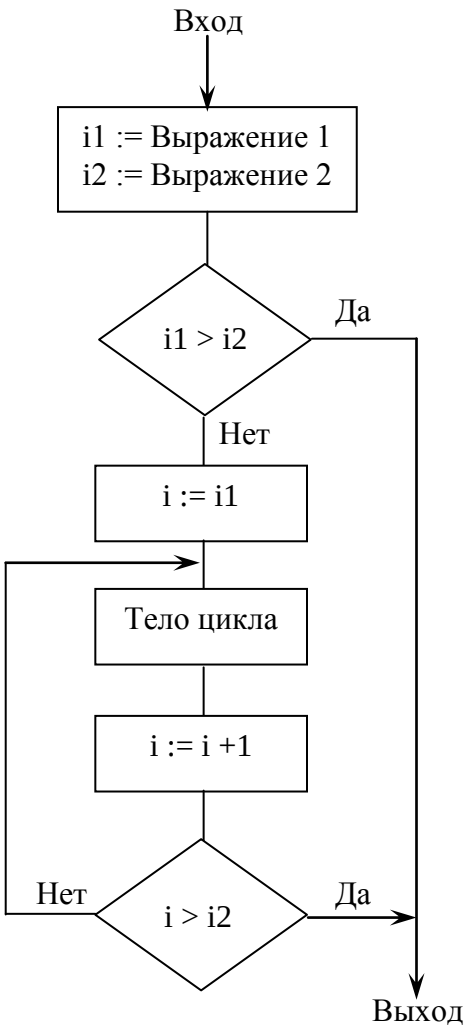
Блок "Оператор" в синтаксической диаграмме - это тело цикла. В частном случае это может быть составной или пустой оператор.

В теле цикла параметр не должен изменяться. Начальное и конечное значения параметра вычисляются только один раз, до начала цикла.

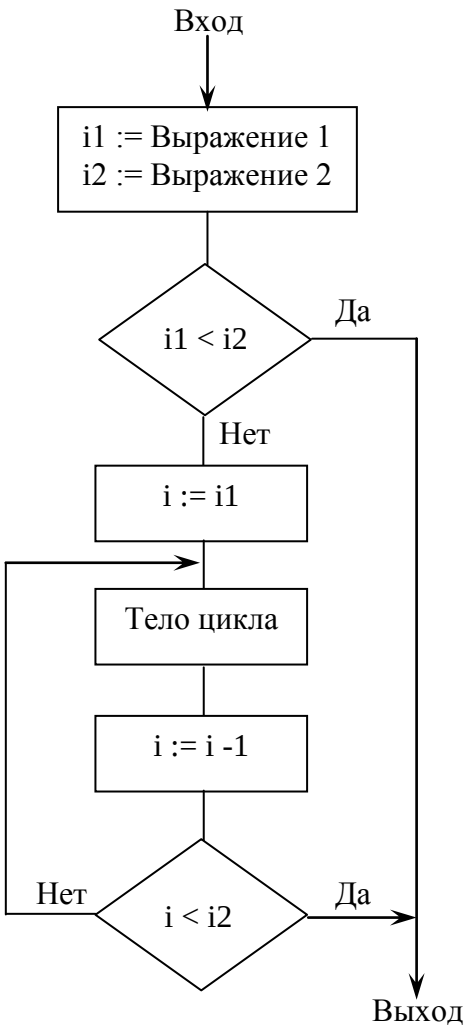
При нормальном завершении цикла значение его параметра считается неопределенным.

Работу оператора цикла можно отобразить следующей блок-схемой:

а) Вариант **to**



б) Вариант **downto**



Из блок-схемы можно вывести такие заключения:

1) если в теле цикла изменяются переменные, входящие в состав выражений 1 или 2, то это не отражается на количестве повторений цикла;

2) если начальное значение параметра цикла больше его конечного значения (вариант ***to***), то цикл ни разу не выполняется;

3) в варианте ***downto*** цикл ни разу не выполняется, если начальное значение параметра цикла меньше его конечного значения.

Пример 1а. Паскаль-программа для примера 1:

```
Program Mult;
Var  x,y : real;
     i,n : word;
Begin
  Ввод и печать  x, n
  y:=1;
  For i:=1 to n do
    y:=y*x;
  Печать y
End.
```

Пример 2а. Паскаль-программа для примера 2:

```
Program Table;
Var  x,xn,xk,hx,y : real;
     i,n : word;
Begin
  Ввод и печать  xn, xk, hx
  n:=Round((xk-xn)/hx)+1; x:=xn;
  For i:=1 to n do
    Begin
      y:=exp(sqrt(x))*sqr(sin(x));
      Печать x,y
      x:=x+hx
    End;
End.
```

Пример 3. Вычисление факториала $y=n! = 1\cdot2\cdot3\cdot\ldots\cdot n$

```
Var  i,n : byte;
       y : longint;
Begin
  Read(n);
  y:=1;
  For i:=2 to n do
    y:=y*i;
  Writeln('y = ',y)
End.
```

Пример 4. Вычисление двойного факториала $y=n!!$

Двойной факториал – это произведение четных чисел $2\cdot4\cdot6\cdot\ldots$ при четном значении переменной n или произведение нечетных чисел $1\cdot3\cdot5\cdot\ldots$ при нечетном n.

```
Var  i,n : byte;
       y : longint;
Begin
  Read(n);
  y:=1;
  For i:=2 to n do
    If odd(i) = odd(n) then
      y:=y*i;
  Writeln('y = ',y)
End.
```

Пример 5. Вычислить ряд

$$S = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{1}{9999} - \frac{1}{10000}$$

четырьмя способами:

1) последовательно слева направо;

2) последовательно справа налево;

3) слева направо отдельно положительные и отрицательные элементы, затем их вычитание;

4) то же, но справа налево.

```
Program Summa;
Var  SumLR,      { сумма, все слева направо }
```

```
      SumRL,     { сумма, все справа налево }
      SumLRNeg,  { сумма, отрицательные эл-ты, слева направо }
      SumLRPos,  { сумма, положительные эл-ты, слева направо }
      SumRLNeg,  { сумма, отрицательные эл-ты, справа налево }
      SumRLPos,  { сумма, положительные эл-ты, справа налево }
      ElemLRNeg, { очередной отриц.элемент, слева направо }
      ElemLRPos, { очередной полож.элемент, слева направо }
      ElemRLNeg, { очередной отриц.элемент, справа налево }
      ElemRLPos  { очередной полож.элемент, справа налево }
             : single;
      i : integer;       { параметр цикла }
Begin
  SumLR:=0; SumRL:=0; SumLRPos:=0; SumLRNeg:=0;
  SumRLPos:=0; SumRLNeg:=0;
  For i:=1 to 5000 do
    Begin
      ElemLRPos:=1/(2*i-1); ElemLRNeg:=1/(2*i);
      ElemRLPos:=1/(10001-2*i); ElemRLNeg:=1/(10002-2*i);
      SumLR:=SumLR+ElemLRPos-ElemLRNeg;
      SumRL:=SumRL+ElemRLPos-ElemRLNeg;
      SumLRNeg:=SumLRNeg+ElemLRNeg;
      SumLRPos:=SumLRPos+ElemLRPos;
      SumRLNeg:=SumRLNeg+ElemRLNeg;
      SumRLPos:=SumRLPos+ElemRLPos;
    End;
  Writeln('SumLR = ',SumLR:10:7);
  Writeln('SumRL = ',SumRL:10:7);
  Writeln('SumLRPos-SumLRNeg = ',(SumLRPos-SumLRNeg):10:7);
  Writeln('SumRLPos-SumRLNeg = ',(SumRLPos-SumRLNeg):10:7);
End.
```

Примечание. Для описания вещественных переменных использован тип *single* вместо типа *real*, чтобы более четко показать влияние погрешности представления вещественных значений на результат вычислений (в типе *single* мантисса имеет длину три байта вместо пяти, как в типе *real*, что определяет меньшее количество значащих цифр в представлении вещественного числа).

Программа печатает следующие результаты:

```
SumLR                0.6930732
SumRL                0.6930972
SumLRPos-SumLRNeg    0.6931019
SumRLPos-SumRLNeg    0.6930976
```

Наиболее точная здесь сумма - *SumRL*.

Разница результатов связана с тем, что при сложении вещественных чисел, представленных как числа с плавающей запятой, может происходить потеря значащих цифр, если эти числа имеют различные порядки.

Рассмотрим, как изменяется число 1/9000 при сложении его с числом 1/8998 или с числом 1/3. Мантиссу будем представлять с семью десятичными цифрами.

$$1/9000 = 0{,}0001111111 = 0{,}1111111 \cdot 10^{-3}$$
$$1/8998 = 0{,}0001111357 = 0{,}1111357 \cdot 10^{-3}$$
$$1/3 \quad = 0{,}3333333 \quad = 0{,}3333333 \cdot 10^{0}$$

а) $0,1111111 \cdot 10^{-3}$
$+\ 0,1111357 \cdot 10^{-3}$
$0,2222468 \cdot 10^{-3}$

б) $0,1111111 \cdot 10^{-3}$
$+\ 0,3333333 \cdot 10^{0}$ $\Rightarrow$ $0,0001111 \cdot 10^{0}$
$+\ 0,3333333 \cdot 10^{0}$
$0,3334444 \cdot 10^{0}$

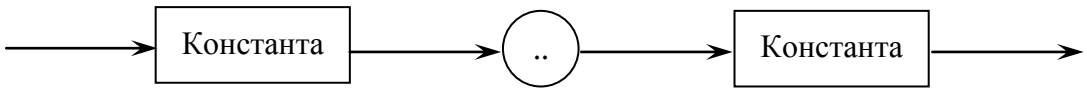
Во втором случае в первом слагаемом потеряны три значащие цифры.

В программе Summa при движении справа налево накапливаемая сумма и очередное слагаемое имеют одинаковый порядок, при движении слева направо эти порядки все более и более различаются. Поэтому при сложении справа налево практически не происходит потеря значащих цифр.

ДИАПАЗОННЫЙ ТИП ДАННЫХ

С помощью диапазонного типа программист может объявить новый ординальный тип как часть ранее определенного типа, который в этом случае называют базовым. Диапазонный тип определяется указанием наименьшего и наибольшего постоянных значений, входящих в диапазон. Переменная диапазонного типа имеет все свойства базового типа, однако ее значение на этапе выполнения программы должно принадлежать заданному диапазону.

Синтаксис:



Примеры.

```
Type  Positive = 1..MaxInt;    { базовый тип integer }
      Interval  = 100..200;    { базовый тип byte }
      Rist  = -20..20;         { базовый тип shortint }
      Parn  = 'a'..'k';        { базовый тип char }
```

В частности, объявление

```
Type  Diap = chr(0)..chr(255);
Var  ch : Diap;
```

эквивалентно объявлению

```
Var  ch : char;
```

Использование диапазонного типа повышает наглядность программы, указывая допустимые границы изменения соответствующих переменных. Еще более важным является то, что при этом появляется возможность автоматического контроля корректности присваивания переменным диапазонного типа новых значений (для этого должна быть включена директива компилятора R+, использование которой рассматривается несколько позже).

МАССИВЫ

Массив - это группа элементов, названная одним именем. Элементы массива обозначают именем массива, их местоположение в массиве определяется приписываемым к имени массива индексом.

Рассмотрим систему линейных алгебраических уравнений n-го порядка:

$$\begin{aligned} &a_{1,1}\,x_1 + a_{1,2}\,x_2 + \ldots + a_{1,n}\,x_n = b_1 \\ &a_{2,1}\,x_1 + a_{2,2}\,x_2 + \ldots + a_{2,n}\,x_n = b_2 \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &a_{n,1}\,x_1 + a_{n,2}\,x_2 + \ldots + a_{n,n}\,x_n = b_n \end{aligned}$$

В матричном виде эту систему можно записать следующим образом:

$$A\overline{B} = \overline{X}\ ,$$

где A - матрица коэффициентов, $\overline{B}$ и $\overline{X}$ - соответственно вектор свободных членов и вектор неизвестных:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,n} \end{pmatrix}; \quad \overline{B} = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix}; \quad \overline{X} = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix}$$

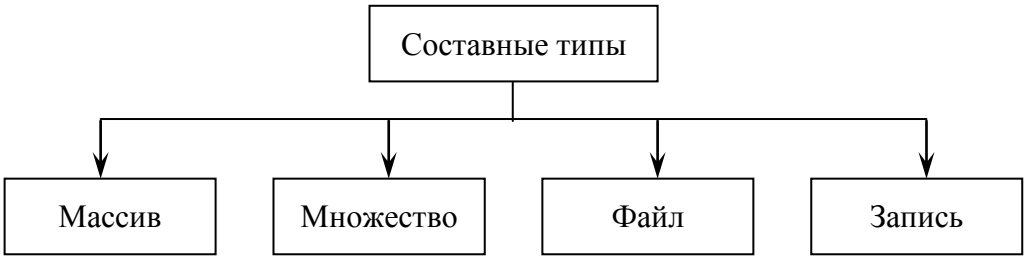
$A, \overline{B}, \overline{X}$ - это массивы. В массивах $\overline{B}$ и $\overline{X}$ положение элемента определяется одним индексом. Это одномерные массивы. В матрице A положение элемента $a_{i,j}$ определяется двумя индексами (номер строки i и номер столбца j). Это двумерный массив. В принципе возможны трехмерные, четырехмерные и т.д. массивы, однако в реальных задачах наиболее часто используются одно- и двумерные массивы, очень редко - трехмерные массивы.

Решение рассматриваемой системы уравнений можно представить в виде

$$\overline{X} = A^{-1}\overline{B}$$

Следовательно, для решения задачи нужно вначале вычислить обратную матрицу A^{-1}, а затем умножить ее на вектор $\overline{B}$. Это означает, что в программе должна выполняться обработка массивов A и $\overline{B}$.

В Паскаль-программе массивы относятся к составным типам.



По отношению к составному типу имеет смысл говорить о его структуре и типе компонентов.

Массив состоит из фиксированного количества компонентов одного и того же типа. Компонент определяется именем массива и индексом. Индекс может вычисляться, поэтому он должен описываться определенным типом. Следовательно, при описании массива необходимо указывать тип компонента и тип индекса.

Описание типа массива:

```
Type Ar=array[T1] of T,
```

где *Ar* - имя нового типа;

T1 - тип индекса, который должен быть ординальным;

T - тип компонента, который может быть любым (кроме файла).

Тип индекса, как и любое имя типа, указывает количество значений, которые может принимать индекс; в частности, это означает, что тип индекса определяет количество компонентов в массиве.

В качестве значения *T1* нельзя применять тип *integer*, так как это будет означать, что индекс изменяется от *-MaxInt-1* до *MaxInt* (-32768 .. 32767), т.е. массив имеет 65536 компонентов, для каждого из которых требуется 6 байт памяти (если считать, что компонент массива имеет тип *real*). Это превышает максимально возможный размер массива, так как любой переменной в программе на Турбо Паскале не может быть выделено свыше 64 Кбайт оперативной памяти.

Массив можно описывать непосредственно в разделе описания переменных. Например, вместо описания

```
Type  Ar = array[1..100] of integer;
Var  A,B,C : Ar;
```

можно использовать описание

```
Var  A,B,C : array[1..100] of integer;
```

Тем не менее рекомендуется в разделе ***Var*** применять только имена типов, а не описания типов. Это связано главным образом с возможностью использования описываемых массивов в процедурах и функциях.

Следует обратить внимание, что "1 .. 100" - это не границы изменения индекса, а описание диапазонного типа (хотя в данном случае это описание однозначно определяет список возможных значений индекса, другими словами, пределы его изменения). Наглядно это можно проиллюстрировать следующим примером:

```
Const  Nmax = 100;
Type  Diap = 1..Nmax;
      Ar = array[Diap] of integer;
Var  A,B,C : Ar;
```

Любой тип, как стандартный, так и нестандартный (например, тип *Ar*), указывает, как должны конструироваться переменные, имена которых перечислены в разделе ***Var***. В этом смысле удачным термином для "***Type***" является "шаблон": компилятор использует шаблон для создания отдельных экземпляров переменных данного типа. В вышеприведенном примере переменные *A, B, C* - это три отдельных экземпляра типа *Ar*, т.е. экземпляры, созданные по шаблону *Ar*.

ВВОД - ВЫВОД ОДНОМЕРНОГО МАССИВА

Ввод и вывод массива, в том числе ввод с клавиатуры и вывод на экран, можно осуществить только по-элементно.

1. Ввод массива с клавиатуры.

```
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,                { параметр цикла }
     n : word;         { текущий размер массива }
     X : Ar;           { массив }
Begin
  Read(n); Writeln('n= ',n);
  For i:=1 to n do
    Read(x[i]);
```

Каждой переменной, как простой, так и составной, при старте программы выделяется столько памяти, сколько определено описанием этой переменной в разделе ***Var***. В данном случае массиву *X* выделяется память для размещения *Nmax* = 500 элементов (500 · 6 = 3000 байт). Это поле памяти может полностью или частично использоваться при работе программы. Очевидно, что при этом должно выполняться отношение $n \le Nmax$, где *n* - текущий размер массива *X*.

Более удобной для пользователя при вводе данных с клавиатуры является такая программа, в которой перед вводом очередной переменной пользователю сообщается, что именно должно быть введено. В данном случае программа может выглядеть следующим образом:

```
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,              { параметр цикла }
     n : word;       { кол-во элементов в массиве, n ≤ N max }
     X : Ar;         { формируемый массив }
Begin
  Writeln('Введите значение n  ');
  Read(n); Writeln('n= ',n);
  For i:=1 to n do
    Begin
      Writeln('Введите значение элемента x[',i,']  ');
      Read(x[i]);
    End;
```

При отладке программы требуется производить ее многократный запуск с одними и теми же исходными данными для обнаружения и устранения синтаксических и семантических ошибок. При большом объеме исходных данных, в частности при наличии в их составе массивов, ввод данных с клавиатуры создает определенные неудобства для пользователя (при каждом запуске программы необходимо заново вводить все компоненты массива, что не исключает появление новых ошибок при вводе с клавиатуры и, как следствие, различие в результатах работы программы). Поэтому более предпочтительным является ввод данных из текстового файла. В этом случае массив формируется в файле один раз, а затем многократно используется в программе.

2. Ввод массива из текстового файла.

Файлы будут подробно рассмотрены несколько позже. Здесь же будут приведены лишь основные сведения, необходимые для организации ввода из текстового файла.

Текстовый файл, содержащий исходные данные, ничем не отличается от файла, содержащего текст Паскаль-программы. Формирование этих файлов производится одним и тем же способом. Если в такой файл записываются числовые данные, то здесь должны быть выполнены лишь два основных требования:

- числа должны иметь форму правильных констант;
- разделителем между числами являются один или несколько пробелов.

При этом не имеет значения, сколько чисел записано в каждой строке текстового файла.

Предположим, что файл на диске имеет имя *'Mas.dat'*. Программа ввода из этого файла может иметь вид:

```
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,n : word;
     X : Ar;
     F : text;      { текстовый файл }
Begin
  Assign(F,'Mas.dat');
  Reset(F);
  Read(F,n); Writeln('n= ',n);
  For i:=1 to n do
    Read(F,x[i]);
  Close(F);
```

Здесь *F* - имя текстового файла, объявленное в разделе ***Var*** (внутреннее имя файла);

Assign - процедура, устанавливающая связь между внутренним именем *F* и именем файла на диске (внешним именем);

Reset, Close - процедуры открытия и закрытия файла.

По объявлению

```
Var  F : text;
```

в программе создается файловая переменная *F*, в состав элементов которой входит, в частности, информация об имени файла на диске. Эта информация содержится в процедуре *Assign* и заносится в переменную *F* при открытии файла с помощью процедуры *Reset.* При срабатывании процедуры *Close* информация о внешнем имени файла удаляется из переменной *F*.

Внешнее имя файла формируется по правилам, принятым в операционной системе MS DOS.

В данном примере первым значением, которое вводится из файла *F*, является количество элементов *n* в массиве *X*.

В файле *Mas.dat* можно не размещать значение переменной *n* (количество элементов массива), возложив определение этого значения непосредственно на программу ввода.

```
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,n : word;
     X : Ar;
     F : text;
Begin
  Assign(F,'Mas.dat');
  Reset(F);
  n:=0;
  While not SeekEof(F) do
    Begin
      n:=n+1; Read(F,x[n]);
    End;
  Close(F);
```

В Турбо Паскале имеются две логические функции, выходным значением которых является *true* при достижении конца файла: *Eof*(*F*) и *SeekEof*(*F*) (eof - сокращение слов End

Of File). Как будет показано позже при рассмотрении текстовых файлов, при вводе числовых массивов более предпочтительна функция *SeekEof*(*F*).

В приведенной выше программе цикл ***While*** работает до тех пор, пока не будет достигнут конец файла *F*.

3. Вывод одномерного массива на экран.

Для компактности изображения массива на экране в каждой строке экрана целесообразно печатать несколько элементов массива.

Пусть формат вывода элемента массива имеет вид 8:2. Устанавливая между числами два пробела, в одной строке экрана можно разместить 8 чисел (строка экрана содержит 80 позиций). Тогда программа вывода может иметь вид:

```
Const  Nmax = 500;
Type   Ar = array[1..Nmax] of real;
Var    k : byte;
       i,n : word;
       X : Ar;
Begin
.....................
  k:=0;
  For i:=1 to n do
    Begin
      k:=k+1;
      If k<8 then
        Write(x[i]:8:2,'  ')
      Else
        Begin
          k:=0; Writeln(x[i]:8:2);
        End;
    End;
  If k>0 then Writeln;
```

Переменная *k* - это счетчик количества чисел, выводимых в одну строку экрана. Пока $k < 8$, работает процедура *Write*, размещая очередные элементы массива в одной и той же строке экрана. При $k = 8$ после вывода числа производится переход на следующую строку экрана, при этом счетчику *k* присваивается нулевое значение.

Предположим, что в массиве *X* количество элементов не кратно 8. Тогда вывод последнего элемента x_n будет осуществлен процедурой *Write*, следовательно, не будет произведен переход на новую строку экрана. Если в программе после вывода массива *X* выполняется еще вывод хотя бы одного числа, то это число будет размещено в той же строке экрана, где расположен элемент x_n, что по крайней мере неэстетично. Для предотвращения такой ситуации в вышеприведенной программе записан оператор "***If*** $k > 0$ ***then*** *Writeln*".

В дальнейшем для увеличения и уменьшения значения целой переменной будем использовать процедуры *Inc* и *Dec* (инкремент и декремент), создающие эффективные машинные коды.

Inc(k) эквивалентно k:=k+1;
Inc(k,m) эквивалентно k:=k+m;
Dec(k) эквивалентно k:=k-1;
Dec(k,m) эквивалентно k:=k-m.

В частности, *Inc*(*k*) работает примерно на 30 % быстрее, чем $k := k+1$.

ПРИМЕРЫ ОБРАБОТКИ ОДНОМЕРНЫХ МАССИВОВ

Пример 1. Вычислить среднее арифметическое значение элементов массива $X = (x_1, x_2, ..., x_n)$.

```
Program MiddleAr;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,              { параметр цикла }
     n : word;       { кол-во элементов массива }
     S : real;       { среднее арифметическое значение }
     X : Ar;         { исходный массив }
Begin
  Ввод  и  печать  n, X
  S:=0;
  For i:=1 to n do
    S:=S+x[i];
  S:=S/n;
  Печать S
End.
```

При старте программы выделяется память для всех переменных, описанных в разделе ***Var,*** т.е. каждому имени переменной ставится в соответствие адрес конкретного поля памяти. Эти поля памяти, как правило, не обнуляются, они могут быть заполнены случайными последовательностями битов. Поэтому в программе MiddleAr перед накоплением в переменной *S* суммы элементов массива *X* производится обнуление этой переменной оператором $S := 0$.

Следует обратить внимание еще на одно обстоятельство. Массиву *X* в программе выделяется память, необходимая для размещения $N_{\max} = 500$ элементов. В принципе в программе можно было бы написать:

```
For i:=1 to Nmax do
  S:=S+x[i];
S:=S/Nmax;
```

Однако это означало бы, что программа может обрабатывать лишь массив, содержащий 500 элементов, но не 499 или 100. Поэтому для обеспечения универсальности работы программы обработке подвергается текущее количество элементов *n* в предположении, что $n \le N_{\max}$.

Пример 2. Вычислить среднее арифметическое значение положительных элементов массива $X = (x_1, x_2, ..., x_n)$.

```
Program MiddlePos;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,              { параметр цикла }
     k,              { кол-во положительных элементов массива }
     n : word;       { общее кол-во элементов массива }
     S : real;       { среднее арифметическое значение }
     X : Ar;         { исходный массив }
Begin
  Ввод  и  печать  n, X
  S:=0; k:=0;
  For i:=1 to n do
    If x[i]>0 then
```

```
      Begin
        S:=S+x[i]; Inc(k)
      End;
  If k>0 then       { блокировка деления на нуль }
    S:=S/n;
  Печать S
End.
```

В частном случае в массиве *X* может не быть ни одного положительного элемента (*k*=0). Тогда при отсутствии оператора «**If** x[i]>0 **then**» было бы прерывание программы с сообщением «*Zero divide*» («Деление на нуль»).

Пример 3. Определить $y = \max(x_1, x_2, ..., x_n)$.

```
Program MaxElem;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i, n : word;
     Xmax : integer;
     X : Ar;
Begin
  Ввод и печать n,X
  Xmax:=x[1];
  For i:=2 to n do
    If x[i]>Xmax then
      Xmax:=x[i];
  Печать Xmax
End.
```

Пример 4. Обменять местами максимальный и минимальный элементы массива *X*.

Здесь требуется найти не только значения переменных *Xmax* и *Xmin*, но и положение (индексы) соответствующих им элементов в массиве *X* (значения переменных *imax* и *imin*).

Вариант 1.

```
Program Exchange;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,n,imax,imin : word;
     Xmax,Xmin : integer;
     X : Ar;
Begin
  Ввод и печать n,X
  Xmax:=x[1]; Xmin:=x[1];
  imax:=1; imin:=1;
  For i:=2 to n do
    Begin
      If x[i]>Xmax then
        Begin
          Xmax:=x[i]; imax:=i
        End;
      If x[i]<Xmin then
        Begin
          Xmin:=x[i]; imin:=i
        End;
    End;
  x[imax]:=Xmin; x[imin]:=Xmax;
```

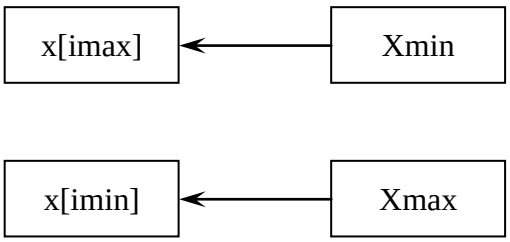
```
  Печать X
End.
```

В каждом цикле элемент x_i сравнивается со значениями переменных *Xmax* и *Xmin.* В то же время очевидно, что если выполняется условие $x_i > Xmax$, то проверка отношения $x_i < Xmin$ является излишней. Указанное замечание учтено в варианте 2.

Вариант 2 (фрагмент).

```
  For i:=2 to n do
    If x[i]>Xmax then
      Begin
        Xmax:=x[i]; imax:=i
      End
    Else
      If x[i]<Xmin then
        Begin
          Xmin:=x[i]; imin:=i
        End;
```

При обмене значений элементов *x*[*imax*] и *x*[*imin*] использовано то обстоятельство, что после окончания обработки массива известны не только значения индексов *imax* и *imin,* но и численные значения переменных *Xmax* и *Xmin.* Обмен в данном случае осуществляется по такой схеме:



Примечание. Было бы совершенно неправильно выполнять обмен максимального и минимального элементов массива *X* операторами

```
R := Xmax; Xmax := Xmin; Xmin:= R;
```

(*R* - переменная типа *real*), так как здесь обмениваются значения переменных *Xmax* и *Xmin*, а не значения элементов массива *X* с индексами *imax* и *imin.*

Пример 5. В массиве *X* найти значение и положение первого отрицательного элемента.

Решение рассматриваемой задачи выполнено в двух вариантах.

Вариант 1.

```
Program NegElem;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,n,ineg : word;
     Xneg : integer;
     X : Ar;
Begin
  Ввод и печать n,X
  ineg:=0;
  For i:=1 to n do
    If x[i]<0 then
      Begin
        Xneg:=x[i]; ineg:=i;
```

```
        Break
      End;
  Writeln('ineg= ',ineg);
  If ineg>0 then
    Writeln('Xneg= ',Xneg);
End.
```

В программе последовательно просматриваются элементы массива *X* и, как только обнаруживается отрицательный элемент, в переменных *Xneg, ineg* запоминаются значение и индекс этого элемента, после чего с помощью процедуры *Break* производится принудительный выход за пределы цикла.

Если в массиве *X* нет ни одного отрицательного элемента, в программе будет отпечатано *ineg* = 0, в противном случае - конкретные значения *ineg, Xneg.*

Примечание. Процедура Break эквивалентна оператору **Goto** Met, где Met – метка после окончания цикла **For.**

Вариант 2 (фрагмент).

```
ineg:=0; i:=1;
While (i<=n) and (ineg=0) do
  If x[i]<0 then
    Begin
      Xneg:=x[i]; ineg:=i;
    End
  Else
    Inc(i);
Writeln('ineg= ',ineg);
If ineg>0 then
  Writeln('Xneg= ',Xneg:12);
```

Вариант 1 работает примерно в 2 раза быстрее (отсутствие проверки *ineg* = 0 в каждом цикле).

Пример 6. В массиве *X* определить местоположение последнего положительного элемента.

Вариант 1.

```
Program PosElem;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,n,ipos : word;
     X : Ar;
Begin
  Ввод и печать n,X
  ipos:=0;
  For i:=1 to n do
    If x[i]>0 then
      ipos:=i;
  Writeln('ipos= ',ipos);
End.
```

В программе просматриваются все элементы массива *X,* что нерационально с точки зрения затрат машинного времени. Более эффективной является программа варианта 2, в которой просмотр массива *X* осуществляется справа налево до обнаружения ближайшего положительного элемента.

Вариант 2 (фрагмент).

```
  ipos:=0;
```

```
For i:=n downto 1 do
  If x[i]>0 then
    Begin
      ipos:=i; Break
    End;
Writeln('ipos= ',ipos);
```

Пример 7. Вычисление значения полинома.

Полином (многочлен)

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$$

целесообразно всегда вычислять по схеме Горнера:

$$P_n(x) = (...(a_n x + a_{n-1})x + a_{n-2})x + ... + a_1)x + a_0 .$$

Схема Горнера обладает следующими преимуществами по сравнению с обычной записью полинома:

- меньшее количество операций умножения;
- меньшая погрешность вычислений;
- удобство реализации вычислений в виде цикла.

Для вычисления полинома по обычной схеме требуется $2n-1$ операций умножения, по схеме Горнера - n таких операций.

Вещественные числа, являющиеся в реальных задачах результатами измерения каких-либо величин, всегда обладают определенной погрешностью. Следовательно, результат обработки таких чисел также содержит в себе некоторую погрешность.

В теории ошибок доказывается, что погрешность произведения равна сумме погрешностей сомножителей. Поэтому схема Горнера, требующая вдвое меньше операций умножения по сравнению с обычной схемой, обеспечивает более высокую точность вычислений.

Вычисление значения полинома по схеме Горнера легко организовать в цикле, если представить коэффициенты полинома как массив.

Вычисления в программе организуются по следующей методике:

```
P := a[n]
P := P * x + a[n-1]
P := P * x + a[n-2]
.....................
P := P * x + a[1]
P := P * x + a[0]
```

```
Program Gorner;
Const  Nmax = 400;
Type  Koef = array[0..Nmax] of real;
Var  i,n : integer;
     P,x : real;
     A : Koef;
Begin
  Ввод и печать n,A,x
  P := a[n];
  For i:=n-1 downto 0 do
    P:=P*x+a[i];
  Печать P
End.
```

Конкретную запись полинома, заданного в виде явного выражения, также рекомендуется вычислять по схеме Горнера. Например, полином

$$y = 8x^5 - 4.1x^4 + 2.7x^2 - 3.6x + 1$$

по схеме Горнера имеет вид:

$$y = (((8x - 4.1)x + 2.7)x \cdot x - 3.6)x + 1.$$

Пример 8. Расположить элементы массива X в обратном порядке.

При выполнении заданного преобразования нужно произвести обмен значений элементов x_1 и x_n, x_2 и x_{n-1}, x_3 и x_{n-2} и так далее до достижения "середины" массива X.

Вариант 1.

```
Program BackOrder;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,n : integer;
     R : real;
     X : Ar;
Begin
  Ввод и печать  n,  X
  For i:=1 to n div 2 do
    Begin
      R:=x[i]; x[i]:=x[n-i+1]; x[n-i+1]:=R
    End;
  Печать массива X
End.
```

Вариант 2 (фрагмент).

```
  i:=1; j:=n;
  While i<j do
    Begin
      R:=x[i]; x[i]:=x[j]; x[j]:=R;
      Inc(i); Dec(j);
    End;
```

Оба варианта примера 8 равнозначны с точки зрения эффективности вычислений, но вариант 2 более нагляден. Второй вариант более удобен также в случае, когда требуется выполнить перестановку не всего массива, а подмассива, например, с номера *n1* до номера *n2*. Тогда перед циклом записываются операторы `i:=n1` и `j:=n2`, остальная часть программы не изменяется.

Вариант 3. Использование буферного массива.

```
Program BackOrder3;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,n : integer;
     X,Y : Ar;
Begin
  Ввод и печать  n,  X
  For i:=1 to n do
    y[n-i+1]:=x[i];
  X:=Y;
  Печать массива X
End.
```

Оператор X:=Y в варианте 3 допустим, поскольку массивы X иY описаны одним и тем же именем типа. При этом производится пересылка всего поля Y в поле X, т.е. Nmax элементов в соответствии с объявлением типа данного массива.

Недостаток варианта 3 вполне очевиден – это использование буферного массива.

ОГРАНИЧЕНИЯ В ИСПОЛЬЗОВАНИИ ОПЕРАТОРА FOR

Ограничения в использовании оператора цикла с параметром естественным образом вытекают из приведенного ранее описания этого оператора. Рассмотрим еще раз по пунктам описание оператора ***For*** и вытекающие из этого следствия.

1. Параметр цикла должен быть ординального типа.

Параметр цикла определяет в конечном счете порядковый номер повторения данного цикла; следовательно, он имеет дискретный характер и должен быть описан одним из ординальных типов. Если в программе описать параметр цикла как вещественную переменную, то при трансляции будет выдано сообщение

Error 97: Invalid FOR control variable
(неправильный параметр цикла FOR).

2. Параметр должен быть описан в том же блоке, где находится сам оператор цикла.

Нарушение этого требования может привести к серьезным ошибкам при использовании подпрограмм и будет рассмотрено при изучении процедур и функций.

3. Выражения, определяющие начальное и конечное значения параметра ("Выражение 1" и "Выражение 2"), **должны быть ординального типа, совместимого с типом самого параметра цикла.**

Параметру цикла перед выполнением цикла присваивается значение выражения 1, в конце каждого цикла текущее значение параметра сравнивается со значением выражения 2, что и определяет требование совместимости типов параметра цикла и значений выражений 1 и 2. В случае нарушения данного требования при трансляции программы генерируется сообщение

Error 26: Type mismatch (несоответствие типов).

4. Шаг изменения параметра цикла равен +1 или -1.

Если в программе шаг изменения параметра должен отличаться от значений +1 и -1, то нужно применить один из следующих способов:

- вместо оператора ***For*** использовать операторы ***While*** или ***Repeat;***
- в цикле ***For*** задать соответствующий алгоритм изменения индексов.

Пример 1. Для массива $X = (x_1, x_2, ..., x_n)$ просуммировать отдельно элементы с четными и нечетными индексами.

```
Type  Ar = array[1..100] of real;
Var   i,n : byte;
      S1,S2 : real;
      X : Ar;
Begin
  Ввод  n, X
```

Вариант 1:

```
  S1:=0; S2:=0; i:=2;
```

```
While i<=n do
  Begin
    S1:=S1+x[i-1]; S2:=S2+x[i];
    Inc(i,2);
  End;
If odd(n) then
  S1:=S1+x[n];
```

Вариант 2:

```
S1:=0; S2:=0;
For i:=1 to n div 2 do
  Begin
    S1:=S1+x[2*i-1]; S2:=S2+x[2*i];
  End;
If odd(n) then
  S1:=S1+x[n];
```

В каждом цикле обрабатываются два элемента массива *X*. Если их количество *n* нечетное, то последний элемент *x*[*n*] добавляется в сумму *S1* после окончания работы цикла.

Вариант 1 наиболее универсален в том смысле, что пригоден при любом шаге перебора элементов массива. Если же шаг равен 2, то для данной задачи наиболее эффективным является вариант 3.

Вариант 3:

```
S1:=0; S2:=0;
For i:=1 to n do
  If odd(i) then
    S1:=S1+x[i]
  Else
    S2:=S2+x[i];
```

5. В теле цикла параметр не должен изменяться.

Изменение параметра при каждом выполнении цикла непосредственно производят машинные команды, реализующие оператор ***For.*** Попытка изменения параметра в теле цикла зачастую приводит к получению неправильных результатов, в ряде случаев - к зацикливанию программы.

Пример 2. Определить индекс первого отрицательного элемента в массиве $X = (x_1, x_2, ..., x_n)$.

```
Type  Ar = array[1..100] of real;
Var   i,k,n : byte;
      X : Ar;
Begin
  Ввод  n, X
  k:=0;
  For i:=1 to n do
    If x[i]<0 then
      Begin
        k:=i; i:=n+1
      End;
  Writeln('k=',k);
```

В данном случае для принудительного завершения цикла использован оператор *i*:=*n*+1 вместо оператора ***Goto***. Результат - зацикливание программы.

6. Начальное и конечное значения параметра цикла вычисляются только один раз, до начала цикла.

Следовательно, изменение выражений 1 или 2 в теле цикла никакого влияния на его работу не оказывает.

Вариант предыдущего примера (фрагмент):

```
k:=0;
For i:=1 to n do
  If x[i]< 0 then
    Begin
      k:=i; n:=0
    End;
```

Здесь будет найден не первый, а последний отрицательный элемент. Кроме того, будет «испорчена» переменная *n*, определяющая текущий размер массива.

7. При нормальном завершении цикла значение его параметра считается неопределенным.

Способ реализации оператора ***For*** зависит от системы машинных команд конкретного типа ЭВМ. В зависимости от способа реализации параметр цикла *i* после завершения работы оператора

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

может иметь значение или *n*, или *n*+1. Поэтому использовать конечное значение параметра цикла при нормальном (не принудительном) его завершении не рекомендуется, поскольку это может привести к получению неправильных результатов.

Пример некорректного использования параметра цикла:

```
For i:=1 to n do
  S:=S+x[i];
k:=i+1;
```

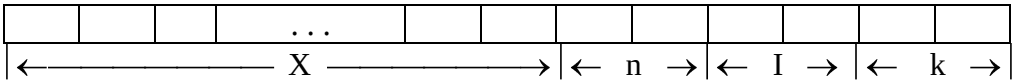
КОНТРОЛЬ ОРДИНАЛЬНЫХ ПЕРЕМЕННЫХ

Пусть мы имеем следующее описание:

```
Const  Nmax = 100;
Type   Diap = 1..Nmax;
       Ar = array[Diap] of real;
Var    X : Ar;
       n,i,k : integer;
```

Ранее уже указывалось, что при старте программы всем переменным (простым и составным), перечисленным в разделе ***Var,*** выделяется память, размер которой зависит от типа переменной. Поля памяти выделяются в порядке перечисления переменных в разделе ***Var.*** В данном случае имеем

1 2 3 599 600 601 602 603 604 605 606



На рисунке байты, входящие в состав полей *X, n, i* и *k*, пронумерованы последовательно. Условно можно считать, что номер байта - это его адрес в памяти.

Адрес произвольного элемента x_i в машинной программе вычисляется следующим образом:

$$Addr(x_i) = X + (i - 1) \cdot SizeOf(real),$$

где *X* - адрес поля *X* (адрес его крайнего левого байта),
SizeOf(*real*) - размер поля памяти типа *real.*

Рассмотрим два варианта фрагмента программы:

```
1.  n:=100; k:=500;
    x[101]:=5;
    Writeln('n= ',n,'  k= ',k);
2.  n:=100; k:=500;
    i:=101; x[i]:=5;
    Writeln('n= ',n,'  k= ',k);
```

В приведенном выше описании индекс массива имеет диапазонный тип $1..N\max \equiv 1..100$. При трансляции для индекса массива отводится некоторое поле памяти и указывается тип переменной, которая размещается в этом поле (в данном случае диапазонный тип *Diap*). Придадим условно индексу массива имя *Ind*. Тогда выражения *x*[*i*], *x*[*k*], *x*[*n*] означают, что переменной *Ind* присваиваются соответственно значения *i, k* или *n*, т.е. имеет место присваивание ординальной переменной нового значения.

В вариантах 1 и 2 числовое значение 5 должно записываться в поле памяти, соответствующее адресу 101-го элемента массива *X,* т.е. в байты 601 .. 606. При этом значения переменных *n, i* и *k* будут испорчены, что в дальнейшем может привести к непредсказуемым последствиям при работе программы.

Указанную выше ошибку, которая заключается в том, что индекс элемента массива принимает значение вне допустимого диапазона 1 .. *Nmax*, называют выходом за границы массива.

Для оператора *x*[101] := 5 информация о том, что индекс элемента массива имеет недопустимое значение, известна по исходному тексту программы, в связи с чем данная ошибка определяется на этапе трансляции. При этом на экран выводится сообщение

Error 76 Const out of range
(Константа нарушает границы)

Аналогичная ошибка для оператора *x*[*i*] := 5 в варианте 2 не может быть обнаружена при трансляции программы, поскольку конкретное значение переменной *i* становится известным лишь при работе программы. В частности, при отработке второго варианта фрагмента программы оператор *Writeln*('*n= ',n,' k=',k*); выдаст на печать значения

```
n = 131    k = 8192.
```

Примечание. Значение числа 5 в формате ***real*** на машинном уровне имеет вид 200000000083. Здесь первые два байта воспринимаются как значение переменной *k*: $2000_{16}= 8192_{10}$; следующие два байта – переменной *i*, последние два байта – переменной *n*: $83_{16}=131_{10}$.

Для индикации ошибки выхода за границы массива при работе программы используют директиву компилятора R (от слова Range - граница). При включенной директиве R компилятор записывает в машинную программу дополнительные команды, которые проверяют корректность присваивания новых значений всем ординальным переменным, в том числе и индексам массивов. При обнаружении некорректного присваивания работа программы прерывается, а на экран выдается сообщение

Error 201 Range check error
(Ошибка при проверке границ)

Индекс элемента массива может быть лишь ординального типа. Его допустимые значения определяются описанием типа массива. В приведенном выше примере индекс элемента массива *X* может принимать значения лишь в диапазоне от 1 до 100. Следовательно, при выходе индекса массива за допустимые границы и включенной директиве *R* будет также сгенерировано прерывание программы с выводом сообщения об ошибке 201.

Пусть мы имеем описание

```
Var  k,l : byte;
     m,n : integer;
     p,q : 1..5000;
```

и два фрагмента программы:

```
1.  k:=400; l:=-1;
    p:=0;   q:=6000;
2.  m:=400; n:=-1;
    k:=m;   l:=n;
    m:=0;   n:=6000;
    p:=m;   q:=n.
```

Ошибка присваивания переменным *k, l, p* и *q* недопустимых значений в фрагменте 1 обнаруживается на этапе трансляции, в фрагменте 2 - при работе программы, если включена директива *R* (пример использования директивы *R* показан в следующем разделе).

При включенной директиве *R* за счет добавления в программу дополнительных машинных команд увеличиваются объем программы и время ее выполнения. Поэтому директиву *R* рекомендуется использовать только при отладке программы, а после окончания отладки удалять ее из состава программы.

ВСТАВКА ЭЛЕМЕНТА В УПОРЯДОЧЕННЫЙ МАССИВ

Методику вставки дополнительного элемента в массив, упорядоченный по возрастанию или по убыванию, рассмотрим на конкретном примере.

Пример.

Задан упорядоченный по возрастанию массив $X = (x_1, x_2, ..., x_n)$ и произвольное число b. Вставить число b в массив X, не нарушая упорядоченности этого массива.

При этом может иметь место один из трех вариантов:

а) $b < x_1$;

б) $x_1 \le b \le x_n$;

в) $b > x_n$.

Пусть массив X имеет вид

$$5\ \ 12\ \ 18\ \ 22\ \ 29\ \ 31\ \ 35\ \ 41\ \ 47\ \ 53\ \ 62\ \ 77\ \ \ (n=12)$$

Если $b=2$, т.е. меньше первого элемента x_1, то требуется сдвинуть весь массив вправо на один элемент, а элементу x_1 присвоить значение b. Если $b=32$, т.е. $x_1<b<x_n$, то определяется индекс k ближайшего элемента $x_k>b$, после чего подмассив $x_k, x_{k+1},...,x_n$ сдвигается вправо на один элемент, а элементу x_k присваивается значение b. Если $b=100$, т.е. $b>x_n$, то сдвиг элементов массива не производится, а значение переменной b присваивается элементу x_{n+1}. В любом случае при включении значения b в состав массива X количество элементов n этого массива увеличивается на 1.

```
{$R+}
Program Insertion;
Const  Nmax = 100;
Type   Ar = array[1..Nmax] of integer;
Var    X : Ar;
       i,k,n : integer;
Begin
  Ввод и печать  n, X, b
  k:=0;
  For i:=1 to n do          { Поиск ближайшего }
    If x[i]>b then          { элемента, большего }
      Begin                 { значения b}
        k:=i; Break
      End;
  If k=0 then               { b >= x[n] }
    x[n+1]:=b
  Else
    Begin                   { Сдвиг подмассива }
      For i:=n+1 downto k+1 do  { при b<x[n]}
        x[i]:=x[i-1];
      x[k]:=b               { Вставка значения b }
    End;
  Inc(n);         { Увеличение размера массива }
  Печать  n, X
End.
```

Примечание 1. Сдвиг части массива *X* вправо оператором

```
For i:=k+1 to n+1 do
  x[i]:=x[i-1];
```

путем перебора его элементов слева направо делать нельзя, так как всем элементам x_{k+1}, x_{k+2}, ... , x_{n+1} будет присвоено одно и то же значение элемента x_k.

Примечание 2. Фраза {$R+} означает включение директивы компилятора R.

Примечание 3. Выход за пределы границ массива *X* будет иметь место при *n = Nmax*. Следствием этого может быть получение неправильных результатов или зацикливание программы. При включенной директиве R в этом случае будет сгенерировано прерывание 201, что позволит индицировать такую ошибку при отладке программы. Для повышения надежности работы программы целесообразно объявление типа массива выполнить в виде

```
Ar = array[1..Nmax+1] of integer,
```

а после ввода массива *X* проверить условие $n \le Nmax$ и в случае его нарушения выдать соответствующее сообщение на экран.

УДАЛЕНИЕ ЭЛЕМЕНТОВ ИЗ МАССИВА

Чтобы удалить элемент x_k из массива $X = (x_1, x_2, ..., x_n)$, необходимо сдвинуть элементы $x_{k+1}, x_{k+2}, ..., x_n$ на один "шаг" влево и уменьшить значение n на 1. Если $k = n$, то в этом случае достаточно выполнить $n := n - 1$.

Пример 1. Из массива *X* удалить минимальный элемент.

```
Program DelMin;
Const  Nmax = 200;
Type  Ar = array[1..Nmax] of integer;
Var  i,imin,n : byte;
     xmin : integer;
     X : Ar;
Begin
  В в о д  n, X
  Xmin:=x[1]; imin:=1;      { Определение положения }
  For i:=2 to n do          { (индекса) минимального}
    If x[i]<xmin then       { элемента }
      Begin
        Xmin:=x[i]; imin:=i
      End;
  For i:=imin to n-1 do     { Удаление элемента }
   x[i]:=x[i+1];            { с индексом imin }
  Dec(n);                   { Уменьшение размера массива }
  П е ч а т ь  n, X
End.
```

Если *imin* = *n*, то второй цикл ***For*** не работает (начальное значение параметра *i* больше его конечного значения), но размер массива n уменьшается на 1.

Пример 2. Из массива *X* удалить все нулевые элементы.

В программе будем последовательно просматривать элементы массива и, если очередной элемент $x_i = 0$, то производим сдвиг подмассива $x_{i+1} .. x_n$ на один элемент влево, одновременно уменьшая количество элементов n.

```
Program Mas;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,j,n : integer;
     X : Ar;
Begin
  Ввод  n, X
  For i:=1 to n do
    If x[i]=0 then
      Begin
        For j:=i to n-1 do
          x[j]:=x[j+1];
        Dec(n);
      End;
  Вывод  n, X
End.
```

По поводу программы Mas можно сделать три замечания.

1. При *i* = *n*, когда анализируется последний элемент массива, параметр *j* изменяется от *n* до *n*-1, т.е. начальное значение параметра цикла больше его конечного значения. Как известно из описания оператора ***For***, цикл в этом случае не выполняется, т.е. оператор цикла эквивалентен пустому оператору. Следовательно, при $x_n = 0$ происходит лишь уменьшение значения *n*, что соответствует алгоритму решения задачи.

2. Начальное и конечное значения параметра цикла вычисляются до начала цикла и в процессе его выполнения не изменяются. Следовательно, уменьшение значения переменной *n* при $x_i = 0$ не изменяет количество повторений цикла по параметру *i*. Это приводит к избыточной работе программы, но не отражается на правильности решения задачи.

3. Предположим, что в массиве *X* имеются подряд идущие нулевые элементы x_k и x_{k+1}. При *i* = *k* будет удален элемент x_k, а на его место перемещается элемент x_{k+1}, также равный нулю. Поскольку при новом повторении цикла ***For*** параметр цикла принимает очередное значение *k*+1, то новый нулевой элемент x_k не будет анализироваться повторно и, как следствие, не будет удален из массива. Поэтому при наличии в массиве подряд идущих нулевых элементов программа Mas работает неправильно.

Для корректного решения поставленной задачи нужно, чтобы программа после удаления нулевого элемента x_k повторно анализировала этот же элемент и переходила к рассмотрению элемента x_{k+1} лишь при $x_k \neq 0$. Для этого нужно в программе вместо цикла ***For*** использовать цикл ***While:***

```
Program DelZero1;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,j,n : integer;
     X : Ar;
Begin
  Ввод n, X
  i:=1;
  While i<=n do
    If x[i]=0 then
      Begin
        For j:=i to n-1 do
          x[j]:=x[j+1];
        Dec(n);
      End
    Else
      Inc(i);
End.
```

Тот же эффект можно достичь с помощью оператора ***For,*** если просмотр массива выполнять справа налево.

```
Program DelZero2;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,j,n : integer;
     X : Ar;
Begin
  Ввод n, X
  For i:=n downto 1 do
    If x[i]=0 then
      Begin
```

```
        For j:=i to n-1 do
          x[j]:=x[j+1];
        Dec(n);
      End;
End.
```

Обнаружение нулевого элемента во внешнем цикле и, как следствие, его удаление из массива приводит к перемещению уже обработанных элементов в "хвосте" массива и не влияет ни на количество повторений внешнего цикла, ни на анализ оставшихся элементов.

Рассмотрим еще один вариант программы, предназначенной для удаления нулевых элементов из массива.

```
Program DelZero3;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,k,n : integer;
     X : Ar;
Begin
  Ввод n, X
  For i:=1 to n do
    If x[i]<>0 then
      Begin
        Inc(k);
        If k<>i then
          x[k]:=x[i];
      End;
  n:=k;
End.
```

Компьютерный эксперимент, проведенный по отношению к программам удаления однотипных элементов из массива (в данном случае нулей), показал, что быстродействие (время обработки одного и того же массива) программ DelZero1, DelZero2 и DelZero3 пропорционально численным значениям 7, 3 и 1.

Рассмотрим причины столь существенного различия в быстродействии приведенных здесь программ.

1) Цикл **For** в общем случае работает в 2 – 2,5 раза быстрее, чем цикл **While** (это объясняется в основном тем, что в цикле **For** используются более эффективные машинные команды).

2) В программе DelZero3 перемещается лишь один элемент массива (и то не в каждом цикле), в то время как в программе DelZero2 при удалении каждого элемента, кроме последнего, передвигается вся правая часть массива, причем чем ближе удаляемый элемент к началу массива, тем больше элементов перемещается справа налево.

Примечание. Удаление подряд расположенных элементов, т.е. подмассива рассмотрено в прил.1 (программа Task102).

«ШКОЛЬНЫЙ» АЛГОРИТМ СОРТИРОВКИ

Сортировка массива (ее также называют группировкой) заключается в такой перестановке элементов массива, при которой они будут расположены в заданном порядке – по возрастанию (группировка по возрастанию) или по убыванию (группировка по убыванию). В первом случае между элементами массива имеет место соотношение

$$x_1 \le x_2 \le x_3 \le ... \le x_n;$$

во втором случае - соотношение

$$x_1 \ge x_2 \ge x_3 \ge ... \ge x_n.$$

В дальнейшем, рассматривая различные методы группировки, речь будет идти о группировке по возрастанию.

На начальном этапе изучения программирования, в частности в школьном курсе информатики программу сортировки одномерного массива обычно представляют в следующем виде:

```
For i:=1 to n do
  For j:=1 to n do
    If x[i]>x[j] then
      Begin
        R:=x[i]; x[i]:=x[j];
        x[j]:=R
      End;
```

Это самая простая из возможных программ сортировки, однако одновременно следует отметить, что более худшего метода сортировки еще не придумано.

Недостатки школьного алгоритма сортировки:

1. Дважды проверяется каждая пара элементов (например, при $i = 5$ и $j = 8$ проверяются элементы x_5 и x_8; при $i = 8$ и $j = 5$ — те же элементы x_5 и x_8).

2. При $i = j$ производится ненужная проверка одного и того же элемента.

3. Алгоритм совершенно не учитывает исходное расположение элементов массива. Это означает, что время обработки массива, элементы которого уже были расположены по возрастанию, будет примерно таким же, как и в случае, когда элементы массива расположены в обратном порядке, т.е. по убыванию.

Избавиться от первых двух недостатков довольно легко. Для этого алгоритм сортировки запишем в следующем виде:

```
For i:=1 to n-1 do
  For j:=i+1 to n do
    If x[i]>x[j] then
      Begin
        R:=x[i]; x[i]:=x[j];
        x[j]:=R
      End;
```

Назовем этот вариант модифицированным школьным алгоритмом сортировки.

Как будет в дальнейшем показано в сравнительной таблице методов группировки, в данном случае время обработки массива сокращается примерно в два раза.

ГРУППИРОВКА МАССИВА МЕТОДОМ ПРЯМОЙ ВЫБОРКИ

В рассмотренном ниже алгоритме последовательно выбирается минимальный элемент в подмассивах $x_j,...,x_n;\ j = 1..(n-1)$; в связи с этим данный метод группировки можно назвать также методом выделения минимального элемента.

Последовательность реализации алгоритма:

1) Просматриваем массив $x_1, x_2, ..., x_n$ и определяем индекс k минимального элемента. Если $k > 1$, то выполняем обмен элементов x_1 и x_k.

2) Просматриваем подмассив $x_2, x_3, ..., x_n$ и определяем индекс k минимального элемента. Если $k > 2$, то выполняем обмен элементов x_2 и x_k.

3) То же по отношению к подмассиву $x_3, ..., x_n$ и т.д.

Последним подмассивом, подвергающимся просмотру, является подмассив x_{n-1}, x_n.

```
Program MinElem;
Const  Nmax = 500;
Type   Ar = array[1..Nmax] of real;
Var    i,j,n,k : word;
       Xmin : real;
       X : Ar;
Begin
  Ввод  и  печать  n, X
  For i:=1 to n-1 do
    Begin
      Xmin:=x[i]; k:=i;
      For j:=i+1 to n do          { определение индекса k }
        If x[j]<Xmin then         { мин.элемента в подмассиве }
          Begin                   { xi, xi+1, ..., xn }
            Xmin:=x[j]; k:=j
          End;
        If k>i then               { обмен элементов }
          Begin                   { xn и xk }
            x[k]:=x[i]; x[i]:=Xmin
          End;
    End;
  Печать массива X
End.
```

Здесь внешний цикл - перебор подмассивов; внутренний - поиск минимального элемента в подмассиве. Количество выполнений внешнего и внутреннего циклов не зависит от исходной упорядоченности массива X.

Примечание. Аналогичным образом можно реализовать метод прямой выборки путем выделения максимального элемента в последовательных подмассивах.

ГРУППИРОВКА МАССИВА МЕТОДОМ ПРЯМОГО ОБМЕНА

Рассматриваемый метод группировки называют также методом "пузырька".

Запишем сверху вниз элементы $x_1, x_2, ..., x_n$:

Просмотр 1	Просмотр 2	Просмотр 3	. . .	Просмотр *n*-1
x_1	x_1	x_1		x_1
x_2	x_2	x_2		x_2
x_3	x_3	x_3		
……	……..	……		
x_{n-2}	x_{n-2}	x_{n-2}		
x_{n-1}	x_{n-1}			
x_n				

Выполним первый просмотр массива X. При этом будем сравнивать пары смежных элементов x_1 и x_2, x_2 и x_3, x_3 и x_4, ... , x_{n-1} и x_n. Если $x_i > x_{i+1}$ $(i = 1..n-1)$, то обменяем местами x_i и x_{i+1}. Если считать значение числа "весом элемента", то при анализе каждой пары (x_i, x_{i+1}) более легкий элемент "всплывает" на один слой вверх, а более тяжелый "погружается" на один слой вниз. После полного просмотра всех пар элементов наиболее тяжелый элемент опустится на "дно", т.е. займет место элемента x_n. Так как максимальный элемент массива X установлен теперь в конце массива, то в дальнейшем нет необходимости его подвергать анализу. Тогда поднимаем "дно" на одну ступеньку вверх. Последнее сводится к уменьшению количества элементов массива, подвергающихся следующему просмотру, на 1.

Во втором просмотре анализируются пары элементов x_1 и x_2, x_2 и x_3, ... , x_{n-2} и x_{n-1}. Вследствие этого самый тяжелый из оставшихся элементов опустится на новое дно, т.е. займет место элемента x_{n-1}.

В третьем просмотре анализируются пары элементов x_1 и x_2, x_2 и x_3, ... , x_{n-3} и x_{n-2} и т.д.

Работа алгоритма заканчивается, если при очередном просмотре будет определено, что ни разу не имело места нарушение упорядоченности элементов массива, т.е. ни разу не было выполнено условие $x_i > x_{i+1}$.

```
Program Bubble;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of real;
Var  i,n,m : word;
     R : real;
     Cond : boolean;
     X : Ar;
Begin
  Ввод и печать n,X
  Cond:=true; m:=n;
  While Cond do
    Begin
      Cond:=false;
      For i:=1 to m-1 do
        If x[i]>x[i+1] then
          Begin
            R:=x[i]; x[i]:=x[i+1]; x[i+1]:=r;
            Cond:=true;
          End;
      Dec(m);
    End;
  Печать X
End.
```

Комментарии к программе.

1. Так как количество просмотров заранее неизвестно, то для их перебора используется цикл ***While***, работающий под управлением булевской переменной *Cond*. После входа в цикл этой переменной присваивается значение *false* в предположении, что массив уже сгруппирован. После этого в цикле ***For*** проверяется справедливость этого предположения. Если обнаружено нарушение упорядоченности, то производится обмен смежных элементов, а пере-

менной *Cond* присваивается значение *true*, что влечет за собой повторение работы цикла ***While***, т.е. выполнение очередного просмотра.

2. Как было указано выше, после каждого просмотра уменьшается количество анализируемых элементов массива. Поскольку при этом нельзя изменять переменную *n,* определяющую количество элементов в исходном массиве, то для этой цели применяется буферная переменная *m.*

Сравним работу рассмотренных алгоритмов группировки.

Пусть исходный массив упорядочен "наоборот", т.е. по убыванию. В этом случае внешний цикл программы метода "пузырька" работает $n-1$ раз, т.е. столько же, сколько и внешний цикл метода наименьшего элемента и школьного алгоритма. Если $x_i > x_{i+1}$, то во внутреннем цикле в методе прямой выборки выполняются два оператора присваивания, в школьном алгоритме – два оператора, в методе пузырька - четыре оператора. Следовательно, в этом случае метод "пузырька" потребует больше машинного времени, чем метод наименьшего элемента и школьный алгоритм.

Пусть исходный массив упорядочен по возрастанию. По методу наименьшего элемента, как и по школьному алгоритму, все равно будет выполнен $n-1$ внешний цикл, по методу "пузырька" - лишь один просмотр исходного массива. Следовательно, затраты машинного времени на обработку массива по методу "пузырька" будут значительно меньше.

По отношению к рассмотренным трем алгоритмам был проведен компьютерный эксперимент. В качестве критерия сравнения было принято среднее время группировки вещественного массива, состоящего примерно из 300 элементов и организованного по возрастанию, по убыванию и произвольным образом. Реультаты эксперимента в условных единицах времени:

Метод пузырька	Метод прямой выборки	Школьный алгоритм
1,0	1,4	1,6

Следовательно, в общем случае целесообразно применять метод пузырька.

Было произведено также более детальное сравнение методов группировки. Здесь программа создавала одномерный массив, состоящий из 1000 элементов. Каждый элемент – это запись, состоящая из ключа (целое число типа word), строки длиной 50 символов и вещественного числа. Содержимое каждой записи формировалось с помощью генератора случайных чисел. Группирование элементов массива производилось по значению ключа. Чтобы время группировки массива было более-менее заметным, процесс обработки повторялся 1000 раз. Результаты обработки представлены ниже в сравнительной таблице. При этом время обработки, указанное в таблице, – это секунды, затраченные на 1000-кратное повторение соответствующего метода группировки.

Метод	Исходный массив	Массив по возрастанию	Массив по убыванию
Школьный алгоритм	18,73	16,87	17,41
Молернизир. школьный алгоритм	8,71	8,51	9,34
Метод прямой выборки	5,44	5,43	5,77

Метод прямого обмена	3,9	0,73	9,06
Быстрая сортировка	1,88	2,03	3,17

Примечание. Рассмотренные методы относят к методам сортировки первого порядка. Существуют также несколько методов второго порядка, работающие значительно быстрее, но реализованные по более сложным алгоритмам. Одним из наиболее эффективных является метод быстрой сортировки [12], программная реализация которого приведена в учебном пособии после рассмотрения динамических переменных.

ПОИСК В УПОРЯДОЧЕННОМ МАССИВЕ

Рассматривается массив целых чисел, сгруппированных по возрастанию (или убыванию). При этом предполагается, что в данном массиве нет повторяющихся элементов, т.е. строго выполняется отношение $x_i < x_{i+1}$. Требуется для заданного целочисленного значения b определить индекс k элемента $x_k = b$. Если в массиве нет элемента, равного значению b, то присвоить переменной k нулевое значение.

Такая задача поиска часто возникает в системах АСУ, САПР и других при выборке информации из различных архивов. Самым простым методом для ее решения является метод прямого перебора, или, как его еще называют, метод линейного поиска. При использовании указанного метода последовательно просматриваются элементы массива до тех пор, пока не будет найден искомый элемент или не станет ясно, после просмотра всего массива, что такого элемента здесь нет.

```
Program DirectSearch;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of word;
Var  i,k,n,b : word;
     X : Ar;
Begin
  Ввод и печать n,X,b
  k:=0;
  For i:=1 to n do
    If x[i]=b then
      Begin
        k:=i; Break
      End;
  Печать  k
End.
```

Если $b = x_1$, то в массиве X анализируется лишь один элемент; если $b = x_n$ или в массиве X отсутствует элемент, равный значению b, то анализируются n элементов. Среднее количество просмотров элементов массива X равно $n/2$.

Если массив неупорядочен, то метод прямого перебора является единственно возможным. Для упорядоченного массива наиболее эффективным является метод двоичного (бинарного) поиска.

Пусть массив X имеет n элементов. Сущность алгоритма двоичного поиска сводится к следующему.

1) Рассматривается диапазон поиска $1..n$. Левая граница диапазона $i_1 = 1$, правая граница $i_2 = n$. Выходной переменной k присваивается нулевое начальное значение.

2) Если $i_1 > i_2$, поиск прекращается.

3) Определяется $m = \frac{i_1 + i_2}{2}$, т.е. индекс середины диапазона.

4) Если $x_m = b$, поиск закончен, переменной k присваивается значение m. В противном случае - переход к шагу 5.

5) Если $x_m > b$, то это означает, что в правой половине подмассива (от m+1 до i_2) не может быть элемента, равного b. Тогда граница i_2 перемещается в точку m-1: $i_2 := m - 1$. При этом диапазон поиска уменьшается вдвое. Дальше - переход к шагу 2.

6) Если $x_m < b$, то аналогичным образом к точке m+1 сдвигается левая граница i_1: $i_1 := m + 1$. В этом случае диапазон поиска также уменьшается вдвое и производится переход к шагу 2.

Если в массиве X нет элемента, равного b, то после нескольких шагов возникает ситуация $i_1 > i_2$, что ведет к прекращению работы алгоритма. В этом случае выходная переменная сохраняет начальное значение $k = 0$.

Примечание. Выполнять в пп.5 и 6 переход в точку m нет смысла, так как элемент x_m уже был проверен.

```
Program BinarySearch;
Const  Nmax = 500;
Type   Ar = array[1..Nmax] of integer;
Var   i1,i2,k,n,m,b : integer;
      X : Ar;
Begin
  Ввод и печать n,X,b
  i1:=1; i2:=n; k:=0;
  While i1<=i2 do
    Begin
      m:=(k1+k2) div 2;
      If x[m]=b then
        Begin
          k:=m; Break
        End;
      If x[m]>b then
        i2:=m-1
      Else
        i1:=m+1;
    End;
  Печать k
End.
```

Среднее количество просмотров элементов массива в методе двоичного поиска равно $\log_2 n = \frac{\ln(n)}{\ln(2)}$, что значительно меньше значения $\frac{n}{2}$.

МНОГОМЕРНЫЕ МАССИВЫ

В описании типа массива

```
array [T1] of T
```

тип *T,* определяющий компоненты массива, может быть любым (кроме файла). Следовательно, тип *T* может быть в частном случае составным типом, в том числе массивом.

Пример.

```
Const  M = 10;  N = 15; K = 1;  L = 12;
Type  T1 = M..N;  T2 = K..L;  T = real;
Var  A : array[T1] of array[T2] of T;
     B : array[M..N] of array[K..L] of T .
```

Компонент такого массива обозначается *a*[*i*][*j*], *b*[*i*][*j*].

Обычно используют сокращенную форму описания массива:

```
Var  A : array[T1,T2] of T;
     B : array[M..N,K..L] of T.
```

В этом случае компонент имеет вид *a*[*i,j*], *b*[*i,j*].

В нашем примере *A* и *B* - это двумерные массивы, т.е. матрицы, элементы которых имеют тип *T* (в данном случае тип *real*).

Массивы могут быть и более чем двумерные. Пример трехмерного массива:

```
Var  C : array[1..10,1..15,1..8] of real.
```

Компоненты многомерного массива располагаются в памяти таким образом, что наиболее быстро изменяется последний индекс, а наиболее медленно - первый индекс. По отношению к матрице это означает, что ее элементы располагаются в памяти по строкам: вначале все элементы первой строки в соответствии с описанием типа матрицы, затем все элементы второй строки и т.д.

Сравним расположение в памяти элементов одномерного массива и матрицы.

```
Const  Mmax = 30;  Nmax = 40;
Type  Ar = array[1..Nmax] of byte;
     Matrix = array[1..Mmax,1..Nmax] of byte;
Var  X : Ar;
     A : Matrix;
     m, n : byte;
```

Предположим, что в программе введено $n = 20$ элементов массива *X* и $m \times n = 15 \times 20$ элементов матрицы *A* ($m \le Mmax$, $n \le Nmax$).

В поле памяти, отведенном для массива *X*, будут последовательно заняты первые 20 байт. В отличие от этого, поле памяти *A* будет заполнено не всплошную, а фрагментарно элементами матрицы: 20 байтов первой строки, 20 свободных байтов, 20 байтов второй строки, 20 свободных байтов, …, 20 байтов пятнадцатой строки, $20 + 5 \times 40$ свободных байтов. Эту фрагментарность необходимо учитывать в ряде случаев при программной обработке матрицы.

Если *A* и *B* - массивы одного типа, то возможно присваивание *B* := *A* . Это связано с общим принципом работы оператора присваивания вида *y* := *x* - пересылка содержимого поля *x* в поле *y*. Если массивы *A* и *B* описаны одним именем типа, то это гарантирует, что структура и длина полей *A* и *B* одинаковы, в связи с чем не может быть нарушений работы программы при такой пересылке.

Пусть многомерный массив имеет *k* индексов (для матрицы $k = 2$). При обработке такого массива в общем случае производится изменение каждого его индекса. В связи с этим

каждому индексу должна соответствовать отдельная переменная, которая при использовании оператора ***For*** является именем параметра цикла. Имена переменных-индексов не должны совпадать между собой.

Обработка многомерных массивов выполняется путем использования вложенных циклов, когда тело внешнего цикла содержит в своем составе другой цикл, являющийся для него внутренним.

ВВОД И ПЕЧАТЬ ЭЛЕМЕНТОВ МАТРИЦЫ

1. Ввод элементов матрицы с клавиатуры.

Как известно, при выполнении процедуры *Read*(список-ввода) работа программы приостанавливается. С тем, чтобы пользователю было ясно, что от него требует программа, рекомендуется перед процедурой *Read* выводить на экран соответствующий запрос.

```
Const  Mmax = 30;  Nmax = 20;
Type  Matrix = array[1..Mmax,1..Nmax] of real;
Var  i,j,m,n : byte;
     A : Matrix;
Begin
  Writeln('Введите размеры матрицы  m, n');
  Read(m,n); Writeln('m= ',m,'  n=',n);
  For i:=1 to m do
    For j:=1 to n do
      Begin
        Writeln('Введите элемент матрицы a[',i,',',j,']');
        Read(a[i,j]);
      End;
```

2. Ввод матрицы из текстового файла.

Ввод матрицы по концу файла, как это было сделано для одномерного массива, выполнить нельзя, так как при этом заранее неизвестно, сколько элементов содержится в одной строке матрицы (одно и то же общее количество элементов матрицы $k = m \cdot n$ можно получить при различных значениях m и n). Поэтому рекомендуется в первой строке текстового файла разместить значения m и n, а дальше - элементы матрицы. Количество таких элементов в одной строке текстового файла не лимитируется. Однако для удобства просмотра файла целесообразно каждую строку матрицы начинать с новой строки файла.

Пример размещения матрицы в файле:

```
                5      8
 22.8   34.9  56  -95  -84.9  77.98  5.0   1.8
-12.8   45.8  47   76   66.7  89.58  4.4  -3.6
 53.6  -14.3  92   48   38.3  44.96 -1.6   7.9
 32.7   69.9  21  -21  -92.6  96.92  8.2   5.0
 42.4   74.2  39  -45   14.1  17.11 -9.8   6.1
```

```
Const  Mmax = 30;  Nmax = 20;
Type  Matrix = array[1..Mmax,1..Nmax] of real;
Var  i,j,m,n : byte;
     A : Matrix;
     FileA : text;
Begin
  Assign(FileA,'Matrix.dat');
```

```
Reset(FileA);
Read(FileA,m,n);
For i:=1 to m do
  For j:=1 to n do
    Read(FileA,a[i,j]);
Close(FileA);
```

Если в файле *Matrix.dat* содержится менее $m \cdot n$ элементов, то недостающим элементам будет присвоено нулевое значение; в противном случае часть данных из файла не будет использована.

3. Вывод матрицы на экран.

Будем считать, что вся матрица может быть размещена на экране.

Требования к программе вывода:

- на каждой строке экрана печатать не более чем k элементов матрицы (например, $k = 5$);
- каждую строку матрицы начинать с новой строки экрана.

```
Const  Mmax = 30;  Nmax = 20;
Type  Matrix = array[1..Mmax,1..Nmax] of real;
Var  i,j,k,m,n : byte;
     A : Matrix;
Begin
  ..................
  For i:=1 to m do
    Begin
      k:=0;
      For j:=1 to n do
        Begin
          Inc(k);
          If k<5 then
            Write(a[i,j]:7:2,'  ')
          Else
            Begin
              k:=0;
              Writeln(a[i,j]:7:2);
            End;
        End;
      If k>0 then Writeln;
    End;
```

ПРИМЕРЫ ОБРАБОТКИ МАТРИЦ

Пример 1. Вычислить первую норму прямоугольной матрицы

$$\left| A \right|_I = \max \sum_{j=1}^{n} \left| a_{ij} \right| , \quad i = 1..m$$

Обозначим сумму модулей элементов первой строки b_1, второй строки - b_2 и т.д.

$$b_1 = \sum_{j=1}^{n} \left| a_{1,j} \right| ; \qquad b_2 = \sum_{j=1}^{n} \left| a_{2,j} \right| ; \quad \ldots ; \quad b_m = \sum_{j=1}^{n} \left| a_{m,j} \right|$$

Тогда

$$\left| A \right|_I = \max(b_1, b_2, ..., b_m)$$

Порядок вычислений:

1. Для каждой строки матрицы сформировать элемент одномерного массива *B* как сумму модулей ее элементов.

2. В массиве *B* найти наибольший элемент и присвоить его значение переменной, определяющей первую норму матрицы.

```
Program Norma1;
Const  Mmax = 20;  Nmax = 15;
Type  Matrix = array[1..Mmax,1..Nmax] of real;
      Vector = array[1..Mmax] of real;
Var  i,j,m,n : byte;
     A1 : real;
     A : Matrix;
     B : Vector;
Begin
  Ввод  и  печать  m, n, A
  For i:=1 to m do           { Формирование }
    Begin                    { массива B }
      b[i]:=0;
      For j:=1 to n do
        b[i]:=b[i]+abs(a[i,j]);
    End;
  A1:=b[1];                  { Определение }
  For i:=2 to m do           { максимального }
    If b[i]>A1 then          { элемента }
      A1:=b[i];              { в массиве B }
  Writeln('A1= ',A1:7:2);
End.
```

Пример 2. Транспонировать квадратную матрицу.

При транспонировании меняются местами строки и столбцы матрицы, т.е. происходит обмен значений элементов a_{ij} и a_{ji} ($i, j = 1 .. n$).

```
Program Trans;
Const  Nmax = 25;
Type  Matrix = array[1..Nmax,1..Nmax] of real;
Var  i,j,n : byte;
     R : real;
     A : Matrix;
Begin
  Ввод  и  печать  n, A
  For i:=1 to n do
    For j:=1 to n do
      Begin
        R:=a[i,j]; a[i,j]:=a[j,i]; a[j,i]:=R
      End;
  Печать A
End.
```

В этой программе дважды производится обмен каждой пары элементов (например, при $i = 1, j = 5$ обмениваются элементы $a_{1,5}$ и $a_{5,1}$, при $i = 5, j = 1$ - элементы $a_{5,1}$ и $a_{1,5}$). Следовательно, после отработки программы матрица *A* сохраняет исходный вид.

В программе Trans наряду с другими элементами обмениваются также элементы главной диагонали: при $i = 1, j = 1$ - элементы $a_{1,1}$ и $a_{1,1}$, при $i = 2, j = 2$ - элементы $a_{2,2}$ и $a_{2,2}$ и

т.д. Последнее не влияет на корректность работы программы, но приводит к ненужным затратам машинного времени.

Однократный обмен пары элементов при транспонировании матрицы можно обеспечить, если выполнять перебор лишь тех элементов, которые расположены выше главной диагонали, и обменивать их значения с соответствующими элементами, расположенными ниже главной диагонали.

Выпишем отдельно индексы элементов над главной диагональю:

```
1,2   1,3   1,4   1,5   ...   1,n
      2,3   2,4   2,5   ...   2,n
            3,4   3,5   ...   3,n
..............................
                          (n-1),n
```

Первый индекс элемента a_{ij} изменяется от 1 до n-1, второй индекс - от i+1 до n, т.е.

$$i = 1 .. (n-1); \quad j = (i+1) .. n .$$

Программа транспонирования после ее коррекции (фрагмент):

```
For i:=1 to n-1 do
  For j:=i+1 to n do
    Begin
      R:=a[i,j]; a[i,j]:=a[j,i]; a[j,i]:=R
    End;
```

Закономерность изменения индексов, использованная в примере 2, может быть такой же и некоторых задачах по обработке одномерных массивов, что иллюстрирует приведенный ниже пример..

Пример 3. На плоскости заданы n точек своими координатами. Определить номера точек, наиболее удаленных друг от друга, после чего удалить их из массива точек.

Для отбора пары максимально удаленных точек необходимо сравнивать расстояния для следующих пар точек:

```
1 - 2   1 - 3   1 - 4   1 - 5   ...   1 - n
        2 - 3   2 - 4   2 - 5   ...   2 - n
                3 - 4   3 - 5   ...   3 - n
                  ..........................
                                    (n-1) - n
```

Если обозначить номера пары точек через (i - j), то

$$i = 1 .. (n-1); \quad j = (i+1) .. n .$$

Вполне очевидно, что работа программы не изменится, если вместо расстояний рассматривать квадраты расстояний, но количество операций при этом заметно сокращается (не нужно вычислять в каждом цикле корень квадратный $sqrt(d)$).

```
Program Distance;
Const  Nmax = 400;
```

```
Type  Ar = array[1..Nmax] of real;
Var  i,j,n,imax,jmax : integer;
     d,dmax : real;
     X,Y : Ar;
Begin
  Ввод и печать n,X,Y
  dmax:=0;
  For i:=1 to n-1 do
    For j:=i+1 to n do
      Begin
        d:=sqr(x[i]-x[j])+sqr(y[i]-y[j]);
        If d>dmax then
          Begin
            dmax:=d; imax:=i; jmax:=j
          End;
      End;
  dmax:=sqrt(dmax);
  Writeln('imax= ',imax,'  jmax= ',jmax,'  dmax= ',
          dmax:7:2);
  For i:=jmax to n-1 do
    Begin
      x[i]:=x[i+1]; y[i]:=y[i+1]
    End;
  Dec(n);
  For i:=imax to n-1 do
    Begin
      x[i]:=x[i+1]; y[i]:=y[i+1]
    End;
  Dec(n);
  Печать  n, X, Y
End.
```

Так как $j \geq i+1$ и, следовательно, *jmax* > *imax,* то в первую очередь удаляется точка с индексом *jmax*. Если первой удалить точку *imax,* то вторая точка смещается влево, при этом во втором цикле была бы удалена точка с индексом *jmax*+1, что привело бы к неверной работе программы.

Пример 4. В каждом столбце прямоугольной матрицы определить среднее арифметическое *S* его нечетных элементов , после чего переставить столбцы в порядке уменьшения значения *S*.

Предварительное замечание. Поскольку речь идет о нечетных элементах, то компонентами матрицы могут быть только целые числа. В то же время средние значения, если об этом особо не оговаривается, всегда представляют собой вещественные числа.

```
Program Example4;
Const  Mmax = 20;  Nmax = 15;
Type  Matrix = array[1..Mmax,1..Nmax] of integer;
Var   i,j,          { параметры цикла }
      m,n,          { текущие размеры матрицы }
      k,            { кол-во нечетных эл-тов в столбце }
      P : byte;     { вспомогательная переменная }
      A : Matrix;   { исходная матрица }
      S : Ar;       { массив средних значений }
      R : real;     { переменная для обмена вещ.значений }
      b : integer;  { переменная для обмена целых значений }
   Cond : boolean;  { управляющая переменная }
```

```
Begin
  Ввод и печать  m, n, A
  For j:=1 to n do              { Формирование массива S }
    Begin
      S[j]:=0; k:=0;
      For i:=1 to m do
        If odd(a[i,j]) then
          Begin
            S[j]:=S[j]+a[i,j];
            Inc(k);
          End;
      If k>0 then
        S[j]:=S[j]/k;
    End;
  Cond:=true; p:=n-1;           { Группировка массива S }
  While Cond do                 { по убыванию с одновременной }
    Begin                       { перестановкой столбцов матрицы }
      Cond:=false;
      For j:=1 to p do
        If S[j]<S[j+1] then
          Begin
            R:=S[j]; S[j]:=S[j+1];
            S[j+1]:=R;
            For i:=1 to m do
              Begin
                b:=a[i,j]; a[i,j]:=a[i,j+1];
                a[i,j+1]:=b
              End;
            Cond:=true;
          End;
      Dec(p);
    End;
  Вывод   m, n, A
End.
```

Примечание. Проверка «**If** k>0 **then** ...» блокирует деление на нуль, если в столбце нет нечетных элементов.

В программе подвергается группировке методом прямого обмена одномерный массив *S*, но при обмене элементов *S*[j] и *S*[j+1] одновременно обмениваются j-ый и (j+1)-ый столбцы матрицы *A*.

Пример 5. Умножение прямоугольных матриц.

$$C = AB; \quad [\, A(m \times l), B(l \times n), C(m \times n) \,]$$

$$c_{i,j} = \sum_{k=1}^{l} a_{i,k} b_{k,j}; \quad i = 1..m; \;\; j = 1..n$$

Примечание. В соответствии с правилами умножения двух прямоугольных матриц *A* и *B*:

1) матрица *B* должна иметь столько же строк, сколько столбцов имеет матрица *A*;

2) если обозначить размеры матрицы *A* через *m* и *l*, а размеры матрицы *B* через *l* и *n*, то результирующая матрица *C* будет иметь размеры *m* и *n*;

3) значение произвольного элемента *a*[i,j] матрицы *A* – это сумма парных произведений элементов *i*-ой строки матрицы *A* на элементы *j*-го столбца матрицы *B*.

```
Program Mult;
Const  Mmax = 20;  Nmax = 15;
```

```
       Lmax = 25;
Var  i,j,k,l,m,n : byte;
     A : array[1..Mmax,1..Lmax] of real;
     B : array[1..Lmax,1..Nmax] of real;
     C : array[1..Mmax,1..Nmax] of real;
Begin
  Ввод  и  печать  m, n, l, A, B
  For i:=1 to m do
    For j:=1 to n 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;
  Печать матрицы C
End.
```

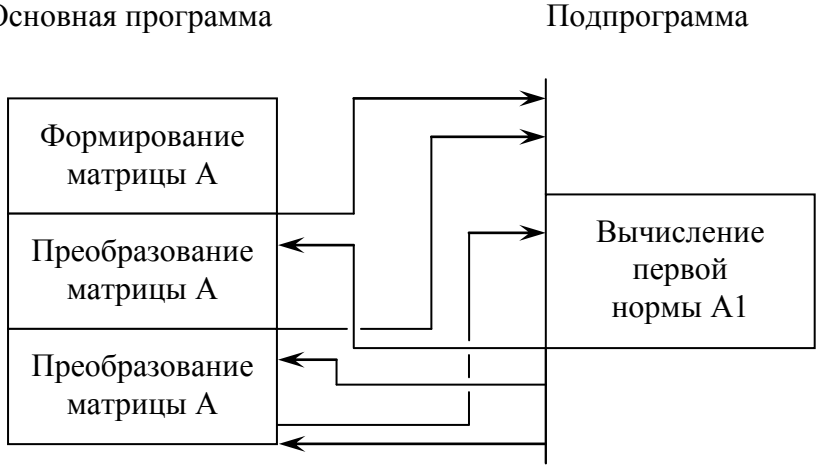
П Р О Ц Е Д У Р Ы

Программа может содержать одинаковые фрагменты. Такие фрагменты целесообразно оформить в виде подпрограммы, которая должна несколько раз выполняться.

Пусть, например, в программе требуется трижды вычислять первую норму матрицы A. Без использования подпрограмм такая программа может иметь следующую структуру:

Формирование матрицы А
Вычисление первой нормы А1
Преобразование матрицы А
Вычисление первой нормы А1
Преобразование матрицы А
Вычисление первой нормы А1

Если оформить вычисление первой нормы А1 в виде подпрограммы, то структура программы станет следующей:



К подпрограмме осуществляется три обращения из основной программы.

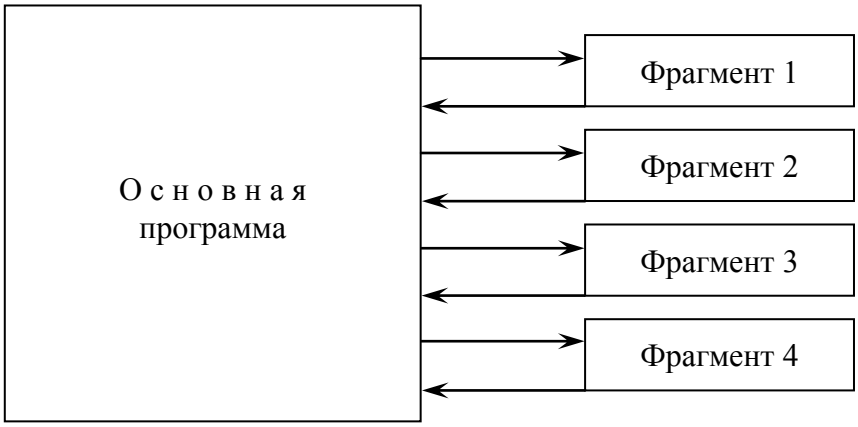
Использование подпрограмм сокращает общий объем программы и улучшает ее понимание.

Большую программу целесообразно разделить на ряд последовательных фрагментов и каждый из них оформить в виде подпрограммы. Хотя каждая из этих подпрограмм выполняется только один раз, т.е. сокращения объема программы не происходит, но при этом улучшаются ее читаемость и удобство отладки.

Исходная большая программа :

Фрагмент 1
Фрагмент 2
Фрагмент 3
Фрагмент 4

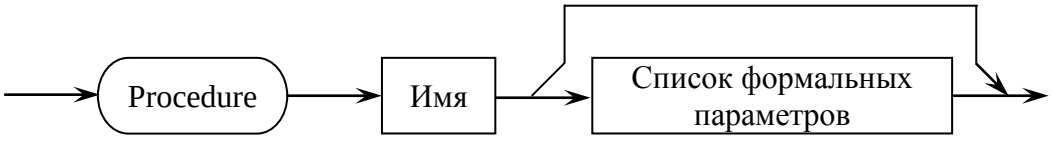
Программа с использованием подпрограмм:



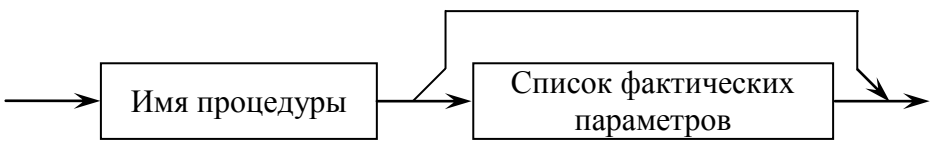
Подпрограммы на Паскале реализуются в виде процедур и функций. Функция отличается от процедуры тем, что в ней в качестве выходного результата может быть только одно простое значение (аналогично функциям *sin*(*x*), *max*(*a,b,c*) и т.п.). Процедура в качестве выходного результата может выдавать одно или больше значений (например, массив); в частном случае процедура может не иметь ни одного выходного значения.

Описание процедуры, располагаемое в разделе описания процедур и функций, состоит из двух частей: заголовок и тело процедуры. Телом процедуры является блок. Следовательно, тело процедуры имеет такую же структуру, как и программный блок, т.е. в него входят разделы описания меток, констант и т.д.

Синтаксическая диаграмма для заголовка процедуры:



Обращение к процедуре осуществляется с помощью оператора процедуры:



Примечание. Фактические параметры часто называют также аргументами.

Наиболее простой является процедура без параметров.

Пример 1. В массиве X присвоить максимальному элементу нулевое значение и определить, как изменилось при этом среднее арифметическое значение элементов массива.

```
Program Middle1;
Const  Nmax = 1000;
Type  Ar = array[1..Nmax] of real;
Var  i,imax : integer;
     S,S1,S2,dS,Xmax : real;
     X : Ar;
{ ---------------------------- }
Procedure MiddleAr1;
Var  i : integer;        { Определение среднего }
Begin                    { арифметического значения }
  S:=0;                  { элементов массива X }
  For i:=1 to n do
    S:=S+x[i];
  S:=S/n;
End { MiddleAr1 };
{ ---------------------------- }
Begin
  Ввод  и  печать  n, X
  MiddleAr1; S1:=S;
  Xmax:=x[1]; imax:=1;  { Определение место- }
  For i:=2 to n do      { положения (индекса) }
    If x[i]>Xmax then   { максимального элемента }
      Begin
        Xmax:=x[i]; imax:=i
      End;
  x[imax]:=0;
  MiddleAr1; S2:=S;
  dS:=S2-S1;
  Печать  S1, S2, dS
End.
```

В данном случае блок процедуры имеет раздел описания переменных и раздел операторов. В разделе ***Var*** процедуры *MiddleAr1* повторно описывается переменная *i*, так как эта переменная используется в блоке процедуры как параметр цикла (как известно, параметр цикла должен быть описан в том же блоке, в котором расположен оператор ***For***). Обращение к процедуре осуществляется путем указания имени этой процедуры в основной программе.

Пример 2.

$$y_i = 2x_i^3 + 3.5x_i^2 - x_i + 1$$

$$z_i = \begin{cases} x_i + 2y_i, & \text{если } y_i > 0 \\ x_i + y_i, & \text{если } y_i \le 0 \end{cases}$$

Определить среднее арифметическое значение элементов массивов X, Y и Z.

```
Program Middle2;
Const  Nmax = 1000;
Type  Ar = array[1..Nmax] of real;
Var  i,n : integer;
     S,Sx,Sy,Sz : real;
     X,Y,Z,Buf : Ar;
```

```
{ ------------------------ }
Procedure MiddleAr2;
Var  i : integer;        { Определение среднего }
  Begin                  { арифметического }
    S:=0;                { значения элементов }
    For i:=1 to n do     { буферного массива Buf }
      S:=S+Buf[i];
    S:=S/n;
End { MiddleAr2 };
{ ------------------------ }
Begin
  Ввод и печать n, X
  For i:=1 to n do
    Begin
      y[i]:=((2*x[i]+3.5)*x[i]-1)*x[i]+1;
      If y[i]>0 then
        z[i]:=x[i]+2*y[i]
      Else
        z[i]:=x[i]+y[i];
    End;
  Buf:=X; MiddleAr2; Sx:=S;
  Buf:=Y; MiddleAr2; Sy:=S;
  Buf:=Z; MiddleAr2; Sz:=S;
  Печать Sx,Sy,Sz
End.
```

Непосредственно в процедуре MiddleAr2 вычисляется среднее арифметическое значение для элементов буферного массива *Buf*. Перед обращением к этой процедуре в массив *Buf* пересылается массив *X*, массив *Y* или массив *Z*.

Использование в программе Middle2 буферного массива *Buf* приводит к дополнительным затратам памяти на размещение этого массива, а также к дополнительным затратам машинного времени на пересылку массивов *X, Y, Z* в массив *Buf*. Чтобы исключить такие непроизводительные затраты, необходимо использовать аппарат формальных и фактических параметров.

ОБЛАСТЬ ДЕЙСТВИЯ ИМЕН И МЕТОК

Рассмотрим следующую программу:

```
Program SubRout;
Var  x,y,z : real;
     k,m,n : integer;
  Procedure P1;
  Var  x,y,q : word;
       m,u : real;
     Procedure PP1;
     Var  y,t : byte;
          n : longint;
     Begin
     ..............
     End { PP1 };
  Begin
  ..............
    PP1;
  ..............
  End { P1 };
```

```
  Procedure P2;
  Var  z,v : char;
       n,r : longint;
  Begin
  ..............
  End { P2 };
Begin
..................
  P1;
..................
  P2;
..................
End.
```

Как уже было отмечено, память для переменных выделяется при старте того блока, в котором объявлены эти переменные, и освобождается после окончания работы этого блока. Следовательно, при старте основной программы память будет выделена лишь для описанных в ее разделе ***Var*** переменных *x, y, z, k, m, n.* Этим переменным в машинной программе будут соответствовать конкретные адреса полей памяти.

При обращении к процедуре *P1* дополнительно выделяется память для описанных в ней переменных *x, y, q, m, u*, а после обращения к ее внутренней процедуре *PP1* - также для переменных *y, t, n*. Следовательно, переменная *y* типа *real,* объявленная в основной программе, переменная *y* типа *word* из процедуры *P1* и переменная *y* типа *byte* из процедуры *PP1* имеют разные адреса памяти, т.е. на уровне Паскаль-программы они считаются разными переменными. После окончания работы процедуры *PP1* память, отведенная для переменных *y, t* и *n,* освобождается; после окончания работы процедуры *P1* освобождается также память, занятая переменными *x, y, q, m, u.*

Область действия переменной - это тот участок программы, где имени переменной ставится в соответствие один и тот же машинный адрес. *В Паскаль-программе областью действия переменной является весь блок, в котором объявлена данная переменная, за исключением тех внутренних блоков, в которых эта же переменная объявлена повторно.*

В рассматриваемом примере область действия переменной *x,* объявленной в основной программе, - вся программа, за исключением процедуры *P1;* переменной *k* - вся программа, в том числе и процедуры *P1, PP1, P2*; переменной *q* - процедура *P1*, в том числе и ее внутренняя процедура *PP1*; переменной *t* - только процедура *PP1* и т.д.

Переменные, объявленные в основной программе, действуют во всей этой программе. Такие переменные называют глобальными (в программе SubRout - это переменные *x, y, z, k, m, n*). Переменные, объявленные в процедуре, действуют лишь в пределах этой процедуры. Это локальные переменные. Более общее определение:

- локальными называют переменные, которые объявлены в разделах ***Var*** процедур и функций;

- глобальными называют переменные, которые объявлены в разделах ***Var*** вне процедур и функций.

При рассмотрении модулей пользователя будет указано, что глобальными являются не только переменные, объявленные в основной программе. Формальное отличие между локальными и глобальными переменными определяется местом их объявления: в блоках процедур и функций или вне этих блоков.

Для размещения переменных в оперативной памяти выделяются две области, которые называются соответственно сегментом данных и сегментом стека. Все глобальные переменные размещаются в сегменте данных при старте основной программы. Следовательно, адреса

этих переменных не изменяются в течение всего времени работы программы. Максимальный размер сегмента данных равен 65520 байт, его реальный размер определяется общим объемом глобальных переменных.

Локальные переменные размещаются в сегменте стека. Память им выделяется при каждом старте соответствующей процедуры или функции. Следовательно, при повторном обращении к процедуре локальная переменная может получить адрес, не совпадающий с тем, который она имела при предыдущем обращении к этой же процедуре. Размер сегмента стека может быть задан программистом в диапазоне от 1024 до 65520 байт. По умолчанию он равен 16384 байта.

Правила, определяющие область действия метки, несколько отличаются от правил, рассмотренных выше по отношению к области действия имен.

Как известно, метка используется в программе только оператором ***Goto***, который производит передачу управления к оператору, отмеченному соответствующей меткой. При этом по правилам языка Паскаль запрещается переход вовнутрь сложного оператора (оператор цикла, условный оператор, составной оператор и др.). Раздел операторов процедуры - это составной оператор (***Begin .. End***). Следовательно, переход во внутреннюю область процедуры из основной программы или из другой процедуры с помощью оператора ***Goto*** недопустим. Вход в процедуру возможен лишь с ее начала путем обращения к этой процедуре.

С учетом вышесказанного *областью действия метки является весь блок, в котором объявлена данная метка, за исключением внутренних блоков вне зависимости от наличия в них объявления меток.*

Принудительный выход из процедуры можно осуществить с помощью оператора ***Goto.***

Пример:

```
Procedure P3;
Label  10;
Var  x : real;
Begin
.............
  If x<0 then
    Goto 10;
.............
  10:
End { P3 };
```

Эта же работа может быть выполнена оператором *Exit,* осуществляющим выход за пределы блока процедуры:

```
Procedure P3;
Var  x : real;
Begin
.............
  If x<0 then
    Exit;
.............
End { P3 };
```

Оба варианта процедуры *P3* действуют совершенно одинаково, но второй вариант предпочтительнее, поскольку в нем не используется в явном виде оператор ***Goto.***

Ранее было указано на два преимущества использования процедур:

- уменьшение текста программы и ее объема в памяти;
- улучшение читаемости программы.

Очень важно еще одно преимущество, определяемое тем, что выделение памяти для объявленных в процедуре переменных происходит только при ее активизации, т.е. при старте процедуры. После окончания работы процедуры память, занятая этими переменными, освобождается.

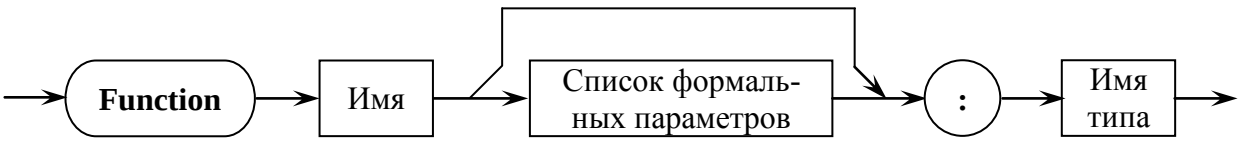
Предположим, что в программе используются три отдельные процедуры, в каждой из которых объявлен один массив, требующий соответственно 10, 20 и 30 Кбайт памяти. Если бы эти массивы были объявлены в основной программе, то для них при старте программы было бы выделено 60 Кбайт памяти. Поскольку указанные процедуры не работают одновременно, то для массивов выделяется не более чем 30 Кбайт. Следовательно, аппарат локальных переменных в процедурах является дополнительным источником экономии памяти в программе.

ФУНКЦИИ

Предположим, что мы используем в программе стандартную функцию *sin*(*x*) (например, в операторе присваивания *y:=sin(x)+1* или в процедуре вывода *Writeln(sin(x))*). Функция *sin* реализуется транслятором как подпрограмма, включаемая в текст программы пользователя. Здесь *x* - это параметр, *sin* - имя подпрограммы. Вычисленное в подпрограмме значение присваивается имени *sin.*

Функция отличается от процедуры тем, что в качестве выходного результата здесь может быть только одно простое значение, которое присваивается имени функции. Следовательно, имя функции играет роль переменной и, естественно, тип этой переменной должен быть явно указан в описании функции.

Описание функции, как и описание процедуры, состоит из заголовка, блока и точки с запятой. Заголовок функции:



Имя типа определяет тип результата. В блоке должно выполняться присваивание имени функции вычисленного выходного значения.

Обращение к функции – это операнд выражения.

Пример 1. Для примера 2 из раздела «Процедуры» оформить вычисление среднего арифметического в виде функции.

Вычисление среднего арифметического для массивов *X, Y* и *Z* отличается друг от друга именем массива (*X, Y* или *Z*) и именем результирующей переменной (*Sx, Sy, Sz*). В данном случае целесообразно имя обрабатываемого массива ввести в состав формальных параметров, а результирующей переменной присваивать значение, определяемое именем функции.

```
Var  i,n : integer;
     Sx,Sy,Sz : real;
     X,Y,Z : Ar;
{ ------------------------ }
```

```
Function MiddleAr3(Var Buf:Ar) : real;
Var  i : integer;  S : real;
Begin
  S:=0;
  For i:=1 to n do
    S:=S+Buf[i];
  S:=S/n;
  MiddleAr3:=S;
End { MiddleAr3 };
{ ------------------------ }
Begin
  Ввод и печать n, X;  формирование  Y, Z
  Sx:=MiddleAr3(X); Sy:=MiddleAr3(Y);
  Sz:=MiddleAr3(Z);
  Печать Sx,Sy,Sz
End.
```

Перенос массива *Buf* из глобальных переменных в формальные параметры имеет принципиальное значение. Как будет показано ниже, в этом случае для массива *Buf* никакой памяти не выделяется, а при обращении к функции в ее блоке происходит замена имени *Buf* на имя фактического массива (*X*, *Y* или *Z*).

Выходное значение как результат работы функции присваивается ее имени, как правило, один раз, в конце раздела операторов.

Пример 2. Дана прямая $a\,x+b\,y+c=0$ и две точки $M_1(x_1,y_1)$ и $M_2(x_2,y_2)$. Определить, находятся ли эти точки по одну сторону от прямой.

Вначале вычислим отклонения точек от прямой:

$$d_1=a\,x_1+b\,y_1+c\,;\quad d_2=a\,x_2+b\,y_2+c\,.$$

Точки находятся по одну сторону от прямой (или обе лежат на этой прямой), если $sign\,(d_1)=sign\,(d_2)$.

```
Program Side;
Var  a,b,c,d1,d2,
     x1,y1,x2,y2 : real;
     Res : boolean;
{ ------------------------------ }
Function Sign(x:real):shortint;
Const  eps = 0.0001;
Begin                    { Определение знака }
  If x>eps then          { переменной x }
    Sign:=1
  Else
    If x<-eps then
      Sign:=-1
    Else
      Sign:=0;
End { Sign };
{ ------------------------------ }
Begin
  Ввод и печать a,b,c,x1,y1,x2,y2
  d1:=a*x1+b*y1+c; d2:=a*x2+b*y2+c;
  Res:=Sign(d1)=Sign(d2);
  Печать Res
End.
```

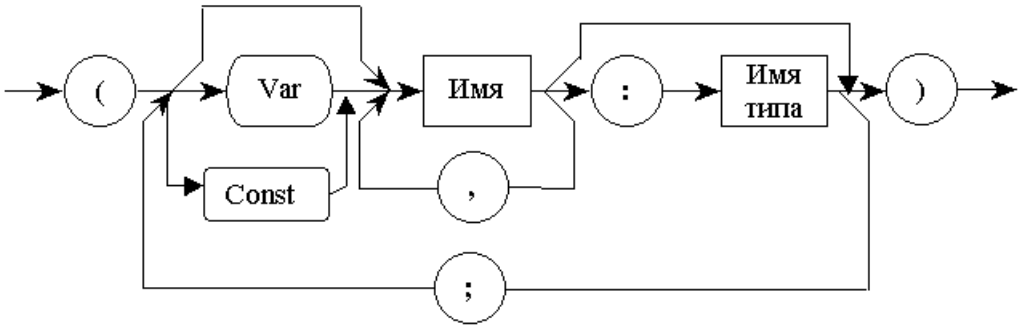
СПИСКИ ПАРАМЕТРОВ

Рассмотрим примеры 1 и 2 предыдущего раздела, в которых применяются соответственно функции *MiddleAr3* и *Sign.* В функции *MiddleAr3* в качестве формального параметра используется массив *Buf*, которому при обращении ставится в соответствие фактический параметр *X, Y* или *Z*. В функции *Sign* формальным параметром является переменная *x*, которой при обращении соответствует фактический параметр *d1* или *d2*.

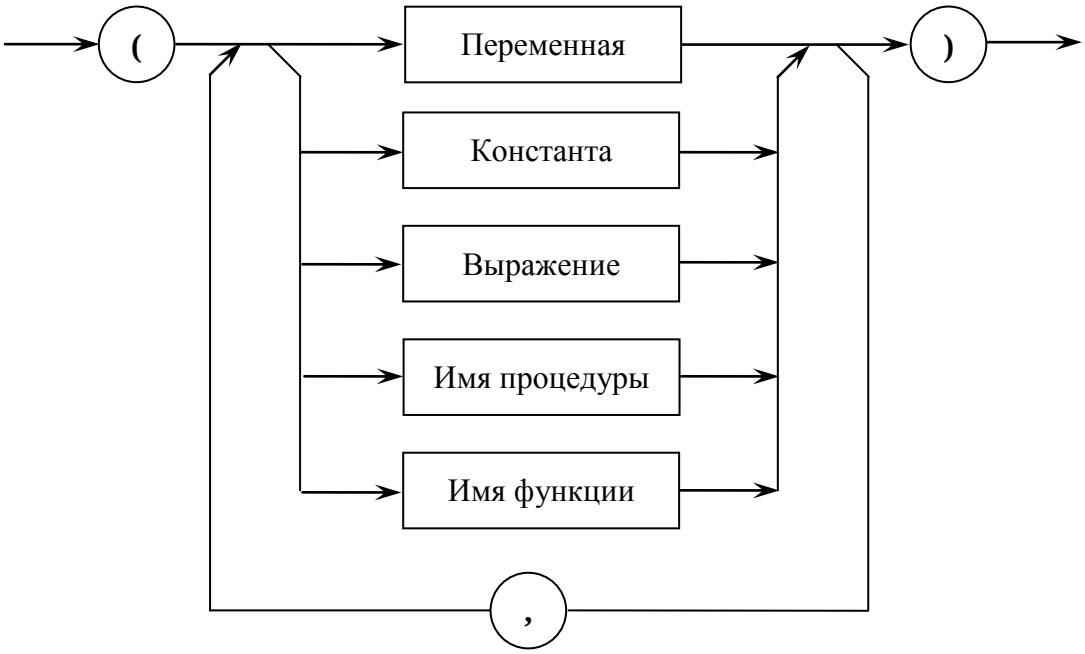
На уровне Паскаль-программы можно считать, что при обращении к процедуре или функции в их теле происходит замена имени формального параметра на имя фактического параметра. Тогда, например, обращение *Sx* := *MiddleAr3*(*X*) эквивалентно выполнению операторов:

```
S:=0;
For i:=1 to n do
  S:=S+x[i];
S:=S/n;
Sx:=S;
```

Синтаксическая диаграмма списка формальных параметров:



Список фактических параметров:



Количество фактических параметров должно быть равно количеству формальных параметров, типы фактических параметров должны соответствовать типам формальных параметров (если последние указаны).

Пусть в заголовке процедуры описан формальный параметр *Z* типа *Ar*, где *Ar* – имя типа массива. Предположим, что при обращении к процедуре формальному массиву *Z* ставится в соответствие фактический массив *X* или фактический массив *Y* того же типа *Ar*. Возникает вопрос – как передать информацию о фактическом массиве в процедуру?

При передаче массива или переменной другого типа можно использовать один из двух возможных способов: передача адреса переменной или передача значения этой переменной. В соответствии с этим параметры разделяются на два вида: параметры-переменные и параметры-значения.

В первом случае формальному параметру назначается в процедуре некоторый фиктивный адрес. При обращении к ней этот адрес заменяется на адрес фактического параметра. Следовательно, формальный и фактический параметры определяют в данном случае одно и то же поле памяти. В нашем примере имени *Z* при первом обращении присваивается адрес, который имеет массив *X*, при втором обращении – адрес массива *Y*.

Если в процедуре описан формальный параметр-значение, то для него отводится отдельное поле памяти, размер которого определяется типом формального параметра. Тогда при обращении к процедуре в это поле переписывается значение фактического параметра. Следовательно, если *Z* – формальный параметр-значение, то при обращении к процедуре имена *Z* и *X*, *Z* и *Y* определяют различные поля памяти.

Обмен информацией между вызывающей программой и процедурой в принципе может быть обеспечен тремя способами, которые можно условно обозначить *in*, *out* и *inout*. В первом случае информация передается только от вызывающей программы в процедуру, во втором случае – только из процедуры в вызывающую программу, в третьем – в обоих направлениях.

В Турбо Паскале для параметра-переменной реализовано два способа: *in* и *inout*. В первом случае перед именем формального параметра записывают слово ***Const*** и называют его параметром-константой. Во втором случае перед именем формального параметра ставят слово ***Var*** и называют его по-прежнему параметром-переменной.

Поскольку формальный параметр-переменная и соответствующий ему фактический параметр имеют один и тот же адрес поля памяти, то любые изменения формального параметра в теле процедуры приводят к таким же изменениям фактического параметра. В частности, это означает, что выходные результаты работы процедуры обязательно должны быть параметрами-переменными.

Различие между параметром-переменной и параметром-константой заключается лишь в том, что по отношению к параметру-константе запрещаются любые изменения ее значения в теле процедуры, т.е. компилятор при трансляции программы запрещает присваивание новых значений такому параметру. Обычно параметры-константы устанавливаются для формальных массивов; в этом случае в процедуру передается адрес массива (4 байта), но изменения фактического массива производиться не будут.

Для параметра-значения в Турбо Паскале реализован лишь один способ обмена информацией – вариант *in*. Перед именем формального параметра-значения в заголовке процедуры ничего не записывается.

При обращении к процедуре формальному параметру-переменной в качестве фактического параметра может соответствовать только переменная (простая или составная), параметру-значению – переменная, константа, выражение, а также имя конкретной процедуры или функции, которые считаются соответственно процедурной или функциональной константой.

Параметры-значения в общем случае требуют больше машинного времени и больше памяти, поэтому их следует применять главным образом тогда, когда в качестве фактического параметра может быть использована константа или выражение.

Рассмотрим несколько подробнее, почему формальному параметру-переменной не могут ставиться в соответствие константа или выражение.

Константа, как и любая переменная, имеет в программе определенный адрес. Однако передача этого адреса в процедуру означало бы, что в теле процедуры может быть изменена данная константа, что недопустимо. Выражение, записанное в списке фактических параметров, не может иметь какого-либо адреса. Перед обращением к процедуре вызывающая программа вычисляет значение такого выражения и передает это значение в процедуру.

Параметры-переменные могут быть нетипизированными, т.е. после имени формального параметра не указывается имя типа. В этом случае им соответствуют фактические параметры любого типа.

Пример 1. Этот пример иллюстрирует разницу между параметром-переменной и параметром-значением.

```
Program Parameters;
Var  m,n : integer;
{ -------------------- }
Procedure Add(x:integer; Var y:integer);
Begin
  x:=x+1; y:=y+1;
  Writeln('x= ',x,'  y=',y);
End { Add };
{ -------------------- }
Begin
  m:=0; n:=0;
  Add(m,n);
  Writeln('m= ',m,'  n= ',n);
End.
```

Будет напечатано:

```
     1    1
     0    1
```

При рассмотрении оператора ***For*** было указано, что описание параметра цикла должно быть расположено в том же блоке, где находится сам оператор ***For.*** Рассмотрим ситуацию, которая может сложиться при нарушении этого требования.

Пример 2. Для заданного вещественного массива X сформировать целочисленный массив Y, в котором значение y_i определяет количество элементов в массиве X, превышающих значение x_i. Вычисление количества таких элементов оформить в виде функции.

```
Program MakeArray;
Const  Nmax = 500;
Type   Ar1 = array[1..Nmax] of real;
       Ar2 = array[1..Nmax] of word;
Var  i,n : word;
     X : Ar1;
     Y : Ar2;
{ -------------------------- }
Function Elem(k:word):word;
Var  p : word;          { Определение количества }
```

```
Begin                    { элементов массива X, }
  p:=0;                  { превышающих значение }
  For i:=1 to n do       { элемента x[k] }
    If x[i]>x[k] then
      Inc(p);
  Elem:=p;
End { Elem };
{ -------------------------- }
Begin
  В в о д   n, X
  For i:=1 to n do
    y[i]:=Elem(i);
   П е ч а т ь   Y
End.
```

Здесь и в основной программе, и в подпрограмме используется параметр цикла *i*, описанный как глобальная переменная. Обращение к функции *Elem* производится при каждом выполнении цикла ***For*** основной программы. После первого же обращения к функции *Elem* переменная *i* будет иметь значение *n*, что приведет к прекращению работы цикла ***For*** основной программы. В результате для массива *Y* будет вычислено лишь одно значение y_1.

Ситуация, демонстрируемая в программе MakeArray, аналогична случаю, когда во внешнем и внутреннем циклах используется один и тот же параметр цикла. Пример:

```
For i:=1 to n do
  Begin
    p:=0; k:=i;
    For i:=1 to n do
      If x[i]>x[k] then
        Inc(p);
    y[i]:=p;
  End;
```

Следовательно, параметр цикла ***For*** необходимо описывать в той же процедуре или функции, где расположен этот оператор.

О СТИЛЕ ПРОГРАММИРОВАНИЯ

Программу MakeArray (пример 2 предыдущего раздела) можно было бы написать следующим образом:

```
Program MakeArray1; Const Nmax=500;Type Ar1=array[1..Nmax] of
real;Ar2=array[1..Nmax] of word;Var i,n:word;X:Ar1;Y:Ar2;
Function Elem(k:word):word;Var p:word;Begin p:=0;For i:=1 to
n do If x[i]>x[k] then Inc(p);Elem:=p;End;Begin В в о д  n, X
For i:=1 to n do y[i]:=Elem(i); П е ч а т ь  Y  End.
```

Не требуется комментариев, чтобы сделать вывод о том, что такую программу очень трудно читать и понимать.

С точки зрения ЭВМ не имеет значения форма написания исходной программы, лишь бы она была правильной. Но хотя программа выполняется на ЭВМ, все-таки читать, использовать и модифицировать ее приходится программисту. Поэтому одним из основных принципов в проектировании программ является следующий: программа составляется в первую

очередь для человека, а не для машины. В связи с этим в тексте программы обязательно должны содержаться средства, облегчающие ее понимание и улучшающие ее читабельность.

Исходный текст программы MakeArray1 занимает меньший объем памяти на диске по сравнению с программой MakeArray, однако это не должно служить критерием для выбора формы ее представления. Если возникает противоречие между экономией машинных ресурсов и читабельностью программы, то в этом случае нужно руководствоваться следующим: рабочее время программиста значительно дороже машинных ресурсов.

Специальных стандартов на тексты программ не существует. Тем не менее профессиональные программисты при написании программы придерживаются определенных правил, направленных на соблюдение указанного выше принципа программирования.

При написании текста программы рекомендуется выполнять приведенные ниже требования (в перечень включены также языковые средства, которые рассматриваются несколько позже).

1) Заголовки модуля и его секций, заголовки программы и каждого ее раздела (разделы ***Label, Const, Type, Var***, раздел описания процедур и функций, раздел операторов) начинать с первой позиции строки.

2) После заголовка модуля (***Unit***) указывать в комментарии основное назначение входящих в модуль процедур и функций.

3) После заголовка процедуры или функции приводить в комментарии краткое описание выполняемой ею работы.

4) Длинные тексты программы разделять комментариями на фрагменты. В комментарии кратко описывать работу фрагмента.

5) После объявления имени переменной в комментарии описывать назначение переменной.

6) Процедуры и функции отделять друг от друга пунктирной линией, заключенной в комментарий.

7) После слова ***End,*** завершающего блок процедуры или функции, в комментарии указывать имя данной процедуры или функции.

8) В качестве символов комментария использовать только фигурные скобки.

9) Каждое слово ***Begin*** начинать с новой строки.

10) Каждое слово ***End*** записывать с новой строки строго с той позиции, с которой начинается относящееся к нему слово ***Begin, Case*** или ***Record.***

11) Текст программы между словами ***Begin*** и ***End***, ***Case*** и ***End***, ***Record*** и ***End*** должен быть сдвинут на один отступ вправо. Отступ равен двум позициям строки.

12) Текст программы после слов ***do, then, else*** должен начинаться с новой строки, сдвинутой на один отступ вправо.

13) Каждое слово ***else*** должно быть расположено под тем словом ***if,*** к которому оно относится.

14) Разделы описания, как правило, располагать в порядке, предусмотренном стандартным языком Паскаль: ***Label, Const, Type, Var***, описание процедур и функций.

15) Имена переменных записывать, как правило, с большой буквы.

16) Имена переменных в программе в общем случае не должны отличаться от обозначений, принятых в исходной постановке задачи. Например, было бы нелогично обозначать в программе ускорение свободного падения именем *Z* вместо общепринятого обозначения *g*, даже снабдив это имя комментарием.

17) Отдельные части имени, состоящего из нескольких слов, выделять большими буквами (например, *FromSetToNumber*).

18) Длина строки программы не должна превышать 68 символов (для печати текста программы стандартным шрифтом на бумаге формата А4).

19) Оператор ***Goto*** разрешается применять в программе в двух случаях:

- для принудительного выхода из цикла;
- для перехода к далеко отстоящему фрагменту программы.

Совершенно недопустимо использовать оператор ***Goto*** для перехода снизу вверх.

Если оператор ***Goto*** производит переход на конец процедуры или функции, то его целесообразно заменить оператором *Exit*, а для принудительного выхода из цикла использовать оператор *Break.*

20) В программе вместо значений констант использовать, как правило, имена констант.

21) В разделе описания переменных использовать имена типов, а не описания типов. Например, вместо описания

```
Var  A : array[1..100] of string[40];
```

целесообразно использовать описание

```
Const  Nmax = 100;
Type  string40 = string[40];
      Ar = array[1..Nmax] of string40;
Var  A : Ar;
```

Последнее требование связано с тем, что в большой программе любая переменная может быть использована где-либо как фактический параметр при обращении к процедуре, а при таком обращении всегда требуется совпадение имени типа формального и фактического параметров. Неприятные последствия использования в разделе ***Var*** описания типа вместо имени типа могут проявиться и в других, более редких случаях.

22) Следует избегать различных программных "трюков", затрудняющих понимание программы.

Например, трудно понять работу такого фрагмента:

```
Var  x,y,z,w : real;
......
  If x>y then
    w:=0
  Else
    w:=y-x;
  z:=x+w;
```

Оказывается, это вычисление значения, равного большему из двух чисел: *x* и *y*.

В данном случае целесообразно написать:

```
If x>y then
  z:=x
Else
  z:=y;
```

Второй пример.

Для обмена значений двух переменных иногда записывают:

```
x:=x-y;  y:=x+y;  x:=y-x;
```

вместо более ясной схемы обмена "по треугольнику":

```
z:=x;  x:=y;  y:=z .
```

Программирование в соответствии с приведенными выше правилами часто именуется структурным программированием, а написанную в таком стиле программу называют структурированной.

ТОЖДЕСТВЕННЫЕ И СОВМЕСТИМЫЕ ТИПЫ

При передаче и преобразовании информации в программе должны соблюдаться некоторые требования, блокирующие некорректные действия программиста. Эти требования могут быть разделены на три группы, соответствующие следующим случаям:

- передача фактического параметра-переменной;
- вычисление выражения;
- выполнение оператора присваивания и передача фактического параметра-значения.

В первом случае должны быть выполнены требования тождественности типов, во втором – совместимости типов, в третьем – совместимости по присваиванию.

Тождественность типов.

Рассмотрим процедуру, в списке формальных параметров которой заданы лишь параметры-переменные (перед именем формального параметра записано слово ***Var***).

Пример 1.

```
Type  Ar1 = array[1..100] of real;
      Ar2 = array[1..100] of real;
      Ar3 = Ar1;
      Ar4 = Ar3;
Var  X : Ar1;  Y : Ar2;  Z : Ar3;  W : Ar4;
     a,b : real;
     m,n : integer;
Procedure Proc1(Var D:Ar1; Var k:integer);
Begin
.........................
End { Proc1 };
```

Ранее было указано, что формальный и соответствующий ему фактический параметр должны иметь одно и то же имя типа. Это не совсем точно.

Обозначим через *Type1* имя типа формального параметра, через *Type2* - имя типа соответствующего ему фактического параметра.

Как известно, при обращении к процедуре фиктивный адрес формального параметра-переменной замещается реальным адресом фактического параметра. В этом случае формальная и фактическая переменные соответствуют одному и тому же полю памяти. Следовательно, имена типов *Type1* и *Type2* этих переменных должны определять переменные одинакового размера и структуры, с одинаковым множеством допустимых значений и операций по их обработке. Последнее возможно, если типы *Type1* и *Type2* тождественны.

Два типа считаются тождественными, если они представляют собой одно и то же имя типа или один из них описан как эквивалентный другому типу.

В примере 1 типы *Ar1, Ar3* и *Ar4* тождественны, *Ar1* и *Ar2* - не тождественны, хотя они и имеют одинаковое описание типа. Поэтому обращения к процедуре *Proc1(X,m), Proc1(Z,m)* и *Proc1(W,m)* считаются правильными, а при обращении *Proc1(Y,m)* будет выведено сообщение "Type mismatch" ("Несоответствие типов").

Примечание. Вполне очевидно, что сказанное выше в равной мере относится также к параметрам-константам.

Совместимость типов. Рассмотрим пример 2.

Пример 2.

```
Var  x,y : real;
     m,n : integer;
     ch : char;
     b : boolean;
Begin  ..............
  y:=x+2*ch;
  b:=(x<ch) and n;
```

Вполне очевидно, что выражения в правой части операторов присваивания не могут быть вычислены, поскольку лишены смысла арифметические операции по отношению к

символьным переменным, сравнение числовых и символьных переменных и т.п. В этих выражениях нарушены требования совместимости типов.

Основные правила совместимости типов:

- операнды выражения имеют численные типы (вещественный, целочисленный, диапазонный);
- операнды определены логическим типом;
- операнды имеют строковый, символьный или диапазонный символьный типы.

Если операнды выражения имеют различные численные типы, то при выполнении арифметических операций производится преобразование их значений к более старшему типу в соответствии со следующими приоритетами:

- *real;*
- *longint;*
- *integer, word;*
- *shortint, byte.*

Совместимость типов по присваиванию. Рассмотрим пример 3.

Пример 3.

```
Var  x,y : real;
     m,n : integer;
     ch : char;
     b : boolean;
Begin  ..............
  { 1 } y:=(m<n) and b;
  { 2 } ch:=x+m;
  { 3 } m:=2*x+y;
  { 4 } y:=3*m-n;
```

Числовой переменной нельзя присвоить булевское значение, символьной переменной - численное значение. Поэтому операторы 1 и 2 не могут быть выполнены, здесь нарушается требование совместимости по присваиванию.

В правой части оператора 3 - вещественное значение, в левой части - целочисленная переменная. Если допустить выполнение такого оператора, то дробная часть вещественного значения должна быть отброшена, т.е. произошла бы потеря точности. В связи с этим считается, что в операторе 3 также нарушаются требования совместимости по присваиванию.

В операторе 4 потеря точности не наблюдается. При его выполнении производится лишь преобразование целочисленного значения, полученного при вычислении выражения в правой части оператора, к типу *real.*

Обозначим тип переменной в левой части *Type1*, тип значения выражения - *Type2.*

Основные требования совместимости по присваиванию:

- *Type1* и *Type2* имеют тождественные типы и ни один из них не является файловым типом;
- *Type1* и *Type2* - целочисленные типы;
- *Type1* - вещественный тип, *Type2* - вещественный или целочисленный тип;
- *Type1* и *Type2* - строковые типы;
- *Type1* - строковый тип, *Type2* - символьный тип.

Более жесткие требования должны соблюдаться, когда в левой и правой частях оператора присваивания записаны составные переменные, в частности, массивы.

Пример 4.

```
Type  Ar1 = array[1..100] of real;
```

```
      Ar2 = array[1..100] of real;
      Ar3 = Ar1;
Var   X,Y : Ar1;  Z : Ar2;  W : Ar3;
```

Здесь типы *Type1* и *Type2* должны быть тождественными. Следовательно, в этом случае операторы *Y:=X* и *Y:=W* являются допустимыми, в то время как для оператора *Y:=Z* будет определено несоответствие типов.

Рассмотрим теперь процедуру, в списке формальных параметров которой имеются параметры-значения.

Пример 5.

```
Var  x,y : real;
     m,n : integer;
Procedure Proc2(k:integer; r,t:real);
Begin
.............
End { Proc2 };
Begin
.............
  { 1 } Proc2(m,n-1,x+m);
  { 2 } Proc2(y,x+y,x);
```

Для параметра-значения в теле процедуры выделяется поле памяти в соответствии с его типом. При обращении к процедуре в это поле пересылается значение фактического параметра; другими словами, формальному параметру присваивается значение фактического параметра. Следовательно, по отношению к параметру-значению должны соблюдаться изложенные выше требования совместимости по присваиванию. В частности, в примере 5 для оператора 1 эти требования соблюдаются, для оператора 2 - не соблюдаются.

ПРИВЕДЕНИЕ ТИПОВ ПЕРЕМЕННЫХ

Имя любой переменной, объявленной в разделе ***Var***, - это адрес поля памяти в машинной программе. Оператор присваивания *a:=b*, рассматриваемый на уровне машинной программы, - это пересылка содержимого поля памяти *b* в поле памяти *a*.

При программировании на машинном языке или на языке ассемблера произвольному полю памяти можно присвоить значение любого другого поля памяти. В этом случае ответственность за правильность выполнения оператора присваивания полностью возлагается на программиста.

Иная ситуация имеет место при программировании на языках высокого уровня, в частности на языке Паскаль. Имя переменной, идентифицирующей адрес поля памяти, всегда имеет вполне определенный тип. Тип переменной однозначно определяет множество значений, которые имеет право принимать переменная данного типа, и набор операций, которые допустимы при ее обработке.

Пример.

```
Var  k,m : byte;
     p,q : char;
Begin
  k:=65; p:='A';
  1) m:=k; q:=p;
  2) m:=p; q:=k;
  3) m:=ord(p); q:=chr(k);
```

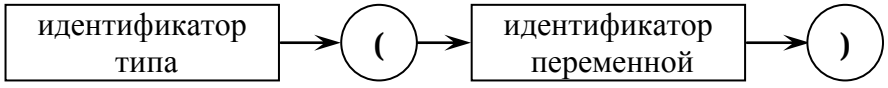
Внутреннее представление переменной *k* и переменной *p* совершенно одинаково (порядковый номер символа 'A' в таблице ASCII равен 65) и имеет вид 0110 0101.

Действие оператора *m:=k* сводится к пересылке содержимого однобайтного поля *k* в поле *m*. Аналогичные действия выполняются для оператора *q := p*. В то же время операторы *m:=p* и *q:=k* не могут быть выполнены, так как при их трансляции будет обнаружено несоответствие типов.

Пересылка содержимого поля *p* в поле *m* может быть выполнена оператором *m:=ord(p).* Функция *ord* не производит никаких преобразований значения переменной *p*, она лишь информирует транслятор, что значение этой переменной следует рассматривать как значение целочисленной переменной и переслать соответственно в поле *m*. Аналогичная ситуация имеет место для оператора *q:=chr(k).* Следовательно, функции *ord* и *chr* выполняют приведение типа, а не преобразование значений переменных.

При программировании на Паскале может возникать необходимость рассматривать одно и то же поле памяти как значения переменных самых различных типов, а не только типов *char* и *byte*. В этом случае используют аппарат приведения типа переменных, с помощью которого обращение к переменной одного типа может рассматриваться как обращение к переменной другого типа.

Синтаксис приведения типа:



Здесь поле памяти, определяемое идентификатором переменной, рассматривается как экземпляр типа, представленного идентификатором типа.

Рассмотрим суть приведения типов в Турбо Паскале на следующих трех примерах.

Пример 1.

```
Var  ch : char;
     b1,b2 : byte;
Begin
  ch:='A'; b1:=byte(ch); b2:=ord(ch);
  Writeln('b1=',b1,'   b2=',b2);
```

Будет отпечатано:

```
b1=65   b2=65
```

Выражение *byte*(*ch*) определяет "наложение" типа *byte* на переменную *ch.* В этом случае биты, содержащиеся в поле памяти *ch*, интерпретируются как значение переменной типа *byte* (*ch* = $01000001_2 = 41_{16} = 65_{10}$).

Примечание. Внешняя форма записи выражения *byte*(*ch*) и функции *ord*(*ch*) одинакова, что позволяет трактовать выражение *byte*(*ch*) как функцию приведения к типу *byte*.

Пример 2.

```
Type  ByteAr = array[1..2] of byte;
Var  W : word;
Begin
  ByteAr(W)[1]:=10; ByteAr(W)[2]:=20;
  Writeln(W,'   ',ByteAr(W)[1],'   ',ByteAr(W)[2]);
  W:=2500;
  Writeln(W,'   ',ByteAr(W)[1],'   ',ByteAr(W)[2]);
```

В результате работы программы будет отпечатано:

```
5130   10   20
2500   196   9
```

В примере 2 происходит наложение "формального" поля *ByteAr* длиной два байта на реальное поле *W*, также имеющее длину два байта. При этом байты, входящие в состав поля *W*, рассматриваются как отдельные элементы байтового массива типа *ByteAr*.

Отпечатанные выше элементы в 16 с/с имеют следующие значения:

```
140A      OA      14
09C4      C4      09
```

Соответствие между элементами полей *W* и *ByteAr* легко просматривается, если учесть, что в переменной типа *word* старшим считается правый байт. Последнее определяется тем, что в процессорах типа Intel числовое значение размещается в поле памяти таким образом, что более старшие разряды числа располагаются в байтах с более старшими адресами. Это положение иллюстрируется ниже на примере переменной *k* типа *longint.*

Переменной *k* выделяется 4 байта памяти. Поскольку адресом поля памяти является адрес его крайнего левого байта, то байты поля k получат следующие адреса:

k + 0	k + 1	k + 2	k + 3

Пусть $k = 2\ 000\ 100\ 101 = \$\ 77\ 37\ 1B\ 05$. Тогда в байтах с адресами k+0 .. k+3 шестнадцатеричные цифры значения переменной *k* будут расположены следующим образом:

k + 0	k + 1	k + 2	k + 3
05	1B	37	77

Приведение типа *T*(*v*), где *T* - имя типа, *v* - имя переменной, позволяет трактовать на машинном уровне биты, содержащиеся в поле *v*, как биты, принадлежащие значению типа *T*.

Пример 3. Определить численные значения байтов, входящих в состав полей памяти типов *integer* и *real*. Здесь фактически идет речь об определении внутреннего представления переменных типов *integer* и *real.*

```
Program PutType;
Type  ByteAr2 = array[1..2] of byte;
      ByteAr6 = array[1..6] of byte;
Var  I : integer;
     R : real;
Begin
   I:=5000; R:=5000;
   Writeln('  I:  ',ByteAr2(I)[1],'  ',ByteAr2(I)[2]);
   Writeln('  R:  ',ByteAr6(R)[1],'  ',ByteAr6(R)[2],
           '  ',ByteAr6(R)[3],'  ',ByteAr6(R)[4],
           '  ',ByteAr6(R)[5],'  ',ByteAr6(R)[6]);
   I:=-5000; R:=-5000;
   Writeln('  I:  ',ByteAr2(I)[1],'  ',ByteAr2(I)[2]);
   Writeln('  R:  ',ByteAr6(R)[1],'  ',ByteAr6(R)[2],
           '  ',ByteAr6(R)[3],'  ',ByteAr6(R)[4],
           '  ',ByteAr6(R)[5],'  ',ByteAr6(R)[6]);
End.
```

Результаты работы программы удобно представить в следующем виде:

```
I:  136  19                    |  I:  88  13
R:  141  0   0   0   64   28   |  R:  8D  00  00  00  40  1C
I:  120  236                   |  I:  78  EC
R:  141  0   0   0   64  156   |  R:  8D  00  00  00  40  9C
```

Здесь в левой части - печатаемые программой результаты (в десятичной системе счисления), в правой части - те же результаты в шестнадцатеричной системе счисления.

Проверка результатов:

5000 = $1388 ; -5000 = $EC78 ;

$1388 = 0,1388 \cdot 16^4 = 0,\ 0001\ 0011\ 1000\ 1000 \cdot 2^{16} = 0,1001110001000 \cdot 2^{13}$

Значение *R* в 16 с/с : 1C 40 00 00 00 8D.

Полученные результаты четко показывают, что числовые данные располагаются в памяти таким образом, чтобы более старшие разряды находились в байтах с адресами большей величины.

Пусть для вещественной переменной *R* компилятор отвел поле памяти с адресом *A*. Так как переменная типа *real* занимает 6 байтов, то в поле *R* входят байты с адресами *A*+0, *A*+1, *A*+2, *A*+3, *A*+4, *A*+5. Тогда для *R*=5000 байты будут иметь следующее содержимое:

(A+0) = 8D ; (A+1) = 00 ; (A+2) = 00 ; (A+3) = 00 ; (A+4) = 40 ; (A+5) = 1C .

Пример 4.

$$y = \begin{cases} 1, & \textit{если} \quad x < 0 \\ 0, & \textit{если} \quad x \ge 0 \end{cases}$$

```
Var  x : real;
y : byte;
```

Вместо условного оператора

```
If x<0 then
  y:=1
Else
  y:=0;
```

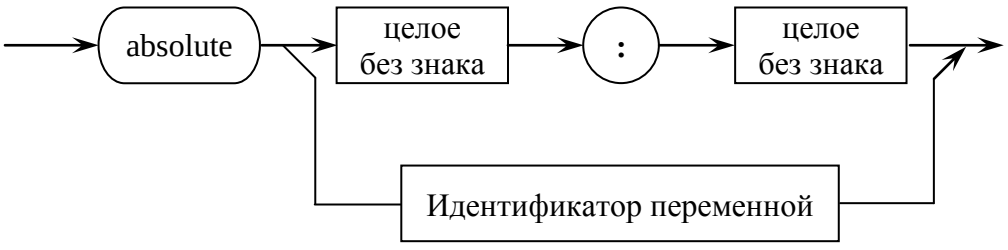
можно написать

```
y:=byte(x<0);
```

АБСОЛЮТНЫЕ ПЕРЕМЕННЫЕ

Как известно, память для переменных, объявленных в разделе ***Var,*** выделяется при старте блока, в котором содержится данный раздел, в порядке объявления имен переменных. Местоположение блока в пространстве оперативной памяти ЭВМ определяет операционная система. В то же время Турбо Паскаль предоставляет возможность программисту так описать переменные, чтобы им были назначены заранее заданные адреса. Такие переменные называются абсолютными; их описание после имени типа должно содержать так называемое абсолютное предложение.

Синтаксис абсолютного предложения:



Структура абсолютного предложения имеет два альтернативных варианта.

В первом варианте производится задание абсолютного адреса в форме, принятой в операционной системе MS DOS (адрес сегмента и смещение - см.раздел "Адресация операндов").

Пример 1.

```
Type  ScreenChar = record
          Symbol : char;
          Attrib : byte;
       end;
       ScreenAr = array[1..25,1..80] of ScreenChar;
Var   y : real absolute $A100:$0000;
      Screen : ScreenAr absolute $B800:$0000;
```

Здесь переменным *y* и *Screen* назначаются конкретные адреса оперативной памяти. В частности, значение $B800:$0000 определяет фиксированный адрес видеопамяти.

Примечание. В конструкции ***record*** .. ***end*** объединяются переменные различного типа (см.раздел "Записи").

Во втором варианте для абсолютной переменной назначается адрес другой, ранее объявленной переменной.

Пример 2 (аналогично примеру 2 раздела "Приведение типов переменных").

```
Type  ByteAr = array[1..2] of byte;
Var   W : word;
      Br : ByteAr absolute W;
Begin
  Br[1]:=10; Br[2]:=20;
  Writeln(W,'   ',Br[1],'   ',Br[2]);
  W:=2500;
  Writeln(W,'   ',Br[1],'   ',Br[2]);
```

В результате работы программы будет отпечатано:

```
 5130    10   20
 2500   196    9
```

Переменные *W* и *Br* адресуют одно и то же поле памяти. Другими словами, одно и то же поле памяти размером два байта имеет два имени: имя *W* и имя *Br*. Если в программе используется имя *W*, то это поле можно обрабатывать как переменную типа *word*, для имени *Br* допустимые операции определяются именем типа *ByteAr*.

При совмещении переменных с помощью директивы *absolute* не рекомендуется в разделе ***Var*** использовать описание типа вместо имени типа. Нарушение этого правила не анализируется компилятором, но может привести к неправильной работе программы, вплоть до ее "зависания", особенно при использовании сопроцессора.

Например, вместо описания

```
Var   A : array[1..6] of longint;
      B : array[1..4] of real absolute A;
```

следует применять описание

```
Type  ArLong = array[1..6] of longint;
      ArReal = array[1..4] of real;
Var   A : ArLong;
      B : ArReal absolute A;
```

ОБРАБОТКА В ПРОЦЕДУРЕ ОДНОМЕРНЫХ МАССИВОВ С РАЗЛИЧНЫМИ ИМЕНАМИ ТИПОВ

Практически для каждого языка программирования создаются пакеты прикладных программ (ППП), реализующих часто встречающиеся процедуры численного анализа (решение нелинейных уравнений, нахождение корней полиномов, обращение матрицы, решение системы дифференциальных уравнений и т.п.). Прикладная программа разрабатывается в виде подпрограммы, обращение к которой производится из программы пользователя. При этом, как правило, ППП поставляется в виде объектных модулей, исключающих возможность изменения их текста пользователем.

Паскаль-подпрограмма, обрабатывающая массив, должна содержать в списке формальных параметров имя этого массива с указанием соответствующего имени типа. В общем случае имя типа формального массива не совпадает с именем типа фактического массива в программе пользователя. Следовательно, прикладная подпрограмма должна обеспечивать совместимость типов формального и фактического массивов.

В Паскаль-программе массив всегда имеет фиксированный размер, определяемый его именем типа. Необходимость использования различных имен типов связана главным образом с тем, что формальный и фактический массивы в общем случае имеют различные размеры. Типы элементов этих массивов, естественно, должны быть одинаковыми.

Пример 1. Предположим, что для массивов *X, Y* и *Z*, имеющих различные имена типов, необходимо вычислить среднее арифметическое их элементов. Тогда программа может иметь следующий вид.

```
Program Middle1;
Type  Xar = array[1..50] of real;
      Yar = array[1..500] of real;
      Zar = array[1..5000] of real;
Var  i,nx,ny,nz : integer;
     Sx,Sy,Sz : real;
     X : Xar;
     Y : Yar;
     Z : Zar;
Procedure MiddleAr(Var Buf:Xar; Var S:real; n:integer);
Var  i : integer;
Begin
  S:=0;
  For i:=1 to n do
    S:=S+Buf[i];
  S:=S/n;
End { MiddleAr };
Begin
  Ввод и печать  nx, ny, nz, X, Y, Z
  MiddleAr(X,Sx,nx);
  MiddleAr(Y,Sy,ny);
  MiddleAr(Z,Sz,nz);
  Печать  Sx, Sy, Sz
End.
```

Здесь при трансляции программы будет правильно воспринято лишь первое обращение к процедуре *MiddleAr;* для остальных обращений будет выдано сообщение о несоответствии типов фактических параметров *Y, Z* и формального параметра *Buf.*

Обработка массивов различного размера (более точно, массивов с разными именами типов) в Турбо Паскале может быть обеспечена путем использования аппарата приведения типов переменных или с помощью абсолютных переменных.

В связи с тем, что фактический параметр-массив при разных обращениях к процедуре может иметь различные имена типов, то ему не может соответствовать ни один формальный параметр с конкретным именем типа. Для обеспечения совместимости типов в процедуре должен указываться формальный параметр без типа. Такому формальному параметру могут соответствовать фактические параметры любого типа.

Формальному параметру без типа имеет смысл передавать лишь адрес, а не значение фактической переменной, поскольку адрес переменной всегда имеет постоянную длину (4 байта), а значение переменной имеет длину, зависящую от типа этой переменной. Следовательно, формальный параметр без типа обязательно должен иметь слово ***Var*** в заголовке процедуры (это так называемый нетипизированный параметр-переменная).

С помощью аппарата приведения типа в процедуре можно организовать обработку массивов с различными именами типов, т.е. массивов, имеющих в общем случае различный размер. При этом элементы указанных массивов должны быть, естественно, одного и того же типа. В частности, программа Middle1 тогда может иметь следующий вид:

```
Program Middle2;
Type  Xar = array[1..50] of real;
      Yar = array[1..500] of real;
      Zar = array[1..5000] of real;
Var  i,nx,ny,nz : integer;
     Sx,Sy,Sz : real;
     X : Xar;
     Y : Yar;
     Z : Zar;
{ ---------------------------------------------- }
Procedure MiddleAr(Var Buf; Var S:real; n:integer);
Type  RealAr = array[1..10000] of real;
Var  i : integer;
Begin
  S:=0;
  For i:=1 to n do
    S:=S+RealAr(Buf)[i];
  S:=S/n;
End { MiddleAr };
{ ---------------------------------------------- }
Begin
  Ввод и печать  nx, ny, nz, X, Y, Z
  MiddleAr(X,Sx,nx);
  MiddleAr(Y,Sy,ny);
  MiddleAr(Z,Sz,nz);
  Печать  Sx, Sy, Sz
End.
```

Хотя "интерпретирующее" поле *RealAr*, накладываемое на формальную переменную *Buf* без типа, длиннее фактических переменных *X, Y* или *Z,* в процедуре *MiddleAr* выхода за пределы реальных массивов не произойдет, если переменные *nx, ny, nz* не превышают соответственно значений 50, 500, 5000.

Более предпочтительным для *RealAr* является объявление вида

```
RealAr = array[1..(2*MaxInt) div SizeOf(real)] of real.
```

Здесь компилятор определяет для *RealAr* максимально возможный размер, самостоятельно вычисляя количество элементов такого массива ($2 \cdot 32767$ ***div*** 6 = 10992).

Тип *RealAr* в процедуре *MiddleAr* накладывается на формальный параметр *Buf* без имени типа, который при обращении к процедуре заменяется именем фактического массива. Так как *RealAr* - это имя типа, а не описание переменной, то объекту с именем *RealAr* никакой памяти не выделяется. Как и любое имя типа, *RealAr* определяет множество значений, которые может принимать переменная с этим именем типа.

Как уже было отмечено, для формального параметра *Buf* в процедуре *MiddleAr* не указано никакого имени типа. Если бы в этой процедуре было записано $S := S + Buf[i]$, то транслятор выдал бы сообщение об ошибке, поскольку в списке формальных параметров нет информации о том, что собой представляет идентификатор *Buf* (простая переменная, массив и т.п.). Можно было бы возразить, что эту информацию транслятор мог бы получить из обращения к процедуре *MiddleAr*. Но ведь описание процедуры предшествует разделу операторов. Следовательно, при трансляции этой процедуры еще не прочитаны обращения к ней, и тип переменной *Buf* остается неопределенным.

Запись приведенного выше оператора в форме $S := S + RealAr(Buf)[i]$ снимает такое недоразумение, поскольку транслятору в этом случае указано, что имя *Buf* следует рассматривать как имя одномерного массива с вещественными компонентами.

Абсолютные переменные можно использовать в процедурах для обработки массивов с разными именами типов аналогично аппарату приведения типов.

Пример 2. В одномерном целочисленном массиве удалить повторяющиеся элементы, оставляя в составе массива лишь первое включение таких элементов. Программу решения задачи оформить в виде процедуры, обрабатывающей массивы с различными именами типов.

Вначале составим отдельную программу обработки одного массива.

Для заданного массива *X* будем использовать буферный массив *Y*, в который пересылаются те элементы массива *X*, которые еще не записаны в массив *Y*. При этом в программе следует учесть, что первый элемент массива X, т.е. элемент x_1, в любом случае должен быть записан в массив Y.

```
Program DelElem;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  X,Y : Ar;
     i,              { параметр цикла }
     n,              { реальный размер массива X }
     m : word;       { реальный размер массива Y }
{ ---------------------------------------- }
Function FindElem(r:integer):boolean;
{ Определение признака включения элемента в массив Y }
Var  i : word;
Begin
  FindElem:=false;
  For i:=1 to m do
    If r=y[i] then
      Begin
        FindElem:=true; Exit
      End;
End { FindElem };
{ ---------------------------------------- }
Begin
  Ввод  и  печать  n, X
  m:=1; y[1]:=x[1];
  For i:=2 to n do
    If not FindElem(x[i]) then  { элемент x[i] записывается }
```

```
      Begin                          { в массив Y,  если  в этом }
        Inc(m); y[m]:=x[i]           { массиве  еще  нет  такого }
      End;                           { элемента }
  X:=Y; n:=m;
  Печать  n,X
End.
```

В программе DelElem создается буферный (временный) массив *Y*, необходимый лишь на период обработки массива *X* в соответствии с использованным здесь алгоритмом. Размер массива *Y* в соответствии с его объявлением равен размеру массива *X* (учитывается частный случай, когда в массиве *X* нет одинаковых элементов). Если программу DelElem оформлять в виде процедуры, то массив *Y* должен создаваться в процессе работы этой процедуры, т.е. переменная *Y* должна быть локальной. Но, как было оговорено в условии задачи, процедура *DelElem* должна обрабатывать массивы с различными именами типов, имеющими в общем случае различные размеры. Поскольку локальный массив, объявляемый в процедуре, должен иметь вполне определенное, заранее указанное в тексте программы, имя типа, то обеспечить размер такого массива равным размеру обрабатываемого массива *X*, невозможно. Следовательно, алгоритм, использованный в программе DelElem, не позволяет оформить эту программу в виде процедуры для обработки массивов с различными именами типов. В данном случае необходимо разработать такой алгоритм решения задачи, для реализации которого не требуется дополнительный буферный массив.

В процедуре *DelElems*, приведенной ниже, производится последовательный просмотр элементов массива *A*, совмещенного по адресу с формальным массивом *U*, и, в случае обнаружения повторяющегося элемента, производится его удаление путем сдвига оставшейся части массива на один элемент влево.

```
Program Delar;
Type  Xar = array[1..50] of integer;
      Yar = array[1..500] of integer;
      Zar = array[1..5000] of integer;
      RealAr = array[1..2*MaxInt div SizeOf(integer)]
               of integer;
Var  i,nx,ny,nz : word;
     X : Xar; Y : Yar; Z : Zar;
     FileX,FileY,FileZ : text;
{ ------------------------------------------- }
Procedure ReadVector(Var F:text; Var U; Var n:word);
Var  A : RealAr absolute U;
Begin                          { Чтение массива }
  Reset(F);                    { из текстового файла }
  n:=0;
  While not eof(F) do
    Begin
      Inc(n); Read(F,a[n])
    End;
End { ReadVector };
{ ------------------------------------------- }
Procedure WriteVector(Var U; n:word);
Var  i,k : word;
     A : RealAr absolute U;
Begin
  k:=0;                        { Вывод массива }
  For i:=1 to n do             { на экран }
    Begin
      Inc(k);
      If k<10 then
```

```
        Write(a[i]:6,'  ')
      Else
        Begin
          k:=0; Writeln(a[i]:6);
        End;
    End;
  If k>0 then Writeln;
End { WriteVector };
{ ---------------------------------------------- }
Procedure DelElems(Var U; Var n:word);
Var  i,j,k,R : integer;
     A : RealAr absolute U;
Begin
  i:=1;
  While i<n do              { Удаление всех }
    Begin                   { повторяющихся }
      R:=a[i]; j:=i+1;      { элементов, кроме }
      While j<=n do         { первого такого }
        If R=a[j] then      { элемента }
          Begin
            For k:=j to n-1 do
               a[k]:=a[k+1];
            Dec(n);
          End
        Else
          Inc(j);
      Inc(i);
    End;
End { DelElems };
{ ---------------------------------------------- }
Begin
  ReadVector(FileX,X,nx);
  WriteVector(X,nx);
  DelElems(X,nx);
  WriteVector(X,nx);

  ReadVector(FileY,Y,ny);
  WriteVector(Y,ny);
  DelElems(Y,ny);
  WriteVector(Y,ny);

  ReadVector(FileZ,Z,nz);
  WriteVector(Z,nz);
  DelElems(Z,nz);
  WriteVector(Z,nz);
End.
```

ОБРАБОТКА В ПРОЦЕДУРЕ МАТРИЦ С РАЗЛИЧНЫМИ ИМЕНАМИ ТИПОВ

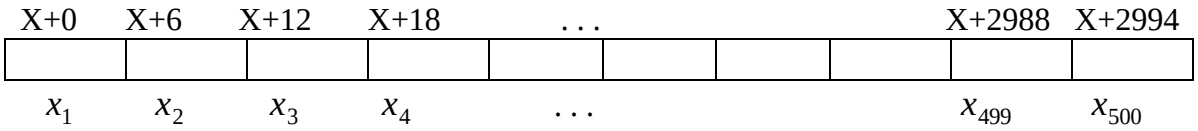
Обработка матриц различного размера (с разными именами типов) организуется в процедуре с помощью аппарата приведения типов или с помощью абсолютных переменных в основном аналогично тому, как это выполняется для одномерных массивов. Разница в их обработке связана с тем, что внутреннее представление одномерного массива и матрицы неодинаково.

Элементы одномерного массива расположены в памяти последовательно.

Пример 1.

```
Type  RealAr = array[1..1000] of real;
      Xar = array[1..500] of real;
Var   X : Xar;
      Y : RealAr;
      c,d : real;
Begin  ..............
  c:=X[100];
  d:=RealAr(X)[100];
```

При старте программы выделяется память для переменных *X, Y, c* и *d*, соответственно 3000, 6000, 6 и 6 байтов. Поле *X* имеет следующую структуру:



На этой схеме снизу указаны элементы массива *X,* сверху - их адреса. Каждый элемент x_i занимает 6 байтов памяти. В частности, элемент x_{100} имеет адрес $X+(100-1) \cdot 6 = X+594$.

Поле *Y* имеет аналогичную структуру, последний элемент y_{1000} расположен по адресу *Y*+5994.

Запись *RealAr*(*X*) означает, что в данном случае поле *X* интерпретируется как переменная, имеющая тип *RealAr*. Поскольку описание *RealAr* определяет одномерный массив с элементами типа *real*, то элемент *RealAr*(*X*)[100] имеет тот же адрес, что и элемент x_{100}. Следовательно, переменные *c* и *d* в приведенном выше примере получают одинаковые значения.

Элементы матрицы расположены в памяти по строкам.

Пример 2.

```
Type  Rem = array[1..100,1..100] of real;
      MatrixA = array[1..20,1..30] of real;
Var   A : MatrixA;  B : Rem;
      c,d : real;
Begin  ..............
  c:=A[10,10];
  d:=Rem(A)[10,10];
```

Переменные *A* и *Rem*(*A*) - это одно и то же поле памяти. Элементы матрицы *A* расположены в памяти согласно трафарета, определенного именем типа *MatrixA,* т.е.

```
a[1,1],     a[1,2],  ... ,   a[1,29],     a[1,30],
a[2,1],     a[2,2],  ... ,   a[2,29],     a[2,30],
.................................................
a[20,1],    a[20,2],  ... ,  a[20,29],    a[20,30]
```

Следовательно, элемент $a_{10,10}$ имеет порядковый номер $9 \cdot 30 + 10 = 280$, его адрес равен $A + (280-1) \cdot 6 = A + 1674$.

Расположение в памяти элементов матрицы *Rem*(*A*) определено именем типа *Rem:*

```
Rem(A)[1,1],  Rem(A)[1,2], ..., Rem(A)[1,99],  Rem(A)[1,100],
Rem(A)[2,1],  Rem(A)[2,2], ..., Rem(A)[2,99],  Rem(A)[2,100],
.....................................................
Rem(A)[100,1],Rem(A)[100,2],...,Rem(A)[100,99],Rem(A)[100,100]
```

Элемент *Rem*(*A*)[10,10] имеет порядковый номер $9 \cdot 100 + 10 = 910$, его адрес равен $A + (910-1)*6 = A + 5454$.

Следовательно, переменным *c* и *d* в примере 2 будут присвоены различные значения.

Совпадение значений *a*[*i,j*] и *Rem*(*A*)[*i,j*] будет иметь место лишь в том случае, когда пределы изменения индексов фактической и наложенной матриц одни и те же. Естественно, при этих условиях процедура не сможет правильно обрабатывать матрицы разного размера.

Чтобы обеспечить в процедуре обработку матриц разного размера, в теле процедуры на фактическую матрицу накладывается фиктивный одномерный массив. Тогда при переходе от фактического к фиктивному (наложенному) массиву по индексам *i, j* матрицы вычисляется порядковый номер *k* одномерного массива, а при обратном переходе - по значению *k* вычисляются индексы *i, j*.

Пусть фактическая матрица *A* имеет объявление типа

```
Matrix = array[1..Mmax,1..Nmax] of real.
```

Тогда порядковый номер элемента *a*[*i,j*]: $k = (i - 1) \cdot Nmax + j$.

Наоборот, элемент с порядковым номером *k* имеет индексы

$$i = (k - 1) \ \textbf{div} \ Nmax + 1; \quad j = k - (i - 1) \cdot Nmax.$$

Пример 3. Разработать процедуру, определяющую значение и местоположение максимального элемента для матриц разного размера.

```
Program MaxElem;
Const  MmaxA = 30; NmaxA = 30;
       MmaxB = 20; NmaxB = 40;
Type  MatrixA = array[1..MmaxA,1..NmaxA] of real;
      MatrixB = array[1..MmaxB,1..NmaxB] of real;
      RealAr = array[1..(2*MaxInt) div SizeOf(real)] of real;
Var  i,j,                  { параметры цикла }
     ma,na,mb,nb,          { размеры матриц A и B}
     imaxA,jmaxA,          { индексы макс.элемента }
     imaxB,jmaxB : byte;   { для матриц A и B }
     Amax,Bmax : real;     { значение макс.элемента }
     A : MatrixA; B : MatrixB;
     FileA,FileB : text;
{ ------------------------------------------------ }
Procedure ReadMatrix(Var F:text; Var C; Nmax:byte; Var m,n:byte);
{ Чтение матрицы из файла }
Var  i,j : byte;
     k : word;
     R : real;
Begin
  Reset(F);
  Read(F,m,n);
  For i:=1 to m do
    For j:=1 to n do
      Begin
        Read(FileX,R);
        k:=(i-1)*Nmax+j; RealAr(C)[k]:=R;
      End;
  Close(F);
End { ReadMatrix };
{ ------------------------------------------------ }
Procedure MaxMatrix(Var C; Nmax,m,n:byte; Var Cmax:real;
```

```
                        Var imax,jmax:byte);
{ Определение значения и местоположения макс.эл-та матрицы }
Var i,j : byte;
    k : word;
Begin
  Cmax:=RealAr(C)[1]; imax:=1; jmax:=1;
  For i:=1 to m do
    For j:=1 to n do
      Begin
        k:=(i-1)*Nmax+j;
        If RealAr(C)[k]>Cmax then
          Begin
            Cmax:=RealAr(C)[k]; imax:=i; jmax:=j;
          End;
      End;
End { MaxMatrix };
{ ---------------------------------------------- }

Begin
  Assign(FileA,'MatrixA.dat');
  ReadMatrix(FileA,A,NmaxA,ma,na):
  MaxMatrix(A,NmaxA,ma,na,Amax,imaxA,jmaxA);
  Writeln('Amax= ',Amax:8:2,'   imaxA= ',imaxA,
          '   jmaxA= ',jmaxA);
  Assign(FileB,'MatrixB.dat');
  ReadMatrix(FileB,B,NmaxB,mb,nb);
  MaxMatrix(B,NmaxB,mb,nb,Bmax,imaxB,jmaxB);
  Writeln('Bmax= ',Bmax:8:2,'   imaxB= ',imaxB,
          '   jmaxB= ',jmaxB);
End.
```

В примере 3 для обработки матриц с разными именами типов применялся аппарат приведения типов. В следующем примере с этой же целью используются абсолютные переменные.

Пример 4. Разработать процедуру, определяющую значение и местоположение максимального элемента среди элементов, расположенных ниже главной диагонали квадратной матрицы.

```
Program MaxMatrix;
Const  NmaxA = 30; NmaxB = 40;
Type   MatrixA = array[1..NmaxA,1..NmaxA] of real;
       MatrixB = array[1..NmaxB,1..NmaxB] of real;
       RealAr = array[1..(2*MaxInt) div SizeOf(real)] of real;
Var   i,j,                  { параметры цикла }
      na,nb,                { размеры матриц A и B }
      imaxA,                { индексы макс.элемента }
      jmaxA,                { для матрицы A }
      imaxB,                { индексы макс.элемента }
      jmaxB : byte;         { для матрицы B }
      Amax,Bmax : real;     { значение макс.элемента }
      A : MatrixA; B : MatrixB;
      FileA,FileB : text;
{ ---------------------------------------------- }

Procedure ReadMatrix(Var F:text; Var C; Nmax:byte;
```

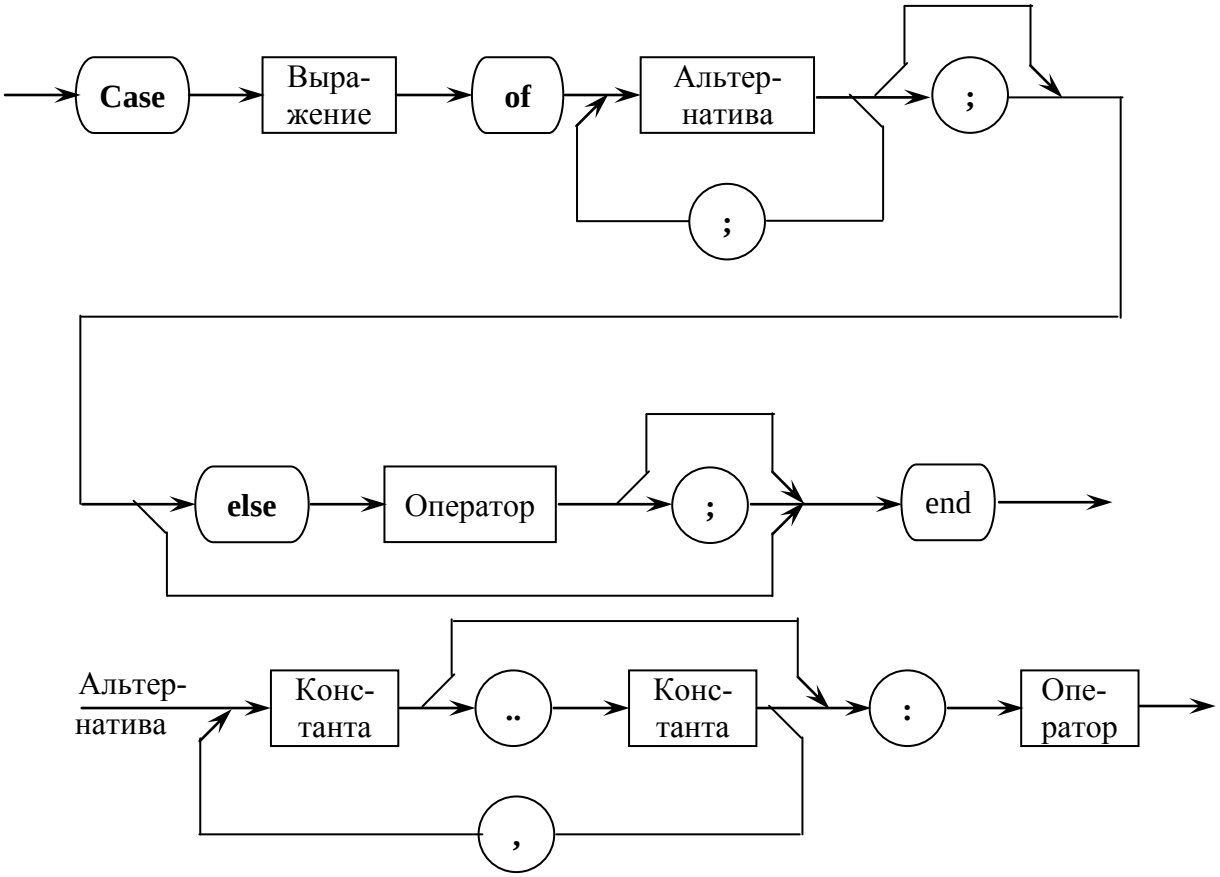
```
                      Var n:byte);
{ Чтение матрицы из файла }
Var  i,j : byte;
     k : word;
     R : real;
     D : RealAr absolute C;
Begin
  Reset(F); Read(F,n);
  For i:=1 to n do
    For j:=1 to n do
      Begin
        Read(F,R);
        k:=(i-1)*Nmax+j; d[k]:=R;
      End;
  Close(F);
End { ReadMatrix };
{ ------------------------------------------- }

Procedure MaxElem(Var C; Nmax, n:byte; Var max:real;
                  Var imax,jmax:byte);
{ Определение значения и местоположения максимального }
{ элемента матрицы ниже главной диагонали             }
Var  i,j : byte;
     D : RealAr absolute C;
Begin
  max:=d[Nmax+1]; imax:=2; jmax:=1;  { первый элемент }
  For i:=3 to n do                   { второй строки }
    For j:=1 to i-1 do
      Begin
        k:=(i-1)*Nmax+j;
        If d[k]>max then
          Begin
            max:=d[k]; imax:=i; jmax:=j;
          End;
      End;
End { MaxElem };
{ ------------------------------------------- }

Begin
  Assign(FileA,'MatrixA.dat');
  ReadMatrix(FileA,A,NmaxA,na):
  MaxMatrix(A,NmaxA,na,Amax,imaxA,jmaxA);
  Writeln('Amax= ',Amax:8:2,'  imaxA= ',imaxA,
          '  jmaxA= ',jmaxA);
  Assign(FileB, 'MatrixB.dat');
  ReadMatrix(FileB,B,NmaxB,nb):
  MaxMatrix(B,NmaxB,nb,Bmax,imaxB,jmaxB);
  Writeln('Bmax= ',Bmax:8:2,'  imaxB= ',imaxB,
          '  jmaxB= ',jmaxB);
End.
```

ОПЕРАТОР ВАРИАНТА (ОПЕРАТОР ВЫБОРА)

Условный оператор выбирает одну из двух альтернатив. Если альтернатив более двух, удобно использовать оператор варианта (оператор **Case**).

Примечание. Слово «case» по-английски означает «случай».

Синтаксис оператора **Case:**



Выражение, входящее в состав оператора варианта, называют селектором. Оператор варианта состоит из селектора и списка операторов, с каждым из которых сопоставляется одна или несколько констант, относящихся к типу селектора. Тип селектора должен быть ординальным.

Оператор варианта выбирает для выполнения оператор, сопоставленный с текущим значением селектора. Если такая константа не обнаружена, выполняется ветвь ***Else,*** при отсутствии которой оператор варианта эквивалентен пустому оператору.

В списке констант допускается использование диапазонов.

Пример 1.

```
Var  k : 1..10;
Begin
..........
  Case k of
       1 : y:=x+1;
       2 : y:=sin(x);
    3..5 : y:=sqrt(x);
       6 : y:=exp(x);
       7 : y:=sqr(x);
  end;
```

Следует особо отметить, что в операторе варианта перед двоеточием стоит не метка, а константа ординального типа.

Пример 2.

```
Var  ch : char;
```

```
Begin
  Case ch of
     'a'         : Writeln('Первая ветвь');
     'b','c','d' : Writeln('Вторая ветвь');
     'p'..'t'    : Writeln('Третья ветвь')
    Else
      Writeln('Альтернативная ветвь');
  end;
```

Примечание. В соответствии с синтаксической диаграммой перед словом ***else*** в операторе ***Case,*** в отличие от условного оператора, может стоять точка с запятой. Следовательно, в предыдущем примере оператор ***Case*** можно записать в виде

```
  Case ch of
     'a'         : Writeln('Первая ветвь');
     'b','c','d' : Writeln('Вторая ветвь');
     'p'..'t'    : Writeln('Третья ветвь');
   Else
     Writeln('Альтернативная ветвь');
end;
```

МАССИВЫ СИМВОЛОВ

Строки, состоящие из одного символа, считаются константами типа *char*. Например, 'a', 'A', '1', '=' .

Строки, состоящие из *n* символов, считаются константами типа

array[1..n] **of** char.

По отношению к символьным массивам действуют в основном те же правила, что и по отношению к числовым массивам. В частности, присваивание *A* := *B* допустимо лишь в том случае, когда типы массивов *A* и *B* тождественны.

Пример 1.

```
Type Ar1 = array[1..10] of char;
     Ar2 = array[1..10] of char;
     Ar3 = Ar1;
Var  A : Ar1;  B : Ar2;  C : Ar3;
Begin
  ......................
  B:=A;  C:=A;
```

Здесь оператор C:=A будет считаться корректным, а для оператора B:=A компилятор выдаст сообщение «Type mismatch», поскольку типы Ar1 и Ar2 не являются тождественными.

Тем не менее имеются и отличия в обработке символьных и числовых массивов.

В частности, символьному массиву можно присвоить значение строки символов, если количество символов в строке и объявленный размер символьного массива одинаковы. Например, для предыдущего примера правильным будет воспринят оператор A:=’0123456789’, а для операторов B:=’01234567’ и C:=’0123456789012’ будет выдано сообщение «Type mismatch».

Для символьных массивов допустимы также операции отношения. Такие операции выполняются слева направо до тех пор, пока не встретятся два неодинаковых символа. Если длины массивов различные, то более короткий дополняется символом #0.

Пример 2.

```
Var  p : boolean;
     A,B,C : array[1..8] of char;
Begin
  A:='ABCDEFGH'; B:='ABCDXYZV'; C:=B;
  p:=A<C;
```

В данном случае получим *p = true.*

Пример 3. Слова в тексте разделены одним или несколькими пробелами. Определить, сколько слов оканчиваются буквой *'a'*.

```
Program Letter;
Const  Nmax = 60;
Var  i,k,n : integer;
     Tex : array[1..Nmax] of char;
Begin
  Read(n);
  For i:=1 to n do          { Ввод текста }
    Read(Tex[i]);           { по одному символу }
  For i:=1 to n do          { Вывод текста }
    Write(Tex[i]);          { по одному символу }
  Writeln;
  k:=0;
  For i:=1 to n-1 do         { если после буквы }
    If (Tex[i]='a') and      { следует пробел, то }
       (Tex[i+1]=' ') then   { это означает, что }
      Inc(k);                { данная буква является }
  If Tex[n]='a' then         { последней в слове }
    Inc(k);
  Writeln('k= ',k);
End.
```

Поскольку текст - это массив символов, то ввод и вывод такого текста осуществляется поэлементно, по одному символу.

Пример 4. Даны два слова одинаковой длины. Определить, состоят ли эти слова из одних и тех же букв. Например, 'СОСНА' = 'НАСОС', но 'СОСНА' ≠ 'НАНОС', так как в слове 'СОСНА' две буквы 'С', а в слове 'НАНОС' - только одна такая буква.

Обозначим заданные слова именами *Slovo1* и *Slovo2*. Будем выбирать по одной букве из *Slovo2* и последовательно сравнивать выбранную букву со всеми буквами в *Slovo1*. Если *i*-ая буква в *Slovo2* равна *j*-ой букве в *Slovo1*, то *j*-ую букву первого слова заменяем на пробел (чтобы эта буква больше не анализировалась при наличии в *Slovo2* повторяющихся букв). Если равенство *Slovo1*[*j*] = *Slovo2*[*i*] не обнаружено, дальнейшая проверка должна быть прекращена.

Обычно в программах не рекомендуется искажать исходные данные (в данном случае в процессе работы программы стираются буквы в *Slovo1*). Поэтому сравнение *Slovo2* будет производиться с буферным (рабочим) словом *SlovoBuf*, в которое на стартовом участке программы копируется значение *Slovo1*.

```
Program Slovo;
Const  n = 20;
Var  i,j : byte;
     Cond : boolean;
     Slovo1,Slovo2,SlovoBuf : array[1..n] of char;
Begin
```

```
  Ввод и печать Slovo1,Slovo2
  SlovoBuf:=Slovo1;
  For i:=1 to n do
    Begin
      Cond:=false;
      For j:=1 to n do
        If Slovo2[i]=SlovoBuf[j] then
          Begin
            SlovoBuf[j]:=' '; Cond:=true;
            Break;
          End;
      If not Cond then
        Break;
    End;
  If Cond then
    Writeln('Слова имеют одинаковые буквы')
  Else
    Writeln('Слова имеют различные буквы');
End.
```

Здесь следует обратить внимание на то, что каждый оператор Break осуществляет выход лишь за пределы того цикла, в котором он сам находится.

СТРОКИ

Обработка текста в виде массива символов сопряжена с рядом недостатков.

1. Символьный массив обрабатывается в цикле по одному элементу, вследствие чего программа становится громоздкой.

2. Символьному массиву можно присвоить значение строки символов лишь в том случае, когда длина строки и объявленный размер массива одинаковы.

Чтобы исключить указанные недостатки, в Турбо Паскале введен тип данных ***string.*** Переменная типа ***string*** в отличие от массива символов может принимать значения переменной длины. При объявлении такой переменной указывается ее максимальная длина.

Объявление строки имеет вид:

```
Var  S : string[v],
```

где *v* - атрибут длины, представляющий собой целочисленное выражение.

Каждый символ строки занимает один байт.

Атрибут длины может принимать значение от 0 до 255. При *v* = 0 имеем строку нулевой длины, т.е. пустую строку. Если атрибут длины не указан в объявлении строки, то по умолчанию он принимается равным 255.

Если строка объявлена в виде ***string***[*n*], то в памяти для нее выделяется *n*+1 байт, которые имеют номера 0, 1, ... , *n*. В нулевом байте хранится текущая длина строки.

Память строке, как и другим переменным, выделяется в соответствии с описанием ее типа. Однако, как и в массиве, эта память может использоваться не полностью. При объявлении ***string***[*Nmax*] текущее количество символов строки, обрабатываемое в программе, может принимать значение $0 \le n \le Nmax$. На это количество указывает содержимое нулевого байта строки. Поскольку численное значение байта изменяется в диапазоне 0 .. 255, то это и определяет минимальное и максимальное количество символов в строке.

Примечание. Хотя содержимое нулевого байта, определяющее текущую длину строки, по физическому смыслу – число, но этот байт, как и остальные байты строки, считается в программе символом. Поэтому более точно можно сказать, что текущая длина строки – это порядковый номер символа, записанного в нулевом байте.

Пример 1.

```
Var  S1,S2 : string[20];
     S3 : string;
Begin
  S1:='abc'; S2:='';
```

Строкам *S1* и *S2* выделяется по 21 байту памяти, строке *S3* - 256 байтов. Текущая длина строки *S1* равна 3 символа, строки *S2* - 0 символов (пустая строка); текущая длина строки *S3* неопределенная, поскольку этой строке не присвоено никакого конкретного значения.

Пример 2.

```
Var  S1 : string[10];
     S2 : string[5];
     S3 : string[20];
Begin
  S1:='abcdefghijklmn';
  S2:=S1; S3:=S1;
```

Как известно, при выполнении оператора присваивания вначале вычисляется значение выражения в его правой части, а затем полученное значение записывается в поле памяти, которое отведено для переменной в левой части. В первом операторе присваивания выражение в его правой части - это строка-константа *'abcdefghijklmn'* длиной 14 символов. Так как эта длина превышает максимальную длину строки *S1*, то в *S1* записываются лишь первые 10 символов, а остальные отбрасываются (они не поместятся в поле памяти, отведенное для строки *S1*).

Если строка в левой части оператора присваивания длиннее, чем значение строкового выражения, то ей устанавливается текущая длина, равная длине строки в правой части.

Содержимое строк в примере 2:

```
S1 : 'abcdefghij';  S2 : 'abcde';  S3 : 'abcdefghij';
```

Поскольку строка неявно представляет собою массив символов, то к отдельным символам строки можно обращаться по индексу.

Пример 3.

```
Var  ch : char;
     S : string;
Begin
  S:='0123456789';
  ch:=S[5]; S[7]:='z'; Writeln('ch=',ch,'  S=',S);
  S[0]:=chr(4); Writeln('ch=',ch,'  S=',S);
```

Будет отпечатано:

```
ch=4  S=012345z789
ch=4  S=0123
```

Нулевой байт *S*[0], определяющий текущую длину строки, рассматривается в составе строки таким же образом, как и остальные байты, т.е. как символ. Поэтому оператор типа *S*[0]:=4 в данном случае недопустим, ибо он требует присвоить символу численное значение.

Для ввода-вывода строк используются те же процедуры, что и для ввода-вывода числовой информации.

Пример 4.

```
Var  S : string[10];
```

```
Begin
  S:='0123456789'; Writeln('  S=',S);
  Writeln('  S=',S:15); Writeln('  S=',S:5);
```

Формат в процедуре печати определяет, сколько позиций на внешнем носителе (в данном случае на экране дисплея) должно быть выделено для размещения выводимого значения. Если формат превышает требуемое количество позиций, то "лишние" позиции слева заполняются пробелами (выводимая строка размещается в правой части поля). Если формат вывода меньше требуемого значения, то на внешнем носителе выделяется столько позиций, сколько содержится символов в выводимом значении.

В примере 4 будет отпечатано:

```
S=0123456789
S=     0123456789
S=0123456789
```

Пример 5. Ввод строковых переменных.

Будем рассматривать ввод из текстового файла *F* (частным случаем текстового файла является клавиатура ПЭВМ). Пусть в строках файла *F*, начиная с их первой позиции, записана следующая информация:

```
abcdefghijklmnopqrst
01234567899876
АБВГДЕЖЗИЙКЛМНОП
```

После открытия файла его указатель устанавливается в первую позицию первой строки файла. Ввод информации всегда производится, начиная с текущей позиции указателя файла. В данном случае имеем:

```
abcdefghijklmnopqrst
↑
01234567899876
АБВГДЕЖЗИЙКЛМНОП
```

Пусть в разделе ***Var*** объявлены следующие переменные:

```
Var  S1 : string[5];
     S2 : string[10];
     S3,S4 : string[15];
```

Рассмотрим выполнение оператора *Read*(*F,S1,S2,S3,S4*), эквивалентного следующим четырем операторам:

```
Read(F,S1); Read(F,S2); Read(F,S3); Read(F,S4).
```

При выполнении первого оператора в строку *S1* будут занесены 5 символов, начиная с текущей позиции файла *F*; при этом строка *S1* получит значение '*abcde*'. Указатель файла переместится в позицию 6 первой строки:

```
abcdefghijklmnopqrst
     ↑
01234567899876
АБВГДЕЖЗИЙКЛМНОП
```

После выполнения второго оператора ввода в переменную *S2* запишется строка '*fghijklmno*', а указатель файла переместится в позицию 16:

```
abcdefghijklmnopqrst
                ↑
01234567899876
АБВГДЕЖЗИЙКЛМНОП
```

При вводе строк *S3* и *S4* будут использованы символы, начиная с текущего положения указателя файла и до конца строки. Тогда строке *S3* будет присвоено значение *'pqrst'*, а строке *S4* - пустое значение.

В этом случае текущая длина строки *S3* станет равной 5, а строки *S4* - 0.

Пусть по отношению к тому же файлу *F* ввод строк выполняют операторы

```
Readln(F,S1); Readln(F,S2); Readln(F,S3); Readln(F,S4).
```

Это эквивалентно следующему:

```
Read(F,S1);  Readln(F);
Read(F,S2);  Readln(F);
Read(F,S3);  Readln(F);
Read(F,S4);  Readln(F).
```

После выполнения первого оператора строка *S1*, как и раньше, получит значение '*abcde*', а указатель файла переместится в позицию 6 первой строки. Выполняемый после этого оператор *Readln*(*F*) перемещает указатель в первую позицию второй строки. После выполнения всех операторов будем иметь:

```
S1 = 'abcde'
S2 = '0123456789'
S3 = 'АБВГДЕЖЗИЙКЛМНО'
S4 = ''.
```

По отношению к строкам выполняется лишь одна операция - операция сцепления (конкатенации), которая обозначается символом "+". Если переменные *S1* и *S2* объявлены как строки, то выражение *S1 + S2* означает, что к концу строки *S1* дописывается строка *S2*.

Пример 6.

```
Var  S1,S2 : string[10];
     S3 : string;
Begin
  S1:='0123456789'; S2:='abcd'; S3:=S1+S2;
  Writeln('  S3=',S3);
```

Будет отпечатано:

```
  S3=0123456789abcd
```

Пример 7.

```
Var  S : string[15];
     i : byte;
Begin
  S:='123456';
  For i:=1 to 6 do
    S:=S+'a';
  Writeln('S=',S);
```

Получим:

```
S=123456aaaaaa  (текущая длина строки равна 12)
```

Пример 8.

```
Var  S : string[15];
     i : byte;
Begin
  S:='123456';
  For i:=1 to 12 do
```

```
    S:=S+'a';
  Writeln('S=',S);
```

Получим:
`S=123456aaaaaaaaa  (текущая длина строки равна 15, но не 18 )`

Пример 9.

```
Var  S : string[15];
     i : byte;
Begin
  S:='123456';
  For i:=7 to 12 do
    S[i]:='a';
  Writeln('S=',S);
```

Получим:
`S=123456  (текущая длина строки равна 6)`

В примере 9, хотя и будут заполнены байты 7 .. 12 символом '*a*', но операции присваивания выполняются по отношению к отдельным символам, а не по отношению к строке в целом. Поэтому текущая длина строки не изменяется. Если после оператора цикла добавить оператор *S*[0] := *chr*(12), то будет напечатано
`S=123456aaaaaa  (текущая длина строки 12).`

Для строк допустимы все операции отношения. Сравнение строк выполняется слева направо по одному байту, пока не будут обнаружены различные байты. Пусть в этих байтах расположены символы *ch1* и *ch2*.

Тогда *ch1* < *ch2*, если *ord*(*ch1*) < *ord*(*ch2*).

Если текущие длины сравниваемых строк неодинаковы, то более короткая строка дополняется справа символом *chr*(0), который меньше любого другого символа таблицы ASCII.

Пример 10.

```
Var  b1,b2,b3 : boolean;
Begin
  b1:='abcdefgh'<'abcdxyzu';
  b2:='300'>'30';
  b3:=''='1234567';
```

Здесь имеем *b1 = true, b2 = true, b3 = false*.

Строчные и прописные буквы в строке считаются разными. В частности, '*a*' ≠ '*A*', так как *ord*('*a*') ≠ *ord*('*A*').

Строки могут быть объявлены не только как переменные (раздел **Var**), но и как константы (раздел **Const**). Например

```
Const   Digits = '0123456789';
```

Здесь нужно обратить внимание на следующее обстоятельство: к элементам строки-константы нельзя обращаться по индексу. Следовательно, мы не можем записать

`ch := Digits[5]` или `Digits[5]:='a';`

Разрешение доступа по индексу означало бы, что строку-константу можно изменять (пример – приведенный выше второй оператор присваивания), *а это недопустимо для констант*.

ПРОЦЕДУРЫ И ФУНКЦИИ ДЛЯ ОБРАБОТКИ СТРОК

Для строк предопределены ряд процедур и функций.

1. Процедура **Delete(Var S:string; Start,n:byte) –** удаление *n* символов строки *S*, начиная с позиции *Start*.

Пример 1.

```
Var  S : string[15];
Begin
  S:='0123456789';
  Delete(S,5,3);
```

Результат: `S='0123789'`.

2. Процедура **Insert(S1:string; Var S2:string; Start:byte) –** вставка строки *S1* в строку *S2*, начиная с позиции *Start*.

Пример 2.

```
Var  S1,S2,S3 : string[10];
Begin
  S1:='xyz'; S2:='01234';
  S3:='0123456789';
  Insert(S1,S2,3); Insert(S1,S3,3);
  Writeln('S1=',S1); Writeln('S2=',S2);
  Writeln('S3=',S3);
```

Будет отпечатано:

```
S1=xyz
S2=01xyz234
S3=01xyz23456
```

3. Процедура **Str(X[:w[:d]]; Var S:string) –** преобразование численного значения арифметического выражения *X* и размещение результата в строке *S*. После выражения *X* можно записывать формат, аналогичный формату вывода в процедурах *Write, Writeln*. Квадратные скобки после имени *X* указывают, что форматы *w* и (или) *d* необязательны.

Пример 3.

```
Var  x,y,z : real;
     k : integer;
     S1,S2,S3,S4 : string[10];
Begin
   x:=15.6789; y:=7; z:=123456789.123456789; k:=-789;
   Str(x:7:2,S1); Str(y:10,S2);
   Str(z:20:9,S3); Str(k:8,S4);
   Writeln('S1=',S1); Writeln('S2=',S2);
   Writeln('S3=',S3); Writeln('S4=',S4);
```

Будет отпечатано:

```
S1=  15.68
S2= 7.000E+00
S3=123456789.
S4=    -789
```

Рассмотрим несколько подробнее работу процедуры *Write* при выводе в текстовый файл (экран дисплея, принтер, диск). При размещении числа на экране или на бумаге каждая

цифра (символ) числа занимает отдельную позицию. Следовательно, число при размещении его в текстовом файле - это строка.

Работа процедуры *Write* разделяется на два этапа:

- преобразование числовой переменной из ее внутримашинного представления (в соответствии с описанием в разделе ***Var***) в строку;
- вывод строки на внешний носитель информации.

Первый этап работы процедуры *Write* эквивалентен работе процедуры *Str*.

4. Процедура **Val(S:string; Var X; Var Code:integer) –**преобразование строки *S* в число *X*. При этом в строке *S* пробелы допускаются лишь до первого значащего символа. Переменная *Code* - это код ошибки. Если преобразование завершилось успешно, то *Code* = 0. В противном случае значение *Code* определяет позицию первого неправильного символа.

Пример 4.

```
Var  S1,S2,S3,S4,S5,S6 : string[10];
     x1,x2,x3 : real;
     k1,k2,k3,Code1,Code2,Code3,
     Code4,Code5,Code6 : integer;
Begin
  S1:='  15.48'; S2:='  15,48'; S3:='  15.ab';
  S4:='   678';  S5:='   6-78'; S6:='   6 78';
  Val(S1,x1,Code1); Val(S2,x2,Code2); Val(S3,x3,Code3);
  Val(S4,k1,Code4); Val(S5,k2,Code5); Val(S6,k3,Code6);
  Writeln('x1=',x1:8:3,'  Code1=',Code1);
  Writeln('x2=',x2:8:3,'  Code2=',Code2);
  Writeln('x3=',x3:8:3,'  Code3=',Code3);
  Writeln('k1=',k1:6,'    Code4=',Code4);
  Writeln('k2=',k2:6,'    Code5=',Code5);
  Writeln('k3=',k3:6,'    Code6=',Code6);
```

Будет отпечатано:

```
x1=  15.480  Code1=0
x2=   0.000  Code2=5
x3=   0.000  Code3=6
k1=   678    Code4=0
k2=     0    Code5=5
k3=     0    Code6=5
```

Рассмотрим несколько подробнее работу процедуры *Read* при вводе из текстового файла (клавиатура, диск). Число в текстовом файле представлено в виде строки, так как каждая цифра (символ) числа занимает отдельную позицию на внешнем носителе информации. Работу процедуры *Read*, как и процедуры *Write*, можно разделить на два этапа:

- преобразование строки в значение числовой переменной в соответствии с ее описанием в разделе ***Var;***
- пересылка значения переменной в соответствующее поле памяти.

Первый этап работы процедуры *Read* совпадает с работой процедуры *Val.*

5. Функция **Length(S:string):byte –** определение текущей длины строки *S*. Действие функции *length*(*S*) эквивалентно вычислению выражения *ord*(*S*[0]) или *byte*(*S*[0]).

6. Функция **Copy(S:string; Start,n:byte):string –** выделение из строки *S* подстроки длиной *n* байт, начиная с позиции *Start*. Если *Start* > *length*(*S*), то возвращается пустая строка.

7. Функция **Concat(S1, S2, ... , Sn:string) : string –** конкатенация (сцепление) строк *S1*, *S2*, ... ,*Sn.* Действие функции *Concat* эквивалентно операции "+".

8. Функция **Pos(S1,S2:string):byte –** определение позиции, с которой отмечается первое появление в строке *S2* подстроки *S1*. Если *S1* не содержится в *S2*, то выходное значение равно нулю.

9. Функция **UpCase(ch:char):char –** преобразование строчной латинской буквы в прописную букву. Символы *ch* вне диапазона '*a*'...'*z*' остаются без изменения. Например, для *UpCase*('*d*') результатом является '*D*'.

Примечание. В таблице ASCII ord('a') = 97, ord('z') = 122, ord('A') = 65. В связи с этим работа функции UpCase по отношению к символу ch сводится к выполнению оператора

```
If (ord(ch)>=97) and (ord(ch)<=122) then
  ch:=chr(ord(ch)-32);
```

Каждую из приведенных выше процедур и функций обработки строк можно реализовать в виде Паскаль-программы. Например, для процедуры *Delete* можно написать:

```
Procedure Delete(Var S:string; Start,n:byte);
Var  i : byte;
Begin
  For i:=Start to length(S)-n do
    S[i]:=S[i+n];
  S[0]:=chr(length(S)-n);
End { Delete };
```

Тем не менее использование предопределенных процедур и функций обработки строк имеет следующие преимущества:

- предопределенные процедуры и функции записаны в библиотеке компилятора в объектном виде и не требуют машинного времени на их трансляцию;
- предопределенные процедуры и функции реализованы наиболее эффективным способом.

ПРИМЕРЫ ОБРАБОТКИ СТРОК

Пример 1. В строке задан текст. Слова текста разделены между собой одним или несколькими пробелами. В начале и в конце строки также могут быть пробелы. Сжать текст, оставив между словами лишь один пробел. Удалить также пробелы в начале и в конце строки.

Удаление из строки пробелов аналогично удалению нулей из одномерного массива (см. раздел "Удаление элементов из массива"). При этом сдвиг символов строки влево и уменьшение ее текущей длины выполняет процедура *Delete.*

```
Program TextDel;
Var  i : byte;
     S : string;
Begin
  Read(S); Writeln('S= ',S);
  For i:=length(S) downto 2 do
    If (S[I]=' ') and (S[I-1]=' ') then
```

```
      Delete(S,i,1);
  If S[1]=' ' then
    Delete(S,1,1);
  If S[length(S)]=' ' then
    Delete(S,length(S),1);
  Writeln('S ',S);
End.
```

Пример 2. В строке задан текст. Слова текста разделены между собой одним или несколькими пробелами. В начале и в конце строки также могут быть пробелы. Определить количество слов в тексте и среднее количество букв в слове.

Предположим, что строка *S* имеет следующий вид (символ "-" использован здесь для обозначения пробела):

```
---abcde----fghijklmn----n-opq--xy—
```

Признаком начала слова является символ, отличный от пробела (непробел), признаком окончания слова - ближайший к нему пробел. Будем присваивать номера позиций, определяющих положение слова в строке, переменным *k1* и *k2*:

```
---abcde----fghijklmn----n-opq--xy--
   ↑    ↑   ↑         ↑
   k1   k2  k1        k2 ...
```

Следовательно, при анализе строки в программе требуется многократно определять местоположения слов или, другими словами, позицию ближайшего пробела или непробела, начиная с заданной позиции *k*. Для выполнения такой работы целесообразно написать отдельные подпрограммы, оформив их в виде функции, поскольку выходом подпрограммы является одно значение - позиция пробела или непробела.

Поиск ближайшего пробела:

```
Function Space(S:string; k:byte):byte;
Var  i : byte;
Begin
  Space:=0;
  For i:=k to length(S) do
    If S[i]=' ' then
      Begin
        Space:=i; Exit
      End;
End { Space };
```

Если в строке *S*, начиная с позиции *k*, пробела не обнаружено, то выходное значение *Space* = 0.

Поиск ближайшего непробела:

```
Function NotSpace(S:string; k:byte):byte;
Var  i : byte;
Begin
  NotSpace:=0;
  For i:=k to length(S) do
    If S[i]<>' ' then
      Begin
```

```
        NotSpace:=i; Exit
      End;
End { NotSpace };
```

Будем считать, что в программе Slovo, реализующей решение поставленной задачи, после раздела **Var** записаны функции *Space* и *NotSpace.*

```
Program Slovo;
Var  S : string;          { анализируемый текст }
     NumWords,            { количество слов }
     NumLetters,          { количество букв }
     k1,k2,               { левая и правая границы слова }
     l : byte;            { длина слова }
     MiddleLet : real;    { среднее количество букв в слове }
     Cond : boolean;      { управляющая переменная }
{ ------------------------------------------------- }
Function Space(S:string; k:byte):byte;
.........................
{ ------------------------------------------------- }
Function NotSpace(S:string; k:byte):byte;
.........................
{ ------------------------------------------------- }
  NumWords:=0; NumLetters:=0; MiddleLet:=0;
  k2:=0; Cond:=true;
  While Cond do
    Begin
      k1:=NotSpace(S,k2+1);
      If k1=0 then
        Cond:=false
      Else
        Begin
          k2:=Space(S,k1+1);
          If k2=0 then
            Begin
              k2:=length(S)+1; Cond:=false;
            End;
          l:=k2-k1;
          Inc(NumWords); Inc(NumLetters,l);
        End;
    End;
  Writeln('Кол-во слов=',NumWords,'  Кол-во букв=',
          NumLetters);
  If NumWords>0 then
    MiddleLet:=NumLetters/NumWords;
  Writeln('Средняя длина слова=',MiddleLet:6:1);
End.
```

В программе производится последовательное выделение слов текста путем определения границ слова *k1* и *k2.* Стартовое значение *k2* принимается равным нулю. Цикл поиска слов продолжается до тех пор, пока истинно значение переменной *Cond.*

В каждом цикле поиска определяются значения *k1* и *k2.* Значение *k1* - это позиция ближайшего непробела, начиная с позиции *k2*+1 (в первом цикле *k2*+1 = 1). Если при обращении к функции *NotSpace* получено *k1*=0, то это означает, что до конца строки больше слов не обнаружено. Тогда переменной *Cond* присваивается значение *false,* что ведет к прекращению работы цикла ***While.***

Если *k1*>0, определяется значение *k2,* т.е. номер позиции ближайшего пробела, начиная с *k1*+1. Если при обращении к функции *Space* получено *k2*=0, то это означает, что до конца

строки нет больше пробелов. Тогда правой границей слова является байт *length*(*S*)+1, т.е. номер байта, расположенного после последней буквы слова. Одновременно значение *k2*=0 указывает, что в тексте найдено последнее слово. Тогда переменной *Cond* присваивается значение *false*, что ведет в дальнейшем к прекращению цикла поиска.

Пример 3. В строке содержатся слова, состоящие из русских букв, и целые десятичные числа. Перед числом может стоять знак "+" или "-". В качестве разделителей слов и чисел в тексте используются пробел, запятая, точка, точка с запятой и двоеточие. Определить:

- количество чисел в тексте;
- сумму десятичных цифр, содержащихся в этих числах.

Поиск конца числа в тексте в основном аналогичен поиску конца слова в предыдущем примере, но искомым признаком является ближайший разделитель, что реализовано в функции SignEnd. Признаком начала числа является цифра или символы "+","-". Поэтому вместо функции *NotSpace* для поиска начала числа будем использовать функцию *Number*.

```
Program Numbers;
Var  S : string;        { анализируемый текст }
     i,                 { параметр цикла }
     NumNumbers,        { количество чисел }
     k1,k2,             { левая и правая границы числа }
     Dig : byte;        { числовое значение цифры }
     Code : integer;    { код преобразования }
     Sum : longint;     { сумма цифр }
     Cond : boolean;    { управляющая переменная }
{ ------------------------------------------------- }
Function Number(S:string; k:byte):byte;
Const  Digits = '+-0123456789';
Var  i : byte;
Begin
  Number:=0;
  For i:=k to length(S) do
    If Pos(S[i],Digits)>0) then
      Begin
        Number:=i; Exit
      End;
End { Number };
{ ------------------------------------------------- }
Function SignEnd(S:string; k:byte):byte;
Const  Separs = ' ,.;:';
Var  i : byte;
Begin
  SignEnd:=0;
  For i:=k to length(S) do
    If Pos(S[i],Separs)>0) then
      Begin
        SignEnd:=i; Exit
      End;
End { SignEnd };
{ ------------------------------------------------- }
Begin
  Readln(S); Writeln('S=',S);
  NumNumbers:=0; Sum:=0;
  k2:=0; Cond:=true;
  While Cond do
    Begin
```

```
      k1:=Number(S,k2+1);
      If k1=0 then
        Cond:=false
      Else
        Begin
          k2:=SignEnd(S,k1+1);
          If k2=0 then
            Begin
              k2:=length(S)+1; Cond:=false;
            End;
          If (S[k1]='+') or (S[k1]='-') then
            Inc(k1);
          For i:=k1 to k2-1 do
            Begin
              Val(S[i],Dig,Code); Inc(Sum,Dig);
            End;
          Inc(NumNumbers);
        End;
    End;
  Writeln('Кол-во чисел=',NumNumbers,'  Сумма цифр=',Sum);
End.
```

Выделяемый из состава числа элемент *S*[*i*] является символом, а не цифрой. Поэтому для преобразования его в численное значение в программе используется процедура *Val*.

Пример 4. В массиве строк содержатся фамилии студентов. Сгруппировать массив по алфавиту.

В таблице ASCII русские буквы записаны по алфавиту. Пробел имеет порядковый номер 32, значительно меньший порядкового номера любой буквы. Поэтому программа группировки текстовых строк по алфавиту отличается от программы группировки одномерного массива по возрастанию лишь тем, что здесь вместо сравнения и обмена местами элементов числового массива сравниваются и обмениваются местами элементы массива строк.

```
Program StudGroup;
Const  Nmax = 100;
Type   string30 = string[30];
       StringAr = array[1..Nmax] of string30;
Var   i,m,n : byte;
      Cond : boolean;
      S : string30;
      Stud : StringAr;
Begin
  Ввод  n, Stud
  Cond:=true; m:=n-1;
  While Cond do
    Begin
      Cond:=false;
      For i:=1 to m do
        If Stud[i]>Stud[i+1] then
          Begin
            S:=Stud[i]; Stud[i]:=Stud[i+1];
            Stud[i+1]:=S; Cond:=true;
          End;
      Dec(m);
    End;
  Печать Stud
End.
```

ПОИСК ОДНОРОДНОЙ ГРУППЫ В МАССИВЕ

Однородная группа (или серия) – это непрерывная последовательность элементов массива, обладающая одним и тем же заданным свойством. Например, серия положительных элементов, серия нечетных элементов и т.п.

Однородные группы можно разделить на два типа:

- группы, для которых можно четко указать признаки начала и конца группы;
- группы, для которых суждение о выполнении заданного свойства может быть сделано лишь в целом по их составу.

Поиск групп (серий) первого типа производился в примерах 2 и 3 предыдущего раздела. По отношению к массиву это может быть, например, серия положительных элементов. Здесь признаком начала группы является индекс первого положительного элемента, а признаком конца группы – индекс ближайшего неположительного элемента.

Примером группы второго типа является серия элементов, симметричных относительно своей середины. Здесь нельзя по значению одного элемента указать начало или конец группы, но можно определить, является ли подмассив, начиная с *i*-го элемента и кончая *j*-ым элементом, симметричным относительно своей середины.

Рассмотрим вначале методику поиска групп первого типа. Алгоритм функционирования программы в этом случае сводится к следующему.

1. Для искомой группы ставятся в соответствие две переменные: переменная *k1*, определяющая позицию первого элемента группы, и переменная *k2*, индицирующая конец группы (позицию элемента, непосредственно следующего за последним элементом группы). В этом случае длина группы (количество входящих в нее элементов) равна *k2* - *k1*. Поиск очередной группы производится, начиная с позиции *k2*+1, где *k2* определяет конечную границу предыдущей группы. Поиск конца текущей группы осуществляется, начиная с позиции *k1*+1, где *k1* определяет начало данной группы.

2. В программу включаются две функции, осуществляющие поиск начала и конца группы по соответствующим признакам. Для серии положительных элементов, как было выше указано, признаком начала группы является выполнение условия $x[i] > 0$, признаком конца группы - $x[i] <= 0$. Если при обращении к функции поиска соответствующий признак не найден, то ее выходное значение принимается равным нулю. Обозначим для общности имена этих функций *SignBegin*(*k:integer*):*integer* и *SignEnd*(*k:integer*):*integer*, где *k* - позиция начала поиска.

3. Общий поиск осуществляется в цикле ***While*** под управлением булевской переменной *Cond*, сохраняющей значение *true* до тех пор, пока продолжается поиск.

4. На старте общего поиска переменной *k2* присваивается нулевое значение, переменной *Cond* - значение *true.*

5. В начальной части цикла ***While*** определяется положение очередной группы: *k1*:=*SignBegin*(*k2*+1).

Если *k1* = 0, то это означает, что в массиве таких групп больше нет. Тогда переменной *Cond* присваивается значение *false*, что ведет к прекращению работы цикла ***While.***

6. Если *k1* > 0, то производится поиск конца группы: *k2* := *SignEnd*(*k1*+1).

Если *k2* = 0, то это означает, что последний элемент массива принадлежит искомой группе. Тогда переменной *k2* присваивается значение *n*+1, где *n* - количество элементов в массиве, а переменной *Cond* - значение *false.*

7. В соответствии с условием задачи производится обработка группы, ограниченной значениями переменных *k1* и *k2.*

Пример 1. В целочисленном массиве определить положение наиболее длинной группы отрицательных элементов, причем в состав группы должно входить не менее трех элементов.

Пусть нам задан массив $X = (x_1, x_2, ..., x_n)$. Признаком начала группы является истинность отношения $x_i < 0$, признаком ее конца - истинность отношения $x_i \geq 0$.

```
Program GroupSearch1;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,k1,k2,n,
     l,                { размер группы }
     lmax,             { размер наиболее длинной группы }
     kmax : integer;   { позиция начального элемента группы }
     Cond : boolean;
     X : Ar;
{ ---------------------------- }
Function SignBegin(k:integer):integer;
Var  i : integer;
Begin
  SignBegin:=0;
  For i:=k to n do
    If x[i]<0 then
      Begin
        SignBegin:=i; Exit
      End;
End { SignBegin };
{ ---------------------------- }
Function SignEnd(k:integer):integer;
Var  i : integer;
Begin
  SignEnd:=0;
  For i:=k to n do
    If x[i]>=0 then
      Begin
        SignEnd:=i; Exit
      End;
End { SignEnd };
{ ---------------------------- }
Begin
  Ввод  n, X
  k2:=0; Cond:=true; lmax:=0;
  While Cond do
    Begin
      k1:=SignBegin(k2+1);
      If k1=0 then
        Cond:=false
      Else
        Begin
          k2:=SignEnd(k1+1);
          If k2=0 then
            Begin
              k2:=n+1; Cond:=false
            End;
          l:=k2-k1;
          If (l>2) and (l>lmax) then
            Begin
              lmax:=l; kmax:=k1
            End;
        End;
    End;
```

```
  Печать  lmax, kmax
End.
```

В этой программе нужно обратить внимание на следующее.

Если в исходном массиве нет отрицательных элементов, то переменной *kmax* не будет присвоено никакого значения. Тогда в общем случае на печать будет выдано некоторое случайное значение этой переменной.

Отмеченную выше ситуацию называют использованием неопределенной переменной, т.е. переменной, которой в процессе работы программы не присвоено никакого конкретного значения. В реальных программах поиск такого рода ошибки может потребовать значительных затрат времени на отладку программы.

Чтобы избежать неопределенности по отношению к переменной *kmax*, в программе GroupSearch1 достаточно до начала цикла ***While*** установить `kmax:=0`.

Некоторой модификацией группы первого типа является такая, в которой для определения ее начала и конца необходимо анализировать не один элемент, а два смежных элемента.

Пример 2. В целочисленном массиве определить наиболее длинную серию элементов, строго упорядоченных по возрастанию.

В искомой серии должно выполняться отношение

$$x[i] < x[i+1] < x[i+2] < \ldots < x[j-1] < x[j] .$$

Следовательно, признаком начала серии является выполнение условия $x[k] < x[k+1]$, признаком конца – выполнение условия $x[k] \geq x[k+1]$.

```
Program GroupSearch2;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,k1,k2,n,
     l,                { размер группы }
     lmax,             { размер наиболее длинной группы }
     kmax : integer;   { позиция начального элемента группы }
     Cond : boolean;
     X : Ar;
{ ---------------------------- }
Function SignBegin(k:integer):integer;
Var  i : integer;
Begin
  SignBegin:=0;
  For i:=k to n-1 do
    If x[i]<x[i+1] then
      Begin
        SignBegin:=i; Exit
      End;
End { SignBegin };
{ ---------------------------- }
Function SignEnd(k:integer):integer;
Var  i : integer;
Begin
  SignEnd:=0;
  For i:=k to n-1 do
    If x[i]>=x[i+1] then
      Begin
        SignEnd:=i; Exit
      End;
```

```
End { SignEnd };
{ ---------------------------- }
Begin
  Ввод  n, X
  lmax:=0; kmax:=0;
  k2:=0; Cond:=true;
  While Cond do
    Begin
      k1:=SignBegin(k2+1);
      If k1=0 then
        Cond:=false
      Else
        Begin
          k2:=SignEnd(k1+1);
          If k2=0 then
            Begin
              k2:=n; Cond:=false
            End;
          l:=k2-k1+1;
          If l>lmax) then
            Begin
              lmax:=l; kmax:=k1
            End;
        End;
    End;
  Печать  lmax, kmax
End.
```

Следует отметить, что в этой программе, в отличие от предыдущей, переменная *k2* определяет индекс последнего элемента серии, а не индекс элемента, расположенного после последнего элемента серии. Поэтому при *k2* = 0 программа устанавливает *k2* := *n*.

Рассмотрим теперь методику поиска групп второго типа. Здесь алгоритм поиска сводится к следующему.

1. Разработать функцию, определяющую выполнение заданного свойства для подмассива, начиная с элемента, имеющего индекс *k1*, и заканчивая элементом с индексом *k2*.

2. В основной программе перебирать все возможные подмассивы и, обращаясь к вышеназванной функции, определять, выполнено ли для данного подмассива заданное свойство.

Пример 3. В целочисленном массиве определить положение наиболее длинной серии, симметричной относительно своей середины.

```
Program GroupSearch3;
Const  Nmax = 500;
Type  Ar = array[1..Nmax] of integer;
Var  i,j,n,
     l,                { размер группы }
     lmax,             { размер наиболее длинной группы }
     kmax : integer;   { позиция начального элемента группы }
     X : Ar;
{ ---------------------------- }
Function YesNo(k1,k2:integer):boolean;
Var  i,j : integer;
Begin
  YesNo:=true;
  i:=k1; j:=k2;
```

```
  While i<j do
    If x[i]<>x[j] then
      Begin
        YesNo:=false; Exit
      End;
End { YesNo };
{ ---------------------------- }
Begin
  Ввод  n, X
  kmax:=0; lmax:=0;
  For i:=1 to n-1 do
    For j:=i+1 to n do
      Begin
        l:=j-i+1;
        If l>lmax then
          If YesNo (i,j) then
            Begin
              lmax:=l; kmax:=i;
            End;
      End;
  Печать  lmax, kmax
End.
```

ТИПИЗИРОВАННЫЕ КОНСТАНТЫ

Память переменным, описанным в каком-либо блоке, выделяется при активизации этого блока, т.е. при старте программы или при обращении к подпрограмме. Однако эти поля памяти ничем не заполнены, их содержимое - случайная комбинация битов. Заполнение полей памяти конкретным содержимым производится в дальнейшем в процессе работы программы с помощью процедуры ввода или оператора присваивания.

В ряде случаев необходимо, чтобы переменные до начала их обработки имели определенные начальные значения, например, нулевые. Это можно было бы сделать с помощью операторов присваивания в начальной части программы.

Турбо Паскаль дает возможность, описывая переменные, сразу же указывать их начальные (стартовые) значения. Единственным требованием при этом является то, что описание переменных должно быть перенесено из раздела ***Var*** в раздел ***Const***. Переменные с заданным начальным значением называются в Турбо Паскале типизированными константами. При старте программы эти значения уже будут содержаться в полях памяти, отведенных для типизированных констант. В дальнейшем значения таких констант могут быть изменены в программе таким же образом, как и значения обычных переменных. В связи с этим типизированные константы называют также переменными с начальным значением.

Простые значения типизированным константам задаются приравниванием после описания типа.

Пример 1.

```
Const  Max : integer = 10000;
       T : real = 0;
       S : string = 'abcd';
       ch : char = #27;
```

Для одномерных массивов стартовые значения задаются путем перечисления их элементов в круглых скобках, причем эти значения должны быть заданы для всех элементов массива.

Пример 2.

```
Type  Ar = array[1..10] of integer;
Const  T : Ar = (0,0,0,1,1,1,-1,-1,-1,0);
```

Стартовые значения для матриц задаются аналогично, но в этом случае каждая строка матрицы окаймляется отдельной парой круглых скобок.

Пример 3.

```
Type  Matrix = array[1..4,1..6] of integer;
Const  A : Matrix = ((0,0,0,1,1,1), (-1,-1,-1,0,0,0),
                     (1,2,3,4,5,6), (-3,-4,-1,1,1,5));
```

Имеются также правила задания стартовых значений для записей, множеств и переменных других типов. Здесь эти правила не рассматриваются.

При старте программы память выделяется для всех типизированных констант вне зависимости от их расположения - в основной программе или в подпрограмме. Эта память выделяется в сегменте данных, т.е. в той же области, которая используется для размещения глобальных переменных. Поэтому адреса полей памяти, соответствующих типизированным константам (переменным с начальным значением) не изменяются в течение всего периода работы программы. Следовательно, если типизированная константа описана в блоке подпрограммы, то при повторном входе в подпрограмму ей, в отличие от локальных переменных, память повторно не выделяется, при этом сохраняется предыдущее значение типизированной константы.

Рекомендуется обратить внимание на следующее полезное применение типизированных констант.

При разработке программы формирования меню тексты позиций меню целесообразно оформить в виде массива строк. Поскольку эти тексты в программе не изменяются, то они представляют собой массив констант. В Паскале массивы констант не допускаются, но могут быть организованы массивы типизированных констант.

Пример 4.

```
Type  StringAr = array[1..9] of string[7];
      MonthAr = array[1..12] of byte;
Const  TextMenu : StringAr = ('File', 'Edit', 'Search',
         'Run', 'Compile', 'Debug', 'Options', 'Window',
         'Help');
       MonthDays : MontyAr = (31, 28, 31, 30, 31, 30
                   31, 31, 30, 31 ,30, 31);
```

Список дней в месяцах в виде типизированной константы MonthDays более компактен по сравнению с присваиванием значений отдельно каждому месяцу в разделе операторов. Если же при обработке календарной задачи будет определен високосный год, то количество дней в феврале легко изменить оператором присваивания: MonthDays[2]:=29.

Пример 5. Определить количество вызовов процедуры, к которой многократно обращаются из различных мест вызывающих программ.

```
..............
Procedure Proc(m:integer; r:real; Var k:word);
Const  Counter : word = 0;
Begin
..............
  Inc(Counter);
  k:=Counter;
..............
End { Proc };
```

МНОЖЕСТВА

Множество в математике - это произвольный набор элементов, понимаемых как единое целое. На вид элементов и их количество никаких ограничений не накладывается. Каждый элемент множества уникальный, т.е. не может повторяться (он или есть в множестве, или его нет). Свойство уникальности элемента - одно из основных отличий множества от массива (в массиве элементы могут повторяться).

Множества в математике могут быть абстрактные и конкретные. Для абстрактного множества имеет значение лишь наличие или отсутствие элемента; тип входящих в множество элементов при этом игнорируется. В конкретных множествах элементы имеют вполне определенный тип, одинаковый для всех. Например, в множество натуральных чисел могут входить лишь целые положительные числа. В Паскале множество может быть только конкретным, поскольку любая обрабатываемая в программе переменная должна быть описана в разделе ***Var.***

Множества в математике могут быть конечные и бесконечные. Например, множество натуральных чисел является бесконечным, множество букв какого-либо алфавита - конечным. В Паскале множество может быть только конечным, поскольку память ЭВМ, в которой размещаются все переменные, сама является конечной.

Множество может быть непрерывным или дискретным. Например, множество вещественных чисел является непрерывным, поскольку между двумя любыми числами можно расположить бесконечное количество других вещественных чисел. Множество натуральных чисел является дискретным. Так, между элементами 8 и 9 других натуральных чисел не существует. В Паскале множество может быть только дискретным. Этому требованию соответствует ординальный тип элементов множества.

Для множеств в математике определены операции пересечения, объединения и разности множеств.

Пусть мы имеем произвольные множества A и B с одинаковым типом элементов. Тогда пересечение множеств $C = A \bigcap B$ означает, что в множество C входят лишь элементы, которые одновременно имеются и в множестве A, и в множестве B. При объединении множеств $C = A \bigcup B$ в C включаются как элементы, находящиеся в A, так и элементы, находящиеся в B. Разность множеств $C = A \setminus B$ определяет, что в множество C записываются все элементы множества A за исключением тех элементов, которые заданы в множестве B.

Для множеств определены все операции отношения.

Если множество A является частью множества B, то это обозначается $A \subset B$. Запись $A \supset B$ означает, что B является подмножеством множества A. Эквивалентность множеств $A = B$ означает, что в оба эти множества входят одни и те же элементы. Фактически записанные здесь соотношения между множествами - это операции отношения. Для конкретных множеств эти отношения могут выполняться или не выполняться и, следовательно, результатом операции являются значения *true* или *false*.

Для отдельного элемента отношением является операция, определяющая его наличие или отсутствие в конкретном множестве: $a \in A$ или $a \notin A$. Результатом этой операции также являются значения *true* или *false*.

Количество элементов в множестве называют его мощностью. Запись $|A| = n$ означает, что в множестве A содержится n элементов. Множество может быть пустым, т.е. не содержать ни одного элемента.

Предположим, что элементами множества P могут быть только объекты, обозначаемые символами a, b, c. Тогда в математической записи множество P может иметь следующие конкретные значения:

$P = [\,]$; $P = [a]$; $P = [b]$; $P = [c]$;
$P = [a\,b]$; $P = [a\,c]$; $P = [b\,c]$; $P = [a\,b\,c]$

Каждый элемент того типа, который определен для множества, может быть или не быть в данном множестве. Поэтому логично поставить в соответствие каждому элементу множества один бит памяти. Тогда значение бита 0 будет означать, что данный элемент отсутствует в множестве, а значение 1 будет свидетельствовать о нахождении его в этом множестве.

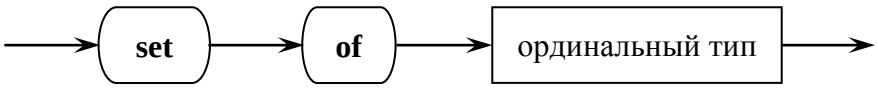
Множество в целом реализуется в Паскале как битовая строка. Для приведенного выше примера достаточно иметь строку длиной три бита. Пусть первый бит - это элемент a, второй - элемент b, третий - элемент c.

Тогда множество P может иметь следующие значения:

0 0 0	1 1 0
1 0 0	1 0 1
0 1 0	0 1 1
0 0 1	1 1 1

Битовая строка не может быть неограниченной, т.е. мощность множества не может быть бесконечной. В Турбо Паскале максимальная длина битовой строки определяется численным значением переменной типа *byte* и равна 256 бит (32 байта).

Объявление множественного типа имеет вид:



Например,

```
set of byte
set of char
set of 1..100
```

Сопоставим описания массива и множества.

В описании массива

```
array[T1] of T2
```

T1 - это тип индекса, *T2* - тип элемента массива. Тип *T1*, в частности, указывает, какие значения имеет право принимать индекс элемента массива и, следовательно, определяет количество элементов массива, т.е. его размер. Например, описание

```
array[byte] of T2
```

означает, что в массиве содержится 256 элементов, а его индекс может иметь целочисленные значения 0, 1, 2, ..., 255. При описании

```
array[char] of T2
```

массив имеет такой же размер, но индекс элемента массива принимает значения символов таблицы ASCII: #0, #1, #2, ..., #255.

Возможные значения элементов множества заранее определены: 0 или 1, поэтому тип *T2* по отношению к множеству не имеет смысла. В описании множества

```
set of T1
```

тип *T1* в основном аналогичен такому же типу для индекса массива: он определяет, какие конкретно элементы могут быть в множестве и количество этих элементов, т.е. мощность множества. Например, описание

```
set of byte
```

означает, что в множество могут входить 256 элементов, имеющих значения 0,1,2,...,255. При описании

```
set of char
```

потенциальными элементами множества являются символы #0, #1, ..., #255. В обоих случаях для множества выделяется поле памяти длиной 32 байта.

Так как тип *T1* определяет, в частности, мощность множества, то объявление типа

```
SetType = set of integer
```

является неправильным, так как битовая строка не может иметь длину 65536 бит (тип *integer* принимает значения -32768, -32767, ..., -1, 0, 1, ... , 32767).

Обычно для описания множества используют диапазонный тип.

Пример.

```
Type  Diap1 = 1..100;
      Diap2 = 'a'..'z';
      SetNumberType = set of Diap1;
      SetLetterType = set of Diap2;
Var  SetNumber : SetNumberType;
     SetLetter : SetLetterType;
```

Здесь элементами множества *SetNumber* являются числа 1, 2, ..., 100, элементами множества *SetLetter* - малые буквы латинского алфавита *a, b, ..., z.*

Память для размещения множества выделяется при старте блока, где описано это множество, но значение множества при этом остается неопределенным. Другими словами, объявление множества *SetNumber* типа *SetNumberType* указывает лишь на принципиально возможный набор значений элементов множества (в данном случае целые числа от 1 до 100), но в поле переменной *SetNumber* никакие значения не записываются.

Простой переменной присвоить конкретное значение можно в разделе операторов путем задания соответствующей константы. Для множества роль константы играет так называемый конструктор множества.

Конструктор множества - это список элементов, заключенный в квадратные скобки. В списке могут быть отдельные элементы и диапазоны.

Например,

```
SetNumber := [];
SetLetter := ['a','d','k'..'p','y'];
```

Остановимся на смысловом значении приведенных выше операторов присваивания для множеств *SetNumber* и *SetLetter.*

Из объявления множества *SetNumber* следует, что для него при старте программы выделяется 100 бит памяти (13 байт, в последнем байте используются 4 бита). При выделении памяти поле *SetNumber* заполнено случайным образом. Оператор *SetNumber*:=[ ] указывает, что всем 100 битам этого поля должно быть присвоено нулевое значение.

Для множества *SetLetter* выделяется 26 бит (количество букв алфавита от *'a'* до *'z'*). В этом поле биты 1,2,3,4 (порядковые номера букв *'a','b','c','d'* в диапазонном типе *Diap2*), биты 11,12,13,14,15,16 и 25 (порядковые номера букв *'k'..'p'* и *'y'*) устанавливаются в единичное положение, все остальные биты - в нулевое.

Если *A* и *B* - множества одного типа, то к ним применимы следующие операции:

1) A * B - пересечение множеств;
2) A + B - объединение множеств;
3) A - B - разность множеств.

В Турбо Паскале определены четыре логические операции: отрицание (***not***), логическое умножение (***and***), логическое сложение (***or***), исключающее ИЛИ (***xor***). Эти операции применимы не только к булевским переменным, но и к операндам целого типа. В последнем случае операция выполняется отдельно для каждого бита, т.е. ее можно рассматривать как операцию над битовыми строками. Поскольку на машинном уровне множество - это битовая строка, то выполнение операций пересечения, объединения и разности множеств сводится к соответствующим логическим операциям.

Операция объединения множеств - это логическое сложение двух битовых строк; пересечение множеств - это логическое умножение битовых строк. Если обозначить биты, входящие в множества *A* и *B*, через *a* и *b*, то операция разности множеств сводится к вычислению для каждой пары битов логического выражения *a* ***and*** (***not*** *b*).

Пересечение и объединение двух множеств часто называют соответственно умножением и сложением множеств. Отсюда и приоритеты этих операций: пересечение старше операций объединения и вычитания, а они в свою очередь старше рассматриваемых ниже операций отношения.

Пример выполнения операций над множествами:

```
Type  Diap = 0..15;
      SetType = set of Diap;
Var  A,B,C,D,E : SetType;
Begin
  A := [0..3, 5, 7..9, 12, 14];
  B := [4..7, 9, 11..13, 15];
  C:=A+B; D:=A*B; E:=A\B;
```

По отношению к битовым строкам имеем:

```
   A = 1111 0101 1100 1010
   B = 0000 1111 0101 1101
```

Тогда

```
   C = 1111 1111 1101 1111,  т.е. C=[0..9,11..15]
   D = 0000 0101 0100 1000,  т.е. D=[5,7,9,12]
   E = 1111 0000 1000 0010,  т.е. E=[0..3,8,14]
```

Для множеств допустимы следующие операции отношения:

1) a **in** A - вхождение элемента *a* в множество *A* ($a \in A$);

2) A = B - равенство множеств;

3) A <> B - неравенство множеств;

4) A < B - множество *A* является подмножеством в *B* ($A \subset B$);

5) A > B - множество *B* является подмножеством в *A* ($A \supset B$);

6) A <= B - множество *A* равно множеству *B* или является его подмножеством ($A \subseteq B$);

7) A >= B - множество *B* равно множеству *A* или является его подмножеством ($A \supseteq B$).

Операция проверки вхождения элемента *a* в множество *A* (*a* ***in*** *A*) эквивалентна вычислению выражения [*a*] * *A*. При проверке равенства и неравенства множеств для каждой пары битов выполняется операция ***xor***. Вычисление отношения *A* <= *B* эквивалентно вычислению логического выражения *A* + *B* = *A*, а вычисление отношения *A* >= *B* заменяется вычислением выражения *A* + *B* = *B*.

Логические операции выполняются значительно быстрее, чем арифметические (например, при логическом сложении нет переноса единицы в старший разряд в отличие от ариф-

метического сложения). Поэтому программы, использующие множества для реализации какого-либо алгоритма, работают быстрее, чем программы, в которых этот же алгоритм реализован другим способом.

Например, вместо оператора

```
If (ch='Д') or (ch='д') or (ch='L') or (ch='l') then S,
```

где *ch* - переменная типа *char*; *S* - произвольный оператор, более целесообразно использовать оператор

```
If ch in ['Д','д','L','l'] then S.
```

Непосредственный ввод-вывод множества не допускается, однако это можно сделать косвенным путем:

```
Type  SetType = set of byte;
Var   k : byte;
      Set1 : SetType;
      F : text;
Begin
.................
  Set1:=[];                { "Обнуление" множества Set1 }
  While not SeekEof(F) do { Ввод множества Set1 }
    Begin
      Read(F,k); Set1:=Set1+[k];
    End;
  For k:=0 to 255 do       { Вывод множества Set1 }
    If k in Set1 then
      Writeln(k);
```

ПРИМЕРЫ ОБРАБОТКИ МНОЖЕСТВ

Логические операции, на которых основана обработка множеств, выполняются значительно быстрее, чем арифметические операции. Это повышает эффективность работы программы, использующей множества для решения поставленной задачи, программная реализация которой в большинстве случаев возможна и без применения множеств.

Пример 1. В строке записана последовательность цифр в заданной системе счисления с основанием $2 \le q \le 10$. В начале и в конце строки могут быть пробелы. Преобразовать цифровую последовательность в число.

Пусть задана последовательность цифр $a_n a_{n-1} a_{n-2} \ldots a_1 a_0$. Ее можно представить в виде полинома

$$P = a_n q^n + a_{n-1} q^{n-1} + a_{n-2} q^{n-2} + \ldots + a_1 q + a_0$$

или по схеме Горнера

$$P = (\ldots(a_n q + a_{n-1}) q + a_{n-2}) q + \ldots + a_1) q + a_0 \ .$$

Выполним два варианта решения задачи: без использования множеств и с их использованием.

```
Program ConVert1;
Const  Digits = '0123456789';
Var   i,k,q : byte;
      Number : longint;
      S : string;
```

```
Begin
  Ввод  и  печать  S, q
  Number:=0;
  For i:=1 to length(S) do
    Begin
      k:=Pos(S[i],Digits);
      If k>0 then
        Number:=Number*q+k-1;
    End;
  Печать Number
End.
```

Если выделенный из строки *S* символ - это пробел, то переменная *Number* не изменяется; в противном случае в ее состав поступает численное значение цифры, которое на 1 меньше позиции этой цифры в строке *Digits* и равно *k*-1.

Примечание. В программе предполагается, что значение числа не превышает *MaxLongInt* = 2^{31}-1. Если для переменной *Number* указать тип ***real***, то значение числа не должно превышать 10^{38}. В то же время, если во всех байтах строки *S* записать цифры 9, то значение числа составляет 10^{256}-1. В этом случае нужно использовать тип ***double*** (максимальное значение 10^{308}) или ***extended*** (максимальное значение 10^{4932}).

```
Program ConVert2;
Var  i,q : byte;
     Number : longint;
     S : string;
     Digits : set of '0'..'9';
Begin
  Ввод  и  печать  S,  q
  Digits:=['0'..'9'];
  Number:=0;
  For i:=1 to length(S) do
    If S[i] in Digits then
      Number:=Number*q+ord(S[i])-ord('0');
  Печать Number
End.
```

В программе Convert1 функция *Pos* определяет, является ли символ *S*[*i*] цифрой или нет, и при положительном ответе ($k > 0$) указывает порядковый номер *k* этого символа в строке-константе *Digits*, который в данном случае равен численному значению цифры, увеличенному на 1.

Функция *Pos* реализована в компиляторе в виде подпрограммы, выполняющей последовательный просмотр байтов строки.

В программе Convert2 проверка принадлежности символа *S*[*i*] к цифрам осуществляется операцией ***in,*** что на машинном уровне сводится к логическому умножению двух битовых строк. Численное значение цифры формируется как разность *ord*(*S*[*i*]) - ord('*0*') (здесь используется тот факт, что в любой таблице символов, в том числе в таблице ASCII, цифры образуют непрерывную возрастающую последовательность элементов таблицы).

Пример 2. В заданной строке могут быть повторяющиеся символы. Сократить строку, сохранив для каждого повторяющегося символа лишь его первое вхождение в эту строку.

Решение задачи также выполним в двух вариантах.

```
Program DelSymbols1;
Var  i,k : byte;
     S : string;
```

```
Begin
  Ввод  и  печать  S
  i:=2;
  While i<=length(S) do
    Begin
      k:=Pos(S[i],Copy(S,1,i-1));
      If k>0 then
        Delete(S,i,1)
      Else
        Inc(i);
    End;
  Печать S
End.
```

```
Program DelSymbols2;
Var  i : byte;
     S : string;
     CharSet : set of char;
Begin
  Ввод  и  печать  S
  CharSet:=[S[1]];
  i:=2;
  While i<=length(S) do
    If S[i] in CharSet then
      Delete(S,i,1)
    Else
      Begin
        CharSet:=CharSet+[S[i]];
        Inc(i);
      End;
  Печать S
End.
```

В программе DelSymbols1 для *i*-го символа с помощью функции *Pos* проверяется, имеется ли такой же символ в предыдущих байтах строки, и в случае подтверждения производится его удаление из строки.

В программе DelSymbols2 формируется множество *CharSet*, в которое заносится *i*-ый символ, если он отсутствует в этом множестве. Если такой символ уже имеется в *CharSet*, то он удаляется из строки *S*. Поскольку операция ***in*** на машинном уровне сводится к логическому умножению, то такая проверка повторяемости символа осуществляется значительно быстрее, чем последовательный перебор байтов строки с помощью функции *Pos*.

ФОРМИРОВАНИЕ СПИСКА ПРОСТЫХ ЧИСЕЛ

Задача формирования списка простых чисел относится к тем задачам, на примере которых наглядно демонстрируется эффективность использования множеств.

Условие задачи. Требуется найти все простые числа в диапазоне 2 .. n, где $n > 2$.

По определению, к простым относятся такие числа, которые делятся без остатка только на 1 и сами на себя.

Выполним поиск простых чисел в диапазоне 2 .. 10000, используя приведенное выше определение.

```
Program Prime1;
Const  n = 10000;
Type  PrimeAr = array[1 .. n div 2] of word;
```

```
Var  i,j,
     NextPrime,           { очередное простое число }
     m,                   { кол-во простых чисел }
     p : word;
     Primes : PrimeAr;    { массив простых чисел }
     Cond : boolean;
Begin
  Primes[1]:=2; m:=1;
  For i:=3 to n do
    Begin
      NextPrime:=i; p:=NextPrime div 2;
      Cond:=true;
      For j:=1 to m do
        If NextPrime mod Primes[j]=0 then
          Begin
            Cond:=false; Break   { выход за пределы цикла по j }
          End
        Else
          If Primes[j]>p then
            Break;               { выход за пределы цикла по j }
      If Cond then
        Begin
          Inc(m); Primes[m]:=NextPrime
        End;
    End;
  Печать массива Primes
End.
```

Количество простых чисел в диапазоне 2 .. *n* по крайней мере не превосходит значения *n*/2. Поэтому размер массива *Primes* принят равным *n* ***div*** 2.

В массиве *Primes* накапливается список простых чисел. В цикле по *i* производится перебор элементов числовой последовательности 3, 4, ..., *n*. Очередное число *NextPrime,* взятое из этой последовательности, делится в цикле по *j* на числа, накопленные к этому моменту в массиве *Primes*. Если остаток от деления равен нулю, то это означает, что значение *NextPrime* не является простым числом. Тогда дальнейший перебор массива *Primes* прекращается. Выход из цикла по *j* производится также в том случае, когда очередное число из массива *Primes* превышает значение *p*, равное *NextPrime* ***div*** 2. Если после окончания работы цикла по *j* переменная *Cond* сохранит значение *true*, то производится добавление в список простых чисел значения переменной *NextPrime.*

Основным недостатком программы Prime1 является то, что при проверке каждого значения переменной *NextPrime* многократно выполняется операция деления, требующая значительных затрат машинного времени.

Для формирования списка простых чисел наиболее эффективным является алгоритм, называемый "решето Эратосфена". Преимуществом этого алгоритма является то, что в нем для нахождения простых чисел не нужно использовать операции умножения и деления.

Последовательность работы алгоритма:

Шаг 1. Поместить все числа от 2 до *n* в "решето".

Шаг 2. Выбрать и удалить из "решета" наименьшее число.

Шаг 3. Поместить это число среди простых чисел.

Шаг 4. Удалить из "решета" все числа, кратные данному.

Шаг 5. Если "решето" не пустое, то перейти к шагу 2.

Проиллюстрируем работу алгоритма на конкретном примере. Для этого в качестве решета будем использовать массив *Sieve*, а для списка простых чисел, как и в программе Prime1, - массив *Primes*.

Начальное состояние массивов:
Sieve = (2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,...)
Primes = ( ).

Первое прохождение алгоритма (шаги 2, 3, 4):
Sieve = (3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,...)
Primes = (2).

Второе прохождение:
Sieve = (5,7,11,13,17,19,23,25,29,31,35, ...)
Primes = (2, 3).

Третье прохождение:
Sieve = (7,11,13,17,19,23,29,31, ...)
Primes = (2, 3, 5).
.....

Вначале реализуем алгоритм Эратосфена без использования множеств. При этом удаляемым из массива *Sieve* числам будем присваивать нулевое значение (это более эффективно, чем сдвиг подмассива влево для удаления очередного числа).

Примечание. Такая форма условного удаления числа из массива допустима лишь в том случае, когда в исходном массиве не может быть нулевых чисел.

```
Program Prime2;
Const  n = 10000;
Type  SieveAr = array[2..n] of word;
      PrimeAr = array[1..(n div 2)] of word;
Var  i,j,
     NextPrime,            { очередное простое число }
     Multiple,             { множитель, кратный NextPrime }
     m : word;             { кол-во простых чисел }
     Primes : PrimeAr;     { массив простых чисел }
     Sieve : SieveAr;      { "решето" натуральных чисел }
     Cond : boolean;
Begin
  For i:=2 to n do         { размещение чисел в "решете" }
    Sieve[i]:=i;
  Cond:=true; m:=0;
  While Cond do
    Begin
      NextPrime:=0;
      For i:=2 to n do            { нахождение минималь- }
        If Sieve[i]<>0 then       { ного числа в массиве }
          Begin                   { Sieve                }
            NextPrime:=i; Break
          End;
      If NextPrime=0 then         { если NextPrime=0, то }
        Cond:=false               { "решето" - пустое    }
      Else
        Begin
```

```
            Inc(m);                  { добавление очередного }
            Primes[m]:=NextPrime;    { числа в массив Primes }
            Multiple:=NextPrime-1;
            While Multiple<=n do     { удаление из решета чи-}
              Begin                  { сел, кратных значению }
                Sieve[Multiple]:=0; { NextPrime }
                Inc(Multiple,NextPrime);
              End;
          End;
      End;
    Печать массива Primes
End.
```

Программа Prime2 работает значительно быстрее, чем программа Prime1, поскольку в ней не используются операции умножения и деления. Однако здесь для реализации алгоритма применены два больших массива *Sieve* и *Primes*, каждый элемент которых занимает два байта памяти.

Реализация алгоритма Эратосфена с использованием множеств:

```
Program Prime3;
Const  n = 10000;
Type  SetType = set of 2 .. n;
Var  Sieve,              { множество-решето }
     Primes : SetType;   { множество простых чисел }
     i,                  { параметр цикла }
     NextPrime,          { очередное простое число }
     Multiple : word;    { удаляемое число, кратное NextPrime }
Begin
   Sieve:=[2..n]; Primes:=[];
   NextPrime:=2;
   Repeat
     While not (NextPrime in Sieve) do
       Inc(NextPrime);
     Primes:=Primes+[NextPrime];
     Multiple:=NextPrime;
     While Multiple<=n do
       Begin
         Sieve:=Sieve-[Multiple];
         Inc(Multiple,NextPrime);
       End;
  Until  Sieve=[];
  For i:=2 to n do
    If i in Primes then
      Writeln(i);
End.
```

Оба множества *Sieve* и *Primes* - это битовые строки длиной $n-1$ бит. На стартовом участке программы все биты множества *Sieve* устанавливаются в положение 1, все биты множества *Primes* - в положение 0. В дальнейшем при нахождении очередного простого числа один бит в *Primes* устанавливается в положение 1, а часть бит в *Sieve* (биты, соответствующие кратным значениям простого числа) - в положение 0.

Каждый элемент множеств *Sieve* и *Primes* занимает один бит памяти, т.е. в 16 раз меньше, чем элементы массивов *Sieve* и *Primes* в программе Prime2. Программа Prime3 одновременно требует меньших затрат машинного времени по сравнению с программой Prime2,

поскольку обработка битовых строк осуществляется быстрее, чем обработка числовых массивов.

Следует отметить, что при заданном значении $n = 10000$ программа Prime3 работать не будет, так как при этом мощность множества превышает максимально допустимое значение $n = 256$. Чтобы обеспечить генерацию простых чисел при $n > 256$, необходимо представить "решето" и последовательность простых чисел в виде массива множеств, что приведет к небольшому усложнению программы.

ОПРЕДЕЛЕНИЕ БИТОВОЙ СТРУКТУРЫ ПОЛЯ ПАМЯТИ

Как было ранее указано, внутреннее представление множества - это битовая строка. Если трактовать какое-либо поле памяти как битовую строку, т.е. как множество, то это позволяет определить битовую структуру этого поля, или другими словами значения всех битов для конкретного содержания заданного поля памяти. Совмещение множества с каким-либо полем памяти можно сделать путем использования аппарата абсолютных переменных.

В примере, приведенном ниже, выводится на печать битовая структура полей памяти типа *integer*, типа *real* и строки длиной три символа.

```
Program Field;
Type  SetType = set of byte;
Var  I : integer;
     R : real;
     S : string[3];
     Set1 : SetType absolute I;
     Set2 : SetType absolute R;
     Set3 : SetType absolute S;
{ ------------------------------------------------- }
Procedure PrintValue(Var SetVal:SetType; n:byte);
Var  k,m : byte;
Begin
  m:=0;
  For k:=0 to n do
    Begin
      If k in SetVal then
        Write('1')
      Else
        Write('0');
      Inc(m);
      If m=4 then
        Begin
          m:=0; Write(' ');
        End;
    End;
  Writeln;
End { PrintValue };
{ ------------------------------------------------- }
Procedure PrintSet;
Var  k1,k2,k3,k4 : byte;
Begin
  Writeln('Переменная I= ',I); PrintValue(Set1,15);
  Writeln('Переменная R= ',R:7:1); PrintValue(Set2,47);
  Writeln('Строка S= ',S); PrintValue(Set3,31);
  k1:=ord(S[0]); k2:=ord(S[1]);
  k3:=ord(S[2]); k4:=ord(S[3]);
  Writeln('k1= ',k1,'  k2= ',k2,'  k3= ',k3,'  k4= ',k4);
```

```
End { PrintSet };
{ ------------------------------------------------- }
Begin
  i:=5000; R:=5000; S:='abc';
  PrintSet;
  i:=-5000; R:=-5000; S:='012';
  PrintSet;
End.
```

Будут отпечатаны следующие результаты:

```
  Переменная I= 5000
0001 0001 1100 1000
  Переменная R=  5000.0
1011 0001 0000 0000 0000 0000 0000 0000 0000 0010 0011 1000
  Строка S= abc
1100 0000 1000 0110 0100 0110 1100 0110
k1= 3   k2= 97   k3= 98   k4= 99
  Переменная I= -5000
0001 1110 0011 0111
  Переменная R=  -5000.0
1011 0001 0000 0000 0000 0000 0000 0000 0000 0010 0011 1001
  Строка S= 012
1100 0000 0000 1100 1000 1100 0100 1100
k1= 3   k2= 48   k3= 49   k4= 50
```

Рассмотрим значения *I* = 5000 и *S* = 'abc'.

5000 = $1388 . Если читать биты в байтах справа налево, то получим 8831. Следовательно, для числовых значений не только байты в памяти расположены справа налево, но и биты в этих байтах пронумерованы справа налево.

Для строки имеем: 3 (текущая длина строки) = $03, 97 = $61, 98 = $62, 99 = $63. Из анализа полученных результатов можно сделать вывод, что байты строки располагаются в памяти слева направо, а биты в этих байтах - справа налево.

ПЕРЕЧИСЛЯЕМЫЙ ТИП ДАННЫХ

Прежде чем описывать перечисляемый тип данных, рассмотрим способы задания допустимых значений для ординальных типов *byte, char* и *boolean.*

В типе *byte* допустимыми значениями являются числа 0, 1, ..., 255; в типе *char* - символы таблицы ASCII с номерами #0, #1, ..., #255. Внутримашинным представлением типа *char* являются однобайтные числа от 0 до 255, т.е. на машинном уровне значения типов *byte* и *char* ничем друг от друга не отличаются.

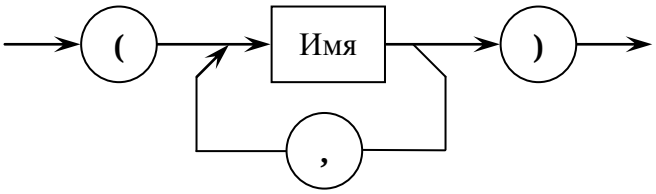
Допустимыми значениями типа *boolean* являются предописанные слова *false* и *true,* причем считается, что *false* < *true.* Внутримашинным представлением этих слов являются числа 0 и 1 длиной один байт.

Следовательно, значениями типа *byte* на уровне Паскаль-программы являются числовые константы, типа *char* - символьные константы, а типа *boolean* - имена логических констант. Другими словами, типу *boolean* ставятся в соответствие имена констант, а типам *byte* и *char* - значения констант. На машинном уровне всем им соответствуют однобайтные числа.

Способ, аналогичный по отношению к типу *boolean,* когда допустимый набор значений определяется путем перечисления имен констант, используется в перечисляемом типе данных для объявления новых ординальных типов.

Перечисляемый тип задает упорядоченное множество значений путем перечисления имен констант, которые обозначают эти значения.

Синтаксическая диаграмма:



Пример 1.

```
Type  Color = (White,Red,Blue,Yellow,Green,Gray,
                Orange,Brown,Black);
      Operator = (Plus,Minus,Times,Divide);
      Metall = (Fe,Cu,Al,Sn,Pb,Ni,Mo,Zn,Ag,Au,Pt,Na);
```

На машинном уровне константы *White, Red, ..., Black* имеют значения 0, 1, ... , 8, константы *Plus .. Divide* - значения 0 .. 3, константы *Fe .. Na* - значения 0 .. 11 .

Если мы объявим

```
Var  R,C : Color;
```

то переменные *R* и *C* могут принимать в программе лишь одно из перечисленных в разделе ***Type*** значений. Например:

```
  C:=Blue; R:=Black.
```

Это в основном аналогично следующему:

```
Const  White = 0;  Red = 1;
       Blue = 2;   Yellow = 3;
       Green = 4;  Gray = 5;
       Orange = 6; Brown = 7;
       Black = 8;
Var  R,C : byte;
Begin  C:=Blue; R:=Black .
```

Другими словами, значения перечисляемого типа - это синонимы констант.

В последнем примере можно было бы написать

```
C:=2;  R:=8.
```

В предыдущем примере это считалось бы ошибкой.

Неверные определения:

```
Type  Week = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
      FreeDay = (Sat,Sun);
```

Здесь одна и та же константа с именем *Sat* должна иметь в программе различные значения (на машинном уровне 5 и 0); такое же замечание относится к константе *Sun.*

Это аналогично тому, что было бы в разделе констант при таком объявлении:

```
Const  Sat = 5;
       Sat = 0;
```

Тип *boolean* также можно объявить перечисляемым:

```
Type  boolean = (false,true) .
```

Следовательно, *false* и *true* - это имена констант, но они предопределены в Паскаль-программе.

Ко всем перечисляемым типам применимы операции отношения, а также предописанные функции *pred, succ, ord.*

Пример 2.

```
false < true
Plus >= Minus
pred(Blue) = Red
ord(Blue) = 2
```

Могут быть операторы вида

```
For c:=White to Black do S .
```

Это эквивалентно операторам

```
For i:=ord(White) to ord(Black) do S
```

или

```
For i:=0 to 6 do S .
```

Здесь *S* - произвольный оператор.

Как уже было отмечено, между значениями перечисляемого типа и порядковыми номерами этих значений установлено однозначное соответствие: первое значение в списке получает порядковый номер 0, второе - 1 и т.д. Максимальная мощность перечисляемого типа - 256 значений, поэтому фактически перечисляемый тип задает некоторое подмножество целого типа *byte*.

Пусть нам заданы следующие типы:

```
Type  Colors = (Black,Red,White);
      Ordinal =(One,Two,Three);
      Days = (Monday,Tuesday,Wednesday);
```

С точки зрения мощности и внутреннего представления все три типа эквивалентны:

```
ord(Black)=0;  ord(Red)=1;     ord(White)=2;
ord(One)=0;    ord(Two)=1;     ord(Three)=2;
ord(Monday)=0; ord(Tuesday)=1; ord(Wednesday)=2.
```

Однако, если определены переменные

```
Var  Col : Colors;
     Num : Ordinal;
     Day : Days;
```

то допустимы операторы

```
Col:=Black; Num:=Two; Day:=Wednesday;
```

но недопустимы

```
Col:=One; Num:=Tuesday; Day:=White.
```

Применение перечисляемых типов делает программу более наглядной и одновременно повышает ее надежность, так как можно контролировать значения, которые получают переменные перечисляемых типов в процессе выполнения программы (частный случай контроля ординальных типов с помощью директивы компилятора R).

Непосредственно вводить и выводить значения переменных перечисляемого типа нельзя. Это связано с тем, что в машинной программе отсутствует связь между именем перечисляемой константы и ее численным значением (в машинной программе значения перечисляемых переменных - это числа, в Паскаль-программе - имена констант). Однако ввод указанно-

го типа переменных возможен, если использовать аппарат приведения типов переменных. Вывод также можно организовать косвенным образом.

Пример 3.

```
Type  Colors = (Black,Blue,Green,Red,Brown,Yellow,White);
Var   k : byte;
      Col : Colors;
Begin
  Readln(k);
  Col:=Colors(k);
  Case Col of
    Black  : Writeln('Black');
    Blue   : Writeln('Blue');
    Green  : Writeln('Green');
    Red    : Writeln('Red');
    Brown  : Writeln('Brown');
    Yellow : Writeln('Yellow');
    White  : Writeln('White');
  end;
End.
```

Значение переменной *k*, вводимой в программе, не должно превышать мощность типа *Colors,* т.е. это значение должно быть в диапазоне от 0 до 6.

З А П И С И

Массив и множество объединяют в себе компоненты только одного типа. В записи можно объединить компоненты различных типов, как простых, так и составных.

Предположим, что в памяти ЭВМ необходимо сформировать архив данных по отделу кадров предприятия. Личная карточка может содержать следующие сведения:

- фамилия, имя, отчество;
- дата рождения;
- национальность;
- должность;
- зарплата по месяцам;
- номер отдела или цеха;
- дата поступления на работу;
- образование и др.

Для графы "Образование" могут указываться более подробные сведения:

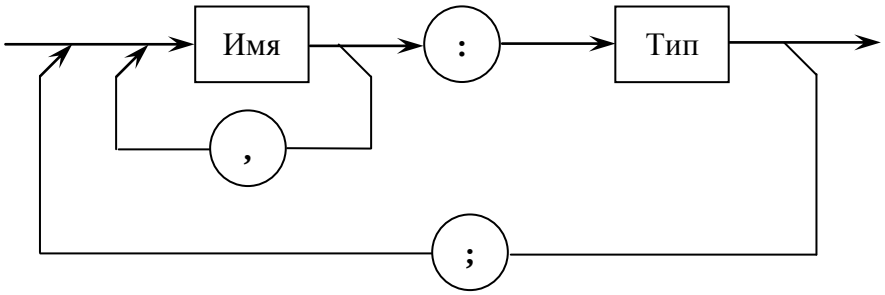
- если образование неполное среднее, то никаких дополнительных сведений;
- если образование среднее, то год окончания школы;
- если образование среднее специальное или высшее, то год окончания учебного заведения, его наименование, полученная специальность.

Данные в личной карточке имеют различный тип (целые числа, вещественные числа, строки и т.п.); кроме того, в этих данных есть фиксированная часть, одна и та же во всех карточках, и переменная часть, зависящая от графы "Образование". Эти данные размещаются соответственно в фиксированной и вариантной частях записи.

Структура типа записи:

Список полей может быть пустым.

Синтаксическая диаграмма списка полей:



Как видно из диаграммы, синтаксис списка полей записи аналогичен синтаксису раздела описания переменной.

Предположим, что нам требуется выполнить действия над комплексными числами. В Паскале нет данных комплексного типа. Такие данные программист должен сам определить в программе.

Комплексное число имеет действительную и мнимую части, например, 5,3 - 2 i ; 0 + i . Поскольку эти две части составляют единое целое, то комплексную переменную удобно представить в программе в виде записи, которая объединяет указанные два поля:

```
Type  Complex = record
        Re : real;      { действительная часть }
        Im : real;      { мнимая часть }
      end;
```

Поскольку типы для полей *Re* и *Im* одинаковы, то их можно объединить:

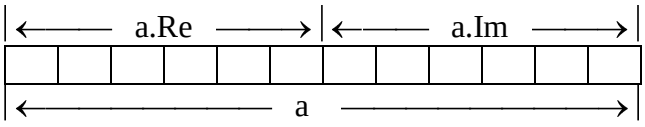
```
Type  Complex = record
        Re,Im : real;
      end;
Var  a,b : Complex;
```

Примечание. Re и Im – общепринятые обозначения в математике для действительной и мнимой частей комплексного числа (от слов Real – действительный и Implicit – неявный, мнимый).

В массиве длины всех его компонентов одинаковы. Поэтому для доступа к компоненту массива достаточно указать порядковый номер компонента, определяемый его индексом, после чего программа легко вычисляет местоположение этого компонента, т.е. его адрес в памяти.

В записи длины компонентов (полей записи) могут быть различными. Поэтому для доступа к полям записи используют составное имя, включающее в себя имя записи и имя поля, разделенные точкой. Например, *a.Re, b.Im* .

Структура поля *a:*



Здесь

a - имя поля памяти длиной 12 байт с адресом *a* + 0;

a.Re - имя поля памяти длиной 6 байт с адресом $a + 0$;
a.Im - имя поля памяти длиной 6 байт с адресом $a + 6$.

Ввод-вывод записей из текстовых файлов, как и массивов, осуществляется поэлементно:

```
Read(a.Re,a.Im);
Writeln('b= ',b.Re,'  ',b.Im).
```

Если две записи имеют одно и то же имя типа, то пересылку одной записи в другую можно выполнить одним оператором присваивания: $b := a$ (это справедливо для любых переменных, как простых, так и составных, если типы этих переменных тождественны).

Пример 1. Для комплексных переменных вычислить $z = x\,y + p\,q$.
Как известно,

$$(a_1 + b_1 i) + (a_2 + b_2 i) = (a_1 + a_2) + (b_1 + b_2) i\,;$$
$$(a_1 + b_1 i)(a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + a_2 b_1) i\ .$$

```
Program Comp;
Type  Complex = record
         Re,Im : real;
      end;
Var  x,y,p,q,z,r : Complex;
{ ---------------------------------- }
Procedure Sum(Var u,v,w : Complex);
Begin
  w.Re:=u.Re+v.Re; w.Im:=u.Im+v.Im;
End { Sum };
{ ---------------------------------- }
Procedure Mult(Var u,v,w : Complex);
Begin
  w.Re:=u.Re*v.Re-u.Im*v.Im;
  w.Im:=u.Re*v.Im+v.Re*u.Im;
End { Mult };
{ ---------------------------------- }
Begin
  Read(x.Re,x.Im,y.Re,y.Im,p.Re,p.Im,q.Re,q.Im);
  Mult(x,y,r); Mult(p,q,z);
  Sum(r,z,z);
  Writeln(z.Re,'  ',z.Im);
End.
```

Пример 2. В трехмерном пространстве заданы n точек своими координатами. Сгруппировать эти точки в порядке их удаления от начала координат.

Точки можно описать как четыре отдельных массива: абсцисс, ординат, апликат и расстояний от начала координат:

```
Type  Koor = array[1..100] of real;
Var  X,Y,Z,D : Koor;
```

В этом случае параметры одной и той же точки будут рассредоточены в четырех различных массивах, а для присваивания одной точке значений параметров другой точки потребуется четыре оператора присваивания:

```
x[i]:=x[j]; y[i]:=y[j]; z[i]:=z[j]; d[i]:=d[j] .
```

Каждая точка - это отдельный объект, обладающий определенными характеристиками, набор которых зависит от поставленной задачи. В данном случае рассматриваются четыре характеристики. Для каждой отдельной точки эти характеристики целесообразно объединить в одну группу, назвав их одним именем. Поскольку параметры *x, y, z* и *d* имеют одинаковый тип *real*, то это можно сделать с помощью массива или записи.

При использовании массива получим:

```
Type  PointType = array[1..4] of real;
Var  Point1,Point2 : PointType;
```

Так как *Point1* и *Point2* - это переменные одного типа, то для присваивания одной точке параметров другой точки достаточно одного оператора присваивания: *Point2:=Point1*.

Характеристики точки в описании ее типа *PointType* кодируются индексами массива. При разработке программы требуется постоянно помнить, что индекс 1 - это абсцисса, индекс 2 - ордината и т.д. Более удобной для описания отдельного объекта и его свойств является запись, позволяющая ставить в соответствие каждой характеристике объекта вполне определенное имя:

```
Type  PointType = record
         x,y,z,d : real
       end;
Var  Point1,Point2 : PointType;
```

Для присваивания точке значений параметров другой точки здесь также требуется лишь один оператор: *Point2:=Point1*.

Кроме вещественных характеристик, точка может иметь также характеристики другого типа. Например, в графических задачах в набор характеристик точки можно добавить признак видимости, принимающий значения *true* или *false*. В этом случае для объединения всех характеристик в одну группу массив вообще не может быть использован.

В программе примера 2 для описания точек используется массив записей, группировка точек производится методом "пузырька". При вводе точек в записях заполняются компоненты *Point.x, Point.y, Point.z;* компонент *Point.d* вычисляется в программе.

```
Program GroupPoints;
Const  Nmax = 1000;
Type  PointType = record
         x,y,z,d : real
       end;
       PointAr = array[1..Nmax] of PointType;
Var  Points : PointAr;       { исходный массив точек }
     Point : PointType;      { отдельная точка }
     i,m,
     n : integer;            { количество точек }
     Cond : boolean;
Begin
  Ввод  и  печать  n, Points
  For i:=1 to n do
    Points[i].d:=sqrt(sqr(Points[i].x)+sqr(Points[i].y)+
                      sqr(Points[i].z));
  Cond:=true; m:=n-1;
  While Cond do
    Begin
      Cond:=false;
      For i:=1 to m do
        If Points[i].d>Points[i+1].d then
```

```
            Begin
              Point:=Points[i]; Points[i]:=Points[i+1];
              Points[i+1]:=Point; Cond:=true
            End;
        Dec(m);
      End;
    Печать Points
End.
```

Пример 3. Для каждого из студентов группы указаны следующие сведения: фамилия и инициалы, дата рождения, национальность, год окончания школы, пол. Отпечатать список юношей старше 18 лет.

```
Program StudGroup;
Const  Nmax = 40;
Type  Date = record                { тип даты }
         Day : 1..31;              { день }
         Month : 1..12;            { месяц }
         Year : 1970 .. 2005;      { год }
       end;
       StudType = record           { тип личной карточки }
         Fam : string[20];         { фамилия и инициалы }
         BirthDay : Date;          { дата рождения }
         Nac : string[15];         { национальность }
         SchoolYear : 1980..2005;{ год окончания школы }
         Sex : char                { пол }
       end;
       Nummer = 0..Nmax;
       StudAr = array[Nummer] of StudType;
Var  i,k,n : Nummer;
     ch : char;
     Today : Date;                { текущая дата }
     Student : StudType;          { личная карточка }
     Group : StudAr;              { массив карточек }
     FileStud : text;             { исходный файл }
Begin
  Assign(FileStud,'Stud.dat');
  Reset(FileStud);
  n:=0;
  While not eof(FileStud) do
    Begin
      Inc(n);
      Read(FileStud,Student.Fam);
      Read(FileStud,Student.BirthDay.Day,
           Student.BirthDay.Month,Student.BirthDay.Year);
      Repeat
        Read(FileStud,ch);
      Until ch<>' ';
      Read(FileStud,Student.Nac);
      Insert(ch,Student.Nac,1);
      Read(FileStud,Student.SchoolYear);
      Repeat
        Read(FileStud,ch);
      Until ch<>' ';
      Student.Sex:=ch;
      Readln(FileStud);
      Group[n]:=Student;
    End;
```

```
  Close(FileStud);
  Writeln('Введите текущую дату: день  месяц  год ');
  Readln(Today.Day,Today.Month,Today.Year);
  Writeln('  СПИСОК ЮНОШЕЙ СТАРШЕ 18 ЛЕТ');
  k:=0;
  For i:=1 to n do
    If (Group[i].Sex='м') and (Today.Year-
        Group[i].BirthDay.Year>18) then
      Begin
        Inc(k);
        Writeln(k:2,'   ',Group[i].Fam);
      End;
End.
```

Комментарии к программе StudGroup:

1. В программе предполагается, что переменная *Student.Fam* записана с первой позиции строки текстового файла.

2. В одной строке входного файла одновременно размещены разнотипные элементы: числа и строки. Эти элементы отделены друг от друга одним или несколькими пробелами. Поскольку при вводе переменной типа ***string*** пробел воспринимается как обычный символ, то для индикации начала текстовой строки в цикле ***Repeat*** перебираются позиции строки файла до тех пор, пока в символьную переменную *ch* не будет прочитан символ, отличный от пробела, после чего читается строковая переменная *Student.Nac*. Поскольку первый символ графы "Национальность" был прочитан в переменную *ch,* то его добавление в переменную *Student.Nac* производится процедурой *Insert.*

3. Ввод строки из текстового файла осуществляется по объявленной в программе длине этой строки. Поэтому при формировании с клавиатуры текстового файла *Stud.dat* необходимо учитывать, что элемент *Student.Fam* должен занимать в строке файла не менее 20 позиций, а элемент *Student.Nac* - не менее 15 позиций.

4. В программе возраст вычисляется как разность между текущим годом и годом рождения. Более точно возраст нужно вычислять с учетом дня и месяца.

Программу StudGroup можно рассматривать как фрагмент программы учетно-административной задачи. Одним из элементов такой задачи является ввод из текстового файла с целью формирования архива соответствующих сведений (в данном случае архива личных карточек студентов).

Учетно-административные программы, как и другие промышленные программные системы, ориентированы на пользователей, которые не являются программистами. Одним из требований, предъявляемым к таким программам, является обеспечение максимально удобного сервиса.

С точки зрения пользователя, программа StudGroup имеет серьезный недостаток: при подготовке текстового файла необходимо постоянно следить, чтобы значение переменной *Student.Fam* занимало в строке файла не менее 20 позиций, а переменной *Student.Nac* - не менее 15 позиций, вне зависимости от текущей длины этих переменных. Для повышения сервиса пользователя более удобно было бы оставить лишь одно ограничение: элементы строки файла должны разделяться одним или несколькими пробелами. Пример реализации ввода записей *Student* из текстового файла с учетом указанного ограничения приведен ниже в программе ReadRecord.

```
Program ReadRecord;
Const  Nmax = 40;
Type  Date = record                { тип даты }
         Day : 1..31;              { день }
```

```
         Month : 1..12;          { месяц }
         Year : 1970 .. 2005;    { год }
       end;
       StudType = record          { тип личной карточки }
         Fam : string[20];        { фамилия и инициалы }
         BirthDay : Date;         { дата рождения }
         Nac : string[15];        { национальность }
         SchoolYear : 1980..2005;{ год окончания школы }
         Sex : char               { пол }
       end;
       Nummer = 0..Nmax;
       StudAr = array[Nummer] of StudType;
Var  i,k,k1,k2,n : byte;
     Today : Date;               { текущая дата }
     Student : StudType;         { личная карточка }
     Group : StudAr;             { массив карточек }
     S1,S2 : string[80];
     Code : integer;             { код преобразования }
     Cond : boolean;
     FileStud : text;            { исходный файл }
{ Текст функций  Space и NotSpace }
Begin
  Assign(FileStud,'Stud.dat');
  Reset(FileStud);
  n:=0;
  While not eof(FileStud) do
    Begin
      Inc(n);
      Readln(FileStud,S1);
      k:=0; k2:=0; Cond:=true;
      While Cond do
        Begin
          k1:=NotSpace(S1,k2+1);
          If k1=0 then
            Cond:=false
          Else
            Begin
              k2:=Space(S1,k1+1);
              If k2=0 then
                Begin
                  k2:=length(S1)+1; Cond:=false;
                End;
              Inc(k);
              S2:=Copy(S1,k1,k2-k1);
              Case k of
                1 : Student.Fam:=S2;
                2 : Student.Fam:=Student.Fam+' '+S2;
                3 : Val(S2,Student.BirthDay.Day,Code);
                4 : Val(S2,Student.BirthDay.Month,Code);
                5 : Val(S2,Student.BirthDay.Year,Code);
                6 : Student.Nac:=S2;
                7 : Val(S2,Student.ScoolYear,Code);
                8 : Student.Sex:=S2[1];
              end;
            End;
        End;
      Group[n]:=Student;
    End;
```

```
  Close(FileStud);
  Обработка массива записей Group и формирование
  выходных документов
End.
```

В программе ReadRecord очередная строка текстового файла вводится в строковую переменную *S1*. После этого с помощью функций *Space* и *NotSpace* выделяются отдельные слова строки *S1*, формирование компонент записи *Student* производится в операторе ***Case*** в соответствии с порядковым номером *k* выделенного слова. При этом учтено, что между фамилией и инициалами студента в строке исходного файла имеется по крайней мере один пробел.

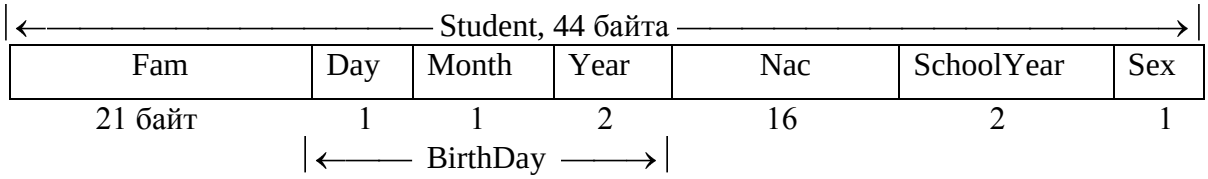
Примечание. Оператор

```
Student.Sex:=S2
```

был бы неверным, так как в левой части оператора записан символ (переменная типа char), а в правой части – строка (переменная типа **string**). В этом случае при трансляции было бы выдано сообщение «Type mismatch» (несоответствие типов).

ОПЕРАТОР ПРИСОЕДИНЕНИЯ

Рассмотрим структуру какого-либо поля памяти, соответствующего конкретной записи, например, поля *Student* типа *StudType* из предыдущего примера.



Каждое имя, объявленное в разделе ***Var*** Паскаль-программы, - это адрес вполне определенного поля памяти в машинной программе. Имя *Student* - это адрес поля памяти длиной 44 байта. Если в каком-либо операторе Паскаль-программы используется это имя, то речь идет об обработке всего поля размером 44 байта.

Пример:

```
Var  Student,Student1 : StudType;
Begin
  Student1:=Student;
```

Здесь производится чтение поля памяти *Student* и запись его содержимого в поле *Student1*, т.е. пересылка содержимого поля памяти из одного места в другое.

Если требуется обработать какое-либо внутреннее поле, входящее в состав записи *Student*, то для этого используется составное имя, в котором после общего имени записи указывается имя внутреннего поля.

Например,

```
  Student1.Fam:=Student.Fam; Student1.Sex:='м';
```

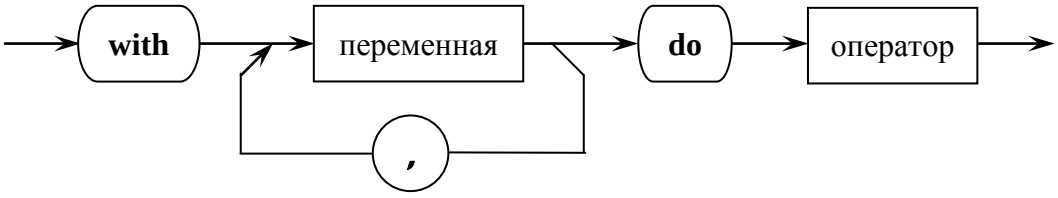
Здесь *Student.Fam* - адрес поля памяти длиной 21 байт, *Student.Sex* - адрес поля памяти длиной 1 байт.

Переменная *Student.BirthDay*, входящая в состав записи *Student*, в свою очередь является записью и определяет поле памяти размером 4 байта. Для доступа к компонентам этого поля аналогичным образом конструируется составное имя:

```
  Student.BirthDay.Day:=10; Student.BirthDay.Month:=11;
  Student.BirthDay.Year:=1975;
```

При обработке записей часто приходится использовать громоздкие составные имена. Для сокращения таких имен можно применить оператор присоединения. Такой оператор открывает область действия, содержащую имена полей записи, так что теперь эти имена можно указывать как имена обычных переменных.

Синтаксис оператора присоединения:



Переменная - это имя записи.

Например, в программе ReadRecord можно было бы записать:

```
With Student,BirthDay do
  Case k of
    1 : Fam:=S2;
    2 : Fam:=Fam+' '+S2;
    3 : Val(S2,Day,Code);
    4 : Val(S2,Month,Code);
    5 : Val(S2,Year,Code);
    6 : Nac:=S2;
    7 : Val(S2,SchoolYear,Code);
    8 : Sex:=S2[1];
  end;
For i:=1 to n do
  With Group[i],BirthDay do
    Writeln(i:2,' ',Fam,' ',Day:2,' ',Month:2,' ',
            Year:4,' ',Nac,' ',SchoolYear,' ',Sex);
```

Здесь в первом фрагменте полным именем переменной *Sex* является имя *Student.Sex,* а переменной *Year - Student.BirthDay.Year;* во втором фрагменте- соответственно *Group[i].Sex* и *Group[i].BirthDay.Year.*

При выполнении оператора ***With*** сначала в области его действия проверяется каждая ссылка на переменную, чтобы установить, является ли она полем записи. Если это так, то осуществляется доступ к полю записи даже в том случае, когда имеется переменная с таким же именем.

ОПИСАНИЕ ФАЙЛА

Ввод данных в ЭВМ производится с внешних устройств (например, с магнитной ленты, магнитного диска, с клавиатуры). Вывод данных также производится на внешние устройства (экран дисплея, принтер, магнитная лента или диск).

Группа данных на внешнем устройстве, объединенная одним именем, называется файлом (Files - картотека, скоросшиватель). Файл состоит из отдельных компонентов. Все компоненты должны быть одного типа. Однако в отличие от массива и множества количество компонентов в файле заранее не ограничивается. В частном случае файл может быть пустым, т.е. не содержать ни одного компонента.

В Турбо Паскале файлом считают также логическое устройство, являющееся потенциальным источником или приемником информации. Таким устройством может быть, например, клавиатура или принтер.

По методу доступа к компонентам файлы разделяются на последовательные и прямые. В последовательном файле чтение и запись компонентов могут быть только последовательными, начиная с первого компонента. Таким является, например, файл на магнитной ленте. В прямом файле можно прочесть или записать любой компонент, указав его номер в файле. При этом перебор предыдущих компонентов не требуется. В частности, на магнитном диске можно организовать как последовательный, так и прямой файлы.

По направлению передачи данных файлы разделяются на входные (ввод из внешнего устройства в оперативную память) и выходные (вывод из оперативной памяти на внешнее устройство).

Существуют три типа файлов:

1) типизированные, компоненты которых имеют жесткую структуру;
2) текстовые, состоящие из строк переменной длины;
3) нетипизированные, рассматриваемые как последовательность байтов.

Нетипизированные файлы здесь не будут обсуждаться. Они применяются для прямого доступа к любому файлу на диске независимо от его внутреннего формата и обеспечивают быстрый ввод или вывод информации; для этих файлов определены свои процедуры ввода-вывода (BlockRead и BlockWrite).

Описание файлового типа или файловой переменной имеет соответственно следующий вид:

1) **file of** <тип> (любое имя типа, кроме типа файла);
2) text;
3) **file.**

Ограничение на имя типа, указанное в первом описании, означает, что в Турбо Паскале не может быть создан файл, компонентами которого являются файлы.

Пример 1.

```
Type  FileInt = file of integer;
      Date = record
         Day : 1..31;
         Month : 1..12;
         Year : 1900 .. 2000
      end;
      Stud = record
         Fam : string[20];
         BirthDay : Date;
         Nac : string[15];
         Sex : char
      end;
      FileStud = file of Stud;
Var   F1 : FileInt;
      F2 : file of real;
      F3 : FileStud;
      F4 : text;
      F5 : file;
      x,y : real;
      b : boolean;
```

Здесь *F1, F2, F3* - типизированные файлы, но для описания *F1* и *F3* использовано имя типа (*FileInt, FileStud*), а для *F2* - непосредственно описание типа (***file of*** *real*). Длина компонента файла *F1* составляет 2 байта, файла *F2* - 6 байтов, файла *F3* - 42 байта.

Например, являются недопустимыми такие записи:

```
b:=F1>F2;
F1:=F2+F3;
F2:=x+y;
```

Переменная файлового типа не может входить в состав выражения, для нее недопустимы операции сравнения, она не может стоять в левой части оператора присваивания.

Имя файла может быть параметром процедуры или функции. В этом случае имя файла должно быть параметром-переменной, т.е. в списке формальных параметров перед ним должно стоять слово ***Var.*** В самом деле, при отсутствии слова ***Var*** файл как параметр-значение должен быть переслан из вызывающей программы в процедуру, что невозможно, поскольку компоненты файла находятся на внешнем устройстве, а не в оперативной памяти. При наличии слова ***Var*** пересылается лишь адрес файловой переменной, который, как и адрес любой переменной, имеет длину 4 байта.

При старте блока, в котором описаны файловые переменные, для них, как и для других переменных, выделяется поле памяти определенного размера: для текстового файла 256 байт, для типизированного и нетипизированного – 128 байт.

ДОСТУП К ФАЙЛАМ

В Паскаль-программе файл является переменной. Следовательно, он должен иметь определенное имя, описанное в разделе ***Var*** (например, имя *FileName*). С другой стороны, операционная система MS DOS использует имена файлов вида *'Point.dat', 'D:\Verta\Map.txt'* и т.п.

Отсюда можно сделать вывод, что один и тот же файл может иметь два разных имени: внутреннее имя, объявленное в разделе ***Var***, и внешнее имя, с которым этот файл известен операционной системе. С тем, чтобы организовать доступ к конкретному файлу, в программе должна быть описана связь между внутренним и внешним именами файла. Без такого описания Паскаль-программе доступны лишь два текстовых файла со стандартными именами *Input* (файл ввода с клавиатуры) и *Output* (файл вывода на экран дисплея). Любые другие файлы, а также логические устройства становятся доступны программе только после выполнения процедуры установления связи между именем файла в программе и именем файла в операционной системе. Для этого используется процедура

```
Assign(файловая-переменная,строка),
```

где "строка" - текстовое выражение, содержащее внешнее имя файла или имя логического устройства. В частном случае строка может иметь вид пути доступа к файлу.

Внешнее имя файла, задаваемое в процедуре *Assign,* формируется по правилам, принятым в системе MS DOS. Если в строке не указано имя диска, то принимается текущий диск; если в строке не указано имя каталога, то принимается текущий каталог.

Возможна другая интерпретация работы, выполняемой процедурой *Assign.*

Пусть мы имеем следующий фрагмент программы:

```
Var  FileName : text;
Begin
  Assign(FileName,'Fn.dat');
```

Здесь в разделе ***Var*** указано, что имя *FileName* - это файловая переменная, но никакого конкретного значения переменной *FileName* не присвоено. Эту работу выполняет процедура *Assign:* файловой переменной *FileName* присваивается значение строки-константы *'Fn.dat',* определяющей имя конкретного файла на диске.

Как уже ранее указывалось, при старте блока всем переменным, описанным в разделе ***Var***, выделяются поля памяти в соответствии с описанием их типа. Заполнение поля памяти переменной производится в разделе операторов. По отношению к файловой переменной можно считать, что имя файла на диске - это константа файлового типа, а процедура *Assign* выполняет присваивание файловой переменной значения файловой константы (при этом происходит частичное заполнение поля памяти, выделенного для файловой переменной; полное заполнение этого поля производится при открытии файла).

Второй вариант приведенного выше фрагмента программы:

```
Var  FileName : text;
     S : string;
Begin
  S:='Fn.dat'; Assign(FileName,S);
```

Здесь файловой переменной *FileName* присваивается значение строки-переменной *S,* которой предварительно было присвоено значение строки-константы *'Fn.dat'*.

ЛОГИЧЕСКИЕ УСТРОЙСТВА

Стандартные аппаратные средства ПЭВМ (клавиатура, экран терминала, принтер, коммуникационные каналы ввода-вывода) определяются в Турбо Паскале специальными именами, которые называются именами логических устройств. Все логические устройства рассматриваются как потенциальные источники или приемники информации.

Основными логическими устройствами являются *Con, Prn, Aux, Nul.*

Con - консоль (клавиатура или экран монитора). Различие между этими устройствами определяется по направлению передачи данных: чтение информации - с клавиатуры, запись информации - на экран.

Ввод с клавиатуры буферируется: символы по мере нажатия на клавиши помещаются в специальный строковый буфер, который передается программе только после нажатия на клавишу Enter. При вводе символов осуществляется их эхо-повтор на экране.

Prn (от PRiNter) - логическое имя принтера. Если к ПЭВМ подключено несколько принтеров, доступ к ним осуществляется по именам *Lpt1, Lpt2, Lpt3* (от Line PrinTer). Имена *Prn* и *Lpt1* первоначально синонимы. Если в программе указана фраза "***Uses** Printer*", то в этом случае стандартный модуль *Printer* объявляет имя файловой переменной *Lst* типа *text* и связывает ее с логическим устройством *Lpt1*, после чего производит открытие файла *Lst.*

Aux (Auxiliary - вспомогательный выход) - логическое имя коммуникационного канала, который обычно используется для связи с другими машинами.

Nul - логическое имя "пустого" устройства. Это устройство обычно используется в отладочном режиме и трактуется как приемник информации неограниченной емкости. При обращении к устройству *Nul* как к источнику информации выдается признак конца файла *eof.*

Пример.

```
Var  F1,F2 : text;
Begin
  Assign(F1,'Aux');
  Assign(F2,'Lpt2');
```

Имена логических устройств являются предопределенными и, как правило, не должны использоваться в программе для других целей.

ОТКРЫТИЕ ФАЙЛА

При открытии файла в поле файловой переменной записывается полная информация об открываемом файле: внутреннее и внешнее имя, тип файла, направление передачи данных, размер файла, дата его создания и др. Одновременно в оперативной памяти создается специальная область, называемая буфером ввода-вывода и предназначенная для ускорения операций передачи данных между памятью и внешними устройствами. Только после открытия файла могут быть выполнены операции ввода или вывода для данного файла.

В Турбо Паскале можно открыть файл только для чтения, только для записи, а также для чтения и записи одновременно. Для этого используются процедуры *Reset, Rewrite, Append.*

Reset(F) - открытие существующего файла. Если это текстовый файл, то допускается только чтение из файла; для типизированного файла допустимы как чтение, так и запись информации. В последнем случае можно реализовать обновление компонентов файла.

Если файл *F* уже был открыт, то при выполнении процедуры *Reset* этот файл сначала закрывается, а затем открывается. При этом указатель файла устанавливается в начальную позицию файла.

Rewrite(F) - открытие нового файла. Если это текстовый файл, то допускается только запись в файл; для типизированного файла допустимы как чтение, так и запись информации. Если файл *F* уже существует, то при срабатывании процедуры *Rewrite* файл *F* предварительно уничтожается.

Append(F) - открытие текстового файла для добавления новых компонентов в конец файла. Если файл уже был открыт ранее с помощью процедур *Reset* или *Rewrite,* то процедура *Append* вначале закрывает данный файл, а затем его открывает для добавления новых компонентов. При этом указатель файла устанавливается перед маркером конца файла.

Если в программе имеется фраза "***Uses** Printer*", то это эквивалентно наличию следующего фрагмента:

```
Var  Lst : text;
Begin
  Assign(Lst,'Lpt1'); Rewrite(Lst);
```

Быстродействие оперативной памяти определяется ее электронными компонентами, быстродействие внешних устройств (например, накопителя на магнитном диске) - их механическими компонентами. Эти показатели быстродействия отличаются между собой на два-три порядка. Для сокращения потерь машинного времени на операции обмена данными между внешними устройствами и оперативной памятью используется буфер ввода-вывода, представляющий собой область памяти, отводимую каждому файлу при его открытии. При записи в файл вся информация сначала направляется в буфер и там накапливается до тех пор, пока весь объем буфера не будет заполнен. Только после этого или после специальной команды сброса буфера происходит передача данных на диск. Аналогично при чтении из файла считывается не столько, сколько запрашивается в процедуре ввода, а сколько поместится в буфер. Процедуры ввода фактически выбирают данные из буфера до его исчерпания, после чего следует новая передача данных из файла в буфер.

По умолчанию размер буфера ввода-вывода для файла равен 2048 байт, но может регулироваться программистом. При вводе с клавиатуры максимальный размер буфера ввода равен 127 символов (соответственно строка символов на экране не может превышать указанного размера).

ПРОЦЕДУРЫ И ФУНКЦИИ ДЛЯ ФАЙЛОВ ЛЮБОГО ТИПА

1. Процедура **Close(F).** Выполняется закрытие файла *F*. Если буфер вывода заполнен не полностью, то его содержимое переносится в файл *F*, после чего из оперативной памяти удаляется служебная информация о файле, в том числе освобождается буфер ввода-вывода. Попытка закрыть уже закрытый или еще не открытый файл вызывает прерывание программы, поскольку в данном случае отсутствует необходимая информация о файле.

2. Процедура **Rename(F, строка).** Внешнему файлу, связанному с файловой переменной *F*, присваивается новое имя, заданное в строке. Применяется только для закрытого файла.

3. Процедура **Erase(F).** Файл *F* уничтожается. Процедура *Erase* применима только для закрытого файла.

4. Процедура **Truncate(F).** Файл усекается по текущей позиции указателя. В эту позицию записывается признак конца файла *eof*. Процедура применима только для открытого файла.

5. Функция **Eof(F):boolean.** Функция возвращает значение *true,* если достигнут конец файла; в противном случае выходное значение равно *false.*

6. Функция **IOResult:word** (Input-Output-Result). Возвращает условный признак последней операции ввода-вывода. Результат равен нулю, если операция ввода-вывода завершилась успешно. В противном случае выходное значение функции указывает на тип возникшей аварийной ситуации (например, 106 – неправильный численный формат, 160 – не включен принтер и т.д.).

Т Е К С Т О В Ы Е Ф А Й Л Ы

Текстовые файлы трактуются в Турбо Паскале как совокупность строк переменной длины. К каждой строке возможен лишь последовательный доступ, начиная с первой строки. В конце каждой строки ставится специальный признак (маркер) *eol* (end of line), а в конце файла - признак (маркер) *eof* (end of file).

Строка, состоящая лишь из маркера *eol,* называется пустой.

Слова *eol* нет в языке Паскаль и в программе оно не указывается. Определить, доступен ли конец строки *eol* в файле во время выполнения прграммы, можно с помощью описанной ниже функции Eoln(F).

При формировании текстового файла с клавиатуры признак *eol* генерируется при нажатии клавиши Enter, признак *eof* - при нажатии клавиш Ctrl+Z. Признак *eol* - это два последовательных байта с кодами по таблице ASCII #13 и #10. Признак *eof* - байт с кодом #26.

Для выполнения операций ввода-вывода в текстовых файлах используются процедуры *Read, Readln, Write, Writeln.*

Процедура *Read*(*F,x1,x2,...,xn*) эквивалентна следующей последовательности процедур:

```
Read(F,x1); Read(F,x2); ...; Read(F,xn).
```

Для *Readln*(*F,x1,x2,...,xn*) имеем аналогично:

```
Read(F,x1); Read(F,x2); ...; Read(F,xn); Readln(F).
```

Процедура *Readln(F)* производит поиск в файле *F* ближайшего маркера *eol* и устанавливает указатель файла на следующую после *eol* позицию, т.е. при этом производится переход на следующую строку текстового файла..

Процедура *Write*(*F,x1,x2,...,xn*) эквивалентна следующему:

```
Write(F,x1); Write(F,x2); ...; Write(F,xn).
```

Для *Writeln*(*F,x1,x2,...,xn*) имеем аналогично:

```
Write(F,x1); Write(F,x2); ...; Write(F,xn); Writeln(F).
```

Процедура *Writeln(F)* записывает в текущую позицию файла *F* маркер *eol.*

Формальные параметры, изначально описанные в процедуре *Read,* являются параметрами-переменными. Следовательно, как уже было указано в разделе «Процедуры ввода-вывода», элементами списка ввода могут быть только простые переменные и переменные с индексами.

Формальные параметры, изначально описанные в процедуре *Write,* являются параметрами-значениями. Следовательно, как уже было ранее указано в разделе «Процедуры ввода-вывода», элементами списка вывода являются выражения; в частном случае ими могут быть простые переменные, переменные с индексами, константы и функции.

Если из текстового файла читаются переменные типа *char,* то в этом случае читается содержимое каждого байта этого файла, в том числе коды #13, #10, #26 и др. Ввод переменных типа ***string*** осуществляется по текущей длине строк файла, если она не превышает объявленную длину строковой переменной; в противном случае читается количество символов, соответствующее длине этой переменной. В общем случае при вводе строк целесообразно использовать процедуру *Readln*, при вводе чисел - процедуру *Read*.

При вводе очередного числа процедура *Read* вначале производит его выделение из строки файла, а затем преобразование в числовой формат аналогично тому, как это делает процедура *Val.*

Для текстовых файлов могут использоваться три предописанные функции, неприменимые для файлов другого типа*: Eoln, SeekEoln* и *SeekEof.*

Функция **Eoln(F):boolean** возвращает значение *true*, если во входном потоке достигнут маркер конца строки *eol*.

Функция **SeekEoln(F):boolean** пропускает все пробелы и знаки табуляции до маркера конца строки *eol* или до первого значащего символа и возвращает значение *true*, если такой маркер обнаружен.

Функция **SeekEof(F):boolean** пропускает все пробелы, знаки табуляции и маркеры конца строки вплоть до маркера конца файла *eof* или до первого значащего символа и возвращает значение *true*, если такой маркер обнаружен.

Ввод из текстового файла рекомендуется организовывать с использованием функций *Eof* или *SeekEof.*

Пример. Ввести из текстового файла числовой массив, количество элементов которого заранее неизвестно.

```
Program InNum1;
Const  Nmax = 1000;
Type  Ar = array[1..Nmax] of real;
Var  n : integer;     { кол-во элементов в массиве }
     X : Ar;
     F : text;
Begin
  Assign(F,'F.dat'); Reset(F);
  n:=0;
  While not eof(F) do
    Begin
      Inc(n); Read(F,x[n]);
    End;
  Close(F);
End.
```

Программа InNum1, используемая для ввода числовых данных, имеет один недостаток. Если в файле между последним числом и маркером *eof* имеется хотя бы один пробел, то это вызовет ввод и соответственно включение в массив *X* дополнительного элемента, равного нулю, что может в дальнейшем исказить результаты обработки массива *X*. Этот недостаток можно устранить, если при вводе числовых данных вместо функции *Eof* использовать функцию *SeekEof*. Тогда соответствующий фрагмент программы InNum1 будет иметь вид:

```
While not SeekEof(F) do
  Begin
    Inc(n); Read(F,x[n]);
  End;
```

При вводе последовательности строк нужно использовать функцию *Eof*, поскольку функция *SeekEof* игнорирует пробелы в файле, несмотря на то, что эти пробелы могут входить в состав вводимых строк.

Ранее было сказано, что ввод переменных типа **string** производится по текущей длине строк файла. Рассмотрим в связи с этим следующий пример:

```
Var S : string;
    F : text;
While not eof(F) do
  Begin
    Read(F,S);
    Writeln(S)
  End;
```

После ввода первой строки текстового файла текущая длина этой строки становится равной нулю. Поскольку процедура Read не производит переход на следующую строку текстового файла, то в дальнейшем многократно вводится пустая строка, т.е. происходит зацикливание. В данном случае вместо процедуры Read нужно применять процедуру Readln.

ТИПИЗИРОВАННЫЕ ФАЙЛЫ

Все компоненты типизированного файла, в отличие от текстового файла, имеют одну и ту же длину. Это позволяет программе определить местоположение любого компонента по его номеру и осуществить прямой доступ к этому компоненту. Нумерация компонентов типизированного файла производится последовательностью чисел 0, 1, 2, 3,...

В любой момент работы программы из всех компонентов файла можно читать или записывать лишь один. Он называется доступным компонентом файла. Номер такого компонента является значением специальной внутренней переменной, которая существует, но не определяется в Паскаль-программе. Эта переменная называется указателем файла. Значение указателя изменяется при выполнении процедур обработки файла.

При открытии типизированного файла его указатель устанавливается на нулевой компонент. После выполнения каждой операции чтения или записи указатель сдвигается на следующий компонент.

В отличие от текстового файла, типизированный файл не содержит в себе маркера конца файла. В служебную информацию о файле операционная система записывает количество его компонентов, что и используется в дальнейшем для определения конца этого файла.

Произведем сравнение текстового и типизированного файлов по двум критериям: скорости передачи информации и объему занимаемой памяти.

Пример 1.

```
Var  k : integer;
     F1 : text;
     F2 : file of integer;
Begin
   Assign(F1,'F1.dat'); Assign(F2,'F2.dat');
   Rewrite(F1); Rewrite(F2);
   k:=1598;
   Write(F1,k); Write(F2,k);
```

В оперативной памяти переменная *k*, соответствующая типу *integer*, занимает два байта. Значение 1598 в шестнадцатеричной записи имеет вид 063B. Компонент файла *F2* также занимает два байта (тип *integer*). Внутреннее представление переменной *k* в оперативной памяти и в файле *F2* одно и то же. Следовательно, никакого преобразования формы представления данных при записи в типизированный файл не происходит.

В текстовом файле, как и в строке, каждая цифра числа занимает один байт (в коде ASCII). Для значения *k* = 1598 в файле *F1* будет отведено 4 байта. Форма представления значения *k* в оперативной памяти и в файле *F1* различна; поэтому при записи чисел в текстовый файл некоторая часть машинного времени будет затрачиваться на преобразование формы их представления. Следовательно, скорость передачи данных в типизированном файле всегда выше, чем в текстовом файле; объем внешней памяти, занимаемой типизированным файлом, в общем случае меньше по сравнению с текстовым файлом.

Некоторым недостатком типизированного файла можно считать то, что содержание этого файла нельзя прочесть или изменить с помощью текстового редактора (в ряде случаев это достоинство, а не недостаток).

Для операций чтения и записи в типизированном файле используются процедуры *Read* и *Write* (но не *Readln, Writeln,* поскольку в типизированном файле нет маркеров конца строки). Для типизированных файлов определены также предописанные процедура *Seek* и функции *FileSize, FilePos.*

1. Процедура **Seek(F, k)** перемещает указатель файла *F* на компонент с номером *k.* Переменная *k* должна иметь тип *longint.*

2. Функция **FileSize(F):longint** возвращает текущий размер файла *F* (количество компонентов файла).

3. Функция **FilePos(F):longint** возвращает номер текущей позиции файла *F.*

С учетом способа нумерации компонентов типизированного файла последний его компонент имеет номер *FileSize*(*F*)-1.

Примеры использования процедуры *Seek* и функций *FileSize, FilePos:*

Seek(*F*,0) - установка указателя на первый компонент файла *F* (на компонент с нулевым порядковым номером);

Seek(*F*,*FileSize*(*F*)) - установка указателя на конец файла *F* для дописывания новых компонентов*;*

Seek(*F*,*FilePos*(*F*)+1) – пропуск компонента файла.

Процедурой *Read* вводится из типизированного файла полностью один компонент вне зависимости от того, является он простым или составным (из текстового файла составные переменные вводятся поэлементно). Процедура *Write* записывает в типизированный файл также один компонент вне зависимости от его типа.

Как и для текстового файла, процедура *Read(F,a,b,c)* эквивалентна следующей последовательности процедур:

Read(F,a); Read(F,b); Read(F,c).

Аналогично процедура *Write(F,a,b,c)* эквивалентна последовательности

Write(F,a); Write(F,b); Write(F,c).

Следует обратить внимание, что в списке вывода процедуры *Write* для типизированного файла, в отличие от текстового файла, могут быть только переменные того же типа, что и тип компонента файла (но не константы, функции или выражения).

Пример 2.

```
Var  F1 : file of integer;
Begin
  Assign(F1,'F1.dat');
  Rewrite(F1);
  Write(F1,5);
```

При трансляции фрагмента, приведенного в примере 2, будет получено сообщение "Error 20: Variable identifier expected" (отсутствует идентификатор переменной), поскольку в процедуре Write указана константа, а не переменная.

Здесь нужно:

```
Var  k : integer;
.....
k:=5;
Write(F,k);
```

Причина запрета использования констант в процедуре Write состоит в следующем. Как было ранее указано, при записи в типизированный файл содержимое поля памяти полностью переписывается в компонент файла, т.е. преобразование этого содержимого не производится. В программе для числовой константы отводится минимально необходимое поле памяти, что не гарантирует совпадение ее типа с типом компонента файла. В частности, в примере 2 константа 5 будет занимать один байт памяти, в то время как компонент файла по описанию имеет размер 2 байта.

Пример 3.

```
Type  Ar = array[1..100] of real;
      RecType = record
        a,b : real;
        m,n : integer;
        S : string
      end;
Var  i : byte;
     P,Q,R : Ar;
     Rec : array[1..10] of RecType;
     F1 : file of Ar;
     F2 : file of RecType;
Begin
  Assign(F1,'F1.dat'); Assign(F2,'F2.dat');
  Reset(F1); Reset(F2);
  Read(F1,P,Q,R);
  For i:=1 to 10 do
    Read(F2,Rec[i]);
```

Компонентами файла *F1* являются массивы типа *Ar*, файла *F2* - записи типа *RecType*. При обращении к файлу *F1* читается полностью один массив (*P*, *Q* или *R*), к файлу *F2* - запись, являющаяся компонентом массива записей *Rec*.

Пример 4.

```
Var  S : string[66];
     F : file of string[66];
Begin
  Assign(F,'F.dat');
  Rewrite(F);
  S:='0123456789';
  Write(F,S);
.....................
```

При трансляции этого фрагмента будет выдано сообщение "Error 26: Type mismatch" (несоответствие типов). Причиной является то, что для переменной *S* и компонента файла *F* использовано описание, а не имя типа, в связи с чем компилятор формально считает их типы не соответствующими друг другу. Отмеченная ошибка, обнаруживаемая при трансляции, аналогична следующему:

```
Var  A : array[1..66] of byte;
     B : array[1..66] of byte;
Begin
  B:=A;
```

Здесь также будет сообщение этапа компиляции о несоответствии типов.

Правильная запись примера 4:

```
Type  string66 = string[66];
      FileString = file of string66;
Var  S : string66;
     F : FileString;
Begin
  Assign(F,'F.dat');
  Rewrite(F);
  S:='0123456789';
  Write(F,S);
.....................
```

В типизированном файле отсутствуют не только маркеры конца строки, но и маркер конца файла. Вместо последнего в поле файловой переменной программа записывает количество компонентов файла, которое содержится в нем в данный момент.

Примечание. Пример 4 подтверждает одно из правил структурного программирования: в разделе ***Var*** всегда нужно использовать имена типов, а не их описания.

ПРИМЕРЫ ОБРАБОТКИ ФАЙЛОВ

В рассматриваемых ниже примерах при обработке файлов массивы не используются, поскольку размер файла в общем случае может превышать максимально возможный размер переменной, равный 64 Кбайт.

Пример 1.

Компонентами типизированного файла являются вещественные числа. Удалить из состава файла максимальный элемент.

Примечание. Практическим примером рассматриваемой задачи является следующая: из архива студентов факультета, расположенного на дисковом файле, удалить запись студента, отчисленного за неуспеваемость.

Ниже приведено два варианта решения поставленной задачи.

В программе DelFile1 вначале производится поиск номера *kmax* максимального компонента файла, после чего все компоненты, расположенные правее *kmax,* сдвигаются на один шаг влево. Поскольку размер файла при удалении компонента должен сократиться, то файл "обрезается" по последнему компоненту с помощью процедуры *Truncate*.

```
Program DelFile1;
Type  FileReal = file of real;
Var  kmax : longint; { номер максимального компонента }
     R,              { значение текущего компонента }
     Rmax  : real;   { значение максимального компонента }
     F : FileReal;
Begin
  Assign(F,'F.dat'); Reset(F);
  If FileSize(F)<2 then
    Rewrite(F)
  Else
    Begin
      Read(F,Rmax); kmax:=0;
      While not eof(F) do        { Определение местоположе- }
        Begin                    { ния максимального компо- }
          Read(F,R);             { нента файла              }
          If R>Rmax then
            Begin
              Rmax:=R; kmax:=FilePos(F)-1;
            End;
        End;
      If kmax<FileSize(F)-1 then
        Begin
          Seek(F,kmax+1);
          While FilePos(F)<FileSize(F) do
            Begin                    { Сдвиг компонентов файла,}
              Read(F,R);             { начиная с номера kmax+1,}
              Seek(F,FilePos(F)-2); { на один компонент }
              Write(F,R);            { влево (при этом }
              Seek(F,FilePos(F)+1); { элемент с номером kmax  }
            End;                     { будет уничтожен)      }
        End;
      Seek(F,FileSize(F)-1);    { Удаление последнего ком- }
      Truncate(F);              { понента файла           }
    End;
  Close(F);
End.
```

В программе DelFile1 учитывается, что после выполнения процедуры Read указатель файла перемещается на следующий компонент, поэтому переменной *kmax* присваивается значение FilePos(F)-1.

При анализе программы DelFile1 следует иметь в виду, что функция FileSize(F) определяет количество компонентов в файле *F*, при этом номером последнего компонента является значение FileSize(F)-1; в процедуре Seek(F,k) параметр *k* - это номер компонента, на который устанавливается указатель файла *F*.

В программе DelFile2 в файл *F2* переписываются из файла *F1* все компоненты, кроме максимального. После этого файл *F1* уничтожается, а файл *F2* переименовывается, получая имя, которое первоначально имел файл *F1*.

Следовательно, после выполнения процедуры Erase(F1) на диске уже не существует файла *F1.dat*, с которым была связана файловая переменная *F1*. При выполнении вслед за этим процедуры Rename(F2,'F1.dat') реально существующий файл *F2.dat*, с которым в настоящий момент связана файловая переменная *F2*, получает новое имя *F1.dat*. Таким образом, в результате выполнения указанных процедур файловые переменные *F1* и *F2* будут связаны с одним и тем же дисковым файлом *F1.dat*.

```
Program DelFile2;
Type  FileReal = file of real;
Var  kmax : longint;     { номер максимального компонента }
     R,                  { значение текущего компонента }
     Rmax : real;        { значение максимального компонента }
     F1,F2 : FileReal;
Begin
  Assign(F1,'F1.dat'); Reset(F1);
  If FileSize(F1)<2 then
    Begin
      Rewrite(F1); Close(F1)
    End
  Else
    Begin
      Assign(F2,'F2.dat'); Rewrite(F2);
      Read(F,Rmax); kmax:=0;
      While not eof(F1) do
        Begin
          Read(F1,R);
          If R>Rmax then
            Begin
              Rmax:=R; kmax:=FilePos(F)-1;
            End;
        End;
      Seek(F1,0);
      For k:=0 to kmax-1 do              { 1 }
        Begin
          Read(F1,R); Write(F2,R);
        End;
      Seek(F1,kmax+1);
      While not eof(F1) do               { 2 }
        Begin
          Read(F1,R); Write(F2,R);
        End;
      Close(F1); Close(F2);
      Erase(F1); Rename(F2,'F1.dat');
    End;
End.
```

Если максимальный компонент расположен в начале файла, то цикл 1 работать не будет; если этот компонент является последним в файле, то не будет отрабатываться цикл 2.

Структура программы DelFile2 более предпочтительна, по крайней мере в смысле читабельности.

Пример 2. Перед *k*-ым компонентом типизированного файла $(0 \le k < FileSize(F))$ вставить новое значение *b*.

Здесь, как и в примере 1, могут быть два варианта решения:

- с помощью процедуры Seek сдвинуть компоненты *k*, *k*+1, *k*+2, … вправо на одну позицию (начиная с последнего компонента), после чего записать в *k*-ый компонент новое значение *b*;

- в буферный файл F2 переписать из файла F1 компоненты 0, 1, …, *k*-1, затем новое значение *b* и все остальные компоненты файла F1; после этого уничтожить файл F1 и переименовать файл F2.

Конкретно программную реализацию примера 2 предлагается выполнить самостоятельно.

Пример 3.

В текстовых файлах *F1* и *F2* содержатся целые числа, упорядоченные по возрастанию. Объединить в файле *F3* содержимое файлов *F1* и *F2,* сохранив их упорядоченность. Сортировку данных не производить. Учесть, что в частном случае файлы *F1* и *F2* могут быть пустыми.

```
Program ComFiles;
Label  10;
Var  k1,k2 : integer;
     Cond : boolean;
     F1,F2,F3 : text;
Begin
  Assign(F1,'F1.dat'); Reset(F1);
  Assign(F2,'F2.dat'); Reset(F2);
  Assign(F3,'F3.dat'); Rewrite(F3);
  If SeekEof(F1) or SeekEof(F2) then
    Goto 10;
  Read(F1,k1); Read(F2,k2); { Чтение первого компонента }
  Cond:=true;
  While Cond do          { Цикл выполняется до тех пор, пока }
    If k1>k2 then        { не будет исчерпан файл F1 или F2  }
      Begin
        Writeln(F3,k2);
        If SeekEof(F2) then { При каждом выполнении цикла  }
          Begin             { из двух сравниваемых чисел   }
            Cond:=false;    { k1 и k2 в файл F3 записывает-}
            Writeln(F3,k1)  { ся меньшее, после чего пере- }
          End               { сланное число заменяется но- }
        Else                { вым значением из соответству-}
          Read(F2,k2);      { ющего файла                  }
      End
    Else
      Begin
        Writeln(F3,k1);
        If SeekEof(F1) then
          Begin
            Writeln(F3,k2); Cond:=false
          End
        Else
          Read(F1,k1);
      End;
    10:
    While not SeekEof(F1) do    { В F3 переписывается ос-}
      Begin                     { таток из файла F1 или  }
```

```
      Read(F1,k1); Writeln(F3,k1) { из файла F2. Если один }
    End;                          { из исходных файлов пус-}
  While not SeekEof(F2) do        { той, в F3 полностью пе-}
    Begin                         { реписывается другой    }
      Read(F2,k2); Writeln(F3,k2){ файл. Если F1 и F2 пус-}
    End;                          { тые, F3 также остается }
  Close(F1); Close(F2); Close(F3);{ пустым.              }
End.
```

АДРЕСАЦИЯ ПАМЯТИ

Турбо Паскаль работает в операционной системе MS DOS, впервые реализованной на процессорах малой производительности 8086, 8088. В этих процессорах максимальный объем оперативной памяти равен 1 Мбайт. Тогда адресное пространство такой памяти - это последовательность байтов с адресами 00000 .. FFFFF (FFFFF = 100000_{16} -1 = 16^5 - 1 = 2^{20}-1). Следовательно, для представления полного физического адреса здесь требуется 20 бит.

Обработка информации в процессоре, в том числе и адресация операндов, выполняется на регистрах, размер которых составляет 16 бит. Поскольку полный физический адрес не может быть записан на таком регистре, то адрес операнда размещается на двух регистрах и записывается в программе в виде пары чисел типа *word*, разделенных двоеточием:

СЕГМЕНТ : СМЕЩЕНИЕ

Сегмент - это блок памяти размером 64 Кбайта. Адрес сегмента может быть произвольным в пределах емкости основной памяти. Смещение - это номер байта по отношению к началу сегмента. Смещение изменяется в пределах $0000 .. $FFFF = 0 .. 65535.

Именно смещение определяет максимальный размер сегмента, а, следовательно, и максимальный размер переменной в Паскаль-программе.

Начальный адрес сегмента всегда кратен 16. Поскольку при этом последние четыре двоичные цифры равны нулю, то эти цифры в сегменте не указываются. Тогда абсолютный адрес байта

```
Adr = 16 Seg + Ofs,
```

где *Seg* - адрес сегмента; *Ofs* - смещение (offset - смещение, уступ).

Сегменты могут перекрываться. Вследствие этого адрес одного и того же байта может быть указан различными способами.

Пример. а) `$0020:$0010;  Adr = $00210`
б) `$001F:$0020;  Adr = $00210.`

Фрагмент памяти размером 16 байт называют параграфом. Поэтому можно сказать, что сегмент адресует память с точностью до параграфа, а смещение - с точностью до байта.

В программе адреса операндов, как правило, нормализованы. Нормализованным считают такой адрес, в котором смещение находится в пределах 0 .. 15 ($0000 .. $000F), т.е. в пределах одного параграфа. В этом случае

```
Seg = Adr div 16;  Ofs = Adr mod 16.
```

Размер сегмента определяет максимальный объем памяти, выделяемой для какого-либо объекта программы (сегмент данных, сегмент стека, программный модуль, динамическая переменная). Этот размер равен 65520 байт (64 Кбайт – 16 байт).

Примечание. Адресное пространство размером 1 Мбайт - это так называемая стандартная память. В современных компьютерах размер оперативной памяти значительно превышает 1 Мбайт. Тем не менее Турбо Паскаль вне зависимости от реального размера памяти компьютера может использовать лишь стандартную память (работу с полным объемом памяти

может обеспечивать Borland Pascal или программная система Delphi, ориентированные на операционную систему Windows).

АДРЕСНЫЙ ТИП ДАННЫХ

Адресный тип данных определяется предописанным словом *pointer* (указатель):

```
Var  P : pointer;
```

Значением переменной типа *pointer* является адрес поля памяти. Для размещения такой переменной отводится два слова, т.е. четыре байта. В первом слове хранится адрес сегмента, во втором - смещение.

Для указателей допустимы операция присваивания и две операции отношения ("=" и "<>"). Операция присваивания применима для двух случаев:

- присваивание значения другого указателя;
- присваивание значения предописанной константы ***nil***, определяющей пустое значение указателя.

Операции ввода-вывода для указателей недопустимы.

Пример 1.

```
Var  P,Q,R : pointer;
Begin
  P:=nil; Q:=P;
  If P<>Q then
    R:=P;
```

Внутреннее представление константы ***nil*** имеет вид $0000:$0000. Это не нулевой адрес памяти. Значение *P* = ***nil*** означает, что указатель *P* является неопределенным, он не адресует никакого поля памяти.

Для работы с указателями могут быть использованы следующие функции:

1) **Addr(X) : pointer** - адрес переменной *X;*
2) **Seg(X) : word** - сегмент адреса переменной *X*;
3) **Ofs(X) : word** - смещение адреса переменной *X*;
4) **Ptr(Sg,Of:word) : pointer** - формирование указателя, задаваемого сегментом *Sg* и смещением *Of.*

Для указателя переменной может быть использован также оператор получения адреса *@*, что аналогично по своему действию функции *Addr: @ X ≡ Addr*(*X*) .

Пример 2.

```
Var  P,Q,R : pointer;
     m,n : integer;
     a,b : real;
     Sg,Of : word;
Begin
  P:=Addr(m); Q:= @b;
  Sg:=Seg(m); Of:=Ofs(m);
  R:=Ptr(Sg,Of);
```

Как уже было отмечено, значение указателя не может быть выведено на печать. Тем не менее адрес, определяемый указателем *P,* может быть отпечатан по частям в виде значений двух целых переменных, определяющих сегмент и смещение. Эта работа может быть выполнена несколькими способами.

Пример 3.

Способ 1.

```
Var  b : real;
     Sg,Of : word;
     P : pointer;
Begin
  P:= @b; Sg:=Seg(b); Of:=Ofs(b);
  Writeln('Сегмент  ',Sg,'  Смещение  ',Of);
```

Способ 2.

```
Var  b : real;
     P : pointer;
Begin
  P:= @b;
  Writeln('Сегмент  ',Seg(b),'  Смещение  ',Ofs(b));
```

Для примера 3 соотношение между полями *P* и *b* может быть изображено так, как это показано на рис.1.

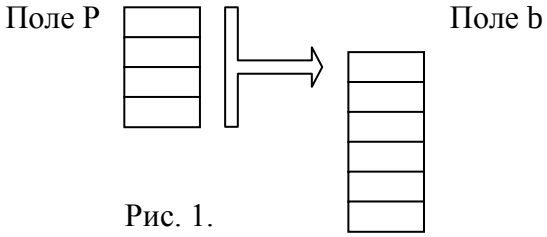


Рис. 1.

Ссылка на поле *b* (чтение содержимого поля *b* или запись значения в поле *b*) может быть организована двумя способами: по имени *b* (прямая адресация) или по адресу, хранящемся в поле *P* (косвенная адресация).

В первом случае записывают *a* := *b* или *b* := 7 . Во втором случае, чтобы указать, что речь идет о содержимом поля памяти, адрес которого находится в поле *P*, пишут *P*^ (читается "P тильда").

Рассмотрим еще один пример.

Пример 4.

```
Var  R : pointer;
     Sg,Of : word;
Begin
  Sg:=$B100; Of:=$0000;
  R:=Ptr(Sg,Of);
```

Здесь соотношение между указателем *R* и полем, которое адресуется этим указателем, проиллюстрировано на рис.2.

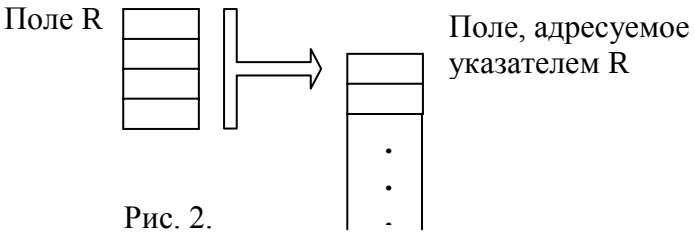


Рис. 2.

Имя *R* в программе определяет лишь то, что в четырехбайтном поле *R* может быть размещен адрес любого поля памяти, но не определяет никаких атрибутов адресуемого поля. Поэтому в данном случае нельзя написать, например, *R*^ := 7 (транслятор не имеет информации о том, что может быть размещено в поле *R*^ - целое число, вещественное число, строка или значение переменной другого типа).

Ранее указывалось, что адрес поля памяти – это адрес его крайнего левого байта. Никакой информации о размере и структуре поля памяти этот адрес не содержит. Такая информация определяется, как правило, типом переменной, которая поставлена в соответствие с данным адресом.

В программе, иллюстрируемой рисунком 1, атрибуты поля *b* известны. Тем не менее здесь, как и для поля *R^*, нельзя написать *P^* := 7, ибо на этапе трансляции невозможно определить, адрес какого поля памяти будет занесен в поле *P* в процессе выполнения программы. Принципиальную непредсказуемость заполнения поля *P* можно наблюдать на следующем фрагменте программы:

```
Read(Sg,Of); P:=Ptr(Sg,Of);
```

Для обработки полей, адресуемых указателем, можно использовать аппарат приведения типов. Тогда мы можем записать, например,

```
real(R^):= 7.85;  integer(P^):= 7.
```

ДИНАМИЧЕСКИЕ ПЕРЕМЕННЫЕ

Все переменные, действующие в программе, перечисляются в разделах ***Var*** блоков программы. Каждая переменная имеет определенное имя. Транслятор, анализируя разделы ***Var,*** формирует специальные таблицы, в которых указывается, сколько памяти в сегменте данных или в сегменте стека должно быть выделено каждой переменной. Реальное выделение памяти для переменной производится при активизации блока, т.е. при запуске программы или при обращении к процедуре или к функции. Память, выделенная переменным при активизации блока, сохраняется за ними на весь период работы блока. Такие переменные называются статическими. Доступ к статической переменной, т.е. чтение или запись ее значения, осуществляется по имени переменной. Здесь имеет место прямая адресация переменной.

В Паскаль-программе имеется также возможность выделять память для переменных в процессе работы программного блока. Такие переменные называются динамическими. Доступ к динамической переменной осуществляется не по имени (ибо имени они не имеют), а по адресу, который называется в этом случае ссылкой, или указателем. Форма доступа в основном аналогична той, которая принята для адресного типа данных. Используемая в этом случае адресация называется косвенной.

Разницу между статическими и динамическими переменными можно проиллюстрировать следующим примером. Предположим, что места в аудитории пронумерованы, как в зрительном зале. Преподаватель может сказать: "Студент Иванов, предъявите домашнее задание". Он может обратиться иначе: "Студент, сидящий на втором месте пятого ряда, выйдите к доске". В первом случае - это обращение по имени (прямая адресация), во втором - обращение по адресу (косвенная адресация).

Динамические переменные наиболее часто используют для реализации связанных структур данных. Отдельные виды таких структур называют стек, очередь, дек, дерево и др.

Проиллюстрируем некоторые особенности связанной структуры на примере очереди к врачу. Каждый пациент занимает произвольное место возле кабинета врача, но знает, за кем он стоит. Другими словами, каждый элемент очереди содержит ссылку на следующий элемент (рис.3).

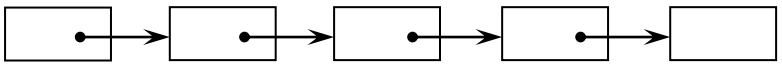


Рис.3.

Пусть один из пациентов покидает очередь. Это не требует перемещения в пространстве остальных пациентов. Просто пациент, следующий после убывшего, запоминает нового человека. В этом случае изменяется значение ссылки у одного элемента очереди (рис.4).

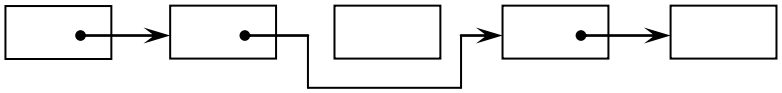


Рис.4.

После исключения элемента из очереди соответствующую ячейку памяти (стул возле кабинета) можно освободить и использовать для других целей.

Чтобы вставить новый элемент в очередь, здесь достаточно изменить две ссылки (рис.5). При этом, как и ранее, не требуется перемещения элементов очереди в пространстве.

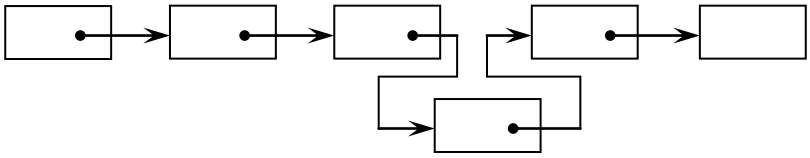


Рис.5.

В программе адрес динамической переменной - это содержимое указателя, который называют также ссылочной переменной. Ссылочная переменная имеет имя, описывается в разделе ***Var*** и является, следовательно, статической**.**

Как уже было отмечено, ссылочная переменная содержит адрес динамической переменной заданного типа. Тип динамической переменной описывается следующим образом:

Type <тип ссылочной переменной> = ^ <тип динамической переменной>

Пример 1.

```
Type  AnsPoint = ^integer;
      BerPoint = ^real;
      RecType = record
        a,b : real;
        m,n : integer
      end;
      RecPoint = ^RecType;
Var  P1,P2 : AnsPoint;
     Q1,Q2 : BerPoint;
     R1,R2 : RecPoint;
```

Адресные переменные, описанные в предыдущем разделе, и ссылочные переменные во многом совпадают между собой. Общим между ними является то, что каждая из них описана в разделе **Var**, занимает четыре байта и определяет адрес некоторого поля памяти. Различия:

1) Присваивание конкретных значений адресной переменной производится оператором @, функцией *Addr* или процедурой *Ptr*. Оператор @ и функция *Addr* присваивают этой переменной адрес уже известной, т.е. объявленной ранее в разделе ***Var*** переменной. Процедура *Ptr* формирует адрес поля памяти по заданным значениям сегмента и смещения. Ссылочным переменным конкретное значение адреса, как это будет показано ниже, присваивается с помощью процедуры *New*.

2) Для адресных переменных атрибуты адресуемых полей памяти на этапе трансляции неизвестны. Поэтому для обработки адресуемого поля используется аппарат приведения ти-

пов. Ссылочные переменные всегда задают адрес поля памяти уже известного типа. Поэтому обработка адресуемого поля может выполняться без приведения типов. Например, для примера 1 можно написать

```
                    P1^:=34;  Q1^:=-18.55;
R1^.a:=5.8;  R1^.b:=-13.3;  R1^.m:=6;  R1^.n:=16.
```

Как адресные, так и ссылочные переменные часто называют указателями. Если по тексту требуется подчеркнуть различие между двумя видами таких переменных, в случае необходимости ссылочные переменные будем называть типизированными указателями.

Рассмотрим формирование указателей для связанных структур данных.

Динамическая переменная, используемая для реализации связанных структур, имеет тип записи. Эта запись содержит две части: информационную (например, числа) и ссылку на другую динамическую переменную такого же типа.

Предположим, что тип указателя мы обозначили именем *PoinType,* а тип динамической переменной - именем *DynType.* Пусть информационная часть динамической переменной - это одно число типа *integer*. Тогда описание типов ссылочной и динамической переменных может иметь следующий вид:

```
Type  PoinType = ^DynType;
      DynType = record
         Inf : integer;
        Next : PoinType;
      end;
Var  L,R,P : PoinType;
```

Имена *L, R, P* - это имена переменных типа *PoinType*, т.е. в данном случае имена указателей. Эти имена описаны в разделе ***Var***; следовательно, *L, R, P* - статические переменные. При активизации программного блока каждой из этих переменных выделяется поле памяти длиной 4 байта, но содержимое этих полей остается неопределенным.

В Паскаль-программе любое имя может быть использовано только после его описания. Однако в приведенном выше описании типа это правило нарушено. В самом деле, транслируя фразу «PoinType = ^DynType», компилятор встречает имя DynType, которое еще не было описано. Если переставить местами описание указателя PoinType и описание записи DynType, то такая же ситуация возникнет по отношению к имени PoinType.

По поводу упомянутого выше правила в Паскале имеется еинственное исключение: описание типа указателя, в котором используется тип динамической переменной, должно предшествовать описанию типа динамической переменной.

Это связано с тем, что в приведенном выше описании `^DynType` – это тип указателя, размер которого всегда равен 4 байта вне зависимости от того, что собой представляет переменная `DynType`. В то же время описание

```
Type  DynType = record
         Inf : integer;
        Next : PoinType;
      end;
      PoinType = ^DynType;
```

будет воспринято компилятором как ошибка, поскольку к моменту трансляции записи `DynType` неизвестно, что определяет еще неописанная переменная `PoinType`.

Резервирование поля памяти для динамической переменной и засылка адреса этого поля в ссылочную переменную выполняется предописанной процедурой *New*(*L*), где *L* - имя

ссылочной переменной (имя указателя). При этом выделяется столько байтов памяти, сколько требует тип динамической переменной, с которой связан указатель *L*.

При запуске Паскаль-программы формируется специальная таблица, содержащая сведения об участках свободной памяти. При срабатывании процедуры *New* из этой таблицы выбирается нужный участок, адрес которого присваивается соответствующему указателю. Информация о выбранном участке вычеркивается из таблицы свободной памяти.

Пусть в памяти ЭВМ размещены два списка динамических переменных (рис.6).

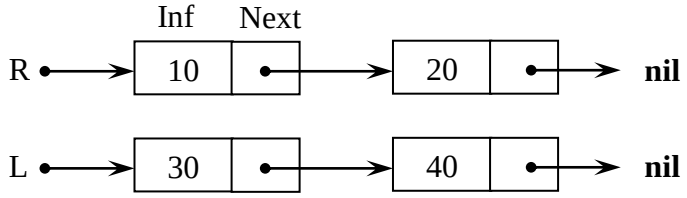


Рис.6.

Рассмотрим действие различных операторов на состояние этих списков.

а) L := R (рис.7).

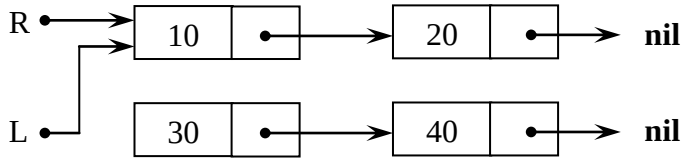


Рис.7.

б) L^ := R^ (рис.8).

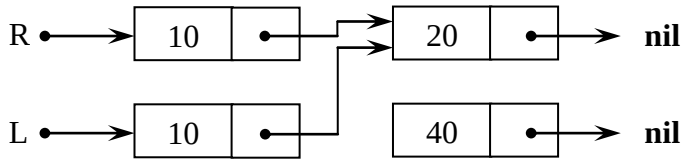


Рис.8.

в) L^.Inf := R^.Inf (рис.9).

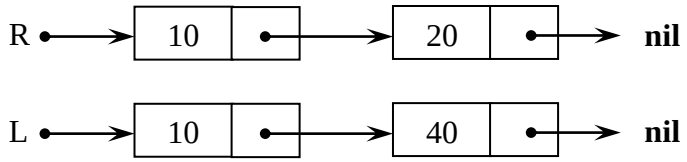


Рис.9.

г) L^.Next := R^.Next (рис.10).

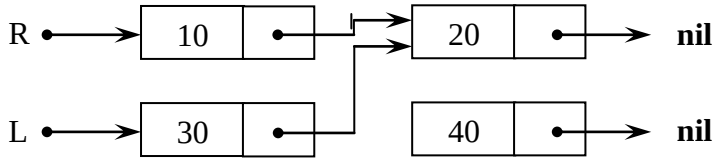


Рис.10

д) L := **nil** (рис.11).

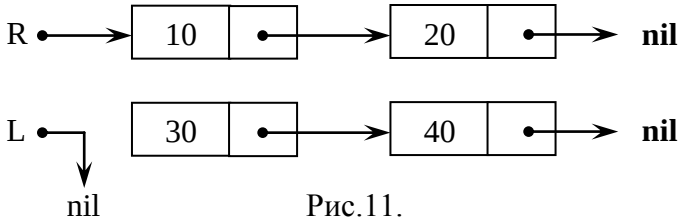


Рис.11.

При изменении указателей в первоначально связанных структурах могут оставаться такие элементы, доступ к которым уже невозможен, так как их адрес потерян при изменении значений соответствующих указателей (а: элементы 30,40; б: элемент 40; г: элемент 40; д: элементы 30,40). Но эти переменные продолжают занимать память, хотя они уже и не нужны. Неиспользуемые динамические переменные образуют "мусор", который загромождает память, и при своем накоплении может сделать невозможным дальнейшее функционирование программы.

Для предотвращения накопления мусора необходимо сведения о динамических переменных, которые становятся ненужными, возвращать в таблицу свободной памяти. Эта работа выполняется предописанной процедурой *Dispose*(*L*), которая освобождает память, затребованную процедурой *New*(*L*) для переменной *L*^, передавая об этом сведения в таблицу свободной памяти. Содержимое поля *L*^ при этом не изменяется, но может быть использовано для размещения другой динамической переменной.

Обычная схема использования динамической переменной имеет вид:

```
New(L);
........
Обработка L^
........
Dispose(L);
L:=nil;
```

Процедура *Dispose* не изменяет указатель *L*. Поэтому после освобождения памяти, адресуемой указателем *L*, ему целесообразно присвоить пустое значение ***nil.***

Если после процедуры *Dispose*(*L*) записать еще раз такую же процедуру, т.е. сделать попытку освободить уже освобожденную память, то это вызовет аварийное завершение программы. То же самое произойдет, если в приведенной выше схеме после оператора `L:=nil` записать процедуру *Dispose*(*L*).

Для выделения динамической памяти может использоваться также процедура

`GetMem(P:pointer; Size:word),`

которая создает новую динамическую переменную *P*^ размером *Size*. В этом случае освобождение динамической памяти должно выполняться процедурой

`FreeMem(P:pointer; Size:word).`

Отличия между процедурами *New, Dispose* и *GetMem, FreeMem*:

- в процедурах *GetMem, FreeMem* указатель *P* может быть любого типа, в процедурах *New, Dispose* это может быть только типизированный указатель;

- процедура *GetMem* (*FreeMem*) может выделять (удалять) любой заданный объем памяти (но не более 64 Кбайта), процедура *New* (*Dispose*) - только тот объем памяти, который определен описанием типизированного указателя.

Пример 2.

```
Type  PoinType = ^DynType;
      DynType  = record
         Inf  : integer;
         Next : PoinType;
      end;
Var  P : PoinType;
Begin
..............
  New(P);
```

```
..............
  Dispose(P);
..............
  GetMem(P,SizeOf(DynType));
.......................
  FreeMem(P,SizeOf(DynType));
.......................
```

Здесь процедуры *New*, *Dispose* и *GetMem*, *FreeMem* действуют совершенно одинаково.

Типизированному указателю, как и нетипизированному, можно присвоить адрес конкретного поля памяти.

Пример 3.

```
Type  ByteAr = array[1..2] of byte;
      ByteArPtr : ^ByteAr;
Var   k : integer;
      p : ByteArPtr;
Begin
  p:=@k;
```

Следовательно, динамическая переменная *p^* и статическая переменная *k* определяют одно и то же поле памяти. Используя в программе имя *k*, мы можем обрабатывать это поле как переменную типа *integer*; при использовании имени *p^* это же поле рассматривается как массив из двух элементов типа *byte*. Нетрудно заметить сходство примененной здесь методики с аппаратом абсолютных переменных.

ДИНАМИЧЕСКИЕ МАССИВЫ

Все глобальные переменные, описанные в программе, размещаются в области памяти, которую называют сегментом данных, Следовательно, их общий объем не может превышать 64 Кбайт. В то же время процедура *New* может выделять для каждой динамической переменной до 64 Кбайт памяти.

Предположим, что в программе содержится объявление

```
Type  Ar = array[1..5000] of real.
Var   X,Y,Z : Ar.
```

Так как каждое число типа *real* занимает 6 байт памяти, то для массивов *X, Y, Z* требуется совместно 90 Кбайт, что невозможно.

Решение этой задачи можно получить, если массивы *X, Y, Z* объявить динамическими. Это можно сделать, например, следующим образом:

```
Type  Ar = array[1..5000] of real;
      ArPtr = ^Ar;
Var   X,Y,Z : ArPtr;
```

Здесь *X, Y, Z* – по-прежнему статические переменные, но для каждой из них в сегменте данных выделяется лишь по 4 байта памяти.

В программе после выполнения процедур

```
New(X); New(Y); New(Z);
```

вместо операторов вида

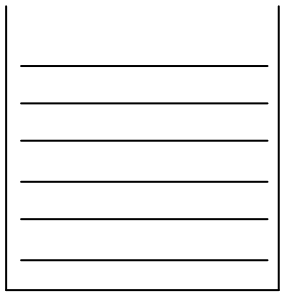
```
Z[i] := X[i] + Y[i]
```

будут записываться операторы

```
Z^[i] := X^[i] + Y^[i].
```

ОБРАБОТКА СТЕКА

Стек - это линейный список, в котором все включения и исключения элементов делаются на одном конце списка. Принцип организации стека: "Последний пришел - первый вышел", что сокращенно обозначают также LIFO (‘Last In – First Out’). Стеки довольно широко применяются в программировании.



Стек можно представить в виде коробки, в которую складываются листы картона, равные размеру коробки (рис.12). Вложив в эту коробку *n* листов, мы не можем извлечь первый лист до тех пор, пока не будут извлечены предыдущие *n* - 1 листов.

Связь между элементами стека показана на рис.13. Здесь *Beg* - указатель начального элемента стека (указатель входа в стек).

Рис.12.

Рис.13.

В дальнейшем будем считать, что в каждом фрагменте программы, иллюстрирующей методы формирования и обработки стека, действует описание

```
Type  PoinType = ^DynType;
      DynType = record
          Inf : integer;
        Next : PoinType;
      end;
Var  Beg,                { указатель входа в стек }
     Run : PoinType;     { текущий указатель }
     k : integer;
     F : text;
```

1. Формирование стека из текстового файла.

```
Assign(F,'F.dat'); Reset(F);
Beg:=nil;
While not SeekEof(F) do
  Begin
    Read(F,k); New(Run);
    Run^.Inf:=k;
    Run^.Next:=Beg;
    Beg:=Run;
  End;
Close(F);
```

Здесь, в отличие от рассматриваемой далее очереди, не имеет значения номер элемента, прочитанного из файла (первый или не первый). Если исходный файл пустой, то стек также будет пустой, о чем свидетельствует значение *Beg=**nil.***

2. Добавление нового элемента в стек.

```
Read(k); New(Run);
```

```
Run^.Inf:=k;
Run^.Next:=Beg;
Beg:=Run;
```

Фактически это однократное повторение работы предыдущего цикла.

3. Удаление элемента из стека.

```
Run:=Beg; Beg:=Beg^.Next;
Dispose(Run);
```

В указателе Run временно запоминается адрес удаляемого в дальнейшем элемента стека.

4. Просмотр элементов стека.

```
Run:=Beg; k:=0;
While Run<>nil do
  Begin
    Inc(k);
    Writeln('k= ',k,'  Inf= ',Run^.Inf);
    Run:=Run^.Next;
  End;
```

5. Удаление стека.

```
While Beg<>nil do
  Begin
    Run:=Beg; Beg:=Beg^.Next;
    Dispose(Run);
  End;
```

6. Реверсирование стека.

При формировании стека из текстового файла информационные элементы стека будут расположены в обратном порядке по сравнению с их расположением в файле. Это может создавать определенные неудобства при обработке стека в программе. В связи с этим возникает задача реверсирования элементов стека.

По отношению к обычному массиву задача реверсирования стека аналогична перестановке элементов массива в обратном порядке. Существенная разница между ними заключается в том, что положение информационных элементов стека в памяти не должно изменяться, изменению подвергаются лишь связи между элементами, т.е. значения указателей.

Выполним два варианта реверсирования стека.

Вариант 1 :

```
Var  BegBuf,RunBuf : PoinType;
Begin
  BegBuf:=nil; Run:=Beg;
  If (Run<>nil) and (Run^.Next<>nil) then { в стеке }
    Begin                        { больше одного эл-та }
      While Run<>nil do          { перепись стека Beg }
        Begin                    { в стек BegBuf      }
          k:=Run^.Inf; New(RunBuf);
          RunBuf^.Inf:=k;
          RunBuf^.Next:=BegBuf;
```

```
        BegBuf:=RunBuf;
        Run:=Run^.Next;
      End;
    While Beg<>nil do          { удаление стека Beg }
      Begin
        Run:=Beg; Beg:=Beg^.Next;
        Dispose(Run);
      End;
    Beg:=BegBuf; BegBuf:=nil;      { переадресация }
End;                               { стека         }
```

В варианте 1 исходный стек переписывается в буферный с начальным указателем *BegBuf,* при этом элементы исходного стека будут расположены в обратном порядке. После этого стек *Beg* удаляется, указателю *Beg* присваивается адрес входа в новый стек *BegBuf*, а буферный указатель принимает пустое значение. Если исходный стек пустой или в нем находится только один элемент, реверсирование не производится.

В программе варианта 1 следует обратить внимание на следующую деталь.

Если стек пустой, т.е. *Beg* = ***nil***, то указатель *Beg^.Next* не существует. В этом случае запись условного оператора в виде

```
If (Run^.Next<>nil) and (Run<>nil) then
```

может привести к неправильной работе программы или к ее зацикливанию.

Вычисление логического выражения в Паскаль-программе производится по так называемой короткой схеме. Это означает, что вычисление выражения прекращается, как только становится очевидным его результат. В частности,

- если несколько операндов связаны между собой операцией логического умножения, то при получении значения *false* для одного из операндов, вычисляемых слева направо, остальные операнды не рассматриваются;

- если несколько операндов связаны между собой операцией логического сложения, то при получении значения *true* для одного из операндов, вычисляемых слева направо, остальные операнды не рассматриваются.

Поэтому в операторе

```
If (Run<>nil) and (Run^.Next<>nil) then
```

при обнаружении (*Run* <> ***nil***) = *false* значение второго операнда не вычисляется.

Недостатки варианта 1 :

- в программе используется буферный стек, что требует выделения дополнительной памяти;

- при формировании буферного стека производится пересылка информационных элементов из исходного стека, что приводит к дополнительным затратам машинного времени (информационные элементы могут быть большого размера).

Вариант 2 :

```
Var  P : PoinType;
Begin
  If Beg<>nil then
    Begin
      Run:=Beg; Beg:=Beg^.Next;
      Run^.Next:=nil;
      While Beg<>nil do
        Begin
          P:=Beg; Beg:=Beg^.Next;
          P^.Next:=Run; Run:=P;
        End;
```

```
      Beg:=Run;
    End;
```

В программе варианта 2 изменяются лишь связи между элементами стека. Чтобы понять работу этой программы, рекомендуется нарисовать карандашом стек, состоящий из трех или четырех элементов, и "проиграть" на нем работу программы реверса, изменяя связи между элементами стека после выполнения каждого оператора программы.

Примечание. Если в программе заранее известно, что последовательность элементов в линейном списке должна быть такой же, как и в исходном файле, то в этом случае целесообразно вместо стека использовать очередь.

7. Определение значения и местоположения максимального элемента в стеке.

Поиск максимального элемента в стеке производится аналогично поиску в массиве, но вместо индекса элемента массива здесь местоположение элемента определяется значением его указателя. Если стек пустой, то указатель максимального элемента должен иметь значение ***nil.***

```
Program MaxElem;
Type  PoinType = ^DynType;
      DynType = record
         Inf : integer;
        Next : PoinType;
      end;
Var  Beg,                  { указатель входа в стек }
     Run,                  { текущий указатель }
     Pmax : PoinType;      { указатель макс.элемента }
     kmax : integer;       { значение макс.элемента }
Begin
  Формирование  стека
  Pmax:=Beg;
  If Beg<>nil then
    Begin
      kmax:=Beg^.Inf; Run:=Beg^.Next;
      While Run<>nil do          { в цикле просматрива- }
        Begin                    { ются элементы стека, }
          If Run^.Inf>kmax then  { начиная со второго   }
            Begin
              kmax:=Run^.Inf; Pmax:=Run
            End;
          Run:=Run^.Next;
        End;
    End;
  Печать  kmax, Pmax^.Inf
End.
```

8. Удаление из стека максимального элемента.

В программе рассматриваются две ситуации, определяющие местоположение максимального элемента:

а) максимальный элемент - это первый элемент стека (рис.14);

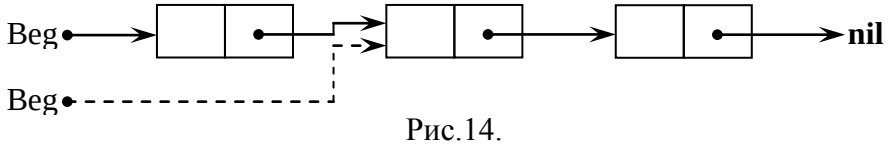


Рис.14.

б) максимальный элемент - это любой элемент стека, кроме первого (на рис.15 - это третий элемент).

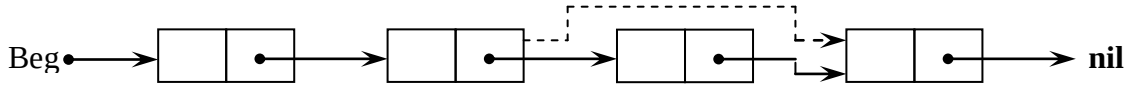


Рис.15.

Из рис.15 можно сделать вывод, что для удаления из стека заданного элемента необходимо знать адрес предшествующего элемента. Поэтому в отличие от предыдущего примера в программе дополнительно используется указатель *Pred,* определяющий адрес элемента, расположенного в стеке перед максимальным элементом. Если стек не пустой и *Pmax=Pred*, то максимальным элементом является первый элемент стека, в противном случае между ними существует следующее соотношение: *Pmax = Pred^.Next.*

```
Program DelMaxElem1;
Type  PoinType = ^DynType;
      DynType = record
         Inf : integer;
        Next : PoinType;
      end;
Var  Beg,             { указатель входа в стек }
     Run,             { текущий указатель }
     Pmax,            { указатель макс.элемента }
     Pred : PoinType; { указатель эл-та, предшествующего Pmax}
     kmax : integer;  { значение макс.элемента }
Begin
  Формирование  стека
  Pmax:=Beg;
  If Beg<>nil then
    Begin
      kmax:=Beg^.Inf;
      Pred:=Beg; Run:=Beg;
      While Run^.Next<>nil do
        Begin
          If Run^.Next^.Inf>kmax then
            Begin
              kmax:=Run^.Next^.Inf;
              Pred:=Run; Pmax:=Run^.Next;
            End;
          Run:=Run^.Next;
        End;
      If Pmax=Pred then
        Beg:=Beg^.Next
      Else
        Pred^.Next:=Pmax^.Next;
      Dispose(Pmax);
    End;
  Печать стека
End.
```

В программе DelMaxElem1 следует обратить внимание на следующую деталь.

Указателю предшествующего элемента стека назначено имя *Pred,* совпадающее с именем предописанной функции, которая определяет предыдущее значение ординальной переменной. Это допускается в Паскаль-программе, поскольку имя *pred* считается предописанным, а не зарезервированным. Однако если нам потребуется в программе записать

kmax:=*pred*(*kmax*), то при компиляции будет выдано сообщение "Error 26: Type mismatch" (несоответствие типов). Тем не менее в этой программе функцию *pred* все же можно использовать, но это должно быть записано в виде *kmax* := *System.pred*(*kmax*), где *System* - имя стандартного модуля, содержащего функцию *pred.*

Указатели *Pred* и *Pmax* жестко связаны друг с другом, а именно *Pmax = Pred^.Next*. Поэтому в приведенной ниже программе DelMaxElem2 вместо двух указателей *Pred* и *Pmax* используется один указатель *Pmax,* но он адресует здесь элемент, предшествующий максимальному. Поскольку для первого элемента стека не существует предшествующего элемента, то *Pmax = Beg* как в случае, когда максимальным элементом стека является первый элемент, так и в случае, когда таким является второй элемент. В связи с этим в качестве признака удаления первого элемента в программе используется логическое выражение `Pmax^.Inf=Beg^.Inf` (если максимальный элемент равен первому элементу стека, то удалению подлежит первый элемент).

```
Program DelMaxElem2;
Type  PoinType = ^DynType;
      DynType = record
          Inf : integer;
         Next : PoinType;
      end;
Var  Beg,              { указатель входа в стек }
     Run,              { текущий указатель }
     Pmax : PoinType;  { указатель предшествующего элемента }
     kmax : integer;   { значение макс.элемента }
Begin
  Формирование  стека
  If Beg<>nil then     { стек не пустой }
    Begin
      kmax:=Beg^.Inf;
      Pmax:=Beg; Run:=Beg;
      While Run^.Next<>nil do
        Begin
          If Run^.Next^.Inf>kmax then
            Begin
              kmax:=Run^.Next^.Inf;
              Pmax:=Run;
            End;
          Run:=Run^.Next;
        End;
      If kmax=Beg^.Inf then
        Begin
          Run:=Beg; Beg:=Beg^.Next
        End
      Else
        Begin
          Run:=Pmax^.Next; Pmax^.Next:=Pmax^.Next^.Next;
        End;
      Dispose(Run);
    End;
  Печать стека
End.
```

9. Добавление элемента в упорядоченный стек.

Будем считать, что элементы стека расположены по возрастанию их численных значений.

Возможны два варианта включения нового элемента:
а) включение в начало стека (рис.16);

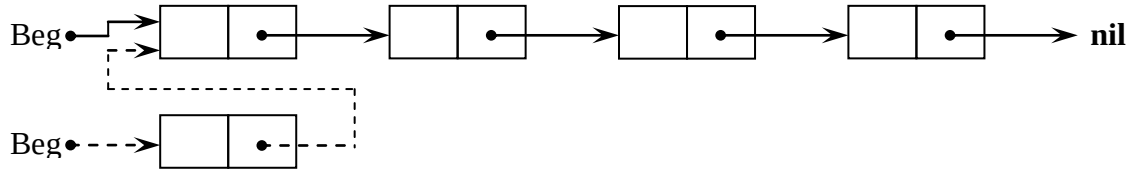


Рис.16.

б) включение в произвольное место стека (рис.17).

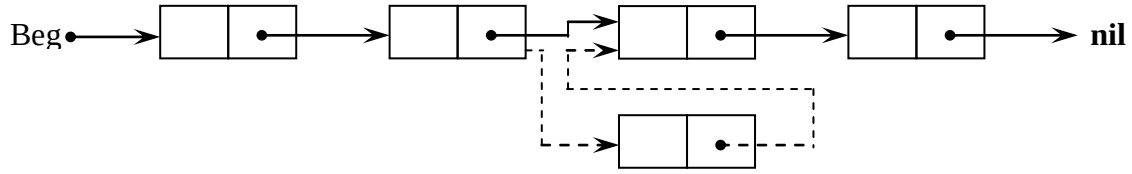


Рис.17.

Во втором случае, как и в программе добавления нового элемента в упорядоченный массив, производим последовательный просмотр элементов стека до обнаружения элемента, большего по значению, чем добавляемый элемент.

Как следует из анализа рис.17, для изменения связей в стеке при добавлении элемента требуется знать адрес предыдущего элемента.

```
Program InsElem1;
Label 10;
Type  PoinType = ^DynType;
      DynType = record
          Inf : integer;
         Next : PoinType;
      end;
Var  Beg,Run,
     Pred,              { указатель эл-та, предшествующего }
                        { включаемому новому элементу }
     NewEl : PoinType; { указатель нового элемента }
     k : integer;
Begin
  Формирование  стека
  Read(k); New(NewEl); NewEl^.Inf:=k;
  If (Beg=nil) or (k<=Beg^.Inf) then
    Begin
      NewEl^.Next:=Beg;    { включение нового эл-та }
      Beg:=NewEl;          { в начало стека         }
    End
  Else
    Begin
      Pred:=Beg; Run:=Beg^.Next;
      While Run<>nil do
        If Run^.Inf > k then
          Begin
            NewEl^.Next:=Run;  { включение нового эл-та }
            Pred^.Next:=NewEl; { в середину стека       }
            Goto 10
          End
```

```
        Else
          Begin
            Pred:=Run;          { перебор элементов }
            Run:=Run^.Next      { стека             }
          End;
        Pred^.Next:=NewEl;      { включение нового эл-та }
        NewEl^.Next:=nil;       { в конец стека          }
    End;
  10:
  Печать стека
End.
```

Если при просмотре стека не обнаружен элемент, превышающий значение *k*, то это означает, что значение *k* должно быть записано после последнего элемента стека.

Как и в предыдущем примере, программу InsElem1 можно реализовать без указателя *Pred*, возложив его функции на указатель *Run*.

```
Program InsElem2;
Type  PoinType = ^DynType;
      DynType = record
          Inf : integer;
         Next : PoinType;
      end;
Var  Beg,Run,
     NewEl : PoinType; { указатель нового элемента }
     k : integer;
     Cond : boolean;
Begin
  Формирование  стека
  Read(k);
  New(NewEl); NewEl^.Inf:=k;
  If (Beg=nil) or (k<=Beg^.Inf) then
    Begin
      NewEl^.Next:=Beg;    { включение нового эл-та }
      Beg:=NewEl;          { в начало стека         }
    End
  Else
    Begin
      Run:=Beg; Cond:=true;
      While (Run^.Next<>nil) and Cond do
        If Run^.Next^.Inf > k then
          Begin
            NewEl^.Next:=Run^.Next; { включение нового эл-та }
            Run^.Next:=NewEl;       { в середину стека       }
            Cond:=false
          End
        Else
          Run:=Run^.Next;
      If Cond then
        Begin
          Run^.Next:=NewEl;     { включение нового эл-та }
          NewEl^.Next:=nil;     { в конец стека          }
        End;
    End;
  Печать стека
End.
```

Здесь следует обратить также внимание, что за счет небольшой модификации программы в InsElem2 исключен оператор ***Goto.***

ОБРАБОТКА ОЧЕРЕДИ

Очередь - это линейный список, в котором все включения производятся на одном конце списка, а исключения делаются на другом конце. Принцип организации очереди: "Первый пришел - первый вышел", что сокращенно обозначают также FIFO (‘First in – first out’).

Указатели *R* и *L* (рис.18) определяют правый (Right) и левый (Left) концы очереди. Указатели *Next*, входящие в состав каждого элемента, определяют адрес следующего справа элемента в очереди. Следовательно, с левого конца можно и добавить, и удалить элемент (как в стеке). Указатель *Next* крайнего правого элемента пустой. Этому элементу неизвестно, где находится предыдущий элемент. Поэтому справа можно только добавлять новые элементы. Следовательно, в очереди добавление элемента должно производиться справа, а удаление - слева.

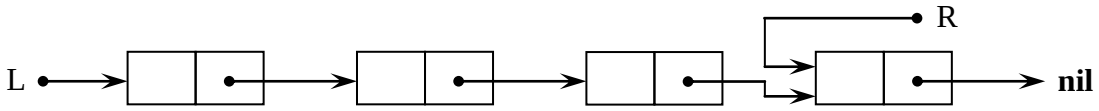


Рис.18.

В дальнейшем будем считать, что в каждом фрагменте программы, иллюстрирующей методы формирования и обработки очереди, действует следующее описание:

```
Type  PoinType = ^DynType;
      DynType = record
          Inf : integer;
         Next : PoinType;
      end;
Var   L,                { указатели левого и правого }
      R,                { концов очереди             }
      Run : PoinType;   { текущий указатель          }
      k : integer;
      F : text;
```

1. Формирование очереди из текстового файла.

В программе формирования очереди учтены два частных случая:

а) исходный файл пустой; тогда и очередь должна быть пустой, о чем свидетельствует значение *L* = ***nil;***

б) в файле содержится лишь одно число; тогда адрес единственного элемента очереди определяют оба указателя *L* и *R*.

Будем считать, что адрес очередного элемента очереди определяет текущий указатель Run, и рассмотрим два этапа формирования очереди:

1) из файла прочитан первый элемент, включаемый в очередь (возможно, этот элемент останется единственным в очереди);
2) из файла прочитан второй элемент.

Признаком первого этапа является равенство *L* = ***nil***. На этом этапе должны быть выполнены следующие действия:

```
Run^.Next:=nil;   L:=Run;   R:=Run
```

Значение L<>**nil** является признаком чтения из файла следующего элемента очереди. В этом случае должно быть выполнено:

```
Run^.Next:=nil;  R^.Next:=Run;  R:=Run
```

Тогда программа формирования очереди будет иметь следующий вид:

```
L:=nil; R:=nil;
While not SeekEof(F) do
  Begin
    Read(F,k);
    New(Run);
    Run^.Inf:=k;
    If L=nil then
      Begin
        Run^.Next:=nil;
        L:=Run; R:=Run
      End
    Else
      Begin
        Run^.Next:=nil;
        R^.Next:=Run;
        R:=Run;
      End;
End;
```

В альтернативах условного оператора имеются одинаковые операторы. Это позволяет записать программу формирования очереди в более компактном виде:

```
L:=nil; R:=nil;
While not SeekEof(F) do
  Begin
    Read(F,k);
    New(Run); Run^.Inf:=k;
    Run^.Next:=nil;
    If L=nil then
      L:=Run
    Else
      R^.Next:=Run;
    R:=Run;
  End;
```

Примечание. Элементы очереди, в отличие от стека, располагаются в том же порядке, что и в файле. Если же почему-либо требуется, чтобы относительный порядок элементов в очереди был обратным, то добавление новых элементов нужно производить слева, а не справа (как в стеке).

2. Добавление нового элемента в очередь.

```
Read(k);
New(Run); Run^.Inf:=k;
Run^.Next:=nil;
If L=nil then
  L:=Run
Else
  R^.Next:=Run;
R:=Run;
```

3. Удаление элемента из очереди.

```
Run:=L; L:=L^.Next;
Dispose(Run);
```

4. Просмотр и удаление очереди выполняются так же, как и для стека.

5. Формирование циклической очереди (рис.19).

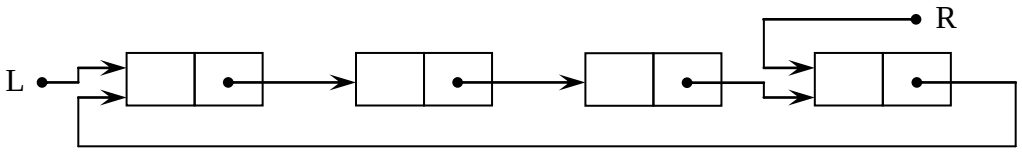


Рис.19.

Для получения такой очереди достаточно установить

```
R^.Next:= L .
```

Для входа в циклическую очередь можно использовать любой из указателей *R* или *L*.

6. Удаление произвольного элемента из очереди.

Предположим, что нам требуется удалить из очереди минимальный элемент. Тогда в частном случае может быть удален один из крайних или промежуточный элемент. При удалении крайних элементов должен быть обеспечен соответствующий перенос указателя *L* или указателя *R*. Как и в стеке, для удаления заданного элемента из очереди необходимо знать адрес предыдущего элемента.

```
Program DelMinElem;
Type  PoinType = ^DynType;
      DynType = record
          Inf : integer;
         Next : PoinType;
      end;
Var  L,                 { указатели левого и правого }
     R,                 { концов очереди             }
     Run,               { текущий указатель          }
     Pmin : PoinType;   { указатель элемента, предшествующего }
                        { минимальному элементу очереди }
     InfMin : integer;  { значение минимального элемента }
     F : text;
Begin
  Формирование  очереди
  If L<>nil then
    Begin
      InfMin:=L^.Inf; Pmin:=L;
      Run:=L;
      While Run^.Next<>nil do
        Begin
          If Run^.Next^.Inf<InfMin then
            Begin
              InfMin:=Run^.Next^.Inf; Pmin:=Run;
            End;
          Run:=Run^.Next;
        End;
      If InfMin=L^.Inf then    { удаление первого элемента }
        Begin                  { с переносом указателя L   }
          L:=L^.Next; Dispose(Pmin);
        End
      Else
        Begin
```

```
          If Pmin^.Next=R then  { перенос R при удалении }
            R:=Pmin;            { последнего элемента    }
          Run:=Pmin^.Next;      { удаление произвольного}
          Pmin^.Next:=Pmin^.Next^.Next;{ элемента (в том }
          Dispose(Run);         { числе последнего)    }
        End;
    End;
  Печать очереди
End.
```

Если минимальным является первый или второй элементы очереди, то в обоих случаях *Pmin = L*. Поэтому для указания о том, является ли первый элемент минимальным, в программе производится проверка *InfMin = L^.Inf*.

7. Добавление нового элемента в произвольное место очереди.

Предположим, что элементы очереди упорядочены по убыванию; требуется добавить в очередь целочисленное значение *b* без нарушения ее упорядоченности.

Вполне очевидно, что в частном случае элемент *b* может быть добавлен слева, справа или в середину очереди.

```
Program InsNewElem;
Label  10;
Type   PoinType = ^DynType;
       DynType = record
           Inf : integer;
         Next : PoinType;
       end;
Var  L,R,               { левый и правый указатели   }
     Run,               { текущий указатель          }
     NewEl : PoinType;  { указатель нового элемента  }
     b : integer;       { значение нового элемента }
     Cond : boolean;
Begin
  Формирование  очереди
  Read(b);
  New(NewEl); NewEl^.Inf:=b;
  If L=nil then               { очередь пустая }
    Begin
      L:=NewEl; R:=NewEl; NewEl^.Next:=nil;
    End
  Else
    If b>=L^.Inf then         { добавление слева }
      Begin
        NewEl^.Next:=L; L:=NewEl
      End
    Else
      If b<=R^.Inf then       { добавление справа }
        Begin
          NewEl^.Next:=nil; R^.Next:=NewEl; R:=NewEl
        End
      Else
        Begin                 { добавление в середину }
          Run:=L;
          While Run^.Next<>nil do
            Begin
              If Run^.Next^.Inf<b then
                Begin
```

```
                    NewEl^.Next:=Run^.Next; Run^.Next:=NewEl;
                    Goto 10
                  End;
                Run:=Run^.Next;
              End;
          End;
  10:
  Печать очереди
End.
```

ОБРАБОТКА ДЕКА

Дек - это двунаправленный список, в котором включение и исключение элементов делаются на обоих концах списка (рис.20).

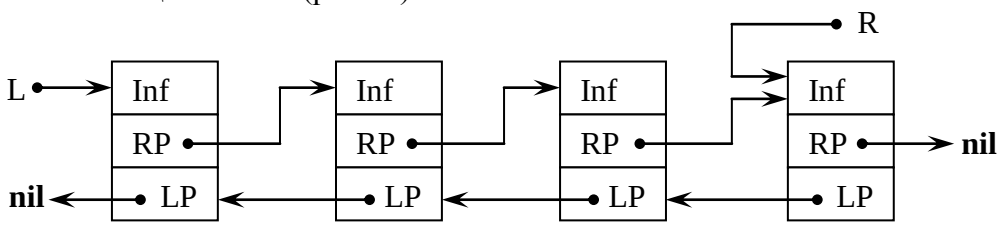


Рис.20.

1. Формирование дека.

Программа формирования дека написана в предположении, что добавление в дек новых элементов производится справа, как и при формировании очереди. В программе учтен случай, когда дек состоит из одного элемента (рис.21).

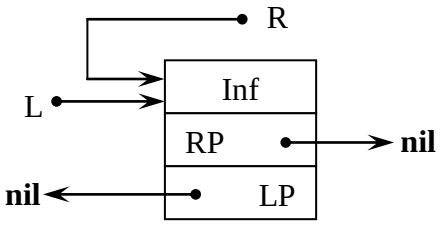


Рис.21.

```
Type  PoinType = ^DekType;
      DekType = record
          Inf : integer;
          LP, RP : PoinType;
      end;
Var  L,R,Run : PoinType;
     k : integer;
     F : text;
Begin
.......................
L:=nil; R:=nil;
While not SeekEof(F) do
  Begin
    Read(F,k); New(Run);
    Run^.Inf:=k; Run^.RP:=nil;
    If L=nil then
      Begin
        Run^.LP:=nil; L:=Run;
```

```
      End
    Else
      Begin
        R^.RP:=Run; Run^.LP:=R
      End;
    R:=Run;
  End;
```

Значение L = ***nil*** означает, что из файла вводится первый элемент, включаемый в дек (возможно, он останется единственным). Методика формирования дека в основном аналогична формированию очереди, но с учетом того, что элемент дека имеет два указателя.

2. Просмотр дека возможен как слева направо, так и справа налево и выполняется аналогично просмотру очереди.

3. Добавление и удаление элементов дека по определению возможны на обоих его концах и выполняются в основном аналогично соответствующим операциям для стека и очереди, но программная реализация здесь проще из-за того, что каждому элементу дека известны адреса его соседей слева и справа.

МЕТОД БЫСТРОЙ СОРТИРОВКИ

Рассмотренные ранее методы прямой выборки и прямого обмена относят к так называемым методам сортировки первого уровня. Существуют также методы второго уровня. Их алгоритмы сложнее, но при этом обеспечивается на один-два порядка более высокое быстродействие программы. К методам второго уровня относятся метод Шелла, метод быстрой сортировки, метод слияния и др. Наиболее эффективным из них является метод быстрой сортировки. Как будет показано ниже, программная реализация этого метода требует использования линейных списков, в данном случае стеков.

Метод быстрой сортировки (метод разделения, метод Хоара), представляет собой усовершенствование метода прямого обмена. При прямом обмене перестановкам подвергаются смежные элементы; в методе Хоара сначала производятся перестановки на больших расстояниях, а затем эти расстояния постепенно уменьшаются.

Алгоритм перестановок по методу Хоара сводится к следующему.

1. Для массива $X = (x_1, x_2, ..., x_n)$ определяется значение срединного элемента $a = x_{n/2}$.

2. $i := 1$; $j := n$.

3. Пока $x_i < a$, выполнять $i := i + 1$.

   Пока $x_j > a$, выполнять $j := j - 1$.

4. Если $i < j$, обменять элементы x_i и x_j.

5. Если $i \le j$, то $i := i+1$, $j := j-1$.

6. Шаги 3 .. 5 повторять, пока не будет выполнено отношение $i > j$.

Работу алгоритма перестановок рассмотрим на примере конкретного массива X:

64 23 18 71 44 15 27 34 55 41 ($n = 10$).

1. $a = x_5 = 44$.

2. $i = 1$; $j = 10$.

3. $i = 1;\ j = 10.$ { $x_1 > a, x_{10} < a;$ *поэтому переменные i и j не изменяются* }

4. Обмен x_1 и x_{10}:

41 23 18 71 44 15 27 34 55 64

5. $i = 2;\ j = 9.$ { $i < j$, *поэтому* $i:=i+1, j:=j-1$ }

6. Переход к шагу 3.

3. $i = 4;\ j = 8.$ { $x_2 < a \Rightarrow i$ *увеличивается*; $x_9 > a \Rightarrow j$ *уменьшается* }

4. Обмен x_4 и x_8:

41 23 18 34 44 15 27 71 55 64

5. $i = 5;\ j = 7.$ { $i < j \Rightarrow i$ *увеличивается*, j *уменьшается* }

6. Переход к шагу 3.

3. $i = 5;\ j = 7.$ { $x_5 = a, x_7 < a \Rightarrow i$ *и* j *не изменяются* }

4. Обмен x_5 и x_7:

41 23 18 34 27 15 44 71 55 64

5. $i = 6;$ j = 6. { $i < j \Rightarrow i$ *увеличивается*, j *уменьшается* }

6. Переход к шагу 3.

3. $i = 7;\ j = 6.$ { $i > j$ }

4. -

5. -

6. Выход.

После окончания работы алгоритма перестановок в массиве *X* слева от элемента со значением $a = 44$ расположены элементы $x_i < a$, справа - элементы $x_i > a$. Переменные *i* и *j* на конечном этапе работы алгоритма перестановок разделили массив *X* на две части: подмассив 1 .. *j* (1 .. 6) и подмассив *i* .. *n* (7 .. 10). Значения границ правого подмассива *i..n* запоминаются в стеке, подмассив 1 .. *j* подвергается обработке по рассмотренной выше схеме.

После перестановки элементов подмассива 1 .. *j* переменные *i* и *j* в свою очередь разделяют его на две части, границы правой части записываются в стек, левая часть как новый подмассив обрабатывается по той же схеме. Когда длина левой части станет равной 1, из стека выбираются границы необработанных подмассивов. Сортировка в целом заканчивается, когда стек границ подмассивов становится пустым.

```
Program RapidSorting;
Const  Nmax = 1000;
Type   Ar = array[1..Nmax] of real;
       PoinType = ^Stack;
       Stack = record
          L,R : word;  { левая и правая границы подмассива }
          Next : PoinType;
       end;
Var    i,j,n,Left,Right : word;
       X : Ar;
       a,Buf : real;
       Beg,Run : PoinType;
Begin
  В в о д   n, X
  New(Beg);
  Beg^.L:=1; Beg^.R:=n;
  Beg^.Next:=nil;
```

```
  Repeat
    Left:=Beg^.L; Right:=Beg^.R;
    Run:=Beg; Beg:=Beg^.Next;
    Dispose(Run);
    Repeat
      i:=Left; j:=Right;
      a:=x[(Left+Right) div 2];
      Repeat
        While x[i]<a do
          Inc(i);
        While x[j]>a do
          Dec(j);
        If i<=j then
          Begin
            If i<j then
              Begin
                Buf:=x[i]; x[i]:=x[j]; x[j]:=Buf;
              End;
            Inc(i); Dec(j);
          End;
      Until i>j;
      If i<Right then
        Begin
          New(Run);
          Run^.l:=i; Run^.R:=Right;
          Run^.Next:=Beg; Beg:=Run;
        End;
      Right:=j;
    Until Left>=Right;
  Until Beg=nil;
  В ы в о д   X
End.
```

РЕКУРСИВНЫЕ ПРОЦЕДУРЫ И ФУНКЦИИ

Рекуррентное отношение в общем случае имеет вид

$$a_{n+1} = f(n, a_n, a_{n-1}, ..., a_1, a_0) \ .$$

Здесь очередной член a_{n+1} числовой последовательности вычисляется по его номеру n и значениям предыдущих членов $a_0, ..., a_n$ этой последовательности.

Например, для вычисления ряда Фибоначчи используется рекуррентное отношение

$$F_{n+1} = F_n + F_{n-1}; \quad F_0 = 0; \quad F_1 = 1 \ .$$

Тогда имеем

$$F_0 = 1; \quad F_1 = 1; \quad F_2 = 2; \quad F_3 = 3; \quad F_4 = 5; \quad F_6 = 8; \quad F_7 = 13; \quad F_8 = 21; \ldots$$

Наиболее часто рекуррентное отношение имеет вид

$$a_{n+1} = f(n, a_n) \ .$$

Рекурсивную функцию можно считать обобщением понятия рекуррентного отношения. В такой функции вычисление заданного значения производится с помощью этой же функции ("рекурсия" означает "возвращение"):

$$f(x) = F(f(x)) \ .$$

Классическим примером рекурсивной функции является факториал

$$f(n) = n!\ .$$

Рекурсивный способ записи факториала:

$$n! = \begin{cases} 1 & при\ n = 0 \\ n(n-1)! & при\ n > 0 \end{cases}$$

Паскаль-программа допускает рекурсивное описание функций. В этом случае в теле такой подпрограммы-функции содержится обращение к этой же функции.

Пример 1. Для факториала имеем:

```
Function Fact(n:word):word;
Begin
  If n=0 then
    Fact:=1
  Else
    Fact:=n*Fact(n-1);
End { Fact };
```

Следует обратить внимание, что в заголовке функции `Fact` формальный параметр *n* – это параметр-значение (перед его именем не стоит слово **Var**). Следовательно, значение переменной *n* изменяется лишь в функции `Fact` и никак не влияет на значение соответствующего фактического параметра в вызывающей программе.

В процессе выполнения рекурсивной функции ее параметры-значения складываются в стек. Происходит развертывание, а затем свертывание рекурсивных вызовов.

Пусть мы имеем $y = 5!$, т.е. $y := Fact(5)$. Схема вычислительных действий:

Fact(5) = 5 * Fact(4)	5 * 24 = 120
Fact(4) = 4 * Fact(3)	4 * 6 = 24
Fact(3) = 3 * Fact(2)	3 * 2 = 6
Fact(2) = 2 * Fact(1)	2 * 1 = 2
Fact(1) = 1 * Fact(0)	1 * 1 = 1
Fact(0) = 1	1

Любое рекурсивное определение функции можно заменить нерекурсивным.

Пример для факториала:

```
Function Fact(n:word):word;
Var i,k : word;
Begin
  k:=1;
  For i:=1 to n do
    k:=k*i;
  Fact:=k;
End { Fact };
```

Рекурсивность - это не свойство самой функции, а способ ее описания. Рекурсивное определение короче и нагляднее, но требует в процессе своего выполнения больше времени и памяти.

Пример 2. Алгоритм Евклида.

$$d(m,n)=\begin{cases} m & , \text{ если } n=0 \\ d(n, m \bmod n), & \text{если } n>0 \end{cases}$$

```
Function Evklid(m,n:word):word;
Var  d : word;
Begin
  If n>m then
    d:=Evklid(n,m)
  Else
    If n=0 then
      d:=m
    Else
      d:=Evklid(n, m mod n);
  Evklid:=d;
End { Evklid };
```

Пример 3. Вычисление чисел Фибоначчи.

```
Function Fib(n:word):word;
Begin
  If n=0 then
    Fib:=0
  Else
    If n<=2 then
      Fib:=1
    Else
      Fib:=Fib(n-1)+Fib(n-2);
End { Fib };
```

Пример 4.

Степенную функцию, рассмотренную в разделе "Вычисление степенной функции", также можно реализовать как рекурсивную.

$$x^n=\begin{cases} 1, & \text{если } n=0 \\ (x^2)^{n/2}, & \text{если } n \text{ четное} \\ x\cdot x^{n-1}, & \text{если } n \text{ нечетное} \end{cases}$$

```
Function Power(x:real; n:word):real;
Begin
  If n=0 then
    Power:=1
  Else
    If not odd(n) then
      Power:=Power(x*x,n div 2)
    Else
      Power:=x*Power(x,n-1);
End { Power };
```

Из приведенных выше примеров видно, для завершения рекурсивных обращений требуется достижение некоторого конечного значения аргумента. Например, в примерах 1 и 2 это выполняется при n = 0, в примере 3 – при n = 0, 1 или 2. Если такое условие не реализовано, то мы получим бесконечную рекурсию. Хорошим примером такой рекурсии является следующий диалог.

У мужа спрашивают:
— Где ты деньги берешь?
— В тумбочке.
— А там они откуда?
— Жена кладет.
— А у нее откуда?
— Я даю.
— А где ты берешь?
— В тумбочке и т.д.

При написании обычных функций необходимо следить, чтобы они случайно не оказались рекурсивными. Особенно опасны в этом отношении функции без параметров.

Пример 5. Предположим, что в программе для одного и того же массива X несколько раз вычисляется среднее арифметическое значение S его элементов. Тогда такое вычисление можно оформить в виде функции без параметров.

```
Const  Nmax = 400;
Type  Ar = array[1..Nmax] of real;
Var  n : word;
     X : Ar;
Function MidAr;
Var  i : word;
Begin
  MidAr:=0;
  For i:=1 to n do
    MidAr:=MidAr+x[i];
  MidAr:=MidAr/n;
End { MidAr };
```

Здесь имя *MidAr* в правой части оператора присваивания воспринимается как обращение к функции *MidAr*. Это приводит к бесконечному рекурсивному вызову функции, следствием чего является прерывание программы с сообщением " 202 Stack overflow error" (переполнение стека), поскольку рекурсивный стек, как и локальные переменные, размещается в сегменте стека.

Чтобы исключить ошибочную рекурсивность функции, рекомендуется присваивать выходное значение имени функции лишь на заключительном этапе ее работы. Для функции *MidAr* получим:

```
Function MidAr;
Var  i : word;
     S : real;
Begin
  S:=0;
  For i:=1 to n do
    S:=S+x[i];
  S:=S/n;
  MidAr:=S;
End { MidAr };
```

В приведенных выше примерах имеет место прямая рекурсия, т.е. обращение к функции содержится в теле самой функции. Может быть также косвенная рекурсия, связывающая две подпрограммы-функции:

$$f_1(y) = F_1(f_2(x)); \quad f_2(x) = F_2(f_1(y))$$

Наряду с рекурсивными функциями могут быть и рекурсивные процедуры. В частности, каждую из вышеприведенных функций *Fact, Evklid, Fib* можно оформить в виде процедуры, перенеся выходное значение в список формальных параметров.

При прямой рекурсии нет нарушения основного принципа Паскаль-программы: любое имя может быть использовано лишь после его описания. При косвенной рекурсии такое нарушение имеет место.

Пусть две процедуры взаимно рекурсивны, т.е. первая вызывает вторую и, в свою очередь, вызывается второй при возврате. Схему их взаимодействия можно было бы представить, например, таким образом:

```
Procedure Proc1(a,b:real);
Var  k,l : integer;
Begin
..........
  Proc2(k,l);
..........
End { Proc1 };

Procedure Proc2(Var m:integer; n:integer);
Var  p,q : real;
Begin
..............
  Proc1(p,q);
.............
End { Proc2 };
```

Во время трансляции процедуры *Proc1* транслятор не может определить, что собой представляет имя *Proc2*, так как это имя еще не было описано в программе. В этом случае трансляция прерывается и на экран выдается сообщение "Error 3: Unknown identifier" (Неизвестный идентификатор). Тот же результат будет, если переставить местами описания процедур *Proc1* и *Proc2* (но теперь неизвестным идентификатором считается *Proc1*).

При трансляции обращения к процедуре компилятору не требуется иметь в своем распоряжении тело этой процедуры. Для выполнения указанной работы достаточно знать заголовок процедуры, содержащий полную информацию о количестве и типах формальных параметров. В связи с этим в Паскаль-программе принята следующая схема записи взаимно рекурсивных процедур.

Одна из процедур, например *Proc1*, записывается обычным способом. Описание второй процедуры разделяется на две части: опережающее и определяющее. В опережающее описание, которое ставится перед процедурой *Proc1,* включается заголовок процедуры *Proc2* и директива ***forward,*** которая указывает транслятору, что тело процедуры *Proc2* будет записано позже. Определяющее описание, которое ставится после описания процедуры *Proc1*, - это имя процедуры и ее блок. Для приведенного выше примера будем иметь:

```
Procedure Proc2(Var m:integer; n:integer);
forward;

Procedure Proc1(a,b:real);
Var  k,l : integer;
Begin
  ..........
  Proc2(k,l);
```

```
  ..........
End { Proc1 };

Procedure Proc2;
Var  p,q : real;
Begin
  ..............
  Proc1(p,q);
  ..............
End { Proc2 };
```

Из примера видно, что в определяющем описании заголовок процедуры записывается без списка формальных параметров, поскольку информация, содержащаяся в этом списке, уже не нужна транслятору.

ПОБОЧНЫЕ ЭФФЕКТЫ ФУНКЦИЙ

Основное назначение подпрограммы-функции - вычисление выходного значения. Это условно можно назвать ее главным эффектом.

Рассмотрим для примера стандартную функцию *sin(x).* Ее главный эффект - вычислить синус угла *x,* заданного в радианах. Никакой другой работы функция *sin(x)* в программе не выполняет. Более того, было бы нецелесообразно допустить, чтобы эта функция выполняла другую работу (например, изменяла значение переменной *x*).

Подпрограмма-функция может изменять значения локальных переменных, описанных в ее блоке. Однако это не оказывает влияния на работу всей программы, тем более что локальные переменные существуют лишь в период работы функции.

Тем не менее функция может выполнять действия, которые обнаруживаются вне тела этой функции, в частности, изменять значения глобальных переменных или формальных параметров-переменных со словом ***Var***. Такие действия называются побочным эффектом функции.

Термин "побочный эффект" взят из фармакологии: многие лекарства имеют побочный эффект, который может быть полезным или вредным.

Пример. Вычисление площади *n*-угольника, заданного координатами его вершин.

$$S = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1} - x_i)(y_{i+1} + y_i), \quad \text{где} \quad x_{n+1} = x_1; \quad y_{n+1} = y_1$$

```
Program Example;
Const  Nmax = 100;
Type  Ar = array[1..Nmax] of real;
Var  X,Y : Ar;
     n : byte;
     P,z,b : real;
Function Area(Var X,Y:Ar; n:byte):real;
Var  i : byte;  R : real;
Begin
  x[n+1]:=x[1]; y[n+1]:=y[1];
  P:=0;
  For i:=1 to n do
    P:=P+(x[i+1]-x[i])*(y[i+1]+y[i]);
  Area:=0.5*P;
  z:=z+sqr(P);
```

```
End { Area };
Begin
  В в о д   n, X, Y
  z:=10;
  S:=Area(X,Y,n);
  z:=z+1;
  Writeln('S= ',S:12,'   z= ',z:12);
End.
```

Из анализа текста только основной программы невозможно установить факт изменения переменной z после обращения к функции Area. Это действие скрыто в описании функции. В этой "завуалированности" проявления побочного эффекта содержится его основная опасность.

Функции, создающие побочный эффект, затрудняют понимание и отладку программы.

Процедура может иметь множество выходных значений, функция - только одно такое значение. Поэтому в общем случае нежелательно, чтобы функция выполняла в программе какую-либо другую работу, кроме вычисления выходного значения.

СТАНДАРТНЫЕ МОДУЛИ

Турбо Паскаль обеспечивает доступ к большому количеству встроенных констант, типов данных, переменных, процедур и функций. Эти элементы разделены на связные группы, называемые стандартными модулями, что позволяет использовать в конкретной программе только те из них, которые необходимы для ее работы.

В Турбо Паскале имеется 6 основных стандартных модулей: *System, Crt, Dos, Printer, Overlay, Graph*. Первые пять находятся в файле *Turbo.tpl* (Turbo Pascal Library), модуль *Graph* - в файле *Graph.tpu* (Turbo Pascal Unit). Текст стандартных модулей представлен в объектном коде, поэтому машинное время на их компиляцию не требуется.

Чтобы включить стандартный модуль в программу, его имя нужно указать в предложении использования ***Uses*:**

```
Uses  Crt, Printer, Dos.
```

Предложение ***Uses*** записывается после заголовка программы.

Модуль System - это системный модуль, обеспечивающий операции ввода-вывода, обработку строк, программную реализацию операций с плавающей запятой, динамическое распределение памяти, реализацию математических функций (*sin, cos, pred, odd* и др.). Модуль *System* автоматически включается в каждую программу, он считается предварительно объявленным. Последнее означает, что этот модуль не нужно указывать в предложении ***Uses***; более того, наличие имени *System* в предложении ***Uses*** воспринимается компилятором как ошибка с сообщением "Duplicate identifier" ("Повторный идентификатор").

Константы, типы, переменные, процедуры и функции, которые содержатся в модуле *System,* считаются стандартными (предопределенными). В принципе, их можно переопределить, например, указать

```
Var  sin : integer;
```

Тогда для обращения к функции синуса нужно использовать составное имя: y:=System.sin(x).

Разумеется, такие переопределения затрудняют понимание программы и применять их не рекомендуется.

Модуль Crt (Cathod ray tube - электронно-лучевая трубка) позволяет использовать все возможности дисплея и клавиатуры, включая управление режимами экрана, расширенные коды клавиатуры, цвет, окна и звуковые эффекты. Ниже перечислены некоторые процедуры и функции модуля *Crt.*

ClrScr (от слов Clear Screen) - очищает экран и помещает курсор в левый верхний угол.

ClrEol - очищает все символы, начиная от позиции курсора до конца строки, без перемещения курсора.

Delay(*m:word*) - задержка решения на *m* миллисекунд.

GotoXY(*x,y:byte*) - установка курсора в столбец x, строку y.

WhereX : byte - выдача номера текущего столбца.

WhereY : byte - выдача номера текущей строки.

KeyPressed : boolean - логическая функция для анализа нажатия клавиши.

ReadKey: char - функция, возвращающая символ нажатой клавиши.

Sound(*Hz:word*) - включение звука с частотой тона *Hz* в герцах.

NoSound - выключение звука.

Примечание. Процедуры и функции модуля *Crt,* как и модуля *System,* считаются предопределенными (предописанными), но не стандартными. Поскольку имя модуля *System* не записывается в составе фразы ***Uses,*** то формальным признаком стандартной функции или процедуры является отсутствие сообщения об ошибке от компилятора при отсутствии в программе фразы ***Uses.***

Такое же замечание справедливо также для модулей *Dos, Graph, Overlay.*

Модуль Dos. С помощью этого модуля реализуется ряд программ операционной системы и программ обработки файлов. Ниже приведены несколько процедур модуля *Dos.*

GetDate(***Var*** *Year, Month, Day, DW : word*) - считывает год, месяц, число и день недели с календаря ПЭВМ.

SetDate(***Var*** *Year,Month,Day: word*) - устанавливает год, месяц и число в календаре ПЭВМ.

GetTime(***Var*** *Hour, Min, Sec, Sec100 : word*) - считывает текущее время (часы, минуты, секунды, сотые доли секунды) по таймеру ПЭВМ.

SetTime(*Hour, Min, Sec, Sec100 : word*) - устанавливает текущее время на таймере ПЭВМ.

FindFirst - осуществляет поиск файла с заданным именем в файловой системе.

Модуль Printer. Это небольшой модуль, созданный для облегчения использования в программе устройства печати. Он описывает текстовый файл *Lst* и связывает его с устройством *Lpt1*.

Модуль Graph. Модуль реализует библиотеку из более чем 50 графических подпрограмм, выполняющих вычерчивание, закрашивание и штриховку графических изображений (точка, прямая, окружность и т.д.).

Модуль Overlay. Модуль используется для формирования оверлейных программ больших размеров, превышающих емкость доступной оперативной памяти.

МОДУЛИ ПОЛЬЗОВАТЕЛЯ

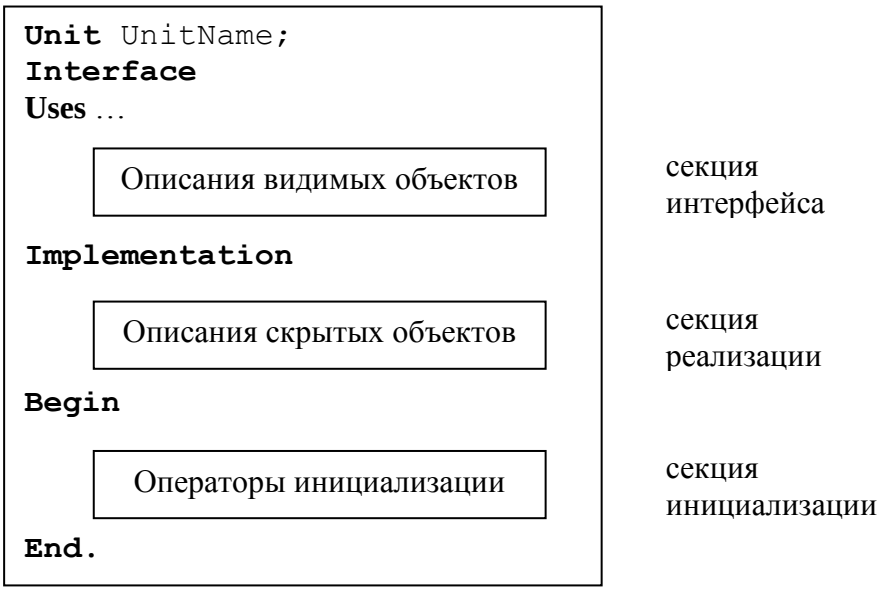
Модули пользователя - это инструмент для создания больших программ, что определяется следующими их свойствами:

- каждый модуль, входящий в состав программы, может иметь объем до 64 Кбайт;

- модуль представляет собой отдельно хранимую и независимо компилируемую единицу;
- количество модулей не ограничивается, оно определяется лишь доступным объемом свободной памяти.

Модуль разделяется на три части: секция интерфейса, секция реализации и секция инициализации.

Общая структура модуля:



Заголовок модуля состоит из зарезервированного слова ***Unit*** и следующего за ним идентификатора, являющегося именем модуля. Имя модуля должно быть уникальным. Предполагается, что имя модуля совпадает с именем файла, в котором хранится данный модуль. Например, модуль с заголовком ***Unit*** *BasUnit* должен находиться в файле с именем *'BasUnit.pas'*. Следствием последнего обстоятельства является то, что имя модуля, как и имя файла, должно состоять не более чем из восьми символов.

Интерфейсная часть начинается со служебного слова ***Interface.*** Если в данном модуле используются другие модули (стандартные или модули пользователя), то после слова ***Interface*** должно следовать предложение ***Uses*** с именами используемых модулей. Интерфейсная часть может содержать описания констант, типов, переменных, процедур и функций. Эти описания считаются глобальными, их может использовать любой другой модуль, во фразе ***Uses*** которого указано имя данного модуля.

Как было ранее отмечено, заголовок подпрограммы содержит всю информацию, необходимую для ее вызова: имя подпрограммы, количество и типы параметров и, если это функция, тип результата. Тело подпрограммы - это блок, определяющий алгоритм ее работы. С точки зрения вызывающей программы вся необходимая и достаточная информация содержится в заголовке подпрограммы, блок подпрограммы носит сугубо внутренний характер по отношению к вызываемой подпрограмме.

В связи с вышесказанным в интерфейсной части размещают только заголовки процедур и функций, имеющие значение при их глобальном использовании. Полное описание процедур и функций переносится в секцию реализации, начинающуюся со служебного слова ***Implementation.***

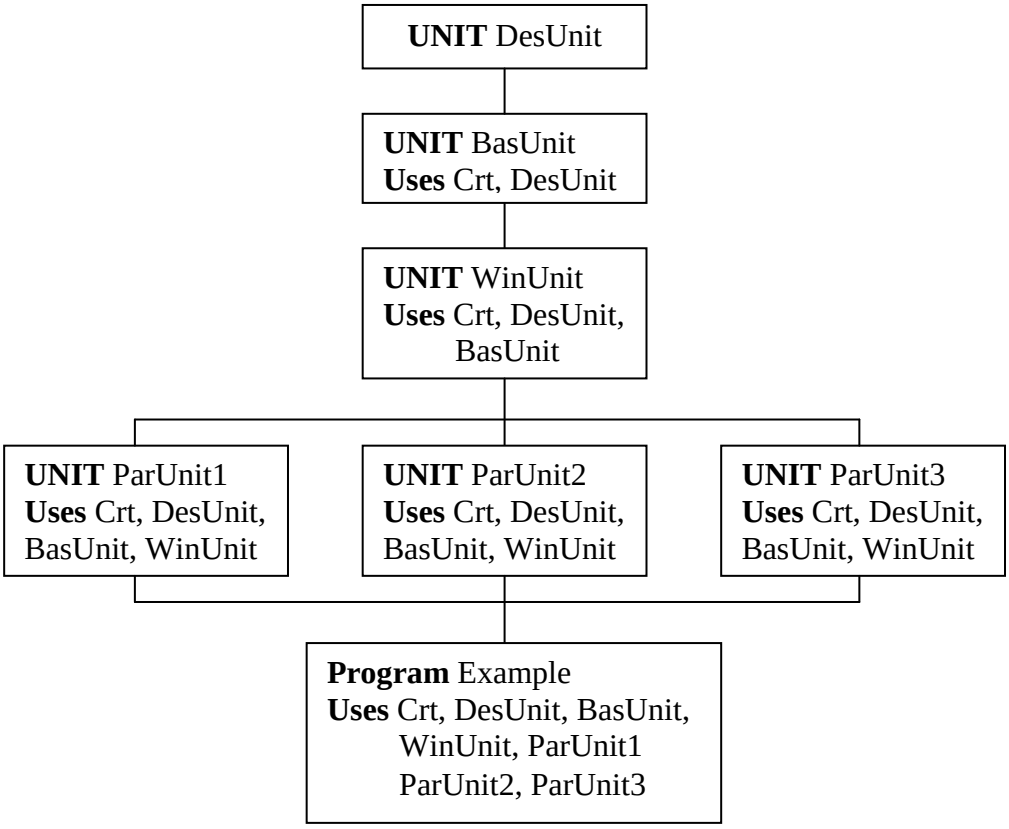
В секции реализации могут быть свои описания констант, типов и переменных, используемых только при работе этого модуля. Здесь могут содержаться также процедуры и функции, заголовки которых отсутствуют в секции интерфейса; эти процедуры и функции недо-

ступны для других модулей и могут быть активизированы только при работе данного модуля.

Поскольку другие модули могут использовать только объекты, описанные в секции интерфейса, но не в секции реализации, то говорят, что секция интерфейса содержит видимые объекты, а секция реализации - скрытые. Различие между секциями интерфейса и реализации можно определить еще следующим образом: секция интерфейса указывает, что делает модуль, а секция реализации описывает, как он это делает.

Секция инициализации является необязательной, ее наличие определяет слово ***Begin.*** Секции инициализации всех модулей вызываются перед запуском основного тела программы. Эту секцию обычно используют для установки начальных значений переменных данного модуля, для открытия файлов, автоопределения типа монитора и др. Секция инициализации может быть пустой, а при отсутствии слова ***Begin*** соответствующий модуль не содержит эту секцию.

Структура многомодульной программы может быть самой различной в зависимости от назначения программы, ее размера, объединения процедур по группам и т.д. Ниже приводится один из возможных вариантов такой структуры.



Здесь в модуле *DesUnit* размещены описания глобальных констант, типов и переменных (Des - от слова description, описание). Секция реализации этого модуля пустая. В модуле *BasUnit* сосредоточены базовые служебные процедуры и функции, используемые во многих других модулях (функция знака, процедуры контроля наличия файлов, двоичного поиска, группировки массивов, вывода массива на экран, удаления стека и т.п.). В модуле *WinUnit* записаны процедуры для организации многооконного интерфейса (включение и отключение курсора, сохранение и восстановление экрана, формирование различного типа меню и т.п.). Модули *ParUnit1, ParUnit2, ParUnit3* содержат процедуры и функции, реализующие различные режимы работы программы. В основной программе *Example* производится выбор режима ее работы путем активизации позиций меню и обращения к соответствующим процедурам.

Компиляция модуля возможна как из интегрированной среды Турбо Паскаля (turbo.exe), так и с помощью компилятора командной строки (tpc.exe). Так как модуль не является непосредственно выполняемой единицей, то в результате его компиляции образуется дисковый файл с расширением *tpu,* при этом имя этого файла копируется с имени файла с исходным текстом модуля. Компиляция модуля возможна, если к началу компиляции созданы все нестандартные модули с расширением *tpu,* имена которых перечислены в предложении ***Uses*** данного модуля.

Как известно, все переменные, которые описаны в разделах **Var,** размещаются в двух сегментах памяти: в сегменте данных и в сегменте стека. При этом сегмент стека используется для локальных переменных, а сегмент данных – для глобальных переменных и типизированных констант.

Локальными считаются все переменные, которые описаны в блоках процедур и функций, глобальными – те, что описаны вне пределов этих блоков. Следовательно, сегмент данных используется не только для переменных, описанных в разделе **Var** основной программы, но и в разделах **Var**, расположенных в секциях интерфейса и реализации данной программы. Хотя последние две группы переменных расположены в одном сегменте памяти, но между ними имеется существенная разница:

- переменные, описанные в секции реализации, могут быть использованы лишь при работе данного модуля;
- переменные, описанные в секции интерфейса, могут быть использованы в данном модуле и во всех других, включающих в себя фразу ***Uses*** с именем данного модуля.

Разумеется, эти ограничения обеспечиваются во время компиляции программы.

Рассмотрим еще раз понятие о глобальных и локальных переменных, но уже с учетом модулей. Как известно, локальными считают переменные, описанные в блоках процедур и функций, а глобальными – переменные, описанные вне этих блоков. Следовательно, если переменная описана в секции интерфейса или в секции реализации, то она является глобальной, и память для нее выделяется в сегменте данных.

СХЕМА РАСПРЕДЕЛЕНИЯ ПАМЯТИ ПРОГРАММЫ НА ТУРБО ПАСКАЛЕ

На следующей странице изображена схема распределения памяти ПЭВМ для программы на Турбо Паскале.

Часть памяти занимает ядро операционной системы. Остальная память предоставляется для программы пользователя.

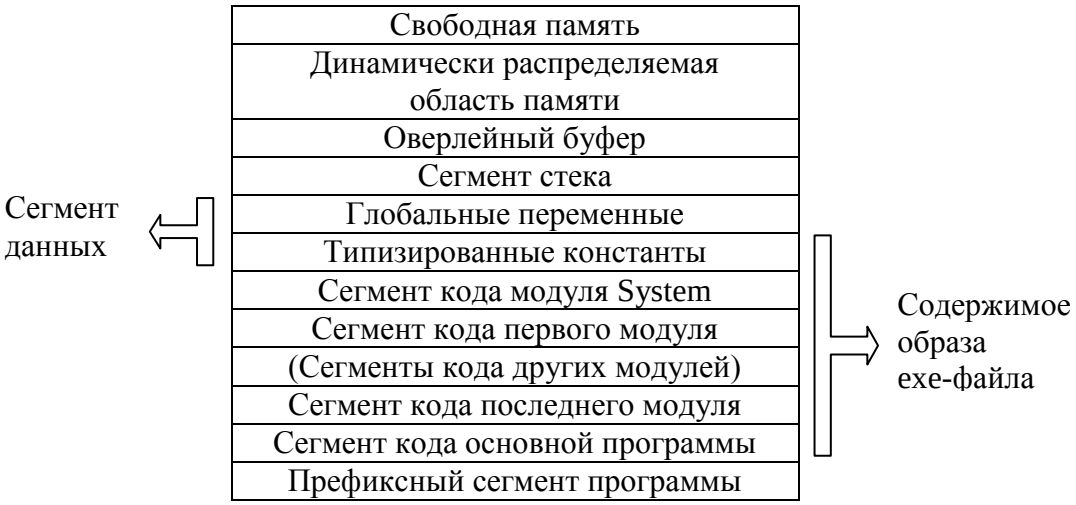
Префиксный сегмент программы *PSP* (Program Segment Prefix) - это область длиной 256 байт, которая строится операционной системой при загрузке файла *exe.* Дальше располагается образ *exe*-файла, т.е. копия файла в оперативной памяти.

Каждому программному модулю соответствует сегмент его кода. Основная программа занимает первый сегмент. Дальше располагаются сегменты модулей в порядке, обратном их перечислению в предложении ***Uses.*** Последний сегмент занимает модуль *System.*

Размер отдельного сегмента не должен превышать 64 Кбайта.

В сегменте данных размещаются статические глобальные переменные и все типизированные константы. Следует еще раз подчеркнуть, что глобальными переменными считаются все переменные, не описанные в блоках процедур и функций. Следовательно, в сегменте данных размещаются переменные, описанные в разделах ***Var*** основной программы, а также в секциях интерфейса и реализации ее модулей.

Сегмент стека отводится для локальных переменных процедур и функций, размещаемых здесь при их активизации. По умолчанию размер этого сегмента 16 Кбайт, однако его можно изменить директивой компилятора $M .



Размер динамической области памяти также можно регулировать директивой $M . По умолчанию максимальный размер этой области принимается равным 1 Мбайт. Каждая отдельная динамическая переменная может занимать область памяти размером не более 64 К.

ПРОЦЕДУРА ЗАПОЛНЕНИЯ FillChar

С помощью процедуры *FillChar* можно заполнить любое поле памяти одним и тем же однобайтным значением. Заголовок процедуры имеет вид

```
        FillChar(Var V; N:word; b:byte)
или     FillChar(Var V; N:word; ch:char).
```

Здесь *V* - переменная любого типа;
N - количество байтов в поле *V*, которые заполняются значением *b* или *ch.*

Поскольку после имени формальной переменной *V* не записано имя типа, то этой переменной может соответствовать фактический параметр любого типа.

Предположим, что нам требуется обнулить целочисленный массив *A* и вещественный массив *B*. В цикле это можно сделать следующим образом:

```
Const  NaMax = 1000;  MbMax = 25;  NbMax = 30;
Type   Ar = array[1..Namax] of integer;
       Matrix = array[0..MbMax,1..NbMax] of real;
Var   i,j : word;
      A : Ar;
      B : Matrix;
Begin
  For i:=1 to NaMax do
    a[i]:=0;
  For i:=0 to MbMax do
    For j:=1 to NbMax do
      b[i,j]:=0;
```

Однако более эффективно (по быстродействию и требуемому объему памяти) эта задача решается с помощью процедуры *FillChar:*

```
FillChar(A,SizeOf(A),0);   FillChar(B,SizeOf(B),0).
```

Здесь процедура *FillChar* заполняет каждый байт полей *A* и *B* нулевым значением.

Если в массиве *A* нужно обнулить только элементы 101 .. 500, а в матрице *B* - ее первые две строки, то это можно сделать следующим образом:

```
FillChar(A[101],400*SizeOf(integer),0);
  FillChar(B,2*NbMax*SizeOf(real),0).
```

Если массив, элементы которого занимают свыше одного байта памяти, нужно заполнить ненулевым значением, то в этом случае использование процедуры *FillChar* приведет к ошибке. Например, при обращении

```
FillChar(A,SizeOf(A),1)
```

каждому элементу массива *A* будет присвоено значение 257, а не 1 (a_i = 00010001).

Процедура *FillChar* может быть использована также для заполнения строки одним и тем же символом. Однако в этом случае нужно позаботиться о принудительном установлении в нулевом байте текущей длины строки.

Пример.

```
Var  S1 : string[80];
     S2 : string;
Begin
  FillChar(S1,SizeOf(S1),' ');  S1[0]:=chr(80);
  FillChar(S2[1],100,#32);      S2[0]:=chr(100);
```

Так как *SizeOf*(*S1*) = 81, то процедура *FillChar* заполняет пробелом все байты строки S1, в том числе и нулевой байт. После этого в нулевой байт принудительно заносится текущая длина строки. При отсутствии оператора S1[0]:=chr(80) в нулевой байт S1[0], как и в другие байты строки S1, будет занесен пробел, номер которого по таблице ASCII равен 32. Следовательно, при дальнейшей обработке строки S1 будет считаться, что ее длина равна 32 символам.

В строке *S2* заполнение пробелом начинается с адреса *S2*[1], т.е. с первого ее символа. Поскольку содержимое нулевого байта при этом не изменилось (в данном случае оно осталось неопределенным), то оператором присваивания формируется конкретное значение текущей длины строки *S2.*

Каждый байт строки, в том числе и нулевой байт, рассматривается как символ. В связи с этим нельзя указывать текущую длину строки оператором *S1*[0] := 80.

ПРОЦЕДУРА ПЕРЕМЕЩЕНИЯ ДАННЫХ MOVE

Процедура *Move* позволяет максимально быстро переместить в оперативной памяти блок данных заданного размера. Заголовок процедуры:

```
Move(Var Source,Dest; n:word),
```

где *Source* - имя исходного поля памяти (источник данных);

Dest - имя конечного поля памяти (приемник данных);

n - количество байтов перемещаемых данных.

Пример 1. Даны два однотипных массива. Присвоить элементам одного из этих массивов значения элементов другого массива.

```
Const  Nmax = 1000;
Type  Ar = array[1..Nmax] of real;
Var  i : word;
     A,B : Ar;
Begin
  A:=B;
```

Эту же работу можно выполнить процедурой *Move*(*B,A,SizeOf*(*A*)).

Эффективность обоих решений здесь одинакова, поскольку на машинном уровне компилятор конструирует в этом случае одну и ту же последовательность команд.

Пример 2. Элементам 101 .. 500 массива *A* присвоить значения элементов 401 .. 800 массива *B*.

Варианты решения:

```
а)  For i:=101 to 500 do
      a[i]:=b[i+300];
б)  Move(B[401],A[101],400*SizeOf(real)).
```

Второй вариант более эффективен по сравнению с первым.

Пример 3. Два массива имеют одинаковый размер, один и тот же тип элементов, но описаны разными именами типа. Присвоить элементам одного массива значения элементов другого массива.

```
Const  Nmax = 1000;
Type  Ar1 = array[1..Nmax] of real;
      Ar2 = array[1..Nmax] of real;
Var  i : word;
     A : Ar1;  B : Ar2;
Begin
```

Варианты решения:

```
а) For i:=1 to 1000 do
     a[i]:=b[i];
б) Move(B,A,SizeOf(A)).
```

Оператор присваивания *A := B* в данном случае недопустим, так как массивы *A* и *B* формально имеют различные имена типов.

Процедура *Move* эффективно работает также при сдвиге элементов массива.

Пример 4. Элементы массива *A*, начиная с a_{101}, сдвинуть на две позиции влево.

```
Type  Ar1 = array[1..1000] of real;
Var  i : word;
     A : Ar1;
Begin
а)  For i:=101 to 1000 do
      a[i-2]:=a[i];
б)  Move(a[101],a[99],900*SizeOf(real)).
```

Пример 5. То же, но на две позиции вправо.

```
а)  For i:=1000 downto 103 do
      a[i]:=a[i-2];
```

б) `Move(a[101],a[103],898*SizeOf(real))`.

Пример 6. Из массива $X(n)$ удалить подмассив элементов с индексами от $k1$ до $k2$ $(k1 \le k2 \le n)$.

Решение:

```
Move(x[k2+1],x[k1],(n-k2)*SizeOf(real))
Dec(n,k2-k1+1);
```

Здесь

m = n-k2 – количество переносимых элементов;

k = k2-k1+1 – количество удаляемых элементов.

УПРАВЛЕНИЕ ЭКРАНОМ В ТЕКСТОВОМ РЕЖИМЕ

Аппаратная реализация вывода текста или изображения на экран осуществляется видеомонитором, который включает в себя экран дисплея вместе с электронно-лучевой трубкой (ЭЛТ) или жидкокристаллической (ЖК) панелью, а также комплекс технических средств, обеспечивающих появление изображения на экране. В ЭЛТ при соударении пучка электронов с поверхностью экрана, покрытой люминофором, образуется светящаяся точка, которую называют пикселем. Электронный луч обегает экран слева направо и сверху вниз 25 раз в секунду, формируя множество близко расположенных пикселей. ЖК-панель представляет собой совокупность ЖК-ячеек, каждая из которых генерирует один пиксель. Программист может управлять светимостью экрана в любом его месте вплоть до отдельного пикселя.

Наиболее важные электронные компоненты видеомонитора - это контроллер (схема управления электронно-лучевой трубкой или ЖК-панелью), порты ввода-вывода, запоминающее устройство для генерации символов и видеопамять. Эти компоненты располагаются на одной печатной плате, которая называется дисплейным адаптером.

Для получения на экране цветного изображения требуется, чтобы видеомонитор имел цветной адаптер и цветную ЭЛТ или ЖК-панель.

Изображение, выводимое на экран, предварительно формируется в видеопамяти. Эта память имеет два входа (два порта). Один порт используется сравнительно медленной программой для чтения или записи информации в видеопамять. Второй порт предназначен для быстродействующей схемы развертки изображения, обеспечивающей его регенерацию с частотой 25 гц.

Первый видеоадаптер (MDA), обладавший довольно скромными возможностями, появился в 1981 г. В дальнейшем его сменили все более совершенные модели (HEC, CGA, EGA, VGA, SVGA). Они отличаются друг от друга по количеству пикселей, объему видеопамяти, количеству цветов и другим параметрам. При этом в качестве стандартного режима вывода текста был принят режим, реализованный в адаптере CGA.

Типичный экран ПЭВМ имеет 200 строк по 640 пикселей в строке, т.е. 128000 пикселей. В монохромном режиме для каждого пикселя в видеопамяти требуется один бит, который определяет максимальную или минимальную яркость этого пикселя. Тогда видеопамять должна иметь объем 128000/8 = 16000 байт. В цветном мониторе каждый пиксель формируется как совокупность трех цветных точек: красной, зеленой и синей. В этом случае объем видеопамяти должен быть соответственно больше.

Различают два режима работы с экраном: текстовый и графический. Эти режимы существенно отличаются друг от друга способом использования видеопамяти.

В текстовом режиме на экран выводятся только символы, которые заранее предопределены таблицей, содержащей 256 их возможных вариантов. Для изображения символа на экране используется матрица из $8 \times 8 = 64$ пикселей, которую называют знакоместом. Первые 8 строк развертки образуют 80 знакомест первой текстовой строки экрана, следующие 8 строк развертки - это символы второй текстовой строки и т.д. Поэтому при выводе символа достаточно указать лишь его номер, для чего требуется один байт. Используя этот номер, электронные схемы развертки находят в специальной области памяти одну из 256 матриц 8×8 разрядов и вычерчивают соответствующий символ.

Примечание. В зависимости от типа адаптера количество пикселей в знакоместе может быть различным. Например, для EGA оно равно 14×8, для VGA 19×8.

В видеопамяти для каждого знакоместа на самом деле отводится не один, а два байта: один байт (байт символа) определяет номер символа, второй байт (байт атрибутов) используется для указания о цвете символа и цвете окружающего фона.

Таким образом, для хранения "карты" экрана в текстовом режиме требуется $80 \cdot 25 \cdot 2 = 4000$ байт видеопамяти.

Объем видеопамяти любой ПЭВМ составляет не менее 16000 байт. Целиком эта память необходима лишь при выводе графической информации. В текстовом режиме используется часть видеопамяти, которая называется страницей текстового экрана. Если емкость экрана 25×80 знакомест, то одна страница - это 80*25*2 = 4000 байт.

Байт атрибутов имеет следующую структуру:

7	6	5	4	3	2	1	0
B	back			H	text		

Здесь *back* - цвет фона, *text* - цвет символа. Каждый разряд в разделах *back* и *text* - это один из основных цветов монитора: красный, зеленый или синий. Смешивание основных цветов дает 8 цветовых комбинаций. *H* - разряд интенсивности цвета. С учетом интенсивности получаем 16 цветов для отображения символа (темные и светлые). Для фона возможны только темные цвета. Разряд *B* управляет мерцанием символа: если *B* = 1, символ мигает с частотой около 2 гц.

К видеопамяти можно обращаться из программы с помощью адресов. Для нее в ПЭВМ зарезервированы адреса с сегментной частью от $A000 до $DFFF, т.е. 256 Кбайт.

В стандартном модуле *Crt* определены константы для всех 16 значений цветового атрибута в соответствии с приведенной ниже таблицей.

Бит	Бит цвета			Десятич.	Имя	Цвет
Н	кр.	зел.	син.	значение	константы	
0	0	0	0	0	Black	Черный
0	0	0	1	1	Blue	Синий
0	0	1	0	2	Green	Зеленый
0	0	1	1	3	Cyan	Голубой
0	1	0	0	4	Red	Красный
0	1	0	1	5	Magenta	Фиолетовый
0	1	1	0	6	Brown	Коричневый
0	1	1	1	7	LightGray	Светло-серый
1	0	0	0	8	DarkGray	Темно-серый
1	0	0	1	9	LightBlue	Ярко-синий
1	0	1	0	10	LightGreen	Ярко-зеленый
1	0	1	1	11	LightCyan	Ярко-голубой
1	1	0	0	12	LightRed	Розовый
1	1	0	1	13	LightMagenta	Малиновый
1	1	1	0	14	Yellow	Желтый
1	1	1	1	15	White	Белый
				128	Blink	Мерцание символа

В этом же модуле определена переменная *TextAttr* типа *byte*, которая задает значение текстового атрибута.

Пример 1.

```
TextAttr:=Yellow+16*Brown+Blink;
Writeln('Мерцание желтых символов на коричневом фоне');
```

Аналогично:

```
TextAttr:=14+16*6+128;
Writeln('Мерцание желтых символов на коричневом фоне');
```

Для управления цветом символов и цветом фона могут быть использованы также следующие две процедуры:

1) *TextColor*(*Color:byte*) - текущий цвет для символов; если *Color* > 16, то принимается *Color* := *Color* ***div*** 16 и добавляется мерцание символа.

2) *TextBackGround*(*Color:byte*) - текущий цвет фона.

Пример 2.

```
TextColor(Yellow+Blink); TextBackGround(Brown);
Writeln('Мерцание желтых символов на коричневом фоне');
```

Аналогично:

```
TextColor(14+128); TextBackGround(6);
Writeln('Мерцание желтых символов на коричневом фоне');
```

СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ ЭКРАНА

При формировании на экране сменяющих друг друга окон, при выборе режима работы программы с помощью экранного меню и других ситуациях требуется предварительно запоминать состояние экрана, а затем его восстанавливать. Эта процедура сводится к копированию видеопамяти в оперативную память и обратно.

В примерах, которые будут в дальнейшем иллюстрировать работу с экраном, используются различные переменные, связанные с видеопамятью. Чтобы не повторять описание переменных, будем считать, что это описание содержится в модуле *DesUnit,* а в обрабатывающей программе задано предложение ***Uses*** *DesUnit*.

Текст модуля *DesUnit:*

```
UNIT DesUnit;
{ Модуль содержит глобальные описания типов и переменных }

Interface

Type
   ScreenColRange = 1..80;  { тип номера столбца экрана }
   ScreenRowRange = 1..25;  { тип номера строки экрана }
   ScreenChar = record      { элемент видеопамяти }
       Symbol : char;       { символ }
       Attrib : byte;       { атрибут }
   end;
   ScreenRowArray = array[ScreenColRange] of ScreenChar;
                            { тип строки экрана }
   ScreenArray = array[ScreenRowRange] of ScreenRowArray;
                            { тип видеопамяти }
   ScreenPointer = ^ScreenArray;
Var  Screen : ScreenArray  absolute  $B800:$0000;
                            { видеопамять }
     RowAr : ScreenRowArray;{ строка видеопамяти }
     BufAr : ScreenPointer; { указатель буферного }
                        { массива для сохранения видеопамяти }
Implementation
End.
```

Сформируем две процедуры, пересылающие из видеопамяти в оперативную память и обратно содержимое экрана, ограниченное строками *Row1* и *Row2*.

```
Procedure MoveFromScreen(Row1,Row2:byte);
Var  Len : word;
Begin
  New(BufAr);
  Len:=(Row2-Row1+1)*80 shl 1;
  Move(Screen[Row1],BufAr^,Len);
End { MoveFromScreen };

Procedure MoveToScreen(Row1,Row2:byte);
Var  Len : word;
  Begin
    Len:=(Row2-Row1+1)*80 shl 1;
    Move(BufAr^,Screen[Row1],Len);
    Dispose(BufAr);
End { MoveFromScreen };
```

Здесь *Row1, Row2* - соответственно номер первой и последней строк сохраняемой или восстанавливаемой области экрана; операция ***shl*** *k* (Shift Left) - сдвиг целочисленной переменной на *k* разрядов влево, что эквивалентно умножению ее на 2^k.

СДВИГ ЭКРАНА

При вертикальном сдвиге текстового экрана вверх строки 2 .. 25 переписываются в строки 1 .. 24, а в строку 25 вводятся данные из оперативной памяти. Сдвиг вниз осуществляется аналогично.

Вертикальный сдвиг экрана называют также скроллингом (scroll - рулон, свиток).

Пусть для заполнения строки экрана в программе используется переменная *BufRow* типа *ScreenRowArray*.

Вертикальный сдвиг вверх:

```
Procedure ScreenToUp;
Begin
  Move(Screen[2],Screen[1],3840);
  Screen[25]:=BufRow;
End { ScreenToUp };
```

Вертикальный сдвиг вниз:

```
Procedure ScreenToDown;
Begin
  Move(Screen[1],Screen[2],3840);
  Screen[1]:=BufRow;
End { ScreenToDown };
```

В программах иногда требуется выполнять также горизонтальный сдвиг экрана, в частности при обработке текста.

Пусть нам требуется сдвинуть экран на пять столбцов влево. В этом случае левые пять столбцов исчезнут, весь текст сдвинется влево, а самые правые пять столбцов будут очищены. Сдвиг вправо осуществляется аналогично.

Ниже приведена процедура для сдвига текстового экрана на один столбец влево.

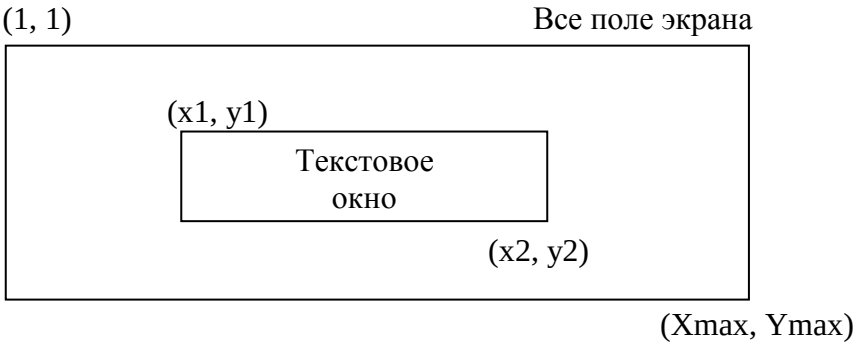
```
Procedure ScreenToLeft;
Var  i : byte;
Begin
  For i:=1 to 25 do
    Begin
      Move(Screen[i,2],Screen[i,1],158);   { 79 · 2 = 158 }
      Screen[i,80].Symbol:=' ';
    End;
End { ScreenToLeft };
```

В столбцы, очищенные при горизонтальном сдвиге, в дальнейшем можно записать какую-либо информацию из оперативной памяти.

ПРОЦЕДУРЫ УПРАВЛЕНИЯ ТЕКСТОВЫМ РЕЖИМОМ ЭКРАНА

Ниже приведены краткие сведения о некоторых процедурах и функциях, используемых для управления текстовым режимом работы экрана. Более подробная информация содержится в [2, 6, 8].

1. Процедура **Window(x1,y1,x2,y2:byte);**
Процедура устанавливает текущее окно на экране:



При этом должны соблюдаться условия:

$$1 \le x_1 < x_2 \le X_{\max}$$

$$1 \le y_1 < y_2 \le Y_{\max}$$

При нарушении этих условий окно не создается.

Параметр $X_{\max}$ может иметь значения 40 или 80, параметр $Y_{\max}$ - 25, 43 или 50 в зависимости от типа адаптера.

После выполнения процедуры *Window* все действия с экраном относятся к той его части, которая определена координатами x_1, y_1, x_2, y_2. Отсчет строк и столбцов для позиционирования курсора теперь производится в координатах текущего окна, а позиция (1,1) - это левый верхний угол окна. Сразу после выполнения процедуры *Window* курсор устанавливается в позицию (1,1) созданного окна.

Координаты очередного создаваемого окна всегда даются в абсолютных экранных координатах, а не в относительных координатах последнего текстового окна.

2. Процедура **ClrScr.**
Процедура очищает текущее текстовое окно, закрашивая его текущим цветом фона. При этом курсор устанавливается в позицию (1,1).

3. Процедура **GotoXY(x,y:byte);**
С помощью этой процедуры можно устанавливать курсор в столбец x и строку y текущего окна. При этом последующая операция вывода текста на экран разместит первый символ выводимой строки в позицию (x, y). Процедура *GotoXY* использует систему координат текущего текстового окна. При выводе символов или другой информации по мере необходимости на экране происходит прокрутка, или сдвиг, изображения. Это всегда имеет место при выводе кодов конца строки (код #10) в последней строке окна операторами *Write* и *Writeln* или когда выводимая строка не помещается в последней строке текстового окна. В то же время вывод типа

```
GotoXY(5,25); Write('строка');
```

не вызовет сдвига вверх, потому что оператор *Write* не переводит строки. Однако вывод хотя бы одного символа в правый нижний угол текстового окна вызовет прокрутку:

```
GotoXY(80,25); Write('*');
```

и символ '*' окажется уже в 24-ой строке, а не в 25-ой.

Устранить указанный эффект можно с помощью прокрутки экрана вниз после вывода символа в последней позиции последней строки (например, с помощью процедуры *InsLine*).

4. Функции **WhereX** и **WhereY**.

Используются для программного опроса текущего положения курсора в текстовом окне.

5. Процедура **ClrEOL.**

Эта процедура может использоваться как для стирания "хвостов" строк, так и для раскраски чистого экрана в полоску максимально быстрым способом. Процедура стирает все символы в строке, начиная с текущей позиции курсора и до правого края текущего окна. Вместо стираемых символов она ставит пробелы, при этом цвет строки определяется цветовым атрибутом фона.

6. Процедуры **InsLine** и **DelLine.**

Эти процедуры позволяют "прокручивать" часть текстового окна или весь экран вверх и вниз. *InsLine* вставляет пустую строку на место той, где находится в текущий момент курсор. Все нижние строки, начиная с нее, смещаются вниз на одну строку. Самая нижняя строка уйдет за нижнее поле окна и исчезнет.

Процедура *DelLine* удаляет строку, в которой находится курсор, подтягивая на ее место все нижестоящие строки. При этом освобождается самая нижняя строка экрана.

Все строки, которые освобождаются при работе процедур *InsLine* и *DelLine*, закрашиваются текущим цветом фона.

ИСПОЛЬЗОВАНИЕ ПРЕРЫВАНИЙ

Многие из процедур работы с экраном не могут быть выполнены путем применения только средств Турбо Паскаля. Для реализации таких процедур, как ввод-вывод в ненулевые страницы видеопамяти, управление курсором и др., необходимо использовать функции операционной системы MS DOS. Обращение к таким функциям осуществляется через так называемые прерывания, которые представляют собой стандартные подпрограммы базовой системы ввода-вывода (BIOS) операционной системы MS DOS.

Обращение к программным прерываниям выполняется с помощью процедуры *Intr,* заголовок которой имеет вид

```
Intr(N:byte; Var Reg:Registers),
```

где *N* - номер программного прерывания;

Reg - запись типа *Registers,* через которую передаются параметры заданного прерывания.

Предописанный тип *Registers* определен в стандартном модуле *Dos* и представляет собой запись с вариантами:

```
Type  Registers = record
        Case integer of
           0 : (AX,BX,CX,DX,BP,SI,ES,Flags : word);
           1 : (AL,AH,BL,BH,CL,CH,DL,DH : byte);
        end;
```

Переменная типа *Registers* служит для доступа к регистрам микропроцессора. Вариант 0 позволяет обращаться к 16-разрядным, а вариант 1 - к 8-разрядным регистрам.

Пример.

```
Var  R1,R2 : Registers;
Begin
  R1.AX:=$01FF;
```

```
  R2.AL:=$CA;
.............
```

Подробно прерывания описаны в техническом руководстве по операционной системе MS DOS.

Значения некоторых прерываний:

$5 - распечатка содержимого экрана;
$10 - управление экраном;
$12 - размер памяти;
$16 - ввод с клавиатуры;
$21 - вызов функции и т.д.

Прерывание $21 содержит в себе большой набор функций операционной системы. Номер функции определяется содержимым регистра AH.

Примеры функций для прерывания $21:

$5 - вывод на печатающее устройство;
$19 - задание текущего дисковода;
$2A - выбор даты;
$2C - выбор времени по таймеру;
$3D - открытие файла и т.д.

Пример. Составить программу, определяющую размер оперативной памяти компьютера. Для выполнения данной задачи можно использовать прерывание $12, которое проверяет оперативную память и помещает размер этой памяти в регистр AX.

```
Var  Size : word;
     Reg : Registers;
Begin
  Intr($12,Reg);
  Size:=Reg.AX;
  Writeln('ПЭВМ имеет ',Size,' Кбайт оперативной памяти');
```

Для управления видеотерминалом используется прерывание $10. Функции, выполняемые этим прерыванием, также определяются содержимым регистра AH. Некоторые функции прерывания $10 приведены на следующей странице.

Функция AH = 2 устанавливает положение курсора в координатах 0 ..79, 0 .. 24. Для каждой страницы имеется свой закрепленный за нею курсор.

Функция AH = 5 используется для того, чтобы сделать видимой нужную текстовую страницу.

При AH = 9 и AH = 10 курсор не перемещается, а записываемый символ и его копии появляются на месте курсора и справа от него. При достижении правого края экрана организуется переход на новую строку.

Функция AH = 14 имитирует вывод на экран в режиме пишущей машинки. Именно этот режим реализуется процедурами *Write* и *Writeln*, но только для нулевой страницы.

Содержимое регистра AH	Действие	Содержимое регистров
1	Установить размер курсора	CH - начальная линия CL - конечная линия BH - номер страницы
2	Установить положение курсора	DH - номер строки DL - номер позиции BH - номер страницы
5	Установить страницу	AL - номер страницы
9	Записать символ с атрибутами	AL - символ BL - атрибуты BH - номер страницы CX - количество копий символа
10	Записать символ без атрибутов	AL - символ BH - номер страницы CX - количество копий символа
14	Записать символ и сдвинуть курсор	AL - символ BH - номер страницы

УПРАВЛЕНИЕ ФОРМОЙ КУРСОРА

Курсор служит двум целям. Во-первых, он указывает позицию экрана, в которую программа должна производить вывод очередного символа. Во-вторых, он делает видимой на экране точку отсчета для пользователя. Только для второй цели курсор должен быть видимым. Тем не менее вне зависимости от того, видим он или невидим, курсор всегда указывает определенную позицию на экране. Это связано с тем, что операционная система в любом случае производит вывод на экран, начиная с текущей позиции курсора.

Программа может управлять формой курсора, вплоть до полного его отключения, с помощью функции 1 программного прерывания $10. Для этого требуется лишь загрузить в регистры CH и CL номера начальной и конечной линий курсора.

Курсор строится из тонких горизонтальных отрезков, верхний из которых называется начальной линией курсора, а нижний - его конечной линией. Размер курсора по вертикали не может превышать размер знакоместа, ширина курсора всегда равна ширине знакоместа. Начальная и конечная линии отсчитываются сверху вниз: 0 .. 7 или 0 .. 13.

В знакоместе 8 × 8 нормальный курсор занимает строки 7 и 8, в знакоместе 14 × 8 - строки 12 .. 14. Значение начальной строки, равное 32 ($20), отключает курсор, т.е. делает его невидимым. Мигание видимого курсора отключить нельзя, так как оно реализовано аппаратно.

Ниже приведена программа, иллюстрирующая управление курсором.

```
Program DemoCursor;
Uses  Crt, Dos;
{ ------------------------------- }
Procedure WaitEnter;
Var  ch : char;
Begin
  Repeat
    ch:=ReadKey;
  Until  ord(ch)=13;
End { WaitEnter };
```

```
{ ------------------------------- }
Procedure SetCursor(Cursor:word);
Var  Reg : Registers;
Begin
  With Reg do
    Begin
      AH:=1;             { Номер функции }
      BH:=0;             { Номер страницы }
      CH:=Hi(Cursor);    { Начальная линия }
      CL:=Lo(Cursor);    { Конечная линия }
    End;
  Intr($10,Reg);
End { SetCursor };
{ ------------------------------- }
Begin
  SetCursor($0607); Writeln('Нормальный курсор');
  WaitEnter;
  SetCursor($0007); Writeln('Блок-курсор');
  WaitEnter;
  SetCursor($2000); Writeln('Отключенный курсор');
  WaitEnter;
  SetCursor($0707); Writeln('Тонкий курсор');
  WaitEnter;
End.
```

ПРОЦЕДУРНЫЕ И ФУНКЦИОНАЛЬНЫЕ ТИПЫ И ПЕРЕМЕННЫЕ

Рассмотрим следующие два примера.

Пример 1. Вычислить определенный интеграл произвольной функции $f(x)$ в интервале $[a,b]$ с погрешностью, не превышающей заданного значения ε.

Как известно, определенный интеграл - это площадь фигуры, ограниченной подынтегральной функцией $f(x)$, осью абсцисс и вертикальными прямыми $x=a$ и $x=b$. Для определения площади можно отрезок $[a,b]$ разделить на n частей и вычислять элементарные площади тем или иным приближенным способом. Наиболее простым и в то же время наименее точным способом является вычисление элементарной площади по формуле прямоугольника; более точным является представление элементарной фигуры как трапеции; еще точнее - аппроксимация трех смежных точек кривой $f(x)$ уравнением параболы.

Предположим, что мы реализовали программу вычисления определенного интеграла каким-либо способом, но в задаче требуется несколько раз вычислять интеграл для различных функций. В этом случае целесообразно такую программу оформить в виде подпрограммы-функции, предусмотрев в списке ее параметров имя формальной функции, а при обращении к ней передавать имя конкретной функции. Следовательно, возникает вопрос о способе представления в Паскаль-программе имени функции как параметра.

Пример 2. Предположим, что составленная нами программа содержит два больших фрагмента, почти полностью совпадающих между собой. Различными являются лишь несколько строк, содержащих не совпадающие последовательности операторов. Здесь целесообразно поступить следующим образом:

- оформить несовпадающие строки операторов как две различные процедуры, например, с именами *Proc1* и *Proc2;*

- объединить два фрагмента в одну процедуру *Frag*, реализовав несовпадающие строки как обращение к формальной процедуре *Proc*;

- при обращении к процедуре *Frag* в первом случае указывать имя фактической процедуры *Proc1*, во втором - имя *Proc2*.

Следовательно, в этом примере возникает вопрос о способе представления в Паскаль-программе имени процедуры как параметра.

Турбо Паскаль позволяет интерпретировать процедуры и функции как объекты, которые можно присваивать переменным и передавать их в качестве параметров. Следовательно, если переменной можно присвоить имя процедуры или имя функции, то эта переменная должна иметь в программе тип процедурной или функциональной переменной. В дальнейшем, если это не оговорено особо, термин "процедурная переменная" будем относить как к собственно процедурным переменным, так и к функциональным переменным. Ниже записано несколько примеров описания процедурных и функциональных типов.

Пример 3.

```
Type  Proc = procedure;
      StringProc = procedure(S:string);
      IntProc = procedure(Var m,n:integer; k:integer);
      MastFunc = function(a,b:real):real;
```

Имена формальных параметров здесь играют иллюстративную роль.

Синтаксис записи процедурного (функционального) типа совпадает с записью заголовка процедуры (функции), но здесь не указывается имя процедуры (функции).

Пример процедурных переменных:

```
Var  F1,F2 : MastFunc;
     P1,P2 : Proc;
     Sp1,Sp2 : StringProc;
```

Процедурной переменной можно присвоить значение процедурного типа. Таким значением может быть другая процедурная переменная или процедурная константа. Процедурной константой является идентификатор процедуры или функции.

Пример 4.

```
Type  StringProc = procedure(S:string);
      MastFunc = function(a,b:real):real;
Var  F1,F2 : MastFunc;
     Sp1,Sp2 : StringProc;
     S : string;
{$F+}
Procedure Rant(St:string);
Begin
......
End { Rant };
Function Vent(p,q:real):real;
Begin
......
End { Vent };
{$F-}
Begin
  F1:=Vent;  Sp1:=Rant;
...................
```

Тогда обращения

```
Rant(S);    y:=Vent(5,10);
```

будут эквивалентны обращениям

```
Sp1(S);    y:=F1(5,10);
```

Внутреннее представление процедурной переменной - это указатель, адрес которого определяет точку входа соответствующей процедуры или функции. Если мы присваиваем процедурной переменной значение процедурной константы, то внутреннее представление этой константы должно быть таким же, т.е. это должно быть четырехбайтное поле, первое слово которого - адрес сегмента, а второе - смещение.

Переменные и константы в Турбо Паскале могут иметь или полный, или краткий адрес. Полный адрес - это обычная запись адреса в виде сегмента и смещения (4 байта), в кратком адресе содержится только смещение (2 байта). Если переменная или константа (в том числе процедурная константа) используются лишь внутри одного модуля, то они имеют краткий адрес, в противном случае для них формируется полный адрес. В частности, адрес входа внутренней процедуры всегда формируется кратким. Использование кратких адресов существенно сокращает общий объем программы.

Если процедура может применяться как процедурная константа для присваивания значения процедурной переменной или для передачи ее в качестве фактического параметра, то адрес входа такой процедуры должен быть полным вне зависимости от того, используется она в одном или в нескольких модулях. Чтобы обеспечить при любых условиях формирование полного адреса точки входа процедуры, применяется директива компилятора {$F+}. Вместо директивы $F можно использовать директиву Far (дальний тип вызова), записываемую после заголовка процедуры или функции.

Для присваивания процедурной переменной какого-либо значения необходимо, как и при любом другом присваивании, чтобы переменная в левой части и значение в правой части оператора присваивания были совместимы по присваиванию.

Процедурные типы, чтобы быть совместимыми по присваиванию, должны иметь одно и то же количество параметров, а параметры на соответствующих позициях должны быть одинакового типа.

Если имя процедуры или функции присваивается процедурной переменной, то дополнительно должны быть выполнены следующие требования:

- процедура должна компилироваться в состоянии {$F+};
- это не должна быть стандартная процедура или функция;
- такая процедура или функция не должна быть вложенной.

Стандартными процедурами и функциями считаются процедуры и функции, описанные в модуле *System* (*writeln, sin, ord* и др.). Процедуры и функции из других модулей (*ClrScr, ReadKey, GetTime* и др.) - это предописанные (предопределенные) процедуры и функции.

Чтобы использовать стандартную процедуру или функцию в качестве процедурной константы, для нее нужно написать специальную "оболочку".

Пример 5.

```
Type  ProcInt = procedure(n:integer);
      FuncReal = function(x:real):real;
Var   k : integer;
      x,y : real;
      P : ProcInt;
      F : FuncReal;
{$F+}
Procedure WriteInt(m:integer);
Begin
  Writeln(m);
End { WriteInt };
Function RealSin(x:real):real;
Begin
```

```
  RealSin:=sin(x);
End { RealSin };
{$F-}
Begin
...........
  P:=WriteInt;
  P(k);
  F:=RealSin;
  y:=F(x);
............
```

Требование о программной оболочке связано с тем, что большинство стандартных процедур и функций могут иметь параметры различного типа, а некоторые из них также переменный по количеству список параметров. Например, аргумент функции *sin*(*x*) может быть как вещественный, так и целочисленный; в списке вывода процедуры *Write* может быть любое количество переменных различного типа. Наличие программной оболочки фиксирует как количество, так и типы параметров.

Процедурный тип может участвовать в описании составного типа.

Пример 6.

```
Type IntProc = procedure(n:integer);
     StringProc = procedure(S:string);
     ProcAr = array[1..10] of IntProc;
     ProcRec = record
        Ap : IntProc;
        Bp : StringProc
     end;
Var  Ss : ProcAr;
     Rr : ProcRec;
```

Пример 7. Вычислить

$$S_1 = \int_0^5 (\sin(x)+1)^2\,dx,\ \varepsilon = 0.001 \quad \text{и} \quad S_2 = \int_1^3 e^{\sqrt{|x|+1}}\,dx,\ \varepsilon = 0.01$$

В программе численного интегрирования для вычисления площади будем использовать метод прямоугольников.

```
Program NumInteg;
Type  FuncType = function(x:real):real;
Var  Int1,Int2 : real;
{------------------------------------------------------- }
Function Func1(x:real):real; Far;
Begin
  Func1:=sqr(sin(x)+1);
End { Func1 };
{------------------------------------------------------- }
Function Func2(x:real):real; Far;
Begin
  Func2:=exp(sqrt(abs(x)+1));
End { Func2 };
{------------------------------------------------------- }
Function Integ(Func:FuncType; a,b,eps:real):real;
{ Численное интегрирование методом прямоугольников }
Var  i,n : word;
     h,S1,S2 : real;
```

```
Begin
  n:=10; h:=(b-a)/n;
  S2:=0;
  For i:=1 to n do
    S2:=S2+h*Func(a+(i-1)*h);
  Repeat
    S1:=S2; n:=2*n; h:=(b-a)/n;
    S2:=0;
    For i:=1 to n do
      S2:=S2+h*Func(a+(i-1)*h);
  Until abs(S2-S1)<=eps;
  Integ:=S2;
  Writeln('n=',n);
End { Integ };
{------------------------------------------------------------ }
Begin
  Int1:=Integ(Func1,0,5,0.001);
  Writeln('Int1=',Int1:14);
  Int2:=Integ(Func2,1,3,0.01);
  Writeln('Int2=',Int2:14);
End.
```

Результаты работы программы:

```
n = 2560
Int1 = 9.0696552E+00
n = 640
Int2 = 1.1365466E+01
```

Для получения решения в *Int1* потребовалось в 4 раза больше делений интервала, чем в *Int2*. Это связано с двумя обстоятельствами:

1) синусоидальная функция гораздо быстрее изменяется по сравнению с экспоненциальной, в связи с чем для ее численного интегрирования требуются более мелкие шаги;

2) для получения значения *S*1 задана более высокая точность по сравнению со значением *S*2.

В тексте функции *Integ* следует обратить внимание на такую деталь. Формальному параметру *Func* типа *FuncType* при обращении к функции *Integ* соответствуют фактические параметры *Func1* и *Func2*, являющиеся именами конкретных функций, т.е. функциональными константами. Следовательно, формальный параметр *Func* в этом случае может быть только параметром-значением, т.е. перед именем *Func* в списке формальных параметров нельзя ставить слово ***Var.***

УПРАВЛЕНИЕ КЛАВИАТУРОЙ

В состав клавиатуры входит встроенный микропроцессор, основной функцией которого является опрос (сканирование) клавишей и формирование кода нажатой клавиши. Этот код называется кодом сканирования (скан-кодом) и представляет собой порядковый номер клавиши. Кодов сканирования ровно столько, сколько клавишей в клавиатуре. Микропроцессор следит также за временем, в течение которого клавиша удерживается в нажатом состоянии. Если это время превышает 0,5 с, то генерируется повторение скан-кодов с частотой 10 гц.

Примеры скан-кодов:

Escape	1
Enter	28
L (l, Д, д)	38

Если микропроцессор обнаружил нажатие клавиши, то вырабатывается прерывание $9, центральный процессор прекращает выполнение программы, после чего начинает работать подпрограмма обработки прерывания $9. Эту подпрограмму называют драйвером клавиатуры, она входит в состав MS DOS.

Драйвер клавиатуры анализирует скан-коды нажатых клавишей, формирует код символа и помещает его в буфер клавиатуры, который является областью оперативной памяти, способной запомнить до 15 вводимых символов. Буфер клавиатуры организован по принципу циклической очереди. При переполнении буфера поступающие символы игнорируются, а драйвер клавиатуры генерирует звуковой сигнал.

Имеются два типа кодов символов: коды ASCII и расширенные коды. Коды ASCII - это однобайтные числа, соответствующие таблице ASCII. В этой таблице первые 32 символа - управляющие коды, которые обычно используются для передачи команд периферийным устройствам. После них идут цифры, большие и малые латинские буквы, специальные символы, буквы национального алфавита, символы псевдографики и др.

Расширенные коды присвоены клавишам или комбинациям клавишей, которые не имеют представляющего их символа ASCII. Это функциональные клавиши F1 .. F10, комбинации с клавишей Alt, клавиши PgUp, PgDn и др. Расширенные коды имеют длину два байта, причем первый байт всегда содержит код ASCII #0.

Примеры расширенных кодов (указывается значение второго байта):

Shift + Tab	15
Alt + A	30
F1	59
Home	71
Ctrl + End	117

Исключением являются четыре функциональные клавиши, для которых генерируется однобайтный код:

BackSpace	8
Tab	9
Enter	13
Escape	27

В буфере клавиатуры для каждого символа отводится два байта. Для однобайтных кодов первый байт содержит код ASCII, а второй - скан-код клавиши. Для расширенных кодов в первом байте содержится нулевое значение, а во втором - номер расширенного кода, который в большинстве случаев совпадает с кодом сканирования.

Процедуры *Read, Readln* и функции *KeyPressed, ReadKey*, выполняющие ввод с клавиатуры, работают фактически с буфером клавиатуры.

Функция *KeyPressed,* предописанная в модуле *Crt,* возвращает значение *true*, если в буфере клавиатуры имеется хотя бы один символ, и *false* в противном случае.

При старте программы буфер клавиатуры обычно пустой. Любое нажатие клавишей, которым соответствует код ASCII или расширенный код, заносит код символа в буфер клавиатуры. Коды символов хранятся в буфере до тех пор, пока они либо не будут считаны, либо буфер не будет очищен самой программой.

Полностью очищают буфер процедуры *Read* и *Readln*. Функция *ReadKey* считывает из буфера клавиатуры первый символ и удаляет его из состава этого буфера. Если буфер пуст,

то функция *ReadKey* приостанавливает работу программы и ждет, пока не будет нажата какая-либо клавиша, генерирующая символьный код.

Если в программе имеется фраза

```
If KeyPressed then
   Оператор,
```

то это не рассматривается как реакция на сиюминутное нажатие клавиши. Это будет реакция на состояние буфера клавиатуры, куда могли быть занесены ранее символы при случайном нажатии клавишей.

Перед опросом клавиатуры и в конце работы программы рекомендуется очищать буфер клавиатуры. Это можно выполнить, например, с помощью такой процедуры:

```
Procedure ClearKeyBuffer;
Var  ch : char;
Begin
  While KeyPressed do
    ch:=ReadKey;
End { ClearKeyBuffer };
```

Процедура *ClearKeyBuffer* удаляет из буфера клавиатуры с помощью функции *ReadKey* по одному символу до тех пор, пока не будет очищен весь буфер.

При выводе информации на экран возникает необходимость приостанавливать работу программы до тех пор, пока не будет нажата определенная клавиша, например клавиша Esc. Такую работу можно выполнить с помощью следующей процедуры:

```
Procedure WaitEscape;
Var  ch : char;
Begin
  ClearKeyBuffer;
  Repeat
    ch:=ReadKey;
  Until  ch=#27;
End { WaitEscape };
```

Для определения кода клавиши или сочетания клавишей можно использовать приведенную ниже программу KeyTest.

```
Program KeyTest;
Uses  Crt;
Var  ch : char;
Begin
  ClearKeyBuffer;
  Repeat
    ch := ReadKey;
    Write(ch,'  ',ord(ch));
    If ord(ch)=0 then
      Begin
        ch:=ReadKey;
        Writeln('  ',ch,'  ',ord(ch));
      End
    Else
      Writeln;
  Until ord(ch)=13;
End.
```

Программа KeyTest опрашивает клавиши до тех пор, пока не будет нажата клавиша Enter, имеющая код ASCII #13.

В различных формах диалога с пользователем (выбор позиции меню, нажатие клавишей по указаниям информационной строки экрана, ответ на прямой вопрос программы и т.п.) программа ожидает нажатия одной из определенных заранее клавишей или группы клавишей. Опрос клавиатуры в этом случае может быть обеспечен с помощью приведенной ниже функции *GetKey.*

```
Function GetKey : byte;
Var  ch : char;
Begin
  ch:=ReadKey;
  If ord(ch)=0 then       { Расширенный код символа }
    GetKey:=ord(ReadKey)
  Else
    GetKey:=ord(ch);      { Нормальный код символа }
End { GetKey };
```

Следует отметить, что функциональная надежность использования функции *GetKey* может быть обеспечена лишь в случае, когда в процессе диалога используются только управляющие клавиши (одинарные или комбинации клавишей). Это связано с тем, что второй байт расширенного кода, как правило, совпадает с нормальным кодом какого-либо алфавитно-цифрового символа. Примеры таких совпадений приведены ниже в таблице.

Ниже приведена таблица.с примерами совпадения расширенного и нормального кодов

Управляющие клавиши	Второй байт расш.кода	Алфавитно-цифр.символы	Нормальный код символа
PgDn	81	Q	81
F7	65	A	65
F10	68	D	68
Alt+N	49	1	49
Alt+M	50	2	50
Shift+F2	85	U	85

Если в программе в процессе диалога могут одновременно использоваться управляющие и алфавитно-цифровые символы, то рекомендуется применять приведенную ниже функцию *GetKeyWord.*

```
Function GetKeyWord : word;
Var  ch : char;
Begin
  ch:=ReadKey;
  If ord(ch)=0 then        { Расширенный код символа }
    GetKeyWord:=word(ord(ReadKey))+256
  Else
    GetKeyWord:=ord(ch);   { Нормальный код символа }
End { GetKeyWord };
```

Алфавитно-цифровые клавиши многозначны в смысле генерируемых при их нажатии кодов. В ряде случаев в программе нужно обеспечить реакцию на нажатие определенной клавиши вне зависимости от связанных с этой клавишей кодов (верхний или нижний ре-

гистр, латинский или русский алфавит). Например, такая необходимость возникает при ответе "Да" или "Нет" на вопрос программы. Альтернативный выбор в таком случае можно обеспечить, например, с помощью приведенной ниже функции *GetYesNo.*

```
Function GetYesNo:boolean;
Var  ch : char;
     Cond : boolean;
Begin
  Cond:=false;
  Repeat
    ch:=ReadKey;
    If ch in ['Д','д','L','l'] then
      Begin
        Cond:=true; GetYesNo:=true;
      End
    Else
      If ch in ['Н','н','Y','y'] then
        Begin
          Cond:=true; GetYesNo:=false;
        End;
  Until Cond;
End { GetYesNo }:
```

Цикл Repeat работает здесь до тех пор, пока не будет нажата клавиша, соответствующая букве «Д», или клавиша, определенная для буквы «Н».

При утвердительном ответе ("Да") выходное значение функции *GetYesNo* равно *true*, при отрицательном - *false.*

ФОРМИРОВАНИЕ МЕНЮ

Наиболее примитивным способом организации меню является выбор позиции меню по ее порядковому номеру с помощью оператора ***Case.***

```
Const  St : array[1..4] of string[32]=
        ('П о з и ц и я  1','П о з и ц и я  2',
         'П о з и ц и я  3','В ы х о д');
Var  i,Switch : byte;
     Cond : boolean;
Begin
  For i:=1 to 4 do
    Writeln(i:2,'  ',St[i]);
  Writeln('Укажите позицию меню');
  Cond:=false;
  Repeat
    Readln(Switch);
    Case Switch of
      1 : Proc1;
      2 : Proc2;
      3 : Proc3;
      4 : Cond:=true;
    end;
  Until Cond;
```

Оператор ***Case*** формирует обращение к одной из процедур *Proc1 .. Proc3* в зависимости от введенного значения переменной *Switch.*

Более совершенным является формирование такого меню, в котором строка-курсор указывает на выбираемую позицию путем ее цветового выделения. При этом клавиши перемещения курсора обеспечивают видимое передвижение по позициям меню, а активизация выбранной позиции производится нажатием заранее определенной управляющей клавиши (например, Enter или Tab).

Ниже приведен фрагмент программы, реализующей таким способом вертикальное меню. При этом в текст программы не включены функция *GetKey* и процедура *SetCursor*, рассмотренные в предыдущих разделах.

```
Uses  Crt,Dos;
Const  St : array[1..5] of string[32]=('П о з и ц и я  1',
                 'П о з и ц и я  2','П о з и ц и я  3',
                 'П о з и ц и я  4','В ы х о д');
    BlueYellow=16*Blue+Yellow; { атрибут выделенной позиции }
    LightGrayBlack = 16*LightGray+Black; { атрибут остальных }
                                         { позиций меню      }
    UpBorder=10;             { верхняя граница рамки }
    LeftBorder=18;           { левая граница рамки }
    LengthString=34;         { длина строки-подсветки меню }
    MenuCount=5;             { кол-во позиций меню }
Type  ScreenChar = record    { тип элемента видеопамяти }
         Symbol : char;      { символ }
         Attrib : byte;      { атрибут }
      end;
      ScreenArray = array[1..25,1..80] of ScreenChar;
                             { тип видеопамяти }
Var  i,
     CurrRow,CurrCol,    { текущие строка и столбец экрана}
     Key,                { ординальный номер символа }
     Switch : byte;      { номер позиции меню }
     CondExit : boolean; { признак выхода из меню }
     Screen : ScreenArray absolute $B800:$0000;{ видеопамять }
{-----------------------------------------------------}
Procedure InsertColorString(Row,Col,Len,Color : byte);
{ Выделение строки Row цветом Color (изображение и фон) }
Var  j : byte;
Begin
  For j:=Col to Col+Len do
    Screen[Row,j].Attrib:=Color;
End { InsertColorString };
{-----------------------------------------------------}
Procedure MoveCursor(UpRow,DnRow,StepRow,Len,NewColor,
                     OldColor:byte);
{ Перемещение курсора-строки длиной Len по экрану: UpRow, }
{ DnRow,StepRow - верхняя,нижняя границы и шаг перемещения;}
{ NewColor,OldColor - цветовые атрибуты строки-курсора и }
{ остального поля  }
Var  Cond : boolean;
Begin
  Repeat
    Key:=GetKey; Cond:=false;
    Case Key of  { верт.перемещение цветового выделения }
     9,13 : Cond:=true;                      { Tab, Enter }
       72 : Begin                                  { Up }
              InsertColorString(CurrRow,CurrCol,Len,OldColor);
              If CurrRow>=UpRow+StepRow then
                CurrRow:=CurrRow-StepRow
```

```
              Else
                CurrRow:=DnRow;
            End;
        80 : Begin                                          { Down }
              InsertColorString(CurrRow,CurrCol,Len,OldColor);
              If CurrRow<=DnRow-StepRow then
                CurrRow:=CurrRow+StepRow
              Else
                CurrRow:=UpRow;
            End;
    end;
    InsertColorString(CurrRow,CurrCol,Len,NewColor);
  Until Cond;
End { MoveCursor };
{-------------------------------------------------------}
Procedure Frame(x1,y1,x2,y2,Color : byte);
{ Вычерчивание прямоугольной рамки с заданными координатами }
{ одинарной линией; цвет и фон - Color }
Var  ChGr,ChVr,         { символы для гор. и верт.линий }
     LfUp,LfDn,         { рамки, для левого верхнего,   }
     RtUp,RtDn : char; { левого нижнего, правого верхнего }
     i,j : byte;        { и правого нижнего углов рамки }
Begin
  TextAttr:=Color;
  ChGr:=#196; ChVr:=#179; LfUp:=#218;
  LfDn:=#192; RtUp:=#191; RtDn:=#217;
  GotoXY(x1,y1); Write(LfUp);
  For i:=x1+1 to x2-1 do
    Write(ChGr);
  Write(RtUp);
  For i:=y1+1 to y2-1 do
    Begin
      GotoXY(x1,i); Write(ChVr);
      GotoXY(x2,i); Write(ChVr)
    End;
  GotoXY(x1,y2); Write(LfDn);
  For i:=x1+1 to x2-1 do
    Write(ChGr);
  Write(RtDn);
  For i:=y1+1 to y2-1 do       { Очистка области экрана,}
    For j:=x1+1 to x2-1 do     { ограниченного рамкой   }
      Begin
        Screen[i,j].Symbol:=' ';
        Screen[i,j].Attrib:=Color;
     End;
End { Frame };
{-------------------------------------------------------}
Begin
  SetCursor($2000);                    { отключение курсора }
  TextBackGround(Magenta); ClrScr;        { очистка экрана }
  Frame(LeftBorder,UpBorder,LeftBorder+LengthString+6,
        UpBorder+2*MenuCount+2,LightGrayBlack);    { рамка }
  For i:=1 to MenuCount do   { печать в рамке позиций меню }
    Begin
      GotoXY(LeftBorder+4,UpBorder+2*i); Write(St[i]);
    End;
  CurrRow:=UpBorder+2;                 { начальное значение }
  CurrCol:=LeftBorder+3;               { позиции экрана }
```

```
  InsertColorString(CurrRow,CurrCol,LengthString,BlueYellow);
                    { выделение начальной позиции экрана }
  CondExit:=false;
  Repeat
    MoveCursor(UpBorder+2,UpBorder+2*MenuCount,2,LengthString,
               BlueYellow,LightGrayBlack); { управление пере-}
    Switch:=(CurrRow-UpBorder) div 2;      { движением строки-}
    Case Switch of                         { курсора }
      1 : Proc1;
      2 : Proc2;
      3 : Proc3;
      4 : Proc4;
      5 : CondExit:=true;
    end;
  Until CondExit;
  SetCursor($0607);                { включение курсора }
End.
```

В основной программе вначале формируется на экране рамка путем обращения к процедуре *Frame*. Цвет рамки и фоновый цвет ее внутренней области определяются значением формального параметра *Color*, которому при обращении присваивается значение константы *LightGrayBlack*.

После отработки процедуры *Frame* на экране печатаются строки массива *St*, определяющие содержание позиций меню. Начальная активизация одной из позиций обеспечивается заданием значений переменным *CurrRow* (номер текущей строки экрана) и *CurrCol* (номер текущего столбца экрана), после чего процедура *InsertColorString* подсвечивает выделенную позицию цветом *BlueYellow*.

Передвижение строки-курсора по позициям меню осуществляется процедурой *MoveCursor*, выход из которой производится при нажатии клавишей Enter или Tab. При этом текущее значение переменной *CurrRow* указывает положение активизированной позиции меню, после чего с помощью оператора ***Case*** осуществляется обращение к процедуре, соответствующей этой позиции.

Аналогичным образом можно организовать формирование горизонтального меню.

ОПРОС И НАЗНАЧЕНИЕ ДАТЫ

Для опроса и назначения даты используются процедуры *GetDate* и *SetDate*, входящие в стандартный модуль *Dos*. Заголовки этих процедур имеют вид:

```
GetDate(Var год,месяц,число,день_недели: word);
SetDate(год,месяц,число: word).
```

Допустимые диапазоны для составных частей даты:

```
год          1980..2099
месяц        1..12
число        1..31
день недели  0..6
```

День недели с номером 0 - это воскресенье.

Процедуры *GetDate* и *SetDate* могут использоваться, например, для определения дня недели любого числа в диапазоне от 1980-го до 2099-го года.

```
Program WeekDay;
Uses  Crt,Dos;
Type  String11=string[11];
```

```
      WeekAr=array[0..6] of String11;
Const WeekDays : WeekAr = ('воскресенье','понедельник',
                 'вторник','среда','четверг',
                 'пятница','суббота');
Var  Y,M,D,Wd : word;
     w : String11;
{ ---------------------------- }
Function WhatDay(Year,Month,Day : word): String11;
Var  Yf,Mf,Df,Wdf : word;
Begin
  GetDate(Yf,Mf,Df,Wdf);
  SetDate(Year,Month,Day);
  GetDate(Year,Month,Day,Wdf);
  WhatDay:=WeekDays[Wdf];
  SetDate(Yf,Mf,Df);
End { WhatDay };
{ ---------------------------- }
Begin
  ClrScr;
  Writeln('Введите год'); Readln(Y);
  Writeln('Введите месяц'); Readln(M);
  Writeln('Введите число'); Readln(D);
  w:=WhatDay(Y,M,D);
  Writeln('         год: ',Y);
  Writeln('       месяц: ',M);
  Writeln('       число: ',D);
  Writeln(' день недели: ',w);
End.
```

Назначением функции *WhatDay* является определение дня недели для произвольной даты. Для этого запоминается текущая дата в локальных переменных *Yf, Mf, Df,* записывается в календарь компьютера заданная дата, при обращении к процедуре *GetDate* определяется день недели, а затем процедурой *SetDate* восстанавливается текущая дата.

ОПРОС И НАЗНАЧЕНИЕ ВРЕМЕНИ

Опрос и назначение времени по таймеру ПЭВМ выполняется процедурами *GetTime* и *SetTime*, входящими в стандартный модуль *Dos*. Заголовки процедур:

```
GetTime(Var Hour,Min,Sec,Sec100 : word);
  SetTime(Hour,Min,Sec,Sec100 : word).
```

Здесь *Hour* - час, *Min* - минута, *Sec* - секунда, *Sec100* - сотая доля секунды.

Процедура *GetTime* читает с таймера значение времени; процедура *SetTime* устанавливает на таймере новое значение времени.

Диапазоны изменения переменных:

```
Hour        0 .. 23
Min, Sec    0 .. 59
Sec100      0 .. 99
```

С помощью процедуры *GetTime* можно определить время работы программы или фрагмента программы.

```
Program Time;
Uses  Crt, Dos;
Var  h,m,s,s100 : word;
```

```
     t1,t2,dt : longint;
Begin
.....................
  GetTime(h,m,s,s100);
  t1:=60*h; t1:=((t1+m)*60+s)*100+s100;
.....................
  GetTime(h,m,s,s100);
  t2:=60*h; t2:=((t2+m)*60+s)*100+s100;
  dt:=t2-t1;
  h:=dt div 360000; dt:=dt mod 360000;
  m:=dt div 6000;   dt:=dt mod 6000;
  s:=dt div 100;    s100:=dt mod 100;
  Writeln('h= ',h,'  m= ',m,'  s= ',s,'  s100= ',s100);
  WaitEscape;
End.
```

Переменные *t1* и *t2* определяют текущее время, выраженное в сотых долях секунды.

Примечание. Если бы в программе было написано

```
t1:=((60*h+m)*60+s)*100+s100,
```

то при вычислении выражения в правой части оператора присваивания могло быть переполнение разрядной сетки переменной типа *word* (все переменные, входящие в состав выражения, имеют тип *word;* следовательно, и значение выражения также будет иметь тип *word*). Например, при *h* = 20, *m* = 0, *s* = 0, *s100* = 0 получим значение 7200000, что значительно превышает максимально возможное для типа *word.* Напротив, в выражении

```
(t2+m)*60+s)*100+s100
```

переменная *t2* имеет тип *longint,* остальные переменные - тип *word*; при вычислении этого выражения производится преобразование разнотипных целочисленных операндов к более старшему типу, т.е. к типу *longint.*

ДИРЕКТИВЫ КОМПИЛЯТОРА

Директивы компилятора управляют режимом компиляции программы. Они представляют собой комментарии со специальным синтаксисом и могут быть поставлены в программе в любом месте, где разрешается комментарий.

Синтаксис директивы компилятора:

{$ <имя> <знак или параметр>}

Имеются три типа директив:

1. Ключевые директивы, которые включают или выключают конкретные режимы компиляции с помощью указания знаков "+" или "-" после имени директивы.

2. Директивы с параметрами, в которых после имени директивы записываются определенные параметры.

3. Директивы условной компиляции, управляющие порядком компиляции частей исходного текста программы.

Директивы могут быть глобальными или локальными. Глобальные директивы влияют на всю компиляцию, а локальные - только на ту часть программы, которая начинается с данной директивы и заканчивается следующим ее появлением.

Глобальные директивы должны указываться перед разделами описаний программы или модуля, в частном случае перед словами ***Program*** и ***Unit.***

Несколько директив могут быть сгруппированы в одном комментарии с разделением их запятыми. Например,

{$B+, R+, S-}

В данном разделе будут рассмотрены наиболее часто используемые директивы.

1. Выбор типа вызова: {$F+} или {$F-} .

По умолчанию принято {$F-}. Тип директивы - локальный.

Как известно, адрес процедуры или функции может быть полным, т.е. состоящим из адреса сегмента и смещения, и коротким, состоящим только из смещения. В первом случае мы имеем дальний тип вызова (*Far*), во втором - ближний (*Near*).

Для процедур и функций, имена которых указаны в интерфейсной секции модуля, всегда генерируется дальний тип вызова, для локальных процедур и функций по умолчанию генерируется ближний тип.

Если задана директива F+, то для всех процедур и функций, кроме внутренних, генерируется дальний тип. Внутренние процедуры и функции всегда имеют короткий адрес вне зависимости от директивы F.

Установление директивы F+ требуется при использовании оверлеев, а также программ с процедурными переменными.

2. Контроль ввода-вывода: {$I+} или {$I-} .

По умолчанию принято {$I+}. Тип директивы - локальный.

Директива задает или отменяет автоматическое генерирование машинных команд, проверяющих результат выполнения операции ввода-вывода. Если процедура ввода-вывода возвращает ненулевой результат ее выполнения при включенной директиве I+, то программа завершает работу с выводом сообщения об ошибке ввода-вывода. В состоянии I- прерывания программы не происходит, а результат выполнения операции ввода-вывода можно проверить с помощью функции *IOResult* типа *word.*

При обращении к функции *IOResult* ее выходное значение обнуляется.

Пример 1. Проверка наличия файла.

```
Procedure ExisFile(Var F:text; S:string);
Begin
  {$I-} Reset(F); {$I+}
  If IOResult<>0 then
    Begin
      Writeln('Файл ',S,' отсутствует'); Halt
    End;
End { ExisFile };
  .....................................
Begin
  Assign(FileInput,'Input.dat');
  ExisFile(FileInput,'Input.dat');
```

В этом случае можно сообщить, какой именно файл отсутствует в заданном каталоге.

Примечание. Если в вышеприведенной программе записать

```
  {$I-} Reset(F); {$I+}
  k:=IOResult;
  If IOResult<>0 then ... ,
```

то в этом случае некорректность операции ввода не будет обнаружена.

Пример 2. Проверка готовности принтера.

```
Procedure PrinterReady(Var Key:byte);
Var  Cond : boolean;
Begin
  Repeat
```

```
    Cond:=true; Key:=0;
    {$I-} Writeln(Lst); {$I+}
    If IOResult<>0 then
      Begin
        Cond:=false;
        Writeln('Включите принтер и проверьте ',
                'его готовность,');
        Writeln('после чего нажмите клавишу Enter.');
        Writeln('Отмена печати - Esc');
        Repeat
          Key:=ord(ReadKey);
          If Key=27 then Exit;
        Until Key=13;
      End;
  Until Cond;
End { PrinterReady };
```

Процедура *PrinterReady* работает до тех пор, пока не будет устранена неготовность принтера или не будет отменена печать результатов. В последнем случае в вызывающей программе должно анализироваться значение выходной переменной *Key*.

Пример 3. Проверка ввода числового ответа.

```
Function GetNumberInt(S:string; MinNumber,
                MaxNumber:longint):longint;
Var  Number : longint;
     k : integer;
     Cond : boolean;
Begin
  Writeln(S);
  Repeat
    Repeat
      {$I-} Read(Number); {$I+}
      k:=IOResult;
      If k<>0 then
        Begin
          Writeln(#7'Неправильный формат числа');
          Writeln('Повторите ввод');
        End
    Until k=0;
    Cond:=(Number<=MaxNumber) and (Number>=MinNumber);
    If not Cond then
      Begin
        Writeln(#7'Значение указано неверно:');
        Writeln('Допустимый диапазон ',MinNumber,'..',
                MaxNumber);
        Writeln('Повторите ввод');
      End;
  Until Cond;
  GetNumberInt:=Number;
End { GetNumberInt };
```

Функция *GetNumberInt* производит проверку формата вводимого числа и диапазона его представления. При обнаружении ошибки пользователю предлагается повторить ввод.

3. Проверка границ: {$R+} или {$R-} .
По умолчанию принято {$R-}. Тип директивы - локальный.

Директива приводит в действие или отменяет генерирование машинных команд для проверки границ. При значении R+ все массивы и строки проверяются на условие принадлежности индекса допустимому диапазону его изменения. Проверке подвергаются также операторы присваивания значения ординальным переменным. При выходе индекса за пределы объявленных границ или при попытке присваивания ординальным переменным внедиапазонного значения программа прерывается и генерируется сообщение об ошибке. При включении директивы R+ компилятор добавляет в объектный код программы значительное количество машинных команд, вследствие чего объем программы увеличивается, а ее работа замедляется. Поэтому рекомендуется при отладке программы всегда использовать директиву R+, а после окончания отладки устанавливать значение R-.

Пример 4.

```
{$R+}
Type  Ar = array[1..100] of real;
      string40 = string[40];
Var   k : byte;
      m : integer;
      n : 1..1000;
      R : real;
      ch : char;
      A : Ar;
      S : string40;
Begin
  1) k := 400;      5) m := 400;  k := m;
  2) n := 2000;     6) m := 2000; n := m;
  3) ch := S[50];   7) k := 50;  ch := S[k];
  4) R := A[200];   8) k := 200;  R := A[k];
```

В этом примере в период компиляции будет зафиксировано нарушение границ для первых четырех операторов. При этом будет выдано сообщение "Error 76: Constant out of range". Для остальных операторов нарушение границ будет зафиксировано в период выполнения программы. При этом будет выдано сообщение "Error 201: Range check error".

4. Проверка переполнения стека: {$S+} или {$S-} .

По умолчанию принято {$S+}. Тип директивы - локальный.

Директива генерирует или отменяет код проверки переполнения стека. Во включенном состоянии этой директивы перед вызовом каждой процедуры и функции проверяется, достаточно ли места в сегменте стека для размещения ее локальных переменных. Если этого места недостаточно, программа прерывается и генерируется сообщение об ошибке.

5. Размеры выделяемой памяти: {$M Стек,Мин_об,Макс_об} .

Тип директивы - глобальный.

Директива определяет параметры распределения памяти программы. Параметр "Стек" указывает размер сегмента стека и может принимать значения в диапазоне 1024 .. 65520. Последние два параметра указывают минимальный и максимальный объем динамически распределяемой области памяти. Параметр "Мин_об" может быть задан в диапазоне 0 .. ..655360, параметр "Макс_об" - в диапазоне Мин_об .. 655360.

АВТОМАТИЧЕСКАЯ ОПТИМИЗАЦИЯ ПРОГРАММ

В Турбо Паскале выполняется несколько типов оптимизации кода компилируемой программы с целью повышения ее быстродействия и уменьшения объема занимаемой памяти.

1. Свертывание констант.

Если операндами выражения являются константы ординального типа, то выражение вычисляется в период компиляции. То же относится к функциям *abs, sqr, succ, pred, odd, lo, hi, swap,* если их аргументами являются константы ординального типа.

Пример 1.

```
Var  k : integer;
Begin
  k:=abs(-15)+sqr(7)-6*12;
```

Для выполняемой программы записанный в исходном тексте оператор будет заменен оператором *k* := -23;

Если индексом массива является константа или выражение, состоящее из констант, то адрес элемента массива вычисляется во время компиляции. Например, доступ к элементам *a*[*sqr*(7)+2,3*8] и *a*[61,24] будет таким же эффективным, как и доступ к простой переменной.

2. Слияние констант.

Если в одном блоке несколько раз встречается одна и та же строковая константа, то в объектном коде программы компилятор разместит только одну ее копию.

Пример 2.

```
  Writeln('Введите значение ','x =');
  ..............................
  Writeln('Введите значение ','y =');
  ..............................
  Writeln('Введите значение ','z =');
  ..............................
```

Для объектного кода программы это эквивалентно фрагменту

```
  Const  S = 'Введите значение ';
  Begin
    Writeln(S,'x =');
    ..............
    Writeln(S,'y =');
    ..............
    Writeln(S,'z =');
    ..............
```

3. Вычисление по короткой схеме.

Логическое выражение в Турбо Паскале вычисляется по короткой схеме. Это означает, что вычисление такого выражения прекращается, как только его результат становится очевидным.

Пример 3.

```
If (x>0) and (x<100) and (y>1) and (y<x) then ...
While (x<0) or (y>1) or (z>x+y) do ...
```

Если x = -1, то вычисление обоих логических выражений прекращается после вычисления значения истинности первого операнда. В первом случае имеем для всего выражения значение *false*, во втором - значение *true.*

4. Удаление неиспользуемого кода.

Операторы, которые никогда не будут выполняться, в объектный код программы не включаются.

Пусть в исходном тексте программы записан оператор

```
If false then
  y:=sqr(x)+1;
```

Такой оператор при компиляции программы игнорируется.

Используя это свойство компилятора, можно в исходном тексте программы подготовить различные ее варианты, формируемые при повторной компиляции.

Пример 4.

```
Const  KeyPrint = 0;  {   Ключ тестовой печати: }
                      { 0 - тестовая печать отсутствует ; }
Begin                 { 1 - активизируется тестовая печать }
.............................
If KeyPrint>0 then
  Begin
    Печать массива A
  End;
.............................
```

Операторы отладочной печати включаются в объектный код программы, если до ее компиляции было установлено значение константы *KeyPrint* = 1.

5. Эффективная компоновка.

Процедуры и функции, к которым в программе нет ни одного обращения, в объектный код не включаются. Не включаются в объектный код также переменные, которые описаны в разделах ***Var***, но в программе не используются.

Данный тип оптимизации кода имеет особое значение при использовании модулей. Если в программе записана фраза

```
Uses Graph;
```

то в объектный код программы будут переписаны только те процедуры из модуля *Graph*, к которым имеются обращения.

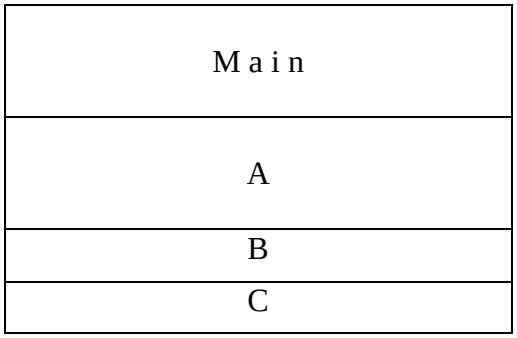
Здесь следует отметить, что при компиляции программы в память эффективная компоновка не работает. Этим объясняется, почему некоторые программы становятся меньше при компиляции их на диск.

ОВЕРЛЕЙНАЯ СТРУКТУРА ПРОГРАММЫ

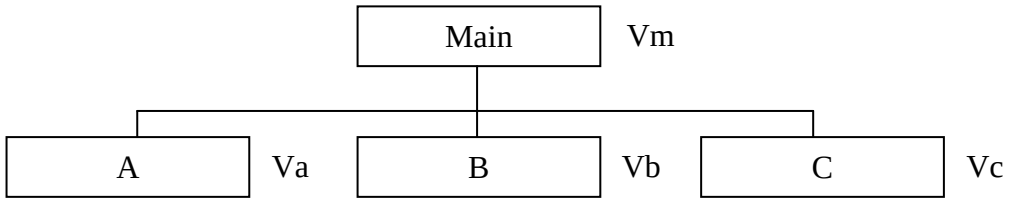
Как уже ранее отмечалось, максимальный размер модуля не может превышать 64 Кбайт, однако количество модулей в программе не ограничивается. Это дает возможность создавать программы, занимающие весь доступный объем оперативной памяти ПЭВМ. В ряде случаев для больших программ этого объема также недостаточно. Тогда программист может использовать механизм оверлеев, позволяющий создавать программы практически неограниченного объема.

Оверлеи представляют собой части программы, которые размещаются в отдельном файле дисковой памяти и в процессе работы программы совместно используют общую область оперативной памяти. В один и тот же момент времени в памяти размещается одна или несколько оверлейных частей программы, а в процессе ее выполнения происходит смена оверлеев. Оверлейные части Паскаль-программы оформляются как отдельные модули.

Предположим, что программа состоит из основной части *Main* и трех модулей *A, B* и *C*, при этом их объем составляет соответственно V_m, V_a, V_b и V_c. Без использования оверлеев для размещения программы требуется $V_m + V_a + V_b + V_c$ байт. Структура такой программы имеет следующий вид:



С использованием оверлеев получим такую структуру:



Для размещения такой программы требуется $V_m + \max(V_a, V_b, V_c)$ байт памяти.

В многомодульной программе содержится основная программа (***Program***), резидентные (неоверлейные) и оверлейные модули. При компиляции такой программы создаются два файла: исполнимый файл с расширением *exe* и оверлейный с расширением *ovr*. При загрузке оверлеев в память они размещаются в оверлейном буфере, расположенном между сегментом стека и динамически распределяемой областью памяти. Размер этого буфера по умолчанию принимается равным размеру наибольшего оверлейного модуля.

Управление оверлеями реализуется в Турбо Паскале с помощью стандартного модуля *Overlay*. При этом в программе пользователя должны быть выполнены следующие требования.

1. Все оверлейные модули должны компилироваться с директивами {$O+, F+}.

Директива {$O+, F+} означает, что данный модуль может быть использован как оверлейный. В связи с этим в программе, имеющей оверлейную структуру, обычно для всех модулей устанавливают такую директиву.

2. В основной программе после предложения ***Uses*** следует в директивах {$O Имя_файла} перечислить имена оверлейных модулей.

3. В основной программе в предложении ***Uses*** имя стандартного модуля *Overlay* должно быть первым.

Инициализация оверлейного файла выполняется процедурой

```
OvrInit(Имя_файла:string),
```

в которой указывается имя файла с расширением *ovr*.

Код результата инициализации сохраняется в предописанной переменной *OvrResult*, которая при успешном завершении работы процедуры *OvrInit* принимает значение 0.

С помощью функции

```
OvrGetBuf : longint
```

можно получить значение размера оверлейного буфера (в байтах).

С помощью процедуры

```
OvrSetBuf(OvrSize:longint)
```

это значение можно увеличить.

Пример.

```
{$O+,F+}
Program OverDemo;
Uses  Overlay,Crt,Unit1,Unit2;
{$O Unit1}
{$O Unit2}
Begin
  OvrInit('OverDemo.ovr');
  If OvrResult<>0 then
    Begin
      Writeln(#7'Оверлейный файл не найден');
      Halt
    End;
  Proc1;
  Proc2;
End.

{O+,F+}
Unit Unit1;
Interface
Uses Crt;
  Procedure Proc1;
Implementation
  Procedure Proc1;
  Begin
    Writeln('Работает модуль Unit1');
  End { Proc1 };
End.

{O+,F+}
Unit Unit2;
Interface
Uses Crt;
  Procedure Proc2;
Implementation
  Procedure Proc2;
  Begin
    Writeln('Работает модуль Unit2');
  End { Proc2 };
End.
```

Так как при компиляции оверлейной программы создаются два файла - с расширением *exe* и с расширением *ovr*, то компиляция такой программы должна осуществляться на диск, а не в память.

Оверлейные модули не должны содержать секции инициализации (в том числе и пустые): при старте программы вначале выполняются секции инициализации всех модулей, что возможно лишь в случае нахождения этих модулей в памяти. Если это условие не выполне-

но, то после запуска программы происходит ее прерывание с выдачей сообщения "Runtime Error 208: Overlay manager not installed" (Не установлена подсистема управления оверлеями).

ИСПОЛЬЗОВАНИЕ СОПРОЦЕССОРА

В сопроцессоре аппаратно реализуется выполнение операций с плавающей запятой, при его наличии в ПЭВМ можно значительно увеличить скорость вычислений и повысить их точность. Для использования сопроцессора программа в самом начале должна иметь директиву компилятора {$N+}:

Сопроцессор предоставляет возможность использовать дополнительные типы вещественных данных. Полный список вещественных типов:

Длина, байт	Название типа	Количество значащих цифр	Диапазон десятичного порядка
4	Single	7 .. 8	-45 .. +38
6	Real	11 .. 12	-39 .. +38
8	Double	15 .. 16	-324 .. +308
10	Extended	19 .. 20	-4951 .. +4932
8	Comp	19 .. 20	$-2^{63} - 1 .. 2^{63} - 1$

Введем следующие обозначения для составных частей вещественных типов данных:

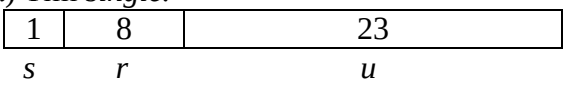
s - знак числа;

u - мантисса;

r - характеристика.

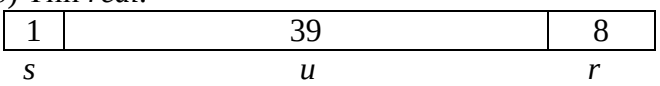
Тогда машинное представление вещественных типов имеет вид:

а) Тип *single.*

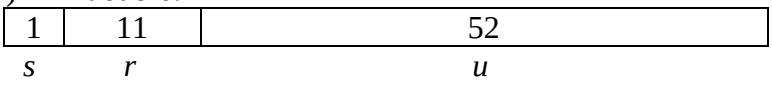


Здесь 1,8,23 - количество бит, отводимых соответственно для знака, характеристики и мантиссы числа.

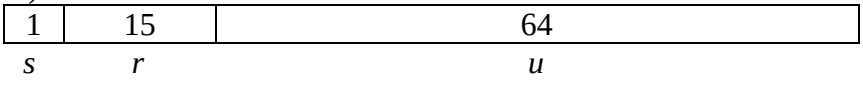
б) Тип *real.*



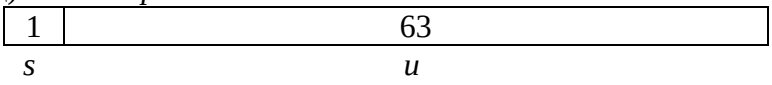
в) Тип *double.*



г) Тип *extended.*



д) Тип *comp.*



Тип *comp* - это фактически длинное целое число, но в программе оно рассматривается как вещественное число с нулевым порядком и мантиссой, в которой записывается только целая часть числа. Обычно тип *comp* используется в бухгалтерских расчетах.

Тип *real* по структуре отличается от типов *single, double, extended*: в *real* мантисса находится перед характеристикой, в остальных типах - после характеристики.

Непосредственно в сопроцессоре операции выполняются только для типа *extended.* Остальные вещественные типы при обращении к сопроцессору преобразуются в тип *extended,* при получении результата от сопроцессора производится обратное преобразование. Преобразования для типов *single* и *double* выполняются просто, так как последовательность составных частей числа здесь одинакова. Для типа *real* необходимо переставить местами мантиссу и характеристику, что требует дополнительного машинного времени.

Турбо Паскаль позволяет работать также в режиме эмуляции сопроцессора. В этом случае операции над вещественными числами при отсутствии сопроцессора выполняются программным путем.

Для включения режима эмуляции перед заголовком основной программы нужно поставить директиву компилятора {$E+}.

Директива {$N+, E+} означает, что программа работает с сопроцессором, если он имеется, и в режиме эмуляции при его отсутствии. Наличие в машине сопроцессора определяется автоматически.

Следует отметить, что тип *real* оптимизирован для работы без сопроцессора. При наличии сопроцессора дополнительное машинное время, затрачиваемое на перестановку мантиссы и порядка числа, сводит на нет преимущества сопроцессора; при этом время выполнения операций для типа *real* может в 2-3 раза превышать время выполнения операций для других вещественных типов. Поэтому при использовании сопроцессора, а современные компьютеры практически всегда обеспечивают аппаратную обработку чисел с плавающей запятой, рекомендуется применять лишь типы *single, double, extended* .

Если в разработанной программе везде объявлен тип *real*, то для перехода, например, к типу *double* достаточно в начале программы установить глобальное объявление

```
Type  real = double .
```

Сопроцессор содержит внутренний стек вычислений, имеющий максимальную глубину восемь уровней. Доступ к переменной, находящейся в стеке сопроцессора, осуществляется значительно быстрее, чем доступ к переменной, находящейся в оперативной памяти. Поэтому для достижения максимальной производительности стек сопроцессора используется для хранения промежуточных результатов вычислений и для передачи параметров процедурам и функциям.

Если для вычисления в сопроцессоре длинного выражения требуется запомнить свыше восьми промежуточных результатов, то возникает переполнение его стека. Такая же ситуация имеет место, когда при обращении к процедуре или функции требуется передать свыше восьми параметров-значений (без слова ***Var***) типов *single, double, extended* или *comp*. При переполнении стека сопроцессора сообщение об ошибке не генерируется, но результат вычислений будет неправильным. Для исключения такой ситуации длинные арифметические выражения следует разделить на части, а в заголовках соответствующих процедур или функций часть параметров-значений преобразовать в параметры-переменные или параметры-константы, добавив к ним соответственно слово ***Var*** или слово ***Const.***

Л И Т Е Р А Т У Р А

1. Джонс Ж., Харроу К. Решение задач в системе Турбо Паскаль. М.: Финансы и статистика, 1991. - 720 с.
2. Поляков Д.Б., Круглов И.Ю. Программирование в среде Турбо Паскаль. М.: Изд-во МАИ, 1992. - 576 с.
3. Немнюгин С.А. Turbo Pascal. – СПб: «Питер», 2001. – 496 с.
4. Немнюгин С.А. Turbo Pascal: практикум. – СПб: «Питер», 2001. – 256 с.
5. Турбо Паскаль 7.0. - К.: Изд.группа BHV, 1996. - 448 с.
6. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс. Учебное пособие. - М.: "Нолидж", 1997. - 616 с.
7. Фаронов В.В. Турбо Паскаль 7.0. Практика программирования. - М.: "Нолидж", 1997. - 432 с.
8. Марченко А.И., Марченко Л.А. Программирование в среде Turbo Pascal 7.0. - К.: ВЕК+, М.: Бином Универсал, 1998. - 496 с.
9. Милов А.В. Основы программирования в задачах и примерах. – Харьков: Фолио, 2002. – 397 с.
10. Программирование на языке Паскаль: задачник / под ред. Усковой О.Ф. – СПб.: Питер, 2003. – 336 с.
11. Ускова О.Ф. и др. Программирование алгоритмов обработки данных. – СПб.: БХВ-Петербург, 2003. – 192 с.
12. Вирт Н. Алгоритмы и структуры данных. – М.: мир, 1989. – 360 с.
13. Фигурнов В.Э. IBM PC для пользователя. - М.: Финансы и статистика, 1994. - 368 с.
14. Богумирский Б. Эффективная работа на IBM PC. - СПб: Питер, 1995. - 688 с.