ГОУ ВПО
КЕМЕРОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ ИНСТИТУТ
ПИЩЕВОЙ ПРОМЫШЛЕННОСТИ

Г.И. Станевко, Т.Г. Колесникова, В.А. Давыденко

ИНФОРМАТИКА:
ПРОГРАММИРОВАНИЕ ОТ ОСНОВ
К ТУРБО ПАСКАЛЮ

Учебное пособие

Кемерово 2011

УДК
ББК
С 76

Рецензенты:
В.Я. Карташов, доктор техн. наук, профессор;
В.С. Черкасов, канд. физ.-мат. наук, доцент.

Рекомендовано редакционно-издательским советом
Кемеровского технологического института пищевой
промышленности

Станевко Г.И.
С 76 Информатика: программирование от основ к Турбо Паскалю / Г.И. Станевко, Т.Г. Колесникова, В.А. Давыденко. – Кемерово: КемТИПП, 2011. – 118 с.
ISBN

В предлагаемом пособии излагаются основы программирования. Приводятся практические рекомендации по разработке программ. Рассмотрено большое количество примеров.

Предназначено для студентов всех специальностей инженерно-технических вузов.

Табл. – ___, библ. назв. - ___.

УДК
ББК

ISBN

СОДЕРЖАНИЕ

ЧАСТЬ 1. ОСНОВЫ ПРОГРАММИРОВАНИЯ

- **Переменная**
- **Классы переменных**
- **Типы переменных**
- **Оператор**
- **Алгоритм**
- **Программа**
- **Подпрограмма**
- **Классификация вычислительных процессов**

◆ Переменная

Фундаментальным понятием программирования является понятие переменной. Это понятие неразрывно связано со значениями данных решаемой задачи и оперативной памятью компьютера, которая состоит из пронумерованных ячеек. Поэтому, если воспринимать имя переменной как номер ячейки, то ***переменная – это поименованная область памяти, хранящая значение этой переменной в конкретный момент времени.***

Значение переменной сохраняется в ячейке до тех пор, пока в эту ячейку не поступит новое значение. Условимся процесс засылки значения в ячейку записывать в виде:

<имя переменной>:= <значение>;

В программировании эта структура называется ***оператором присваивания***. Совокупность символов **:=** означает *присвоить, положить равным,* а для ячеек – *заслать.*

Например, *X:= 2* следует читать:

- переменной *X присвоить* значение, равное *2*;

- *X положить равным 2*;
- в ячейку *X заслать 2*.

В отличие от математической формулы, оператор присваивания выражает краткую запись действий компьютера над содержимым ячеек.

Благодаря этому, записи вида:

$$X := X + A; \quad (1)$$
$$X := X * A; \quad (2)$$
$$X := X / A; \quad (3)$$

и т.д. *неверные в математике*, являются *верными и весьма популярными в программировании.* В переводе на ячейки памяти ЭВМ это означает, что соответствующее действие выполняется над числом *A* и числом, находящимся в ячейке *X перед началом действия и во время его выполнения*, а *при его завершении* в ячейку *X* записывается *новое число***,** являющееся *уже результатом***.**

Например, последовательный счет по формулам:

$$X := -2; \quad X := X + A; \quad X := X + C;$$

при $C = 4$ и $A = 3$

изменяет содержимое ячейки с именем ***X***:

$$X = \mathbf{-2}; \quad X = \mathbf{-2} + 3 = \mathbf{1}; \quad X = \mathbf{1} + 4 = \mathbf{5}$$

и завершается записью в неё числа ***5***.

Значения переменной X, стоящие в правых частях формул (1), (2), (3), называются её предыдущими значениями, а в левых – последующими. Таким образом, в формуле (1) последующее значение переменной *X* равно сумме ее предыдущего значения и значения переменной *A*; в формуле (2) – произведению ее предыдущего значения и значения переменной *A*; в

формуле (3) – частному от деления ее предыдущего значения на значение переменной *A*. Повторяя счет по формулам:

$$X := X + 1; \qquad P := P * X;$$

при начальном значении $X = 0, P = 1$ будем получать последовательно:

$$X = \mathbf{0} + 1 = \mathbf{1}; \quad P = \mathbf{1} * 1 = \mathbf{1};$$
$$X = \mathbf{1} + 1 = \mathbf{2}; \quad P = \mathbf{1} * 2 = \mathbf{2};$$
$$X = \mathbf{2} + 1 = \mathbf{3}; \quad P = \mathbf{2} * 3 = \mathbf{6};$$
$$X = \mathbf{3} + 1 = \mathbf{4}; \quad P = \mathbf{6} * 4 = \mathbf{24}.$$

Если мы по своей рассеянности забыли в процессе решения задачи *задать конкретное значение какой-то переменной*, то ЭВМ вправе присвоить ей *свое значение*, чаще нулевое, *и попытаться довести решение задачи до конца.* Такие переменные в программировании называются ***неопределенными***. Появление неопределенных переменных при решении задач на ЭВМ приводит либо к *неверным результатам*, либо *к невыполнимым действиям.* При работе с числовой информацией к невыполнимым действиям относятся: деление на нуль, извлечение квадратного корня из отрицательного числа, взятие логарифма от числа отрицательного или равного нулю и т.д. При обнаружении таких ситуаций компьютер прекращает решать задачу, т.е. завершает работу аварийным остановом.

Имя переменной. В программировании приняты свои правила образования имён. В отличие от математических обозначений, в которых могут использоваться буквы любого алфавита, в программировании используется только *латинский алфавит.*

Имя переменной представляет собой идентификатор, т.е. ***последовательность букв, цифр и знака подчёркивания, начинающуюся с буквы или знака подчёркивания.***

Таким образом, имена

S, RY, x5, v6M45, _Turbo_Pascal – *идентификаторы*, а

8R, X 24, AH-24 – идентификаторами *не являются*, так как:

- первое начинается с цифры;
- второе содержит пробел между *X* и *24*;
- третье записано через тире.

При решении конкретных задач мы можем:

- обозначать переменные общепринятыми именами, например, путь через *S*, скорость через *V*, а время через *t*; длины сторон треугольника – через *A, B, C;*
- связывать имена со смысловыми значениями переменных решаемой задачи, так, переменную, являющуюся значением интеграла, назвать *Integral*, периметр треугольника назвать составным именем *Perim_Tr*, а многоугольника – *Perim_Mn*;
- или выбирать имена по своему желанию.

Длина идентификатора. ***Количество символов в идентификаторе, воспринимаемое компьютером, называется его длиной.*** На длину идентификатора в каждом языке программирования установлены ограничения. Так, максимальная длина идентификатора в Turbo Pascal версии 5.0 равна 8, а версии 7.0 – 63. Если в идентификаторе больше символов, чем допускается конкретным языком, то символы, выходящие за пределы допустимой длины, компьютером не учитываются.

◆ Классы переменных

Переменные могут существовать *самостоятельно* или объединяться в *группы под общим именем.*

Самостоятельно существующие переменные называются простыми. Простые переменные обозначаются просто идентификаторами.

Упорядоченные группы переменных называются массивами. Все элементы массива имеют *одно имя* и различаются только *порядковыми номерами* согласно занимаемому месту.

Порядковые номера называются индексными выражениями или просто индексами, а элементы массивов – индексными переменными.

Встает вопрос: как же ЭВМ различает простые и индексные переменные? Она различает их только *по написанию*:

- если в записи переменной указано *только имя*, то для ЭВМ это *простая переменная.* Например, *Y2* – читается: *игрек два;*
- если запись состоит из *имени и скобок [ ],* внутри которых указано индексное выражение (число, переменная или арифметическое выражение), то это *индексная переменная.* Например, *Y[2]* – читается: *игрек вторая* или как *второй элемент массива Y.*

Такая классификация весьма существенна. Она позволяет по-разному размещать переменные в памяти компьютера, по-разному записывать их в программах и манипулировать с ними при решении задач.

Наглядно группу элементов можно представить в виде одной строки или одного столбца, или в виде матрицы, состоящей из нескольких строк и столбцов.

В программировании группа–столбец или группа–строка называется одномерным массивом. Матрица представляет двумерный массив. В некоторых языках программирования массивы могут иметь размерность больше двух.

Для выбора элемента из одномерного массива достаточно указать в качестве индекса только один номер, например, *T[i]*. Для выбора элемента из двумерного массива – два номера: номер строки и номер столбца, на пересечении которых находится требуемый элемент, например, *Z[i, j]*, где *i* – номер строки, *j* – номер столбца.

Описание размерности массива. Это понятие включает в себя не только информацию *о количестве индексов* (индексных выражений) в записи индексной переменной, но также и *допустимые границы изменения индексов,* которые записываются в скобках за именем массива. Например, нам известно, что для решаемого класса задач одномерный массив *X*, будет содержать

не более *23* элементов. Описание размерности этого массива мы можем задать структурой *X[1..23]*. Она содержит информацию о том, что:

- элементы имеют сквозную нумерацию от *X[1]* до *X[23]*;
- индекс *i* переменной *X[i]* подчиняется условию $1 \le i \le 23$.

В описании размерности двумерного массива допустимые границы изменения индексов записываются через запятую. Например, структура *Z[1..12, 1..3]* указывает нам на то, что:

- массив *Z* двумерный;
- может содержать не более *12* строк и *3* столбцов;
- для индексной переменной *Z[i, j]* индексы могут изменяться только в пределах: $1 \le i \le 12; 1 \le j \le 3$.

В некоторых языках программирования индексы могут иметь нулевые и даже отрицательные значения.

Правила образования индексов. Образование индексов подчиняется следующим правилам:

- индексное выражение (индекс) может быть числом, переменной или выражением. Так, в записях *a[4], a[i], a[k[3]], a[i+5]* индекс *4* – число, *i* – простая переменная, *k[3]* – индексная переменная, *i+5* – выражение;
- индексное выражение всегда имеет *целое значение*;
- численные значения индекса не должны выходить за пределы, определенные описанием размерности массива. Выход индекса за границы приводит к неопределенным переменным и сопровождается соответствующим сообщением системы программирования.

Размещение переменных в памяти компьютера. Для *простых переменных* компьютер отводит свободные *ячейки в любом месте памяти*, предназначенной для переменных (переменные и программа занимают разные участки памяти).

Под *индексные переменные*, относящиеся к массиву с конкретным именем и указанным *N* (количеством элементов этого массива), компьютер отводит *только группу подряд расположенных N ячеек.* В первой ячейке этой группы находится значение первого элемента данного массива, во второй – второго и т.д. по порядку. Если требуемого количества подряд расположенных ячеек нет, то выдаётся сообщение о нехватке памяти для размещения этого массива.

Считывание из памяти и запись в память значений переменных при выполнении действий. При выполнении действий над *простыми переменными* компьютер обращается непосредственно к соответствующим ячейкам по именам переменных. При работе с *индексными переменными* компьютер проделывает следующую работу:

- по имени массива отыскивает начало соответствующего ему массива ячеек;
- отсчитывает от найденного начала количество ячеек, равное численному значению индекса, определяя тем самым номер ячейки, соответствующей данной индексной переменной;
- и только тогда считывает её содержимое для выполнения действия, или записывает в эту ячейку результат.

Организация классов переменных. Вопрос, к какому классу отнести переменные задачи, решается отдельно в каждом конкретном случае и зависит от многих условий. Например, при решении квадратного уравнения

$$ax^2 + bx + c = 0$$

нет необходимости в использовании индексных переменных, а при решении задачи о средней заработной плате рабочих завода использование только простых переменных просто немыслимо.

Заметим, что удачная классификация переменных позволяет сокращать длину программ, экономя тем самым время на их написание и набор.

◆ Типы переменных

При решении задач возникает необходимость выполнять обработку информации различного характера. Это могут быть числа (целые и дробные), строки (наборы символов и даже предложения) и т.д.

Так в задаче, связанной с выбором наименьшего времени обработки N различных деталей, количество деталей может быть только *целым числом*; время обработки может быть *вещественным* (целым, дробным или смешанным числом); наименование детали – *символьным*.

Для каждой группы переменных в программировании установлены присущие только ей операции. Над символьными переменными не выполняются операции, присущие группе числовых переменных (умножение, возведение в степень и т.д.), а операции над целыми числовыми переменными выполняются в программировании по правилам, отличным от операций над вещественными. Следовательно, ***тип можно определить как характеристику, объединяющую переменные по совокупности выполняемых над ними операций.***

◆ Оператор

Оператором называется инструкция по выполнению компьютером некоторого действия. Набор операторов и их структуры определяются конкретным языком программирования. Однако любой современный язык содержит средства, позволяющие выполнять:

- ввод исходных данных в память компьютера;
- вывод результатов из памяти;
- присваивания переменной результата выполнения некоторого действия;
- выбор направления вычислительного процесса по условию;

- повторное выполнение некоторого участка программы требуемое количество раз;
- вызов прикладного стандартного или разработанного Вами программного обеспечения.

◆ Алгоритм

Решение любой задачи связано с исходными данными и результатами.

Под алгоритмом понимается последовательность действий, приводящая от исходных данных к результату.

При разработке алгоритмов всегда исходят из принципа *определенности переменных*, который заключается в том, что ***к моменту использования переменной она должна быть определена, т. е. иметь конкретное значение!***

Уважающий себя программист, прежде чем сесть за компьютер, непременно продумывает алгоритм решения задачи, описывает его словесно или использует какой-то другой наглядный и обозреваемый способ.

При словесном описании алгоритма используется обычный разговорный язык, элементы алгоритма нумеруются. Ниже приведен пример словесного описания алгоритма нахождения наибольшего из чисел *A, B, C*:

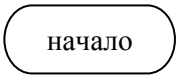
Начало
1. Введем значения переменных A, B, C.
2. Если A > B, то идём к пункту 3 , иначе идём к пункту 4.
3. Если A > C, то печатаем значение A и идём на конец.
4. Если B > C, то печатаем значение B, иначе печатаем значение C.
Конец

Одним из наглядных способов является *графический.* В нём имеется небольшой стандартный набор геометрических фигур, *закрепленных за конкретными действиями* или *операторами.*

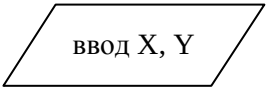
На размер фигур установлен ГОСТ, который надо соблюдать в публикациях, документах и т.д. Мы же позволим размер фигур выбирать произвольно.

- **Элементы блок-схем**

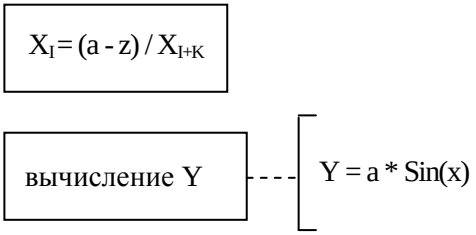
Элемент пуска / останова определяет начало / конец процесса выполнения программы. Если элемент соответствует началу алгоритма, то внутри записывается слово ‘***начало***’, для конца – слово ‘***конец***’.



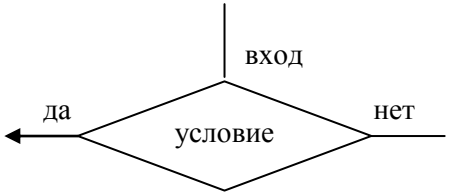
Элемент ввода – вывода соответствует вводу исходных данных решаемой задачи и выводу результатов. Для ввода в него записывается слово ‘***ввод***’ и перечисляются переменные, значения которых подлежат вводу, т.е. записывается *список* ввода. При выводе – слово ‘***вывод***’ или ‘***печать***’ и список вывода.



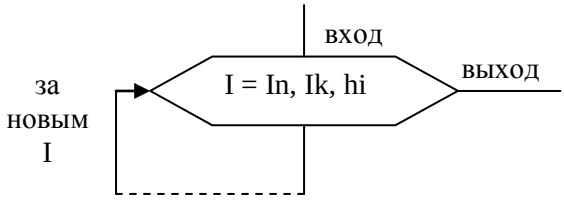
Элемент оператора присваивания содержит внутри запись *формулы* или слово ‘***вычисление***’, а далее имя переменной, являющейся результатом выполнения данного оператора, рядом в виде комментария записывается формула.



Элемент выбора дальнейшего направления решения. Выбор осуществляется в зависимости от выполнения условия, записанного внутри элемента. Если условие *выполняется* (ответом на вопрос условия является слово ‘да’), то продолжение вычислительного процесса осуществляется по ветке ‘***да***’. В противном случае – по ветке ‘***нет***’.



Элемент, управляющий повторным выполнением участка программы (счетный цикл), имеет структуру шестиугольника, внутри которого записывается *переменная, управляющая процессом повторения – параметр цикла*, её начальное, конечное значения и шаг изменения.



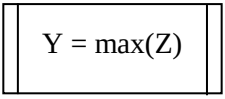
В приведённом элементе это: *I, In, Ik, hi.* Повторяющийся участок программы выполняется для каждого значения *I* от *In* до *Ik* при изменении *I* по закону:

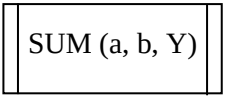
- *I:= I + hi* (каждое последующее значение переменной *I* равно её предыдущему значению, увеличенному на значение шага *hi*).

- *I:= I – hi* (каждое последующее значение переменной *I* равно её предыдущему значению, уменьшенному на значение шага *hi*).

При выходе из этого цикла, значение переменной *I* – параметра цикла, становится равным *Ik.*

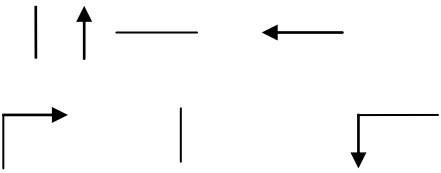
Элемент вызова процедур и функций. В поле этого элемента для функций записывается формула, содержащая функцию; для процедур – оператор обращения к процедуре.





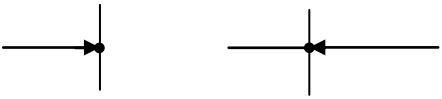
➢ **Соединение элементов в блок-схемах**

Блоки соединяются в схему ***линиями потока (отрезками горизонтальных и вертикальных прямых).***



Направления линий потока слева – направо и сверху – вниз считаются естественными, а противоположные направления – неестественными, и их окончание положено фиксировать стрелочками.

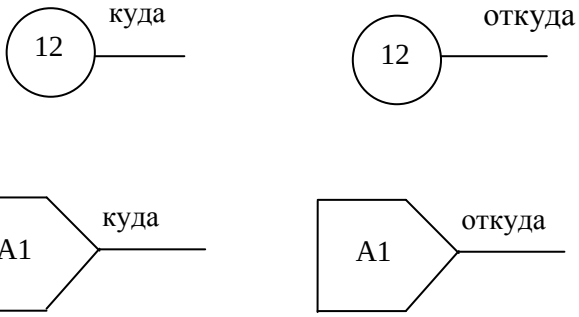
Линии потока могут пересекаться и образовывать **слияние.** Место слияния фиксируется точкой.



Часто из-за пересечения линий потока блок-схемы теряют наглядность. При такой ситуации лучше их разрывать, а для

условного соединения пользоваться ***соединителями***, расположенными в начале и в конце разрыва линии потока.

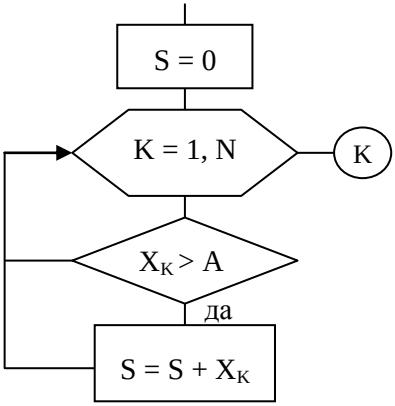
Разрывы изображаются фигурами: страничный – кругом, межстраничный – открытым конвертом. Внутри фигур, относящихся к одной и той же линии, *записываются одинаковые метки*: буквы, числа, символы или их сочетания, по Вашему желанию.

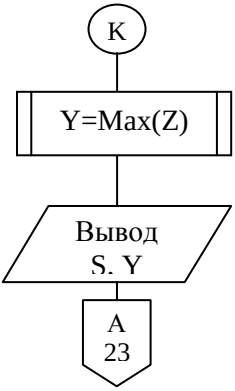


В межстраничных соединителях можно указывать две метки, взяв в качестве второй для соединителя, расположенного в начале разрыва, номер страницы, на которой следует искать *продолжение* данной линии, а для соединителя в конце – номер страницы, на которой надо искать *начало* данной линии.

При желании все блоки схемы можно пронумеровать сквозными номерами, проставив их слева в разрывах верхних линий. Блоки начала и конца не нумеруются.

Пример начертания фрагмента блок-схемы





➢ **Свойства алгоритма**

Однозначность алгоритма – единственность толкования исполнителем правил и порядка выполнения действий.

Конечность алгоритма – обязательность завершения каждого действия, составляющего алгоритм, и завершимость алгоритма в целом.

Результативность алгоритма предполагает, что выполнение алгоритма должно завершиться получением определённых результатов.

Массовость – возможность применения данного алгоритма для решения целого класса задач. Для обеспечения этого свойства следует составлять алгоритмы в *обозначениях переменных*, избегая конкретных значений.

Правильность алгоритма – способность алгоритма давать правильные результаты решения поставленных задач.

При разработке алгоритма необходимо обеспечить выполнение его свойств.

◆ Программа

Программа – это запись алгоритма на языке программирования. Изящество, краткость и прочие качества программы в значительной мере зависят от классификации переменных, т.е. от того, какие переменные Вы отнесли к классу *простых*, а какие объединили *в массивы*. Поэтому прежде чем сесть за компьютер, следует серьезно потрудиться карандашом!

Любая программа должна иметь хороший ***интерфейс – среду для общения с пользователем***. Это требование может быть удовлетворено организацией вывода *сообщений* или *запросов* в процессе выполнения программы. Кроме этого, программа должна *хорошо читаться*, т.е. быть понятной для человека, решившего её изучить или проверить на предмет соответствия алгоритму. Это достигается за счёт *комментариев*, сопровождающих отдельные участки программ и расчленением программы на отдельные фрагменты – подпрограммы.

Комментарии записываются *обычным разговорным языком и являются невыполняемыми* структурами программы.

◆ Подпрограмма

Подпрограммы – это мини-программы, имеющие ту же структуру, которой обладает и основная программа. Вызов подпрограмм осуществляется по *их именам*. При их вызове выполнение основной программы *приостанавливается*, и *управление передаётся в подпрограмму*. *По окончании работы подпрограммы управление возвращается основной программе*. Преимущество такой организации программ заключается в том, что, во-первых, один и тот же фрагмент можно использовать многократно в одной или даже в нескольких программах. Во-вторых, такие программы легче читаются, легче отлаживаются, у них более чёткая логическая структура.

◆ Классификация вычислительных процессов

Отдельные участки программ могут быть охарактеризованы как:

- линейные;
- разветвляющиеся;
- циклические.

В линейном участке (процессе) операторы выполняются один за другим в естественном порядке.

В разветвляющемся – происходит выбор дальнейшего направления в зависимости от условия.

В циклическом – осуществляется повторное выполнение некоторого участка программы до выполнения некоторого условия.

Любой цикл содержит две части:

- главную;
- управляющую.

Главная часть называется телом и включает основные действия решения требуемой проблемы.

Управляющая часть ***содержит информацию,*** определяющую повторение или окончание выполнения тела цикла.

ЧАСТЬ 2. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ТУРБО ПАСКАЛЕ

Турбо Паскаль относится к языкам высокого уровня. Он имеет мощное и гибкое математическое обеспечение. Кроме алфавита, набора зарезервированных слов и операторов, в его состав входит набор стандартных подпрограмм (процедур, функций) и модулей.

Стандартные подпрограммы – это специальным образом оформленные встроенные в язык программные участки, оформленные в виде процедур и функций. В них реализованы алгоритмы решения ряда задач: вычисление значений элементарных математических функций, организация ввода – вывода данных и т.д. Обращаясь к ним по их зарезервированным именам, можно построить свою программу как здание из готовых блоков.

Стандартные модули представляют тематические библиотеки, в которых реализованы процедуры и функции решения задач по конкретным темам. Так, модуль **Crt** содержит средства управления дисплеем и клавиатурой; модуль **Graph** содержит пакет графических средств; модуль **System** – сердце языка, содержащиеся в нём подпрограммы обеспечивают работу всех остальных модулей системы. В составе языка семь стандартных модулей. Подробную информацию о стандартном математическом и программном обеспечении можно получить в [1] – [8].

Глава 1. Алфавит

- **Символы, используемые в идентификаторах**
- **Специальные символы**
- **Символы, используемые в строках и комментариях**

Совокупность допустимых в языке символов образует алфавит языка.

Все компоненты языка формируются из множества символов стандарта ASCII, которые задаются кодами от 0 до 255 *(Приложение 1)*.

◆ Символы, используемые в идентификаторах

Идентификаторы (имена) формируются из букв, цифр и знака подчёркивания.

В качестве букв используются только строчные и прописные буквы латинского алфавита, от **a** до **z** и от **A** до **Z**.

В программах прописные и строчные буквы не различаются, например, слова **END, End, end** трактуются как одно слово. Отметим, что в идентификаторах нельзя использовать буквы кириллицы.

Цифрами могут быть только арабские цифры от **0** до **9**. В идентификаторах они могут занимать любую позицию, кроме первой.

Знак подчёркивания в идентификаторах заменяет пробел. Он может занимать любую позицию, включая и первую, например, *_MySum, Prim_23*.

Длина идентификатора может быть любой, но значимы только первые 63 символа.

Идентификаторы подразделяются на зарезервированные слова, стандартные идентификаторы и идентификаторы пользователя.

Зарезервированные слова и стандартные идентификаторы составляют основу языка и имеют строго фиксированное начер-

тание. Программист не имеет права их изменять или использовать не по назначению. Мы будем знакомиться с ними по ходу изложения конкретных структур языка. Идентификаторы пользователя определяются им самим.

◆ Специальные символы

Список специальных символов составляют символы:

+ – * / = < > [ ] . , ( ) ; : { } _ ^ @ $ # и **пробел**.

Пробел отображает пустое место. Его можно использовать для отделения друг от друга зарезервированных слов, идентификаторов, чисел, частей программ и т.д.

Комбинации специальных символов образуют составные символы:

<> не равно <= меньше или равно

>= больше или равно := присвоить.

В программе эти пары символов нельзя разделять пробелами.

◆ Символы, используемые в строках и комментариях

Строки символов в программах и комментарии могут записываться любыми символами и любыми алфавитами. Мы, естественно, будем пользоваться русским языком.

Если строка символов представляет в программе значение переменной, то её следует заключать в апострофы, например *Name:= ‘Юрий’*.

Комментарий – это пояснительный текст к различным участкам программы. Он заключается в фигурные скобки { } и может занимать любое количество строк.

Глава 2. Данные

- **Простые типы данных**
- **Константы и переменные**
- **Числа**
- **Выражения и операции**
- **Ещё о простых типах**
- **Строки**
- **Структурированные типы**
- **Математические функции и процедуры**
- **Ввод – вывод**

◆ Простые типы данных

Основной (стандартный) набор простых типов, т.е. определяющих тип только одного значения, приведён в таблице 2.1.

Для описания каждого стандартного типа закреплён свой идентификатор.

Таблица 2.1

Простые типы данных

Название типа	Зарезервированное слово	Диапазон значений
Целый	Byte Integer	0..255 -32768..32767
Логический	Boolean	True (истина), False (ложь)
Символьный	Char	Символы кодовой таблицы ASCII
Вещественный	Real	$2.9 \cdot 10^{-39}..1.7 \cdot 10^{38}$

Рассматриваемые типы, кроме вещественного, принято называть порядковыми.

Пользователь может образовывать собственные нестандартные типы данных и давать им произвольные имена. Для объявления нестандартных типов используется зарезервированное слово **Type**.

Например:

```
Type   TM1_r = array[1..10] of real;
       TRec = record
                        a: real;
                        s: char;
              end;
       TS = string[8];
```

◆ Константы и переменные

Все данные программы интерпретируются как константы или переменные. ***Константы не изменяют своего значения до завершения работы программы*** и объявляются зарезервированным словом **Const**, за которым следует список имён констант, каждому из которых с помощью символа « = » присваивается значение:

```
Const
    My_Stature = 190;
    My_Name = ‘Willi’;
```

Значение констант можно задавать выражением:

```
Const
    C = -54;
    D = 324;
    Xmin = D – C;
    Xmax = D + C;
```

Кроме обычных констант, можно применять так называемые *типизированные константы*, значения которых можно менять. При их объявлении дополнительно указывается тип:

```
Const
    Ind: byte = 23;
    A: integer = 687;
```

Переменные в отличие от констант могут менять своё значение неограниченное число раз. Для их объявления используется зарезервированное слово **Var**, за которым следуют идентификаторы переменных и через двоеточие указывается их тип. Каждая группа переменных отделяется от другой точкой с запятой.

Например:

```
Var
    num, ind, i, k: integer;
    Amin, Amax: TM1_r;
```

◆ Числа

Целые числа записываются в обычной десятичной форме: 45, -34.

Дробные числа имеют две формы записи: обычную десятичную и научную (экспоненциальную).

В обычной десятичной форме целая часть отделяется от дробной точкой: 2.3098; -0.67; -3.005.

Запись числа в научной форме значительно отличается от его математической записи. Во-первых, число записывается в строку, во-вторых, вместо основания 10 записывается буква E.

Например, числа $5.678 \cdot 10^3$; $-342 \cdot 10^{-2}$ можно записать в виде *5.678E+3; 56.78E+2; -342E-2.*

◆ Выражения и операции

Переменные и константы всех типов используются в выражениях.

Выражение задает порядок выполнения действий над элементами данных и состоит из операндов (констант, переменных, обращений к функциям), круглых скобок и знаков операций.

Операции определяют действия, которые надо выполнить над операндами. В простейшем случае выражение может состо-

ять из одной переменной или константы. Операции подразделяются на: арифметические операции, операции отношения, логические (булевские) операции, операции со строками. По типу операндов и характеру операций выражения тоже называются арифметическими, отношений, булевскими, строковыми. Рассмотрим основные операции, без которых не обходится практически ни одна программа.

Арифметические операции выполняют арифметические действия в выражениях над значениями операндов целочисленных и вещественных типов (таблица 2.2).

Таблица 2.2

Арифметические операции

Операция	Действие	Тип операндов	Тип результата
Бинарные:			
+	сложение	целый вещественный	целый вещественный
-	вычитание	целый вещественный	целый вещественный
*	умножение	целый вещественный	целый вещественный
/	деление	целый вещественный	вещественный вещественный
div	целочисленное деление	целый	целый
mod	остаток	целый	целый
Унарные:			
+	сохранение знака	целый вещественный	целый вещественный
-	отрицание знака	целый вещественный	целый вещественный

Операции отношения выполняют сравнение двух операндов и определяют, истинно значение выражения или ложно (таблица 2.3).

Сравниваемые величины могут принадлежать к любому типу данных. Результат всегда имеет булевский тип.

Таблица 2.3

Операции отношения

Операция	Действие	Выражение	Результат
=	равно	A = B	True, если A равно B
<>	не равно	A <> B	True, если A не равно B
>	больше	A > B	True, если A больше B
<	меньше	A < B	True, если A меньше B
>=	больше или равно	A >= B	True, если A больше или равно B
<=	меньше или равно	A <= B	True, если A меньше или равно B
in	принадлежность	A in M	True, если A есть в списке M

Логические операции выполняются над операндами типа B**oolean.** Результатом выполнения логических (булевских) операций является логическое значение **True** или **False** (таблица 2.4).

Таблица 2.4

Логические операции

Операция	Действие	Выражение	A	B	Результат
not	Логическое отрицание	not A	True False		False True
and	Логическое И	A and B	True True False False	True False True False	True False False False
or	Логическое ИЛИ	A or B	True True False False	True False True False	True True True False
xor	Исключающее ИЛИ	A xor B	True True False False	True False True False	False True True False

Приоритет операций играет существенную роль при записи и выполнении выражений. Выполнение каждой операции происходит с учетом ее *приоритета*. Значения приоритетов указаны в таблице 2.5.

Таблица 2.5

Приоритет выполнения операций

Операция	Приоритет	Вид операции
Not	Первый (высший)	Унарная операция
*, /,div, mod, and	Второй	Операции типа умножения
+, -, or, xor	Третий	Операции типа сложения
=, <, >, <=, >=, in	Четвёртый (низший)	Операции отношения

Операции в выражениях выполняются слева направо с учётом старшинства. Для изменения порядка применяются круглые скобки, в этом случае выполнение операций всегда начинается с них.

```
Var
        x, y, z, t, a, b, c, d: real;
    Begin
        a:= 12;
        b:= 20;
        c:= 6;
        d:= 3;
        x:= b – a / d – c;          {результат x = 10}
        y:= (b – a) / (d – c);      {результат y = -2.666}
    End.
```

◆ Ещё о простых типах

Простые типы являются основой, на которой строятся все другие структуры данных. Кроме стандартных типов, рассмотренных выше, допускаются простые пользовательские типы (типы, *сформированные пользователем*).

К ним относятся *перечислимый* и *интервальный* типы. Данные этих типов занимают в памяти только 1 байт, поэтому любой пользовательский тип не может содержать более 256 элементов. Объявление этих типов начинается служебным словом **Type**.

Перечислимый тип задаётся непосредственно перечислением всех значений, которые может принимать переменная данного типа. Отдельные значения указываются через запятую, а весь список заключается в круглые скобки.

```
Type
      {объявление типов:}
       Day = (‘понедельник’, ‘среда’, ‘пятница’);
        Val = (3, 5, 7, 45);
Var
      {объявление переменных:}
       My_day: Day;
             Ur: Val;
        …
      {операторы программы:}
       My_day:= ‘среда’;
             Ur:= 7;
        …
```

Интервальный тип позволяет задавать две константы, определяющие границы значений для данной переменной. Обе константы должны принадлежать одному из стандартных типов, кроме типа **real**. Значение первой константы должно быть меньше значения второй. Компилятор при каждой операции с переменной интервального типа генерирует подпрограммы проверки, определяющие, остаётся ли значение переменной внутри установленного для неё диапазона.

Например:

```
Const
      Nmin = 1;
      Nmax = 46;
```

```
Type
    Dip = Nmin..Nmax;
Var
    Sdip: Dip;
    ...
    Sdip:= 34;
    Sdip:= 67;              {ошибка: выход из диапазона!}
    …
```

- **Строки**

Строка – это последовательность символов кодовой таблицы ASCII. При использовании в выражениях строка заключается в апострофы. Количество символов в строке (длина строки) может изменяться от 0 до 255. Для объявления данных строкового типа используется зарезервированное слово **string**, за которым следует заключённое в квадратные скобки значение максимально допустимой длины строки данного типа. Если это значение не указано, то по умолчанию длина строки равна 255.

Например:

```
Type
      FioStr = String;        {тип строки длиной не более 255
                               символов}
   PhoneStr = String[10];     {тип строки длиной не более 10
                               символов}
Var
       Fio: FioStr;
     Phone: PhoneStr;
     …
{операторы программы:}
Fio:= ‘Вахрушев Александр Сергеевич’;
Phone:= ‘37-24-55’;          {в строке 8 символов: 6 цифр и 2
                              дефиса}
```

К отдельному символу строки можно обратиться по номеру (индексу) этого символа в строке. Индекс определяется выражением целого типа, которое записывается в квадратных скобках сразу за идентификатором строковой переменной:

Writeln (Fio[6]); {результат = ‘ш’}
Writeln (Phone[2]); {результат = ‘7’}

Нулевой индекс обеспечивает доступ к нулевому байту, который содержит значение текущей длины строки:

Writeln (Fio[0]); {результат = 28}

Подробное описание структур обработки строк изложено в [8].

◆ Структурированные типы

Структурированные типы определяют упорядоченную совокупность простых типов и характеризуются типом своих компонент. Из этих типов рассмотрим тип *массив* и тип *множество*.

Тип массив – это тип данных, состоящий из фиксированного числа элементов, имеющих один и тот же тип. Число элементов массива фиксируется при его описании и остаётся неизменным в процессе выполнения программы. Для описания массива используется зарезервированное слово **array**. Для объявления типа одномерного массива необходимо использовать структуру:

Type
<имя типа> = array [In..Ik] of <тип элементов массива>;

Для объявления типа двумерного массива:

Type
<имя типа> = array [In1..Ik1, In2..Ik2] of <тип элементов массива>;

где *In, In1, In2* – целые числа или целые константы, характеризующие нижние границы индексов;
Ik, Ik1, Ik2 – целые числа или целые константы, характеризующие верхние границы индексов.

Переменные *In, In1, In2, Ik, Ik1, Ik2* могут принимать не только положительные и нулевые значения, но и отрицательные. Условимся значения нижних границ брать равными 1, а верхних – положительными, большими или равными 1.

✓ **Переменная типа массив** может быть описана непосредственно или через ссылку на ранее объявленный тип.

Например:

```
Const
    In = 1;
    Ik = 30;
Type
    {объявление типа одномерного массива:}
    TM1_r = array[1..10] of real;

    {объявление типа двумерного массива, содержащего 2 строки и 30 столбцов}
    TM2_i = array[1..2, In..Ik] of integer;

    {объявление переменных – массивов:}
Var
    {непосредственное объявление:}
    Fio: array[1..5] of string[25];

    {объявление через объявленные ранее типы:}
    X, Y: TM1_r;
    Xm2: TM2_i;
```

Доступ к отдельному элементу массива осуществляется путём индексирования элементов массива. Индексы представляют собой выражения любого простого целого типа и записываются в квадратных скобках сразу же после имени массива:

```
Const
    I = 5;
    K = 7;
    …
```

```
{операторы программы:}
Fio[2]:= 'Толстой Лев Николаевич'; {в операции участвует
                                    второй элемент мас-
                                    сива Fio}
Sm2:= Xm2[2, 15] + 5.67;           {в операции участвует
                                    элемент матрицы, рас-
                                    положенный на пересе-
                                    чении второй строки и
                                    пятнадцатого столбца}
Sm1:= X[i+2] – Y[k-3];             {в операции участву-
                                    ют: седьмой элемент
                                    массива X и четвёртый
                                    элемент массива Y}
```

Иногда удобно применять массивы, описанные как типизированные константы. Элементы таких массивов используются как значения переменных, которые при необходимости можно изменять:

```
Const
    X_M1: array[1..7] of string[2] = ('Ма', 'ри', 'на', '-', 'Ни',
                                      'ки', 'та');
    …
    {операторы программы:}
    X_M1[1]:= 'Ка';
    X_M1[4]:= 'и';     {результат: 'Ка', 'ри', 'на', 'и', 'Ни', 'ки', 'та'}
```

Все действия над массивами выполняются, как правило, с их элементами. Массивы в целом могут участвовать только в операциях отношения «равно», «не равно» и в операторе присваивания. При этом оба массива должны быть объявлены через один и тот же тип:

```
Var
    MasA, MasB: array [1..20] of real;

    {операторы программы:}
    MasA:= MasB;    {копируется поэлементно MasB в MasA}
    MasB:= 0;       {ошибка, так нельзя обращаться к масси-
                     ву в целом}
```

✓ **Тип множество**, так же, как и тип массив, объединяет группу однотипных элементов. В отличие от массивов, *множества* формируются только из *значений порядковых типов.*

Простые типы – **byte** и c**har** являются порядковыми (их элементы можно поштучно назвать), значит, могут служить основой для построения множеств. Если этих типов недостаточно, то можно создать свой порядковый тип.

Например:

```
Type
    VideoAdapterType = (Hercules, AGA, CGA, MCGA, EGA,
                        VGA);
```

и использовать переменную

```
Var

    VideoAdapter: VideoAdapterType;
```

которая может иметь только перечисленные в объявлении типа значения.

Каждый объект множества называется ***элементом*** этого множества. ***Область значений*** типа множество – ***набор*** всевозможных ***подмножеств***, составленных из элементов ***базового типа***, т.е. того типа, из элементов которого составлено множество.

В описании множества как типа используется конструктор **Set of** (множество из …), и следующее за ним указание базового типа. Возможны несколько способов задания множеств:

```
Type
    SetOfChar = Set of char;     {тип множества из символов
                                  ASCII}
    SetOfByte = Set of byte;     {тип множества из чисел 0..255}

    SetOfVideo = Set of VideoAdapterType; {тип множества из
                                  названий видеоадаптеров}
    Val_09 = Set of 0..9;        {тип множества из чисел от 0 до 9}
    Simbol = Set of '0'..'9';    {тип множества из символов
                                  '0', '1', …, '9'}
```

Как видно из примеров, можно в задании типа множества «урезать» базовый тип с целью создания нового множества.

✓ **Объявление переменных типа множества** можно выполнить непосредственно или через объявленный ранее тип:

```
Var
   Letter: Set of 'a'..'z';        {– непосредственное}
   Pr1, Pr2: Prostoe_09;           {– через объявленный ранее тип}
   Num: Val_09;                    {– через объявленный ранее тип}
   Simbol_apifm: SetOfChar;        {– через объявленный ранее тип}
```

В выражениях значения элементов множества указываются в квадратных скобках:

```
Simbol_apifm:= [‘+’, ‘-’];
        Num:= [1, 2, 7, 9];
     Letter:= [‘a’..‘r’];
        Pr1:= [5];
        Pr2:= [ ];                 {пустое множество}
```

Порядок следования элементов внутри скобок не имеет значения так же, как не имеет значения число повторений. Например, многократное включение элемента в множество:

Num:= [5, 2, 1, 2, 5];

эквивалентно однократному его упоминанию:

Num:= [5, 2, 1]; или Num:= [1, 2, 5]; или Num:= [2, 1, 5];

во всех случаях результатом будет множество **Num:= [1, 2, 5]**.

В Турбо Паскале разрешено определять множества, состоящие не более чем из 256 элементов. Столько же элементов содержат типы **byte** и **char**, и это же число является ограничением количества элементов в любом другом порядковом базовом типе множества, задаваемом программистом. Каждый элемент множества имеет сопоставимый номер.

Для типа **byte** номер равен значению числа; в типе **char** номером символа является его ASCII-код (Приложение 1). Нумерация всегда идет от 0 до 255.

Множества имеют весьма компактное машинное представление: один элемент расходует 1 бит. Поэтому для хранения 256 элементов достаточно 32 байт. Для меньшего диапазона значений множеств эта цифра будет еще меньше.

В операциях над множествами в качестве элементов множеств могут включаться константы и переменные соответствующих базовых типов. Более того, можно вместо элементов подставлять выражения, если тип их результата совпадает с базовым типом множества:

```
Var
    X: byte;
    S: Set of byte;
       …
    X:= 3;              {X – переменная}
    S:= [1, 2, X];      {множество из чисел: 1, 2, 3}
    S:= S + [X+4];      {множество из чисел: 1, 2, 3, 7}
```

Операции, применимые к множествам, сведены в таблицу 2.6.

Таблица 2.6

Операции над множествами

	Название	Форма	Комментарий
=	Проверка на равенство	**S1 = S2**	Результатом будет логическое значение, равное **True**,если **S1** и **S2** состоят из одинаковых элементов независимо от порядка следования, и **False** в противном случае
<>	Проверка на неравенство	**S1 <> S2**	Результатом будет логическое значение, равное **True**, если **S1** и **S2** отличаются хотя бы одним элементом, **False** в противном случае
>=	Проверка на надмножество	**S1 >= S2**	Результатом будет логическое значение, равное True, если все элементы S2 содержатся в S1, и False в противном случае

Продолжение табл. 2.6

	Название	Форма	Комментарий
<=	Проверка на подмножество	**S1 <= S2**	Результатом будет логическое значение, равное **True**, если все элементы **S1** содержатся в **S2** независимо от их порядка следования, и равное **False** в противном случае
in	Проверка вхождения элемента в множество	**E in S1**	Результатом будет логическое значение **True**, если значение **E** принадлежит базовому типу множества и входит в множество **S1**. Если множество не содержит в себе значения **E**, то результатом будет **False**
+	Объединение множеств	**S1 + S2**	Результатом объединения будет множество, полученное слиянием элементов этих множеств и исключением дублированных элементов
-	Разность множеств	**S1 – S2**	Результатом операции взятия разности **S1 – S2** будет множество, составленное из элементов, входящих в **S1**, но не входящих в **S2**
*	Пересечение множеств	**S1 * S2**	Результатом пересечения будет множество, состоящее только из тех элементов **S1** и **S2**, которые содержатся одновременно и в **S1**, и в **S2**

Некоторые примеры операций приведены в таблице 2.7.

Таблица 2.7

Примеры выполнения операций над множествами

Операции	Результат
[1, 2, 3, 4, 4] + [3, 4, 4, 5]	[1, 2, 3, 4, 5]
[1..6] + [7, 8, 9]	[1..9]
[X] + [ ]	[X]
[X] + [X + 1] + [X + 2]	[X..X + 2]
[‘4’, ‘20’] – [‘4’]	[‘20’]
[X] – [ ]	[X]
[ ] – [X]	[ ]
[‘a’, ‘c’] * [‘z’, ‘y’]	[ ]

[5] * [1..8] * [1, 2, 3, 5]	[5]

Стандартное программное обеспечение для обработки переменных множественного типа располагает процедурами **Include (S, ch)** и **Exclude (S, ch)**, позволяющими добавлять элемент *ch* в множество *S* и исключать его из множества.

◆ Математические функции и процедуры

Турбо Паскаль в своём составе имеет набор математических процедур и функций. Список процедур приведён в таблице 2.8, функций – в таблице 2.9.

Таблица 2.8

Стандартные математические процедуры

Описание	Назначение
Randomize	Гарантирует несовпадение последовательностей случайных чисел, выдаваемых функцией Random
Inc (Var X: целое)	Увеличивает значение *X* на *1*
Dec (Var X: целое)	Уменьшает значение *X* на *1*
Inc (Var X: целое; N: целое)	Увеличивает значение *X* на *N*
Dec (Var X: целое; N: целое)	Уменьшает значение *X* на *N*

Таблица 2.9

Стандартные математические функции

Вызов функции	Тип Аргумента	Тип значения	Назначение функции
Abs (X)	целый вещественный	целый вещественный	Абсолютное значение *X*
Pi	-	вещественный	3.1415…
Sin (X)	вещественный	вещественный	Синус *X* радиан
Cos (X)	вещественный	вещественный	Косинус *X* радиан
ArcTan (X)	вещественный	вещественный	Арктангенс *X* радиан
Sqrt (X)	целый вещественный	целый вещественный	Квадратный корень из *X*, $X \geq 0$
Sqr (X)	целый	целый	Значение X^2

	вещественный	вещественный	

Продолжение таблицы 2.9

Вызов функции	Тип Аргумента	Тип значения	Назначение функции
Exp (X)	вещественный	вещественный	Значение e^X
Ln (X)	вещественный	вещественный	Натуральный логарифм *X*, *X > 0*
Trunc (X)	вещественный	LongInt*	Целая часть значения *X*
Frac (X)	вещественный	вещественный	Дробная часть значения *X*
Int (X)	вещественный	вещественный	Целая часть значения *X*
Round (X)	вещественный	LongInt	‘Правильное’ округление *X* до ближайшего целого
Random	-	вещественный	Случайное число *(0..1)*
Random (X)	Word	Word**	Случайное число *(0..X)*
Odd (X)	Целый	логический	Возвращает True, если *X* – нечётное число

Тип LongInt и Word – целочисленные типы.

*Тип LongInt может принимать значения в диапазоне от -2147483648 до 2147483647.

**Word – в диапазоне от 0 до 65535.

◆ Ввод – вывод

В Турбо Паскале отсутствуют операторы ввода – вывода. Их роль выполняют процедуры **Read**, **Readln** (читать); **Write, Writeln** (писать). Вводить данные можно с клавиатуры, считывать из файла и т.д. Выводить можно на экран дисплея, на печатающее устройство, записывать в файл. В данном пособии рассмотрен ввод данных с клавиатуры и вывод на экран дисплея.

Ввод данных осуществляется через буфер ввода. **Процедуры ввода Read** и **Readln** обеспечивают ввод числовых данных, символов, строк в память компьютера для последующей их обработки операторами программы.

Процедура чтения **Read** имеет формат:

Read (X1, X2, …, Xn);

где *X1, X2, …, Xn* – список идентификаторов вводимых переменных.

Значения переменных *X1, X2, …, Xn* набираются на клавиатуре минимум через один пробел и высвечиваются на экране. После набора данных для одной процедуры **Read** нажимается клавиша ввода [Enter].

Сброс данных в память осуществляется *только после полного заполнения буфера*.

Процедура чтения Readln при вводе с клавиатуры отличается от процедуры **Read** только тем, что *сброс данных из буфера* в память осуществляется *сразу после выполнения этой команды*. При работе с файлами после считывания значений всех переменных для одной процедуры **Readln**, данные для следующей процедуры **Readln** будут считываться с начала новой строки файла.

В программах процедуры **Read** и **Readln** без параметров дают возможность задерживать процесс решения задачи до нажатия любой клавиши на клавиатуре. Мы будем пользоваться этим приёмом для задержки экрана с результатами.

Процедура вывода Write производит вывод числовых данных, символов, строк и булевских значений. После вывода курсор *остается в той же строке*.

Процедура **вывода Write** имеет формат:

Write (Y1, Y2, …, Yn);

где *Y1, Y2, …, Yn* – результаты выполнения выражений, константы, имена переменных.

Процедура Writeln аналогична процедуре **Write,** но после её выполнения происходит перевод курсора *в начало следующей строки*.

Например:

```
Write (45 + 89);
Write (‘Разность 12 – 5 = ’, 12 – 5);
Writeln (‘Xmin = ’, Xmin, ‘ Z = ’, Z);
Writeln (‘X = ’, X + 2);
```

Совместное использование процедур ввода и вывода позволяет сопровождать ввод данных запросами:

Write (‘Введите через пробел A, B, C: ’);
Read (A, B, C);

Writeln; {переход в начало следующей строки}

Write (‘X [’, k, ‘] = ’); {при k = 3 выводится запрос: X[3] = }
Readln (X[k]);

В процедурах вывода имеется возможность указать константу или выражение, определяющие ширину поля вывода, т.е. организовать *форматированный вывод*. Ширина поля – это количество позиций экрана, на которых мы хотим разместить выводимую величину. Она указывается через двоеточие сразу после выводимого элемента:

Writeln (20 + 25:10, 5:8)*;* {Результат: **45** **5**}
(7 пробелов; 8 пробелов)

Writeln (‘Сумма = ’, 6 – 18:4); {Результат: **Сумма = -12**}

Вещественные значения могут выводиться как в общепринятой, так и в научной форме. В научной форме указывается только ширина поля вывода, в общепринятой – дополнительно указывается количество выводимых знаков дробной части.

Например:

Writeln (234.32:10); {Результат: 2.3E+0002 – соответствует математической записи $2{,}3432 \cdot 10^2$ }

Writeln (234.32:9:3); {Результат: 234.320 – число отодвинуто вправо, впереди добавлено два пробела}

Writeln (234.32:5:3); {Результат: 234.320 – количество позиций система расширила ав-

томатически до приемлемого значения (7) }

Глава 3. Структура программы

В Турбо Паскале установлен стандарт программы, включающий:

- **Заголовок программы**
- **Предложение Uses**
- **Описательную часть**
- **Тело программы (исполнительную часть)**
- **Рекомендации по написанию программ**

◆ Заголовок программы

Заголовок необязателен. Если заголовок присутствует, то он начинается со служебного слова Program, за которым размещается её наименование. Заголовок заканчивается, как и любой раздел, точкой с запятой.

Например:

```
Program Prim_1;
```

◆ Предложение Uses

Этот раздел состоит из служебного слова **Uses** и списка имён подключаемых стандартных и пользовательских библиотечных модулей:

```
Uses Crt, Graph, My_Mod;
```

◆ Описательная часть

Описательная часть содержит расположенные в строгом порядке следующие необязательные разделы:

- раздел объявления меток;
- раздел объявления констант;
- раздел объявления типов;

- раздел объявления переменных;
- раздел объявления процедур и функций.

✓ **Раздел объявления меток**

Метка – специальная конструкция языка, позволяющая выполнить прямой переход на оператор, перед которым она стоит, из любого места программы. Метка состоит из имени и следующего за ней двоеточия. Именем может служить идентификатор, цифра или целое число. Максимальная длина имени метки ограничена 127 символами. Перед употреблением метка должна быть объявлена. Объявление начинается со служебного слова **Label**, за которым следует список меток. Завершается список точкой с запятой:

Label 23, A5, 45;

В структурном программировании применение меток запрещено.

✓ **Раздел объявления констант**

Раздел начинается со служебного слова **Const**, за которым размещаются структуры:

<идентификатор константы> = < значение константы>;

Например:

Const

A = 2000;

T = ‘Демонстрационная программа’;

✓ **Раздел объявления типов**

Началом служит служебное слова **Type**, за которым размещаются структуры:

<идентификатор типа> = <тип>;

Например:

```
Type
    TM1_r = array[1..27] of real;     {тип одномерного массива}
    TStr = string[45];                {строковый тип}
    T4 = 1..200;                      {интервальный тип}
```

- ✓ **Раздел объявления переменных**

Раздел начинается служебным словом **Var** (переменная), за которым следуют структуры:

<список однотипных переменных> : <тип>;

Например:

```
Var
    X, A, Z: real;
    S1: T1;
    Y, R: array [1..3, 1..4] of integer;
```

- ✓ **Раздел объявления процедур и функций**

Этот раздел начинается для процедур с заголовка

Procedure <имя> (C);

для функций

Function <имя> (C): <тип функции>;

где *C* – список формальных параметров и их типов.

За каждым заголовком располагается *описательная часть* и *тело* процедуры или функции. Подробно структуры процедур и функций описаны в [8].

- ◆ **Тело программы (исполнительная часть)**

Тело программы – обязательный блок. Он начинается с зарезервированного слова **Begin** (начало). Далее следуют *операторы языка*, отделённые друг от друга точкой с запятой и ре-

ализующие алгоритм решения задачи. Завершается блок зарезервированным словом **End (**конец) и **точкой**:

```
Begin
        <оператор>;
              …
        <оператор>;
End.
```

◆ Рекомендации по написанию программ

Из-за сложной структуры Паскаль-программы трудно читать. Для удобства чтения и облегчения поиска ошибок, их записывают в ступенчатом виде.

Условимся:

- операторы языка и их логически связанные структуры располагать на одну строку ниже с левым отступом до 5 позиций;
- ограничений на использование строчных и прописных букв накладывать не будем, однако, в повторном написании какого-либо идентификатора не будем заменять строчные буквы заглавными и наоборот, т. е. на протяжении всей программы сохранять первоначальное написание идентификатора;
- сопровождать участки программ, особенно операторные скобки **Begin..End**, комментариями.

Пример написания программы:

```
Program Prim_2;
Uses Crt;
Type
        Tstr = string[25];
Var
        A, B, C: integer;
        S: Tstr;
BEGIN
          ClrScr;          {очистка экрана}
                S:= ‘Целая часть от деления ’;
                Writeln (‘Введите два целых числа:’);
```

```
        Write (‘A = ’);
        Readln (A);
        Write (‘B = ’);
        Readln (B);
        C:= A div B;
        Writeln (S, A, ‘ на ’, B, ‘ = ’, C:5);
    Readln;        {задержка экрана результатов}
END.               {конец программы}
```

Напомним, что запросы и сопроводительные тексты программы, выводимые в протокол решения задачи, называются ***интерфейсом*.** Интерфейс должен быть корректным и дружественным. Для его организации используются стандартные процедуры **Write и Writeln.** Тексты, подлежащие выводу, записываются в программах в апострофы **‘ ’**, а выводятся в протокол без них.

Глава 4. Операторы

- **Общие сведения**
- **Простые операторы**
- **Структурные операторы**

◆ Общие сведения

Тело программы представляет собой последовательность операторов, каждый из которых выполняет некоторое действие над данными. *Концом* любого оператора служит *точка с запятой*. Все операторы подразделяются на простые и структурные.

В дальнейшем будем рассматривать операторы, допустимые только *в технологии струтурного программирования*.

◆ Простые операторы

Операторы, не содержащие других операторов, называются простыми. К ним относятся операторы:

- присваивания;
- вызова процедур;
- пустой.

✓ **Оператор присваивания** (:=) предписывает выполнить выражение, заданное в его правой части, и присвоить результат переменной, идентификатор которой расположен в левой части. Переменная и выражение должны быть совместимы по типу.

Например:

```
Var
    Sum, X, Y: real;
    Flag: Boolean;
    St: string [20];
Begin
    ClrScr;
    Flag:= True;
    St:= ‘Здравствуйте!’;
    Sum:= X + Y;
End.
```

✓ **Оператор вызова процедуры** служит для активизации предварительно определённой пользователем или стандартной процедуры и имеет структуру:

<имя процедуры> (<список фактических параметров>);

При вызове процедур, не имеющих параметров (процедур без параметров), список фактических параметров отсутствует.

Например:

```
ClrScr;    {вызов стандартной процедуры без параметров}
Dec (X);   {вызов стандартной процедуры с параметром X}
Input_M1 (N, X);    {вызов пользовательской процедуры с параметрами N, X}
```

✓ **Пустой оператор** не содержит никаких символов и не выполняет никаких действий**.**

◆ Структурные операторы

Структурные операторы представляют собой конструкции, построенные из других операторов по строго определённым правилам. Все структурные операторы подразделяются на три группы:

- составные;
- условные;
- цикла.

✓ **Составной оператор** или *блок* представляет собой группу из произвольного числа операторов, отделённых друг от друга точкой с запятой, и ограниченную операторными скобками **Begin** и **End**:

Begin
<оператор;>
...
<оператор;>
End;

Составной оператор воспринимается как единое целое, и может находиться в любом месте программы, где синтаксис языка допускает его наличие.

Группу условных операторов составляют два оператора: **If** и **Case.**

✓ **Условный оператор If** является одним из самых популярных средств, изменяющих естественный порядок выполнения операторов программы. Он соответствует выбору одного из двух вариантов действий (рис. 2.1).

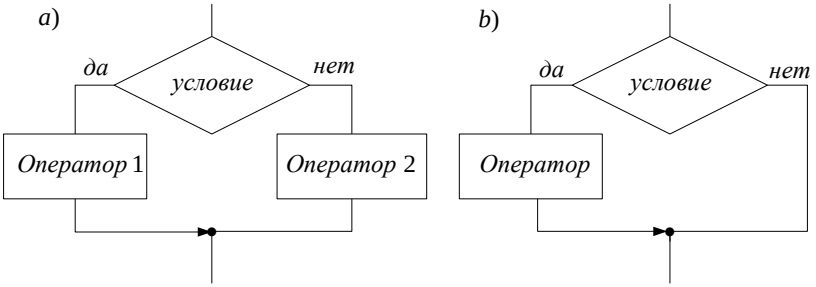


Рис. 2.1. Структура условного оператора:
а) полная форма, b) неполная форма

где у*словие* – это выражение типа *Boolean*, принимающее значение *True* (да) или *False* (нет);

Оператор 1, Оператор 2, Оператор – простые или составные операторы программы.

Программная реализация полной формы имеет вид:

If <условие> **then** <оператор 1> **else** <оператор 2>;

Например,

```
If X <> A then
        Y:= X / (A – X)
    else
      Writeln (‘Функция не определена – деление на нуль’);
```

Неполной форме соответствует программная реализация вида:

If <условие> **then** <оператор>;

Например,

```
If (X[I] >= A) and (X[I] <= B) then S:= S + X[I];
```

Операторы **If** могут быть вложенными, т. е. каждый из элементов: <оператор 1>, <оператор 2>, <оператор> может быть опять оператором **If**.

Например,

```
If A = B then
        Writeln (‘числа равны’)
        else
          If A > B then
                  Writeln (‘наибольшее = ’, A:8:2)
                else
                  Writeln (‘наибольшее = ’, B:8:2);
```

✓ **Оператор выбора Case** позволяет сделать выбор из произвольного числа вариантов. На рисунке 2.2 представлена блок-схема этого оператора.

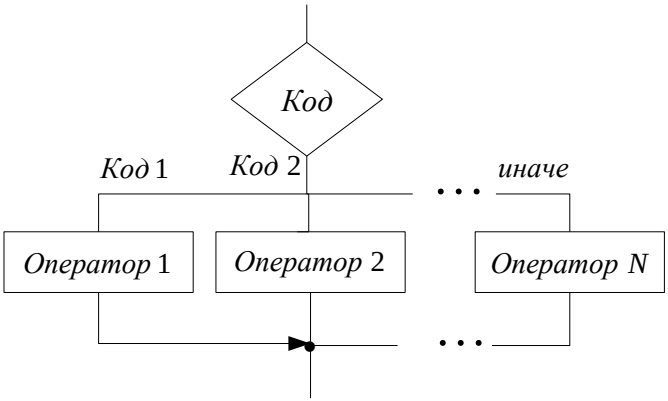


Рис. 2.2. Блок-схема оператора выбора:

где *код* – управляющая переменная или выражение любого типа, кроме типа **real** и **string**;

код1, код2, ..., кодN – конкретные значения управляющего кода, при которых необходимо выполнить соответствующий оператор, игнорируя остальные варианты кода;

Оператор 1, Оператор 2, ..., Оператор N – простые или составные операторы.

Программную реализацию оператора выбора продемонстрируем фрагментом программы, в котором в зависимости от значения переменной *P* на экран выводится одна из фраз:

- Мы изучаем Турбо Паскаль. {при P = 1}
- Мы изучаем Ассемблер. {при P = 2}
- Мы изучаем Си. {при P = 3}
- Мы изучаем математику. {при P = любому целому числу, не равному 1, 2 или 3}

Фрагмент программы

```
Writeln (‘Введите целое число равное 1, 2, 3’);
Write (‘или любое другое, отличное от них’);
Readln (P); {P – управляющий код}
Write (‘Мы изучаем ’);
Case P of
        1: Writeln (‘Турбо Паскаль.’);
        2: Writeln (‘Ассемблер.’);
        3: Writeln (‘Си.’)
           else Writeln (‘математику.’);
End; {конец Case}
```

✓ **Операторы цикла**

Циклом называется многократно повторяющийся участок программы. Любой цикл состоит из двух частей:

- части, *управляющей* повторением цикла,
- части, *выполняющей* вычисления согласно алгоритму решения задачи.

Структура управляющей части определяет принадлежность цикла к одному из трех, имеющих место в программировании, циклов.

- *счетный цикл* – цикл с заранее заданным количеством повторений;
- *цикл – пока*, определяющий повторение действий, пока не будет нарушено управляющее условие, выполнение которого проверяется в начале цикла – предусловный цикл;
- *цикл – до*, обозначающий повторение действий до выполнения управляющего условия, проверка которого осуществляется после выполнения действий в цикле – постусловный цикл.

Блок-схемы циклов представлены на рисунке 2.3.

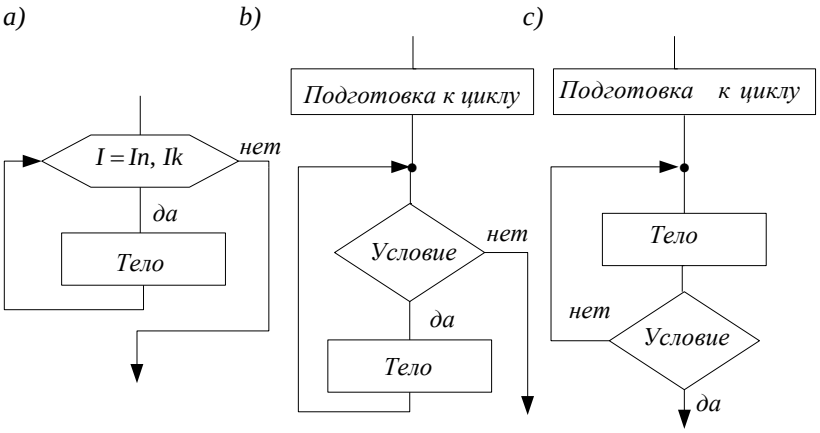


Рис. 2.3. Блок-схемы циклов:
а) счетного цикла, b) цикла – пока, c) цикла – до

Счетный цикл – это цикл с заранее известным количеством повторений (рис. 2.3, а)). Его управляющая часть называется ***заголовком цикла***. Заголовок содержит управляющую переменную, называемую ***параметром цикла***, ее начальное, конечное значения. Шаг изменения параметра цикла равен *±1*.

В счетном цикле *I* – параметр цикла (простая переменная любого целого или символьного типа), *In*, *Ik* – начальное, конечное значения параметра цикла.

Счетный цикл с шагом изменения параметра цикла равным *+1* реализуется структурой:

For <параметр цикла>:= *In* **to** *Ik* **do** <тело цикла>;

При шаге изменения параметра цикла равным *-1* структура имеет вид:

For <параметр цикла>:= *Ik* **downto** *In* **do** <тело цикла>;

В краткой структуре заголовка заложены в строгой последовательности следующие действия:

1. Формирование начального значения параметра цикла: *I:= In*;
2. Проверка условия $I \le Ik$? Ответом может быть '*да*' или '*нет*'. При '*да*' переходим к пункту 3. При '*нет*' работа цикла заканчивается и осуществляется выход из него.
3. Выполнение тела цикла.
4. Формирование следующего значения параметра цикла: $I:= I \pm 1$.
5. Переход к пункту 2.

Внимание!
При выходе из цикла параметр цикла становится равным своему конечному значению!

Фрагмент программы при изменении параметра цикла с шагом +1

```
S:= 0;
For I:= 1 to N do
        begin
            If X[I] < A then
                                S:= S + X[I];
            If X[I] mod A = 0 then
                                    X[I]:= A;
        end;
```

Фрагмент программы при изменении параметра цикла с шагом -1

```
S:= 0;
For I:= N downto 1 do
        begin
            If X[I] < A then
                                S:= S + X[I];
            If X[I] mod A = 0 then
                                    X[I]:= A;
        end;
```

Цикл – пока (рис. 2.3, б)) реализуется структурой:

While <управляющее условие> **do** <тело цикла>;

Фрагмент программы с циклом – пока

```
S:= 0;
I:= 1;
While I <= N do
        begin
            If X[I] < A then
                                S:= S+X[I];
            If X[I] mod A = 0 then
                                        X[I]:= A;
            I:= I + 1;
        end;
```

Цикл – до (рис. 2.3, в)) реализуется структурой:

Repeat <тело> **Until** <управляющее условие>;

Фрагмент программы с циклом – до

```
S:= 0;
I:= 1;
Repeat
        If X[I] < A then
                            S:= S + X[I];
        If X[I] mod A = 0 then
                                X[I]:= A;
        I:= I + 1;
Until I > N;
```

ЧАСТЬ 3. РАЗРАБОТКА АЛГОРИТМОВ И ПРОГРАММ

В этой части на конкретных примерах показаны некоторые приёмы алгоритмизации задач, разработки и оформления программ. Изложение не претендует на единственность алгоритмов рассматриваемых задач. Во всех программах информация в фигурных скобках { } является комментарием.

Глава 1. Обработка данных, представленных простыми переменными

- **Обмен значениями двух переменных *X* и *Y***
- **Определение целой части и остатка частного *X / Y (Y ≠0)***
- **Определение суммы цифр натурального числа *N***
- **Вывод натурального числа в обратном порядке**
- **По заданному натуральному числу *N (N > 2)* вывести строку *S***
- **Определить принадлежность точки *R (X, Y)* прямоугольнику *ABCD***
- **Дан интервал натуральных чисел от *N* до *M*. Определить все простые числа в этом интервале**
- **Найти сумму степенного ряда**

- **Вычислить значения функции *y = f (x)* при изменении аргумента *x* на интервале *[a, b]* с шагом *h (a < b)***

- **Обмен значениями двух переменных *X* и *Y***

Обсуждение проблемы

Воспользуемся дополнительной переменной R.

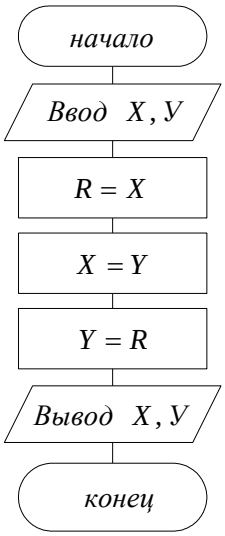
Выполним следующие действия:

```
R:= X;  {копируем значение переменной X в R}
X:= Y;  {копируем значение переменной Y в X}
Y:= R;  {копируем значение переменной R в Y}
```

Переменные могут быть любого типа. Мы рассмотрим вариант программы с переменными типа **real**.

Переменные *X, Y* – являются одновременно как исходными данными, так и результатами. Переменная *R* – вспомогательная.

Блок-схема алгоритма:




```
Program Prim1_1;
Uses Crt;        {подключение стандартного модуля Crt}
{Раздел объявления переменных:}
Var
   X, Y, R: real;
BEGIN            {начало тела программы}
   ClrScr;       {очистка экрана}
      Writeln ('* Поменять местами численные значения двух переменных *');
      Write ('Введите два числа: ');
      Write ('X = '); Readln (X);
      Write ('Y = '); Readln (Y);
      R:= X;
      X:= Y;
      Y:= R;
{Числа выводятся через запятую; размещаются на 6 позициях с 1 знаком после
```

```
запятой:}
        Writeln (‘Результат: ’, X:6:1, ‘, ’,
                Y:6:1);
{Задержка экрана с результатами:}
    Readln;
END.            {конец программы}
```

Протокол

```
* Поменять местами численные значения двух переменных *
Введите два числа:
X = -34.5
Y = 45
Результат:  45.0  -34.5
```

Реализация алгоритма с любым другим типом данных потребует внесения изменений в объявления типа переменных и в операторы ввода-вывода.

◆ Определение целой части и остатка частного *X / Y* *(Y ≠ 0)*

Обсуждение проблемы

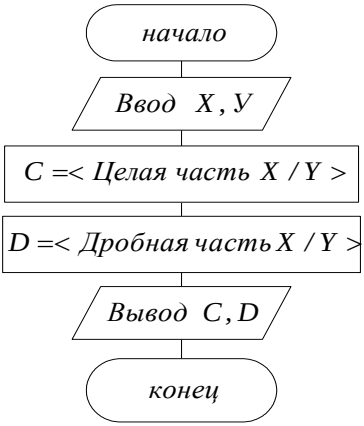
Обозначим целую часть через *C*, дробную через *D*. Переменные *X, Y* – исходные данные, *C, D* – результаты. Числа *X, Y* могут быть как целыми, так и вещественными, поэтому задачу будем решать в двух вариантах.

Вариант А. Пусть числа *X* и *Y* целые, типа **integer**. Для решения задачи подходят операции **div** и **mod**.

Вариант В. Пусть числа *X* и *Y* вещественные, типа **real**. Для решения задачи необходимо применить стандартные математические функции **Trunc ( )** и **Frac ( )**.

В программе вычисления и вывод совместим в операторах **Writeln**.

Блок-схема алгоритма:



Вариант А:

```
Program Prim1_2A;
Uses Crt;        {подключение стандартного модуля Crt}
Var
    {Объявление переменных целого типа:}
    X, Y: integer;
BEGIN
    ClrScr;
        Writeln (‘* Определить целую часть и остаток частного
                X / Y *’);
        Write (‘Введите через пробел два числа: ’);
        Readln (X, Y);
        {Вывод значений целых переменных:}
        Writeln (‘Целая часть частного ’, X, ‘ / ’, Y, ‘ = ’, X div Y);
        Writeln (‘Дробная часть = ’, X mod Y);
END.
```

Протокол

```
* Определить целую часть и остаток частного X / Y *
Введите через пробел два числа: 17 5
Целая часть частного 17 / 5 = 3
Дробная часть = 2
```

Вариант B:

```
Program Prim1_2B;
Uses Crt;
Var
    X, Y: real;
BEGIN
    ClrScr;
        Writeln (‘* Определить целую часть и остаток частного
                X / Y *’);
        Write (‘Введите через пробел два числа: ’);
        Readln (X, Y);
        {Результат функции Trunc целого типа, поэтому указана
        только ширина поля вывода:}
        Writeln (‘Целая часть частного ’, X, ‘ / ’, Y, ‘ = ’, Trunc (X / Y):5);
        {Результат функции Frac вещественного типа, поэтому ука-
        заны и ширина поля вывода, и число знаков после запятой:}
        Writeln (‘Остаток = ’, Frac (X / Y):5:3);
    Readln;
END.
```

Протокол

```
* Определить целую часть и остаток частного X / Y *
Введите через пробел два числа: 17 5
Целая часть частного 17 / 5 =    3
Остаток = 0.400
```

◆ Определение суммы цифр натурального числа *N*

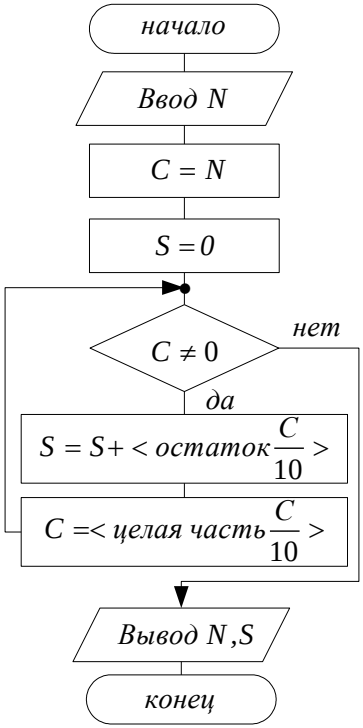
Обсуждение проблемы

Натуральное десятичное число содержит разряды: единиц, десятков, сотен и т.д. При делении его на 10 остаток будет равен цифре разряда единиц, а целая часть в младшем разряде уже будет содержать цифру разряда десятков исходного числа. Если оставшуюся целую часть вновь разделить на 10, то новый остаток будет равен цифре разряда десятков. Повторив в цикле процесс деления новых целых частей на 10 до целой части равной 0, получим все цифры исходного числа. Остаётся только просуммировать эти цифры. Обозначим целую часть через *C*, сумму –

через *S*. Если задать начальное значение суммы равным 0, то суммирование цифр в цикле можно провести по формуле:

S = S + <остаток C / 10>.

Блок-схема алгоритма:




```
Program Prim1_3A;
Uses Crt;
Var
    N, C, S: integer;
BEGIN
    ClrScr;
    Writeln (‘* Нахождение сумм цифр натурального числа *’);
    Write (‘Введите число = ’);
    Readln (N);
    C:= N; S:= 0;
    While C <> 0 do
        begin
            S:= S + C mod 10;
            C:= C div 10;
        end;
    Writeln (‘Сумма цифр числа’, N, ‘ = ’, S);
            {N, S – целые; ширину поля вывода можно не указывать}
    Readln;
END.
```

Протокол

* Нахождение суммы цифр натурального числа *
Введите число = 759
Сумма цифр числа 759 = 21

На примере решения этой задачи познакомимся с использованием в программировании **стандартного участка функции**. Оформим процесс последовательного выделения и суммирования остатков функцией **Sum_N**, которая через свой параметр *N* будет получать из основной программы значение натурального числа *N*, а в основную программу передавать сумму его цифр. Переменная *N* для функции будет исходным данным, а имя **Sum_N** – результатом.

Объявление функции

```
Function Sum_N (N: integer): integer;
Var
    S: integer; {S – вспомогательная переменная для вычисления результата – внутренний ресурс для функции}
Begin
      S:= 0;
      While N <> 0 do
                  begin
                        S:= S + N mod 10;
                        N:= N div 10;
                  end;
      Sum_N:= S; {через имя Sum_N результат передается в программу}
End;
```

```
Program Prim1_3B;
Uses Crt;
Var
   N: integer;
Function Sum_N (N: integer): integer;
Var
   S: integer;
Begin
      S:= 0;
      While N <> 0 do
                  begin
                        S:= S + N mod 10;
                        N:= N div 10;
                  end;
      Sum_N:= S;
End;
```

```
{Основной блок программы:}
BEGIN
    ClrScr;
        Writeln (‘* Нахождение суммы цифр натурального
                числа *’);
        Write (‘Введите число = ’);
        Readln (N);
        Writeln (‘Сумма цифр числа ’, N, ‘ = ’, Sum_N (N));
    Readln;
END.
```

◆ Вывод натурального числа в обратном порядке

Обсуждение проблемы

Задача, как и предыдущая, может быть решена с использованием арифметических действий **div** и **mod**. Для решения разработаем процедуру **Order_N** с входным параметром *N*.

Объявление процедуры

```
Procedure Order_N (N: integer);
Begin
    While N <> 0 do
            begin
                Write (N mod 10); {вывод цифр в одну строку}
                N:= N div 10;
            end;
    Writeln;                          {переход на новую строку}
End;
```

Программа

```
Program Prim1_4;
Uses Crt;
Var
    N: integer;
Procedure Order_N (N: integer);
Begin
    While N <> 0 do
            begin
                Write (N mod 10);   {вывод цифр в одну строку}
```

```
            N:= N div 10;
        end;
    Writeln;                          {переход на новую строку}
End;

{Основной блок программы:}
BEGIN
    ClrScr;
        Writeln ('* Вывод натурального числа в обратном порядке *');
        Write ('Введите число = ');
        Readln (N);
        Writeln ('Исходное число = ', N);
        Writeln ('Результат = ');
        Order_N (N);  {обращение к процедуре}
    Readln;
END.
```

Протокол

```
*Вывод натурального числа в обратном порядке*
Введите число = 375
Исходное число = 375
Результат = 573
```

- **По заданному натуральному числу *N (N > 2)* вывести строку**

$$S = \begin{cases} 2 * 4 * ... * N, \text{ если } N \text{ четное}, \\ 1 * 3 * ... * N, \text{ если } N \text{ нечетное}. \end{cases}$$

Обсуждение проблемы

Организуем циклический процесс по переменной *I*. Её значения будут изменяться **для чётных** сомножителей от *2* до *N-2*, а **для нечётных** – от *1* до *N-2*, **с шагом** *2 в обоих случаях.*

Последний сомножитель *N* будем выводить после окончания работы цикла.

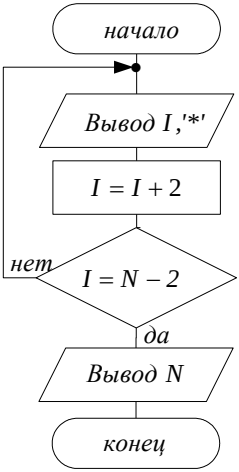
Так как цикл For допускает только шаг равный *1* или *-1*, а в нашем случае шаг равен *2*, то воспользуемся циклом **Repeat … Until** (можно и циклом **While**).

Представим оформление цикла в виде стандартного участка процедуры под именем **Stroka** с параметром, характеризующим начальное значение *I*.

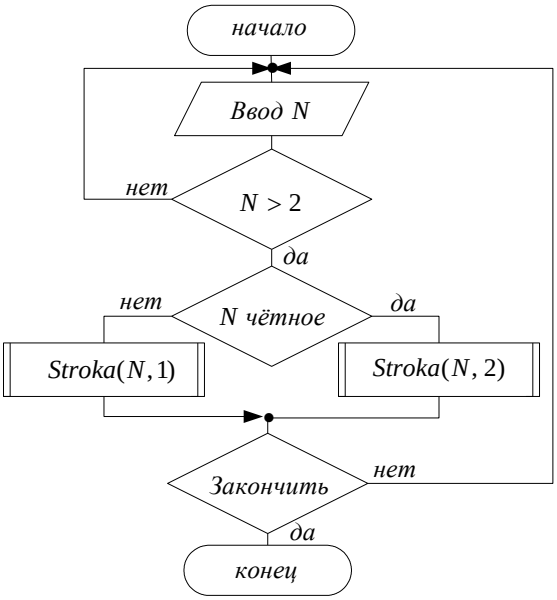
Обратившись к ней при *I = 2*, выведем чётные сомножители. Обратившись при *I = 1*, выведем нечётные сомножители.

Процедура **Stroka (N, I: integer);**

Блок-схема процедуры:



Блок-схема программы:




```
Program Prim1_5;
Uses Crt;
Var
    N: integer;
    C: char;      {Переменная C объявлена для организации
                   запроса на повторное решение задачи}
Procedure Stroka (N, I: integer);
Begin
      Repeat
            Write (I, ' * ');
```

```
            I:= I + 2;  {можно Inc (I, 2) }
        Until I >= N – 2;  {конец цикла по I}
        Writeln (N);        {вывод последнего сомножителя}
End;

{Основной блок программы:}
BEGIN
    ClrScr;
        Writeln (‘* Вывести строку S = 2*4*…*N, если N чётное *’);
        Writeln (‘*          и строку S = 1*3*…*N, если N нечётное *’);
        Repeat {начало цикла для повторного решения задачи}
              Repeat        {начало цикла по контролю ввода числа N}
                    Write (‘Введите натуральное число = ’);
                    Readln (N);
              Until N > 2;  {конец цикла по контролю ввода числа N}
              Write (‘S = ’);       {вывод сопроводительного текста}
              If N mod 2 = 0 then Stroka (N, 2)   {обращение к
                                          процедуре Stroka для
                                          вывода чётных сомно-
                                          жителей}
                             else Stroka (N, 1);  {обращение к
                                          процедуре Stroka для
                                          вывода нечётных
                                          сомножителей}
              Write (‘Повторить? Y/N ’);
              Readln (C);
        Until UpCase (C) = ‘N’;    {выход из цикла, как только будет
                                    нажата клавиша «n» или «N»}
END.
```

Протокол

```
* Вывести строку S = 2*4*…*N, если N чётное *
*          и строку S = 1*3*…*N, если N нечётное *
Введите натуральное число = 12
S = 2*4*6*8*10*12
Повторить? Y/N   Y
Введите натуральное число = 11
S = 1*3*5*7*9*11
Повторить? Y/N   N
```

- **Определить принадлежность точки *R (X, Y)* прямоугольнику *ABCD***

Обсуждение проблемы

Для задания прямоугольника достаточно указать координаты двух точек, расположенных на одной диагонали. Пусть этими точками будут точки *A (XA, YA)* и *C (XC, YC)*. Точка *R будет принадлежать* прямоугольнику, *если* её координаты будут *одновременно* удовлетворять условиям:

$$XA \le X \le XC, \qquad YA \le Y \le YC.$$

В Турбо Паскале для реализации «одновременного удовлетворения условиям» предназначена логическая операция **and *(логическое И)***.

В данном случае логическое выражение можно записать в виде:

(XA ≤ X) and (X ≤ XC) and (YA ≤ Y) and (Y ≤ YC)

или

(X ≥ XA) and (X ≤ XC) and (Y ≥ YA) and (Y ≤ YC).

Результатом выполнения этих выражений будет **True**, результатом невыполнения – **False**.

Применив условный оператор **If … then … else**, получим решение задачи. Проверку принадлежности или не принадлежности нескольких точек можно организовать циклическим процессом, в тело которого войдёт *ввод координат точки R, оператор* **If** *и вывод результатов.*

В этом алгоритме продемонстрируем организацию циклического процесса решения задачи для нескольких точек.

Программа

```
Program Prim1_6;
Uses Crt;
Const
    PR = ‘прямоугольнику’;    {константа PR предназначена для
                               формирования сопроводительного
                               текста при выводе результатов}
```

```
Var
    X, Y: integer;        {координаты точки}
    XA, YA: integer;      {координаты вершины A}
    XC, YC: integer;      {координаты вершины C}

{Основной блок программы:}
BEGIN
    ClrScr;
        Writeln ('* Определить принадлежит или не принадлежит *');
        Writeln ('* точка R (X, Y) прямоугольнику ABCD *');
        Write ('Введите через пробел координаты вершины A ');
        Readln (XA, YA);
        Write ('                          координаты вершины C ');
        Readln (XC, YC);
        Repeat {начало цикла решения задачи для нескольких
                точек}
            Write ('Введите через пробел координаты точки ');
            Readln (X, Y);
            Write ('Точка R (', X:3, ',', Y:3, ')');
            If (X >= XA) and (X <= XC) and (Y >= YA) and
            (Y <= YC) then Writeln ('принадлежит ', PR)
                      else Writeln ('не принадлежит ', PR);
            Writeln ('Повторить? Y/N ');
        Until (ReadKey = 'n') or (ReadKey = 'N');   {выход из
                                цикла, как только будет нажата
                                клавиша «n» или «N»}
END.
```

- **Дан интервал натуральных чисел от *N* до *M*. Определить все простые числа в этом интервале**

Обсуждение проблемы

Целое положительное число *K* называется простым, если оно делится только на себя и на единицу, в противном случае число *K* называется составным. В алгоритме решения задачи предусмотрим вывод простых чисел при их наличии и вывод фразы «простых чисел нет» при их отсутствии. Введём в алгоритм две переменные *F* и *L*.

Факт, что конкретное число *K* из интервала *[M, N]* является простым или таковым не является, будет фиксировать переменная *F*, являющаяся выходным параметром подпрограммы

Simple (K: integer; Var F: Boolean).
{Служебное слово **Var** *перед результатом* ***обязательно***}

В начале подпрограммы будем предполагать, что очередное число *K* является простым *(F:= True)*. Будем делить число последовательно на делитель *J = 2, 3, 4 и т.д. до величины D, равной целой части K / 2*. Если окажется, что число делится на очередной делитель без остатка (является составным), переопределим значение переменной *F (F:= False)*, и прекратим процесс перебора делителей. Таким образом, при выходе из подпрограммы значение переменной *F* будет удовлетворять соотношению

$$F = \begin{cases} True, & если\ число\ K - простое, \\ False, & в\ противном\ случае \end{cases}$$

Значение переменной *L* будем формировать в основном блоке программы по формуле

$$L = \begin{cases} количеству\ простых\ чисел\ на\ интервале\ [M, N], \\ 0, \quad в\ противном\ случае \end{cases}$$

В программу включим контроль ввода исходных данных по условию:

(N > 0) и (M > 0) и (N < M).

```
Program Prim1_8;
Uses Crt;
Var
    N, M, K, L: integer;
    F: Boolean;
    P: char;
```

```
Procedure Simple (K: integer; Var F: Boolean);
Var J, D: integer;
Begin
    F:= True;  {предполагаем, что число K – простое}
    J:= 1;
    D:= K div 2;
    Repeat {начало цикла по перебору делителей}
          Inc (J);
    Until (J > D) or (K mod J = 0);
    if K mod J = 0 then F:= False; {Число K – составное}
End;

{Основной блок программы:}
BEGIN
    ClrScr;
        Writeln (‘* Дан интервал натуральных чисел от N до M. *’);
        Writeln (‘* Определить все простые числа в этом интервале. *’);
        Repeat {начало цикла решения задачи с новыми данными}
            {Ввод чисел M и N с контролем:}
            Repeat
              Writeln (‘Введите нижнюю границу интервала = ’);
              Readln (M);
              Writeln (‘Введите верхнюю границу интервала = ’);
              Readln (N);
            Until (M > 0) and (N > 0) and (M < N);
            {Обработка данных:}
            Writeln (‘В интервале от ’, M, ‘ до ’, N);
            L:= 0;
            For K:= M to N do
                  begin
                      Simple (K, F);
                      if F = True then
                                    begin
                                        Writeln (K, ‘ – простое’);
                                        Inc (L);
                                    end;
                  end;
            if L = 0 then Writeln (‘простых чисел нет’);
            Writeln;
```

```
        Write (‘Повторить? y/n – ’);
        Readln (P);
    Until UpCase (P) = ‘N’;          {конец цикла решения задачи с новыми данными}
```

END.

Протокол

```
* Дан интервал натуральных чисел от N до M. *
* Определить все простые числа в этом интервале. *
Введите нижнюю границу интервала = 25
Введите верхнюю границу интервала = 52
В интервале от 25 до 52
29 – простое
31 – простое
37 – простое
41 – простое
43 – простое
47 – простое

Повторить? y/n – y
Введите нижнюю границу интервала = 14
Введите верхнюю границу интервала = 16
В интервале от 14 до 16
простых чисел нет

Повторить? y/n – n
```

- **Найти сумму степенного ряда**

$$1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots;\ n = 0, 1, 2, \ldots$$

Обсуждение проблемы

Степенные ряды успешно применяются для приближенного вычисления значения некоторой функции $y = f(x)$ при конкретном значении аргумента x. Эта задача может быть решена

только для сходящихся рядов. Следовательно, при ее решении необходимо знать область сходимости ряда – интервал изменения значений аргумента, в каждой точке которого сумма ряда имеет конечное значение. В данном случае ряд записан для функции *Cos (x)* и сходится при $-\infty < x < +\infty$. Если область сходимости ряда неизвестна, то предлагаемые алгоритмы решения не дадут положительных результатов.

Реализуем две задачи, связанные с вычислением суммы ряда при конкретном значении аргумента *x*.

Задача 1. Вычислить сумму первых M членов ряда.

Задача 2. Вычислить сумму ряда с заданной точностью ε.

Алгоритмы решения этих задач можно реализовать в общих обозначениях переменных. Обозначим через

S – сумму ряда;

R – член ряда;

N – порядковый номер члена ряда;

M – количество членов ряда для решения задачи 1;

epsilon – точность ε для решения задачи 2.

Алгоритмы решения опишем циклическими процессами, в теле которых воспользуемся операторами:

S:= S + R;
N:= N + 1;
R:= R * g;

где g – множитель, на который последующее значение члена ряда отличается от его предыдущего значения.

В приведенных операторах отсутствует формула вычисления множителя g. Вычислим этот множитель через отношение последующего *(n-го)* члена ряда к предыдущему *(n-1-му)*:

$$g = \frac{(-1)^n x^{2n}}{(2n)!} : \frac{(-1)^{n-1} x^{2(n-1)}}{[2(n-1)]!} = -\frac{x^2}{(2n-1)(2n)};$$

Проверим справедливость полученной формулы.

Для первого члена ряда $n = 1$ и $R = -\frac{x^2}{2}$.

Для второго члена $n = 2$, $g = -\frac{x^2}{3*4}$, значит, второй член при вычислении по формуле $R:= R * g$ будет равен $\frac{x^4}{4!}$.

Для третьего члена ряда $n = 3$, $g = -\frac{x^2}{5*6}$, значит, сам член равен $\frac{x^4}{4!}\cdot\left(-\frac{x^2}{5*6}\right)=\frac{x^6}{6!}$.

Отсюда следует, что нумерация членов ряда выбрана нами верно и правильно выведена формула для вычисления множителя g.

Рекомендация:
всегда проверяйте правильность начального значения N и формулы для вычисления множителя g.

Алгоритм решения задачи 1 опишем словесно последовательностью действий:

1. Ввод значений переменных X, M.
2. Подготовка к выполнению счетного цикла – задание начальных значений:

   S:= 1; R:= 1;

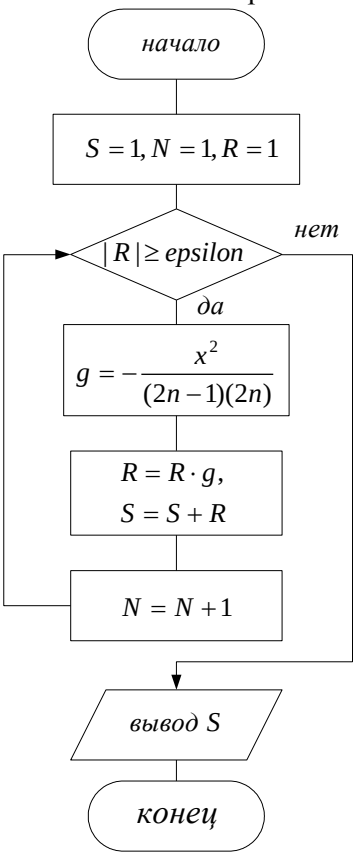
3. Организация заголовка счетного цикла:

   For N:= 1 to M do

4. Организация тела цикла – составного оператора, выполняющего последовательность действий:
   - вычисление множителя g;
   - вычисление очередного значения члена ряда R;
   - накопление суммы: S:= S + R;
5. Вывод значения суммы на экран.

Алгоритм решения задачи 2 представим в виде блок-схемы и программы с протоколом ее решения.

Блок-схема алгоритма




```
Program Prim_9;
Uses Crt;
Var
    X, epsilon, S, R, g: real;
    Num : integer;

{Основной блок программы:}
BEGIN
    ClrScr;
      Writeln (‘* Найти сумму ря-
              да с заданной точ-
              ностью *’);
      {Ввод исходных данных:}
      Write (‘Введите значение
              точности = ’);
      Readln (epsilon);
      Write (‘ значение X = ’);
      Readln (X);
      {Подготовка к циклу:}
      S:= 1; Num:= 1;
      R:= -X * X / 2;
      While Abs (R) >= epsilon do
            begin
              g:= -X * X / ((2 *
                    Num-1) * (2 *
                    Num));
              R:= R * g;
              S:= S + R;
              Inc (Num);
            end;
        {Вывод результатов:}
        Writeln (‘Сумма ряда = ’,
S:6:3);
    Readln;
END.        {Конец программы}
```

Протокол
* Найти сумму ряда с заданной точностью *
Введите значение точности = 0.001
значение X = 2
Сумма ряда = -0.416

- **Вычислить значения функции *y = f (x)* при изменении аргумента *x* на интервале *[a, b]* с шагом *h* *(a < b)***

Обсуждение проблемы

Данную задачу принято называть задачей табулирования функции. Функция на заданном интервале может быть непрерывной или иметь точки разрыва (точки, в которых функция не определена). Эти точки возникают, например, при делении на нуль, при извлечении квадратного корня из отрицательного числа, при взятии логарифма от отрицательного или равного нулю аргумента. Если в программе не предусмотреть реакцию на подобные ситуации, то каждый раз при их возникновении работа программы будет заканчиваться аварийным прерыванием с выдачей сообщения об ошибке.

Обычно при аргументе, приводящем к разрыву, вместо значения функции выводят текстовое сообщение «не определена», «разрывна», или значение функции полагается равным большой (маленькой) величине, заведомо не равной её значениям на заданном интервале, например, **99999999**.

Задача имеет несколько вариантов решения в зависимости от классификации переменных. Мы рассмотрим только один, в котором исходными данными являются переменные *a*, *b* и *h*; результатами *x* и *y*, и все эти переменные *простые вещественного типа.*

Начальное значение переменной *x* будем определять перед циклом, а закон её изменения в цикле организуем по формуле *x:= x + h*. Вычисления организуем для функции

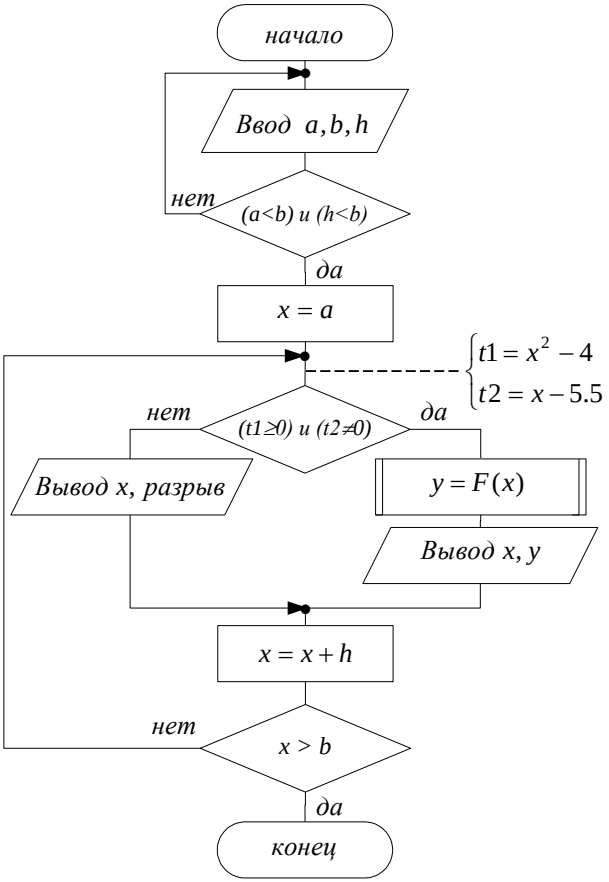
$$y = \sqrt{x^2 - 4} + \frac{2 + x}{x - 5.5},$$

имеющей разрывы при

$$x^2 - 4 < 0 \text{ и } x - 5.5 = 0.$$

В точках разрыва вместо значения функции будем выдавать сообщение «**разрыв**».

Блок-схема алгоритма



```
Program Prim1_7;
Uses Crt;
Var
    A, B, H: real;          {исходные данные: A, B – начало и конец
                             интервала изменения аргумента, H – шаг}
    X, Y: real;             {результаты}

{Объявление функции:}
Function F (X: real): real;
Begin
      F (X):= sqrt (X * X – 4) – (2 + X) / (X – 5.5)
End;

{Основной блок программы:}
BEGIN
    ClrScr;
      Writeln (‘* Табулирование функции
              Y = sqrt (X * X – 4) – (2 + X) / (X – 5.5) *’);
{Ввод исходных данных с контролем A < B и H < B:}
      Repeat
              Write (‘Введите начало интервала = ’);
              Readln (A);
              Write (‘              конец интервала = ’);
              Readln (B);
              Write (‘                                  шаг = ’);
              Readln (H)
      Until (A < B) and (H < B);
{Вычисление значений функции и вывод результатов:}
      Writeln (‘      X        Y’);
      X:= A;    {задание начального значения X}
      Repeat
          If (X * X – 4 >= 0) and (X – 5.5 <> 0)
                                  then
                                    begin
                                        Y:= F (X);
                                        Writeln (X:8:2, ‘    ’, Y:8:2)
                                    end
                                  else Writeln (X:8:2, ‘    ’, ‘разрыв’);
```

```
            X:= X + H;
        Until X > B;
    Readln;
END.
```

Протокол

```
* Табулирование функции Y = sqrt (X * X – 4) – (2 + X) / (X – 5.5) *
Введите начало интервала = -0.5
              конец интервала = 5.5
                        шаг = 1.5
   X       Y
  0.50   разрыв
  1.00   разрыв
  2.50   3.00
  4.00   7.46
  5.50   разрыв
```

Комментарий

Если возникает необходимость вычислить количество точек на интервале, то можете воспользоваться соотношением

$$N = \text{целая часть}\left[\frac{b-a}{h}\right] + 1$$

с программной реализацией

```
N:= Trunc ((b – a) / h) + 1;
```

В этом случае можно организовать цикл **For I:= 1 to N do**, а значение аргумента *X* формировать в теле цикла:

```
X:= A – H;
For I:= 1 to N do
            begin
                  X:= X + H;
                  Y:= F (X);
                  Writeln (X:8:2, ‘    ’, Y:8:2)
            end;
```

Глава 2. Типовые алгоритмы обработки одномерных массивов

- **Задание значений и вывод элементов массива**
- **Вычисление суммы и произведения всех элементов массива**
- **Обработка элементов массива по условию**
- **Нахождение наибольшего (наименьшего) элемента массива и его номера**
- **Сортировка элементов массива**
- **Вставка в массив новых элементов**
- **Удаление элемента из массива**

Работа с массивами сводится к действиям над его элементами. Для того, чтобы указать какой элемент в данный момент используется, достаточно задать имя массива и индекс (порядковый номер) используемого элемента. Чаще всего просматриваются *все элементы массива в естественном порядке,* поэтому наиболее употребляемым циклом является цикл **For ... do**.

◆ Задание значений и вывод элементов массива

Обсуждение проблемы

Элементы массива могут быть использованы для вычислений только после того, как их значения будут введены в память компьютера. Элементы массива могут быть заданы:

- как типизированная константа при объявлении её через тип массив;
- введены с клавиатуры;
- сформированы датчиком случайных чисел;
- считаны из файла;

В данном пособии предлагаются первые три способа. Работа с файлами подробно описана в [8].

В дальнейшем все участки программ, реализующие алгоритмы решения задач, будем оформлять в виде стандартных участков процедур и функций, а в теле программы обращаться к объявленным процедурам и функциям, как к предопределенным процессам.

Во всех дальнейших программах будем работать с массивом **Y [1..M]**, и использовать константы и типы, объявленные в виде:

```
Const
    Nm = 25;            {для размерности массива}
    Ns = 8;             {для символьной переменной}
Type
    Tstr = string [Ns];   {строковый тип}
    TM1_r = array [1..Nm] of real;        {тип вещественного
                                           массива}
    TM1_i = array [1..Nm] of integer;     {тип целого массива}
```

Обе константы и все типы должны быть объявлены как глобальные (в разделе объявлений основной программы).

✓ **Задание значений элементов массива как типизированной константы**

Типизированная константа-массив объявляется *в основной программе* после объявления типа массива.

Список численных значений элементов массива записывается в круглых скобках, а их количество должно точно соответствовать количеству элементов массива в объявленном типе, например при типе **TM1_r = array [1..5] of real**

```
Const C: TM1_r = (2, 4, -5.6, 8, -4);
```

✓ **Ввод значений элементов массива с клавиатуры**

Этот способ ввода является наиболее распространённым. Вводимое значение набирается на клавиатуре и отображается на экране. Алгоритм ввода заключается в организации циклического процесса, на каждом шаге которого вводится значение только одного элемента. Если этот элемент обозначить через *Y[I]*, то

параметром цикла будет переменная *I*, изменяющаяся в интервале от *1* до *M* с шагом *1*.

Оформим ввод вещественного массива с клавиатуры в виде процедуры Input_M1 с тремя параметрами:

Procedure Input_M1 (Name: Tstr; Var M: integer; Var Y: TM1_r);

Первый параметр **Name** будет являться переменной символьного типа **Tstr** и представлять имя массива при организации сопроводительных текстов.

Второй параметр **M** типа **integer** будет характеризовать количество элементов массива, подлежащих вводу.

Третий параметр **Y** типа **TM1_r** будет связан с полной характеристикой массива (именем и типом).

Служебное слово **Var** перед вторым и третьим формальными параметрами необходимо, т. к. эти *переменные являются результатами выполнения процедуры и должны передаваться во внешнюю среду* (в основную программу или в другие подпрограммы). Параметр **Name** является для процедуры *исходным данным,* поступающим из внешней среды, поэтому объявлено **без Var**.

Объявление процедуры

```
Procedure Input_M1 (Name: Tstr; Var M: integer; Var Y: TM1_r);
Var I: integer; {I – локальная для процедуры переменная, объ-
                 явленная для организации цикла}
Begin
    Writeln (‘* Ввод элементов массива ’, Name, ‘: *’);
    Write (‘Введите количество элементов = ’);
    Readln (M);
    {Цикл ввода элементов массива:}
    For I:= 1 to M do
        begin
            Write (Name, ‘ [’, I, ‘] = ’); {вывод текста:
                                    <имя>[<значение I>] = }
            Readln (Y[I]);
        end;
End;
```

✓ **Задание значений элементов массива через датчик случайных чисел**

В Турбо Паскале три датчика случайных чисел:

- процедура **Randomize** – включение датчика случайных чисел;
- функция **Random (1),** выбирающая случайное число вещественного типа из отрезка **от 0 до 1**;
- и функция **Random (D)**, выбирающая случайное число целого типа **Word** из отрезка **от 0 до D-1**.

Формируемые значения скрыты от нас. Чтобы увидеть их, включим в алгоритм и вывод элементов массива.

Алгоритм этого способа по структуре аналогичен алгоритму ввода с клавиатуры.

Программную реализацию алгоритма представим процедурой InputR_M1.

Задание значений организуем датчиком **Random (D)**.

Объявление процедуры

```
Procedure InputR_M1 (Name: Tstr; Var M: integer; Var Y: TM1_i);
Var I: integer; {I – локальная для процедуры переменная, объ-
                 явленная для огранизации цикла}
Begin
   Writeln (‘* Ввод элементов массива ’, Name, ‘: *’);
   Write (‘Введите количество элементов = ’);
   Readln (M);
   Randomize;          {включение датчика случайных чисел}
   {Цикл задания элементов массива:}
   For I:= 1 to M do
          begin
             Y[I]:= -13+Random (24);   {задание значений Y[I]
                                        в интервале [-13, 10]}
             Writeln (Name, ‘[’, I, ‘] = ’, Y[I]);   {вывод значе-
                                                      ний Y[I]}
           end;
End;
```

✓ **Вывод значений элементов массива**

Обсуждение проблемы

Алгоритм вывода на экран дисплея значений *M* элементов массива *Y* аналогичен алгоритму ввода с клавиатуры. Разница только в том, что в теле цикла *вместо блока ввода* должен работать *блок вывода.*

Вывод значений элементов массива, сформированных датчиком случайных чисел, уже описан в процедуре InputR_M1.

Процедуру вывода массива, сформированного через ввод с клавиатуры, назовём **Output_M1**. Она будет иметь те же три параметра, что и процедура ввода с клавиатуры. Однако все они будут только исходными данными. Их значения поступают из основной программы, поэтому они объявлены **без Var**.

Объявление процедуры

```
Procedure Output_M1 (Name: Tstr; M: integer; Y: TM1_r);
Var I: integer;
Begin
    Writeln (‘* Вывод элементов массива ’, Name, ‘: *’);
    {Цикл вывода:}
    For I:= 1 to M do
            Writeln (Name,‘[’, I, ‘] = ’, Y[I]:5:2);
End;
```

Задача. Организовать задание массива *X [1..Nx]* датчиком случайных чисел, массива *C [1..5]* как типизированной константы, ввести массив *Z [1..Nz]* с клавиатуры. Вывести массивы *X, C* и *Z.*

```
Program Prim2_1;
Uses Crt;
Type
   Tstr = string[Ns];      {тип строковый}
   TM1_c = array [1..5] of real;   {тип для объявления массива
                                    как типизированной константы}
   TM1_r = array [1..Nm] of real;  {тип вещественного массива}
   TM1_i = array [1..Nm] of integer;     {тип целого массива}
Const
   Nm = 25;                {для размерности массива}
```

```
    Ns = 8;                 {для символьной переменной}
    C: TM1_c = (2, 4, -5.6, 8, -4);     {типизированая константа-
                                         массив}
Var
    Nx, Nz: integer; {количество элементов массива X и Z}
    X: TM1_i;
    Z: TM1_r;

Procedure Input_M1 (Name: Tstr; Var M: integer; Var Y: TM1_r);
Var I: integer; {I – локальная для процедуры переменная, объ-
                явленная для организации цикла}
Begin
    Writeln (‘* Ввод элементов массива ’, Name, ‘: *’);
    Write (‘Введите количество элементов = ’);
    Readln (M);
    {Цикл ввода элементов массива:}
    For I:= 1 to M do
        begin
            Write (Name, ‘[’, I, ‘] = ’); {вывод текста:
                                    <имя>[<значение I>] = };
            Readln (Y[I]);
        end;
End;

Procedure InputR_M1 (Name: Tstr; Var M: integer; Var Y: TM1_r);
Var I: integer; {I – локальная для процедуры переменная, объ-
                явленная для организации цикла}
Begin
        Writeln (‘* Ввод элементов массива ’, Name, ‘: *’);
        Write (‘Введите количество элементов = ’);
        Readln (M);
        Randomize;    {включение датчика случайных чисел}
        {Цикл задания элементов массива:}
        For I:= 1 to M do
            begin
                Y[I]:= -13 + Random (24); {задание значений Y[I]
                                           в интервале [-13, 12]}
```

```
                Writeln (Name, '[', I, '] = ', Y[I]); {вывод значений Y[I]}
            end;
End;

Procedure Output_M1 (Name: Tstr; M: integer; Y: TM1_r);
Var I: integer;
Begin
    Writeln ('* Вывод элементов массива ', Name, ': *');
    {Цикл вывода:}
    For I:= 1 to M do
            Writeln (Name,'[', I, '] = ', Y[I]:5:2);
End;

{Основной блок программы:}
BEGIN
ClrScr;
    {Задание массивов – обращение к процедурам при фактических параметрах, т. е. при параметрах, которые действуют в теле программы:}
    InputR_M1 ('X', Nx, X);
    Input_M1 ('Z', Nz, Z);
    Writeln;
    {Вывод массивов:},
    Output_M1 ('X', Nx, X);
    Output_M1 ('Z', Nz, Z);
    Output_M1 ('C', Nc, C);
END.
```

Протокол

```
* Ввод элементов массива X: *
Введите количество элементов = 3
X[1] = -11
X[2] = 8
X[3] = 6
* Ввод элементов массива Z: *
Введите количество элементов = 4
Z[1] = -3.21
```

```
Z[2] = 5
Z[3] = 3.45
Z[4] = 23
* Вывод элементов массива X: *
X[1] = -11.00
X[2] = 8.00
X[3] = 6.00
* Вывод элементов массива Z: *
Z[1] = -3.21
Z[2] =  5.00
Z[3] =  3.45
Z[4] = 23.00
* Вывод элементов массива C: *
C[1] =  2.00
C[2] =  4.00
C[3] = -5.60
C[4] =  8.00
C[5] = -4.00
```

◆ Вычисление суммы и произведения всех элементов массива

Обсуждение проблемы

Свяжем с суммой переменную **Sum**, с произведением переменную **Pro**. Алгоритм суммирования представляет циклический процесс, на каждом шаге которого к сумме предыдущих элементов прибавляется значение следующего по порядку элемента.

Если начальное значение суммы положить равным *0*, то алгоритм реализуется формулой

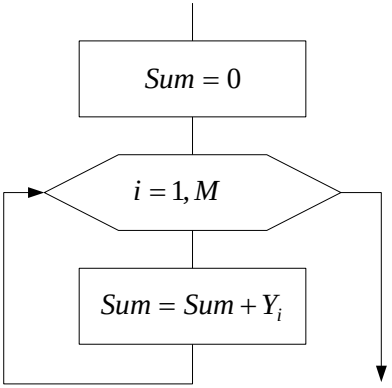
Sum:= Sum + Y[I];

при значении *I*, изменяющемся на интервале от *1* до *M* с шагом *1*.

Для произведения начальное значение полагается равным *1*, а в цикле по параметру *I* вычисление осуществляется по формуле

Pro:= Pro * Y[I];

В силу подобия алгоритмов рассмотрим только фрагмент блок-схемы алгоритма вычисления суммы.



Программную реализацию алгоритма оформим функцией **Sum_M1** с входными параметрами **M** и **Y**.

Имя функции всегда является идентификатором переменной, значение которой передаётся в основную программу, поэтому **Sum_M1 – результат**.

Будем полагать, что элементы массива вещественные, тогда сумма тоже вещественная.

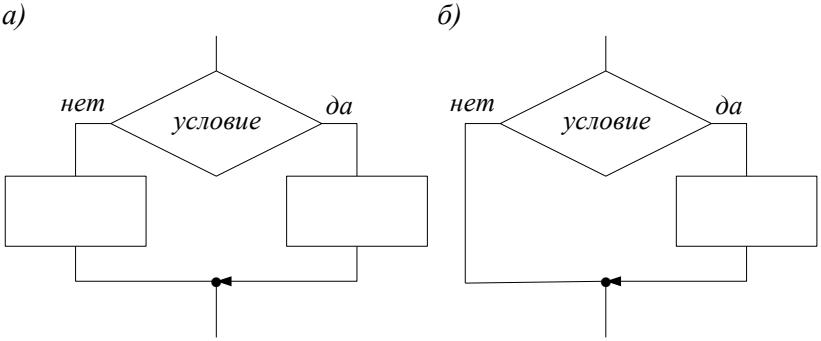
Объявление функции

```
Function Sum_M1 (M: integer; Y: TM1_r): real;
Var I: integer;   {I – параметр счетного цикла}
    Sum: real;   {Sum – вспомогательная переменная}
Begin
     Sum:= 0;
     For I:= 1 to M do
              Sum:= Sum + Y[I];
     Sum_M1:= Sum;
End;
```

◆ Обработка элементов массива по условию

Обсуждение проблемы

В таких задачах цикл по обработке содержит условный оператор в полной **If ... then ... else** (а) или в сокращенной **If ... then** (б) форме:



Часто проверку условия включают составной частью в сложное условие, управляющее выполнением цикла, например:

While **(I <= M) and (Y[I] = A)** do <тело цикла>;

В перечень типовых алгоритмов можно включить:

- ✓ вычисление суммы, произведения, количества элементов, больших (меньших) некоторого числа или равных ему; кратных некоторому числу А;
- ✓ решение вопроса о наличии (отсутствии) элементов с заданными свойствами;
- ✓ замену числом А элементов, удовлетворяющих условию;
- ✓ формирование нового массива из значений или порядковых номеров элементов исходного массива с заданными свойствами;
- ✓ нахождение первого, последнего элемента массива с заданным свойством и его номера и т.д.

✓ **Вычисление количества элементов, удовлетворяющих условию**

Обсуждение проблемы

Этот алгоритм отличается от предыдущего только формулой вычисления результата в теле цикла. Рассмотрим задачу определения **количества элементов, кратных числу А** (оста-

ток от деления элемента массива на число A равен нулю). Задача имеет смысл только для массивов **целого** типа. Алгоритм оформим функцией **Kol_M1**.

Объявление функции

```
Function Kol_M1 (M: integer; Y: TM1_i; A: integer): integer;
Var I, Kol: integer;
Begin
    Kol:= 0;
    For I:= 1 to M do
                If Y[I] mod A = 0 then Inc (Kol);  {Kol:= Kol + 1};
    Kol_M1:= Kol;
End;
```

✓ **Вычисление суммы элементов, удовлетворяющих некоторому условию**

Обсуждение проблемы

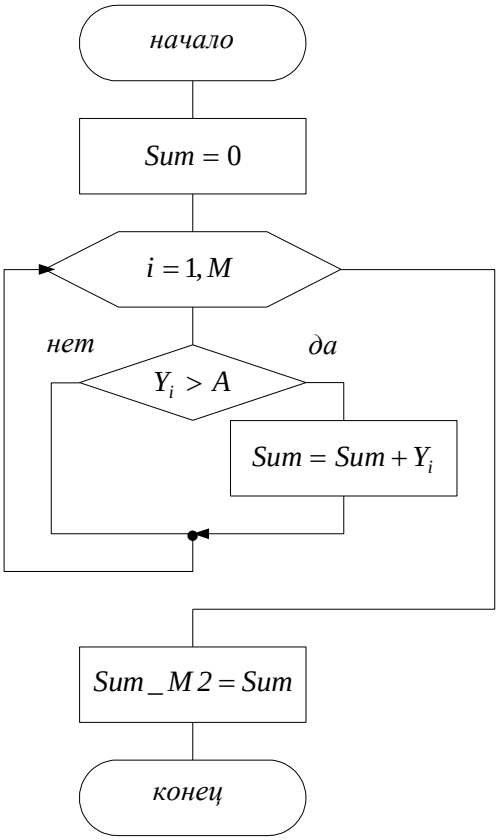
Принципы построения алгоритмов вычисления суммы и произведения всех элементов массива были рассмотрены в предыдущем разделе, поэтому сразу приведём блок-схему вычисления **суммы элементов, больших числа A**.

Программную реализацию оформим функцией **Sum_M2**. Переменные **M**, **A** и массив **Y** представляют исходные данные, **имя функции – результат**.

Объявление функции

```
Function Sum_M2 (M: integer; Y: TM1_r; A: real): real;
Var I: integer;
    Sum: real;
Begin
    Sum:= 0;
    For I:= 1 to M do
                    If Y[I] > A then Sum:= Sum + Y[I];
   Sum_M2:= Sum;
End;
```

Блок-схема алгоритма:



✓ **Решение вопроса о наличии элементов с заданными свойствами**

Обсуждение проблемы

Такая проблема имеет место при решении поисковых задач. В некоторых алгоритмах интересующий нас вопрос может быть решен *на основе анализа конечного результата решения задачи.* Например, ответ на вопрос может быть получен, если при решении предыдущей задачи *в основную программу* включить оператор

```
If Kol_M1 = 0
      then Writeln (‘Элементов кратных ’, A:8, ‘ нет’)
      else Writeln (‘Количество элементов = ’, Kol_M1:8);
```

Однако данный алгоритм даст сбой при его применении к некоторым задачам накопления сумм и произведений элементов. В частности, сумма элементов массива Y = {-3,-4,5,-2,2,1,7}, попавших в интервал [-3, 3], окажется равной нулю, несмотря на то, что элементы с заданными свойствами присутствуют.

Универсальным является алгоритм, в котором в качестве признака наличия или отсутствия элементов с заданными свойствами используется некоторая величина, исполняющая роль **флага**.

Суть алгоритма заключается в просмотре массива до первого элемента с заданными свойствами. Этот прием позволяет досрочно закончить цикл, как только встретится такой элемент.

Продемонстрируем реализацию этого алгоритма на примере решения задачи **о наличии или отсутствии в массиве элементов, кратных некоторому числу A**.

Реализуем алгоритм решения задачи функцией **Flag_M1**. Условие, управляющее работой цикла, представим сложным логическим выражением:

(**I <= M**) and (**Y[I] mod A <> 0**),

которое принимает значение **True** только в случае одновременного выполнения его операндов (составных условий);
операция **mod** – арифметическая операция, определяющая остаток от деления значения *Y[I]* на *A*;

Объявление функции

```
Function Flag_M1 (M: integer; Y: TM1_i; A: integer): Boolean;
Var I: integer;
Begin
    I:= 1;
    While (I <= M) and (Y[I] mod A <> 0) do Inc (I);
    If I > M then        {цикл завершился просмотром всех элементов массива:}
```

```
            Flag_M1:= False
        else        {цикл завершился обнаружением первого
                    элемента с заданными свойствами:}
            Flag_M1:= True;
End;
```

Комментарий

Широкое практическое применение имеет случай решения задачи на проверку наличия четных или нечетных элементов. Число, без остатка делящееся на 2, называется **четным**, в противном случае – **нечетным**. В этом случае рекомендуем использовать стандартную логическую функцию *Odd (X)*, принимающую значение **True** для *нечетного X* и **False** в противном случае.

✓ **Замена числом A элементов, удовлетворяющих некоторому условию**

Обсуждение проблемы

Алгоритм рассмотрим на примере решения задачи: **элементы массива, имеющие дробную часть, заменить числом *A*.**

Естественно, обрабатываемый массив должен быть вещественным. Числом *A* будем заменять элементы, дробная часть которых не равна *0*. Дробная часть вещественного числа определяется стандартной функцией **Frac (Y[I])**. Фрагмент решения будет иметь вид:

```
For I:= 1 to M do
            If Frac (Y[I]) <> 0 then Y[I]:= A;
```

✓ **Формирование нового массива из элементов с заданными свойствами**

Обсуждение проблемы

Рассмотрим алгоритм на примере решения задачи: **из номеров элементов, принадлежащих интервалу *[a, b]*, сформировать новый массив.**

Пусть *Nums* – требуемый массив, *K* – количество его элементов. Заранее величина *K* неизвестна. Положим **перед цик-**

лом значение *K* равным *0. В цикле* будем изменять *K* по закону *K:= K + 1,* как только очередное значение *Y[I]* будет удовлетворять условию:

(Y[I] >= a) and (Y[I] <= b)

Алгоритм оформим процедурой **Mas_Num** с параметрами **M**, **Y [1..M]**, **a**, **b**, **K** и **Nums [1..k]**.

Переменные M, a, b и массив Y [1..M] – исходные данные, переменная **K** и массив **Nums [1..K] – результаты**. M, K и массив Nums [1..K] – целые; a, b и массив Y [1..M] – вещественные.

Объявление процедуры

```
Procedure Mas_Num (M: integer; Y: TM1_r; a, b: real;
                   Var K: integer; Var Nums: TM1_i);
Var I: integer;
Begin
    K:= 0;
    For I:= 1 to M do
        If (Y[I] >= a) and (Y[I] <= b)  then
                   begin
                       Inc (K);
                       {Формируем очередной член массива:}
                       Nums [K]:= I;
                   end;
End;
```

Комментарий

По значению переменной *K*, переданному в основную программу, можно решить вопрос о наличии или отсутствии в массиве элементов, принадлежащих интервалу *[a, b]*.

✓ **Определение номера первого элемента с заданным свойством**

Обсуждение проблемы

Алгоритм решения этой проблемы во многом аналогичен алгоритму решения задачи о наличии или отсутствии в массиве

элементов с заданными свойствами, поэтому приведем только объявление подпрограммы – процедуры, на примере решения задачи: **в массиве найти номер первого элемента, не превосходящего некоторое число *A***.

Формальный параметр – результат выполнения процедуры обозначим через *K*:

$$K = \begin{cases} 0, если\ элементов\ с\ заданным\ свойством\ нет \\ номеру\ первого\ элемента\ с\ заданным\ свойством \end{cases}$$

Объявление функции

```
Procedure FirstEl (M: integer; Y: TM1_r; A: real; Var K: integer);
Var I: integer;
Begin
    K:= 0;     {предполагаем отсутствие элементов с заданным
               свойством}
    I:= 0;
    Repeat
        Inc (I);
    Until (I = M) or (Y[i] <= A)
    if Y[I] <= A then K:= I;  {цикл завершился досрочно или в
                              массиве только последний элемент
                              Y[M] обладает заданным свойством}
End;
```

✓ **Определение номера последнего элемента с заданным свойством**

Обсуждение проблемы

Алгоритм решения этой проблемы можно реализовать двумя способами:

- циклом **For … do**, просматривая ***все*** элементы массива
- циклом **Repeat … Until**, просматривая элементы массива в обратном порядке, ***до первого обнаружения элемента*** с заданными свойствами (аналогично предыдущей задачи).

Реализуем алгоритм с помощью цикла **Repeat … Until**:

```
Procedure LastEl (M: integer; Y: TM1_r; A: real; Var K: integer);
Var I: integer;
Begin
    K:= 0;
    I:= M + 1;
    Repeat
        Dec (I); {выполняется оператор I:= I - 1}
    Until (I = 1) or (Y[i] <= A);
    if Y[I] <= A then K:= I; {цикл завершился досрочно или в
                              массиве только первый элемент Y[1]
                              обладает заданным свойством}
End;
```

◆ Нахождение наибольшего (наименьшего) элемента массива и его номера

Обсуждение проблемы

При решении задач, связанных с наибольшими (наименьшими) элементами, часто требуется определять не только наибольшее (наименьшее) значение, но и его номер, а при нескольких наибольших (наименьших) элементах их количество и порядковые номера. Как и в предыдущих пунктах, решение задач будем оформлять в виде стандартных участков процедур или функций.

Обсуждение проблемы проведем для задач нахождения наибольшего элемента и его номера. Обозначим через **Max** наибольший элемент, через **Nmax** – его номер.

При организации цикла в качестве параметра воспользуемся переменной **I**. Перед циклом будем полагать

```
Nmax:= 1;
Max:= Y[1];
```

В цикле производить сравнение каждого элемента, начиная со второго, со значением **Max** и при необходимости переопределять величины **Max** и **Nmax**.

Переменные **M** и массив **Y[M]** – исходные данные, **Max** и **Nmax** – результаты.

Оформим алгоритм процедурой **Max_Num**.

Объявление процедуры

```
Procedure Max_Num (M: integer; Y: TM1_r; Var Max: real;
                   Var Nmax: integer);
Var I: integer;
Begin
    Nmax:= 1;
    Max:= Y[1];
    For I:= 2 to M do
            If Y[I] > Max then
                    begin
                        Nmax:= I;
                        Max:= Y[I]
                    end;
End;
```

◆ Сортировка элементов массива

Обсуждение проблемы

К задачам сортировки массивов можно отнести задачи:

- ✓ размещение элементов в порядке возрастания (убывания) их значений;
- ✓ перестановки элементов массива в обратном порядке;
- ✓ различные перестановки элементов по условию и т. д.

Почти во всех алгоритмах сортировки требуется поменять местами два элемента массива, пусть **Y[K1]** и **Y[K2]**. Алгоритм реализуем на основе процедуры **Swap (Var A: real; Var B: real)**, выполняющей обмен значениями простых переменных A и B.

Объявление процедуры

```
Procedure Swap (Var A: real; Var B: real);
Var C: real; {C – вспомогательная переменная}
Begin
    C:= A;
    A:= B;
    B:= C;
End;
```

Оператор вызова этой подпрограммы для элементов массива будет иметь вид:

Swap (Y[K1], Y[K2]);

- ✓ **Размещение элементов в заданном порядке**

Рассмотрим алгоритм решения задачи: **элементы массива расположить в порядке: отрицательные, нулевые, положительные**.

Обсуждение проблемы

Самое простое решение задачи заключается в организации двух последовательных циклов.

В первом цикле, начиная с первого элемента, проводится проверка элементов на отрицательность, и первый отрицательный элемент меняется местами с первым элементом массива. Затем следующий отрицательный элемент меняется местами со вторым элементом массива и т.д. В результате первые **K** элементов массива будут содержать отрицательные значения.

Во втором цикле, просматривается усечённый массив с ***K+1-го*** элемента и до **M.** Проводится проверка его элементов на равенство нулю. Первый нулевой элемент этой части массива меняется местами с *K+1*-ым, второй – с *K+2*-ым и т.д. По завершенiь этого цикла начало усечённого массива будет занято нулевыми элементами, а конец – положительными.

Алгоритм оформим процедурой **Sort_1**, в которой количество элементов массива **M** и массив **Y [1..M]** выполняют функции как входных, так и выходных данных.

Объявление процедуры

```
Procedure Sort_1 (Var M: integer; Var Y: TM1_r);
Var I, K, J: integer;      {I – параметр цикла; K, J – вспомогательные переменные}
Begin
    K:= 1;
    For I:= 1 to M do
        If (Y[I] < 0) and (I <> K)
```

```
                    then
                        begin
                            Swap (Y[I], Y[K]);
                            Inc (K);
                        end;
    J:= K + 1;
    For I:= J to M do
        If (Y[I] = 0) and (I <> J)
                    then
                        begin
                            Swap (Y[I], Y[J]);
                            Inc (J);
                        end;
End;
```

✓ **Сортировка элементов в порядке возрастания**

Обсуждение проблемы

Существует довольно много различных методов сортировки, отличающихся друг от друга степенью эффективности, под которой понимается количество сравнений и количество обменов, произведенных в процессе сортировки.

Наиболее простыми из них являются **метод простого выбора** и **метод простого обмена**.

Для достижения цели методом простого выбора будем действовать следующим образом:

1. выбираем наименьший элемент массива;
2. меняем местами найденный элемент с первым элементом и исключаем его из дальнейших просмотров;
3. среди оставшихся (*M-1*) элементов, т. е. в усеченном массиве, опять выбираем наибольший элемент, меняем его местами со вторым элементом, исключаем и его из дальнейших просмотров, и так далее, пока в усеченном массиве не останется только один элемент – самый большой, уже стоящий на своем месте.

Реализация метода представляет собой структуру *вложенных циклов*. Внешнему циклу поручим управлять номером просмотра, а внутреннему – поиск наименьшего элемента в усечен-

ном массиве. Обозначим через *J* – параметр внешнего цикла, а через *I* – параметр внутреннего цикла. Оформим метод процедурой **Sort_2**.

Объявление процедуры Sort_2

```
Procedure Sort_2 (Var M: integer; Var Y: TM1_r);
Var I, J: integer;          {I, J – параметры циклов}
    Num: integer;           {Num – номер наименьшего элемента}
    Min: real;              {Min – значение наименьшего элемента в
                             усеченном массиве}
Begin
    {Цикл по номеру просмотра:}
    For J:= 1 to M-1 do
        begin               {J}
            Num:= J;  Min:= Y[J];
            {цикл поиска наименьшего элемента в усеченном
            массиве:}
            For I:= J + 1 to M do
                If Y[I] < Min then
                                    begin
                                        Min:= Y[I];
                                        Num:= I;
                                    end;
            {Перестановка элементов Y[J] и Y[Num]:}
            Swap (Y[J], Y[Num]);
        end;                {J}
End;
```

Метод *простого обмена* заключается в последовательных просмотрах массива от начала к концу и обмене местами соседних, «неправильно» расположенных элементов, то есть таких, что $Y_i > Y_{i+1}$.

Опишем его подробнее.

Просмотрим весь массив, начав с первой пары элементов (Y_1 и Y_2), если $Y_1 > Y_2$, то меняем их местами, иначе оставляем на месте. Затем берем вторую пару элементов (Y_2 и Y_3), если

$Y_2 > Y_3$, то меняем их местами и так далее. В результате максимальный элемент переместится в конец массива.

Поскольку самый большой элемент уже находится на своем месте, то теперь будем просматривать часть массива без этого элемента, то есть *с первого до (M-1)-го* элемента. Повторив предыдущие действия для этой части массива, переместим второй по величине элемент массива в конец рассматриваемой части, то есть на *(M-1)-е* место всего массива.

Эти действия будем повторять до тех пор, пока количество элементов в просматриваемой части массива не уменьшится до двух. Последнее сравнение позволит упорядочить и эти элементы. При сортировке этим методом необходимо выполнить *M-1* просмотр массива.

Реализацию метода оформим процедурой **Sort_3**.

Объявление процедуры Sort_3

```
Procedure Sort_3 (Var M: integer; Var Y: TM1_r);
Var I, J: integer;                    {I, J – параметры циклов}
Begin
    For J:= 1 to M-1 do               {цикл по номеру просмотра}
          For I:= 1 to M-J do
                If Y[I+1] > Y[I]
                                      then {перестановка элементов:}
                                          Swap (Y[I], Y[I+1]);
End;
```

◆ Вставка в массив новых элементов

Обсуждение проблемы

Чтобы вставить в массив новый элемент, необходимо:

- указать место вставки – номер элемента, перед которым будем вставлять новый элемент;
- сдвинуть *вправо* все элементы массива, расположенные правее места вставки;
- на освободившееся место поставить новый элемент.

При вставке элемента количество элементов массива увеличивается на *1*.

Пусть **M** – размерность массива;

K – место вставки;

A – вставляемый элемент.

Для решения задачи разработаем процедуру вставки **InsM1**, для которой K и A – исходные данные, **M** и массив **Y** представляют как **исходные данные**, так и **результаты**.

Объявление процедуры InsM1

```
Procedure InsM1 (K: integer; A: real; Var M: integer;
                 Var Y: TM1_r);
Var I: integer;  {I – параметр цикла}
Begin
    M:= M + 1;
    I:= M;
    Repeat
      Y[I]:= Y[I-1];
      Dec (I);
    Until I = K;
    Y[K]:= A;
End;
```

◆ Удаление элемента из массива

Обсуждение проблемы

Для удаления элемента из массива необходимо:

- указать его номер K;
- сдвинуть все последующие элементы, начиная с *K+1*, на одну позицию *влево*.

При удалении элемента количество элементов в массиве уменьшается на *1*.

Оформим описанный процесс процедурой **DelM1**.

Объявление процедуры

```
Procedure DelM1 (K: integer; Var M: integer;
                 Var Y: TM1_r);
Var I: integer;  {I- параметр цикла}
Begin
    I:= K;
```

```
    Repeat
        Y[I]:= Y[I+1];
        Inc (I);
    Until I = M;
    M:= M-1;
End;
```

Глава 3. Типовые алгоритмы обработки двумерных массивов

- **Задание значений и вывод элементов массива**
- **Нахождение суммы, произведения всех элементов массива**
- **Обработка элементов массива по условию**
- **Нахождение наибольшего (наименьшего) элемента массива**
- **Обработка с целью получения одномерных массивов**
- **Обработка части массива**
- **Перестановка элементов и частей массива**

Обсуждение проблемы

Двумерный массив представляет матрицу, имеющую **M** строк и **N** столбцов. Положение каждого элемента массива определяется номером строки **I** и номером столбца **J**, на пересечении которых расположен этот элемент. Алгоритмы обработки двумерных массивов отличаются от соответствующих алгоритмов обработки одномерных только тем, что представляют вложенные циклы. При обработке **по строкам** внешний цикл организуется по **I**, внутренний по **J**. При обработке **по столбцам** – внешний по **J**, внутренний по **I.** Наиболее употребляемым циклом является **цикл For.**

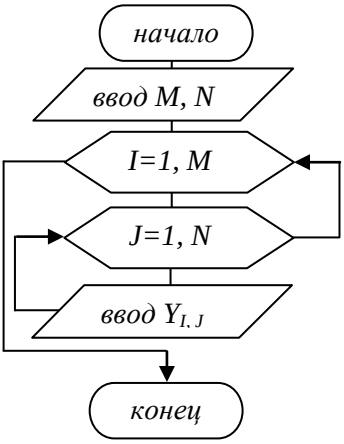
В данной главе, в качестве примера организации циклов при обработке двумерных массивов, приводится только одна блок-схема, описывающая алгоритм ввода элементов массива с клавиатуры.

Рассматриваемые алгоритмы, как и в главе 2, будем оформлять процедурами или функциями с использованием следующих констант и типов:

```
Const
   N = 25; M = 30;      {для объявления размерности массива}
   Ns = 8;              {для символьной переменной – имени массива}
Type
   Tstr = string [Ns];  {строковый тип}
   {Типы одномерных массивов:}
   TM1_r = array [1..M] of real;
   TM1_i = array [1..M] of integer;
   {Типы двумерных массивов:}
   TM2_r = array [1..M, 1..N] of real;      {тип вещественного
                                             массива}
   TM2_i = array [1..M, 1.. N] of integer;  {тип целого массива}
```

◆ Задание значений и вывод элементов массива

✓ Ввод с клавиатуры



Алгоритм ввода с клавиатуры можно реализовать вводом по строкам, как на блок-схеме или вводом по столбцам. Программную реализацию можно огранизовать так, что вводимые элементы будут отображаться на экране:

- по одному элементу в каждой строке;
- в виде матрицы.

Ввод при отображении данных по одному элементу в каждой строке оформим процедурой **Input_M2_1**, а ввод в виде матрицы оформим подпрограммой **Input_M2_2**.

Объявление процедуры Input_M2_1

```
Procedure Input_M2_1 (Name: Tstr; Var M: integer;
                      Var N: integer; Var Y: TM2_r);
Var I, J: integer;
Begin
    Writeln ('* Ввод элементов массива ', Name, ': *');
    Write ('Введите количество строк      = '); Readln (M);
    Write ('Введите количество столбцов = '); Readln (N);
    {Цикл построчного ввода элементов массива:}
    Writeln ('Вводите элементы строки через пробел, в
              конце [Enter]');
    For I:= 1 to M do
      begin                          {I}
        Write ('Строка ',I, ':');    {вывод текста:
                                      Строка <номер>:}
        For J:= 1 to N do
                                  Readln (Y[I, J]);
      end;                         {I}
End;
```

Протокол

```
* Ввод элементов массива X *
Введите количество строк      = 2
Введите количество столбцов = 3
Вводите элементы строки через пробел
Строка 1: 4 5.6 89
Строка 2: 44 -5 6.56
```

Комментарий

Из протокола видим, что процедура **Input_M2_1** хотя и отображает массив как бы в виде матрицы, но при разной значности элементов массива сдвигает столбцы.

Наилучший вариант ввода основан на использовании процедуры управления курсором **GoToXY (nx, ny: byte)** модуля **Crt**, где **ny –** номер строки экрана, **nx –** номер позиции курсора в строке. Стандартно величина **ny** изменяется от **1** до **25**, **nx** – от **1** до **80**.

Оформим этот вариант процедурой **Input_M2_2**.

Объявление процедуры Input_M2_2

```
Procedure Input_M2_2 (Name: Tstr; Var M: integer;
                      Var N: integer; Var Y: TM2_r);
Var I, J: integer;
    nx, ny: byte; {nx – номер строки экрана,
                   ny – номер позиции в строке}
Begin
   ClrScr;      {Очистка экрана}
      Writeln (‘* Ввод элементов массива ’, Name, ‘: *’);
      Write (‘Введите количество строк      = ’);  Readln (M);
      Write (‘Введите количество столбцов = ’);  Readln (N);
      {Определение номера строки экрана: первые 3 строки
      заняты информацией предыдущих операторов вывода,
      четвёртую строку оставим пустой, размещение матрицы
      начнём с пятой строки:}
      ny:= 5;
      For I:= 1 to M do
          begin                {I}
             nx:= 6; {задание начального номера позиции}
             For J:= 1 to N do
                 begin        {J}
                    GoToXY (nx, ny);{первое число каждой
                                     строки размещаем с 6-
                                     ой позиции}
                    Readln (Y[I, J]);
                    nx:= nx + 7; {следующие – с 13, 20 и т. д.}
                 end;         {J}
             Inc (ny);        {переход на следующую строку}
          end;                {I}
End;
```

Протокол

```
* Ввод элементов массива X *
Введите количество строк      = 2
Введите количество столбцов = 3
        4       5.6    89
        44      -5     6.56
```

✓ **Задание значений в виде типизированной константы**

При задании в виде типизированной константы список численных значений элементов массива записывается по строкам. Элементы каждой строки записываются в круглые скобки. Количество строк должно точно соответствовать количеству строк массива в объявленном типе, например:

```
Type TM2_r = array [1..2, 1..3] of real;
Const C2: TM2_r = ((3, -8.6, 5), (0, 4, -5));
```

Задание массива *датчиком случайных чисел* рассматривать не будем в надежде, что Вы сможете это сделать сами.

✓ **Вывод значений элементов массива**

Блок-схема вывода аналогична вводу, с той лишь разницей, что блок ввода надо заменить на блок вывода. Для вывода массива в виде матрицы достаточно поле вывода сформировать с учётом значности значений элементов и пробелов между ними. Вывод оформим процедурой **Output_M2**.

Объявление процедуры

```
Procedure Output_M2 (Name: Tstr; M, N: integer; Y: TM2_r);
Var I, J: integer;
Begin
      Writeln ('* Вывод элементов массива ', Name, ': *');
      {Цикл вывода:}
      For I:= 1 to M do
          begin{I}
             For J:= 1 to N do
                          Write (Y[I, J]:7:2);{отводим 7 позиций
                                                 под элемент}
             Writeln;     {переход на новую строку}
           end;{I}
End;
```

Вывод элементов в виде матрицы, размещенной с позиций *nx*, *ny* экрана, предлагаем выполнить самостоятельно.

- **Нахождение суммы, произведения всех элементов массива**

Обсуждение проблемы

Из предыдущего материала известно, что в таких алгоритмах начальные значения суммы – **Sum**, произведения – **Pro**, задаются перед циклом операторами:

Sum:= 0; Pro:= 1;

В теле цикла вычисления организуются по формулам:

Sum:= Sum + Y[I,J];
Pro:= Pro * Y[I,J];

Представим алгоритм вычисления суммы в виде функции **Sum_M2**, произведения в виде функции **Pro_M2**.

Объявление функции Sum_M2

```
Function Sum_M2 (M, N: integer; Y: TM2_r): real;
Var I, J: integer;
    Sum: real;
Begin
        Sum:= 0;
        For I:= 1 to M do
        For J:= 1 to N do
                        Sum:= Sum + Y[I, J];
        Sum_M2:= Sum;
End;
```

Объявление функции Pro_M2

```
Function Pro_M2 (M, N: integer; Y: TM2_r): real;
Var I, J: integer;
    Pro: real;
Begin
        Pro:= 0;
        For I:= 1 to M do
        For J:= 1 to N do
                        Pro:= Pro * Y[I, J];
        Pro_M2:= Pro;
End;
```

◆ Обработка элементов массива по условию

Обсуждение проблемы

В типовые алгоритмы обработки массива по условию можно включить нахождение суммы, произведения, количества элементов массива, удовлетворяющих (не удовлетворяющих) некоторому условию. В силу их подобия, рассмотрим только один:

✓ **Нахождение количества элементов, принадлежащих интервалу *[a, b]***

Алгоритм представим функцией **Kol_M2**.

Объявление функции

```
Function Kol_M2 (M, N: integer; Y: TM2_r; A, B: real): integer;
Var I, J: integer;
    Kol: integer;
Begin
        Kol:= 0;
        For I:= 1 to M do
        For J:= 1 to N do
              If (Y[I, J] >= a) and (Y[I, J] <= b) then Kol:= Kol + 1;
        Kol_M2:= Kol;
End;
```

◆ Нахождение наибольшего (наименьшего) элемента массива

Обсуждение проблемы

Обозначим наибольший элемент через **Max**, его место в массиве свяжем с переменными **Ni** – номер строки, **Nj** – номер столбца. Алгоритм представим процедурой **Max_M2**. Начальные значения **Max**, **Ni** и **Nj** зададим перед внешним циклом операторами:

```
Ni:= 1;    Nj:= 1;    Max:= Y[Ni, Nj];
```

Объявление процедуры

```
Procedure Max_M2 (M, N: integer; Y: TM2_r; Var Ni, Nj: integer;
                  Var Max: real);
```

```
Var I, J: integer;
Begin
      Ni:= 1;
      Nj:= 1;
      Max:= Y[Ni, Nj];
      For I:= 1 to M do
      For J:= 1 to N do
                If Y[I,J] > Max then
                                  begin
                                      Max:= Y[I,J];
                                      Ni:= I;
                                      Nj:= J;
                                  end;
End;
```

◆ Обработка с целью получения одномерных массивов

Обсуждение проблемы

При обработке двумерных массивов часто требуется найти сумму, произведение или другие величины элементов строк или столбцов. Результатами в таких алгоритмах являются *одномерные массивы.* При обработке строк длина массива равна количеству строк матрицы, при обработке столбцов – количеству её столбцов. В качестве демонстрационной рассмотрим задачу **нахождения суммы чётных элементов каждой строки.**

Обозначим массив сумм по строкам через **Sum [1..M]**. Принципиальное отличие рассматриваемого алгоритма от алгоритма вычисления суммы всех элементов массива состоит в том, что начальное значение сумм надо задавать перед внутренним циклом оператором **Sum[I]:= 0;**

Оформим алгоритм процедурой **Sum_S**.

Объявление процедуры

```
Procedure Sum_S (M, N: integer; Y: TM2_r; Var Sum: TM1_r);
Var I, J: integer;
Begin
    For I:= 1 to M do
        begin
            {Задание начальных значений сумм:}
```

```
            Sum[I]:= 0;
            For J:= 1 to N do
                  If  Y[I, J] mod 2 = 0 then
                      Sum[I]:= Sum[I] + Y[I, J];
      end;
End;
```

◆ Обработка части массива

Обсуждение проблемы

В набор типовых алгоритмов этого раздела включены алгоритмы обработки элементов двумерных массивов, отображающих квадратные матрицы (*M = N*), расположенных на главной диагонали, а также выше или ниже её.

- Для элементов, принадлежащих главной диагонали их индексы равны между собой

  I = J; **Y[I, I] = Y[J, J]**,

  поэтому при разработке алгоритмов достаточно построить только один цикл или по строкам, или по столбцам, и в его теле использовать переменную Y[I, I] или Y[J, J].
- Индексы элементов масива, расположенных *ниже* главной диагонали изменяются по закону

  I = 2..M, J = 1..I-1;

  и заголовки циклов For … do будут иметь вид:

```
For I:= 1 to M do
…
    For J:= 1 to I-1 do
```

- Индексы элементов матрицы, расположенных *выше* главной диагонали изменяются по закону:

  I = 1..M-1, J = I+1..N;

и заголовки цикла For … do будут иметь вид:

```
For I:= 1 to M-1 do
…
    For J:= I+1 to N do
```

В качестве демонстрационной рассмотрим задачу: **элементы массива, расположенные ниже главной диагонали, заменить величиной *A***.

Обсуждение проблемы

Оформим алгоритм решения процедурой **Insert_Nd**, для которой **M**, **N**, **A** – исходные данные, массив **Y** является исходным и результирующим.

Объявление процедуры

```
Procedure Insert_Nd (M, N: integer; A: real; Var Y: TM2_r);
Var I, J: integer;
Begin
      For I:= 2 to M do
          For J:= 1 to I-1 do
                              Y[I, J]:= A
End;
```

◆ Перестановка элементов и частей массива

Обсуждение проблемы

При обработке двумерных массивов нередко возникают задачи, в которых требуется переставить отдельные элементы, целые строки или столбцы. Идея организации перестановок подробно была изложена в предыдущей главе, поэтому сразу перейдём к записи алгоритмов для двумерных массивов.

Алгоритм перестановки двух элементов **Y[K1, L1]** и **Y[K2, L2]** оформим процедурой **Swap_YY**, в которой **N**, **M**, **K1**, **K2**, **L1**, **L2** – исходные данные, массив **Y** одновременно и исходный и результирующий.

Объявление процедуры

```
Procedure Swap_YY (M, N, K1, K2, L1, L2: integer;
                   Var Y: TM2_r);
Var R: real;
Begin
    R:= Y[K1, L1];
    Y[K1, L1]:= Y[K2, L2];
    Y[K2, L2]:= R;
End;
```

Алгоритм перестановки строк оформим процедурой **Swap_RR**, в параметры которой, кроме **N**, **M**, **Y**, включим и номера переставляемых строк **K1** и **K2**.

Объявление процедуры

```
Procedure Swap_RR (M, N, K1, K2: integer; Var Y: TM2_r);
Var J: integer;
Begin
      For J:= 1 to N do
                  {Меняются местами J-ый элемент строки
                  K1 с J-ым элементом строки K2:}
                  Swap_YY (M, N, K1, K2, J, J, Y)
End;
```

Алгоритм перестановки столбцов оформим процедурой **Swap_CC**, с параметрами **N**, **M**, **Y**, **L1** и **L2**.

Объявление процедуры

```
Procedure Swap_CC (M, N, L1, L2: integer; Var Y: TM2_r);
Var I: integer;
Begin
    For I:= 1 to M do
                  {Меняются местами I элемент столбца L1
                  с I элементом столбца L2:}
                  Swap_YY (M, N, I, I, L1, L2, Y)
End;
```

Алгоритм транспонирования матрицы (замена строк столбцами) применим только для квадратных матриц.

Оформим его процедурой **Swap_Tr** с параметрами **N** и **Y**.

Объявление процедуры

```
Procedure Swap_Tr (N: integer; Var Y: TM2_r);
Var I, J: integer;
    R: real;
Begin
    For I:= 1 to M do
        For J:= 1 to N do
            If I <> J then      {элементы главной диагонали не переставляются}
            R:= Y[I, J];        {элемент строки пересылается в R}
            Y[I, J]:= Y[J, I];  {элемент столбца пересылается в строку}
            Y[J, I]:= R;        {элемент строки пересылается в столбец}
End;
```

Приложение 1

Таблица кодировки символов

Символы, имеющие коды от 0 до 127:

0-	16- ►	32-	48- 0	64- @	80- P	96- `	112- p
1- ☺	17- ◄	33- !	49- 1	65- A	81- Q	97- a	113- q
2- ☻	18- ↕	34- "	50- 2	66- B	82- R	98- b	114- r
3- ♥	19- !!	35- #	51- 3	67- C	83- S	99- c	115- s
4- ♦	20- ¶	36- $	52- 4	68- D	84- T	100- d	116- t
5- ♣	21- §	37- %	53- 5	69- E	85- U	101- e	117- u
6- ♠	22- ▬	38- &	54- 6	70- F	86- V	102- f	118- v
7- •	23- ↨	39- '	55- 7	71- G	87- W	103- g	119- w
8-	24- ↑	40- (	56- 8	72- H	88- X	104- h	120- x
9-	25- ↓	41-)	57- 9	73- I	89- Y	105- i	121- y
10-	26- →	42- *	58- :	74- J	90- Z	106- j	122- z
11- ♂	27- ←	43- +	59- ;	75- K	91- [	107- k	123- {
12- ♀	28- ∟	44- ,	60- <	76- L	92- \	108- l	124- |
13-	29- ↔	45- -	61- =	77- M	93-]	109- m	125- }
14- ♫	30- ▲	46- .	62- >	78- N	94- ^	110- n	126- ~
15- ☼	31- ▼	47- /	63- ?	79- O	95- _	111- o	127- ⌂

Продолжение приложения 1

Символы, имеющие коды от 128 до 255:

128- А	144- Р	160- а	176- ▒	192- └	208- ╨	224- р	240- Ё
129- Б	145- С	161- б	177- ▒	193- ┴	209- ╤	225- с	241- ё
130- В	146- Т	162- в	178- ▓	194- ┬	210- ╥	226- т	242- Є
131- Г	147- У	163- г	179- │	195- ├	211- ╙	227- у	243- є
132- Д	148- Ф	164- д	180- ┤	196- ─	212- ╘	228- ф	244- Ї
133- Е	149- Х	165- е	181- ╡	197- ┼	213- ╒	229- х	245- ї
134- Ж	150- Ц	166- ж	182- ╢	198- ╞	214- ╓	230- ц	246- Ў
135- З	151- Ч	167- з	183- ╖	199- ╟	215- ╫	231- ч	247- ў
136- И	152- Ш	168- и	184- ╕	200- ╚	216- ╪	232- ш	248- °
137- Й	153- Щ	169- й	185- ╣	201- ╔	217- ┘	233- щ	249- •
138- К	154- Ъ	170- к	186- ║	202- ╩	218- ┌	234- ъ	250- ·
139- Л	155- Ы	171- л	187- ╗	203- ╦	219- █	235- ы	251- √
140- М	156- Ь	172- м	188- ╝	204- ╠	220- ▄	236- ь	252- №
141- Н	157- Э	173- н	189- ╜	205- ═	221- ▌	237- э	253- ¤
142- О	158- Ю	174- о	190- ╛	206- ╬	222- ▐	238- ю	254- ▪
143- П	159- Я	175- п	191- ┐	207- ╧	223- ▀	239- я	255-

Символы с кодами 7, 8, 10 и 13 являются служебными и на экран не отображаются. Каждому служебному символу соответствует определенное **действие**.

Код символа Действие

07 Короткий звуковой сигнал

08 Перемещение курсора влево на одну позицию

10 Перемещение курсора на следующую строку без изменения его положения в колонке

13 Перемещение курсора в начало текущей строки

ЛИТЕРАТУРА

1. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс. Учебное пособие. – М.: Нолидж, 1997. – 616 с., ил.
2. И.А. Бабушкина, Н.А. Бушмелёва, С.М. Окулов, С.Ю. Черных. Практикум по Турбо Паскалю. Учебное пособие по курсам «Информатика и вычислительная техника», «Основы программирования». – Москва, АБФ, 1998. – 384 с.
3. Гусева А.И. Учимся программировать: PASCAL 7.0. Задачи и методы их решения. – 2-е изд. перераб. и доп. – М.: «Диалог-МИФИ»,1999.-256 с.
4. А. Епанешников, В. Епанешников. Программирование в среде Turbo Pascal 7.0. – 4-е изд., испр. и дополн. – М.: «Диалог-МИФИ», 2000. – 367 с.
5. Культин Н.Б. Turbo Pascal в задачах и примерах. – СПб.: БХВ-Санкт-Петербург, 2000. – 256 с.: ил.
6. Климов Ю.С. и др. Программирование в среде Turbo Pascal 6.0: Справ. пособие / Ю.С. Климов, А.И. Касаткин, С.М. Мороз. – Мн.: Выш. шк., 1992. – 158 с.: ил.
7. Поляков Д.Б., Круглов И.Ю. Программирование в среде Турбо Паскаль (версия 5.5): Справ.-метод. пособие. – М.: Изд-во МАИ, 1992. – 576 с.
8. Программирование на Турбо Паскале с использованием сложных структур языка / Г.И. Станевко: Кемеровский технологический институт пищевой промышленности. – Кемерово, 2000. – 64 с.

УЧЕБНОЕ ИЗДАНИЕ

Станевко Галина Ивановна
Колесникова Татьяна Геннадьевна
Давыденко Вероника Анатольевна

ИНФОРМАТИКА: ПРОГРАММИРОВАНИЕ ОТ ОСНОВ К ТУРБО ПАСКАЛЮ

Учебное пособие

Для студентов вузов

Зав. редакцией: А.С. Обвинцева
Редактор: Н.В. Шишкина
Технические редакторы: Т.В. Васильева, Е.К. Матвеева
Художественный редактор: Л. П. Токарева

ЛР № _____ от _____
Подписано в печать _____ Формат $60\times84^{1/16}$
Бумага типографская. Гарнитура Times
Уч.-изд. л. ____. Тираж _____ экз.
Заказ № ____

Оригинал-макет изготовлен в редакционно-издательском центре
Кемеровского технологического института пищевой промышленности
650056, г. Кемерово, б-р Строителей, 47

ПЛД № _____ от _____
Отпечатано в лаборатории множительной техники
Кемеровского технологического института пищевой промышленности
650056, г. Кемерово, ул. Красноармейская, 52