МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИННОВАЦИОННЫХ ТЕХНОЛОГИЙ И ПРДПРИНИМАТЕЛЬСТВА

КАФЕДРА «ПРИКЛАДНАЯ ИНФОРМАТИКА»

Питеркин В.М.

ИНФОРМАТИКА И ПРОГРАММИРОВАНИЕ

Учебное пособие

Часть 2

Москва 2011

Питеркин В.М.

Информатики и программирование. Часть 2: учебное пособие/В.М. Питеркин. - М.:РГУИТП, 2011. - 192 с.

Вводятся основные понятия программирования как вида профессиональной деятельности . Рассматриваются общие для всех языков программирования темы: структуры данных, классические управляющие конструкции, способы и механизмы передачи параметров, тесты и принципы их разработки, характеристики качества программы, принципы модульного и объектно-ориентированного программирования. излагаемый материал иллюстрируется на языке программирования Паскаль.

Для студентов , обучающихся по направлению подготовки 230400 «Информационные системы и технологии» и 230700 «Прикладная информатика». Может быть рекомендовано также студентам других специальностей на начальных этапах изучения программирования.

Рецензент: д.т.н., профессор Вишнеков А.В., зав.
кафедрой «Вычислительные системы и сети»
(Московский государственный институт
электроники и математики (МИЭМ))

© Российский государственный университет
инновацонных технологий и предпринимательства, 2011
© Кафедра "Прикладная информатика", 2011
© Питеркин В.М., 2011

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ 4

1. ЭТАПЫ РАЗРАБОТКИ ПРОГРАММЫ 5

2. ЯЗЫК ДЛЯ ЗАПИСИ АЛГОРИТМОВ 18

3. ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ ПАСКАЛЬ 33

4. РАЗРАБОТКА АЛГОРИТМОВ МЕТОДОМ ПОШАГОВОЙ ДЕТАЛИЗАЦИИ 47

5. ТЕСТИРОВАНИЕ 65

6. ХАРАКТЕРИСТИКИ КАЧЕСТВА ПРОГРАММЫ 75

7. ТИПОВЫЕ АЛГОРИТМЫ ОБРАБОТКИ МАССИВОВ 83

8. ОБРАБОТКА СИМВОЛЬНОЙ ИНФОРМАЦИИ 93

9. ТИПЫ ДАННЫХ, ЗАДАВАЕМЫЕ ПОЛЬЗОВАТЕЛЕМ 101

10. ФАЙЛЫ 106

11. ПОДПРОГРАММЫ 127

12. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ 138

13. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ ТИПА «ДЕРЕВО» 151

14. МОДУЛИ 163

15. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ 175

ЗАКЛЮЧЕНИЕ 189

БИБЛИОГРАФИЧЕСКИЙ СПИСОК 192

ПРЕДИСЛОВИЕ

Программирование следует рассматривать как специфический вид профессиональной деятельности, направленной на создание программ решения конкретных задач на компьютере, в случае если процесс решения можно представить в виде алгоритма. Несмотря на наличие в настоящее время достаточно большого числа языков программирования и на то обстоятельство, что алгоритм в некоторой степени определяется спецификой того языка программирования, для которого он разрабатывается, процесс разработки программы практически одинаков для любого языка программирования.

Данное пособие и посвящено рассмотрению процесса разработки программы как последовательности действий, которые однотипны (инвариантны) для любого языка программирования. Единственное замечание, которое необходимо сделать к этому утверждению, заключается в том, что здесь речь идет о разработке алгоритмов только на языках процедурного программирования, представителями которых являются широко известные и активно используемые в настоящее время языки Бейсик, Паскаль и Си. В большинстве случаев именно языки этого типа применяются, когда решение задачи можно представить в виде алгоритма как упорядоченной последовательности вычислительных операций.

Программирование - практическая дисциплина, поэтому изложение теоретического материала в пособии сопровождается иллюстрацией применения всех рассматриваемых вопросов на примере языка программирования Паскаль. При этом не ставится цель изложить исчерпывающие сведения о языке, а применяются только те средства языка, которые необходимы для кодирования на нем рассматриваемых алгоритмов. Такой подход объясняется следующими соображениями. Во-первых, данное пособие посвящено изложению основ программирования и ориентировано как на начинающих, так и на лиц, имеющих некоторый, но несистематизированный опыт написания программ. Во-вторых, литературы, в которой приводится полное описание языка Паскаль, вполне достаточно, так что нет необходимости приводить полное описание еще и здесь. В-третьих, многие средства языка не требуются для иллюстрации излагаемого материала, а потребность в них может возникнуть у читателей только после приобретения определенного опыта разработки программ. И еще одно. Изложение материала о языке Паскаль в пособии построено так, чтобы предоставить читателю некоторую схему ознакомления с новым для него языком программирования, воспользоваться которой можно будет впоследствии и при изучении другого нового для него языка.

В заключение считаю необходимым отметить, что основу данного пособия составил материал лекций, которые читались в течение ряда лет в РГУИТП, на кафедре вычислительной техники МИЭМ и на кафедре информационных технологий МАТИ.

1. ЭТАПЫ РАЗРАБОТКИ ПРОГРАММЫ

Процесс разработки программы можно разбить на несколько этапов, последовательности выполнения которых рекомендуется строго придерживаться.

Перечислим их:

1. Постановка задачи
2. Выбор метода решения
3. Разработка внешней спецификации программы (или ее сценария)
4. Разработка алгоритма
5. Кодирование алгоритма на языке программирования
6. Испытания программы на тестах и ее отладка.

Результаты должны быть документированы. Минимальная документация на программу должна включать:

- Внешнюю спецификацию или сценарий
- Систему тестов
- Комментарии в тексте программы на языке программирования.

Рассмотрим содержание отдельных этапов.

1.1. Формальная постановка задачи

Любая задача имеет целью получить некоторые результаты в определенных условиях. Для того чтобы решить задачу, необходимо сформулировать ее. Решение задачи всегда оценивается по конечному результату. И если у нас нет четкого понимания, какими должны быть конечные результаты, то можно получить совсем не то, что предполагалось. Отсюда, умение четко поставить задачу служит залогом успеха.

Постановка задачи, в первую очередь должна строго однозначно указать, «***что дано?»*** и ***«что требуется*** получить»***?***

Постановка задачи может быть конкретной или обобщенной. В **конкретной** задаче исходные условия определяются однозначно, и ее решение состоит в получении конкретных результатов, отвечающих постановке.

Проиллюстрируем сказанное на примере.

Задача: Найти вещественные корни уравнения $x^2-1001*x+1000=0$.

В этой задаче дано конкретное уравнение, и ее решение имеет целью найти пару вещественных чисел, являющихся корнями этого уравнения.

В **обобщенной** постановке допускается определенная вариация исходных условий, а результаты решений, вообще говоря, зависят от этих условий. Поэтому в обобщенной постановке задачи дополнительно определяется, какими должны быть допустимые входные условия и какова зависимость (связь) требуемых результатов от исходных условий.

Та же самая математическая задача в обобщенном виде может быть записана в следующей форме:

```
Задача: найти вещественные корни квадратного уравнения
                 a*x^2+b*x+c=0
Дано:  a, b, c - коэффициенты уравнения [вещественные числа].
Требуется: x1, x2 - корни уравнения [вещественные числа].
Связь: a*x_i^2+b*x_i+c=0,  (i=1,2) - тождество.
При:  a ≠ 0.
```

Можно предложить следующую унифицированную (единообразную) форму записи формальной постановки задачи:

```
Задача: <содержательная-формулировка-задачи>
Дано:   <перечисление-исходных-объектов>
Треб.:  <перечисление-требуемых-объектов>
Связь:  <связи-между-требуемым-и-исходным>
При:    <условия-допустимости-исходных-данных>
```

Угловыми скобками в этой форме выделены переменные части постановки, содержание которых зависит от конкретной проблемы, подлежащей решению.

Прокомментируем содержание составных частей формальной постановки задачи:

<Перечисление-исходных-объектов> - это список исходных данных с разъяснением их содержательной роли и указанием типов принимаемых значений.

Аналогично оформляется **<Перечисление-требуемых-объектов>.**

Эти части формальной постановки выполняют также роль терминологических словарей.

В разделе **<Связи-между-требуемым-и-исходным>** фиксируется формальное описание зависимостей между требуемыми результатами и исходными данными. Эти описания должны быть строгими в такой мере, чтобы по любым конкретным исходным данным и конкретным результатам решения задачи можно было бы однозначно определить, являются ли эти результаты правильными (иначе, адекватны ли результаты исходным данным). Для математических задач формальное описание зависимостей представляется в виде систем уравнений, связывающих исходные данные и результаты. Для ряда задач формальное описа-

ние связи выразить математически не удается, и в этом случае может использоваться словесное описание.

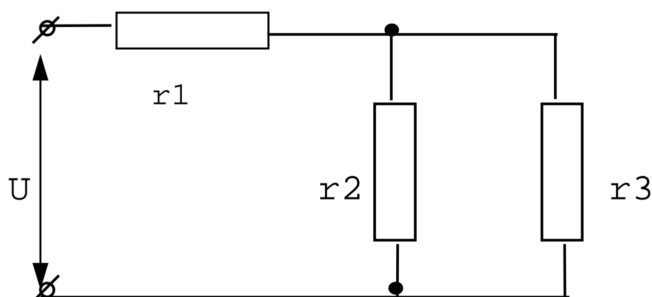
В разделе **<Условия-допустимости-исходных-данных>** описываются ограничения на область допустимых значений для исходных данных. Эти ограничения определяют возможность получения решений поставленной задачи.

Вообще, формальная постановка задачи необходима для того, чтобы получить возможность объективно проверить правильности разработанной программы:

Проверка правильности программы должна проводиться только относительно допустимых исходных данных. Программа считается правильной, если при всех допустимых исходных значениях она дает решения. И, с другой стороны, программа неправильная, если существуют ситуации, при которых ее результаты не соответствуют исходным значениям. Следовательно, условия допустимости исходных данных должны быть строго и полно сформулированы в постановке задачи.

Рассмотрим еще один пример - формальную постановку элементарной физической задачи.

Задача: Определить мощность, потребляемую в приведенной ниже электрической цепи (*Примечание: любое из сопротивлений может быть равным нулю*).



Дано: r1, r2, r3 - сопротивления цепи,
u - входное напряжение.
Треб.: w - потребляемая мощность.
Связь:
w = u*i1 - определение мощности,
i1 = i2 + i3 - уравнение для токов,
u = i1*r1 - уравнение первого контура
(источник питания, r1 и r2),
0 = i2*r2 - i3*r3 - уравнение второго контура
(замкнутый контур r2 и r3),

Здесь
i1,i2,i3 - величины токов, протекающих через r1, r2 и r3 соответственно.

При:

```
    (r1,r2,r3 >= 0)  и  (r1+r2 ≠ 0)  и  (r1+r3 ≠ 0).
```

1.2. Выбор метода решения

Цель данного этапа – определить теоретическую возможность решения и, в таком случае, нахождение формального правила получения решения.

Данный этап плохо формализуется. Это связано с чрезвычайно широким многообразием задач и методов их решения.

Метод решения может быть представлен

- в виде системы формул (безусловной или условной)
- в виде словесного изложения последовательности действий
- в виде их комбинаций.

Описание метода может содержать ограничения на исходные данные, накладываемые методом.

Приведем примеры.

Пример 1. Метод решения задачи «Решение квадратного уравнения» сводится к системе формул:

$$d = b^2 - 4ac$$

$$x_1 = \frac{-d + \sqrt{d}}{2a}$$

$$x_2 = \frac{-d - \sqrt{d}}{2a}$$

Применять их можно только при условии, что

$$a > 0$$

$$d \geq 0$$

Пример 2. Метод решения задачи «Расчет электрической цепи»

Метод можно представить в виде следующей системы формул:

(1) $W = \frac{U}{r_{экв}}$ при $r_{экв} \neq 0$,

а при $r_{экв} = 0$ имеет место короткое замыкание и расчет невозможен.

(2) $r_{экв} = r_1 + r_{23}$

(3) $r_{23} = \frac{r_2 \cdot r_3}{r_2 + r_3}$ при $(r_2 + r_3) \neq 0$

$r_{23} = 0$ при $(r_2 + r_3) = 0$

Примечание. Поскольку это не алгоритм, здесь не обязательно указывать формулы в порядке их вычисления. Важно только, чтобы искомая величина W за счет их применения могла быть вычислена как некая функция от заданных значений напряжения и трех сопротивлений.

Пример 3. Метод решения задачи нахождения наибольшего общего делителя (НОД) двух целых чисел.

В математике известно несколько способов решения этой задачи. Наиболее популярный – разложение исходных чисел на простые сомножители и затем выбор сомножителей, присутствующих в разложении как первого, так и второго чисел.

Реализовать этот метод в виде алгоритма для ЭВМ для начинающих не очень просто. Разработать его самостоятельно будет предложено после получения определенного практического опыта в разработке алгоритмов, а сейчас приведем другой, более простой для понимания и реализации метод – метод Эвклида.

Метод Эвклида позволяет найти НОД двух натуральных чисел и заключается в следующем:

Пусть имеются два натуральных числа. (*Напомним, что натуральные числа – беззнаковые, а минимальное из их равно 1*).

Для нахождения НОД надо многократно (циклически) повторять следующее действие: большее число уменьшить, вычтя из него меньшее. Указанную операцию надо повторять до тех пор, пока оба числа не станут равными. Последнее значение этих чисел и будет наибольшим общим делителем исходных чисел.

Проиллюстрируем метод на примере.
Пусть заданы числа a=20 и b=34. Тогда:

Шаг	a	b	Действие
Исходное состояние	20	34	
1	20	34	Уменьшить b на 20
2	20	14	Уменьшить a на 14
3	6	14	Уменьшить b на 6
4	6	8	Уменьшить b на 6
5	6	2	Уменьшить a на 2
6	4	2	Уменьшить a на 2
7	2	2	Остановка: НОД найден

Обратите внимание, что величина НОД, в принципе, беззнаковая, в связи с чем НОД для чисел (-20, 36) также равен 2. Однако применить описанный метод Эвклида непосредственно нельзя – процесс вычитания меньшего из большего становится бесконечным. Посмотрим, что получится, если применить метод Эвклида для указанных чисел:

Шаг	a	b	действие
Исходное состояние	–20	34	
1	–20	34	Уменьшить b на –20
2	–20	54	Уменьшить b на –20
3	-20	74	Уменьшить b на -20
4	-20	94	. . .

Как видно из таблицы, процесс получается бесконечный, и равенства значений ***a*** и ***b*** не будет никогда. Это связано с тем, что число (–20) является целым, но не натуральным. Однако если взять исходные числа по абсолютной величине, никаких проблем не будет.

Особый случай имеет место, когда хотя бы одно из исходных чисел равно нулю. Ноль входит во множество целых чисел, и поэтому также может быть предъявлен в качестве исходного при программной реализации вычисления НОД. Как видно из алгоритма, НОД в этом случае вычислить невозможно, да в этом и нет необходимости, из-за особого статуса нуля.

С учетом изложенного можно предложить следующий метод решения задачи:

Если оба числа не равны нулю, взять их абсолютные величины и применить алгоритм Эвклида. В противном случае – сообщить об отсутствии решения (Например, выдав сообщение «НОД не определен»).

1.3. Внешняя спецификация программы

Решение задачи на ЭВМ предполагает наличие соответствующей прикладной программы. Процесс решения заключается в подготовке исходных данных, вводе их в ЭВМ и получении результатов выполнения программы. Внешняя спецификация программы – это полное и точное описание результатов ее выполнения для всевозможных входных ситуаций.

Внешняя спецификация программы может рассматриваться, с одной стороны, формальной инструкцией по использованию программы, а с другой – формальным техническим заданием на ее разработку.

Разработанные алгоритм и программа считаются правильными, если результаты их выполнения при любых входных данных строго соответствуют требованиям внешней спецификации. Любое несоответствие результатов считается свидетельством ошибки алгоритма или программы.

Внешняя спецификация прикладной программы включает:

- назначение программы;
- описание входных данных;
- описание формы представления результатов при допустимых входных данных;
- перечисление аномалий во входных данных и реакций программы на них.

Описание входных данных включает перечень данных, которые должны быть введены с внешнего устройства в процессе выполнения программы, и их типы: числа целые или вещественные, символы или строки символов и др. (Понятие типа данных будет введено позже).

Аномалии входных данных - это различные нарушения условий допустимости входных данных. К аномалиям относят такие значения входных данных, для которых нельзя применять реализованный в программе метод решения. Так, для описанного выше метода Эвклида, входные данные не могут быть равны нулю, так что аномалией можно считать случай, когда либо первое, либо второе входное число равно нулю.

В качестве примера приведем внешнюю спецификацию программы решения квадратного уравнения, формальная постановка которой была приведена ранее.

Внешняя спецификация:

Назначение:

Решение квадратного уравнения

Входн. данные:

a, b, c - вещественные числа

Вых. данные:

```
КВАДРАТНОЕ УРАВНЕНИЕ
<a>*X^2+<b>*X+<c>=0
< результаты-решения >
```

где результаты-решения:

```
КОРНИ: X1 = <x1>  X2 = <x2>
```

где x1, y1 - вещественные числа.

Аномалии входных данных:

(1) при (a=0)

```
Недопустимо:  a = 0
```

(2) при (b^2-4ac) < 0

```
Вырождено: b^2-4ac < 0
```

Программа может быть хорошей и плохой.

В случае *плохой программы* могут иметь место:

- непонятный текст выходных данных;
- аварийное завершение ее при недопустимых входных данных. (При этом часто отсутствуют сообщения с разъяснением причины).

Программа считается *хорошей*, если выполнены следующие требования:

- текст выходных данных легко понимается, и в него включены не только результаты решения, но и входные данные решаемой задачи;
- тексты с выходными данными легко записываются и легко исправляются;
- при обнаружении аномалий формируется легко понимаемый текст с диагностикой выявленных ошибок в данных.

Для описания форм текстов выходных данных часто используют специальные средства, называемые нормальными формами Бэкуса. Нормальные формы позволяют описывать некоторую совокупность текстов с определенной структурой составляющих. Их можно использовать также и при описании форм записи алгоритмов. Мы воспользуемся некоторыми из них. В частности, *переменные элементы формы и части текстов в нормальных формах мы будем обозначать заключенными в угловые скобки словами или словосочетаниями, в которых отдельные слова соединены знаками дефиса.*

Например, текст выходных данных программы, решающей задачу расчета мощности электрической цепи, мог бы получить следующую форму:

<table>
<tr><th>Вых. данные:</th><th>Образец:</th></tr>
<tr><td>ЭЛЕКТРИЧЕСКАЯ ЦЕПЬ:
<r1>+(<r2>||<r3>) ом
НАПРЯЖЕНИЕ: U=<u> В
<результаты-решения></td><td>ЭЛЕКТРИЧЕСКАЯ ЦЕПЬ:
1.0 + (1.0 || <1.0) ом
НАПРЯЖЕНИЕ: U= 3.0 В
МОЩНОСТЬ: W= 6.0 ВТ</td></tr>
</table>

Здесь

<r1>,<r2>,<r3> - обозначения конкретных числовых значений сопротивлений,

<u> - обозначение конкретного значения напряжения,

а *<результаты-решения>* - это составляющая выходного текста, которая варьируется в зависимости от значений входных данных.

Структура составляющих текстов может задаваться раздельным описанием форм отдельных составляющих, перечисленных в разделах "где". Та же самая структура составляющих может быть описана раскрытием описаний отдельных частей в рамках общих форм.

Общая форма выходных текстов программ должна включать заголовок, входные данные и результаты решения поставленной задачи.

Ниже приводится конкретный пример двух вариантов описания выходных текстов для задачи расчета электрической цепи.

Вариант 1.

Вых. данные:

<заголовок>

<входные-данные>

<результаты-решения>

где *заголовок*:

ЭЛЕКТРИЧЕСКАЯ ЦЕПЬ:
R1 + (R2 || R3) ом

где *входные-данные*:

R1=<r1> R2=<r2> R3=<r3>
НАПРЯЖЕНИЕ: U=<u> В

где *результаты-решения*:

а) МОЩНОСТЬ: W = < w>ВТ

б) КОРОТКОЕ ЗАМЫКАНИЕ: Rобщ=0

в) СОПРОТИВЛЕНИЕ < 0

Вариант 2.

Вых. данные:

ЭЛЕКТРИЧЕСКАЯ ЦЕПЬ:
R1 + (R2 || R3) ом
R1=<r1> R2=<r2> R3=<r3>
НАПРЯЖЕНИЕ: U=<u> В

а) МОЩНОСТЬ: W = <w> ВТ

б) КОРОТКОЕ ЗАМЫКАНИЕ: Rобщ=0

в) СОПРОТИВЛЕНИЕ < 0

Для альтернативных вариантов различных составляющих в нормальных формах используют фигурные скобки, в которых перечисляются альтернативы.

В приведенных же выше описаниях вместо фигурных скобок альтернативы помечены буквами а), б), в). Альтернативами здесь являются результаты решения, вид которых зависит от конкретного набора входных данных.

Для описания составляющих, которые могут войти или не войти в текст выходных данных, обычно применяют квадратные скобки.

Пример.

результаты-решения:

[<промежуточные-результаты>]

<требуемые-результаты>

образец:

[Дискриминант: D = <d>]

Корни: X1=<x1>

1.4. Разработка алгоритма

Существует несколько определений алгоритма. Мы приведем неформальное определение алгоритма, которое полезно с позиций содержания данного этапа разработки программы.

Алгоритм – упорядоченная совокупность действий, приводящая за конечное число шагов к результатам, соответствующим исходным данным.

Любой алгоритм предполагает исполнителя. Исполнителем алгоритма может быть человек или автоматическое устройство. Очевидно, что набор действий зависит от возможностей исполнителя, и что любой исполнитель умеет выполнять ограниченный набор действий. А для случая, когда разработчик алгоритма не является его исполнителем, необходимо каким-либо образом представить его в такой форме, чтобы он был понятен и доступен исполнителю.

Компьютер – автоматическое устройство, которое может исполнить алгоритм только в том случае, если алгоритм представлен в виде упорядоченной совокупности элементарных действий, которые называют командами. Полный набор команд компьютера называется системой команд.

Компьютеры разных фирм-производителей и даже разных моделей одной и той же фирмы, как правило, имеют несовпадающие системы команд. Отсюда становится очевидным, что представлять алгоритм сразу на уровне системы команд конкретного компьютера в подавляющем большинстве случаев нецелесообразно, поскольку перенести алгоритм на компьютер с отличающейся системой команд без переделки невозможно. Кроме того, разработчик алгоритма в этом случае обязан хорошо знать систему команд и особенности операционной системы того компьютера, на котором будет исполняться разработанный алгоритм. Эти обстоятельства существенно ограничивают круг лиц, которые могут в разумные сроки и качественно решить задачу сразу на языке компьютера (в среде программистов говорят – в кодах компьютера).

Кроме того, язык системы команд – искусственный, и программисты (в первую очередь, непрофессиональные) все равно вначале должны представить алгоритм на естественном и понятном для них языке. Однако живые естественные языки (русский, английский и другие), кроме несомненных достоинств, имеют и один существенный недостаток применительно к записи алгоритмов: запись на таком языке может быть неоднозначной, в связи с чем разные исполнители могут интерпретировать ее неодинаково. А это недопустимо. Поэтому для записи алгоритмов используются искусственные языки. Именно их стали называть алгоритмическими языками. Языки, которые в качестве исполнителей предполагают компьютеры, называются языками программирования. Средства этих языков должны быть такими, чтобы на них можно было предста-

вить алгоритм, отвечающий следующим основным свойствам: однозначность, результативность, массовость, правильность.

Однозначность алгоритма заключается в однозначности правил его исполнения любым исполнителем. Результативность алгоритма означает, что его исполнение завершается определенными результатами. Результативность алгоритма – наиболее важное свойство алгоритма, поскольку деятельность, не приводящая ни к каким результатам бесполезна. Массовость алгоритма заключается в возможности его применения для решения целого класса задач с различными конкретными исходными данными. Правильность алгоритма заключается в получении результатов, однозначно соответствующих конкретным исходным данным.

Очевидно, что для лиц, начинающих изучение программирования для компьютеров и не знающих, или знающих недостаточно хорошо языки программирования, разработку алгоритмов значительно проще проводить на языке, ориентированном на возможности человека. За короткую историю существования компьютеров было предложено несколько видов таких языков. На сегодня наиболее популярны два вида языков для записи алгоритмов, используемые на этапе разработки алгоритма – язык блок-схем и псевдокод.

На языке блок-схем алгоритм изображается графически. При этом используется некоторый ограниченный набор операций, каждая из которых имеет свое собственное обозначение (образ). Псевдокод – язык, на котором алгоритм записывается в виде текста, предложения которого описывают определенные действия. Языка псевдокода также содержат ограниченный набор действий и очень несложный синтаксис. И хотя начинающим язык блок-схем представляется более предпочтительным, чем псевдокод, последний значительно удобней.

К основным достоинствам псевдокода относится следующее:

- алгоритм на нем записывается с использованием лексики на родном для разработчика языке, в связи с чем его легко понять и однозначно выполнить;
- запись алгоритма не зависит не только от модели компьютера, но и от языка программирования, вследствие чего алгоритм при необходимости можно перевести без изменений (перекодировать) на заданный язык программирования.

Язык псевдокода и его использование будут подробно рассматриваться в дальнейшей части пособия. Здесь же отметим, что данный этап является наиболее трудоемким среди прочих. Процесс разработки алгоритма должен быть организован так, чтобы обеспечить его правильность. А это достигается в результате применения определенной технологии.

Под технологией понимают определенную совокупность средств, методов и действий, направленных на достижение поставленной цели. А когда речь идет о разработке (создании) программ, термин технология используется в сочетании со словом “информационная”. Информационная технология – процесс, использующий совокупность средств и методов сбора, обработки и передачи первичной информации для получения информации нового качества (информационного продукта).

В дальнейшей части пособия мы подробно познакомимся с технологией разработки, которая получила название метода пошаговой детализации. Результатом данного этапа будет алгоритм решения задачи, записанный на языке псевдокода, а также набор тестов, которые были разработаны и использовались в процессе разработки алгоритма. Определение теста и принципы разработки тестов будут рассматриваться позже.

1.5. Кодирование алгоритма на языке программирования

Только на данном этапе от разработчика требуется знание языка программирования, на котором предполагается записать алгоритм, получив в результате этого программу, которая может быть выполнена на компьютере. Когда алгоритм уже разработан и записан, и у разработчика имеется уверенность в его правильности, перевод на язык программирования должен быть организован так, чтобы не внести в него изменений. С этой целью рекомендуется использовать формализованные правила перевода (кодирования). Эти правила будут рассматриваться в дальнейшем при описании конкретного языка программирования (в нашем случае это язык Паскаль). И еще. Использование псевдокода определяет, какие средства языка надо освоить в первую очередь при изучении нового языка программирования.

Результатом данного этапа является программа на языке программирования, а трудоемкость его невелика, так что между моментом окончания разработки алгоритма и моментом начала этапа испытания программы проходит минимальное время.

1.6. Испытания программы на тестах

К этому моменту разработки программы, наконец, требуется компьютер. Цель данного этапа – убедиться, что программа полностью отвечает требованиям постановки задачи и внешней спецификации программы. Для этого текст программы «набирается» на клавиатуре, сохраняется во внешней памяти (на дискете или на жестком диске) и компилируется на язык машины. Возможны два вида ошибок в программе: синтаксические и семантические.

Синтаксические ошибки связаны с нарушением правил записи на языке программирования. Эти ошибки обнаруживает компилятор, выводя

диагностические сообщения. Так что поиск и устранение ошибок данного вида – задача несложная.

После того, как синтаксические ошибки устранены, начинаются испытания на тестах. Тест – контрольный пример, для которого известны как входные данные, так и правильный (эталонный) результат для них. Если все тесты проходят успешно (результаты выполнения совпадают с эталонными), принимается решение, что разработка программы завершена.

Если же при выполнении того или иного теста получены результаты, не совпадающие с эталонными, обнаружена ошибка и необходимо найти и устранить ее причину. Процесс поиска причины ошибки и ее устранение называется отладкой. Успешность и продолжительность процедуры отладки программы зависит от того, насколько удачно разработан набор тестов.

Для сокращения времени отладки при разработке тестов рекомендуется использовать специальные подходы, результатом которых будет система тестов. Под системой тестов понимается минимально возможное количество тестов, достаточное для выявления всех возможных семантических ошибок в программе. Иначе, если при испытаниях программы на системе тестов не выявлено ни одной ошибки, можно утверждать, что ошибок в программе нет, и разработку программы можно считать завершенной.

Этап испытания программы на тестах часто называют просто этапом тестирования программы.

После успешного завершения этого этапа программа документируется, т.е. снабжается документацией. Перечень и содержание документов будут рассматриваться позже.

Вопросы для самоконтроля

1. Какова цель этапа постановки задачи?
2. В какой форме может быть представлено описание метода решения задачи?
3. Какой смысл отражает слово «внешняя» в названии этапа «Внешняя спецификация программы»?
4. Каково назначение внешней спецификации программы?
5. Что такое «тип данных»? Какую информацию о данных он несет?
6. Поясните, что такое «аномалия входных данных», и что содержит описание аномалии.
7. В чем основное отличие терминов «алгоритмический язык» и «язык программирования»?
8. Какие виды языков применяются для записи алгоритмов?
9. Что такое «псевдокод»?
10. В чем отличие содержания тестирования и отладки программы?

2. ЯЗЫК ДЛЯ ЗАПИСИ АЛГОРИТМОВ

Содержательно алгоритм – это упорядоченная совокупность действий (операций) над некоторыми объектами. Для алгоритмов, выполняемых на ЭВМ, объектами действий служат данные, размещаемые в ее внутренней памяти.

Рассмотрим простой пример:

Вне компьютера	Алгоритм и внутренние переменные
Входные данные: ***a*** - вещественное число Выходные данные: SIN(<***a***>)=<***b***> где ***b*** – вещественное число, и ***b*** = sin ***a***	**Алг**оритм “Вычисление синуса” Внутренние переменные: ***x*** [ ] вещ ***y*** [ ] вещ **нач**ало ввод (***x***) ***y*** := sin (***x***) вывод (“SIN(“,***x*** , “) =”, ***y***) **кон**ец

Здесь:

Объекты действий:

- вещественное число ***a*** , вводимое с устройства ввода (например, с клавиатуры);
- внутренние переменные ***x*** и ***y*** , используемые для хранения аргумента и значения синуса во время исполнения программы;
- выходной текст, включающий эти величины (точнее, их представление в виде последовательности букв и цифр).

Совокупность действий над этими объектами:

- операция ввода значения аргумента во внутреннюю переменную ***x;***
- операция присваивания переменной ***y*** значения синуса от аргумента ***x;***
- операция вывода текста со значениями ***x*** и ***y*** в виде, описанном во внешней спецификации, элементы которой приведены в первом столбце таблицы.

Технология так называемого “надежного программирования”, которая поддерживается практически всеми современными языками программирования, требует, чтобы все ***переменные*** в алгоритме (и программе) были объявлены (иначе декларированы). Декларация переменной включает указание имени, типа, размерности (для массивов), и при необходимости – начальных значений (если они известны).

В первую очередь введем определение программной переменной:

Программная переменная обозначается символически (именем), размещается во внутренней памяти ЭВМ и всегда имеет какое-либо значение, иногда неизвестное (!)

Отсюда следует, что значение переменной необходимо всегда инициализировать, т.е. присваивать ей значение перед первым использованием в выражениях, где она используются как операнд.

Тип переменной определяет множество значений, которые она может принимать, и множество операций, которые разрешены над этими значениями.

В качестве примера приведем ***целые числа***, которые принимают целочисленные значения в интервале ***-МаксЦел*** до ***+МаксЦел,*** и для которых определены следующие арифметические операции: сложение, вычитание, умножение, целочисленное деление, и операции отношения: больше, меньше, больше или равно, меньше или равно, не равно и равно (эквивалентно).

В отличие от привычных действий в арифметике, результаты арифметических действий над целыми числами в ЭВМ имеют особенности. Так, результат операции сложения двух чисел, вычитания и умножения не могут выходить за пределы (-МаксЦел … +МаксЦел), а результат деления целых – всегда является целым числом. При этом в качестве результата предъявляется целая часть от деления, а остаток от деления отбрасывается вне зависимости от его величины.

Отметим, что значение МаксЦел (максимальное целое) зависит от архитектуры ЭВМ, и в разных ЭВМ могут отличаться. Для компьютеров, совместимых с IBM PC, эта величина зависит от системы программирования, и, например, для языка Турбо Паскаль это число равно 32767.

Язык для записи алгоритмов, который будет использован в данном пособии, мы будем называть Русским Алгоритмическим Языком (РАЯ – версия псевдокода, использующая русскую лексику). Этот язык в качестве исполнителя предполагает исключительно человека, и поэтому правила записи алгоритма на нем должны быть удобны и понятны именно человеку. Самое важное отличие РАЯ от русского языка – ограниченный набор фраз (предложений) языка и строгие правила их построения для обеспечения строгой однозначности смысла текста алгоритма, чтобы результат его выполнения не зависел от того, кто будет исполнять алгоритм.

Приведем краткую характеристику средств языка РАЯ.

2.1. Базовые типы величин

Скалярные величины:

- целые (цел),
- вещественные (вещ) – применяются для обозначения величин, кото-

рые могут иметь целую и дробную части и записываются с использованием десятичной точки между ними

- символьные (символ, строка символов)
- логические (лог)

Массивы. Массив – регулярный тип данных, который содержит несколько величин (элементов) одного и того же скалярного типа, которому присвоено единственное имя, а для указания одного элемента используется целочисленный номер (номера) его, называемый индексом. В этом случае говорят об индексированной переменной (переменной с индексами). Количество индексов, необходимых для указания на один элемент массива, определяет размерность массива: одномерный, двухмерный и т.п.

В любом языке программирования имеется более обширный набор типов данных. Ряд языков позволяет также и программисту вводить свои типы данных. Такие языки называют языками с абстрактными типами данных. Язык Паскаль, который мы будем использовать в данном пособии, является именно таким языком.

В пособии абстрактные типы будут рассматриваться только при необходимости, поскольку на начальном этапе изучения программирования в этом нет острой необходимости.

2.2. Объявление величин в алгоритме

Константы. Константы любого типа в объявлении не нуждаются, они выражают значение своим написанием (видом).

В РАЯ числовые константы можно записывать так, как это делается в математике, а вот для записи символьных констант используются апострофы (верхние одинарные кавычки).

Тип ***символ*** применяется для записи одиночных символов используемого в алгоритме алфавита – ‘a’, ‘D’, ‘2’, ‘&’ и т.п. Величины данного типа относятся к так называемым перечислимым типам данных (список возможных значений задается перечислением).

Тип ***строка*** применяется для записи цепочек символов как единого значения. Он используется для записи слов, предложений, строк знаков, изображающих число и т.п.

Логический тип определяет всего две константы – истина (TRUE) и ложь (FALSE).

Переменные. В программе должны быть объявлены. Цель этого – указать их тип. Примеры объявления переменных:

а) ***Скалярные переменные*:**

a, t, t12 : цел {имена трех скалярных переменных целого типа}
w, q : вещ {имена двух скалярных переменных вещественного типа}
error : лог {скалярная переменная логического типа}

б) ***Массивы***:

x[1 . . 12]: вещ {одномерный массив, в котором можно разместить от 1 до 12-ти элементов - вещественных чисел; запись в квадратных скобках называется ***граничной парой***; она задает диапазон возможных значений для индекса; одномерный массив используется как аналог математического понятия вектора}

s[1 . . 4, 1 . . 6] : симв {двухмерный массив – таблица из четырех строк и шести столбцов, элементами которой являются отдельные символы; для указания на один элемент требуется указать два индекса; запись S[2,4] указывает на символ, находящийся во второй строке и четвертом столбце данного массива}.

в) ***Структуры***:

Структура – это совокупность нескольких элементов, которой сопоставляется одно имя. Элементы структуры называются полями и могут быть неодинакового типа. Структура считается абстрактным типом данных; она объявляется программистом, а ее состав задается в секции деклараций (объявлений) типов данных в программе. Для указания на элемент структуры используется специальная запись вида:

```
<переменная типа структуры>.<имя поля>
```

Такой вид записи называется ***квалифицирующим***, а точка служит разделителем между именами структуры и ее элемента (его называют также подструктурой).

Например, если тип данных по имени ”TStudent” объявляется как структура

```
TStudent : структура
    имя : строка
    возраст :  цел
    адрес : строка
```

то для указания на поле *имя* вначале необходимо объявить переменную этого типа:

```
Студент : TStudent
```

и тогда указание на поле, содержащее имя студента, примет вид:

```
Студент.имя
```

2.3. Структура компьютера с позиций программы

Структура компьютера с позиций программы представлена на рис. 2.1. После загрузки программы в оперативную память, текст программы на языке компьютера, ее константы и переменные размещаются в оперативной памяти. А реализация алгоритма обеспечивается всего тремя видами базовых операций. Это ввод исходных данных (O1), формирование

новых данных в соответствии с применяемым методом (О2) и вывод результатов (О3).

Направление передачи данных при выполнении этих операций показано стрелками. Ввод данных производится с одного из внешних устройств в оперативную память. Результатом операции О2 является новое значение, которое присваивается какой-либо переменной программы, в связи с чем ее называют операцией присваивания или просто присваиванием. При ее выполнении исходные данные считываются из оперативной памяти, результат записывается вновь в оперативную память, а формирование (вычисление) результата производится в центральном процессоре. Вывод результатов работы программы в целом производится из оперативной памяти на одно из внешних устройств.

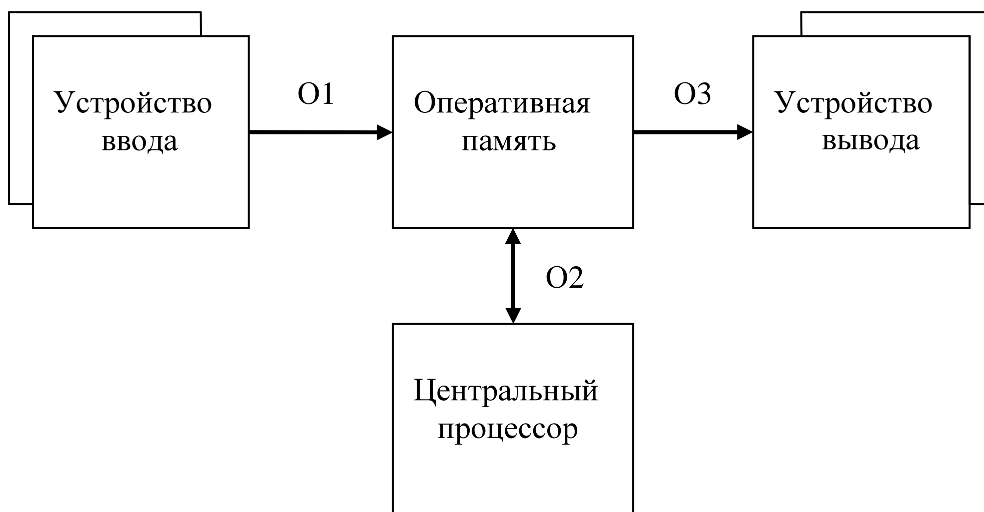


Рис. 2.1. Структура ЭВМ с позиций программы.

В связи с этим для записи алгоритма, вне зависимости от формы записи алгоритма, в его составе должны быть заданы правила представления этих операций, как основных. Рассмотрим их.

2.4. Базовые операции

Ввод. Ввод – это процесс копирования данных (констант) с внешнего устройства во внутренние переменные программы. При этом никаких новых значений не формируется, но производится преобразование формы их представления. Например, код клавиши, нажатой на клавиатуре, преобразуется в двоичное число – номер символа в таблице символов.

Запись операции ввода в алгоритме имеет следующий вид:

```
ввод(список ввода)
```

где ***список ввода*** – перечень имен переменных, записанных через запятую, значения которых должны быть заменены теми, которые будут поступать с внешнего устройства.

Отметим, что в результате выполнения данной операции переменные из списка ввода ***всегда получают новые значения*** (с потерей старых значений), и эти значения присваиваются в том порядке, в котором переменные указаны в списке.

Пример:
Пусть с входного устройства поступает последовательность чисел 12, 3.5, -2.
И пусть выполняется операция "ввод (**a**, **s[7]**, ***x***)"
Тогда после ее выполнения переменные примут следующие значения:

a = 12; **s[7]** = 3.5 ; ***x*** = -2

Присваивание. Это операция формирования нового значения для внутренней переменной. Запись ее имеет следующий вид:

```
<имя переменной> := <выражение>
```

Операция заключается в вычислении выражения и присваивании его значения переменной, имя которой записано слева от записи "**: =**", называемой ***символом присваивания***. Переменная, находящаяся слева от знака присваивания, должна быть скалярной, т.е. такой, которая одновременно может хранить одно единственное значение. Выражение строится так же, как это принято в математике: оно может содержать константы, переменные и функции, а для задания порядка вычислений, не совпадающих со старшинством (приоритетностью) операций, можно использовать круглые скобки.

В общем случае имя некоторой переменной может присутствовать одновременно как слева, так и справа от знака присваивания. При вычислении выражения всегда используется то значение переменной, которое она имела до начала операции. Присваивание нового значения переменной (справа от знака "**: =**") производится строго после завершения вычислений.

Примеры записи операции:

y := sin(***x***) {присвоить значение функции синус от аргумента ***x***}
d[2] := exp(-s/(t+2)) {присвоить новое второму элементу массива ***d***}
k := k+1 {увеличить значение переменной ***k*** на единицу}

Отметим одно из условий, которое должно быть обязательно выполнено: типы выражения и переменной слева от знака присваивания должны совпадать.

Вычисление выражений имеет ряд особенностей, которые необходимо учитывать. Одной из них является неопределенность некоторых алгебраических операций и элементарных функций при некоторых значениях аргументов.

Примеры.

1. Некорректность деления целых (упоминавшаяся выше).
2. Неопределенность деления на ноль – в этом случае происходит аварийное завершение программы. Аналогичная ситуация возникает и при делении на достаточно малое (и не равное нулю) делимое.
3. Неопределенность значения функции логарифм от вещественного числа, близкого к нулю.

Причиной исключительных ситуаций может служить возникновение промежуточных значений, превышающих диапазоны допустимых значений:

1. Результат выражения 10^{100} превышает МаксВещ и не может быть представлен в памяти компьютера.
2. Результат выражения e^x при $x > \ln$ (МаксВещ) превышает МаксВещ и не может быть представлен в памяти компьютера.
3. Результат выражения $\frac{a*x}{b}$ при $|a*x| > МаксВещ$ превышает МаксВещ и не может быть представлен в памяти компьютера.

Вывод. Вывод – это процесс формирования на внешнем устройстве цепочки символов. При выводе на экран или на принтер результатом операции будут сформированные строки текста. Правила размещения выводимой на печать (экран) информации определяются во внешней спецификации, а в алгоритме могут задаваться в одной из двух форм: списком вывода или образцом вывода.

Запись операции ***вывода по списку*** имеет вид

вывод (<список вывода>)

где элементом списка вывода может быть

- символьная константа,
- имя переменной или выражение, значение которых в виде символьной строки должно быть отображено на устройстве вывода.

При использовании списка вывода можно задать только порядок вывода элементов, но не правила их размещения на внешнем устройстве.

Запись операции ***вывода по образцу*** выглядит следующим образом:

вывод (<образец вывода>)

Пример записи операции ввода по образцу:

вывод

SIN(<***a***>) = <***b***>

Здесь значения переменных ***a*** и ***b***, подлежащие выводу, будут помещены в позиции образца, заключенные в угловые скобки, а прямоугольная рамка условно отделяет образец от остального текста алгоритма (служит границами образца).

Пример вывода той же информации по списку имеет вид:

вывод(‘SIN(‘, ***a*** ,’) =’ ,***b***)

При выводе последовательности элементов можно использовать обобщенную компактную форму записи.

Примеры:

1. Вывести в соответствии с заданным образцом все элементы одномерного массива ***Z***, содержащего ***n*** элементов по 5 чисел в строке:

   Вывод

   <z[1]> <z[2]> . . . <z[5]>
   <z[6]> <z[10]>
   . . . .<z[n]>

2. Вывести все элементы массива А, не оговаривая формы их размещения:

   вывод (a[k], a[k+1], . . . a[k+n])

2.5. Управляющие структуры

Для организации разнообразной последовательности действий в алгоритме используются три вида управляющих структур, обеспечивающих требуемую последовательность. Это – следование, альтернативный выбор и циклическое повторение. При их использовании рекомендуется применять определенные правила записи, которые позволят наглядно представить структуру алгоритма. Такая запись получила название ***структурной***. Далее, при рассмотрении управляющих структур будут излагаться и правила структурной записи.

Рассмотрим их.

Следование. Данное правило определяет следующее: два действия (в частности, например, две операции) связаны отношением следования в том случае, если второе действие выполняется после окончания первого, начинает выполняться в момент завершения первого и выполняется вне зависимости от результата первого действия.

При структурной записи следования двух действий в алгоритме их тексты должны располагаться друг под другом, и начинаться они долж-

ны с одинаковым отступом от края листа. Если для записи некоторого действия требуется более одной строки, все строки кроме первой должны записываться с отступом вправо по отношению к первой строке. Это позволит сделать более наглядной структуру алгоритма.

Ниже приведены графическое и текстовое изображение двух действий, связанных отношением следования.

Действие 1

Действие 2

```
Действие-1-для-записи-
  которого-требуется-
  более-одной-строки
Действие-2
```

Два действия, связанные отношением следования, при изображении алгоритма можно рассматривать как одно, имеющее одну точку входа и одну точку выхода.

Конкретные примеры использования как данной, так и других структур будут приведены ниже.

Альтернативный выбор. Данная конструкция применяется в случае, когда в процессе выполнения алгоритма дальнейшие действия связываются с выполнением некоторого условия. Различают следующие варианты данной конструкции: одноальтернативная (а), двухальтернативная (б) и многоальтернативная (в).

При записи данной конструкции используются специально зарезервированные (ключевые) слова – **если, то, инес** (от слов иначе-если), **иначе** и **все**. Для того чтобы в тексте алгоритма эти слова выделялись, они будут изображаться подчеркнутыми. Ниже приведены типовые формы их записи.

(а)

```
если <условие> то
     действие-1
все
```

Вариант (а) изображает конструкцию, в которой действие-1 выполняется, если истинно условие. В противном случае ничего не выполняется, но в любом случае после этого выполняется действие, следующее за данной конструкцией.

(б)

```
если <условие> то
    действие-1
иначе
    действие-2
все
```

Вариант (б) используется, когда действие-1 выполняется при истинности условия, а во всех остальных случаях выполняется действие-2.

(в)

```
если <условие1> то
    действие-1
инес <условие2> то
    действие-2
иначе
    действие-3
все
```

Вариант (в) представляет наиболее общий случай. Он изображает ситуацию, когда выполнение любого действия, кроме последнего, связано с истинностью (выполнением) соответствующего условия. Из этой конструкции можно построить структуру с произвольным числом альтернатив. Для этого достаточно повторить элемент **инес** требуемое число раз: 0, 1, 2, …, а элемент **иначе**, при необходимости, можно опустить.

Любой вариант конструкции "альтернативный выбор" имеет одну точку входа (слово **если**) и одну точку выхода (слово **все**). И в этом смысле она может быть связана отношением следования с любым другим действием алгоритма. Точно так же любая из альтернатив может рассматриваться не как единое действие, а как несколько действий, связанных отношением следования.

Циклическое повторение. Данная конструкция предназначена для задания многократного повторения одного и того же действия (или группы действий). В программировании эту конструкцию называют просто конструкцией ***цикл***, а повторяющиеся действия – ***телом цикла***.

Можно выделить три основных варианта данной конструкции: "цикл с предусловием", "цикл с постусловием" и "цикл со счетчиком". Рассмотрим их.

(а) **Цикл с предусловием** записывается следующим образом:

```
цикл-пока <условие>
    тело-цикла
кцикл
```

Принцип выполнения конструкции:

Вначале проверяется условие, записанное после ключевого слова **цикл-пока**, и если оно истинно, выполняется тело цикла. После достижения ключевого слова **кцикл** (аббревиатура от слов "конец цикла") происходит возврат на повторную проверку условия, записанного в начале конструкции. В случае нарушения истинности условия выполнение конструкции завершается и происходит переход к действию алгоритма, следующему за циклом.

Из изложенного следует, что тело цикла может быть не выполнено ни разу, если в первый момент условие не выполняется, и может повторяться неопределенное число раз. Если условие выполняется всегда, выхода из данной конструкции может не произойти никогда. Такая ситуа-

ция называется ***зацикливанием***, и разработчик алгоритма должен сам позаботиться о том, чтобы избежать ее. Очевидно, что для ограничения числа повторений в теле цикла должны происходить изменения переменных, которые используются в условии.

Пример конструкции "цикл с предусловием" (ее иногда называют "цикл-пока") во фрагменте алгоритма:

```
S := 0;
k := 1;
цикл-пока S<35
   S := S + k*k
   k := k + 1
кцикл
```

Здесь тело цикла составляют две операции присваивания, связанные отношением следования. Их выполнение повторяется до тех пор, пока истинно условие "S<35", а поскольку значение переменной S монотонно возрастает, то в некоторый момент условие перестанет выполняться и, соответственно, прекратится выполнение всей конструкции.

(б) **Цикл с постусловием** записывается следующим образом:

```
цикл
   тело-цикла
кцикл-до <условие>
```

Принцип выполнения:

Вначале выполняется тело цикла, затем проверяется условие, записанное после ключевого слова **кцикл-до**. Если условие не выполняется (ложно), происходит возврат на начало, и тело цикла выполняется вновь. Этот процесс повторяется до тех пор, пока при очередной проверке условие не примет истинного значения, и в этом случае выполнение конструкции завершается и происходит переход к действию алгоритма, следующему за циклом. Здесь также возможно зацикливание.

Описываемая конструкция, как и предыдущая, является конструкцией с неопределенным числом повторений тела цикла. Основное отличие этой конструкции от предыдущей заключается в том, что в ней тело цикла выполняется хотя бы один раз.

Пример конструкции "цикл с постусловием" (ее иногда называют "цикл-до") во фрагменте алгоритма:

```
S := 0;
k := 1;
цикл
   S := S + k*k
   k := k + 1
кцикл-до S<35
```

(в) **Цикл со счетчиком** записывается следующим образом:

```
цикл-для <сч> от <нач> до <кон>
    тело-цикла
кцикл
```

Здесь

сч - имя скалярной переменной целого типа, выполняющей функции счетчика числа повторений тела цикла (ее называют счетчиком или параметром цикла)

нач - целочисленное выражение, которое задает начальное значение счетчика цикла

кон - целочисленное выражение, которое задает конечное значение счетчика цикла.

Полагается, что с каждым повторением тела цикла значение счетчика изменяется (увеличивается или уменьшается) на единицу. Отсюда, данная конструкция такова, что число повторений тела цикла в ней всегда конечно и его можно точно вычислить. Поэтому ее называют конструкцией цикла с определенным числом повторений.

Принцип выполнения: Изложим его для случая, когда с каждым повторением тела цикла значение счетчика увеличивается на 1. Вначале счетчику цикла присваивается начальное значение. Затем производится проверка, не превысило ли текущее значение счетчика конечной величины. Если это условие истинно, выполняется тело цикла. После этого значение счетчика увеличивается на 1, и производится возврат в начало цикла, где вновь проверяется, как соотносятся между собой текущее значение счетчика цикла и конечного значения. Процесс повторений прекращается и происходит переход к действию, записанному после рассматриваемой конструкции цикла, когда текущее значение счетчика превысит конечное значение.

Анализ описанной логики исполнения показывает, что, во-первых, тело цикла может быть не выполнено ни разу, и, во-вторых, что здесь невозможно зацикливание.

Имеется одно существенное замечание: счетчик цикла можно использовать, но нельзя изменять в теле цикла.

Пример конструкции “цикл со счетчиком” во фрагменте алгоритма:

```
S := 0;
цикл-для k от 1 до
n
   S := S + k*k
```

Здесь k – счетчик цикла, и он принимает последовательно значения: 1, 2, . . ., n. За счет этого, после завершения цикла, переменная S примет зна-

чение суммы квадратов указанных выше значений счетчика цикла: $S = \sum_{k=1}^{n} k^2$.

2.6. Структура алгоритма

Запись алгоритма представляется композицией описанных выше операций и управляющих структур и имеет строго определенную структуру. Ниже приведены два вида структур алгоритма – структура независимого алгоритма, который на языке программирования представляется основной программой, и структура вспомогательного алгоритма, представляемого на языке программирования подпрограммой.

Структура главного (независимого) алгоритма имеет следующий вид:

Алгоритм “<название-алгоритма>”
Внутренние **перем**енные:
    <список-переменных>
Начало
    <последовательность-исполнимых-действий>
Конец

Здесь

<список-переменных> - объявление переменных алгоритма, т.е. перечень отдельных переменных с указанием их типов, а для массивов – также и с указанием граничных пар для каждого индекса.

<последовательность-исполнимых-действий> - описание метода решения в виде упорядоченной последовательности отдельных вычислительных операций, построенной с использованием управляющих структур.

Использованные ***ключевые слова*** (Алгоритм, Внутренние переменные, Начало и Конец) записаны полностью, но отдельные части в них выделены жирным шрифтом и подчеркнуты. Это сделано для того, чтобы в последующих текстах использовать только выделенные части.

Алгоритм называется вспомогательным в связи с тем, что он не решает всей задачи и исполняется по инициативе другого алгоритма. Эти два алгоритма – вызываемый и вызывающий, должны обмениваться между собой информацией. Этот обмен производится через переменные. В тексте вспомогательного алгоритма для данных, которые поступают из вызывающего алгоритма, используется раздел входных переменных. А раздел выходных данных содержит данные, являющиеся результатом работы данного алгоритма и возвращаемые в вызывающий алгоритм.

Структура вспомогательного алгоритма:

```
Алгоритм “<название-алгоритма>”
Входные данные:
      <список-переменных>
Выходные данные:
      <список-переменных>
Внутренние переменные:
      <список-переменных>
Начало
      <последовательность-исполнимых-действий>
Конец
```

Пример записи вспомогательного алгоритма:

```
Алг “НОД по Евклиду”
Входн дан:
      a, b : цел  {входные числа, НОД которых требуется найти}
Выходн дан :
      c : цел      {выходная переменная с НОД(a,b) }
      exist : лог {признак существования НОД}
Внутр перем :
a1, b1 : цел {рабочие переменные}
Нач
      если (a=0 или b=0) то
         exist := false  {НОД не существует}
      иначе
         a1:=abs(a)
         b1:=abs(b)
         цикл-пока (a1<>b1)
            если (a1>b1) то
               a1:=a1 – b1
            иначе
               b1:=b1 – a1
            все
         кцикл
         c:=a1     {искомый НОД}
         exist := false
      все
кон
```

И если у вас имеется алгоритм, то из языка программирования, на котором этот алгоритм вы собираетесь записать, вам в первую очередь надо изучить средства, необходимые для кодирования объявлений переменных, вычислительных операций, управляющих структур и структуры программ (подпрограмм) в целом. Процесс записи алгоритма на языке программирования – задача следующего этапа – этапа кодирования.

Вопросы для самоконтроля

1. В чем существенное отличие переменной в математике от программной переменной?
2. Перечислите базовые (основные) типы величин в алгоритмах.
3. Что такое массив и чем он характеризуется в декларации (объявлении)?
4. Как объявить (декларировать) переменную, массив, структуру?
5. Что такое квалифицирующая переменная и как она записывается?
6. Охарактеризуйте назначения и правила записи базовых операций?
7. Что понимается под термином «образец вывода»?
8. Когда желательно использовать операцию вывод по образцу?
9. Что такое управляющая структура? Каковы функции управляющих структур?
10. Когда используются структура «альтернативный выбор»?
11. Что такое тело цикла?
12. Что такое параметр цикла?
13. В каких случаях целесообразнее использовать управляющую структуру «цикл-со-счетчиком»?
14. В чем отличие в структуре основного и вспомогательных алгоритмов?

3. ВВЕДЕНИЕ В ЯЗЫК ПРОГРАММИРОВАНИЯ ПАСКАЛЬ

Литературы о языке программирования Паскаль достаточно много, поэтому в данном разделе приводится только такие сведения, которых необходимо и достаточно, для кодирования алгоритмов, построенных и записанных на псевдокоде. Материал данного раздела может использоваться в качестве примера того, с чего надо начинать знакомство с новым языком программирования для лиц, которые имеют некоторый опыт разработки программ на других языках.

3.1. Краткая характеристика языка

3.1.1. Алфавит, лексемы, разделители

Алфавит включает:

- 26 букв латинского алфавита: A, B, C, .. , Z (большие и малые буквы не различаются и могут использоваться по усмотрению программиста);
- 10 арабских цифр;
- набор специальных символов, в число которых входят символ пробела, скобки круглые, квадратные, фигурные, знаки >, <, +, =, -, точка, точка с запятой, запятая и другие.

Отдельные символы языка будут использованы при рассмотрении тех конструкций языка, в которых они применяются.

Лексемами называют минимальные значимые единицы текста программы. В Паскале можно выделить следующие категории лексем:

- специальные символы;
- зарезервированные (ключевые) слова;
- идентификаторы: стандартные (предопределенные) и введенные программистом;
- числа: десятичные и шестнадцатеричные;
- строки;
- комментарии.

Специальные символы – это набор одиночных символов для записи выражений (например, точка, запятая, скобка, звездочка и др.) и составные символы { := | . . | >= | <> | <= | (* | *) }.

Ключевые слова имеют строго определенное назначение, которое не может быть изменено, поэтому использование их для других целей не допускается. Отметим, что эти слова в окне редактирования интегрированной среды Турбо Паскаля выделяются цветом. Примеры ключевых слов:

program, begin, array, and, const, while, end.

Идентификатором считается последовательность букв, цифр и знаков подчеркивания, которая начинается с буквы или символа подчерки-

вания и не содержит пробелов. Идентификаторы применяются в качестве имен констант, типов, переменных, процедур, функций, полей в записях, программ. Идентификатор может иметь произвольную длину, однако значимыми являются только первые 63 символа для внутренних и первые восемь символов для внешних имен.

Стандартными идентификаторами являются имена всех встроенных в язык процедур и функций (***read, write, sin*** и другие), типов (***integer, real, char*** и другие) и директив (***private, public, forward*** и другие). Переопределение стандартных идентификаторов, в принципе, допускается, но считается плохим стилем и не рекомендуется.

Числа. В Паскале используются целые десятичные, целые шестнадцатеричные и вещественные десятичные числа.

При записи ***целых*** чисел можно использовать только арабские цифры. Перед старшей значащей цифрой шестнадцатеричного числа помещают символ $, который и выступает признаком этого вида чисел.

Примеры записи: -12 +456 47
$0 $3E $A10

Допустимый диапазон для целых чисел зависит от модели компьютера и всегда может быть установлен программистом либо из справочника, либо обратившись из программы к встроенной функции ***MaxInt.***

Вещественные числа записываются или в виде обычной десятичной дроби, в которой целая часть отделяется от дробной десятичной точкой, или в показательной форме с основанием 10. В этом случае вместо основания 10 ставится буква Е (прописная или строчная), непосредственно за которой указывается показатель степени в виде целого числа со знаком + (плюс), – (минус) или без знака.

Примеры записи: 6.3 -146.789 7.9e7 -0.2745e-10

Строка символов – последовательность символов из расширенного набора символов кода ASCII, заключенная в одиночные кавычки (апострофы).

Примеры: ’Turbo Pascal 7.0’ ’Н.Вирт – автор языка Паскаль’

Комментарий – это фрагмент текста программы, заключенный в фигурные скобки { и } или символы (* и *). Комментарии игнорируются компилятором и не влияют на работу программы. Они записываются с использованием набора символов, имеющегося в распоряжении программиста на компьютере, и, следовательно, могут быть записаны на естественном языке, например, на русском.

Разделители. В качестве разделителей лексем друг от друга применяются символы пробела, табуляции, признак перевода на новую строку, а также ряд управляющих символов таблицы ASCII. Между любыми двумя лексемами допускается произвольное количество символов-разделителей.

3.1.2. Структура программы

Неформально структуру программы можно представить следующим образом:

```
Program < Имя_программы >;
<Раздел указания используемых модулей>
<Раздел деклараций>:
        < Секция объявления констант>
        < Секция объявления типов>
        < Секция объявления переменных>
        < Секция объявления процедур и функций>
begin
        <Операторы, описывающие алгоритм>
end.
```

Раздел указания используемых модулей помещается в том случае, если в тексте программы используются константы, переменные, процедуры или функции, определенные в стандартных модулях Турбо-Паскаля или модулях, созданных пользователем.

Пример записи:

Uses CRT, Mylib; {Использовать модули CRT, Mylib }

Раздел деклараций (объявлений объектов программы) не является обязательным, однако без его использования можно написать только самые примитивные программы. Раздел состоит из нескольких секций, причем однотипные секции могут повторяться в разделе произвольное число раз. Порядок секций должен быть установлен в соответствии со следующим правилом:

> Если в объявлении какого-либо элемента В (константы, типа, переменной, процедуры, функции) используется элемент А (константа, тип и т.п.), то элемент А должен быть описан перед элементом В.

Каждая секция раздела идентифицируется и начинается специальным ключевым словом, а элементы, объявляемые в ней, отделяются символом точка с запятой. Признаком окончания секции является ключевое слово следующей секции или ключевое слово **begin,** выполняющее функцию начала исполнимой части программы, содержащей операторы.

Приведем примеры записи отдельных секций.

Секция констант используется для того, чтобы присвоить идентификаторы (имена) константам. В этом случае далее в тексте программы можно вместо констант использовать эти имена.

Пример:

```
Const
        nmax = 10;
        name = ' file.dat';
```

Секция типов появляется тогда, когда программисту недостаточно так называемых стандартных типов, предоставляемых по умолчанию. Применительно к данным тип определяет диапазон или перечень разрешенных значений, а также перечень допустимых операций над ними.

К стандартным типам относятся:

- целые типы (integer, byte, word и другие);
- вещественные типы (real, double и другие);
- булевский тип (boolean);
- символьный тип (char);
- строковый тип (string);

и ряд других.

Булевский тип определяет всего две константы (TRUE и FALSE), а операции над величинами булевского типа – операции булевой алгебры, для обозначения которых используются следующие буквенные обозначения:

```
and  - конъюнкция (логическое умножение)
or   - дизъюнкция (логическое сложение)
xor  - отрицание равнозначности
not  - отрицание
```

Переменные и константы символьного типа могут принимать значения из множества символов таблицы ASCII (американский стандартный код обмена информацией).

Переменные и константы строкового типа представляют собой цепочки символов из множества символов таблицы ASCII, рассматриваемые как единое целое. В то же время с позиций обработки строку можно рассматривать и как массив символов ASCII.

```
Type
Season = (Spring, Summer, Autumn, Winter);
DayOfMonth = 1 .. 31;
```

В дополнение к стандартным в Паскале можно объявить и пользовательские типы. В качестве примера приведем следующие объявления:

Здесь


```
and  - конъюнкция (логическое умножение)
or   - дизъюнкция (логическое сложение)
xor  - отрицание равнозначности
not  - отрицание
```


`Season` – перечислимый тип (значения констант этого типа задаются перечислением в круглых скобках), а допустимые операции – операции сравнения и присваивания;

`DayOfMonth` – интервальный тип (константы этого типа – все целочисленные значения в интервале от 1 до 31), а допустимые операции – те

же, что и для целых чисел, с тем только ограничением, что результаты арифметических операций над ними не должны выходить за границы интервала.

Выше были рассмотрены только так называемые скалярные типы – это типы, в которых любая переменная может одновременно принять значение только одной константы.

В то же время в языке определены и так называемые ***структурные*** типы

- массив (array);
- множество (set);
- запись (record);
- файл (file);
- объектный тип (object).

Структурные типы служат для объявления и использования переменных, которые могут одновременно хранить много значений. В качестве примера приведем объявление типа массив:

```
Type
     Arr10 = array [1 .. 10] of Integer;
```

Здесь определен тип данных, представляющих собой массив целых чисел. Их количество равно 10. В квадратных скобках указывается, что номер отдельного элемента массива может принимать значения от 1 до 10. Эта запись называется ***граничной парой***.

Секция переменных предназначена для объявления типов программных переменных, используемых в программе. Синтаксис секции имеет вид:

```
Var
     <Имя переменной> : <имя типа>;
     <Список имен переменных> : <имя типа>;
     <Имя массива> : array [<граничная пара>] of <имя типа элемента массива>;
```

Примеры объявления переменных:

```
Var
     x : real;
     i, j : integer;
     st : array[1..25] of char;
     a, b : array[1..100] of real; {Объявление двух массивов}
```

Объявление процедуры и функции имеет такую же структуру, что и программа, с небольшими отличиями, и будет рассмотрено позже.

Раздел операторов содержит запись алгоритма в виде последовательности операторов Паскаля. В данном разделе мы рассмотрим только те операторы, которые необходимы для кодирования базовых операций и управляющих структур, использованных для записи алгоритма на псевдокоде. Прочие операторы можно изучить позже, когда будут приобретены навыки разработки программ и когда в этом возникнет необходимость.

Оператор – это предложение на языке Паскаль, которое идентифицируется одним или несколькими ключевыми словами, и признаком конца которого является символ «точка с запятой». Операторы имеют различающиеся правила построения (синтаксис), и знакомство с ними лучше всего обеспечить при рассмотрении каждого в отдельности.

3.2. Средства кодирования вычислительных операций

3.2.1. Операция ввода

Для кодирования операции ввода с внешнего устройства предусмотрены следующие операторы:

Read (<список переменных>);	Ввод с клавиатуры значений для переменных по списку.
ReadLn(<список переменных >);	В конце ввода удалить остаток строки (если имеется) без обработки.
Read(<fl>,< список переменных >);	Ввод (чтение) из файла, на который указывает файловая переменная fl.
ReadLn(<fl>,< список переменных >);	Ввод из файла, на который указывает файловая переменная fl, и перемещение указателя на начало следующей строки файла. (Относится только к текстовым файлам).
ReadKey	Чтение ASCII-кода одного символа с клавиатуры (без эхо-печати). ReadKey – функция, и ее вызов правильно помещать следующим образом: ch := ReadKey;

Элементами списка ввода могут быть исключительно скалярные переменные. (Напомним, что скалярной переменной является такая, которая в любой момент времени может принимать только одно значение).

Скалярной переменной считается простая переменная, элемент массива, отдельное поле записи, символы и строки символов.

Элементы в списке переменных отделяются друг от друга запятыми. Число переменных в списке – произвольное. Элементы на устройстве ввода должны отделяться пробелами. Исключение составляет случай, когда переменная в списке – строкового типа. Суть исключения предлагается посмотреть в справочной литературе.

Примеры:

readln(a, b); {чтение с клавиатуры двух чисел }

read(w[5]); {чтение с клавиатуры одного числа и запись его в 5-й элемент массива w}

Ввод с клавиатуры может производиться в двух режимах:

- ввод ***с эхо-печатью*** производится следующим образом: код нажатой клавиши направляется в буфер ввода и одновременно на экран (обеспечивается операторами read и readln);
- ввод ***без эхо-печати***: код нажатой клавиши направляется в буфер ввода и оттуда в переменную символьного типа; при этом код на экран не отображается (ввод обеспечивается оператором ReadKey).

Ниже приведен пример кодирования оператора ввода:

ввод(w[3], b, c)	readln(w[3], b, c);

3.2.2. Операция присваивания

Синтаксис операции:

<имя-переменной> **:=** <выражение>**;**

Выражение, записанное справа от знака присваивания (**:=**), и переменная слева от него должны быть одного и того же типа. Элементами выражения могут быть константы, скалярные переменные и указатели функций. Набор операций и их приоритетность (старшинство) для выражений можно найти в любом описании языка. Для изменения порядка вычисления выражения используют круглые скобки. Если выражение не помещается в одной строке, его можно продолжить в следующей. При этом важно знать, что разбивать выражение можно только в том месте, где располагается разделитель (пробел, скобка, запятая).

Примеры записи:

```
(1) x1 := (-b + sqrt(b * b - 4 * a * c)) / (2*a); {тип real}
(2) ok := x1 <= 0.87e-2; {тип результата - boolean}
(3) st := 'Корни уравнения' + ' не вещественные'; {тип string}
```

3.2.3. Операция вывода

Для кодирования операции вывода на внешние устройства предусмотрены следующие операторы:

Write (<список-вывода>);	Вывод на внешнее устройство значений списка согласно по списку вывода.
WriteLn(<список-вывода>);	В конце операции сменить строку. (применяется только при выводе текстовой информации)

Write(<fl>,< список-вывода >);	Вывод значений в файл, на который указывает файловая переменная fl.
WriteLn(<fl>,< список-вывода >);	То же, что и выше, и в конце сменить строку (Относится только к текстовым файлам).

Элементами списка ввода могут быть

- при выводе на экран, на принтер или в текстовый файл - скалярные переменные или выражения (таких типов, которые при необходимости могут быть преобразованы в строки символов);
- при выводе в файл (типизированный, но не текстовый) переменные скалярные или переменные структурного типа; вывод производится без какого либо преобразования информации.

При выводе информации на экран монитора, на принтер или в текстовый файл преобразование элемента списка вывода, не являющегося символьным, в цепочку символов производится автоматически. При этом вид выводимого значения числа зависит от его типа и либо имеет вид «по умолчанию», либо задается пользователем в списке вывода специальным образом (этот случай называется форматным выводом). ***Форматный вывод*** позволяет точно указать форму представления выводимых значений. Детали можно найти в описании языка.

Следует иметь в виду, что один оператор **`Write`** выводит цепочку символов, длина которой определяется списком вывода, причем вся цепочка размещается в пределах одной строки. Кроме того, следующий оператор **`Write`** продолжит вывод в той же строке, добавляя новые символы в конец имеющейся строки. Только после вывода с помощью оператора **`WriteLn`** формирование строки завершится и произойдет переход к следующей строке. В результате этого вывод следующим оператором **`Write`** или **`WriteLn`** начнет формировать следующую строку. Отсюда следует, что, при необходимости вывести несколько строк текста, потребуется такое же количество операторов **`WriteLn`**.

Примеры кодирования:

вывод('Коэффициенты ',a,b)	`Write('Коэффициенты ',a,b);`
	или
	`WriteLn('Коэффициенты ',a,b);`

вывод(k,x)	`Write(k:3,x:12:2);`
	или
	`WriteLn(k:3,x:12:2);`

Во втором примере переменные выводятся в режиме форматирования: целая переменная ***k*** должна быть размещена в трех позициях строки (знакоместах), а вещественная переменная ***x*** - в 12-ти позициях, в естественной форме и такой, в которой дробная часть числа будет содержать 2 цифры.
Так, например, если значения переменных равны ***k***=-2 и ***x***=-12.367, то при выполнении оператора **«Write(k:3,x:12:2);»** будет сформирована следующая строка символов:

	-	2							-	1	2	.	3	7	

В последней позиции условно изображен курсор, чтобы показать позицию, в которую будет выводиться первый символ следующим оператором вывода.

3.3. Средства кодирования управляющих конструкций

Управляющая конструкция «Следование», как и при записи алгоритма, не требует ничего. Она обозначается за счет одинакового отступа каждого нового оператора от левого края листа. Ниже будут приведены сведения о кодировании двух других управляющих конструкций.

3.3.1. Кодирование структуры «Ветвление»

Для кодирования этой структуры в первую очередь может быть использован оператор **if-then-else**.

Синтаксис оператора if допускает два варианта записи:

if <условие> then <составной оператор>;
if <условие> then <составной оператор> else <составной оператор>;

Здесь ***составной оператор*** - это один (любой) оператор Паскаля или любое количество операторов Паскаля, ограниченное ключевыми словами ***begin*** и ***end***.

Ниже приведены различные варианты кодирования управляющей структуры «Ветвление»:

а) Одноальтернативная конструкция:

если <условие> то	**if** <условие> **then**
операция	<оператор>;
все	{end if}

или

если <условие> то	**if** <условие> **then** **begin**
операция операция	<оператор> . . . <оператор>
все	**end**; {if}

б) Двухальтернативная конструкция:

если <условие> то	**if** <условие> **then**
операция	<оператор>
иначе	**else**
операция	<оператор>;
все	{end if}

или

если <условие> то	**if** <условие> **then**
операции	**begin** <оператор> . . . <оператор> **end**
иначе	**else**

операции	**begin**
	<оператор>
	. . .
	<оператор>
	end;

все	{if}

в) Многоальтернативная конструкция.

Ее можно рассматривать как базовую, из которой можно построить и приведенные выше, удалив из данной ненужные части.

(Приводится пример только для случая, когда каждая альтернатива заключается в выполнении одного оператора).

если <условие 1> то	**if** <условие 1> **then**
<операция 1>	<оператор 1>
инес <условие 2> то	**else if** < условие 2> **then**
<операция 2>	< оператор 2>
инес <условие 3> то	**else if** < условие 3> **then**
<операция 3>	< оператор 3>
.....	. . .
иначе	**else**
<операция k>	< оператор k>**;**
все	<end if>

В языке Паскаль имеется еще один оператор, который может быть использован для кодирования многоальтернативного выбора – это оператор ***Case <селектор> of***. Его применяют в том случае, если каждый из вариантов сопоставляется одному из значений селектора. В качестве селектора может быть использовано выражение или переменная перечислимого типа. В качестве значения селектора чаще всего используются целые числа и символы алфавита. Подробности и примеры его использования предлагается взять в литературе.

3.3.2. Кодирование структуры «Цикл»

Для организации циклов в языке Паскаль имеются три оператора. В двух из них в теле цикла, согласно синтаксису, может быть помещен только один оператор. Однако их можно использовать для кодирования циклических конструкций, содержащих в теле цикла произвольное число операторов. Эта возможность обеспечивается за счет использования составных операторов, правила образования которых были приведены выше.

а) **Цикл-пока**

цикл-пока <условие>	**while** <условие> **do**
<операции>	<составной оператор>;
кцикл	{end while}

б) **Цикл-до**

цикл	**repeat**
<операции>	<оператор 1> . . . <оператор k>
кцикл-до <условие>	**until** <условие>;

в) **Цикл-со-счетчиком**

цикл-для <сч> от <нач> до <кон>	**for** <сч> := <нач> **to** <кон> **do**
<операции>	<составной оператор>**;**
кцикл	{end for}

Примечание. В Паскале определено два варианта оператора **for:**

- 1-й вариант: **for** <сч> := <нач> **to** <кон> **do** <оператор>;
- 2-й вариант: **for** <сч> := <нач> **downto** <кон> **do** <оператор>;

В первом операторе, после каждого выполнения тела цикла, значение счетчика (целочисленная скалярная переменная) увеличивается на единицу; и в последний раз цикл выполняется при значении счетчика, равном ***кон***. А, следовательно, начальное значение ***нач*** должно быть меньше конечного.

Во втором операторе после каждого выполнения тела цикла значение счетчика цикла, наоборот, должно уменьшаться на единицу, и, следовательно, конечное значение должно быть меньше начального.
Примеры записи циклов.

```
{Конструкция цикл-пока:}
k:=0;
while (k<=10) do begin
   writeln(k);
   k := k + 1;
end; {while}
```

```
{Конструкция цикл-до:}
k:=0;
repeat
   writeln(k);
   k := k + 1;
until (k>10);

{Конструкция цикл-со-счетчиком}
for k:=0 to 10 do begin
   writeln(k);
   k := k + 1;
end; {for}
```

3.4. Кодирование алгоритма в целом

Выше были рассмотрены рекомендации по кодированию отдельных элементов алгоритма на языке Паскаль. Теперь можно изложить правила кодирования алгоритма в целом.

1. Процесс кодирования рекомендуется проводить последовательно с первой строки до последней. И до тех пор, пока кодирование текущей строки (объявления объекта, операции или элемента управляющей структуры) не закончено, не приступать к кодированию следующей строки.
2. Признаком конца любого оператора языка является символ «;» (точка с запятой).
3. Признаком конца программы является символ «точка», помещенный после ключевого слова ***end*.**

4. При кодировании заголовка алгоритма («Алгоритм <имя-алгоритма>») используется оператор ***program***. Его синтаксис имеет вид:

```
program <имя-программы>;
```

   Следует иметь в виду, что имя программы – это идентификатор, оно считается внешним, и поэтому не должно превышать восьми символов. Отметим, что, в соответствии с формальным описанием языка, заголовок программы (но не подпрограммы) можно опускать. Тем не менее, рекомендуется заголовок кодировать, присвоив при этом программе имя, совпадающее с именем файла, под которым текст программы будет храниться во внешней памяти компьютера.
5. Ключевые слова <u>начало</u> и <u>конец</u> кодируются ключевыми словами языка Паскаль ***begin*** и ***end*** соответственно.
6. Все строки алгоритма, для которых в Паскале нет аналогов, рекомендуется сохранять и в программе в виде комментариев. Примеры реализации этой рекомендации можно увидеть выше, в разделах, в которых излагаются правила кодирования управляющих структур. Так, например, в конструкции «если-то-иначе-все» для случая, когда альтернатива, завершающаяся ключевым словом «<u>все</u>», представляется одним оператором, в Паскале не имеет аналога. В этом случае строку со словом «все» в программе рекомендуется представить в виде комментария в одной из форм:

{все}
{end if}

   А при кодировании слова «кцикл» в конструкциях «цикл с предусловием» и «цикл со счетчиком», в случае, когда тело цикла кодируется одним оператором Паскаля, в комментарии рекомендуется поместить не только слово ***end***, но и ключевое слово, показывающее вид конструкции (***end for*** или ***end while***).
7. При записи текста на Паскале обязательно сохранять ступенчатую запись, строго соответствующую ступенчатой записи в кодируемом алгоритме. Это позволит сохранить не только семантическое, но и структурное (графически) соответствие алгоритма и программы, а это важно не только при отладке, но при последующей модернизации программы, если в этом возникнет необходимость.
8. Все заголовки планов (фрагментов) алгоритма, которые при сборке алгоритма в единый (детальный) алгоритм были записаны в виде комментариев, рекомендуется, также в виде комментариев, сохранить и в программе. В этом случае программа приобретет свойство самодокументированности, чрезвычайно важное для этапа эксплуатации программы.

9. В дополнение к сказанному в начале текста программы (перед или непосредственно после заголовка) рекомендуется поместить комментарии, указав в них содержательное описание назначения программы и дату создания или последнего изменения ее. При необходимости, там же нужно поместить сведения об авторе и/или его контактные данные.

Заключение

В языке Паскаль имеется значительно большее количество операторов. Однако на начальном этапе вполне достаточно рассмотренных выше. И вам, читатель, предлагается начать разработку программ, используя эти рассмотренные. По мере приобретения опыта вы сможете самостоятельно овладеть и остальными средствами языка.

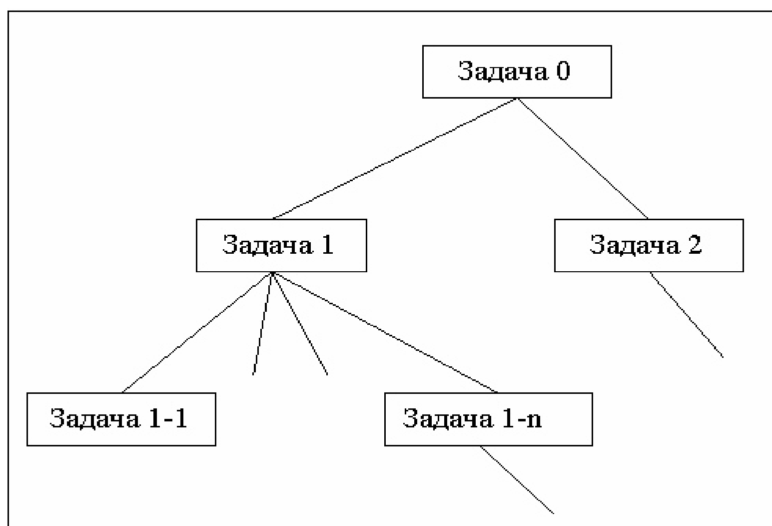
Исключение составляют средства оформления подпрограмм, которые реализуют так называемые вспомогательные алгоритмы. Эти средства языка будут рассмотрены несколько позже в отдельном разделе.

Вопросы для самоконтроля

1. На какие группы можно разделить набор символов языка Паскаль?
2. Что такое лексема?
3. Какие категории лексем вам известны?
4. Что представляет собой идентификатор?
5. Допустимо ли использование зарезервированных слов Паскаля в качестве идентификаторов пользователя?
6. Чему равна допустимая длина идентификатора?
7. Какие идентификаторы являются стандартными?
8. Какие формы записи чисел используются в языке Паскаль?
9. Какой символ используется для обозначения шестнадцатеричного числа?
10. Что представляет собой комментарий и как он оформляется?
11. Какие символы используются в качестве разделителей лексем?
12. Какое количество символов-разделителей допускается между любыми двумя лексемами?
13. В чем отличие операторов `write` и `writeln`?
14. Что понимается под выражением «форматный вывод»?
15. Как в тексте на Паскале представляется конструкция «следование»?
16. Какие операторы языка могут использоваться для кодирования альтернативных конструкций?
17. Какие операторы Паскаля предназначены для кодирования циклов?

4. РАЗРАБОТКА АЛГОРИТМОВ МЕТОДОМ ПОШАГОВОЙ ДЕТАЛИЗАЦИИ

Одним из основных методов решения сложной задачи в любой области человеческой деятельности является метод декомпозиции, когда сложная задача разбивается на несколько более простых задач - на подзадачи. Этот процесс продолжается до тех пор, пока выделенные после очередной декомпозиции подзадачи не будут упрощены настолько, что их в состоянии будут решать конкретные исполнители. Процесс декомпозиции можно изобразить графически:



Аналогичный подход используется и при разработке программ, а метод, использующий его, получил название ***метода пошаговой детализации*** или метода нисходящего проектирования программы. При методе пошаговой детализации алгоритм решения исходной задачи представляется как совокупность более простых алгоритмов, которые, в свою очередь, разбиваются на еще более простые и т.д. Выделенные алгоритмы могут быть как не имеющими законченного функционального назначения, так и функционально независимыми. В первом случае выделяется часть алгоритма, которую проще разработать отдельно и затем вставить в исходный алгоритм.

Во втором случае выделенные алгоритмы можно проектировать как самостоятельные задачи и оформлять на языке программирования в виде подпрограмм. Как известно, подпрограмма не решает всей задачи, она – часть программы, реализующая строго определенные функции.

Разработка подпрограммы должна начинаться с разработки внешней спецификации, в которой обязательно выделяются входные и выходные данные, которые в языках программирования называются соответственно входными и выходными параметрами. Некоторые входные данные могут быть одновременно и выходными, т.е. допускать изменение значений в подпрограмме.

Во внешней спецификации также обязательно фиксируются ограничения на значения входных аргументов, которые отражаются в разделе “Аномалии входных данных”. Реакции подпрограммы на аномалии рекомендуется обеспечивать через ее выходные данные.

Процесс пошаговой детализации алгоритма конечен, а число шагов зависит от сложности задачи и квалификации разработчика. Его рекомендуется завершать тогда, когда на очередном шаге получается текст, запись которого на языке программирования не представляет проблем для разработчика. В пределе этот процесс можно завершить тогда, когда каждая строка алгоритма может быть заменена одним оператором языка программирования.

4.1. Структура алгоритма

Запись каждой детализации должна быть оформлена так, как это изображено ниже: (а) – для случая несамостоятельной части алгоритма; (б) – для случая алгоритма, выделяемого в самостоятельную подпрограмму.

План алгоритма “Название”
Внутренние переменные: <список-переменных>
Начало
    <Описание действий>
Конец

(а)

Алгоритм “Название”
Входные переменные: <список-переменных>
Выходные переменные: <список-переменных>
Внутренние переменные: <список-переменных>
Начало
    <Описание действий>
Конец

(б)

Отметим, что для случая (а) раздел внутренних переменных вставляется только в том случае, если при разработке указанного плана недостаточно внутренних переменных алгоритма предшествующего уровня. В любом случае данные переменные являются глобальными для этой части алгоритма. ***Глобальными*** переменными считаются такие, которые определены для алгоритма верхнего уровня, с которого осуществляется переход на текущий уровень.

Примечание. В дальнейшем с целью сокращения записи вместо слов “план алгоритма” будет употребляться одно слово “план”.

4.2. Описание действий

Описание действий в алгоритме (или плане) должно быть выполнено исключительно по правилам структурного программирования, то есть только с использованием конструкций “следование”, “ветвление” и “цикл”. Главным следствием этих требований является то, что в описании действий будет только одна логическая точка начала и одна точка конца. И, кроме того, запись будет иметь явно выраженную структуру, которая может быть использована для построения минимально необходимого количества тестов, достаточного для формирования уверенности в правильности работы алгоритма (плана).

После записи последовательности действий алгоритма (плана), они должны быть проверены, с целью выявить и устранить возможные ошибки.

При проверке правильности записанного алгоритма (плана) необходимо исходить из предположения, что действия, не детализированные на данном уровне, всегда выполняются правильно, и, следовательно, проверяется только логика действий данного уровня детализации.

В основе этого лежит следующее правило: задачей каждого шага детализации является построение фрагмента алгоритма, не содержащего ошибок. Поэтому правильность не детализированного в каждом шаге действия будет обеспечена на том шаге, в котором оно будет «раскрываться».

В зависимости от степени сложности алгоритма проверка может заключаться в простом просмотре его текста, либо в выполнении тестов. Поскольку на данном этапе работы полного текста на языке программирования, как правило, нет, эта проверка выполняется человеком вручную. Такая проверка получила название ***трассировки***. Для получения уверенности в том, что в результате трассировки выявлены все возможные ошибки, проверка выполняется не на одном, а на нескольких тестах.

Минимально необходимое число тестов определяется на основе применяемой стратегии тестирования. Поскольку проверке подвергается «раскрытый» текст алгоритма, для тестирования используется стратегия «белого ящика». Количество же необходимых тестов определяется критерием, положенным в основу разработки тестов. Напомним, что могут быть использованы следующие критерии: «все операторы», «все решения», «все условия» и другие.

Результатом проверки алгоритма должно быть не только убеждение в безошибочности алгоритма, но и набор тестов, использованный при его разработке.

4.3. Обратная подстановка

После построения всех планов (детализированных алгоритмов) необходимо выполнить (обратную) подстановку их в основной алгоритм. При этом все внутренние глобальные переменные собираются вместе и помещаются в разделе “Внутренние переменные” основного алгоритма.

Процесс подстановки операций, реализующих алгоритм, рекомендуется выполнить следующим образом:

Подстановка начинается с основного алгоритма. При этом любая строка алгоритма, рассматриваемая как имя плана, реализующего подзадачу, превращается в комментарий, а затем в следующих строках помещается текст соответствующего плана, записанный в нем между словами «Начало» и «Конец».

Если в тексте алгоритма встречается строка, действие в которой при детализации будет представлено подпрограммой, то в соответствующей позиции алгоритма помещается только указатель (вызов) подпрограммы.

Процесс подстановки повторяется во всех случаях, когда встречается очередная ссылка на другой план. В результате описанного процесса за один раз из множества частных планов, представленных на разных уровнях детализации, будет получен детальный алгоритм решения задачи.

Если описанный процесс выполняется сразу на языке программирования, то его результатом будет программа. Если же алгоритм записывается на языке псевдокода или русском алгоритмическом языке, то его надо перекодировать на заданный язык программирования. Отметим, что процесс кодирования рутинный и не вносит (не должен вносить!) никаких изменений в алгоритм.

В дополнение к этому надо собрать воедино все тесты, которые были разработаны на этапах проверки отдельных планов и алгоритмов. Этот набор тестов можно будет использовать для испытания программы на ЭВМ.

4.4. Пример

Проиллюстрируем изложенный материал на примере разработки программы, которая будет использована для иллюстрации работы подпрограммы сортировки чисел. После постановки задачи и выбора метода решения разрабатывается внешняя спецификация будущей программы. Опустим первые два этапа и приведем сразу внешнюю спецификацию.

4.4.1. Разработка внешней спецификации

Вид внешней спецификации приведен на следующей странице.

Внешняя спецификация

Назначение: Программа сортировки чисел методом полного перебора.

Входные данные:

$a_1, a_2, \dots a_n$ - неупорядоченный набор чисел [вещественные]

n - количество чисел [целое]

Выходные данные:

Получить на экране монитора результаты в следующем виде:

Сортировка чисел.
Алгоритм “Полный перебор”
Всего чисел = <n>
Исходный массив :
<a_1> <a_2> . . . <a_5>
<a_6> … … <a_{10}>
. . .
<a_n>
Упорядоченный массив:
<b_1> <b_2> . . . <b_5>
<b_6> … … <b_{10}>
. . .
<b_n>

где:

$b_1, b_2, \dots b_n$ - числа [вещественные]

Аномалии входных данных:

1. n <= 0
   Вывести сообщение “Ошибка” и завершить работу программы
2. n > nmax {nmax – константа, будет определена в программе}
   Вывести сообщение “Объем массива превышает <nmax>”.
   Присвоить переменной n значение nmax и продолжить работу.

4.4.2. Разработка главного алгоритма

Алгоритм будет разрабатываться на псевдокоде (русском алгоритмическом языке). Вначале запишем укрупненный алгоритм.

```
Алгоритм "Сортировка чисел"
Внутренние переменные:
        a[1..nmax] : массив, вещ
        n : цел
        ошибка : лог {флаг ошибки во входных данных}
Начало
        Вывод-представления
        Ввод-и-печать-входных-данных  {формирует ошибку}
        Если (нет ошибки) то
                Сортировка
                Вывод-результата
        Все
Конец
```

Это первый уровень детализации. В нем все действия алгоритма еще не раскрыты и подлежат детализации. Однако перед ее началом мы должны проверить представленный здесь алгоритм, чтобы быть уверенными, что в нем нет ошибок. В нашем случае проверка будет сведена к внимательному прочтению текста. Как видно из текста, логика алгоритма ошибок не содержит: сортировка будет выполняться только в том случае, если при вводе исходных данных не было ошибки. Поэтому можно переходить к детализации отдельных операций алгоритма.

Детализацию начнем с первой операции. Она будет составной частью основного алгоритма.

```
План " Вывод-представления "
Начало
        Вывод("Сортировка чисел.
                     Алгоритм "Полный перебор"")
Конец
```

Проверка приведенного здесь плана также может быть сведена к простому чтению текста на предмет поиска описки (опечатки). Просмотр показывает, что ошибок в тексте нет. Кроме того, мы видим, что далее детализировать операции нет необходимости: в языке программирования Паскаль для кодирования этого текста потребуется два оператора **writeln** (по одному на каждую строку выводимого текста).

Далее приступим к раскрытию операции «Ввод-и-печать-входных-данных». Поскольку вводимые данные будут размещаться в переменных, которые уже объявлены выше, в приводимом тексте нет раздела "Внутренние переменные".

```
План “Ввод-и-печать-входных-данных”
Начало
        Вывод (“Всего чисел = ”)
        Ввод (n)
        Если (n <= 0) то
                Вывод(“Ошибка”)
                Ошибка := true
        Иначе
                Если (n > nmax) то
                    Вывод(“Объем массива превышает”, nmax)
                    n := nmax
                все
                Ошибка := false
                Ввод-массива  {Требуется детализация}
                Вывод (“Исходный массив :”)
                Вывод-массива  { Требуется детализация}
        Все
Конец
```

Проверим полученный алгоритм (точнее план алгоритма).

Он сводится к последовательному выполнению трех действий: выводу запроса, вводу числа ***n***, и конструкции если (трехальтернативной), действия в которой связываются с введенным значением ***n***.

Обратите внимание, что в третьей альтернативе имеется две операции, которые потребуют последующей детализации. Но при проверке данного алгоритма мы будем считать, что они выполняются безошибочно. Эта безошибочность будет обеспечена нами при реализации указанных операций в последующих планах.

Итак, приступим к проверке.

В нашем случае выделено четыре варианта поведения проверяемого алгоритма в зависимости от значения введенной величины *n*. Для проверки используется один из методов стратегии разработки тестов, получившей название ***стратегия белого ящика***. В кругу программистов вместо слова «стратегия» чаще используется слово «метод». Как содержание, так и виды различных методов тестирования будут рассмотрены позже. Сейчас отметим только, что для проверки данного плана алгоритма будет использован метод «все пути».

По критерию «все пути» требуется ровно 3 теста: два теста для двух аномалий из внешней спецификации и третий – для корректного значения n (для случая, когда ни одна аномалия не имеет места). Положим, что *nmax* = 4, и поместим тесты в таблице.

№	Назначение	Входные данные	Правильный ответ	Результат проверки
1	Реакция на (n < 0)	–10	Текст “Ошибка” Ошибка := true	Правильно
2	Реакция на (n > nmax)	24, 12, 6, –5, 55, 0, 45, …	Текст “Объем массива превышает 4”. Ошибка := false Ввод и печать массива четырех чисел : 12, 6, -5, 55	Правильно
3	Реакция на допустимое значение n	3, 12, 6, -5	Ошибка := false Ввод и печать массива чисел : 12, 6, –5	Правильно

Вывод. Поскольку все тесты дали правильные результаты, считаем, что алгоритм ошибок не содержит, и можно перейти к детализации других операций.

Теперь разберем алгоритм операции “Ввод-массива”. Положим, что данные будут вводиться по запросу с указанием номера вводимого числа.

```
План “Ввод-массива”
Внутренние переменные:
        j : цел  {рабочая переменная – счетчик цикла}
Начало
        цикл-от  j:=1 до  n
                 вывод (“<j>-е число = ”)
                 ввод (a[j])
        кцикл
Конец
```

При проверке данного алгоритма исходим из того, что значение переменной ***n*** уже определено и имеет допустимое значение (смотрите предыдущий алгоритм). С учетом сказанного для его проверки достаточно одного теста:

Запись результатов трассировки приведена ниже.

Обратите внимание, что этот тест совпадает по данным с тестом номер 3 (смотрите выше).

Выполнение теста показывает совпадение результата с ожидаемым.

```
Дано:
n=3;
Входные числа :  12, 6, -5
Результаты выполнения теста:
        1-е число = 12
        2-е число = 6
        3-е число = -5
```

Вывод: Результаты верны. Следовательно, алгоритм ошибок не содержит.

Далее разберем алгоритм “Вывод-массива”.

Пусть в соответствии с внешней спецификацией элементы выводимого массива должны размещаться по пять чисел в строке. Поэтому в алгоритме необходимо предусмотреть подсчет выведенных чисел и смену строки. Для выявления факта смены строки используем операцию целочисленного деления «остаток от деления по модулю» - ***mod***.

```
План “Вывод-массива”
Внутренние переменные:
        j : цел  {рабочая переменная – счетчик цикла}
Начало
        цикл-от  j:=1 до  n
                вывод (a[j])
                если (j mod 5)=0 то
                        сменить-строку
                все
        кцикл
Конец
```

Для проверки этого алгоритма также достаточно одного теста. Только в этом случае количество выводимых чисел должно быть более 5, чтобы убедиться в том, что строка меняется вовремя. Можно модифицировать тест номер 3, добавив еще несколько чисел.

Новая версия теста номер 3:
Дано: 7, 12 , 6, -5, 55, 0, 45, 2   {*Первое число* (7) *–количество чисел*}

Требуется:

12	6	-5	55	0
45	2			

Трассировки этого теста мы выполнять не будем. Достаточно выполнить ее «в уме». Она покажет, что ошибки нет.

Теперь разберем операцию “Вывод-результата”:

```
План "Вывод-результата"
Внутренние переменные:
        j : цел  {рабочая переменная – счетчик цикла}
Начало
        Вывод ("Упорядоченный массив:")
        Вывод-массива
Конец
```

Проверку этого алгоритма можно свести только к внимательному прочтению его текста, поскольку все составляющие его операции нами уже проверены ранее, и ошибку мы можем внести только по невнимательности. Дополнительного теста также не требуется, поскольку согласно основному алгоритму, если выполнялась операция "Ввод-и-печать-входных-данных" для теста номер 3, то выполнится и данная операция.

Итак, все операции основного алгоритма, кроме операции "Сортировка" нами детализированы. Детализировать же эту последнюю операцию мы не будем, решив реализовать ее в виде самостоятельной подпрограммы, поскольку алгоритм функционально обособлен, а программа по уже разработанному алгоритму может быть использована в качестве испытательной (или демонстрационной) для подпрограмм, реализующих различные методы сортировки.

Теперь, для того чтобы закончить разработку основного алгоритма, достаточно разработать внешнюю спецификацию подпрограммы "Сортировка", одним из результатов чего будет правильная запись указателя (вызова) подпрограммы сортировки в нашем алгоритме. Отметим, что поскольку подпрограмма должна вызываться из программы на Паскале, в ее внешней спецификации должны быть учтены особенности Паскаля.

Во-первых, имя подпрограммы должно быть построено по правилам построения идентификатора Паскаля: оно может содержать только буквы латинского алфавита и цифры, а также не превышать по длине восьми символов.

Во-вторых, если подпрограмма сортировки оформляется как внешняя, и, следовательно, может располагаться в отдельном модуле, более правильным будет отказаться от использования глобальных переменных для размещения упорядочиваемых чисел и передавать их в качестве параметров. А для этого необходимо определить новый тип данных – массив вещественных чисел заданного объема.

С учетом сказанного, внешнюю спецификацию подпрограммы сортировки можно представить следующим образом:

Назначение: “Сортировка по неубыванию методом полного перебора”
Входные данные:
a[1..n] : вещ
n : цел
Выходные данные:
a[1..n] : вещ
Аномалии входных данных: отсутствуют

В основной программе и в данной подпрограмме используется следующий тип данных:

Arr1 = array[1..nmax] of integer;

где

nmax – константа целого типа, задающая максимально возможный объем массива.

Заголовок подпрограммы на Паскале должен иметь следующий вид:

procedure SortAll(var a:Arr1; n:integer);

Используя эту внешнюю спецификацию, в основном алгоритме вызов подпрограммы сортировки можно записать одним из двух способов:

Первый способ: {с использованием обозначений псевдокода}

a := SortAll (a, n)

Примечание. Обратите внимание, что непосредственно из записи следует, что параметр **a** является как входным, так и выходным.

Второй способ: { с использованием обозначений, принятых в Паскале}

SortAll (a, n)

4.4.3. Подстановка

Теперь, наконец, можно построить полный и детальный алгоритм.

Получившийся после подстановки итоговый алгоритм приведен на следующей странице.

```
Алгоритм “Сортировка чисел”
Внутренние переменные:
        a[1..nmax] : массив, вещ
        n :  цел
        ошибка : лог {флаг ошибки во входных данных}
        j : цел   {рабочая переменная – счетчик цикла}
Начало
                {Вывод-представления}
        Вывод(“Сортировка чисел.
                Алгоритм “Полный перебор””)
                {Ввод-и-печать-входных-данных}
        Вывод (“Всего чисел = ”)
        Ввод (n)
        Если (n < 0) то
                Вывод(“Ошибка”)
                Ошибка := true
        Иначе
                Если (n > nmax) то
                        n := nmax
                все
                Ошибка := false
                        {Ввод-массива}
                цикл-от  j:=1 до  n
                        вывод (“<j>-е число =”)
                        ввод (a[j])
                кцикл
                Вывод (“Исходный массив :”)
                        {Вывод-массива}
                цикл-от  j:=1 до  n
                        вывод (a[j])
                        если (j mod 5)=0 то
                                сменить-строку
                        все
                кцикл
        Все
        Если (нет ошибки) то
                        {Сортировка}
                a := SortAll (n, a)
                        {Вывод-результата}
                Вывод (“Упорядоченный массив:”)
                        {Вывод-массива}
                цикл-от  j:=1 до  n
                        вывод (a[j])
                        если (j mod 5)=0 то
                                сменить-строку
                        все
                кцикл
        Все
Конец
```

4.4.4. Кодирование на языке Паскаль

В заключении приведем запись полученного алгоритма на языке Турбо-Паскаль.

```
Program Sorting; { Сортировка чисел }
Uses
     Module1; {Модуль содержит объявление константы nmax
              и типа Arr1=array[1..nmax] of real }
var
   a: Arr1;
   n: integer;
   Err: boolean; {флаг ошибки во входных данных}
   j: integer; {рабочая переменная – счетчик цикла}
begin
              {Вывод-представления}
   writeln('Сортировка чисел.');
   writeln('Алгоритм "Полный перебор"');
              {Ввод-и-печать-входных-данных}
   writeln('Всего чисел = ');
   readln(n);
   if (n < 0) then begin
      writeln('Ошибка');
      Err := true;
   end
   else begin
      if (n > nmax) then
         n := nmax;
      {все}
      Err := false;
         {Ввод-массива}
      for j:=1 to n do begin
         write (<j>'-е число = ');
         readln (a[j]);
      end; {кцикл}
      writeln ('Исходный массив :');
         {Вывод-массива}
      for j:=1 to n do begin
         write (a[j]);
         if (j mod 5)=0 then
            writeln;
         {все}
      end; {кцикл}
   end; {Все}
   if (not Err) then begin
         {Сортировка}
      SortAll (n, a);
         {Вывод-результата}
      writeln('Упорядоченный массив: ');
      for j:=1 to n do begin
         write(a[j]);
         if (j mod 5)=0 then
            writeln;
         {все}
      end; {кцикл}
   end; {Все}
end.
```

Для проверки разработанной программы можно даже не разрабатывать алгоритм сортировки, а подготовить «пустую» подпрограмму, то есть такую, которая между строками “начало” и “конец” не содержит ни одного оператора. Она не будет сортировать элементы входного массива, а сохранит их первоначальное расположение. Тем не менее, для проверки разработанного (главного) алгоритма этого достаточно.

4.4.5. Разработка алгоритма подпрограммы сортировки

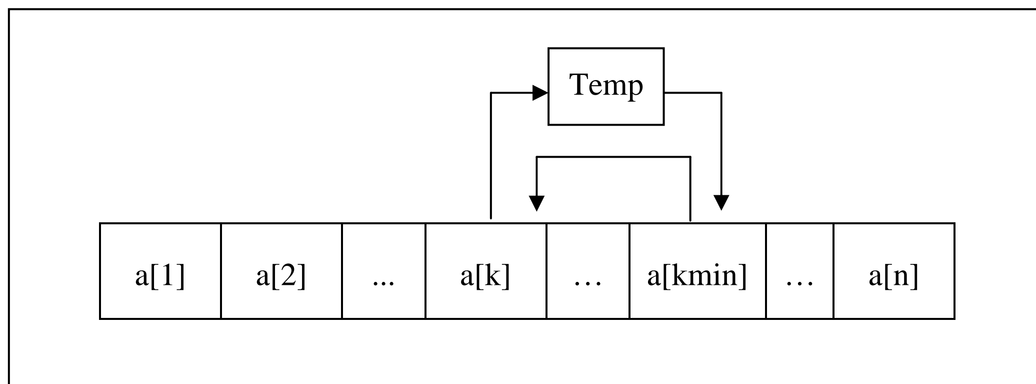
Теперь можно разработать алгоритм собственно сортировки, внешняя спецификация которого была разработана ранее.

Вначале изложим идею сортировки (упорядочивания) чисел по неубыванию методом полного перебора. В результате ее выполнения для любой пары соседних чисел должно выполняться условие $a_i \le a_{i+1}$. (Здесь a – имя массива, i – индекс элемента массива).

Положим, что подлежащий упорядочиванию массив разбит на две части: в первой части, содержащей (k-1) элементов с номерами (1,..,k-1) все элементы уже упорядочены, а во второй – с номерами (k, .., n) – нет. Это означает, что во второй части массива нет ни одного элемента, значение которого было бы меньше значений элементов с номерами (1,..,k-1).

Идея сортировки состоит в том, чтобы выбрать наименьший элемент в этой второй части и обменять его с элементом, находящимся на k-той позиции. В результате этого число упорядоченных элементов увеличится на единицу, а неупорядоченными останутся элементы с номерами (k+1, .., n). Указанный процесс надо повторить для k = 1,2, …, n-1, и в результате все n элементов массива окажутся на своих местах.

Приведем графическую иллюстрацию этого процесса.



Теперь можно построить укрупненный алгоритм подпрограммы сортировки с именем SortAll.

```
Алгоритм SortAll
Входные данные:
        a[1..n] : вещ  {массив}
        n : цел  {число элементов в массиве}
Выходные данные:
        a[1..n] : вещ  {массив}
Внутренние переменные:
        k : цел   {номер текущего элемента}
Начало
        цикл-от  k:=1 до (n-1)
                Найти-минимальный-элемент
                Переместить-мин.-элемент-на-позицию-k
        кцикл
Конец
```

Проверку данного алгоритма можно выполнить «в уме». Единственное условие, подлежащее здесь проверке, – правильно ли задается последнее значение счетчика цикла “j=(n-1)”. Это условие правильное, поскольку при этом значении j минимальный элемент ищется среди предпоследнего (n-1) и последнего (n) элементов массива.

Теперь можно приступить к детализации операций в теле цикла.
Выше уже говорилось, что минимальный элемент надо искать среди элементов с номерами (k, k+1, ... , n). Этот фрагмент алгоритма приведен ниже:

```
План “Найти-минимальный-элемент”
Внутренние переменные:
        kmin : цел      {номер минимального элемента}
        j : цел         {номер проверяемого элемента}
Начало
        kmin := k
        цикл-от j:=k+1 до n
                если a[j] < a[kmin] то
                        kmin] := j
                все
        кцикл
Конец
```

Проверим полученный фрагмент (плана алгоритма).

Во-первых, согласно условиям, выработанным выше для переменной k, цикл будет выполняться, по меньшей мере, один раз.

Во-вторых, для проверки тела цикла по критерию «все условия» необходимо, чтобы условие *a[j] < a[kmin]* один раз выполнялось, а другой раз – нет.

Отсюда, для проверки алгоритма требуется два теста.

Для проверки того, корректно ли записано условие неупорядоченности,, можно проверить алгоритм для случая, когда имеются два равных и минимальных по величине элемента. Для этой цели может быть добавлен еще один (третий) тест.

Поместим эти тесты в таблице:

№	Назначение	Входные данные	Правильный ответ	Результат проверки
1	Проверка выполнения условия неупорядоченности	n=2 a: [5, 12] k=1	k=1	Правильно
2	Проверка невыполнения условия неупорядоченности	n=2 a:[12, 2] k=1	k=2	Правильно
3	Проверка условия на граничное значение	n=2 a:[12, 12] k=1	k=1	Правильно

Замечание. Для испытаний программы на компьютере все эти тесты можно объединить в один, поскольку результаты его исполнения будут отображены на экране.

Теперь разработаем алгоритм «Переместить-мин.-элемент-на-позицию-k». Он имеет вид:

```
План "Переместить-мин.-элемент-на-позицию-k"
Внутренние переменные:
        temp : вещ      {буфер обмена}
Начало
        temp := a[k]
        a[k] := a[kmin]
        a[k] := temp
Конец
```

А для проверки его достаточно просто внимательно прочитать текст, чтобы убедиться в его правильности.

Теперь можно объединить полученные планы, получив итоговый алгоритм. Итоговый алгоритм приведен на следующей странице.

```
Алгоритм SortAll
Входные данные:
a[1..n] : вещ {массив}
n : цел        {число элементов в массиве}
Выходные данные:
a[1..n] : вещ {массив}
Внутренние переменные:
      k : цел          {номер текущего элемента}
      kmin : цел       {номер минимального элемента}
      j : цел          {номер проверяемого элемента}
      Temp : вещ       {буфер обмена}
```

```
Начало
      цикл-от  k:=1 до (n-1)
                    {Найти-минимальный-элемент}
            kmin := k
            цикл-от j:=k+1 до n
                    если a[j] < a[kmin] то
                          kmin] := j
                    все
            кцикл
                    {Переместить-мин.-элемент-на-позицию-k}
            Temp := a[k]
            a[k] := a[kmin]
            a[kmin] := Temp
      кцикл
Конец
```

И, наконец, можно закодировать полученный текст на языке программирования. Текст подпрограммы сортировки на языке Паскаль в форме процедуры (procedure), приведен на следующей странице.

Отметим, что ее испытания на компьютере можно начать только после того, как будет установлено, что вызывающая программа (в нашем случае программа Sorting , разработанная выше) работает правильно. А для испытаний процедуры SortAll достаточно одного приведенного выше теста.

И еще одно замечание. Для того чтобы транслировать и выполнить подпрограмму, по правилам языка Паскаль ее надо либо вставить в текст программы в область деклараций, либо поместить в отдельный файл, оформив его как файл типа **Unit** и включить ссылку на него в текст главной программы. Как это сделать, будет показано позже.

```
procedure SortAll(var a : Arr1; n : integer);
var
   k : integer;    {номер текущего элемента}
   kmin : integer; {номер минимального элемента}
   j : integer;    {номер проверяемого элемента}
   Temp : real;    {буфер обмена}
begin
   for k:=1 to (n-1) do begin
       {Найти-минимальный-элемент}
    kmin := k;
    for j:=k+1 to n do
       if a[j] < a[kmin] then
         kmin] := j;
       {все}
    {кцикл}
        {Переместить-мин.-элемент-на-позицию-k}
     Temp := a[k];
     a[k] := a[kmin];
     a[kmin] := Temp;
   end; {кцикл}
end;
```

4.5. Вопросы для самоконтроля

1. Какова структура главного и вспомогательного алгоритма?
2. В чем отличие внутренних переменных от входных или выходных переменных?
3. Какие переменные называются глобальными?
4. Когда отдельные действия алгоритма целесообразно оформлять в виде вспомогательных алгоритмов (подпрограмм)?
5. В какой форме будет представлена часть алгоритма, которая при разработке оформляется как план алгоритма?
6. Что означает термин «трассировка»?
7. Как выполнить обратную подстановку?
8. В какой форме в алгоритме можно записать ссылку на вспомогательный алгоритм?
9. На каком этапе правильно разрабатывать тесты?

5.ТЕСТИРОВАНИЕ

Тестирование – процесс испытания программы на тестах с целью обнаружения в ней ошибок.

Тест – контрольный пример, в котором для конкретных данных известен правильный (эталонный) ответ. Иными словами, тест - совокупность входных данных и эталонного результата. Результаты, фактически полученные при выполнении теста, сопоставляются с эталонными, и на основе этого делается вывод о наличии или отсутствии ошибки.

При разработке тестов для получения эталонного ответа или просто эталона рекомендуется по возможности использовать алгоритм, отличающийся от того, который проверяется в программе. Эта рекомендация связана с тем, чтобы при разработке теста избежать повторения логической ошибки в случае, если таковая была допущена в процессе разработки программы.

Установлено, что для получения программы, свободной от ошибок по результатам тестирования, необходимо более одного теста. Число тестов определяется структурой программы и применяемой стратегией тестирования.

Тот факт, что результат, полученный при выполнении программы на тесте, не совпадает с эталонным ответом, свидетельствует о наличии ошибки в ней. Устранение ошибок в программе происходит в процессе ***отладки***. Этот процесс включает три этапа: этап локализации ошибки, этап внесения исправлений в программу и последующее тестирование. Локализация ошибки имеет целью найти то место в программе, в котором имеет место ошибка. Если тесты построены правильно, то локализовать ошибку удается сравнительно просто. Иногда локализовать место ошибки на основании одного теста не удается. Тогда возможно потребуется разработать дополнительные тесты, опираясь при этом на некоторую гипотезу о причине ошибки.

После внесения исправлений в программу рекомендуется испытать программу не только на том тесте, который обнаружил ошибку, но и на других, а именно на тех, которые проходили правильно. Эта рекомендация связана с тем, что внесенные исправления могут привести к некорректному поведению программы в других ситуациях. Процессы отладки познаются в практической деятельности и, и здесь в полном объеме не рассматривается. Отметим только, что в данной главе рассматриваются методы разработки тестов, позволяющие максимально упростить процедуру локализации ошибки.

Вопросы разработки тестов рассматриваются во многих публикациях, но систематизированного изложения их в доступной литературе найти практически невозможно. Из наиболее интересных можно рекомендо-

вать следующие книги, посвященные именно вопросам тестирования программ:

1. Майерс Г. Надежность программного обеспечения. – М.: Мир. - 1980 г.
2. Майерс Г. Искусство тестирования программ/Пер. с англ. под ред. Б.А.Позина. – М.: Финансы и статистика. - 1982 г.
3. Канер С. Тестирование программного обеспечения. – М.: Диасофт. - 2000г.

В данной главе рассматриваются основные подходы к тестированию.

5.1. Общие принципы тестирования

При разработке тестов рекомендуется руководствоваться следующим:

1. Хорошим является тест, который обнаруживает ошибку, если она имеется.
2. Каждый тест, по возможности, должен быть предназначен для обнаружения отдельной ошибки алгоритма (программы). Тест должен быть построен так, чтобы обеспечить не только факт выявления ошибки, но и локализовать место в программе, где она была допущена.
3. Тест построен неудачно, если он формирует одно и то же сообщение для нескольких ошибок.
4. Необходимо проверять, не делает ли программа того, чего не должна делать.
5. Тесты следует разрабатывать также и для неправильных или непредусмотренных входных данных.
6. Количество тестов должно быть таким, чтобы с их помощью можно было обнаружить все возможные ошибки алгоритма. Это можно обеспечить только за счет систематизированного подхода к их построению. В основе систематизации при построении лежит та или иная стратегия. Совокупность тестов, покрывающая все возможные ошибки, будем называть системой тестов.
7. При испытаниях программы порядок применения тестов не безразличен и должен быть таким, чтобы облегчить локализацию одной ошибки, если в программе их несколько.
8. Желательно избегать окончательного тестирования программы автором ее разработки.

5.2. Виды тестирования

Учитывая большое разнообразие классов решаемых задач, для их тестирования используются несколько видов тестирования. Рассмотрим их.

Статическое тестирование предполагает поиск ошибок путем просмотра текста программы и выполнения ее «вручную», т.е. «в уме»

или на бумаге (Правила фиксации результатов при этом описываются в разделе «Правила записи трассировки»). Применяется в процессе разработки алгоритма.

Детерминированное тестирование заключается в подготовке контрольных примеров (тестов) на основе некоторой методики. Оно применяется в случае, когда для полноценных испытаний программы достаточно иметь ограниченное количество тестов. И при этом для каждого теста можно задать не только определенные константы в качестве входных данных, но и в качестве эталонного ответа. Такое тестирование называется «численным», хотя для многих задач значения могут быть не числовыми, а символьными. Данное тестирование может применяться как в процессе разработки алгоритма, так и в процессе испытаний и отладки на компьютере.

Стохастическое тестирование используется тогда, когда, в силу сложности проверяемого алгоритма, количество тестов, которое требуется разработать на основе любой методики, слишком велико. В этом случае для выработки (генерации) входных данных тестов используют программные датчики случайных чисел, а вместо эталонных ответов определяются только области данных, которым должны принадлежать эталонные ответы. Количество тестов выбирается на основе методов теории планирования экспериментов. Данный вид тестирования применяется при испытаниях программы на компьютере. В данном пособии стохастическое тестирование не рассматривается.

Следует отметить также, что имеются классы задач, в которых вычисления при исполнении теста могут быть чрезмерно трудоемкими, а иногда и непосильными, в первую очередь на этапе разработки алгоритма. В первую очередь это относится к решению физических и математических задач. Целью тестирования в таких случаях обычно является стремление установить факт, что некоторая последовательность действий в алгоритме обеспечивает корректную реализацию заданной системы уравнений. При таком тестировании эталонного ответа в традиционном «числовом» виде предложить невозможно, и эталон представляется в виде строки-формулы. Такое тестирование называют символьным. В данном пособии этот вид тестирования также не рассматривается.

5.3. Стратегии тестирования

Под стратегией тестирования мы будем понимать совокупность принципов, положенных в основу построения системы тестов. Рассмотрим две наиболее популярные стратегии: стратегию «черного ящика» и стратегию «белого ящика». Первая предполагает, что при разработке тестов известны только функции программы, но не известно, как они реализу-

ются. Иначе говоря, данная стратегия опирается на то, что разработчик тестов знает, ЧТО должна делать программа, но не знает КАК.

Вторая стратегия (стратегия «белого ящика») опирается на знание внутренней структуры (содержания действий) программы. При этом разработчик тестов знает, и ЧТО и КАК делает программа. И, следовательно, ее применяют, в первую очередь в процессе разработки алгоритма, а также при внесении изменений в готовую программу.

В рамках этих стратегий предложено несколько различных методов. Рассмотрим их.

5.3.1. Методы стратегии «черного ящика»

Тестирование в рамках данной стратегии предполагает управление по входным и выходным данным. Тесты разрабатываются на основе внешней спецификации. Никаких знаний о структуре программы не предполагается. Эта стратегия может быть использована на любом этапе разработки и при эксплуатации (использовании) программы. Отметим, что именно она используется при проведении приемо-сдаточных испытаний, т.е. при сдаче заказчику готовой программы.

Применяемые методы:

- метод эквивалентных разбиений
- метод граничных условий
- метод функциональных диаграмм
- метод, основанный на предположениях об ошибке

Метод эквивалентных разбиений основывается на разбиении области возможных значений входных данных на конечное число классов эквивалентности. В основе выделения класса лежит утверждение, что для любого значения внутри класса эквивалентности поведение программы идентично, и, следовательно, проверка программы для каждого класса может быть проведена всего одним тестом.

При таком подходе можно выделить два этапа разработки тестов: первый – выделение классов эквивалентности (не только для допустимых, но и для недопустимых входных данных) и затем разработка по одному тесту для каждого класса. Суть изложенного продемонстрируем на примере.

Пример. Пусть требуется разработать тесты для проверки следующих функций, реализуемых программой работы с базой данных:

- ввод с клавиатуры (1)
- сохранение в файле (2)

Выбранная функция задается пользователем номером, указанным в скобках.

Пусть структура записи в базе имеет следующий вид:

- фамилия (от 2 до 20 символов)

- возраст (от 0 до 100)

И пусть число записей в базе не должно быть больше 2000.
Примечание. В базе данных не может быть двух одинаковых записей.

Вначале выделим классы эквивалентности и представим результаты в таблице.

Входные условия	*Правильные классы*	*Неправильные классы*
Классы для кода операции		
Операция	Число: 1 или 2 (*соответствуют номерам существующих операций с базой*)	Любое другое число (*несуществующая операция*)
Классы для операции «Ввод»		
Фамилия	2 <= *длина* <= 20	*длина* < 2 *длина* > 20
Возраст	0 . . 100	*значение* < 0 *значение* > 100
Наличие в базе определенной фамилии	да / нет	–
Номер введенной записи	1, > 1 (до 2000)	–
Классы для операции «Сохранение в файле»		
Наличие записей в базе	да / нет	–
Наличие в базе одноименного файла	да / нет	–
Наличие свободного места на диске	да / нет	–

Теперь приступим к составлению тестов. Тесты будем составлять таким образом, чтобы каждому классу эквивалентности соответствовал, по крайней мере, один тест (входной набор).

Анализ таблицы показывает, что для проверки реакции программы на ввод номера операции требуется 2 теста, по одному для правильного и неправильного классов, например, ввести числа 2 и 5.

Для проверки операции «Ввод» требуется 10 тестов:

- по три теста (для правильного и неправильных классов) при вводе фамилии и возраста;

- по два теста для правильных классов для двух остальных проверяемых ситуаций (для случая «да» и «нет», и для случая «1» и «>1»)

Для проверки операции «Сохранение в файле» при аналогичном подходе нужно шесть тестов.

Метод граничных условий. Тесты в данном случае выбираются так, чтобы проверить ситуации, возникающие непосредственно **НА** границах**, ВЫШЕ** или **НИЖЕ** границ входных данных (а иногда и в выходных) для классов эквивалентности.

Необходимость применения данного метода станет очевидной, если проанализировать тесты, предложенные в методе эквивалентных классов для проверки реакции программы на корректное задание возраста. Там предложено разработать три теста: один – для правильного класса и два – для неправильных. Согласно этой рекомендации можно предложить, например, следующие входные данные для этих тестов: 10 – для правильного класса, –3 и 120 – для неправильных классов (при проверке введенного возраста). Очевидно, что эти тесты не позволят проверить, как ведет себя программа на границах между правильным и неправильными классами (при входных значениях 0 и 100). А поскольку текст программы неизвестен, то этих трех тестов становится недостаточным для получения уверенности, что программа ведет себя правильно при любом числе, предъявляемом на входе при вводе возраста. Метод граничных значений и используется для снятия этих ограничений.

При его применении, как правило, сначала подготавливают тесты по методу эквивалентных разбиений, а затем они дополняются тестами, построенными на основе знания границ правильных классов, если первоначального набора тестов недостаточно. Используемый метод определяет, сколько и какие тесты должны быть добавлены.

Итак, в отличие от метода эквивалентных разбиений в данном методе для любого класса эквивалентности значения входных данных выбираются таким образом, чтобы проверить тестами каждую границу класса эквивалентности.

Отсюда видно, что использование данного метода приводит к увеличению количества тестов по сравнению с предыдущим. Так, например, для проверки операции ввода фамилии для правильного класса нужен не один, а два теста, с фамилией длины 2 символа и длины ровно 20 символов.

Отметим, что при разработке тестов по методу граничных значений иногда учитываются не только входные условия, но и пространство результатов.

Метод функциональных диаграмм. К недостаткам рассмотренных выше методов относится то, что при их применении не предполагается проверять поведение программы при различных комбинациях входных данных. Метод функциональных диаграмм позволяет сделать это.

Для этого на основе внешней спецификации строятся функциональные диаграммы, связывающие причины и следствия, а затем по этим диаграммам строятся тесты. Для получения представления о сути метода рекомендуется обратиться к работам Г.Майерса [8].

Метод, основанный на предположениях об ошибке. Это метод чаще всего применяется в процессе отладки, т.е. в случае, когда выясняется, что программа ведет себя неправильно. При данном методе перечисляются возможные ошибки и ситуации, в результате которых могут эти ошибки появиться, и для них составляются тесты.

5.3.2. Методы стратегии «белого ящика»

В основу построения тестов положено знание структуры программы. Именно знание правил интерпретации структурных элементов «альтернативный выбор» и «цикл» определяет количество тестов, необходимых и достаточных для полноценной проверки программы. В рамках данной стратегии применяются следующие методы:

- покрытие операторов («все операторы»);
- покрытие решений («все пути»);
- комбинированное покрытие условий.

Сравним методы на примере фрагмента программы, приведенного ниже.

Пусть надо разработать тесты для проверки следующего фрагмента:

```
если (a>1) и (b=0) то
    x:=x/a
все
если (a=2) и (x>1) то
    x:=x+1
все
```

```
if (a>1) and (b=0) then
   x := x / a;
{end if}
if (a=2) and (x>1) then
   x := x + 1;
{end if}
```

Метод покрытия операторов.

Число тестов выбирается таким, чтобы каждый оператор был выполнен, по меньшей мере, один раз. В нашем случае достаточно одного теста:
Тест 1. {a=2, b=0, x=6} Эталон: x=4

Метод покрытия решений.

Число тестов выбирается таким, чтобы каждое решение было выполнено, по меньшей мере, один раз. Здесь под решением понимается конкретное значение условия в альтернативной или циклической структуре вне зависимости от сложности условия. В нашем случае требуется два теста, и их можно реализовать одним из двух вариантов:

- <u>Вариант 1</u>. Первый тест – для случая, когда оба решения: первое -

{(a>1) **и** (b=0)} и второе -{ (a=2) **и** (x>1) }, не выполняются, а второй – когда оба они выполняются;

- Вариант 2. Первый – для случая, когда первое решение{(a>1) **и** (b=0)} выполняется, а второе { (a=2) **и** (x>1) } не выполняется, а во втором случае – наоборот, не выполняется первое и выполняется второе решение.

Приведем тесты для первого варианта.

Тест 1. {a=2, b=0, x=3} Эталон: x=2,5 ⇨ Оба решения выполняются.
Тест 2. {a=3, b=1, x=1} Эталон: x=1 ⇨ Оба решения не выполняются.

Метод комбинированного покрытия условий.

Все возможные комбинации условий в каждом решении должны быть выполнены хотя бы один раз.

Приведем вначале возможные комбинации условий для нашего фрагмента:

первый оператор if	второй оператор if
1) a>1 **и** b=0	5) a=2 **и** x>1
2) a>1 **и** b<>0	6) a=2 **и** x<=1
3) a<=1 **и** b=0	7) a<>2 **и** x>1
4) a<=1 **и** b<>0	8) a<>2 **и** x<=1

Теперь можно предложить и тесты, в которых обеспечивается комбинированное покрытие приведенных условий. Анализ таблицы показывает, что нам можно ограничиться четырьмя тестами:

Тест 1. {a=2, b=0, x=4} Эталон: x=3
Тест 2. {a=2, b=1, x=4} Эталон: x=5
Тест 3. {a=1, b=0, x=2} Эталон: x=2
Тест 4. {a=1, b=1, x=1} Эталон: x=1

В заключение хотелось бы отметить, что и в случае, когда доступен текст программы, при разработке системы тестов удобно воспользоваться следующим приемом:

Как правило, сначала подготавливают тесты по стратегии «черного ящика», а затем, в случае, если первоначального набора тестов недостаточно, он дополняется тестами, построенными на основе знания логической структуры программы. Используемый метод определяет и то, сколько и какие тесты должны быть добавлены.

Детальный пример разработки тестов в процессе проектирования программы приведен в четвертой главе

5.4. Правила записи трассировки

Под трассировкой понимается выполнение алгоритма последовательно, операция за операцией, с фиксацией результатов каждой операции. Трассировка может выполняться вручную и на компьютере, используя

при этом имеющиеся программные средства отладки. При выполнении тестов «вручную» с фиксацией результатов на бумаге рекомендуется придерживаться следующих правил:

1. В качестве результатов операций ввода и присваивания записываются новые значения переменных (в виде тождеств).
2. В качестве результата операции вывода изображается текст, сформированный для выводимых значений.
3. Условие (в операциях ветвления и цикла) записывается в виде отношений, в которых все переменные заменены их текущими значениями. Условие помещается в фигурные скобки и затем записывается результат в форме «да/нет».
4. Трассировку следует начинать с первой операции исполнимой части алгоритма, а их результаты записываются последовательно, в порядке их выполнения.
5. Для всех, не детализированных в данном алгоритме, действий и подпрограмм результаты считать сформированными правильно.
6. Трассировка должна быть прекращена только по достижении строки «конец» алгоритма.
7. Только после этого надо приступить к анализу результатов алгоритма. Именно за счет этого появляется возможность увидеть такие результаты, которых программа не должна делать, но из-за ошибки в алгоритме делает.

В качестве примера приведем запись трассировки изображенного выше фрагмента из двух последовательных конструкций «если» для теста номер 1, построенного по методу комбинированного покрытия условий.

Дано: {a=2, b=0, x=4}

Требуется: {x=3}

Трассировка:

$\{ \{2>1\}_{да}$ **и** $\{0=0\}_{да} \}_{да}$

x = 2

$\{ \{2=2\}_{да}$ **и** $\{2>1\}_{да} \}_{да}$

x = 3

Конец.

Вывод: Полученный результат (x=3) совпадает с эталонным. Ошибки нет.

Вопросы для самоконтроля

1. Что такое «тест»? Каково его назначение?
2. Какой тест является хорошим и какой плохим?
3. Сколько тестов требуется для получения уверенности, что программа работает правильно?
4. Дайте качественное определение термина «система тестов».
5. Перечислите виды тестирования?
6. Какой вид тестирования более удобен на этапе разработки программы?
7. Что понимается под стратегией тестирования?
8. В чем отличия стратегий «черного ящика» и «белого ящика»?
9. Что понимается под классом эквивалентности?
10. Какие критерии используются при выделении классов эквивалентности?
11. В чем суть метода граничных условий?
12. Перечислите и вкратце охарактеризуйте методы стратегии «белого ящика».
13. В чем отличия методов покрытия решений и покрытия условий?
14. Что понимается под термином «трассировка»?
15. Перечислите правила записи трассировки?

6. ХАРАКТЕРИСТИКИ КАЧЕСТВА ПРОГРАММЫ

Разработка программы представляет собой комбинаторный процесс. В результате его может быть получен не один, а несколько вариантов программ, решающих одну и ту же задачу. И разные варианты будут отличаться по своим характеристикам. Рассмотрим характеристики качества программы на примере программы решения следующей задачи.

Задача: Определить, является ли заданное число простым. Напомним, что простым считается такое целое число, которое делится только само на себя и на единицу. Ниже приведен текст программы, призванной решить задачу.

```
Program Simple;
  var
     N : integer;
     I : integer;
     F : boolean;
  begin
     readln(N);
     F:= false;
     for I:=2 to N-1 do
        if (N mod I) = 0 then
          F:= true;
     if F then
        writeln('Число не является простым')
     else
        writeln('Число простое');
  end.
```

Основная идея алгоритма заключается в следующем: если число N не будет делиться без остатка ни на одно из чисел натурального ряда, меньшее чем N, то оно, согласно определению, будет простым.

Приведенная программа написана на языке Паскаль и имеет имя Simple.

В разделе описаний объявлены следующие переменные:

N – типа целых – исходное (проверяемое) число

I – типа целых – возможный делитель (параметр цикла)

F – логический (булевский) тип – признак «*делитель найден*».

Последовательность действий в программе следующая:

Вводим число N. Присваиваем признаку F значение “ложь”. В цикле просматриваем все возможные делители. Если остаток от деления равен нулю, устанавливаем признаку F значение «истина».

Наконец, анализируем значение признака F и выводим результат в виде строки текста.

Рассмотрим, отвечает ли наша программа критериям качества.

Корректность (правильность) - это строгое и полное соответствие конечных результатов выполнения программы требованиям постановки задачи.

Нашу программу можно считать правильной, поскольку результат ее выполнения соответствует условиям задачи определения принадлежности числа множеству простых чисел. Однако она не удовлетворяет многим другим критериям качества.

Надежность - это способность программы обеспечивать устойчивость функционирования при возникновении отклонений, вызванных ошибками входных данных, сбоями технических средств и обслуживания, а также способность программы сохранять работоспособность в заданных режимах и при заданных объемах обрабатываемой информации.

Наша программа не контролирует правильность ввода исходного числа: если пользователь введет отрицательное число, результат будет ошибочным.

Чтобы избавиться от этой ошибки, введем контроль введенного значения и завершение работы программы в случае ошибки. Текст программы с указанными изменениями теперь выглядит так:

```
Program Simple;   {Версия 2}
   label LabErr;
   var
      N : integer;
      I : integer;
      F : boolean;
   begin
      readln(N);
      if N<=0 then  goto  LabErr;
      F:= false;
      for I:=2 to N-1 do
         if (N mod I) = 0 then
         F:= true;
      if F then
      writeln('Число не является простым')
      else
      writeln('Число простое');
   LabErr:
   end.
```

Теперь программа будет функционировать надежнее.

Удобство применения – это способность программы обеспечивать простоту ввода исходных данных и получение результатов в наиболее понятной и наглядной форме.

Для этого программа должна:

- формировать запрос на ввод данных в понятной форме
- снабжать комментариями вывод результатов

- информировать пользователя об ошибках при задании входных данных и (при диалоговом режиме работы программы) дать ему возможность исправить ошибку.

Наша программа (ее вторая версия):

- не выводит запрос на ввод данных
- не информирует об ошибке (при вводе отрицательного числа)
- не позволяет исправить эту ошибку.

Исправим программу следующим образом:

Вместо оператора «**goto** LabErr» используем конструкцию «**repeat … until**», обеспечив повторение запроса на ввод данных до тех пор, пока они не будут введены правильно. Кроме этого дополнительно поместим операторы вывода сообщения о назначении программы.

Теперь текст программы примет следующий вид:

```
Program Simple;    {Версия 3}
   var
      N : integer;
      I : integer;
      F : boolean;
   begin
     writeln('Программа определения принадлежности');
     writeln('   числа множеству простых чисел');
     repeat
        write('Введите исходное натуральное число ');
        readln(N);
        if N<=0 then
        writeln('Число должно быть больше 0');
     until N>0;
     F:= false;
     for I:=2 to N-1 do
        if (N mod I) = 0 then
           F:= true;
     if F then
     writeln('Число не является простым')
     else
     writeln('Число простое');
   end.
```

Эффективность по быстродействию и по затратам памяти характеризуется зависимостью времени работы программы или затрат основной памяти от размера исходных данных. В этом контексте говорят о ***временнόй*** (ударение на «о») и ***емкостной*** сложности программы.

Поведение временнόй сложности в пределе при увеличении размера задачи называется асимптотической временнόй сложностью программы.

Зависимости могут быть различными, например:

- линейными, когда временнáя сложность линейно зависит от N (например, при вычислении суммы элементов массива)

- квадратичными, когда временна́я сложность имеет порядок N^2 (например, при сортировке массива)
- логарифмическими, когда время решения пропорционально $\log_2 N$ (например, при поиске в упорядоченном массива методом деления интервала пополам).

Асимптотическая временна́я сложность – важная характеристика алгоритма и программы, построенной на ее основе. Выбор более оптимального алгоритма, если это в принципе возможно, с меньшей асимптотической временно́й сложностью позволит значительно сократить время работы программы.

В нашей программе асимптотическая временна́я сложность – линейная (за счет использования цикла со счетчиком «**for** I:=2 **to** N-1 **do**». Однако можно заметить, что для поиска делителя достаточно просмотреть числа от 2 до корня квадратного от N, и в этом случае асимптотическая временна́я сложность программы уменьшится и станет пропорциональной корню квадратному от N. При этом вместо приведенного выше оператора цикла потребуется два:

```
imax:=round(sqrt(N));
for I:=2 to imax do
```

Кроме того, время работы программы можно сократить еще, если прекращать выполнение цикла, как только будет найден первый делитель. Для этого заменим оператор цикла со счетчиком (**for**) на оператор цикла с предусловием (**while**).

Текст новой (четвертой) версии программы примет следующий вид:

```
Program Simple; {Версия 4}
var
   N : integer;
   I : integer;
   F : boolean;
   imax : integer;
begin
   writeln(‘Программа определения принадлежности’);
   writeln(‘числа множеству простых чисел’);
   repeat
      write(‘Введите исходное натуральное число ’);
      readln(N);
      if N<=0 then
      writeln(‘Число должно быть больше 0’);
   until N>0;
   F:= false;
   imax:=round(sqrt(N));
   I:=2;
   while (not F) and (I<= round(sqrt(N))) do
   begin
      if (N mod I) = 0 then
         F:= true
      else I:=I+1;
   end;
   if F then
   writeln('Число не является простым')
   else
   writeln('Число простое');
end.
```

Эта программа будет работать значительно быстрее. А говорить о ее емкостной сложности нет смысла.

Удобство сопровождения - это способность программы обеспечивать простоту процесса поддержания программы (программного обеспечения) в работоспособном состоянии и простоту внесения в нее необходимых модификаций**.**

Это свойство обеспечивается правильным оформлением текста программы и наличием необходимой программной документации.

Для простоты сопровождения требуется:

- Структурированность программы
- Наличие комментариев в тексте программы
- Имена переменных должны нести смысловую нагрузку или соответствовать общепринятым обозначениям. (В нашем примере вместо безличного имени F для признака «делитель найден» лучше использовать более информативное имя, например, FlagFound).

Структурированность программы достигается за счет выполнения следующих требований к ее записи:

Программа составляется на базе ограниченного набора простых конструкций (следование, ветвление, цикл) без использования переходов с помощью оператора **goto.**

Конструкции записываются с отступами так, чтобы была видна структура программы.

В нашем случае программа структурирована только частично. В ней нарушены правила ступенчатой записи конструкции «если» в теле цикла *repeat . . until* и в конце программы.

Наличие комментариев обеспечивает так называемую самодокументированность программы. Для достижения этого рекомендуется снабдить комментариями все основные переменные и части программы.

Текст программы (версия 5), после внесения в него изменений, примет вид, удовлетворяющей рассмотренным выше критериям. *Он приведен на следующей странице.*

Мобильность универсальность) – это возможность перенесения программного обеспечения с одного типа ЭВМ на другой или с одной платформы (операционной среды) на другую.

Наша программа построена с использованием версии языка Паскаль, соответствующей стандарту ISO (Международной организации стандартов), и поэтому она может быть легко перенесена в другие операционные среды и на другие типы ЭВМ

Вопросы для самоконтроля

Перечислите характеристики качества программы

1. Какая характеристика программы обеспечивает достоверность ее результатов?
2. Как обеспечить удобство применения программы?
3. Что означает «емкостная сложность» программы, и для каких задач важно обеспечить хорошее значение ее?

```
{Определение принадлежности множеству простых чисел}
{Разработал Питеркин В.М.}
Program Simple;  {Версия 5}
var
   N : integer;   {исходное число}
   I : integer;   {очередной делитель}
   FlagFound : boolean;   {признак "делитель найден"}
   imax : integer;{максимальный делитель}
begin
   {ввод данных}
   writeln('Программа определения принадлежности');
   writeln('числа множеству простых чисел');
   {повторять, пока число не будет введено правильно}
   repeat
      write('Введите исходное натуральное число ');
      readln(N);
      if N<=0 then
         writeln('Число должно быть больше 0');
      {end if}
   until N>0;
   {вычисление результата}
   FlagFound:= false;
   imax:=round(sqrt(N));
   I:=2;
   {повторять, пока делитель не найден и/или }
   {не просмотрены все возможные делители    }
   while (not FlagFound) and (I<= round(sqrt(N))) do
   begin
     {если число делится без остатка - делитель найден}
      if (N mod I) = 0 then
         FlagFound:= true
      else
         { перейти к следующему возможному делителю}
         I:=I+1;
      {end if}
   end;
   {вывод результатов}
   if FlagFound then
      writeln('Число не является простым')
   else
     writeln('Число простое');
   {end if}
end.
```

Вопросы для самоконтроля (продолжение)

4. Почему для примера, рассмотренного выше, емкостная сложность не важна (не может быть улучшена)?
5. Как обеспечить удобство сопровождения программы? Для какой категории лиц наиболее важна эта характеристика программы?
6. Перечислите характеристики качества программы
7. Какая характеристика программы обеспечивает достоверность ее результатов?
8. Как обеспечить удобство применения программы?
9. Что означает «емкостная сложность» программы, и для каких задач важно обеспечить хорошее значение ее?
10. Почему для примера, рассмотренного выше, емкостная сложность не важна (не может быть улучшена)?
11. Как обеспечить удобство сопровождения программы? Для какой категории лиц наиболее важна эта характеристика программы?

7. ТИПОВЫЕ АЛГОРИТМЫ ОБРАБОТКИ МАССИВОВ

Анализ программ обработки массивов позволяет выделить несколько типовых (базовых) алгоритмов. Ниже приводится несколько часто встречающихся алгоритмов. Алгоритмы будут представлены в основном на языке «псевдоПаскаль», т.е. в таком виде, когда на Паскале записаны «инвариантные» части алгоритма, а на русском языке - та часть текста, которая зависит от специфики применения алгоритма.

А1. Ввод массива с клавиатуры

Данный алгоритм представлен на псевдокоде:

```
План ”Ввод-массива-вещественных-чисел”
Внутр. перем.:
    n, j : цел  {n – фактический объем массива}
    a[1..nmax] : вещ { nmax = 100 - константа}
Нач
    {ввод числа элементов массива}
    цикл
        вывод (’Введите число элементов ’)
        ввод (n)
        если (n<1) или (n>nmax) то
            вывод (’Ошибка’)
        все
    кцикл-до (n>=1) и (n<=nmax)
    {ввод элементов массива}
    цикл-для j от 1 до n
        вывод (’введите ’,j,’-й элемент ’)
        ввод (a[j])
    кцикл
Кон
```

Из текста исполнимой части можно видеть, что алгоритм сводится к последовательному выполнению двух действий: вводу числа элементов и вводу элементов. Логика первого действия построена таким образом, что оно будет завершено только в том случае, когда значение вводимого числа будет допустимым, т.е. принадлежать интервалу [1 .. nmax].

Примечание. Объявление внутренних переменных приведено только для того, чтобы при кодировании на Паскале использовать корректные средства этого языка. Однако приведенный алгоритм пригоден без переделки и для случая массива целых.

Исполнимая часть алгоритма приведена ниже в виде фрагмента на Паскале:

```
    {ввод числа элементов массива}
    repeat
       write ('Введите число элементов ');
       readln(n);
       if (n<1) or (n>nmax) then
          writeln('Ошибка');
       {if}
    until (n>=1) and (n>=nmax);
    {ввод элементов массива}
    for j:=1 to n do begin
       write('введите',j,'-й элемент');
       readln(a[j]);
    end; {for}
```

При этом предполагается, что в программе, где расположен этот фрагмент, объявлены следующие переменные:

```
Const
    nmax = 100; {максимальный объем массива}
Var
    n, j : integer; {n – фактический объем массива a}
                    {j – счетчик цикла}
    a : array [1..nmax] of real;
```

А2. Вывод одномерного массива целых по k чисел в строке

Идея алгоритма заключается в следующем: в теле цикла каждый раз выводится без смены строки один элемент, а строка меняется только после вывода элемента, номер которого кратен значению **k.**

Текст А2 на Паскале приведен ниже.

```
for j:=1 to n do begin  {повторить для каждого элемента}
   write (a[j]:7);  {напечатать текущий в 7-ми позициях}
   if (j mod k)=0 then  {если текущий k-й в строке то}
      writeln;          {сменить строку}
   {end if}
end; {for}
writeln;                {перейти на новую строку}
```

А3. Вывод элементов прямоугольной матрицы по строкам

Алгоритм приведен для случая, когда число столбцов матрицы позволяет напечатать строку матрицы в одной строке экрана (при выводе на экран монитора в текстовом режиме ширина строки равна 80 символам).

```
{Пусть матрица имеет n строк и m столбцов,           }
{ее элементы - вещественные числа,                    }
{а все m элементов помещаются в одной строке печати}
for i:=1 to n do begin
   for j:=1 to m do  {печатать все элементы j строки i}
      write (a[i,j]:8:2);{напечатать число в 8-ми позициях, }
                         {отобразив в нем 2 десятичные цифры}
   {end if}
   writeln;  {сменить строку печати перед выводом
              новой строки матрицы}
end;
```

А4. Сумма элементов массива

```
S := 0;
for j:=1 to n do
   S := S + a[j]; {Повторить для каждого элемента}
{end for}
```

А5. Поиск максимального / минимального элемента

Ниже будут приведены два варианта алгоритма – первый для случая, когда требуется получить величину искомого элемента, а второй для случая, когда надо найти индекс в массиве. При этом алгоритм 5а позволяет найти величину максимального элемента массива, а алгоритм 5б – номер минимального элемента.

Алгоритм 5а	Алгоритм 5б

`amax := a[1];`
`for j:=2 to n do`
`   if a[j]>amax then`
`      amax := a[j];`
`   {end if}`
`{end for}`
`{максимальное значение`
`     в переменной amax}`
	`jmin := 1;`
	`for j:=2 to n do`
	`   if a[j]<a[jmin] then`
	`      a[jmin] := a[j];`
	`   {end if}`
	`{end for}`
	`{индекс минимального элемента jmin,}`
	`{а само значение - в a[jmin]       }`

А6. Подсчет числа элементов массива, удовлетворяющих условию

```
np := 0;  { np : integer }
for j:=1 to n do
   if (a[j] удовлетворяет условию) then
      np := np + 1;
   {end if}
{end for}

{После завершения цикла переменная np содержит
 число элементов массива, удовлетворяющих условию}
```

В этом алгоритме надо указать то условие, которое необходимо в конкретных условиях. Так, например, при подсчете количества положительных чисел, это условие имеет вид «(a[j]>0)», а в случае подсчета количества простых чисел «Simple(a[j])», где Simple – имя функции, возвращающей значение TRUE. если a[j] простое число, и FALSE – в противоположном случае. Отметим, что функция Simple должна быть объявлена до первого вызова ее в операторе if.

В принципе проверку того, удовлетворяет ли текущий элемент требуемому условию, можно провести и в теле цикла, записав результат проверки в переменную логического типа, а ее значение проверить затем в операторе if.

А7. Выбор элементов по условию

Данный алгоритм в приведенном ниже примере используется при формировании нового массива, в который включаются те элементы исходного массива, которые удовлетворяют некоторому условию.

```
{Пусть a[1..nA] - исходный массив }
{      c[1..nC] - новый массив    }
{      nC – фактическое число элементов в новом масси-
ве}
{---------------------------------}
  nC := 0;
  for j:=1 to n do {просмотреть каждый элемент в 'a'}
     if (a[j] удовлетворяет условию) then
     begin
        nC := nC + 1;  {}
        c[nC] := a[j]; {добавить новый элемент в 'c'}
     end; {if}
  {end for}
```

A8. Проверка выполнения некоторого условия

В приведенном ниже фрагменте производится проверка факта существования в заданном массиве хотя бы одного элемента, удовлетворяющего некоторому условию.

Просмотр элементов в алгоритме производится с использованием конструкции «цикл-пока». Это объясняется тем, что просмотр надо выполнить не менее одного раза и можно прекращать сразу после того, как будет найден элемент, удовлетворяющий условию. Однако поскольку возможна ситуация, что в массиве вообще нет элементов, удовлетворяющих условию, цикл надо закончить либо когда найден первый элемент, удовлетворяющий условию, либо когда просмотрены все элементы массива, но в массиве нет ни одного элемента, удовлетворяющего условию. Для того чтобы своевременно закончить просмотр элементов и, после выхода из цикла установить, найден или нет требуемый элемент, в программе введена переменная булевского типа FlagFound, выполняющая функцию признака «найден/нет».

```
FlagFound := FALSE; {не найден пока}
j := 1;
while (j<=n) and (not FlagFound) do
   if (a[j] удовлетворяет условию) then
      FlagFound := TRUE {элемент найден}
   else
      j := j + 1;
   {end if}
{end while}
{-------------------------------------------}
{если после выхода из цикла FlagFound=TRUE,
          элемент найден}
```

Переменные типа FlagFound очень активно применяются в случаях, когда требуется организовать циклическую обработку с неопределенным числом повторений тела цикла, и при условии, когда максимально возможное количество циклов ограниченно. В первую очередь это относится к ряду алгоритмов обработки элементов массива.

A9. Поиск номера элемента, удовлетворяющего условию

Для решения данной задачи можно воспользоваться алгоритмом A8, вернув результат в переменной `j`, выполняющей функцию счетчика цикла и индекса элемента массива одновременно, и связав факт того, найден элемент или нет, со значением переменной `FlagFound`.

A10. Вставить элемент Z в позицию j массива чисел

Для решения данной задачи надо освободить позицию j массива, сдвинув вправо (скопировать) все элементы, начиная с указанной позиции. А для этого сдвиг надо начинать с самого правого элемента:

```
for k:=n downto j do {сдвиг вправо}
   a[k+1] := a[k];
{end for}
a[j] := z;   {запись нового элемента}
n := n + 1;
```

А11. Удалить из массива элемент, занимающий позицию j

Чтобы решить данную задачу, надо сдвинуть влево (скопировать) все элементы, располагающиеся справа от удаляемого элемента, начиная с элемента с номером j+1.

```
for k:=j+1 to n do
   a[j] := a[k+1]};
{end for}
n := n - 1;
```

Примеры использования типовых алгоритмов

Пример 1. Сформировать массив ”С”, включив в него четные элементы массива ”А”, которые присутствуют в массиве ”В”.

Вначале запишем укрупненный алгоритм:

```
Алг ”Выбор из массива”
нач
    ввод-массивов-А-и-В
    формирование-массива-С
    вывод-массива-С
кон
```

Анализируя этот алгоритм мы видим, что ”*ввод-массивов-А-и-В*” можно реализовать, применив дважды алгоритм А1 (поменяв при этом имена переменных и скорректировав тексты сообщений на экран), а ”*вывод-массива-С*” несложно обеспечить, воспользовавшись алгоритмом А2. Нам остается только разработать алгоритм ”*формирование-массива-С*”.

Приведем план решения этой задачи:

```
План "формирование-массива-C"
нач
   nC := 0;  {массив C пуст}
   for i:=1 to nA do  {повторить для каждого
                        элемента массива A}
      if (A[i] четное) then begin
         искать-A[i]-в-массиве-B  {установить FlagFound}
         if FlagFound then
            добавить-A[i]-в-массив-C
         {end if}
      end; {if}
   {end for}
кон
```

Теперь нужно раскрыть планы действий "*искать-A[i]-в-массиве-B* " и "*добавить-A[i]-в-массив-C*".

Первый план – это модификация алгоритма A8:

```
FlagFound := FALSE; {не найден пока}
j := 1;
while (j<=nB) and (not FlagFound) do
   if (A[i] = B[j]) then
      FlagFound := TRUE {значение A[i] имеется в массиве B}
   else
      j := j + 1;
   {end if}
{end while}
```

Второй план (аналог телу цикла в алгоритме A7):

```
План "добавить-A[i]-в-массив-C"
нач
   nC := nC + 1;
   C[nC] := A[i];
кон
```

И, наконец, выражение (*A[i] четное*) можно реализовать, проверив остаток от целочисленного деления числа `A[i]` на 2:

```
      (A[i] mod 2) = 0
```

Задача разработки отдельных фрагментов решена, и теперь остается только получить итоговый (детальный) алгоритм, выполнив подстановку отдельных планов в укрупненный алгоритм. Эту работу читателю предлагается выполнить самостоятельно.

Пример 2. "Найти значение элемента, встречающегося в массиве наибольшее количество раз".

Вначале представим укрупненный алгоритм:

```
План
нач
   ввод-массива
   вычисление-числа-повторений
   вывод-результата
кон
```

При решении этой задачи можно воспользоваться алгоритмами ввода массива, выбора максимального числа и алгоритма подсчета числа элементов, удовлетворяющих условию.

Планы ”*ввод-массива*” и ”*вывод-результата*” мы рассматривать не будем. Сосредоточимся на плане ”*вычисление-числа-повторений*”. При этом текст запишем на псевдоПаскале. Это означает, что те операции алгоритма, для которых имеется однозначный аналог на Паскале, будем записывать полностью по правилам языка Паскаль, а остальные на языке с русской лексикой. Итак, данный план будет иметь следующий вид:

```
План ”вычисление-числа-повторений”
                             var
   MaxCount : integer; {макс. число повторений}
   Count : integer; {число повторений текущего элемента}
   Elem : integer; {величина элемента, встретившегося чаще всего}
   A:array[1..10] of integer;
   n : integer;   {число элементов в массиве A}
нач
   MaxCount := 0;
   for j:=1 to n do
   begin
      подсчет-числа-повторений-A[j]-в-массива-A
                             {результат записать в Count}
      if Count>MaxCount then
      begin
         MaxCount := Count;
         Elem := A[j];
      end; {if}
   end; {for}
кон
```

Раскроем план для подсчета числа повторений A[j] в массиве A (это модификация алгоритма A6):

```
План "подсчет-числа-повторений-A[j]-в-массива-A"
нач
   Count := 0;
   for i:=1 to n do
      if A[j]=A[i] then
         Count := Count +1;
      {end if}
   {end for}
кон
```

Детальный алгоритм решения задачи читателю предлагается выполнить самостоятельно.

Пример 3. В матрице A[1..n, 1..m] поменять местами первый и третий отрицательные элементы, встретившиеся при просмотре матрицы по строкам слева направо и сверху вниз.

Разработаем только алгоритм, реализующий основную цель задания, предоставив читателю самостоятельно написать фрагменты, обеспечивающие ввод исходных данных и вывод результата.

Вначале приведем описание переменных:

```
Const
   MaxCols = 6;
   MaxRows = 6;
Var
   A : array[1.. MaxRows, 1.. MaxCols] of real;
   i, j : integer; {индексы текущего элемента при просмотре}
   i1,j1: integer; {индексы 1-го отрицательного элемента}
   i3,j3: integer; {индексы 3-го отрицательного элемента}
   CountNeg : integer; {количество отрицательных элементов}
   FlagFound3 : boolean; {признак того,
                           что найден 3-й элемент}
   Temp : real; {вспомогательная переменная для обмена}
```

Теперь можно привести укрупненный алгоритм.

```
CountNeg := 0;
FlagFound3 := FALSE;
i := 1; {для всех строк начиная с 1-й}
while (i<=n) and (not FlagFound3) do
begin
   j := 1; {для всех столбцов начиная с 1-го}
   while (j<=m) and (not FlagFound3) do
   begin
      обработать-элемент-A[i,j]
      поменять-местами-элементы-1-и-3
      j := j + 1; {увеличить номер столбца}
   end;
   i := i + 1; { увеличить номер строки}
end;
```

В этом алгоритме предполагается, что оба цикла могут закончиться в случае, когда найден третий отрицательный элемент и в этот момент будет произведена перестановка его и первого элемента. Если трех элементов в массиве нет, циклы закончатся после просмотра всех элементов матрицы. Перестановка элементов будет выполняться в алгоритме "*обработать-элемент-A[i,j]*".

Приведем вначале алгоритм перестановки:

```
if FlagFound3 then begin
   Temp := a[i1, j1];
   a[i1, j1] := a[i3, j3];
   a[i3, j3] := Temp;
end;
```

И, наконец, приведем алгоритм "*обработать-элемент-A[i,j]*":

```
План "обработать-элемент-A[i,j]"
нач
   if A[i,j]<0 then
   begin
      CountNeg := CountNeg + 1;
      if CountNeg=1 then begin
         i1 := i;
         j1 := j;
         end
      else if CountNeg=3 then begin
         i3 := i;
         j3 := j;
         FlagFound3 := TRUE;
      end; {if}
   end; {if}
кон
```

Теперь читатель может самостоятельно построить детальный алгоритм, обеспечивающий полное решение поставленной задачи.

8. ОБРАБОТКА СИМВОЛЬНОЙ ИНФОРМАЦИИ

Символьная информация может быть представлена в программе либо с использованием типа «символ» (char), либо с использованием типа «строка» (string).

В первом случае текст можно рассматривать как массив, элементами которого являются одиночные символы. Во втором, текст представляется как единое целое – строка (сцепленная последовательность символов). Для обработки текста, представленного как массив символов, полностью подходят алгоритмы обработки массивов, рассмотренные ранее. Ниже рассматриваются средства языка, которые могут быть использованы при обработке текста, представленного в формате **string**.

8.1. Обработка строк

Строка в Паскале представляет собой цепочку символов, ASCII-код каждого из которых располагается в отдельном байте. Длина строки - число символов в ней. Для того чтобы ограничить строку по длине, в языках программирования применяют различные способы.

В языке Турбо Паскаль строка представляется массивом следующего вида:

Номер байта	0	1	2	.	.	.	k
Содержимое	k	c_1	c_2	.	.	.	c_k

Здесь байт с номером 0 содержит беззнаковое целое, характеризующее текущее число символов (c) в строке (т.е. длину строки). Считается, что символы строки всегда нумеруются от 1. Отсюда следует что, если строка имеет длину k символов, то это соответствует массиву с граничной парой индексов [0..k]. При этом символы строки располагаются в позициях от 1-й до k-той, а в элементе с индексом 0 записана длина строки. И доступ к символам строки, как элементам массива, разрешен только с первого символа.

Следовательно, длина строки может принимать значения из диапазона от 0 до 255. Отсюда можно сделать два вывода:

- минимальная длина строки равна нулю (это пустая строка и занимает она один байт с индексом 0);
- максимальная длина строки – 255.

Для описания строки предусмотрен специальный тип – string. И при этом возможны два варианта объявления строки. Продемонстрируем это на примерах:

```
var
   city:string; { строка, произвольной длины }
   name:string[20]; {строка, содержащая не более 20 символов }
```

Отметим, что для второго варианта объявления переменной фактическая длина строки может иметь длину от нуля до 20.

Строковые константы заключаются в апострофы:

'Москва', 'Тверь'

Определены следующие **операции со строками**:

- операции отношения: =, <>, <,>, <=, >=
- операция конкатенации: + (это не арифметическая операция, а операция «склеивания» двух строк, в результате которой получается новая строка суммарной длины).

Поскольку строку можно рассматривать как массив символов, при обработке символьной информации также можно использовать алгоритмы обработки массивов (с учетом ограничений на допустимые операции).

Кроме этого, для обработки символьной информации имеются следующие **процедуры и функции**:

Функция «длина строки» – число символов в ней

```
function Length(s:string):integer;
```

Выделение подстроки:

```
function Copy (S:string,NB,Len:integer):string;
```

Вернуть новую строку, выделив Len символов из строки S, начиная с символа номером NB.

Пример:

```
var
    Name,FullName:string;
begin
    FullName='John Brown';
    Name:=copy(FullName,6,2);
    {Результат : в переменной Name записано 'Br'}
```

Функция поиска подстроки в строке:

```
function Pos (SubStr,Str:string):integer;
```

Вернуть номер первого символа SubStr в строке Str, если строка найдена, или ноль в противном случае.

Пример:

```
FullName='John Brown';
P:=Pos('B',FullName)  {=6 (SubStr - подстрока; Str - строка)}
       'N'            =4
       'WN'           =9
       'BR'           =6
       'AR'           =0 {подстрока не найдена}
```

Процедура вставки подстроки в строку:

```
procedure Insert (SubStr:string;var Str:string; I:integer);
```

Пример:

```
{Вставить строку 'Peter' начиная с позиции номер 6 }
{          в строку FullName                        }
Insert('Peter ', FullName,6);
{Результат: 'John Peter Brown'}
```

Процедура удаления подстроки:

```
procedure Delete (var S:string;NBy,Len:integer)
```

```
{исходная строка: 'John Brown'}
Delete(FullName,3,2);
{результат: 'Jo Brown'}
```

Из строки S удалить Len символов, начиная с позиции NBy.

Процедура преобразования числового значения (V) в строку:

```
procedure Str(V:<числовой тип>;S:string);
```

Здесь V может быть любого разрешенного в языке числового типа, как целого, так и вещественного.

```
Str(25,S); {Результат: S='25'}
x:=25e2;
Str(x:10:2,S); { Результат: '   2500.00'}
```

После числового значения могут быть указаны параметры, определяющие, в каком формате будет сформирована строка. Эти параметры полностью совпадают с параметрами, используемыми при форматном выводе. В приведенном примере значение переменной X будет сформировано с использованием формата «:10:2».

Процедура преобразования строки в числовое значение

```
Val(s:string; v: <числовой тип>; code:integer);
```

Здесь переменная code указывает, успешно ли завершилось преобразование текстовой строки S в число v:

```
code  =  0     {нет ошибки}
code  <> 0     {недопустимый символ в строке}
```

И если code не равно 0, значение переменной **v** не изменилось (или не определено).

Повторим, что кроме рассмотренных процедур и функций, при обработке символьной информации можно использовать алгоритмы обработки массивов (с учетом ограничений на допустимые операции).

Примеры

Пример 1. Удалить все символы в начале и в конце строки

Идея алгоритма заключается в следующем: Вначале организуется цикл, в котором, если первый символ является пробелом, он удаляется. Условие выхода из цикла – первый символ – не пробел. Затем организуется цикл, в котором, аналогично изложенному выше, удаляется один символ в конце строки, если он является пробелом.

В первом цикле для удаления символа используется функция выделения подстроки (`Copy`), а во втором процедура удаления подстроки (`Delete`).

```
var
   st:string;
begin
   st:='   Иванов Сергей   ';
   {Удалить все пробелы в начале строки}
   while st[1]=' ' do  {Первый символ строки - пробел}
      st:=Copy(st, 2, Length(st)-1);
   {end while}
   writeln('<',st,'>');
   {Удалить все пробелы в конце строки}
   while st[Length(st)]=' ' do {Последний символ - пробел}
      Delete(st, Length(st), 1);
   {end while}
   writeln('<',st,'>');
end.
{Результат первого цикла:
     <Иванов Сергей   >}
{ Результат второго цикла:
     <Иванов Сергей>}
```

Пример 2. Подсчитать, сколько раз входит в исходную строку символ ’а’.

В данном случае используется метод, полностью совпадающий с методом, изложенным для обработки массива чисел.

```
{Подсчитать количество вхождений символа "а" в строку
  Использован алгоритм A6 обработки массивов}
var
   st:string;
   i,k:integer;
begin
   st:='Исходная строка содержит несколько букв <а>';
   k := 0;
   for i:=1 to Length(st) do
      if st[i]='а' then
         k:=k+1;
      {end if}
   {end for}
   {Результат : k = 3}
end.
```

Эту задачу можно решить и используя функцию Pos:

```
{Подсчитать количество вхождений символа "а" в строку}
var
   st:string; {Исходная строка}
   stwork:string; {Рабочая переменная}
   p:integer; {Позиция первого вхождения символа в строку}
   k:integer; {Количество найденных символов}
begin
   st:='Исходная строка содержит несколько букв <a>';
   k := 0;
   stwork:=st;
   p:=Pos('a',stwork);
   while p>0 do begin
      k:=k+1;
      stwork:=Copy(stwork,p+1,255);
      {Скопировано будет не 255 символов,
       а оставшая часть строки, начиная с позиции <p+1>}
      p:=Pos('a',stwork);
   end;
   {Результат : k=3 }
end.
```

В данном случае используется следующий подход: Исходная строка копируется в новую. Далее организуется цикл, в котором ищется позиция вхождения искомого символа в новую строку. Если символ найден (P>0), в счетчик (переменная **k**) добавляется 1, после чего из строки удаляется подстрока, начиная с позиции 1 (от начала строки) до позиции, содержащей найденный символ. После этого цикл повторяется для оставшейся части строки. Процесс завершается, когда в оставшейся части строки больше нет искомого символа (когда P=0).

Пример 3. Сформировать новую строку из исходной, оставив в ней строго по одному пробелу во всех местах, где имеются более одного пробела подряд.

Идея метода: Организуется циклический просмотр исходной строки по одному символу. Если очередной символ является пробелом, это фиксируется в переменной **blank**. И если при просмотре следующего символа выяснится, что он также является пробелом, этот символ не вставляется во вновь формируемую строку. Во всех остальных случаях просматриваемый символ исходной строки копируется в новую строку.

Текст программы, реализующей этот алгоритм:

```
{Сформировать новую строку, удалив лишние пробелы
    в исходной строке}
var
   st:string; {Исходная строка}
   stnew:string;  {Формируемая строка}
   i:integer;
   blank:boolean; {true - если пробел первый,
                   false - если пробел не первый}
begin
   st:='  Иванов   Сергей   ';
   blank:=false;
   stnew := ''; {Строка пустая}
   for i:=1 to Length(st) do
      if st[i]<>' ' then begin
         stnew:=stnew+st[i];
         blank:=false;
      end
      else if (not blank) then begin
         blank:=true;  {первый пробел}
         stnew:=stnew+st[i];
      end;
   {end for}
   writeln('<',stnew,'>');
end.
{Результат :  < Иванов Сергей >}
```

Отметим, что задачу удаления повторяющихся символов «пробел» можно решить и иначе, не вводя новой строки. Обнаружив, что после первого пробела имеется еще один пробел, его можно удалить, используя процедуру Delete.

Для закрепления изложенного рекомендуется обратиться к электронному задачнику ’Тестер’.

8.2. Особенности ввода информации строкового вида

Напомним, что переменная типа string может быть объявлена как произвольной, так и ограниченной длины. Эти различия необходимо учитывать при вводе информации.

- Если в списке ввода имеется одна переменная типа string, все символы, набранные на клавиатуре до нажатия клавиши Enter, будут записаны в эту переменную,
- Если в списке ввода имеется несколько переменных, то при вводе в переменную типа string, все символы до конца строки присваиваются этой переменной. И если указанная переменная не последняя в списке ввода, операция ввода не завершается, хотя на экран ничего не выводится. Это приводит к необходимости вводить следующую строку, из которой будет извлекаться информация для оставшихся переменных.

- Если в списке ввода имеется переменная, объявленная как строка ограниченной длины, из строки, вводимой с клавиатуры, извлекается количество символов, не превышающее того, сколько указано при ее объявлении. А оставшаяся часть набранной строки может быть использована для присвоения значений переменной, записанной в списке ввода.

Проиллюстрирует сказанное примерами.

Пусть в программе объявленные следующие переменные

```
var
   st : string;
   st1 : string[11};
   n : integer;
```

И пусть в клавиатуры вводится строка

`Иванов А.В. 124`<*Enter*>

Результаты ввода для различных операторов приведены в таблице:

№№	Оператор ввода	Результат его выполнения
1	`readln(st);`	st = ’`Иванов А.В. 124`’
2	`readln(st,n);`	st = ’`Иванов А.В. 124`’ Операция не завершена и ожидается ввод с клавиатуры числа для переменной ***n*** из следующей строки
3	`readln(st1);`	st1 = ’`Иванов А.В.`’
4	`readln(st1,n);`	st1 = ’`Иванов А.В.`’ n = `124`

Задача корректного ввода последовательности «строка и затем число» может быть решена следующим образом:

```
{Дополнительно объявим переменные:
var
    txt : string;
    i, error : integer;   }
Последовательность операторов:
   readln (txt);
   i := 1;
   while not (txt[i] in ['0'..'9']) do
      i := i + 1;
   {end while}
   st : = copy(txt,1,i-1);
   Delete(txt,1,i);
   Val(st,n,error);
Если error=0, в переменной n будет занесено значение 124, а в переменной st –
значение 'Иванов А.В.'
```

В операторе while здесь используется новый тип данных – множество. Поскольку этот тип данных будет рассматриваться позже, поясним смысл выражения «**not** (txt[i] **in** ['0'..'9'])».

Здесь

- ['0'..'9'] - множество символов, изображающих цифры;
- **in** – операция принадлежности символа txt[i] указанному множеству, которая принимает значение true, если проверяемый символ – цифра и false в противном случае;
- **not** – операция логического отрицания.

Отсюда, выражение «**not** (txt[i] **in** ['0'..'9'])» принимает значение true, если символ – не цифра.

Вопросы для самоконтроля

1. Что представляет собой тип данных **string**?
2. Какими способами можно объявить переменную строкового типа?
3. В каком диапазоне может изменяться длина строки в Турбо-Паскале?
4. Каким образом можно рассматривать строку с позиций обработки?
5. Сколько байтов в памяти программы занимает величина строкового типа?
6. Какая функция позволяет получить значение длины строки?
7. Какие операции (функции) над строками имеются в Турбо-Паскале?
8. В чем особенности ввода информации строкового типа из текстового файла или с клавиатуры?

9. ТИПЫ ДАННЫХ, ЗАДАВАЕМЫХ ПОЛЬЗОВАТЕЛЕМ

Кроме рассмотренных ранее предопределенных типов данных программист может использовать и данные, тип которых он может задать самостоятельно в соответствии с особенностями решаемой задачи. Речь идет о множествах и записях. Рассмотрим их.

9.1. Множества

Термин "множество" в программировании используется аналогично его математическому пониманию. Отличие состоит в том, что в Турбо Паскале множества могут включать элементы только порядковых (перечислимых) типов. Элементы какого-либо конкретного множества (переменной или типизированной константы) должны принадлежать одному типу, который называется *базовым типом*. Максимальное количество значений базового типа множества называется его *мощностью*.

В Турбо Паскале в качестве базовых могут использоваться порядковые типы, мощность которых не превышает 256-ти значений. Кроме того, порядковые значения верхней и нижней границы базового типа не должны выходить за пределы диапазона от 0 до 255. Поэтому, в качестве базовых типов множеств не могут использоваться Shortint, Integer, Longint, Word.

Операции, допустимые для работы с множествами приведены в таблице:

Обозначение в Паскале	Действие	Тип результата
*	Пересечение	Множество
+	Объединение	Множество
–	Разность	Множество
in	Принадлежность элемента множеству	boolean
<=	Является подмножеством	boolean
>=	Включает подмножество	boolean

Более подробную информацию о множествах в Паскале можно найти в рекомендованной литературе.

Множества представляют собой гибкий и наглядный механизм для решения многих задач. В частности, их удобно использовать при обработке символьной информации.

В качестве примера рассмотрим следующую задачу: Подсчитать в строке отдельно количество цифр и латинских букв.

```
Program TestSets;
Type
   CharSet = set of Char;
Const
   Digits : Charset = ['0'.. '9'];
   Letters: CharSet = ['a'..'z','A'..'Z'];
var
   CountDig,       {количество цифр}
   CountLet : word;{количество букв}
   st : string;
   i : integer;
begin
   st := 'Apache on Win32 has not yet been optimized';
   for i:=1 to Length(st) do
      if (st[i] in Digits) then
         CountDig := CountDig + 1
      else if (st[i] in Letters) then
         CountLet := CountLet + 1;
      {end if}
   {end for}
   {Результат : CountDig = 2
                CountLet = 33}
end.
```

Используемое в программе множество при необходимости можно получить из двух других множеств операцией объединения (сложения):

```
Letters := SmallLetters + BigLetters;
```

Возможно также и обратное действие: требуемое множество получить операцией разности множеств:

```
SmallLetters := Letters - ['A'..'Z'];
BigLetters := Letters - ['a'..'z'];
```

9.2. Записи

Запись (в ряде языков программирования ее называют структурой), в отличие от массивов и множеств, является составной структурой данных. Если отдельно взятые массив или множество всегда включают элементы одинакового типа, то записи могут объединять в единое целое любое количество структур данных разных типов: простых переменных, массивов, множеств и других записей.

В языке Турбо-Паскаль различают фиксированные (обычные) и вариантные записи. В данной главе мы рассмотрим только фиксированные записи. После получения опыта их использования, читатель может самостоятельно ознакомиться по литературе и с вариантными записями.

Обычная фиксированная запись состоит из одного или нескольких полей, для каждого из которых при объявлении (в секции **Type)** указывается имя (идентификатор) и тип.

В качестве примера приведем запись, описывающую сведения о студенте.

```
type
     TStudent:=record {запись}
          {Список полей:}
           Name:string[20] {имя и тип поля}
           Age:integer;{возраст от 1 до 250}
           Sex:Char;  {'м'/'ж'}
           Phone:string[9];
           MathAnal : byte; {Мат. анализ}
           LinAlg   : byte; {Лин. алгебра}
           Prog     : byte; {Программирование}
           Phys     : byte; {Физика}
     end;
```

В приведенной записи есть группа полей, несущих одинаковую смысловую нагрузку (MatAnal, LinAlg, Prog, Phys). С точки зрения удобства работы с текстом программы и удобства выполнения групповых операций, такие поля целесообразно объединить в отдельную структуру данных типа record.

В результате тип TStudent примет такой вид:

```
type
     TStudent:=record {запись}
          {Список полей:}
           Name:string[20] {имя и тип поля}
           Age:integer;{возраст от 1 до 250}
           Sex:Char;  {'м'/'ж'}
           Phone:string[9];
           Marks = record
              MathAnal : byte; {Мат. анализ}
              LinAlg   : byte; {Лин алгебра}
              Prog     : byte; {Программирование}
              Phys     : byte; {Физика}
           end;
      end;
```

А еще лучше сделать это следующим образом:

```
type
     TMarks = record
           MathAnal : byte; {Мат. анализ}
           LinAlg   : byte; {Лин алгебра}
           Prog     : byte; {Программирование}
           Phys     : byte; {Физика}
     end;
     TStudent:=record {запись}
           Name:string[20] {имя и тип поля}
           Age:integer;{возраст от 1 до 250}
           Sex:Char;  {'м'/'ж'}
           Phone:string[9];
           Marks : TMarks; {оценки за семестр}
     end;
```

В программе обращение к полям записей выполняется с помощью квалифицируемых (уточненных) идентификаторов, в которых указывается вся цепочка имен от идентификатора переменной типа record до идентификатора требуемого поля. Имена полей в такой записи разделяются точками.

Покажем это на примере, в котором используется приведенное выше объявление записи TStudent.

```
Type
    TGrouup = array[1..25] of TStudent;
var
   Group : TGroup;
begin
   . . .
   Group[1].Name := 'Андреев А.Б.';
   Group[1].Age := 18;
   Group[1].Sex := 'м';
   Group[1].Phone := ''; {Пустая строка}
   Group[1].Marks.Prog := 4; {Присвоить значение полю
               Prog, входящему в состав структуры Marks,
               в свою очередь входящую в состав структуры
               типа Tstudent, являющейся 1-м элементом
               массива Group}
   . . .
```

9.3. Оператор with

Для упрощения работы с записями и придания тексту программы большей наглядности в Турбо Паскале имеется специальный оператор присоединения with. С использованием этого оператора приведенный выше фрагмент присвоения значений полям первого элемента массива Group будет иметь следующий вид:

```
with Group[1] do begin
   Name := 'Андреев А.Б.';
   Age := 18;
   Sex := 'м';
   Phone := ''; {Пустая строка}
   Marks.Prog := 4;
end;
```

При заполнении информацией структур данных типа record необходимо помнить, что с клавиатуры и из текстовых файлов допускается вводить данные только некоторых стандартных типов данных. Поэтому, в операторах Read и Readln могут располагаться только идентификаторы самых внутренних полей, которые имеют допустимые для ввода типы, например:

```
readln(Group[i].Name, Group[i].Marks.Prog);
или
with Group[i] do
    readln(Name, Marks.Prog);
```

К переменным типа «запись» могут в особых случаях применяться и групповые операции присваивания, когда одновременно присваиваются новые значения всем внутренним полям. Это, в частности, имеет место при работе с типизированными файлами (будет рассматриваться отдельно).

Пример использования переменной типа «запись» будет приведен в разделе «Динамические структуры данных».

Вопросы для самоконтроля

1. Как задать множество?
2. Каковы особенности работы с множествами?
3. Что такое запись?
4. Для чего используются уточненные идентификаторы?
5. Зачем используется оператор присоединения with ?
6. Каковы особенности ввода с клавиатуры значений для полей записи?

10. ФАЙЛЫ

10.1. Введение

Практические все рассмотренные ранее средства языка программирования Паскаль не зависят от того, на компьютере какой архитектуры и в среде какой операционной системы будет выполняться программа. Несколько иначе обстоит дело в отношении операций ввода и вывода на внешние устройства. Указанные операции по-разному выполняются на компьютерах различающихся архитектур и под управлением разных операционных систем. Чтобы освободить программиста от необходимости детального учета указанных особенностей, и введено понятие файла. А в набор средств языка программирования введен ряд подпрограмм работы с файлами, имена и функциональное назначение которых в нем стандартизованы, и, следовательно, не зависят ни от архитектуры компьютера, ни от типа операционной системы.

Тип данных ***файл*** в программе используется в качестве логического представителя некоторого набора данных, размещенного на внешнем устройстве. Какое это устройство, программист может задать либо при написании текста программы, либо в процессе её выполнения, указав имя файла по правилам файловой системы той операционной системы, под управлением которой будет исполняться программа.

Примечание. В данном пособии указанные действия будут записываться по правилам операционной системы MS DOS.

Итак, с одной стороны, файл – это именованная область внешней памяти, содержащая какую-либо информацию. В таком понимании это – физический файл, то есть существующий физически на некотором материальном носителе информации и доступный из программы с одного из внешних устройств компьютера: магнитного или оптического дисковода, клавиатуры и ряда других.

Структура физического файла представляет собой простую последовательность байтов информации:

байт	байт	байт	. . .	. . .	байт	байт

С другой стороны, файл – это одна из многих структур данных, используемых в программировании. В таком понимании речь идет о логическом файле, то есть существующим только в нашем представлении с позиций программы. В программе файл представляется файловой переменной определенного типа.

Структура логического файла – это способ восприятия файла в программе. В этом смысле файл рассматривается как последовательность элементов определенного для него типа данных. Элементом файла мо-

жет быть один или несколько байтов, целое или вещественное число, массив чисел, запись, символ или строка текста и другие. Ниже приведены примеры, иллюстрирующие сказанное.

file of byte (файл, состоящий из байтов)

байт	байт	байт	. . .	. . .	байт	Eof

file of integer (файл, состоящий из чисел типа «целые» во внутримашинных кодах)

целое со знаком	целое со знаком	. . .	. . .	целое со знаком	Eof

file of T, (файл, состоящий из записей типа T)
где T = record

```
           a : byte;
           b : char;
           c : integer;
     end;
```

первый элемент			. . .	*последний элемент*			
байт	код символа	целое со знаком	. . .	байт	код символа	целое со знаком	Eof

Примечание. Последний элемент файла, Eof (от слов End of file), обозначает физический конец файла, за которым нет ни одного элемента, принадлежащего ему. В старых версиях MS DOS для обозначения физического конца файла использовался специальный однобайтовый код. В последних версиях конец файла никак не кодируется, а файловая система своими средствами определяет, когда достигнут конец файла.

Логическая структура файла очень похожа на структуру массива. Различия между массивом и файлом заключается в следующем:

- У массива количество элементов фиксируется в момент распределения памяти в процессе трансляции, и он целиком размещается в оперативной памяти. Нумерация элементов массива выполняется согласно нижней и верхней границам для индексов, указанным при его объявлении.
- Файл располагается на внешнем носителе информации. Количество элементов файла не фиксировано. В процессе работы программы оно может изменяться и в каждый момент времени неизвестно. Зато, как уже упоминалось выше, известно, что в конце файла располагается специальный символ конца файла Eof. А определить длину файла и

выполнить другие часто необходимые операции с ним можно с помощью стандартных процедур и функций, предназначенных для работы с файлами. Нумерация элементов файла выполняется слева направо, начиная с нуля (исключая текстовые файлы).

В любой момент из программы файл доступен («виден») не целиком. Доступен только один его логический элемент (байт, целое число в машинной форме, запись и т.п.). В этом контексте можно говорить о том, что из программы в любой момент времени «файл виден через логическое окно», ширина которого точно равна размеру одного элемента файла. Отметим также, что в литературе часто вместо термина «элемент файла» употребляется также термин «запись», смысл которого в общем случае не совпадает с термином record.

10.2. Классификация файлов в Турбо-Паскале

По особенностям логической организации и способам доступа к отдельным элементам файлов, а, следовательно, и по операциям с ними, файлы можно разбить на несколько классов.
По типу (по логической структуре) различают файлы:

- типизированные;
- текстовые;
- нетипизированные.

По методу доступа к элементам файла различают:

- файлы последовательного доступа
- файлы прямого доступа.

Типизированные файлы рассматриваются как совокупность элементов (записей) одного и того же типа. При этом размер записи определяется транслятором из объявления типа записи. Например, файл, тип записи которого – массив целых объемом 10 элементов, имеет размер записи – 20 байтов.

Нетипизированные файлы рассматриваются как состоящие из записей, каждая из которых есть некоторое число байтов. Сколько именно байтов входит в одну запись, задает программист. Название нетипизированный –характеризует то обстоятельство, что, поскольку тип записи не определен, только программист может знать, какие операции над содержимым файла возможны и корректны. А по умолчанию разрешены только операции пересылки (копирования).

Текстовые файлы рассматриваются как совокупность строк символьной информации.

Файл *последовательного доступа* рассматривается как упорядоченная последовательность записей. При этом записи не нумеруются, и для того, чтобы прочитать k-ю запись, надо прочитать последовательно все, расположенные перед ней, (k-1) записей. Отметим также, что операции

чтения и записи (а это основные операции при работе с файлами) выполняются всегда только от начала к концу файла.

Файл *прямого доступа* – это такой файл, для которого обеспечивается возможность непосредственного доступа к любой записи по ее номеру.

Файлами последовательного доступа могут быть файлы любого типа, а файлами прямого доступа – только типизированные и нетипизированные файлы, т.е. такими файлами не могут быть текстовые файлы.

10.3. Объявление файла

При необходимости работы с файлом, в программе прежде всего надо объявить его логический тип. Объявление типа файла заключается в объявлении типа его логического элемента.

Объявление файла производится в секции переменных (var) и имеет следующий вид:

```
Объявление типизированного файла:
Var < имя файловой переменной > : file of <тип элемента>;

Объявление текстового файла::
Var < имя файловой переменной > : text;

Объявление нетипизированного файла:
Var < имя файловой переменной > : file;
```

Assign (Назначение файла)

Для работы с физическим файлом, находящемся на внешнем устройстве, необходимо первоначально связать его с файловой переменной (логическим файлом), с помощью которой будет осуществляться доступ к этому физическому файлу. Это связывание выполняется процедурой Assign, которая может быть применена только к закрытому файлу. Структура вызова этой процедуры имеет вид:

```
Assign (<имя файловой переменной>,<имя физического файла>);
```

Пример назначения файла для случая, когда имя физического файла записано по правилам MS DOS:

```
Assign (fl, 'MyFile.dat');
Assign (fout, 'c:\mylib\base.dat');
```

После выполнения назначения все операции с файлом производятся с использованием только логического имени файла (файловой переменной).

После закрытия файла можно произвести новое назначение для файловой переменной, указав в операторе Assign имя другого физического файла. Это позволит выполнять одни и те же операции обработки для большого числа разных физических файлов, не меняя текста программы. Достаточно только поменять имя физического файла. Указанная возможность, без повторной компиляции программы, может быть обеспечена за счет того, что имя физического файла может быть задано не только в виде символьной константы, но и в виде символьной переменной, значение которой можно менять в процессе выполнения программы:

```
Assign (fl, filename);
где filename - переменная типа string
```

10.4. Открытие и закрытие файла

Основные операции с файлами – чтение из файла и запись в него. Перед этим файл необходимо открыть, а после завершения работы с файлом его необходимо закрыть. Этой цели служат следующие процедуры:

reset(<файловая переменная>);	открыть файл для чтения (такой файл считается входным)
rewrite(<файловая переменная>);	открыть файл для записи (такой файл считается выходным)
close(<файловая переменная>);	закрыть файл

Процедура reset открывает существующий физический файл, который связан с файловой переменной. При открытии указатель текущей записи файла устанавливается в его начало (на первый элемент). Условно будем считать, что текущий элемент файла виден через так называемое «окно файла».

Если файл не существует (в той директории, которая указана в процедуре Assign), возникает ошибка времени выполнения, и программа аварийно завершается по инициативе операционной системы. Эту реакцию операционной системы (ее называют стандартной реакцией операционной системы) можно подавить, предоставив программе возможность самостоятельно отреагировать на ошибку (открытия).

Для этого в среде MS DOS используется специальная директива компилятора {$I–}, которая должна быть помещена до вызова процедуры reset. Она означает следующее: «выключить стандартную реакцию операционной системы на ошибку ввода/вывода». При этом факт ошибки регистрируется, но выполнение программы продолжается. Установить, успешно или неудачно выполнилась процедура открытия, можно воспользовавшись функцией IOResult, которая возвращает значение 0, если

операция закончилась успешно, и ненулевое значение (код ошибки) в противном случае.

При чтении из файла полезна еще одна функция. Это функция EOF (от End Of File):

```
EOF(<файловая переменная>);
```

Она возвращает значение **FALSE**, если «окно файла» указывает на имеющийся элемент, и значение **TRUE**, если «окно» вышло за пределы файла (за последний элемент его).

Процедура rewrite создает новый (пустой) физический файл. Если такой физический файл уже существует, он уничтожается, а вместо него создается новый пустой файл. При открытии «окно файла» устанавливается в позиции, в которую будет произведена запись первого элемента (в момент выполнения операции записи в файл).

Процедура close разрывает связь логического и физического файлов. После этого можно использовать файловую переменную для связи с другим файлом или, при необходимости, для изменения статуса логического файла (например, вместо входного сделать файл выходным).

10.5. Чтение и запись

Процедуры чтения и записи – применительно к операциям с файлами были приведены в разделе «Кодирование вычислительных операций». Тем не менее, повторно приведем их структуру:

read(<ф.п.>,<имя переменной>);	присвоить переменной значение, прочитанное из файла
write(<ф.п.>,<имя переменной>);	записать в файл значение переменной

Здесь *ф.п.* – файловая переменная.

После выполнения любой из приведенных операций указатель перемещается на следующий элемент файла. Именно в этом случае при чтении из файла, когда окно выходит за пределы файла, возникает ситуация «Конец файла». При этом, естественно, операцию чтения из файла выполнить невозможно, и попытка чтения вызывает аварийное завершение программы.

Еще одной полезной функцией является функция «количество записей в файле» - Filesize(<файловая переменная>) . Она может быть применена только для типизированных и нетипизированных файлов, и возвращает целое число типа longint (длинное целое), поскольку число записей файла может превышать МаксЦел, которое в Паскале равно 32767.

Ниже приведен пример, иллюстрирующий изложенный материал.

```
{ Вычислить и вывести на экран сумму целых чисел, }
{ являющихся элементами типизированного файла }
Program FSumm;
var
   fl : file of integer;
   x  : integer;
   sum: longint;
begin
   Assign(fl,'myfile.dat');
   {$I-}  {выключить стандартную реакцию
                    на ошибку ввода/вывода}
   reset(fl);
   {$I+}  {включить стандартную реакцию
                    на ошибку ввода/вывода}
   if (IOResult<>0) then
      writeln('Ошибка открытия файла. Конец работы.')
   else begin
      {Вычисление суммы элементов файла}
      Sum := 0;
      while (not EOF(fl)) do begin
         read(fl, x);
         Sum := Sum + x;
      end;
      Close(fl);
      writeln('Сумма элементов файла = ',Sum);
   end;
end.
```

10.6. Текстовые файлы

В отличие от типизированных файлов, о которых речь шла выше, работа с текстовыми файлами несколько отличается. Рассмотрим основные отличия.

- Во-первых, структура текстового файла, размещенного на внешнем устройстве (например, на магнитном диске), имеет следующий вид:

<table>
<tr><td>код символа</td><td>код символа</td><td>. . .</td><td>Eoln</td></tr>
<tr><td>код символа</td><td>код символа</td><td>. . .</td><td>код символа</td><td>Eoln</td></tr>
<tr><td>код символа</td><td>Eoln</td></tr>
<tr><td>код символа</td><td>код символа</td><td>. . .</td><td>код символа</td><td>код символа</td><td>Eoln</td></tr>
<tr><td>код символа</td><td>код символа</td><td>. . .</td><td>код символа</td><td>Eoln</td></tr>
</table>

Здесь Eoln – код конца строки (конца элемента файла). В файловой системе MS DOS конец строки кодируется двумя байтами: 0D (возврат каретки) и 0A (переход к новой строке). Приведем пример представления нескольких строк в текстовом файле в системе MS DOS:

Текст	первая		третья	четвертая
Коды	AF A5 E0 A2 A0 EF OD OA	OD OA	E2 E0 A5 E2 EC E5 0D 0A	EF A5 E2 A2 A5 E0 E2 A0 EF OD OA
№ строки	1	2	3	4

Здесь вторая строка пустая. Она представлена только символами {0D 0A} – конец строки.

- Во- вторых, процедуры чтения и записи имеют следующую структуру

```
read(<ф.п.>,< список ввода >);
readln(<ф.п.>,< список ввода >);

write(<ф.п.>,< список вывода >);
writeln(<ф.п.>,< список вывода >);
```

Здесь *список ввода* и *список вывода* – последовательность элементов через запятую. Правила построения списков изложены в разделе «Кодирование вычислительных операций», и здесь повторяться не будут.

- В-третьих, предусмотрены по две функции для операции ввода и вывода из текстового файла – read (write) и readln(writeln). Их отличие заключается в следующем:
  - После выполнения функций readln и writeln «окно файла» перемещается на начало следующей строки файла (строка – элемент текстового файла). При этом, если в текущей строке остается неиспользованной некоторая часть информации, она не обрабатывается.
  - После выполнения функций read и write «окно файла» остается направленным на ту же самую строку файла; только указатель позиции в ней смещается вправо, так что следующая операция чтения (записи) начнется с обработки текущей строки, начиная с позиции указателя.
  - Если при чтении из файла строка заканчивается, а список ввода остается не удовлетворенным до конца, «окно файла» автоматически будет перемещено на следующую строку, и ее сканирование продолжится.

Проиллюстрируем на примере специфику работы с текстовым файлом.

Пусть имеется текстовый файл 'input.txt' (см. ниже), в котором содержится входные данные для массива целых чисел:

– количество элементов = 5
– элементы : {12, -3, 5, 70, 65}

```
     5 количество чисел в масси-
ве
     12  -3  5
     70
```

Для того, чтобы комментарий в первой строке (’количество чисел в массиве’) не вызвал ошибки при чтении чисел, надо, чтобы он не был прочитан. Это может быть обеспечено так, как показано в программе на Паскале, приведенной на следующей странице.

В результате выполнения программы на экране будет сформирован следующий текст

```
Объем массива = 5
  12  -3   5
  70  65
```

Следует обратить внимание на то, что файл должен быть закрыт (close) только в том случае, если его удалось открыть. Попытка же закрыть неоткрытый файл приведет к аварийному завершению программы.

```
Program Demo; {Вывод на экран чисел из файла}

var
    A:array[1..100] of integer; {входной массив}
    nA : integer;  {фактический объем массива}
    j : integer; {рабочая переменная}
    fl : text;  {файловая переменная}
begin
    Assign(fl,'input.txt');
    {$I-}
    reset(fl);
    {$I+}
    if IOResult=0 then begin
       readln(fl,nA);  {из текущей строки взять только
           значение числа и после этого перейти на начало
           следующей строки}
       writeln('Объем массива = ',nA);
       for j:=1 to nA do begin
          read(fl,A[j]);
          write(A[j]:4);
          if (j mod 3)=0 then
             writeln; {сменить строку}
          {end if}
       end; {for}
       if (nA mod 3)<>0 then
          writeln; {сменить строку}
       {end if}
       close(fl);
    end
    else
       writeln('Ошибка открытия файла');
    {end if}
end.
```

Кроме процедуры открытия файла для записи (rewrite), призванной обеспечить создание нового файла, имеется еще одна процедура – append. Её назначение, открыть существующий текстовый файл для дописывания в его конец новых строк. Если же указанного в процедуре assign файла к этому моменту нет, создается новый пустой файл, как и при использовании процедуры rewrite. Данная процедура используется, например, при ведении системного журнала для регистрации определенных событий в системе.

В заключение отметим, что в Турбо-Паскале имеется еще ряд процедур и функций для работы с файлами. Но читателю предлагается изучить их самостоятельно.

10.7. Нетипизированные файлы

Нетипизированный файл – файл, элементами которого являются байты, а типы данных, размещенных в этих байтах безразличны. Рассмотрим операторы Паскаля, которые предусмотрены для работы с нетипизированными файлами.

Объявление файла:

```
var  <fp> : file;
    {здесь fp – файловая переменная}
```

Назначение файла:

```
assign ( <fp>, <имя физического файла>);
```

Открытие файла:

`reset (<fp>, <size>);`	Открыть файл для чтения, установив размер записи равным size байтов.
`rewrite (<fp>, <size>);`	Открыть файл для записи, установив размер записи равным size байтов.

Примеры.

```
reset(FromF, 1); {открыть файл с размером записи в 1 байт}
reset(fil, 256); {открыть файл с размером записи в 256 байтов}
rewrite(ToF, 12); {открыть файл с размером записи в 12 байт}
```

Чтение и запись

`BlockRead(<fp>,<buf>, <size>,<k>);`	Прочитать size записей из файла fp в буфер buf. Вернуть в переменной k, сколько записей прочитано фактически.
`BlockWrite(<fp>,<buf>, <size>,<k>);`	Записать size записей в файл fp из буфера buf. Вернуть в переменной k, сколько записей помещено в файл фактически.

Для данного типа файлов нет необходимости в применении функции EOF. После выполнения процедуры BlockRead достаточно сравнить значения числа size и k, и если k меньше чем size, достигнут конец файла.

Оператор закрытия файла совпадает с рассмотренными для других типов файлов.

Примеры использования нетипизированного файла можно найти в литературе и во встроенной в Турбо Паскаль справочной системе.

10.8. Пример. Программа работы с файлами

Задача. Разработать простую программу сопровождения базы данных.

База данных должна храниться на диске в виде типизированного файла, содержащего следующие сведения о результатах экзаменов студентов: фамилия и три оценки, полученные на экзаменах. Названия предметов не указываются, это сделано для упрощения программы. Допустимые значения оценок: 0, 2, 3, 4, 5. (Оценка 0 соответствует случаю, когда экзамен не сдавался).

Требуемые операции:

- занесение новых данных с клавиатуры
- просмотр содержимого базы данных на экране
- сохранение данных в типизированном файле
- загрузка (восстановление) данных из файла
- вывод содержимого базы данных в текстовый файл в табличном виде
- поиск информации в базе по ключу: поиск и отображение на экране сведений о студентах, имеющих неудовлетворительные оценки.

Примечание. В оперативной памяти содержимое базы данных должно размещаться в массиве записей.

Вначале приведем сценарий.
После запуска программы на экране появляется следующее меню:

```
Операции базы данных:
   добавить запись (1)
   просмотреть содержимое (2)
   искать неуспевающих (3)
   вывести в текстовый файл (4)
   сохранить базу (5)
   конец работы (6)
Введите номер операции _
```

Примечание. Восстановление содержимого базы данных из типизированного файла (загрузка в память программы) производится автоматически в момент запуска программы из файла с именем 'group.dat'. Если файл с этим именем не найден, база создается вновь, как пустая. Запись в этот файл должна происходить по желанию пользователя (операция 5) только в том случае, если в этот момент массив не пуст.

После того как пользователь вводит номер операции, в зависимости от введенного номера, выполняются следующие действия:

Операция 1. «Добавить запись».

На экране появляется запрос на ввод данных:

```
   Фамилия _
Оценка 1 _
Оценка 2 _
Оценка 3 _
```

Пользователь должен последовательно вводить ответы для каждой строки. При этом будет проводиться проверка допустимости вводимых данных. После завершения ввода, новая запись добавляется в массив, содержащий текущее значение базы данных, и на экране вновь появляется меню.

Операция 2. «Просмотр содержимого базы».

На экране появляются сведения обо всех элементах в табличном виде:

	Фамилия	Оценка 1	Оценка 2	Оценка 3
<фам.1>	<оц1>	<оц2>	<оц3>	
<фам.j >	<оц1>	<оц2>	<оц3>	

Если массив записей пуст, на экране появляется сообщение «База пуста».

Операция 3. «Поиск неуспевающих».

На экране появляется таблица, такая же, как и для предыдущей операции, но в ней должны присутствовать только строки со сведениями о студентах, у которых имеется по меньшей мере одна двойка или ноль. Если таковых в базе нет, на экране появляется сообщение «Неуспевающих нет».

Операция 4. «Вывод в текстовый файл».

Если массив записей не пуст, на экране появляется запрос имени файла и затем сообщение «Содержимое базы данных выведено в файл <имя файла>». В противном случае операция игнорируется.

Операция 5. «Сохранение базы данных».

Если массив записей не пуст, база сохраняется в файле ‘group.dat’, после чего на экране появляется сообщение «База данных сохранена». Если же массив пуст, операция игнорируется.

Операция 6. «Конец работы».

Работа программы завершается.

Примечание. После выполнения любой операции на экране вновь появляется меню.

Разработка программы.

При разработке программы почти все операции с базой данных реализованы как подпрограммы. А поскольку при выполнении всех операций используется один и тот же массив записей в памяти программы, подпрограммы оформляются как процедуры без списка параметров, а все необходимые данные являются для них глобальными.

Ниже программа приведена по частям.

1). Основной алгоритм.

а) Раздел деклараций:

```
Program Base;
       {Программа работы с простейшей базой данных.       }
       {Содержимое базы данных в памяти хранится в массиве }
       {     записей, а на диске -  в типизированном файле }

Uses CRT;  {Подключить модуль CRT, чтобы получить возможность
            использовать имеющиеся в нем процедуры }
```

```
Const
   MaxStudents = 10; {максимально возможное число записей
                      в базе. Для иллюстрации достаточно 10.}
Type
   TStudent = record  {запись со сведения об одном студенте}
        Name : string[25];  {фамилия}
        Estimations:array[1..3] of 2..5;  {оценки}
   end;
Var
   Students : array[1.. MaxStudents] of TStudent;
            {текущая база в оперативной памяти}
   NStud : integer;  {фактическое число записей в базе}
   OperCode:char; {код выбранной операции}
   FBase : file of TStudent; {файловая переменная}
```

б) Раздел операторов:

```
begin
   clrscr;  {очистить содержимое экрана. (Из модуля CRT) }
   OpenBase;  {восстановление базы из файла ’group.dat’}
   repeat
      ShowMenu;                {вывод меню}
      Readln(OperCode);        {читать код клавиши}
      case OperCode of
       ’1’ : AddStudent;       {добавить запись в базу}
       ’2’ : View;             {отображение содержимого базы}
       ’3’ : Search;           {поиск и отображение двоечников}
       ’4’ : ToFile;           {вывод в текстовый файл}
       ’5’ : SaveBase; {сохранение базы в файле ’group.dat’}
      end;
   until OperCode=’6’;
   writeln(’Работа завершена.’);
end.
```

Как видно из текста, после восстановления базы данных из файла, которое выполняется автоматически, последующие действия представляются циклом, в теле которого предусмотрены три действия:

- вывод меню,
- считывание с клавиатуры номера операции,
- ее исполнение.

Подпрограммы, реализующие отдельные операции, будут приведены ниже, хотя по правилам Паскаля они должны быть размещены в секции объявления процедур и функций, т.е. до оператора **begin** программы**,** или в другом модуле (смотрите главу 14).

2). Восстановление базы из файла

```
procedure OpenBase;
var
   n:integer;
begin
   NStud := 0;
   assign(FBase, 'group.dat'); {назначить физический файл}
   {$I-} Reset(FBase); {$I+} {исключить стандартную реакцию
                               при ошибке открытия файла}
   if IOResult=0 then begin  {если не было ошибки открытия}
      if SizeOf(FBase)<>0 then begin { объем файла не ноль}
         while (not EOF(FBase)) do begin
               {пока не достигнут конец файла}
            Inc(NStud); { аналог NStud:=NStud + 1;}
            read(FBase,Students[NStud]); {читать одну запись}
         end;
      end;
      Close(FBase);
   end;
end; {OpenBase}
```

3) Вывод меню

Производится каждый раз в начале цикла. Последний оператор процедуры выполняет функцию запроса на ввод. При этом используется оператор **write**, так чтобы обеспечить ввод номера функции в той же строке.

```
procedure ShowMenu;
begin
   writeln('---------------------------------------------');
   writeln('Операции базы данных:');
   writeln('  добавить запись (1)');
   writeln('  просмотреть содержимое (2)');
   writeln('  искать неуспевающих (3)');
   writeln('  вывести в текстовый файл (4)');
   writeln('  сохранить базу (5)');
   writeln('  конец работы (6)');
   write('Введите номер операции _');
end;
```

4). Отображение на экране содержимое базы

Для упрощения записи переменных в списке вывода используется оператор **with,** что позволяет получить более компактную запись оператора вывода.

Для сравнения покажем, как выглядит вывод без использования **with**:

```
      write(Students[k].name:15);
      write(Students[k].Estimations[1]:6);
      write(Students[k].Estimations[2]:11);
      writeln(Students[k].Estimations[3]:11);
```

Текст процедуры отображения выглядит следующим образом:

```
procedure View; {отобразить содержимое базы}
var
   k:integer;
begin
   writeln('-----------------------------------------------');
   if NStud=0 then
      writeln('База пуста')
   else begin
      writeln('   Фамилия    Оценка 1   Оценка 2   Оценка 3');
      for k:= 1 to NStud do
         with Students[k] do begin
               {печать отдельных полей записи}
            write(name:15);
            write(Estimations[1]:6);
            write(Estimations[2]:11);
            writeln(Estimations[3]:11);
         end;
      {end for}
   end;
end; {View}
```

5). Добавление новой записи

Алгоритм: После ввода сведений с клавиатуры производится проверка, не присутствует ли уже запись с тем же именем в поле name . В случае отрицательного ответа запись добавляется в базу, а в случае положительного – информация об этом выводится на печать, и запись в базу не добавляется. Добавления в базу данных не производится также в том случае, если массив заполнен.

```
procedure AddStudent;
        {Добавить новую запись в массив Students,
         если в нем есть место,
         и если записи с тем же именем нет}
var
   k:integer;
   estimate:integer;
   exist:boolean;
begin
   writeln('----------------------------------------------');
   if NStud<MaxStudents then begin
      Inc(NStud); {увеличить на 1 число элементов в массиве}
      with Students[NStud] do begin
         write('Фамилия_');
         readln(Name);
         for k:=1 to 3 do begin
            repeat
               write('Оценка ',k,'_');
               readln(estimate);
            until estimate in [0, 2..5];{проверка корректности}
            Estimations[k]:=estimate;
         end;
      end;
      {проверить, не существует ли записи
       с тем же именем студента}
      exist:=false; {пока новое имя не найдено}
      k:=1;
      while (not exist) and (k<NStud) do
      begin
          if Students[k].Name=Students[NStud].Name then
          begin
             writeln(Students[NStud].Name,
                       ' уже имеется в базе. ',
                       ' Запись не произведена.');
             exist:=true; {искомое имя в базе уже есть}
          end
          else
            Inc(k);
          {end if}
      end; {while}
      if exist then begin
          {удалить новый элемент}
         Dec(NStud); {аналог Ntud := NStud -1}
      end;
   end
   else
      writeln('База полна. Запись в нее невозможна.');
   {end if}
end;
```

6). <u>Сохранение базы в типизированном файле.</u>

```
procedure SaveBase; {сохранение базы в файле 'group.dat'}
var
   k:integer;
begin
   writeln('-----------------------------------------------');
   if NStud=0 then
      writeln('База пуста.')
   else begin
      rewrite(FBase); {переменная FBase уже была назначена
                       при открытии базы }
      for k:= 1 to NStud do
         write(FBase,Students[k]); {вывести одну запись}
      {end for}
      close(FBase);
      writeln('База данных сохранена');
   end;
end;
```

7). <u>Поиск в базе</u>

<u>Алгоритм</u>: Последовательно просматриваются записи, и каждая, в которой имеется хотя бы одна двойка или ноль, выводится на печать. При выводе первой записи, содержащей двойку, сначала выводится заголовок.

```
procedure Search; {поиск и отображение неуспевающих}
var
   k:integer;
   n:integer; {число неуспевающих}
begin
   writeln('----------------------------------------------');
   if NStud=0 then
      writeln('База пуста')
   else begin
      n:=0;
      for k:= 1 to NStud do
         with Students[k] do begin
            if (Estimations[1] in [0,2]) or
               (Estimations[2] in [0,2]) or
               (Estimations[3] in [0,2]) then
            begin
               if n=0 then begin {найдена первая запись}
                  writeln('     Фамилия     Оценка 1 ',
                          '  Оценка 2   Оценка 3');
                  Inc(n);
               end;
               write(name:15);
               write(Estimations[1]:6);
               write(Estimations[2]:11);
               writeln(Estimations[3]:11);
            end;
         end; {with}
      {end for}
      if n=0 then
         writeln('Неуспевающих нет')
      {end if}
   end;
```

8). Вывод в текстовый файл

```
procedure ToFile; {вывод в текстовый файл}
var
   namefile:string; {имя текстового файла}
   textfile:text;   {файловая переменная текствого файла}
   k:integer;
begin
   writeln('----------------------------------------------');
   if NStud=0 then
      writeln('База пуста. Вывод не производится.')
   else begin
      write('Введите имя текствого файла _');
      readln(namefile);
      assign(textfile, namefile);
      rewrite(textfile); {открыть файл для записи в него}
      writeln(textfile, '    Фамилия     Оценка 1  ',
                        ' Оценка 2   Оценка 3');
      for k:= 1 to NStud do
         with Students[k] do begin
            write(textfile, name:15);
            write(textfile, Estimations[1]:6);
            write(textfile, Estimations[2]:11);
            writeln(textfile, Estimations[3]:11);
         end;
      {end for}
      Close(textfile);
      writeln('Содержимое базы выведено в файл ',namefile);
   end;
end;
```

В заключение читателю предлагается собрать все приведенные тексты в один (по правилам Паскаля) и испытать полученную программу. А после этого добавить функцию «Добавить в базу новые записи из текстового файла». Пусть каждая строка этого файла содержит сведения об одном студенте, а ее структура пусть имеет вид:

```
<фамилия> пробел <оценка1>пробел <оценка2>пробел <оценка3>
```

Пример такого файла:

```
Сергеев А. 3 4 3
Пиотровский Б. 5 4 2
Ян Е. 4 4 4
Семенов 5 2 4
```

Вопросы для самоконтроля

1. Что такое файл?
2. В чем различие между структурой физического и структурой логического файлов?
3. Какие виды файлов вы знаете?
4. Что понимается под термином «способ доступа»?
5. Что следует из того, что некоторый файл является файлом последовательного доступа?
6. В чем состоит сходство и различие массива и файла?
7. Что необходимо сделать для того, чтобы получить возможность записи или чтения из файла?
8. Какой формат имеет процедура write для типизированного файла?
9. В чем особенности текстовых файлов?
10. Когда возникает ситуация EOF?
11. Как обеспечить возможность обработки ошибки при работе с файлом в программе?

11. ПОДПРОГРАММЫ

Подпрограмма – средство представления вспомогательного алгоритма на языке программирования.

В Паскале подпрограммы могут быть оформлены либо как процедуры (`procedure`), либо как функции (`function`). Подпрограмма оформляется как функция чаще всего в том случае, когда она возвращает в качестве результата одно единственное скалярное значение. В остальных случаях более предпочтительно оформлять ее как процедуру. И те, и другие имеют одинаковую структуру, т.е. состоят, как и программа, из секций объявлений и исполнимой части, содержащей операторы.

Структура подпрограммы имеет следующий вид

```
<Заголовок подпрограммы>;
<Раздел указания используемых модулей>
<Раздел деклараций>
        < Секция объявления локальных констант>
        < Секция объявления локальных типов>
        < Секция объявления локальных переменных>
        < Секция объявления внутренних процедур и функ-
ций>
begin
        <Операторы, описывающие алгоритм>
end;
```

Отличия в описании функции и процедуры касаются только заголовка и наличия в функции, среди прочих операторов, по крайней мере одного оператора присваивания, в котором имени функции присваивается значение возвращаемого результата. Отметим также, что текст подпрограммы заканчивается ключевым словом **end** и символом ";" (точка с запятой), а не точкой, как в случае программы.

Структура заголовка программы приведена ниже

```
function <имя> (<список формальных параметров>):<тип результата>;
   {Вызов функции может выполняться там, где допускается
     записывать выражение, в частности, в правой части оператора
     присваивания или в списке вывода.
     Например:  Sum := Summa(Vector);
                Writeln('Сумма элементов = ', Sum(Vector));}

procedure <имя> (<список формальных параметров>);
    {Вызов процедуры выполняется отдельным оператором.
     Например:   Summa(n, Vector); }
```

Примечание. Число параметров подпрограммы может быть как большим, так и малым. В принципе, возможны ситуации, когда параметры отсутствуют совсем. В этом случае заголовок подпрограммы может иметь вид, приведенный ниже.

```
function <имя>:<тип результата>;
procedure <имя>;
```

11.1. Область действия идентификаторов

Областью действия (сферой видимости) идентификатора называется часть программы, где он может быть использован.

Область действия идентификаторов определяется местом их объявления. Если идентификаторы можно использовать только в пределах одной подпрограммы, то какие идентификаторы называются **локальными**. Если же действие идентификатора распространяется на несколько (не менее одной) вложенных друг в друга подпрограмм, то такие идентификаторы называются **глобальными**.

Отметим, что понятия «глобальные» и «локальные» следует понимать ***относительно*** – по отношению к конкретной подпрограмме.

Продемонстрируем сказанное на примере. Пусть программа имеет следующую структуру:

```
Program Scope;
   var A0, B0, C0 : integer;
   procedure P1;
      var A1, B1, C1 : integer;
      procedure P2;
         var A2, B2, C2 : integer;
         begin
            Допустимо использование
            как глобальных A0, B0, C0, A1, B1, C1,
            так и локальных A2, B2, C2
         end; {procedure P2}
      begin
         Допустимо использование
         как глобальных A0, B0, C0,
         так и локальных A1, B1, C1
      end; {procedure P1}
begin
   Допустимо использование только A0, B0, C0,
end.
```

В данном примере:

- `A0, B0, C0` будут глобальными для всех подпрограмм, используемых в программе;
- `A1, B1, C1` будут глобальными для всех подпрограмм, описанных внутри процедуры `P1` (в данном примере для процедуры `P2`), и одновременно локальными для самой процедуры `P1`;

- `A2, B2, C2` – видимы (локальны) только в процедуре `P2.`

Сформулируем правила определения области действия для идентификаторов подпрограмм (процедур и функций):

- действуют все идентификаторы, определенные внутри подпрограммы;
- действуют все идентификаторы окружающего контекста, если имена отличаются от имен, объявленных внутри подпрограммы;
- локальные идентификаторы подпрограммы во внешнем окружении действовать не будут никогда;
- в случае совпадения имен глобального и локального идентификаторов действовать будет только внутренний локальный идентификатор. Это означает следующее: объявление во внутренней подпрограмме данных с идентификаторами, совпадающими по имени с данными внешних программы или подпрограмм, отменяет действие внешних идентификаторов и вводит свои локальные описания, независимо от того, совпадают они по типу, или нет.

Примечание. Локальные данные создаются при вызове подпрограммы и существуют только во время ее выполнения. Выделение памяти для локальных данных происходит автоматически в начале выполнения подпрограммы, а освобождение этой памяти – как только выполнение подпрограммы заканчивается. Следствие этого: *значения локальных данных существуют, пока подпрограмма выполняется*. Как только она завершается, все изменения значений локальных данных, сделанные операторами подпрограммы, исчезнут вместе с освобождением занимаемой ими памяти.

11.2. Способы передачи параметров

Взаимодействие программной единицы (программы или подпрограммы) с некоторой подпрограммой происходит в результате обращения к последней (вызова её). Прежде чем рассмотреть способы обмена информации между ними, введем следующее обозначение. Ту подпрограмму, к которой происходит обращение, будем называть вызываемой подпрограммой, а ту программную единицу, которая вызывает подпрограмму, в данном разделе будем называть вызывающей программой (хотя вызывающей может быть как программа, так и подпрограмма). Это будет сделано только для более компактного изложения материала.

Существуют два варианта обмена информацией между вызывающей программой и вызываемой подпрограммой:

1) с помощью механизма формальных - фактических параметров;
2) с использованием глобальных переменных .

Вначале коснемся способа, основанного на использовании глобальных переменных. Напомним, что глобальные переменные – это переменные, описанные в вызывающей и доступные в вызываемой подпро-

грамме. Поэтому значения всех глобальных переменных, которые были изменены в подпрограмме, без всяких дополнительных действий могут быть использованы и в вызывающей программе.

Несмотря на очевидную простоту данного способа взаимодействия, он имеет и недостатки. В первую очередь это относится к тому, что в процессе работы подпрограммы кроме переменных, в которых должны быть возвращены результаты работы вызываемой подпрограммы, могут быть изменены также значения тех переменных вызывающей программы, которые не должны были изменяться. Это приведет к тому, что гарантировать правильную работу вызывающей программы только на основе анализа ее алгоритма, станет невозможно. Могут возникнуть проблемы также и при разработке подпрограммы, поскольку обеспечить правильность ее работы в процессе разработки также будет непросто, а иногда и невозможно, в связи с тем, что нельзя обеспечить гарантированно известные значения глобальных для нее переменных в подпрограмме к моменту начала ее работы после вызова.

Следствием сказанного становится невозможность применения в чистом виде технологии нисходящего проектирования, а это усложняет процедуру и увеличивает время разработки программы.

Теперь рассмотрим способ, основанный на передаче параметров.

При активизации (вызове) подпрограммы ей можно передать параметры. Параметры, указываемые в заголовке подпрограммы при ее описании, называются **формальными**. Посредством этих параметров вызывающая программа может передать входные данные в вызываемую подпрограмму, а при завершении работы последней получить результаты ее работы.

Параметры, которые содержат исходные данные для работы подпрограммы, называются *входными*. Их значение обязательно должно быть определено к моменту вызова подпрограммы. Параметры, в которых подпрограмма формирует результаты работы, возвращаемые в вызывающую программу, называются *выходными*. Значения выходных параметров могут быть не определены (не инициализированы) при вызове подпрограммы, но обязательно получат значения в процессе ее выполнения.

Некоторые параметры могут быть одновременно входными и выходными, т.е. вызывающая программа должна инициализировать их до или в момент вызова подпрограммы, а после завершения работы последней получит в них новые значения (результаты). Такие параметры оформляются также как и выходные, но при разработке подпрограммы учитывается, что их значения в момент вызова инициализированы.

Формальные параметры обеспечивают возможность задать формальные функциональные связи между входными и выходными параметрами.

Параметры, указываемые при вызове подпрограммы, называются **фактическими**. Они определяются при вызове подпрограммы и в них используются данные, внешние по отношению к вызываемой подпрограмме. В момент вызова формальные параметры заменяются фактическими.

При передаче параметров необходимо обеспечить выполнение следующих требований:

- число формальных и фактических параметров должно совпадать;
- порядок перечисления формальных и фактических параметров должны совпадать;
- каждый фактический параметр должен соответствовать по типу своему формальному параметру.

Сфера действия имен параметров такая же, как и локальных данных.

По способу передачи параметры различаются:

1. По взаимодействию вызывающей и вызываемой подпрограммы:
   - только входной параметр;
   - только выходной параметр;
   - как входной, так и выходной параметр (входной-выходной).
2. По механизму передачи:
   - передача по значению;
   - передача по наименованию (по ссылке)

Суть передачи *по значению* заключается в следующем: в момент вызова значение фактического параметра, присваивается переменной, являющейся формальным параметром. Если формальный параметр выходной, то при возврате из подпрограммы, его значение присваивается соответствующему фактическому параметру.

При передаче *по наименованию* присваивания значений не происходит, а вместо этого, адрес в памяти формального параметра замещается адресом фактического параметра (ссылкой на фактический параметр).

В Турбо Паскале реализованы следующие способы передачи:

- если входной параметр скалярный – он может передаваться как по значению, так и по ссылке;
- если входной параметр не скалярный (например, массив) – он может передаваться только по ссылке;
- выходной или входной-выходной параметры могут передаваться только по ссылке.

При описании параметра в заголовке подпрограммы необходимо придерживаться следующих правил:

- если параметр передается по значению, его описание в заголовке имеет вид

```
<идентификатор> : <тип>
```

- если параметр передается по ссылке, его описание в заголовке имеет вид

```
var <идентификатор> : <тип>
```

Требования к способу представления фактического параметра:

- если параметр подпрограммы входной и передается по значению, фактическим параметром может быть любое выражение (константа или переменная – это частный случай выражения), имеющее тип, совместимый по присваиванию с типом формального параметра;
- если параметр подпрограммы передается по ссылке, фактическим параметром может быть только идентификатор переменной, совместимый по присваиванию с типом формального параметра.

11.3. Примеры

Пример 1. В первом примере рассмотрим программу, в которой используется подпрограмма, обменивающаяся с вызывающей программой посредством параметров.

Задача:

Данные о служащих содержат фамилию и оклад.

Требуется занести данные с клавиатуры в массив записей и вывести фамилии служащих, имеющих оклад выше среднего.

Ограничение: число служащих не больше 10-ти.

Порядок ввода данных:

- число служащих;
- последовательность строк, содержащих разделенные пробелом фамилию и оклад служащих.

Порядок вывода результата:

- после строки "Результат: " – фамилии служащих, имеющих оклад выше среднего на отдельных строках.

Прежде всего, выберем структура данных в памяти программы. Пусть она имеет следующий вид :

Data	Name	Name	...	Name	...	Name		
	Salary	Salary		Salary		Salary		Sum
	[1]	[2]		[i]		[N]		

Здесь

- Data – массив записей, каждая из которых состоит из двух полей (Name, Salary) и содержит сведения об одном служащем;
- Sum – переменная, в которой будет записана сумма окладов всех служащих.

Для этого в тексте программы надо поместить следующее объявление типа `TData`:

```
const
   MaxSize = 10; {Максимальное число служащих}
type
   TData = record {тип записи о служащем}
      Name: string[20]; {фамилия}
      Salary: real;{оклад; стандартный тип Real выбран для
              представления чисел больше 32767            }
   end;
```

И, в соответствии с приведенным объявлением типа, объявим следующие переменные:

```
var
   Data: array[1..MaxSize] of TData; {массив записей
                                        о служащих}
   N: Integer; {число служащих}
   Sum: Integer; {суммарный оклад}
   MeanSalary: Real; {средний оклад}
```

Теперь можно приступить к разработке алгоритма. Вначале приведем укрупненный алгоритм на псевдокоде:

```
Алг 'Demo'
Начало
   Ввод массивов записей  {в массив Data}
   Вычисление среднего оклада  {В переменной Mean}
   Вывод фамилий сотрудников
Конец
```

Ввод сведений об одном служащем будем производить в отдельной подпрограмме, оформив ее как функцию с именем `ReadData`. Текст функции приведен на следующей странице.

Обратите внимание, что параметр `Rec` этой функции имеет атрибут `var`, показывая, что это возвращаемый (выходной) параметр и что информация передается в программу по ссылке (по адресу переменной в вызывающей программе). Результат работы функции – число, свидетельствующее об успешности ввода.

```
{ Функция ввода записи о служащем                                  }
 { Формат строки ввода:                                            }
 {       <фамилия><пробел><оклад>                                  }
 { Результаты функции - код ошибки при вводе:                      }
 {       при успешном вводе результат = 0                          }
 {       если строка пустая, результат = -1                        }
 {       если оклад не задан или задан неверно, результат = 1      }
 {    При отсутствии ошибки в переменной Rec возвращаются          }
 {                               сведения о сотруднике.            }
 { Примечание: для пользователя удобнее при вводе                  }
 {             отдельно запрашивать имя и оклад                    }
 {                                                                 }
 function ReadData(var Rec: TData): Integer;
 var
   S: string[80];{ строка ввода }
   P: integer;{ номер позиции в строке }
   SalaryS: string[10];{ строка, содержащая оклад }
   ErrorCode: integer;{код ошибки при преобразовании
                       строки в число;
begin
    readln(S);
    if S = '' then  {Строка пустая}
       ReadData := -1
    else begin
       P := Pos(' ',S);
       if P = 0 then {Пробела нет}
          ReadData := 1
       else begin
          Rec.Name := Copy(S, 1, P-1);
          SalaryS := Copy(S, P+1, 255{до конца строки});
          Val(SalaryS, Rec.Salary, ErrorCode);
          if ErrorCode <> 0 then {Ошибка}
             ReadData := 1
          else
             ReadData := 0;
          {end if}
       end; {if}
    end;
end;
```

Теперь приведем текст программы, решающей основную задачу.

```
program Demo;
  {Программа заносит данные с клавиатуры в  массив  записей}
  {и выводит фамилии сотрудников, имеющих оклад выше среднего}
   <Здесь вставить объявление константы MaxSize, типа TData
     и текст функции ReadData (приведены выше)>
var
   Data: array[1..MaxSize] of TData;
         {массив записей о служащих}
   N: integer; {число служащих}
   ReadResult: integer; {результат ввода записи}
   Sum: real; {суммарный оклад}
   MeanSalary: real; {средний оклад}
   i: integer; {номер очередного элемента массива}

begin
   writeln('Программа заносит данные с клавиатуры в массив ',
           'записей и');
   writeln('выводит фамилии сотрудников, имеющих оклад выше ',
           'среднего.');
      {Ввод числа записей}
   repeat
      write('Задайте число служащих (1..',MaxSize,') ');
      readln(N);
      if (N <= 0) or (N > MaxSize) then
         writeln('Вы ошиблись! Повторите!');
   until (N > 0) and (N <= MaxSize);
      {Ввод массива записей}
   writeln('Введите фамилии служащих и через пробел их ок-
лад');
   for i := 1 to N do begin
      repeat
         ReadResult := ReadData(Data[i]);
         if ReadResult <> 0 then
            writeln('Вы ошиблись! Повторите!');
         {end if}
      until ReadResult = 0;
   end;

      {Вычисление среднего оклада}
   Sum := 0;
   for i := 1 to N do
      Sum := Sum + Data[i].Salary;
   MeanSalary := Sum / N;

      {Вывод фамилий сотрудников}
   writeln('Результат:');
   for i := 1 to N do
      if Data[i].Salary > MeanSalary then
         writeln(Data[i].Name);
      {end if}
   {end for}
end.
```

Вызов подпрограммы-функции в программе производится в операторе

```
ReadResult := ReadData(Data[i]);
```

Здесь `Data[i]` – фактический параметр функции `ReadData.` В него возвращается прочитанная запись о служащем.

Пример 2. В данном примере для той же самой задачи, что и в первом примере, подпрограмма ввода записи о служащем оформлена как процедура.

Здесь приведем только текст процедуры и оператор вызова ее в программе. Остальные операторы программы полностью совпадают с приведенными выше. А вызов процедуры в программе оформляется отдельным оператором вида:

```
ReadData(Data[i], ReadResult);
```

Текст процедуры приведен на следующей странице.

Вопросы для самоконтроля

1. Какой вид имеет структура описания процедуры и функции?
2. В чем состоит отличие описания процедуры и функции?
3. Что такое область действия идентификатора?
4. Каковы основные правила определения области действия для идентификаторов процедур и функций?
5. Какие параметры называются формальными, и какие – фактическими?
6. По каким признакам различаются параметры?
7. Какие способы передачи параметров реализованы в Turbo Pascal?
8. Каковы правила передачи параметров по значению?
9. Каковы правила передачи параметров по ссылке?

Текст процедуры ReadData:

```
 { Процедура ввода записи о служащем                          }
 { Формат строки ввода:                                       }
 {      <фамилия><пробел><оклад>                              }
 { Результаты функции - код ошибки при вводе ( 0 или 1 )      }
 { При отсутствии ошибки в переменной Rec возвращаются    }
 {                               сведения о сотруднике.       }
{                                                            }
 procedure ReadData(var Rec: Tdata; var ErrorCode: Integer);
 var
   S: string[80];{ строка ввода }
   P: integer;{ номер позиции в строке }
   SalaryS: string[10];{ строка, содержащая оклад }
 begin
    readln(S);
    if S = '' then  {Строка пустая}
       ErrorCode := -1
    else begin
       P := Pos(' ',S);
       if P = 0 then {Пробела нет}
          ErrorCode := 1
       else begin
          Rec.Name := Copy(S, 1, P-1);
          SalaryS := Copy(S, P+1, 255{до конца строки});
          Val(SalaryS, Rec.Salary, ErrorCode);
          if ErrorCode <> 0 then {Ошибка}
             ErrorCode := 1
          else
             ErrorCode := 0;
          {end if}
       end; {if}
    end;
end;
```

12. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Используемые в программировании данные можно разделить на две группы:

- данные статической структуры – это такие данные, взаиморасположение и взаимосвязи которых в теле программы фиксируются на этапе ее трансляции и всегда остаются постоянными;
- данные динамической структуры – это данные, внутреннее строение которых формируется по какому-либо закону, но количество элементов, их взаиморасположение и взаимосвязи могут динамически изменяться во время выполнения программы.

В данном разделе рассматриваются вопросы создания и использования динамических структур данных.

12.1. Указатели

В результате построения текста исполнимой программы область памяти, занимаемая ею, разбивается на две части – область кодов, в которой располагаются команды программы, и область данных, в которой размещаются статические данные (константы и переменные).

Отметим, что статические переменные – это такие переменные, которые объявляются в секции переменных (напомним, что идентификатором секции является ключевое слово `var`).

Ниже приведена схема размещения отдельных областей программы в оперативной памяти ЭВМ:

Статическая область программы		Свободная область ЭВМ (*Heap*)
Область кодов (команды программы)	Область статических данных (константы и переменные)	Область динамических переменных

Динамические переменные создаются в процессе работы программы, по ее запросу, размещаются вне тела программы, созданного при ее построении, в так называемой динамической области. Для этого предусмотрены специальные процедуры.

Поскольку требуемая для размещения переменной память находится в динамической области, в программе для связи с ней должна располагаться статическая переменная, в которую можно поместить указание на местонахождение динамической переменной. Очевидно, что тип этой статической переменной должен быть особым. И такой тип определен в языке и называется он **указателем** (или ссылкой)**.**

Значением указателя является адрес первого байта области, занимаемой динамической переменной. Однако в языке удобно считать, что ука-

затель – это средство связи с динамической переменной, которое не зависит от особенностей архитектуры конкретного компьютера, на котором выполняется программа (т.е. от того, каким образом представляется адрес в конкретной модели компьютера). Для этого типа переменных определена одна константа – `NIL` , обозначающая «никуда не показывает» (аналог нуля для чисел).

Объявить переменную типа «указатель» можно двумя способами:

объявив тип «указатель» в секции типов (используется ниже для объявления переменной `Student`);

в секции переменных (используется ниже для объявления переменной `Stud`).

```
type
   TStudent=record
      Name:string[30];
      Age:integer;
   end;
   PStudent= ^Tstudent; {объявление типа «указатель»}
   TGroup = array[1..100] of TStudent;
var
   Student : PStudent; {переменная-указатель на
                         переменную типа TStudent}
   Stud : ^TStudent; {объявление переменной-указателя
                      на переменную типа TStudent}
   Students : ^TGroup; {указатель на массив TGroup }
```

Покажем, как объявляются динамические переменные в Паскале.

Здесь:

`Student` – статическая переменная типа «указатель» на динамическую область, в которой может быть помещена запись с информацией об одном студенте;

`Stud` – статическая переменная типа «указатель» на динамическую область, в которой может быть помещена запись с информацией об одном студенте

`Students` – статическая переменная типа «указатель» на динамическую область, в которой может быть размещен массив записей с информацией о студентах.

На момент начала выполнения программы значения переменных типа «указатель» не определены, поскольку динамические переменные еще не созданы. Для того чтобы точно зафиксировать этот факт, этим переменным необходимо присвоить значение `NIL`.

Для того чтобы создать динамическую переменную, необходимо выполнить специальную процедуру **New,** указав имя переменной типа «указатель», в которую эта процедура должна вернуть адрес первого

байта созданной переменной. Освободить эту область, когда в ней больше не будет необходимости, можно с помощью процедуры **Dispose**.

Для того чтобы получить доступ к самой динамической переменной, используется запись вида:

```
^<имя переменной-указатель>
```

Размер памяти в байтах, выделенный процедурой **New,** определяется автоматически из объявления типа. Так, для приведенных выше переменных `Student^` и `Stud^` в Турбо-Паскале будет выделено по 33 байта (31 – для поля `Name`, плюс 2 для поля `Age`) и для переменной `Students^` – 3300 байтов.

Ниже приведены примеры использования динамических переменных и их элементов:

```
New (Stud);  {выделить память для переменной Stud}
Stud^.Name:='Петров А.Б.'; Stud^.Age := 19;
New(Students); {выделить память для массива записей}
Students^[1].Name:='Андреев С.А.';
Readln(Student^[12].Age);
writeln(SizeOf(Students^)); {напечатать размер
                                области Students^}
```

Допустимые операции для переменных типа «указатель»:

- присваивание (например, `Student:=Stud;`)
- сравнение: либо = ; либо <> (например, `Stud<>NIL`).

Кроме указателей на переменные определенного типа, которые объявляет программист в тексте программы, в языке Паскаль предопределен тип Pointer (так называемый анонимный указатель). Он используется в случае, когда программист желает запросить из динамической области некоторое количество байтов. Для выделения и последующего освобождения памяти без указания типа данных, размещенных в ней, используются процедуры **`GetMem`** и **`FreeMem`** соответственно. Эти процедуры часто используются для создания в программе буферов для временного хранения информации, например, при копировании, хотя возможны и другие применения (см. ниже).

Подведем итоги изложенному выше.

1. Динамические переменные применяют тогда, когда в процессе выполнения алгоритма эти переменные нужны только ограниченное время.
2. Адрес динамической переменной становится известным только в момент ее создания, и для его хранения необходима статическая переменная типа «указатель».

3. Тип «указатель» может быть определен в секции **Type**, но имеется и предварительно определенный тип Pointer (анонимный указатель).
4. Переменная типа «указатель» может принимать значение NIL или адреса переменной.
5. NIL - специальное значение (константа), обозначающее пустой или нулевой указатель.
6. Если P – указатель, то P^ - динамическая переменная.
7. Если T – тип, то ^T – тип указателя на переменную типа T.
8. Если A и B – переменные типа «указатель», то для них возможны операции присваивания (A := B) и отношения (A=B и A<> B).

В Турбо-Паскале предусмотрены следующие процедуры создания и уничтожения динамических переменных

New (P)	Создать динамическую переменную, т.е. выделить область памяти в HEAP, размер которой равен размеру переменной P^, а адрес этой области записать в переменную P.
Dispose(P)	Освободить область памяти, занимаемую переменной P^, для другого использования (уничтожить переменную)
GetMem(P, Size)	Создать динамическую переменную P^ размером Size [байтов]
FreeMem(P, Size)	Освободить Size байтов памяти, занимаемую P^. (Освободить можно меньше памяти, чем было выделено по GetMem)

Функция **SizeOf(V)** - вычисляет размер, который занимает переменная V. Функция может быть использована для создания динамической переменной, размер которой на этапе разработки программы неизвестен, и определяется в процессе выполнения программы. Примером такого применения функции является создание динамического массива:

Пусть в программе объявлены следующие переменные:

```
type
    PStudents=array[1..100] of TStudent;
    PTStudent=^TStudents;
var
    PStudent:PTStudent;
    NStudents:Integer;
```

Тогда в исполнимой секции программы возможны следующие операторы:

```
   write (' Введите число студентов');
   readln(NStudents);
   GetMem(PStudent,SizeOf(TStudent)*NStudents); { зарезеровать
          память для размещения массива, содержащего не 100,
          а NStudents элементов}
```

Основной областью использования динамических переменных в программировании является организация и обработка динамических структур данных типа списков и деревьев. Динамические структуры данных типа «дерево» будут рассматриваться позже, а сейчас перейдем к рассмотрению структуры типа «список».

12.2. Динамические структуры типа «Список»

Список – это динамическая структура данных, представляющая собой некоторое множество однородных элементов, связанных друг с другом только отношением следования. Элементы списка – всегда динамические переменные, а связь элементов списка обеспечивается переменными типа «указатель на элемент». Число элементов в списке может быть любым; в частности список может быть пустым, то есть не содержать ни одного элемента.

Вначале приведем ряд определений.

По ***способу связи*** между элементами списка различают линейные и кольцевые списки, а также односвязные и двухсвязные списки:

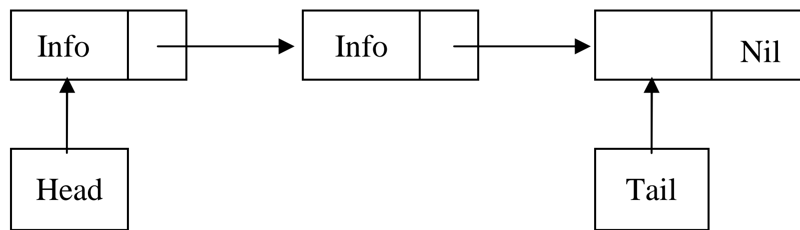
- Линейный список – это динамическая структура данных, которая представляет собой совокупность линейно связанных элементов, для которых разрешается добавлять элементы между любыми двумя другими и удалять любой элемент.
- Кольцевой или замкнутый список – это динамическая структура данных такой же структуры, что и линейный список, но имеющая дополнительную связь между последним и первым элементами списка.
- Односвязный список – это список, в котором каждый элемент связан («знает» о местонахождении) только с одним, следующим за ним элементом.
- Двухсвязный список – это список, в котором каждый элемент связан с двумя соседними для него элементами, как находящимся перед, так и находящимся после него.

По ***типу доступа*** к элементам списка различают очереди и стеки:

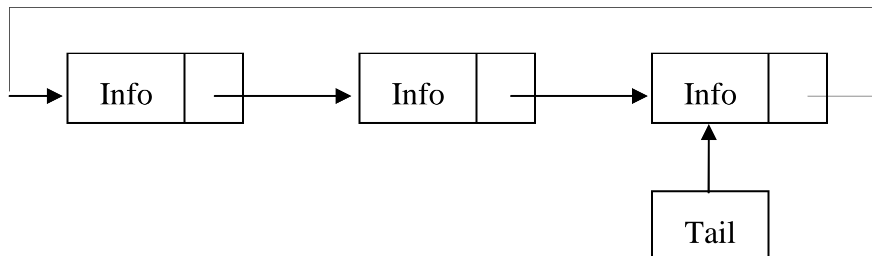
- Очередь – это частный случай линейного списка, для которого добавление нового элемента разрешено только в конец списка, а удаление – только из начала (*из головы*) списка.
- Стек – это частный случай линейного списка, для которого добавление нового элемента и удаление существующего всегда производится только из начала списка. Голову такого списка часто называют *вершиной стека*.

Приведем графические иллюстрации списков (здесь приведены частные примеры организации списков):

а) *Линейный односвязный* список (Head – переменная-указатель на первый элемент или голову списка; Tail – переменная-указатель на последний элемент списка; Info – область элемента, в которой хранится полезная информация)



б) *Кольцевой односвязный* список



12.3. Средства языка Паскаль для организации списков

Элемент списка создается для размещения в нем полезной информации (некоторых данных). Типы этих данных зависит от назначения списка, и могут быть какими угодно. В приводимых ниже примерах эта часть данных элемента списка будет иметь частный характер. Остальные части структуры элемента – переменные, используемые для организации списка, и их типы не зависят от назначения списка, то есть имеют не частный характер.

- Структура элемента в Паскале описывается с помощью конструкции типа `record`.
- Для связи элементов между собой используется тип «указатель на элемент».
- В программе для связи со списком должны быть объявлены одна или несколько статических переменных-указателей на элемент списка. Это может быть голова списка (Head), конец списка (Tail) и другие.

В качестве примера приведем объявление типа элемента, информационная компонента которого задается полем Data типа TData.

```
type
   PItem = ^TItem; {указатель на элемент двухсвязного списка}
   TItem = record  {описание структуры элемента
                    двухсвязного списка}
       Data : TData; {информационное поле элемента}
       Prev : PItem; {указатель на предыдущий элемент}
       Next : PItem; {указатель на следующий элемент}
   end;
        {При работе с односвязным списком
         указатель на предыдущий элемент не нужен:}
   PList = ^TList; {указатель на элемент односвязного списка}
   TList = record  {описание структуры элемента
                     односвязного списка}
        Data : TData; {информационное поле элемента}
        Next : PList; {указатель на следующий элемент}
   end;
```

Для создания элемента списка как динамической переменной используют процедуру New:

```
 {Пусть в программе объявлена следующая переменная:  }
var
   P : Pitem: {рабочая переменная для временного хранения
               указателя на элемент списка}
 {Тогда процедура создания динамической переменной для элемента
        списка может быть записана следующим образом:}
   New(P);
```

12.4. Типовые алгоритмы работы со списками

При рассмотрении алгоритмов тип информационного поля в них может отличаться. А переменные, которые будут использованы, считаются объявленными следующим образом:

```
var
    Head : Pitem; { Указатель на голову списка }
    Tail : Pitem; { Указатель на «хвост» списка }
    P : Pitem;    { Указатель на элемент списка }
```

А1. Инициализация списка. (Создание нового и пустого списка)

```
procedure InitList;
begin
   Head := NIL;
   Tail := NIL;
end;
```

A2. Добавить элемент в конец односвязного списка

В процедуре реализован следующий алгоритм:

1) создать элемент, подлежащий добавлению (зарезервировать память)
2) полям данных присвоить значения входных параметров
3) Если список не пуст:
   - установить указателю Next последнего элемента ссылку на новый
   - присвоить указателю Next нового элемента NIL
   - передвинуть Tail на новый элемент
4) Если список пуст
   - установить указатель Head на новый элемент
   - установить указатель Tail на новый элемент
   - присвоить указателю Next нового элемента NIL

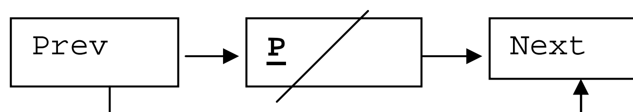
```
var
   pnew :PList; {указатель на новый элемент}
begin
   new(pnew);
   pnew.data:=indata; {присвоить полю Data значение}
   if Head <> nil then {список непустой ?}
      Tail^.Next := pnew
   else
      Head := pnew;
   {end if}
   pnew.Next:=nil;
   Tail := pnew;
end;
```

A3. Удалить элемент из середины списка с освобождением памяти

(для случая односвязного списка):

P – указатель на удаляемый элемент

Prev - указатель на элемент, стоящий перед удаляемым элементом



Фрагмент удаления имеет вид:

```
   P := Prev^.next;
   Prev^.Next := P^.Next;
   Dispose(P); {освободить память}
```

А4. Исключить элемент из начала списка (вернув поле данных)

```
procedure OutList(var OutDat:TData);
var
   P:Plist;
begin
   OutData := Head^.Data;
   p:=Head;
   Head:=Head^.Next; {присвоить указателю
               на голову списка ссылку на новый первый
              (бывший второй) элемент}
   Dispose(p);  {освободить память, занятую элементом}
end;
```

А5. Включить элемент (pnew) **в середину списка**

```
{Для случая односвязного списка}
   pnew^.next:=prev^.next;
   prev^.next:=pnew;
```

А6. Обработать каждый элемент линейного списка

```
P := Head;
while P <> NIL do begin
   Выполнить операцию обработки элемента P^
   P := P^.Next;  {перейти к следующему элементу}
end;
```

Пример. Пусть информационное поле элемента – целое число. Задача обработки: подсчитать количество элементов, значение информационных полей которых превышает значение Value:

```
Count := 0;
P := Head;
while P <> NIL do begin
   if P^.Data > Value then
      Count := Count +1;
   {end if}
   P := P^.Next;
end;
```

А7. Найти первый элемент, удовлетворяющий некоторому условию

```
var
    flagFound : boolean;
begin
    P := Head;
    flagfound := false;
    while  (P<>NIL) and (not flagfound) do
       if (выполнено условие) then
          flagfound := true  {элемент найден}
       else
          P := P^.Next;
    {end while}
end;
```

А8. Уничтожить список с освобождением памяти

В данном алгоритме используется следующий принцип: удалять последовательно элементы, начиная с первого в списке.

Это производится следующим образом: Вначале указатель на первый элемент (тот, который записан в указателе на голову списка *Head*) помещается в рабочую переменную P. Затем переменной *Head* присваивается значение указателя на следующий за ним. Это значение в односвязном списке размещается только в поле предыдущего (т.е. удаляемого) элемента. С этого момента бывший первый элемент выведен из списка, и его можно удалить из памяти компьютера, выполнив процедуру Dispose.

```
P := Head; {Установить указатель на удаляемый элемент}
while P<>NIL do begin
   Head := P^.Next; {установить ссылку на следующий}
   Dispose(P);  {освободить память,
                 занятую удаляемым элементом}
   P := Head;  {установить новый удаляемый элемент }
end;
```

А9. Добавить элемент в упорядоченный односвязный список
(сохранив упорядоченность)

```
procedure InSortList(indata:TData);
var
    p, pnext, pnew : plist;
begin
   new(pnew);
   pnew^.data := indata;
   if (Head=NIL) or (Head^.data > indata) then begin
         {добавить элемент в начало списка}
      pnew^.next := Head;
      Head := pnew;
   end
   else begin
      p := Head;
      pnext := p^.next;
      while (pnext <> NIL) and (pnext^.data < indata) do
      begin
         p := p^.next;
         pnext := pnext^.next;
      end;
          {включить элемент}
      pnew^.next := pnext;
      p^.next := pnew;
   end;
end;
```

Дополнительные сведения по организации и работе со списками рекомендуется получить из учебной литературы по программированию на Паскале, в частности, рекомендованной для данного курса.

В заключение отметим, что в данном разделе рассматриваются только процедуры New и Dispose, обеспечивающие работу с памятью на логическом уровне, т.е. с переменными, типы которых в программе явно объявлены. Отсюда при выполнении процедуры New(P) выделяется столько байт памяти, каков размер, занимаемый переменной P, и этот объем вычисляется автоматически.

Еще раз повторим, что процедуры GetMem и FreeMem обеспечивают работу с данными, типы которых им безразличны. Они оперируют с памятью в байтах. Поэтому, при их использовании, вычислять требуемые объемы памяти должен программист. Примеры использования указанных процедур приводятся при рассмотрении работы со структурами (**record**) и файлами.

Задачи для закрепления материала

Для закрепления материала предлагается решить следующие задачи:

Задача 1. Имеются две последовательность строк, в каждой из которых размещаются списки фамилий. Признак окончания последовательности – пустая строка. Последовательности могут вводиться из файла или с клавиатуры. Сформировать два однонаправленных линейных списка, поместив в каждый содержимое своей последовательности. Затем сравнить списки, и в качестве результата, после слов «Результат:», вывести одно из приведенных ниже сообщений:

- «Списки совпадают»
- «Первый список включает второй» (если второй список совпадает с началом первого)
- «Второй список включает первый»
- «Списки не совпадают» (в остальных случаях).

Алгоритмы заполнения списков и их сравнения оформить в виде подпрограмм.

Задача 2. Дана строка символов. Найти и вывести все подстроки, заключенные в круглые скобки.

Порядок вывода: после строки со словом «Результат:» вывести в отдельных строках все найденные подстроки. Если одна подстрока является частью другой, то сначала вывести более короткую.

Пример:

```
Исходная строка:   ((4+6)*7)+(6*(8+7))

Результат:
4+6
(4+6)*7
8+7
6*(8+7)
```

Вопросы для самоконтроля

1. В чем особенность объявления динамических переменных?
2. Что такое указатель и для чего он нужен?
3. Какие процедуры используются для создания и уничтожения динамических переменных?
4. Какие значения может принимать переменная типа «указатель»?
5. В чем сходство и различие между линейным и упорядоченным списками?
6. Какова структура элемента двухсвязного списка?
7. Чем отличаются и что объединяет структуры данных «список» и «очередь»?
8. В чем состоит особенность описания типов для создания динамических структур данных?
9. Сколько указателей требуется для создания очереди?
10. Какие действия необходимо выполнить для создания очереди?
11. Как добавить или удалить элемент очереди?

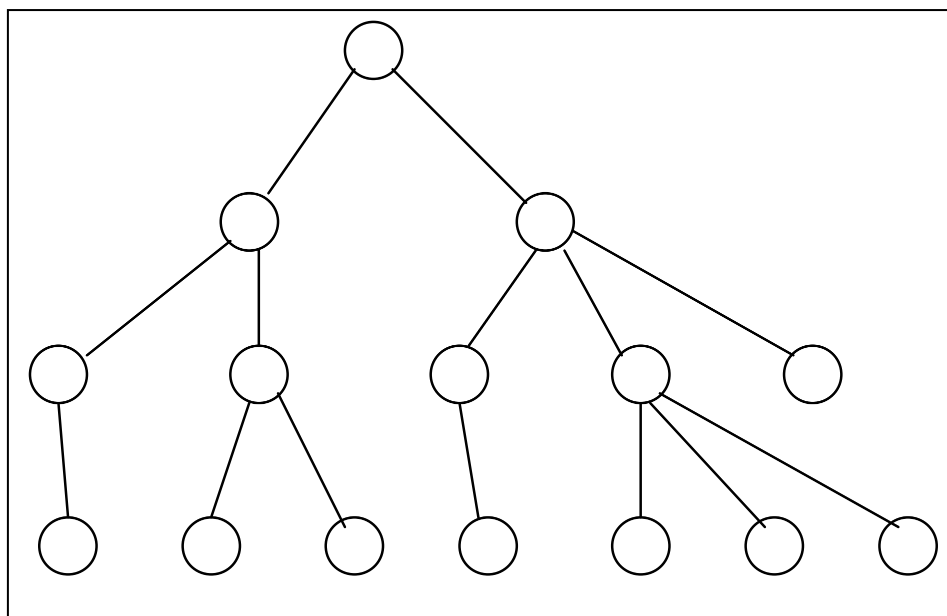
13. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ ТИПА «ДЕРЕВО»

13.1. Определение дерева и способы представления в программе

Структуру данных типа «дерево» можно определить следующим образом:

Древовидная структура с базовым типом Т это либо пустая структура, либо узел типа Т, с которым связано конечное число древовидных структур с базовым типом Т, называемых *поддеревьями*. Рассмотренная ранее структура данных типа «линейный разомкнутый список» может рассматриваться как вырожденное дерево.

Дерево можно изобразить несколькими способами. Наиболее наглядный способ – в виде графа. В этом случае граф выглядит как перевернутое дерево (отсюда и название структуры – «дерево»):



Приведем несколько определений, связанных со структурами данного вида.

Верхний узел дерева называют узлом уровня 1 или *корнем* дерева. Далее номера уровней возрастают. Максимальный уровень дерева называют также *глубиной* или высотой дерева. В приведенном выше графе глубина дерева (максимальный уровень узлов дерева) равна четырем.

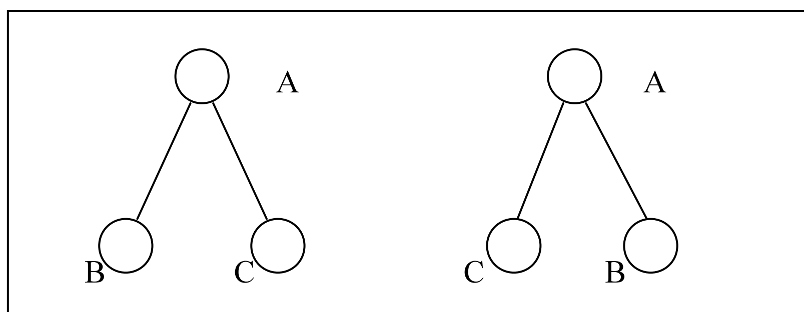
Два узла У (уровня k) и X (уровня k+1) определяют следующим образом:

узел У считается предком узла X, а узел X – потомком узла У.

Если узел не имеет потомков, он называется *терминальным* узлом или *листом*. Все прочие узлы называют *внутренними* узлами.

Упорядоченное дерево – дерево, у которого ветви каждого узла упорядочены.

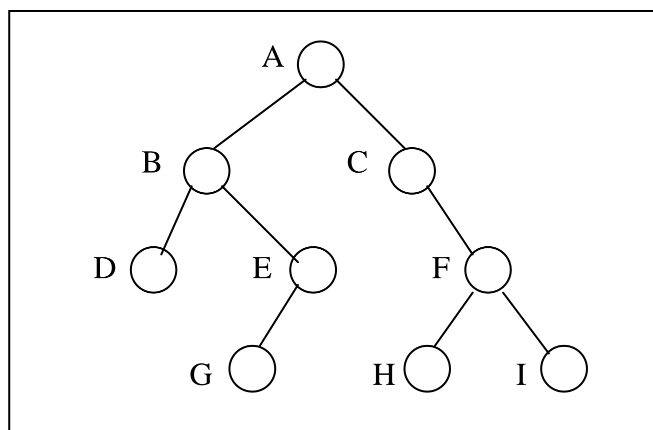
Приведенные ниже графы двух упорядоченных деревьев не идентичны (отличаются).



Дерево, каждый узел которого имеет не более двух потомков, называют *бинарным***.** По аналогии с этим *троичное* дерево – дерево, узлы которого не имеют более трех потомков. А дерево, отдельные узлы которого имеют более трех потомков, называют сильно ветвящимся.

Упорядоченное бинарное дерево – конечное множество элементов (узлов), каждый из которых либо пуст, либо состоит из корня (узла), связанного с двумя различными бинарными деревьями, называемыми левым и правым поддеревьями корня.

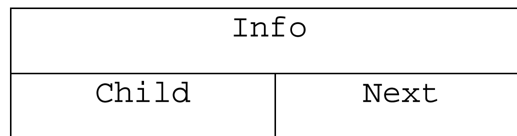
Ниже приведен пример бинарного дерева:



Здесь:

A – корень дерева
B – корень левого поддерева дерева A
C – корень правого поддерева дерева A
D, G, H, I – листья
B – левый потомок дерева A
C – правый потомок дерева A

В программе на Паскале дерево представляется в виде иерархического списка, элемент которого (изображающий узел дерева) можно рассматривать как структуру следующего вида:



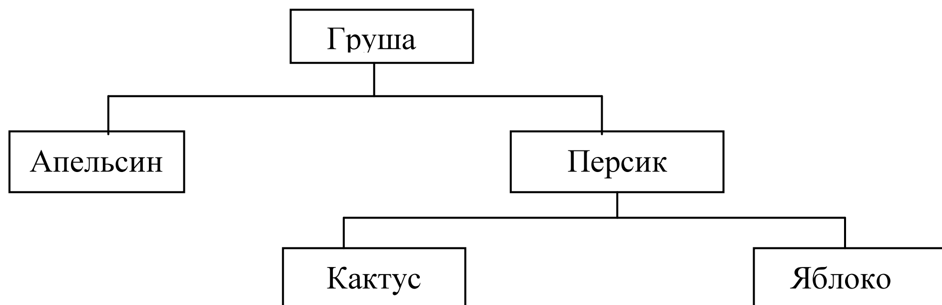
Здесь

Info – поле, содержащее полезную информацию

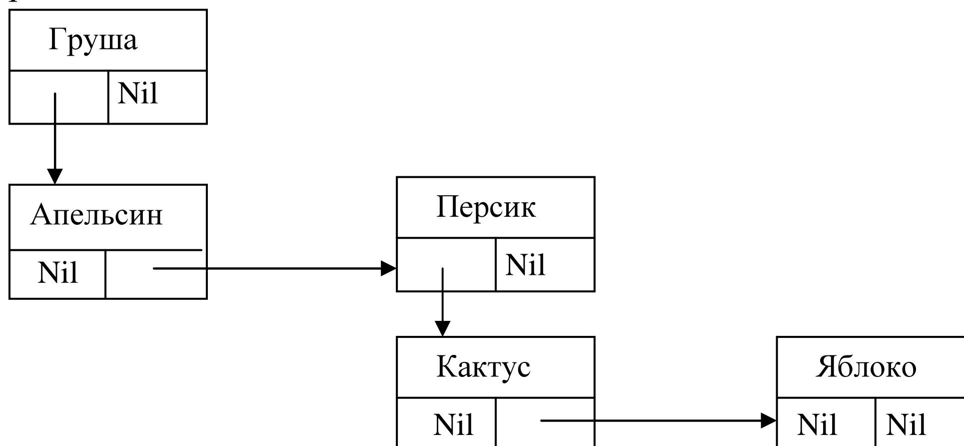
Child – ссылка на элемент-потомок

Next – ссылка на элемент того же уровня (на элемент, являющийся *братом* данного).

Ниже приведен пример упорядоченного дерева в виде графа и в виде иерархического списка:



Это же дерево в программе можно представлено в виде следующей структуры:



Бинарные деревья (как упорядоченные, так и нет) широко используются в программировании. По этой причине мы сосредоточимся в основном на рассмотрении процедур работы с ними.

Для описания узла бинарного дерева в программе введем тип, имеющий вид следующей записи:

```
type
    PNode = ^TNode;
    TNode = record
        Info : string; {поле данных}
        Left,Right : PNode; {указатели на левое и
                             правое поддеревья}
    end;
```

Приведем несколько процедур, иллюстрирующих работу с деревьями.

Пример 1. Процедура создания нового узла.

```
{ Создание нового узла со значением информационного поля X.
 Возвращается указатель на новый узел}
function NewNode(X:string):PNode;
var
   P : PNode;
begin
   New(P);
   P^.Info := X;
   P^.Left := Nil;
   P^.Roght := Nil;
   NewNode := P;
end;
```

Пример 2. Процедура создания потомка (поддерева).

```
{Процедура создания левого потомка для узла P}
procedure SetLeft(P:PNode; X:string);
begin
   P^.Left := NewNode(X);
end;
```

Пример 3. Создание бинарного дерева:

- данные (целые числа) заносятся с клавиатуры;
- дубликаты не включаются (но выводятся на экран);
- признаком окончания ввода является ввод числа 0;
- результат – бинарное упорядоченное дерево.

```
Program BuildTree;
Type
   PNode = ^TNode;
   TNode = record
      key : integer;
      left,right : PNode;
   end;
var
   Root:PNode; {корень дерева}
   NewData:PNode;{новый узел}
   n:integer;
procedure Add(NewData:PNode;var Root:PNode);
begin
   if Root=Nil then  {дерево пустое}
      Root := NewData
   else begin
      if NewData^.key < Root^.key then
         Add(NewData,Root^.left)
      else if NewData^.key > Root^.key then
         Add(NewData,Root^.right)
      else begin {дубликат}
         writeln(' Дубликат ', NewData^.key );
         Dispose(NewData);
      end;
   end;
end; {Add}
begin {main program}
   Root := Nil; {инициализировать пустое дерево}
   repeat
      write('Введите число : ',n);
      readln(n);
      if n<>0 then begin
         NewData := NewNode(n); {функция создания узла
                                 приведена выше}
         Add(NewData,Root);
      end;
   until n=0;
   . . .{здесь можно поместить операторы, в которых
         созданное дерево используется для решения
         конкретной задачи}
end.
```

13.2. Рекурсия

При обработке динамических структур типа «дерево» чаще всего используются рекурсивные алгоритмы.

Рекурсия предполагает определение некоторого понятия с использованием самого этого понятия. Никлаус Вирт приводит следующее утверждение о возможностях рекурсии:

«… мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания».

Примером рекурсивного определения является факториал неотрицательного числа:

$$n! = 1 \quad \textit{при } n=0$$
$$n! = n*(n-1)! \quad \textit{при } n>0$$

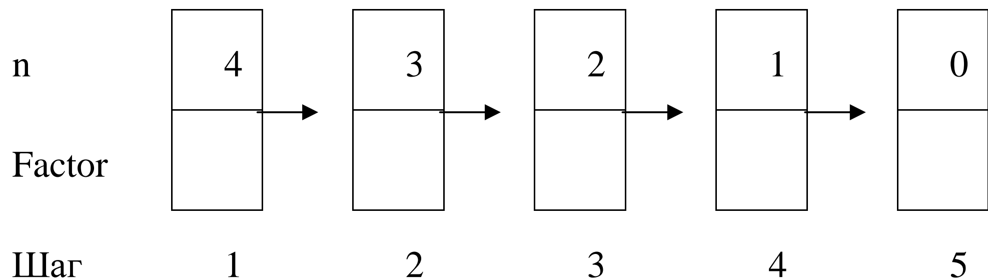
В Турбо-Паскале рекурсия разрешена: подпрограмма может вызывать сама себя:

```
{Демонстрация вычисления функции факториала}
program FactDemo;
var
   k : integer;
function Factor(n:integer):integer;
begin
   if n=0 then
      Factor : 1
   else
      Factor := n * Factor(n-1);
   {end if}
end;
{====== текст программы: ====}
begin
   write('Введите целое число ');
   readln(k);
   if k>=0 then
      writeln('Факториал',n:1,'=',Factor(k));
   {end if}
end.
```

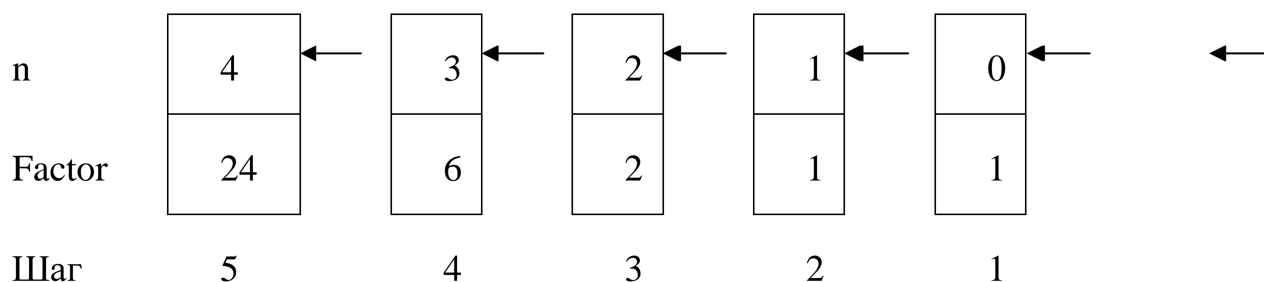
С помощью выражения `Factor(n-1)` приведенная функция `Factor` будет вызывать сама себя до тех пор, пока значение ее входного параметра не станет равным нулю. При каждом повторном вызове функции `Factor` создается новый экземпляр памяти для всех локальных переменных и для значения самой функции, который запоминается в стеке.

Продемонстрируем это на примере вычисления факториала от 4.

Вначале изобразим процесс развертывания рекурсии:



Первое значение `Factor` вычисляется, когда n=0, оно подставляется в соответствующий экземпляр памяти. Теперь на каждом очередном шаге значения всех членов выражения `n*Factor(n-1)` известны, и алгоритм подставляет в это выражение значение, полученное на предыдущем шаге. Это происходит в процессе свертывания рекурсии:



Сформулируем два важных свойства рекурсивных алгоритмов:

1. Наличие тривиального случая.

Правильный рекурсивный алгоритм не должен создавать бесконечную последовательность вызовов самого себя. Для этого он обязательно должен содержать нерекурсивный выход: т.е. при некоторых входных данных вычисления в алгоритме должны производиться без вызовов его самого.

Для функции «факториал» тривиальный случай: «0!=1».

2. Определение сложного случая в терминах более простого.

При любых входных данных нерекурсивный выход должен достигаться за конечное число рекурсивных вызовов. Для этого каждый новый вызов рекурсивного алгоритма должен решать более простую задачу. Иными словами рекурсивный алгоритм должен содержать определение некоторого сложного случая в терминах более простого случая.

Для функции «факториал» вместо вычисления n! заменяется умножением n*(n-1)!, и при этом с каждым вызовом значение n уменьшается, стремясь к нулю и достигая его за конечное число вызовов.

Из определения структуры типа «дерево» видно, что она рекурсивна по определению, а в силу этого рекурсивными являются и практически все алгоритмы работы с деревьями. Для этого достаточно посмотреть на приведенный выше пример создания бинарного упорядоченного дерева. В процедуре `Add` имеется тривиальный случай (когда дерево пустое) и рекурсивные вызовы: добавить в левое и правое поддеревья - `Add(NewData,Root^.left)` и `Add(NewData,Root^.right)`.

Рассмотрим ряд алгоритмов работы с деревьями, обращая при этом внимание также на правильность их построения, как рекурсивных алгоритмов.

13.3. Алгоритмы работы с деревьями

В приведенных ниже алгоритмах предполагается, что узел (элемент) дерева декларирован следующей записью:

```
Type
    PNode = ^TNode;
    TNode = record
       Data : integer; {информационное поле}
       left,right : PNode;
    end;
```

А1. Вычисление суммы значений информационных полей элементов

Алгоритм реализован в виде функции, возвращающей значение суммы информационных полей всех элементов. Тривиальным считается случай, когда очередной узел – пустой, и, следовательно, не имеет информационного поля.

```
function Sum(Root : PNode) : integer;
begin
   if Root=Nil then {узел - пустой}
      Sum := 0
   else
      Sum := Root^.Data + Sum(Root^.left)
                        + Sum(Root^.right);
   {end if}
end;
```

Для нетривиального случая результат вычисляется как значение информационного элемента в корне (`Root^.Data`) плюс суммы информационных полей левого и правого поддеревьев.

А выражение `Sum(Root^.left)`представляет собой рекурсивный вызов левого поддерева для данного корня `Root`.

А2. Подсчет количества узлов в бинарном дереве

```
function NumElem(Tree:PNode):integer;
begin
   if Tree = Nil then
      NumElem := 0
   else
      NumElem := NumElem(Tree^.left)
               + NumElem(Tree^.right) + 1;
   {end if}
end;
```

А3. Подсчет количества листьев бинарного дерева

```
function Number(Tree:PNode):integer;
begin
   if Tree = Nil then
          Number := 0     {дерево пустое - листов нет}
   else if (Tree^.left=Nil) and (Tree^.right=Nil) then
      Number := 1  {дерево состоит из одного узла - листа}
   else
      Number := Number(Tree^.left) + Number(Tree^.right);
   {end if}
end;
```

Анализ приведенных алгоритмов показывает, что для получения ответа в них производится просмотр всех узлов дерева. Ниже будут приведены алгоритмы, в которых порядок обхода узлов дерева отличается. И в зависимости от порядка обхода узлов бинарного упорядоченного дерева, можно получить различные результаты, не меняя их размещения.

Примечание: Просмотр используется не сам по себе, а для обработки элементов дерева, а просмотр сам по себе обеспечивает только некоторый порядок выбора элементов дерева для обработки. В приводимых ниже примерах обработка не определяется; показывается только место, в котором предлагается выполнить обработку текущего.

А4. Алгоритмы просмотра дерева

Самой интересной особенностью обработки бинарных деревьев является та, что при изменении порядка просмотра дерева, не изменяя его структуры, можно обеспечить разные последовательности содержащейся в нем информации. В принципе возможны всего четыре варианта просмотра: слева-направо, справа-налева, сверху-вниз и снизу-вверх. Прежде чем увидеть, к каким результатам это может привести, приведем их.

<u>а. Просмотр дерева слева – направо</u>

```
procedure ViewLR(Root:PNode); {LR -> Left - Right }
begin
   if Root<>Nil then
   begin
      ViewLR(Root^. left); {просмотр левого поддерева}
      {Операция обработки корневого элемента -
          вывод на печать, в файл и др.}
      ViewLR(Root^.right); { просмотр правого поддерева }
   end;
end;
```

б. Просмотр справа налево

```
procedure ViewRL(Root:PNode); {LR -> Right - Left}
begin
   if Root<>Nil then
   begin
      ViewRL(Root^.right); {просмотр правого поддерева}
      {Операция обработки корневого элемента -
          вывод на печать, в файл и др.}
      ViewRL(Root^.left); { просмотр левого поддерева }
   end;
end;
```

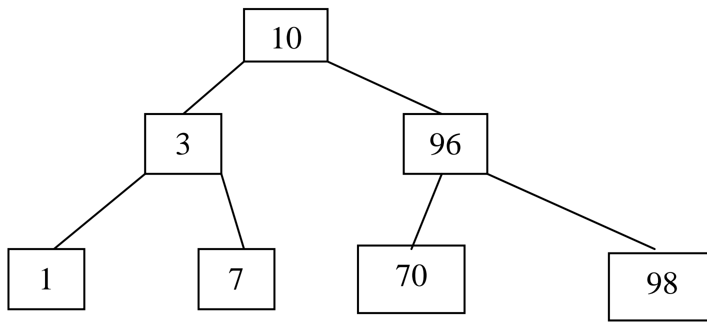
в. Просмотр сверху – вниз

```
procedure ViewTD(Root:PNode); {TD -> Top-Down}
begin
   if Root<>Nil then
   begin
     {Операция обработки корневого элемента -
          вывод на печать, в файл и др.}
      ViewTD(Root^.left); {просмотр левого поддерева}
      ViewTD(Root^.right); { просмотр правого поддерева }
   end;
end;
```

г. Просмотр снизу-вверх

```
procedure ViewDT(Root:PNode); {DT -> Down - Top}
begin
   if Root<>Nil then
   begin
      ViewDT(Root^.left); {просмотр левого поддерева}
      ViewDT(Root^.right); { просмотр правого поддерева }
      {Операция обработки корневого элемента -
          вывод на печать, в файл и др.}
   end;
end;
```

Пример 1. Рассмотрим результаты просмотра для приведенных алгоритмов, при условии, что обработка корневого элемента сводится к выводу значения его информационного поля, а дерево в этот момент имеет следующие узлы:

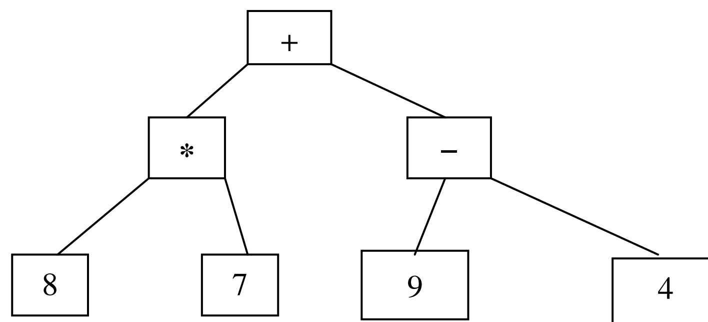


Результаты просмотра:

Алгоритм «Слева направо»	1, 3, 7, 10, 70, 96, 98
Алгоритм «Справа налево»	98, 96, 70, 10, 7, 3, 1
Алгоритм «Сверху вниз»	10, 3, 1, 7, 96, 70, 98

Из приведенной таблицы видно, что, просто изменяя порядок просмотра дерева (слева-направо и справа-налево), можно получить отсортированные по возрастанию или по убыванию числа.

Пример 2. Пусть в узлах дерева расположены элементы арифметического выражения:



Результаты просмотра:

«Слева направо»	8 * 7 + 9 – 4	инфиксная форма записи выражения
«Сверху вниз»	+ * 8 7 – 9 4	префиксная форма записи выражения
«Снизу вверх»	8 7 * 9 4 – +	постфиксная форма записи выражения

A5. Поиск элемента в двоичном упорядоченном дереве

```
{ Определить существование значения SearchValue
  и вернуть указатель на элемент, содержащий его,
  или вернуть Nil, если элемент не найден}
function Search(SearchValue:integer;Root:PNode):PNode;
begin
   if (Root=Nil) or (Root^.Data=SearchValue) then
      Search := Root
   else if (Root^.Data > SearchValue) then
      Search := Search(SearchValue,Root^.left)
   else
      Search := Search(SearchValue,Root^.right);
   {end if}
end;
```

Вывод. Тексты приведенных алгоритмов очень компактны и просты в понимании.

В заключение отметим, что рекурсивные алгоритмы широко используются в базах данных и при построении компиляторов, в частности для проверки правильности записи арифметических выражений, синтаксис которых задается с помощью синтаксических диаграмм.

Для закрепления материала предлагается решить следующую задачу: Данные о студентах содержат фамилию и три оценки, полученные на экзаменах. Занести их с клавиатуры или из текстового файла в бинарное дерево поиска, упорядоченное по значению средней оценки. Затем вывести на экран список студентов, упорядоченный по убыванию средней оценки. Кроме фамилий вывести все три оценки и их среднее значение с точностью до одного знака после запятой.

Вопросы для самоконтроля

1. Приведите определение структуры «дерево».
2. Какие виды деревьев были определены?
3. Что такое «бинарное дерево»?
4. Что такое «упорядоченное дерево»?
5. В чем отличие представления на Паскале бинарного дерева и дерева с числом потомков, большим двух?
6. Приведите определение и свойства рекурсивного алгоритма.
7. Как изменяются результаты при изменениях порядка просмотра упорядоченного бинарного дерева?

14. МОДУЛИ

14.1. Введение

Основным принципом модульного программирования является принцип "разделяй и властвуй". *Модульное программирование* –это организация программы как совокупности небольших независимых блоков, называемых *модулями,* структура и поведение которых подчиняются определенным правилам. Заметим, что нужно различать использование слова "модуль" когда имеется в виду синтаксическая конструкция языков программирования (unit в Турбо Паскале), и когда имеется в виду единица дробления большой программы на отдельные блоки (которые могут быть реализованы и в виде процедур, и в виде функций).

Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

Упрощается процесс повышения эффективности программ, так как критичные по времени модули могут многократно переделываться независимо от других. Кроме того, модульные программы значительно легче понимать, а модули могут использоваться как строительные блоки и в других программах.

Термин *модуль* в программировании начал использоваться в связи с внедрением модульных принципов при создании программ. В 70-х годах под модулем понимали какую-либо процедуру или функцию, написанную в соответствии с определенными правилами. Например так:

"Модуль должен быть простым, замкнутым (независимым), обозримым (от 50 до 100 строк), реализующим только одну функцию задачи, имеющим только одну входную и только одну выходную точку".

Однако общепризнанных требований не было, и модулем очень часто называли любую процедуру размером до 50 строк.

Первым основные свойства программного модуля более-менее четко сформулировал Парнас (Parnas): *"Для написания одного модуля должно быть достаточно* ***минимальных*** *знаний о тексте другого".*

В соответствии с этим определением, модулем могла быть любая отдельная процедура (функция) как самого нижнего уровня иерархии (уровня реализации), так и самого верхнего уровня, на котором происходят только вызовы других процедур-модулей.

Таким образом, Парнас первым выдвинул концепцию сокрытия информации в программировании.

Однако существующие в языках 70-х годов только такие синтаксические конструкции, как процедура и функция, не могли обеспечить *надежного* сокрытия информации, поскольку подвержены влиянию гло-

бальных переменных, поведение которых в сложных программах зачастую бывает трудно предсказуемым.

Решить эту проблему можно было, только разработав *новую синтаксическую конструкцию, которая не подвержена влиянию глобальных переменных.*

Такая конструкция была создана и названа *модулем.* Изначально предполагалось, что при реализации сложных программных комплексов модуль должен использоваться наравне с процедурами и функциями как конструкция, *объединяющая* и *надежно скрывающая* детали реализации определенной подзадачи.

Отсюда, количество модулей в комплексе должно определяться декомпозицией поставленной задачи на независимые подзадачи. В отдельном случае модуль может использоваться даже для включения в него всего лишь одной процедуры, если необходимо, чтобы выполняемое ею локальное действие было *гарантированно независимым* от влияния других частей программы при любых изменениях и коррекциях. Такое использование модуля характерно для класса задач реального времени, в которых критерий надежности и предсказуемости поведения программы является ключевым.

Впервые специализированная синтаксическая конструкция модуля была предложена Н.Виртом в 1975 г. и включена в его новый язык Модула. В этом же году была сделана опытная реализация языка Modula.

После некоторой переработки этот новый язык был окончательно реализован в 1977 г. и получил название Modula-2. Впоследствии, аналогичные конструкции, с некоторыми отличиями, были включены и в другие языки программирования: Pascal Plus (Уэлш и Бастард, 1979 г.), Ada *(1980),* Turbo Pascal, начиная с версии 4.0. В настоящее время средства поддержки модульного программирования реализованы практически во всех современных языках процедурного программирования, например, таких как C++ и Delphi.

14.2. Форма модульной программы

Организация программы в виде иерархии модулей позволяет улучшить процесс ее разработки. При этом следует иметь в виду следующее: число модулей, которые вызываются каким-либо модулем, и число модулей, которые его вызывают, оказывают влияние на сложность программы. Йодан (Yourdon) назвал число модулей, вызываемых из данного модуля, *размахом или шириной* управления модулями. Наряду с большим размером модуля, очень маленькая или очень большая ширина управления является признаком плохой схемы разбивки на модули.

Как показывает опыт, ширина управления модуля не должна превышать 10-ти. Это число связано с "магическим" числом 7, которое базируется на положениях психологии и, в особенности, на теории "кусков" ("chunking") информации. Кратковременная память человека имеет ог-

раниченные способности сохранения "кусков" информации. Психологические эксперименты показали, что способность нашей кратковременной памяти находится в пределах 5-9 "кусков" (в среднем – 7). Она может одновременно оперировать около 7 "кусками" информации. Когда человек превышает этот предел, он более склонен к ошибкам.

Реорганизация информации с разбивкой на подходящие части важна для эффективного использования кратковременной памяти человека и для улучшения понимаемости материала. Во многих жизненных ситуациях люди делают такую реорганизацию бессознательно. Однако программист может сам себе помочь, сознательно не допуская ширины управления модулями, которая превышает число семь.

14.3. Стандарты структурного программирования

Программа, выполненная в соответствии со стандартами структурного программирования, должна удовлетворять следующим рекомендациям:

1. Программа должна разделяться на независимые части, называемые модулями.
2. Модуль – это независимый блок, код которого физически и логически отделен от кода других модулей.
3. Модуль выполняет только одну логическую функцию.
4. Размер модуля не должен превышать 100 операторов.
5. Модуль имеет одну входную и одну выходную точку.
6. Взаимосвязи между модулями устанавливаются по иерархической структуре.
7. Каждый модуль должен начинаться с комментария, объясняющего
   - его назначение,
   - назначение переменных, передаваемых в модуль и из него,
   - модулей, которые его вызывают, и
   - модулей, которые вызываются из него.
8. Следует избегать ненужных меток и вообще не использовать оператор goto. В виде исключения этот оператор можно использовать только для переходов на выходную точку модуля.
9. Идентификаторы всех переменных и модулей должны быть смысловыми.
10. Родственные группы идентификаторов должны начинаться с одинакового префикса.
11. Следует использовать только стандартные управляющие конструкции (выбор, цикл, выход, блок).
12. В одной строке записывать не более одного оператора. Если для записи оператора требуется больше, чем одна строка, то все последующие строки записываются с отступами вправо.
13. Не допускать вложенности операторов if более 3-х уровней.

Примечание. Приведенные рекомендации были сформулированы уже более двадцати лет назад. С тех пор, как уже упоминалось выше, средства языков программирования, ориентированные на обеспечение возможностей построения программ по модульному принципу, значительно усовершенствовались. И сегодня в модуль, как правило, помещается не одна подпрограмма, а несколько, логически связанных. При этом в модуле стало возможным помещать не только подпрограммы, но и объявление типов, констант и переменных, специфических только для них. Например, в составе стандартных модулей системы Турбо Паскаль, имеется модуль Graph, содержащий весь спектр подпрограмм управления выводом на экран монитора информации в графическом режиме, а также и некоторый набор констант и переменных, используемых, почти исключительно, этими подпрограммами.

14.4. Модули в турбо-паскале

Турбо Паскаль содержит средства для организации модульного программирования. Эти средства позволяют помещать в других файлах подпрограммы, необходимые для реализации программы (которая размещается в отдельном файле). Для обеспечения возможности использовать их в программе, файлы должны быть оформлены специальным образом – как модули Турбо Паскаля.

Модуль Турбо Паскаля представляет собой совокупность программных ресурсов, предназначенных для использования другими модулями и программами. Все программные ресурсы можно разбить на две части:

- объекты, предназначенные для использования другими программами или модулями, и
- объекты рабочего характера, используемыми в теле модуля.

В соответствии с этим модуль, кроме заголовка, имеет две основные части, называемые *интерфейсом* и *реализацией*. В интерфейсной части модуля сосредоточены описания объектов, доступных из других программ. В части реализации помещаются рабочие объекты Турбо Паскаля.

Заголовок модуля составляется из служебного слова **`Unit`** и следующего за ним идентификатора, являющегося именем модуля. Заголовок завершается символом ";".

Интерфейсная часть начинается словом **`Interface`**, за которым следует совокупность обычных объявлений (список используемых модулей, описание констант, типов, переменных, заголовки процедур и функций).

Часть реализации начинается служебным словом **`Implementation`**, за которым идут описание скрытых объектов (описание внутренних констант, типов, реализация процедур и функций). Завершает модуль, как и программу, служебное слово end и символ "." (точка).

Кроме перечисленных частей, модуль может содержать так называемый **раздел инициализации**, предназначенный для задания начальных значений переменных модуля перед его использованием. Этот раздел помещается после раздела реализации, начинается со служебного слова begin и содержит последовательность операторов.

Общая структура модуля может быть представлена следующей схемой:

```
Unit <Имя модуля>;
Interface
   <Описание видимых объектов>
Implementation
  <Описание скрытых объектов>
begin
  <Операторы инициализации объектов модуля>
end.
```

Примечания.

1. Имя модуля должно быть построено по правилам Паскаля и не превышать по длине восьми символов, поскольку в среде MS DOS, имя, под которым модуль должен быть сохранен на диске, должно точно совпадать с именем модуля.
2. Если в тексте модуля не требуется инициализировать объекты модуля, можно опустить и ключевое слово begin.

Описание видимых объектов (в разделе Interface) может включать

- объявление констант, типов и переменных, которые могут быть использованы в других модулях
- заголовки процедур и функций, которые могут быть использованы в других модулях
- имена других модулей, видимые объекты которых будут использоваться в данном модуле.

Примечание. Напомним, что любой объект программы может быть объявлен только один раз. Поэтому, если некоторый объект объявлен в одном из модулей, то, поместив его имя в операторе Uses другого модуля, можно использовать его так, как будто он объявлен в данном.

Описание скрытых объектов (в разделе Implementation) включает полные тексты подпрограмм, заголовки которых приведены в интерфейсной части модуля, и объекты, необходимые для их реализации: объявления констант, типов, переменных, а также подпрограммы, которые необходимы для реализации, но такие, область действия которых локализована в данном модуле.

14.5. Использование модулей

При необходимости воспользоваться информацией из других модулей, в программе необходимо поместить специальный оператор Uses (использовать), в котором перечисляются через запятую имена этих модулей. Как уже говорилось выше, необходимо, чтобы имена модулей совпадали с именами файлов, в которых они размещаются. Оператор Uses должен располагаться в самом начале раздела деклараций.

Оператор Uses может быть помещен:

- в тексте программы непосредственно после заголовка
- в разделе Interface модуля Unit
- в разделе Implementation модуля Unit.

Пример: Uses CRT, Graph;

14.6. Стандартные модули Турбо-Паскаля

В Турбо-Паскале имеется некоторый набор модулей, устанавливаемых в процессе инсталляции по умолчанию. Наиболее часто используемые из них: DOS, CRT, GRAPH. Приведем краткие сведения о них.

Модуль DOS содержит информацию, обеспечивающую возможность:

- работать с файлами (чтение, поиск и т.д.);
- запрашивать из программы текущие время и дату;
- узнавать размер свободной памяти на диске.

Модуль CRT обеспечивает управление персональным компьютером при работе:

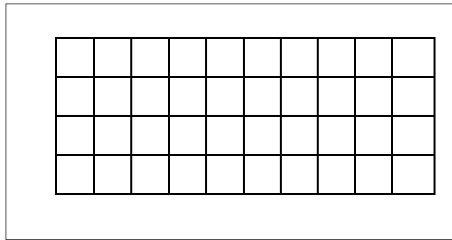
- с экраном монитора,
- со звуком,
- с клавиатурой и др.

Модуль GRAPH – содержит широкий набор процедур и функций, обеспечивающих вывод информации на экран в графическом режиме.

Во встроенном справочнике системы Турбо Паскаль и в любой книге по Паскалю приводятся полный перечень «стандартных» модулей и полные сведения о функциях каждого модуля.

Здесь мы рассмотрим несколько более подробно только функции модуля CRT, обеспечивающие вывод на экран информации в текстовом режиме.

Прежде всего, учтем, что с точки зрения стандартного Турбо Паскаля экран монитора при работе в текстовом режиме представляется матрицей *знакомест*:



В одном знакоместе может быть размещен только один символ (в том числе и символ пробела). Каждое знакоместо характеризуется, кроме символа, еще цветом символа и цветом фона, на котором он выделяется. В одном из знакомест размещается также так называемый **курсор**. Вывод символа на экран всегда производится в позицию курсора, после чего курсор перемещается вправо, в соседнее знакоместо. Из самого правого знакоместа в строке курсор перемещается в начало следующей строки. Если это происходит с последней строкой экрана, производится скроллинг вверх. При этом самая первая строка экрана исчезает, а все остальные – смещаются вверх. В этом случае курсор оказывается в самом начале последней (новой) строки.

По инициативе программы монитор может быть переведен в несколько различных текстовых режимов, которые отличаются количеством знакомест и цветов.

В модуле CRT объявлены константы, определяющих режимы монитора:

- константы, определяющие число знакомест (наиболее распространенный режим 25 x 80 – 25 строк по 80 знакомест в строке);
- константы, определяющие цветовую палитру: монохромный (черно-белый) режим или цветной (16 цветов – от черного до белого)
- и ряд других.

Цвета. В текстовом режиме (для цветного монитора) предусмотрено 16 цветов. Каждому цвету сопоставлена константа, а объявление цветов размещается в интерфейсной части модуля.

```
const
      Black=0;
      Blue=1;
      . . . .
      White=15;
```

Цвет фона может принимать значения восьми первых (от 0 до 7) цветов, а цвет символа – всех шестнадцати:

Процедуры определения (задания в программе) цвета:

- цвета символа:
  ```
  procedure TextColor(Color:Byte);
  ```
- цвета фона:
  ```
  procedure TextBackGround(Color:Byte);
  ```

Процедуры яркости вывода символов:

```
Procedure HighVideo;
```

```
Procedure LowVideo;
Procedure NormVideo;
```

Очистка экрана:

`Procedure ClrScr` - полностью;

`Procedure ClrEol` - до конца текущей строки от позиции курсора.

Управление курсором:

`Function WhereX:Byte;` - вернуть значение X-координаты курсора

`Function WhereY:Byte;` - вернуть значение Y-координаты курсора

`Procedure GoToXY(X:Byte;Y:Byte);` - переместить курсор в (X,Y)

`Procedure Window(X1,Y1,X2,Y2:Byte);` - объявить окно в виде прямоугольника с координатами (`X1,Y1`) для верхнего левого знакоместа и (`X2,Y2`) для правого нижнего знакоместа. После выполнения данной процедуры вывод на экран будет проводиться только внутри данного окна. При этом левое верхнее знакоместо будет иметь координаты (1,1) , а нижнее правое – координат (`X2-X1+1, Y2-Y1+1`). При этом все знакоместа вне пределов данного окна будут оставаться неизменными.

Клавиатура:

Для обработки нажатой клавиши:

`Function ReadKey:Char` - если нажата ASCII- клавиша, то функция возвращает символ, а если нажата функциональная клавиша, то функция возвратит значение 0. Чтобы считать код символа при повторном вызове существует

`Function KeyPressed:Boolean` - значение истинно, если нажата какая-либо клавиша.

Звук:

`Procedure Sound(Hz:Word);` – вывести на встроенный динамик синусоидальный сигнал с частотой Hz (Гц).

`Procedure NoSound;` – выключить динамик.

`Procedure Delay(Ms:Word)` - задержка на определенный интервал времени.

14.7. Пример использования модулей

В качестве примера, иллюстрирующего использование модуля, воспользуемся программой работы с базой данных, рассмотренной выше (см. раздел 10.8). Текст, разработанной там программы предложено оформить как «монопрограмму», включив в неё, в разделе объявлений, все подпрограммы, реализующие отдельные операции с базой.

Применение модулей позволит разбить единственный файл на несколько файлов, поместив в один из них основную программу, а в другие файлы (один или несколько по желанию программиста) - подпрограммы, размещенные в модулях Турбо Паскаля. Ниже приведен один из возможных вариантов такой организации программы.

Основной алгоритм (файл первый с именем ‘base.pas’)

```
Program Base; {Программа работы с простейшей базой данных.
          Содержимое базы данных в памяти хранится в массиве
         записей, а на диске -  в типизированном файле      }
Uses CRT,
     MyProc,  {Подключить процедуры работы с базой в памяти}
     MyFile;  {Подключить модуль работы с файлом}
Var
    OperCode:char; {код выбранной операции}
procedure ShowMenu;
begin
   writeln('----------------------------------------------');
   writeln('Операции базы данных:');
   writeln('  добавить запись (1)');
   writeln('  просмотреть содержимое (2)');
   writeln('  искать неуспевающих (3)');
   writeln('  вывести в текстовый файл (4)');
   writeln('  сохранить базу (5)');
   writeln('  конец работы (6)');
   write('Введите номер операции _');
end;

begin
   clrscr;  {очистить содержимое экрана. }
   OpenBase;  {восстановить базу из файла 'group.dat'}
   repeat
      ShowMenu;              {вывод меню}
      Readln(OperCode);  {читать код клавиши}
       case OperCode of
       '1' : AddStudent;    {добавить запись в базу}
       '2' : View;          {отображение содержимого базы}
       '3' : Search;        {поиск и отображение двоечников}
       '4' : ToFile;        {вывод в текстовый файл}
       '5' : SaveBase; {сохранение базы в файле 'group.dat'}
      end;
   until OperCode='6';
end.
```

Модуль работы с файлами (хранится в файле 'myfile.pas').

```
Unit MyFile;
Interface
    Const
   MaxStudents = 10; {максимальное число записей в базе.}
    Type
   TStudent = record {запись со сведениями об одном студенте}
        Name : string[25];  {фамилия}
        Estimations:array[1..3] of 2..5;  {оценки}
   end;
    Var
   Students : array[1.. MaxStudents] of TStudent;
            {текущая база в оперативной памяти}
   NStud : integer;  {фактическое число записей в базе}

procedure OpenBase;  {Загрузить базу в оперативную память}
procedure SaveBase;  {Сохраненить базу в файле 'group.dat'}
procedure ToFile; {вывод в текстовый файл}

Implementation
procedure OpenBase;
var
   n:integer;
begin
   <здесь поместить текст процедуры>
end; {OpenBase}

procedure SaveBase; {сохранение базы в файле 'group.dat'}
var
   k:integer;
begin
   <здесь поместить текст процедуры>
end;

procedure ToFile; {вывод в текстовый файл}
var
   namefile:string; {имя текстового файла}
   textfile:text;   {файловая переменная текствого файла}
   k:integer;
begin
   <здесь поместить текст процедуры>
end;

end.
```

Модуль работы с базой в оперативной памяти (в файле 'myproc.pas').

```
Unit MyProc;
Interface
Uses
     MyFile; {подключить для использования переменных
                  NStud и Students, объявленных в
                  разделе Interface этого модуля        }
procedure AddStudent; {добавить запись в массив Students}
procedure View;       {отобразить содержимое базы}
procedure Search;     {поиск и отображение двоечников}

Implementation

procedure Search; {поиск и отображение двоечников}
var
   k:integer;
   n:integer; {число неуспевающих}
begin
   <здесь поместить текст процедуры>
end; {}

procedure View; {отображение содержимого базы}
var
   k:integer;
begin
   <здесь поместить текст процедуры>
end;

procedure AddStudent; {Добавить новую запись
                            в массив Students}
var
   k:integer;
   estimate:integer;
   exist:boolean;
begin
   <здесь поместить текст процедуры>
end;

end.
```

Как видно из текста примера, программа, разбитая на три файла, стала более наглядной, а выделение процедур в отдельные файлы позволяет проводить отладку процедур по частям, что упрощает процесс отладки. Разумеется, в этом случае может потребоваться написать отладочные программы (одну или несколько). Но это намного проще и удобнее для отладки. А уже после того, как отдельные процедуры будут гарантированно правильно работать, можно использовать их в главной программе.

Вопросы для самоконтроля

1. Какие проблемы возникают при создании больших программных комплексов?
2. Что такое структурная методология?
3. Каковы цели структурного программирования?
4. Каковы основные принципы структурной методологии?
5. В чем состоит суть принципа абстракции?
6. В чем заключается принцип формальности?
7. В чем заключается принцип "разделяй и властвуй"?
8. В чем заключается принцип иерархического упорядочения?
9. Что такое модульное программирование?
10. В чем преимущества модульного программирования?
11. В чем суть концепции скрытия информации в программировании?
12. Какова структура и назначение отдельных частей модуля Unit в Турбо Паскале?
13. Что помещается в интерфейсной секции?

15. ОСНОВЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

15.1. Основные понятия

Объектно-ориентированное программирование (ООП) – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является реализацией определенного ***класса*** (типа особого вида), причем классы образуют иерархию на принципах наследуемости. Под термином ***объект*** понимается «осязаемая сущность, которая четко проявляет свое поведение».

Объект характеризуется

- совокупностью всех своих свойств и их текущих значений и
- совокупностью допустимых для него действий (их называют *методами*)

В качестве примера объекта можно привести животное, свойствами которого могут быть голова (большая, маленькая, …), уши (длинные, короткие, … или другие), а методами (умение принимать пищу, стоять, сидеть, идти, бежать и т.п.).

Объектно-ориентированная методология программирования базируется на следующих принципах:

- инкапсуляция;
- наследование;
- полиморфизм.

Инкапсуляцией называется объединение в одном объекте как свойств, так и методов (действий над ними)**.**

Наследование предполагает существование иерархии классов (объектных типов), в которой между классами, расположенными на разных уровнях иерархии, имеют место отношения наследования свойств и методов - от объектов вышележащих (*родительских*) к объектам нижележащих (*дочерних*) классов.

Наследование можно определить так:

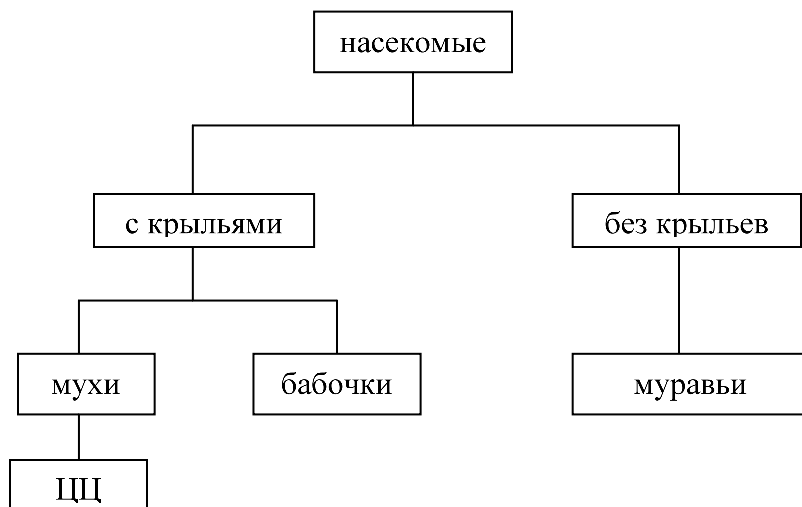
> *Наследование* – это такое отношение между объектами, когда один объект повторяет структуру и поведение другого, принадлежащего к более высокому уровню в иерархии.

Родительские классы называют просто родителями (*прародителями*), а дочерние – *потомками*.

Классы верхних уровней иерархии, как правило, не имеют конкретных экземпляров объектов. Не существует, например, конкретного живого организма (объекта), который бы сам по себе назывался «Млекопитающее» или «Насекомое». Такие классы называются *абстрактными*.

Конкретные экземпляры объектов, или просто объекты, принадлежат нижележащим уровням иерархии.

Ниже приведен пример иерархии для насекомых:



Полиморфизм – это способность объектов разных классов в иерархии выполнять одно и то же действие по-своему. (Или иначе, полиморфизм – это придание одного и того же имени методам для объектов разных классов в иерархии, хотя действия в методах различаются). При объектно-ориентированном программировании программист только указывает, какому объекту, какое из присущих ему действий требуется выполнить, и, однажды объявленные (описанные в программе), объекты сами будут выполнять их характерными именно для них способами.

Например, действие «бежать» свойственно большинству животных. Однако каждое из них (лев, слон, черепаха) выполняет это действие различным образом. В данном примере действие «бежать» будет называться полиморфическим действием, а многообразие форм проявления этого действия – полиморфизмом.

Рассмотрим средства языка Турбо Паскаль для объектно-ориентированного программирования.

15.2. Объявление классов объектов

Объявление (описание) класса состоит из двух частей – из объявления перечня свойств и методов и описания реализации каждого метода. Полную информацию о синтаксисе объявления класса можно найти в литературе. Мы приведем только общие сведения.

Для объявления перечня свойств и методов класса используется структурная запись следующего вида:

```
type
    <Имя-класса>=object(<имя-родительского-класса>)
        <объявление-свойств>
        <объявление-методов>
      end;
```

Как видно из этой записи, объявление класса помещается в секции Type. При этом свойства объекта задаются в программных переменных, и, следовательно, объявление свойств сводится к объявлению переменных.

Объявление методов в этой «структуре» представляется заголовками подпрограмм, реализующих отдельные методы. При этом в качестве метода может выступать как процедура (**procedure**), так и функция (**function**). Класс самого верхнего уровня (прародитель всех остальных в иерархии) не имеет родительского класса, и поэтому в его объявлении после слова **object** не записывается ничего.

Любой объект из класса является динамическим, то есть имеет конечный срок «жизни»:

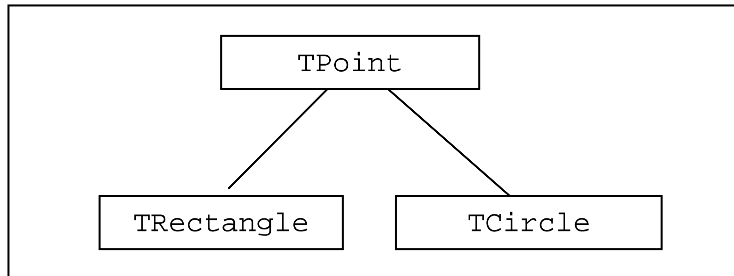
- объект должен быть создан во время исполнения программы с помощью специального метода (для этого метода вместо ключевого слова *procedure* введено особое ключевое слово - **constructor**),
- объект должен быть уничтожен, также во время исполнения программы, и также специальным методом (для этого метода вместо ключевого слова *procedure* введено особое ключевое слово - **destructor**),
- между этими двумя моментами времени объект существует и может выполнять прочие методы, определенные для него в объявлении класса, к которому он принадлежит.

В дополнение к приведенному описанию, ниже в текст программы на Паскале должны быть помещены описания текстов подпрограмм, в которых реализуется логика (алгоритм) каждого метода. При этом в заголовке каждого метода перед его именем обязательно помещается имя класса.

Пример 1.

Проиллюстрируем сказанное на примере создания иерархии классов, представителями которых являются объекты, отображаемые (видимые) на экране монитора компьютера.

Вначале изобразим иерархию классов:



Кратко охарактеризуем их.

Класс TPoint.

Экземплярами-объектами данного класса являются точки на поверхности экрана, которые могут быть видимыми. Полный перечень свойств объектов этого класса:

- декартовы координаты (*x,y*) - характеризуют местоположение на плоскости;
- цвет, в который окрашена точка;
- признак видимости: точка может быть видима на экране и невидима (чтобы точка была невидима, ее достаточно окрасить в цвет фона).

Объявление класса имеет следующий вид:

```
type
   TPoint = object
       x, y : integer; {координаты объекта}
       color : byte; {цвет объекта}
       visible : boolean; (признак видимости)
       Constructor Init(ix,iy:integer; {кординаты}
                        icolor: byte); {цвет}
       function GetX:integer; {сообщить x-координату}
       function GetY:integer; {сообщить y-координату}
       procedure Show; virtual; {показать на экране}
       procedure Hide; virtual; {сделать невидимым}
       function IsVisile:boolean; {сообщить,
                                 видим объект или нет}
       procedure MoveTo(xnew,ynew:integer);{переместить
                           в позицию с новыми координатами}
       procedure Drag(Step:integer);   {буксировать объект}
       Destructor Done; virtual;    {уничтожить объект}
   end;
```

Замечание. Все дочерние классы (классы-потомки) для класса TPoint объединяет то, что каждый объект из этих классов располагается на плоскости и, следовательно, ассоциируется с некоторой базовой для него позицией («точкой привязки»). И если для класса TPoint точка (x,y) полностью определяет весь объект, то для других дочерних классов эта точка – одна из точек, относительно которой располагается ее образ (фигура).

Так, для объекта «окружность» точкой привязки может выступить центр окружности, а для прямоугольника одна из его вершин или координаты центра.

Ниже приведено объявление методов класса TPoint. Обратите внимание, что перед именем каждого метода записано имя класса, отделенное от него точкой. Такая запись называется квалифицирующей записью. Так, запись TPoint.Show означает, что речь идет о методе Show объекта класса TPoint.

```
{  Описание методов класса TPoimt:  }
  Constructor TPoint.Init(ix,iy:integer; icolor:byte);
  begin
     x:=ix;  {присвоить значение полю x создаваемого объекта}
     y:=iy;  {присвоить значение полю y создаваемого объекта  }
     color:=icolor;  {присвоить значение цвета}
     visible:=false;  {принять, что в момент создания  объект невидим }
  end;

  function TPoint.GetX:integer;
  begin
     GetX:=x;  {вернуть значение x-поля}
  end;

  function TPoint.GetY:integer;
  begin
     GetY:=y;  {вернуть значение y-поля}
  end;

  procedure TPoint.Show;
  begin
     PutPixel(x,y,color);  {окрасить точку в цвет color}
     visible:=true;
  end;

  procedure TPoint.Hide;
  var
     TempColor:byte;  {переменная для временного сохранения
                                        цвета объекта-точки                }
  begin
     TempColor:=color;
     color:=GetBkColor;{установить для точки цвет фона}
     Show;              {нарисовать объект новым цветом – цветом фона}
     visible:=false;
     color := TempColor;  {восстановить исходный цвет точки}
  end;

  function TPoint.IsVisible:boolean;
  begin
     IsVisible:=visible;  {вернуть значение поля  visible}
  end;

  procedure TPoint.MoveTo(xnew,ynew:integer);{}
  var
     Flag:boolean;  {видима ли точка перед перемещением}
  begin
     Flag:=IsVisible;
     if Flag then
        Hide;    {спрятать, если точка была видима}
     x:=xnew;
     y:=ynew;
     if Flag then
        Show;    {показать вновь после перемещения}
  end;
```

```
procedure TPoint.Drag(Step:integer); {буксировать объект}
var
   Ch:char;
begin
   repeat
      Ch := ReadKey;
      if Ord(Ch)=0 then begin
         Ch := ReadKey;
         Case Ord(Ch) of
           72: MoveTo(x, y-Step); {Up}
           75: MoveTo(x-Step, y); {Left}
           77: MoveTo(x+Step, y); {Right}
           80: MoveTo(x, y+Step); {Down}
         end; { case}
      end;
   until Ord(Ch)=13; {прекратить буксировку и завершить
                      процедуру при нажатии клавиши Enter}
end;

destructor TPoint.Done;
begin
   Hide;   {убрать с поверхности экрана}
end;
```

Приведем некоторые комментарии к тексту на Паскале.

1. Создание объекта-экземпляра производится конструктором («методом») с именем *Init*. В Турбо-Паскале все конструкторы должны иметь имя *Init.*
2. Уничтожение объекта производится деструктором («методом») *Done*. Этот метод нужен для объектов, которые создаются как динамические (Этот вопрос будет рассмотрен позже). В рассматриваемом примере для данного класса в теле процедуры *Done* помещен только вызов метода *Hide*, убирающий объект с экрана. Однако этим действия по уничтожению объекта не ограничивается. Производится освобождение памяти, выделяемой менеджером памяти для объекта. Эти действия выполняются автоматически.
3. Назначение атрибута *virtual* в объявлении ряда методов класса будет обсуждаться позже.
4. Метод *Drag* обеспечивает буксирование объекта-точки под воздействием клавиш навигации в любом из четырех направлений. При этом однократное нажатие на клавишу обеспечивает перемещение объекта в соответствующем направлении на *Step* пикселов. Буксирование прекращается при нажатии клавиши *Enter*.

Теперь рассмотрим, как можно использовать объекты объявленного выше класса.

15.3. Статические и динамические объекты

Чтобы создать объект, надо выполнить его конструктор.

Статический объект создается в статической области памяти программы. На этапе компиляции для него выделяется память, а при выполнении конструктора в эту память записываются значения полей (свойств объекта) и адреса точек входа в методы.

Для того, чтобы создать объект в динамической области памяти используют процедуру New. При этом возможны два способа создания объекта:

- Первый заключается в выполнении двух действий: вначале получить необходимую память для объекта, используя процедуру *New*, а затем выполнить конструктор *Init* (этот процесс изображен ниже для объекта PL1^).
- Второй способ основывается на модифицированной процедуре *New*, которая имеет два параметра: первый – имя переменной – указателя на объект, а вторая – вызов конструктора.

Так, запись

```
New(PL2, Init(130,251,Red));
```

имеет следующий смысл: выделить для объекта память в динамической области, вернув указатель на нее в переменную PL2, и выполнить конструктор для этого объекта, установив его свойства следующими: координаты (x=130, y=251) и красный цвет.

Вместе с модифицированной процедурой *New* для обеспечения работы с объектами можно использовать и модифицированную процедуру освобождения памяти из динамической области *Dispose*, в которой также имеется два параметра. Покажем это на примере уничтожения объекта PL2:

```
Dispose(PL2, Done);
```

Эта запись означает следующее: уничтожить объект, на который указывает переменная PL2, и затем освободить занимаемую им память.

Иллюстрация сказанного приведена ниже.

```
type
     Ppoint=^TPoint;  {тип – указатель на объект}
var
     L1: TPoint;     {имена статических объектов}
     PL1, PL2 : PPoint; {имена динамических объектов,
                         точнее – имена указателей на объекты,
                         которые будут созданы в динамической
                         области программы}
begin
   L1.Init(10,20,Blue);   {создать статический объект «точка»}

   PL1:=New(PPoint);       {запросить память под объект}
   PL1^.Init(10,20,Blue);{и создать его}

   New(PL2, Init(130,251,Red)); {второй вариант создания
       динамического объекта, за счет модифицированной процедуры New}
. . .
   L1.Show;    {Показать на экране объект}
   PL2^.Show;       {Показать на экране объект}
   PL2^.MoveTo(45,30);  {Переместить объект на новое место}
. . .
   Dispose(PL2, Done);  {Уничтожить объект и освободить память}
. . .
```

Теперь введем два новых класса видимых объектов – ”Окружности” и ”Прямоугольники”, которые объявим потомками класса ”Точки”.

Оба класса являются потомками класса TPoint и в силу этого наследуют все методы родительского класса. По этой причине в объявлении классов требуется всего два метода – конструктор и Show. Наличие конструктора обязательно для любого класса объектов, если дочерний класс имеет новые свойства (поля). А метод Show необходим в связи с тем, что объекты каждого класса ”выглядят” иначе, чем другие.

Приведем их объявления:

```
Type
   PCircle = ^TCircle;
   Tcircle = object (TPoint) {Класс типа Окружность}
       Radius : integer;      { радиус }
       Constructor Init(ix,iy:integer; {кординаты}
                        iRadius:integer; {}
                        icolor: byte); {цвет}
       procedure Show;virtual;
   end;

   PRect = ^TRect;
   TRect = object(TPoint) { Класс типа Прямоугольник}
       Width,Height : integer; {}
       Constructor Init(ix,iy:integer; {кординаты}
                        iWidth,IHeight:integer; {}
                         icolor: byte); {цвет}
       procedure Show;virtual;
   end;
```

Как видно из объявления иерархии классов, в каждом классе объявлен метод с одним и тем же именем Show, но реализующий неодинаковые алгоритмы. Это и есть пример полиморфизма имени Show.

А теперь можно вернуться к объяснению назначения атрибута virtual. При рассмотрении класса TPoint мы описали логику метода Hide следующим образом: установить цвет объекта равным цвету фона и выполнить его (объекта) метод Show. В этом описании никак не детализировался алгоритм последнего метода: Мы использовали то обстоятельство, что объект сам себя умеет нарисовать. Но такое же утверждение справедливо и для объектов вновь объявленных классов – они тоже умеют себя нарисовать, и следовательно, алгоритм унаследованного ими метода Hide полностью подходит и для них (потомков).

Единственная проблема, которая может оказаться существенной – в методе Hide нет указания того, какому классу принадлежит вызываемый в нем метод Show. Так вот, если в объявлении класса TPoint убрать атрибут virtual, то при выполнении метода Hide будет вызываться метод Show класса TPoint, а при его наличии будет осуществляться поиск метода Show в том классе, которому принадлежит объект. Именно по этой причине в объявлении новых классов даже не упоминается виртуальный метод Hide, хотя его логика соответствует объектам именно этих классов.

Ниже приведено объявление еще одного класса – строки текста. Этот класс построен как потомок класса TRect. Выбор такого наследования связан с тем, что строка текста занимает прямоугольник. Как расположена точка «привязки» этого прямоугольника относительно текста,

можно увидеть из конструктора класса (TText.Init). А его метод Show показывает, как будет выглядеть объект.

```
type
     PText = ^TText;
     TText = object(TRect) {потомок класса Прямоугольник}
         Txt:string; {текст, отображаемый на экране}
         Constructor Init(ix,iy:integer; {кординаты}
                          itext:string;  {текст}
                          icolor: byte); {цвет}
         procedure Show;virtual;
     end;

Constructor TText.Init(ix,iy:integer;    {кординаты}
                          itext:string;  {текст}
                          icolor: byte); {цвет}
     var
        w,h:integer;
     begin
        w:=(Length(itext)+2)*TextWidth('W');
        h:=2*TextHeight('W');
        Inherited Init(ix,iy,w,h,icolor);
        txt:=itext;
     end;

     procedure TText.Show;
     var
       TempColor:byte;
     begin
        TempColor:=GetColor;
        SetTextJustify(CenterText,CenterText);
        SetColor(color);
        OutTextXY(x+width div 2,y+height div 2,txt);
        SetColor(TempColor);
        Visible:=true;
     end;
```

Ниже приведен фрагмент программы, иллюстрирующий работу с объектами описанных классов.

```
. . . .
var
   c1:PCircle;
   r1:PRect;
   t1:PText;
. . . .
begin
. . . .
   Инициализация графического режима (Здесь не показана)
. . . .
   New(c1, Init(120,120,20,Yellow));{создание
                                  объекта «окружность»}
   c1^.Show;
   c1^.Show;
   c1^.Drag(2);
   Dispose(c1, Done); {уничтожение объекта «окружность»}


   New(r1, Init(10,10,45,20,red)); {создание объекта
                                         «прямоугольник»}
   r1^.Show;
   r1^.MoveTo(50,24);

   New(t1, Init(300,300,'Example',Magenta)); {создание объекта
                                              «строка текста»}
   t1^.Show;
   t1^.Drag(5);
   repeat until KeyPressed;
   CloseGraph;
end.
```

15.4. Правила построения и использования объектов

а. Правила наследования

1. Информационные поля и методы родительского класса наследуются всеми дочерними классами независимо от числа промежуточных уровней иерархии.
2. Доступ к полям и методам родительских классов для дочерних выполняется так, как будто бы они описаны в самом дочернем классе.
3. Ни в одном из дочерних классов не могут использоваться идентификаторы полей, совпадающие с идентификаторами полей каких-либо родительских классов. Это же относится и к идентификаторам формальных параметров, указанным в заголовках методов.
4. Дочерний класс может доопределить произвольное число собственных методов и информационных полей.

5. Любое изменение текста в родительском методе автоматически оказывает влияние на все методы порожденных дочерних классов, которые его вызывают.
6. В противоположность информационным полям (правило 3) идентификаторы методов в дочерних классах могут совпадать с именами методов в родительских классах. В этом случае дочерний метод подавляет одноименный ему родительский, вследствие чего для объекта дочернего класса будет вызываться метод именно дочернего, а не родительского класса. В то же время остается и возможность вызова родительского класса. Для этого необходимо использовать квалифицированный (уточненный) идентификатор, в котором перед именем метода помещается имя требуемого родительского класса. Пример записи квалифицируемого идентификатора – TPoint.Show.

б. Виртуальные методы

1. Виртуальными следует объявлять те методы, которые изменяются в классе-потомке.
2. Если метод одного из классов в иерархии объявлен как виртуальный, то все классы иерархии должны также объявить его виртуальным.
3. Список формальных параметров любого виртуального метода во всей иерархии должен совпадать и по типам и по именам параметров.

в. Ранее и позднее связывание

Ранее связывание – процесс, в результате которого вызовы статических правил однозначно разрешаются компилятором во время трансляции метода.

Позднее связывание: вызывающий и вызываемый методы не могут быть связаны во время компиляции. Их связывание происходит в момент вызова. Это обеспечивается за счет создания (автоматического) так называемой таблицы виртуальных методов.

г. Совместимость классов объектов

Совместимость классов относительно операции присваивания простирается *только от потомков к прародителям*, но не наоборот:

```
var
   point : PPoint;
   circle : PCircle;
   rect : PRect;
   txt : PText;
begin
      допустимо                     недопустимо
   point := circle;            circle := point;
   rect := txt;                txt := rect;
   point := txt;
```

Следствие этого: имеется возможность включить объекты разных классов одной и той же иерархии в один список, при условии, что в ка-

честве указателя на объект элемента списка указан тип *указателя на прародителя* всей иерархии классов. Пример использования этой возможности для объекта класса TText приведен на следующей странице. В нем объект данного класса включается в список, элементами которого могут быть объекты из классов, принадлежащих всей иерархии, начиная от объектов класса TPoint.

```
type
     Pitem = ^Titem;
     TItem=record
        Info : PPoint;   {указатель на объект класса TPoint}
        pred, next :Pitem;
     end;

  end;
  procedure Add(var Head:Pitem; p:Pitem);
  begin
     . . . .
  end;
var
     Head:Pitem;
     newItem:Pitem;
     tt:Ptext; { указатель на объект TText;              }
               { TPoint - для него прародительский класс }
begin
. . . .
     Head:=nil;
     New(tt, Init(100,100,'Слово',blue));
     New(newItem);
     NewItem^.Info := tt;
     Add(Head, newItem);
 . . .
```

Вопросы для самоконтроля

1. Что такое объектно-ориентированное программирование?
2. Что такое инкапсуляция?
3. Что такое наследование свойств?
4. Что такое полиморфизм?
5. В чем отличия типа «объект» от типа «запись»?
6. Что называется методом?
7. В чем отличия описания методов и подпрограмм?
8. Каковы правила наследования в Турбо-Паскале?
9. Для чего предназначено ключевое слово Inherited?
10. Какие существуют правила вызова наследуемых методов?
11. Что такое виртуальный метод?
12. Что такое конструктор и деструктор для чего они нужны?
13. Каковы правила совместимости объектных классов?

ЗАКЛЮЧЕНИЕ

При разработке прикладной программы программист решает две задачи: реализации алгоритма обработки, соответствующего постановке задачи, и обеспечения удобного интерфейса пользователя с программой. Интерфейс пользователя реализуется, как правило, с использованием разнообразных видимых активных элементов, таких как окна, кнопки, строки редактирования, средства прокрутки списков, выпадающие и всплывающие меню и других.

При использовании традиционных языков программирования реализация необходимых объектов интерфейса требует значительных затрат времени программиста, особенно в среде с графическим интерфейсом, например, в среде операционной системы Windows. В языки программирования нового поколения, получившие название языков визуального программирования (иногда их называют языками визуального проектирования программ), в ответ на эти потребности встроены средства обеспечения интерфейса, за счет чего организация интерфейса значительно упрощена.

Элементы интерфейса реализованы в этих языках в виде объектов (в терминологии, принятой в этих языках, их называют компонентами). Для любого компонента определен набор полей (иначе – свойств), набор функций (методов) и набор событий, ассоциированных с ним. Наиболее известные языки визуального программирования - Visual Basic (VB), Delphi , Visual C и Borland C++. Эти языки – объектно-ориентированные потомки широко известных языков Бейсик, Паскаль и Си.

Основу интерфейса любой программы в языке визуального программирования представляет так называемая **форма** – прямоугольная область на экране, которая может быть использована программистом для отображения на ней той информации и тех элементов интерфейса, которые обеспечивают взаимодействие пользователя с программой.

При использовании визуальных компонентов реализация интерфейса программы сводится к выбору необходимых элементов интерфейса и размещению их в выходной форме. Программист при этом может установить желаемые значения свойств (размеры, местоположение, цвет и другие), указать, какие методы необходимо использовать и на какие события, связанные с конкретным элементом интерфейса, должна быть обеспечена реакция в программе. В ответ на это система программирования автоматически подготовит значительную часть кодов на языке, а программист должен разработать только коды, которые реализуют алгоритмы обработки, специфические для конкретной задачи.

В состав компонентов языка включено также значительное количество и невизуальных объектов, которые также часто применяются в про-

граммах. В качестве примера можно привести объекты «Список» и «Таймер».

Для большинства прикладных программ интерфейс пользователя, как правило, реализуется не одной, а несколькими формами. Форма также является компонентом, и ей также сопоставляется набор свойств, методов и событий. А при ее выборе автоматически формируется отдельный файл стандартной структуры, который содержит сведения о форме и включенных в нее элементах интерфейса, а также описание функций, ассоциированных с ними.

В системе программирования Delphi файлы, ассоциированные с каждой формой, оформляются как модули (Unit). В процессе разработки конкретной формы программист размещает на ней необходимые элементы интерфейса. При этом для каждого метода этого элемента и для каждого выбранного программистом события в модуль подставляется шаблон подпрограммы (функции). Программисту остается только разработать алгоритм и поместить его в тело подпрограммы.

Поскольку при такой организации программы она создается как некоторый набор модулей, система программирования берет на себя функции автоматического включения сведений о них в специальный файл проекта.

Разумеется, в соответствии с особенностями решаемой задачи программист может разрабатывать самостоятельно и другие модули, которые не сопоставляются ни с какой формой, но должны содержать требуемые алгоритмы. Как правило, эта возможность используется для тех алгоритмов, которые не связаны с визуальными компонентами. Отметим, что структура и правила использования модулей Unit в Delphi полностью совпадают с таковыми в Турбо-Паскале.

Основным следствием сказанного является то, что при использовании языка визуального программирования несложно организовать качественный интерфейс, и программист может сосредоточиться на реализации алгоритма. За счет этого не только сокращается время на разработку программы, но и уменьшается количество ошибок в ней при разработке.

В заключение приведем характерные особенности языка визуального программирования:

- он содержит широкий набор компонентов (объектов), значительно упрощающих процесс реализации качественного интерфейса программы;
- программа в нем представляется как событийно-ориентированная, т.е. представляющая собой набор подпрограмм (функций и/или процедур), которые активизируются при наступлении соответствующих событий. Для сравнения отметим, что на языках-предках: Паскаль,

Си, Бейсик, прикладная программа управлялась не событиями, а внутренней логикой. Читатель, наверняка, встречался с примерами событийно-ориентированных программ. В качестве конкретного примера можно указать интегрированную среду Турбо-Паскаль;

- в языке имеются средства поддержки модульного принципа организации программы, в силу чего он более пригоден для разработки сложных программ.

В заключение отметим, что хотя языки визуального программирования существенно упрощают процесс разработки программ, их освоение целесообразно начинать после приобретения навыков разработки алгоритмов с использованием подпрограмм и концепций объектно-ориентированного программирования.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

Литература по программированию на языке Паскаль

1. Иванова Г.С. Основы программирования. , 2-е изд., перераб. и доп.: Учеб. пособие для вузов / Г. С. Иванова. — М.: Изд-во МГТУ им. Н. Э. Баумана, 2002. — 416 с.: ил.
2. Марченко А.И., Марченко Л.М. Программирование в среде Turbo Pascal 7.0 – К.:ВЕК+, М.: Бином Универсал, 1988. – 406 с.
3. Фаронов В.В. Турбо Pascal 7.0. Начальный курс - М.: Нолидж, 1998 – 620с.
4. Фаронов В.В. Турбо Pascal 7.0. Практика программирования. Учебное пособие – М.: Нолидж, 1998 –432 с.
5. Питеркин В.М. Основы программирования на языке высокого уровня. Учебное пособие. – Московский государственный институт электроники и математики, М., 2002.

Литература по технологии разработки программ

1. Вирт Н. Алгоритмы + структуры данных = программы - М.: Мир, 1985.
2. Ахо А., Хопкрорт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов - М.: Мир, 1979.
3. Майерс Г. Надежность программного обеспечения. – М.: Мир. - 1980.
4. Майерс Г. Искусство тестирования программ – М.: Мир, 1981.
5. Сэм Канер. Тестирование программного обеспечения – М.: ДиаСофт, 2000.
6. Каймин В.А., Питеркин В.М. Основы информатики и вычислительной техники. Учебное пособие. – М.: Изд. МИЭМ, 1985.
7. Арменский Е.В., Каймин В.А., Питеркин В.М. Методы составления и проверки правильности программ. В кн.: Проблемы компьютерного обучения: Сб. статей. – М.: Знание, 1986.
8. Каймин В.А., Жданов В.С, Питеркин В.М., Уртминцев А.Г. Информатика: Учебное пособие для учащихся старших классов общеобразовательных учреждений и абитуриентов. – М.: М.:АСТ., 1996.

ПИТЕРКИН ВЯЧЕСЛАВ МИХАЙЛОВИЧ

Информатика и программирование

часть 2

учебное пособие

Гарнитура Таймс. Печать - ризография.
Формат 60х90 1/16. Усл. печ. л. 12,0
Тираж 60 экз. Заказ № 06/11
Отпечатано в РИО РГУИТП.

Российский государственный университет
инновационных технологий и предпринимательства

РИО РГУИТП
107078, г. Москва, ул. Новая Басманная, д. 9.разделр