Министерство образования и науки Украины
Севастопольский национальный технический университет

Методические указания

по дисциплине

“Операционные системы ПК”

для студентов дневной и заочной формы обучения
специальностей 7.090801 и 7.090804

Севастополь

2005

УДК681.3
Операционные системы ПК
/ Д.Г. Мурзин Учебное пособие. – Севастополь: СевНТУ, 2005. – 20с.

Целью учебного пособия является оказание помощи студентам в подготовке к выполнению курсового проекта по дисциплине «Операционные системы ПК».

Учебное пособие предназначено для студентов, обучающихся по направлению «Электроника».

Учебное пособие рассмотрено и утверждено на заседании кафедры электронной техники (протокол № 1 от 25 августа 2005 г.)

Рецензент Астраханцев А.В. канд. тех. наук, доцент кафедры ЭЛТ.

СОДЕРЖАНИЕ

Предисловие

Эта книга написана на основе курса лекций, читавшегося автором студентам 1-2 курсов математического факультета. Она не рассчитана на использование в качестве справочника по языку Паскаль, но может быть полезна начинающим программистам, желающим самостоятельно изучить язык или слушающим соответствующий лекционный курс. В книге не приведена информация о среде программирования Turbo Pascal, за исключением некоторых важных моментов, поскольку эту программу лучше изучать непосредственно за компьютером. Кроме того, предполагается, что читатель хотя бы в минимальной степени знаком с операционной системой DOS и понимает, например, что такое имя файла, каталог, диск, путь и т.п. Книга содержит некоторое количество примеров, записанных как фрагмент программы или законченная программа. Все примеры оттестированы и не содержат ошибок. Однако автор хотел бы предостеречь от некритического использования этих программ читателем - большинство из них лишь иллюстрирует возможности языка и не претендует на оптимальность. Синтаксис языка соответствует среде программирования Borland Pascal Version 7.0.

Автор заранее благодарен тем читателям, которые сообщат ему свои замечания по содержанию этой книжки.

1. Общая схема решения задачи на персональном компьютере

В общем виде процесс решения любой программистской задачи на ПК можно представить в виде последовательности следующих действий:

1) разработка алгоритма решения задачи;
2) создание текста программы;
3) отладка программы;
4) тестирование программы.

Все эти этапы (иногда в неявной форме) обязательно выполняются любым программистом при решении любой задачи. Рассмотрим их подробнее.

Алгоритмом называется некоторая заранее определенная последовательность действий, позволяющая на основании имеющейся информации получить искомый результат. Обязательной составной частью алгоритма является определение состава и формы представления входной и выходной информации. При решении некоторых наиболее тривиальных задач может показаться, что этап разработки алгоритма отсутствует, однако это означает лишь, что вы пользуетесь уже известным вам алгоритмом.

На этапе создания *текста программы* вы записываете алгоритм на языке программирования. Один и тот же алгоритм можно запрограммировать множеством различных способов, но вы должны стремиться написать оптимальную программу. Хорошо написанная программа, как правило, содержит меньше ошибок и гораздо быстрее отлаживается.

Этап *отладки* включает в себя компиляцию программы и проверку ее на простейших тестах. Компиляция (или трансляция) программы - это процесс перевода ее с языка программирования на машинный язык, его осуществляет специальная программа - компилятор (транслятор). При этом вы постепенно исправляете допущенные при написании программы *синтаксические* ошибки, следите за сообщениями компилятора - он указывает, какая обнаружена ошибка и где именно. После того, как вы исправите все синтаксические ошибки и компилятор сообщит об успешном завершении компиляции, будет создан файл с именем, таким же, как у вашего исходного файла, и с расширением **exe** (от EXEcutive - выполняемый); этот файл содержит *программу* (в отличие от исходного файла, содержащего лишь *текст* программы), которая может быть *выполнена*. Необходимо отчетливо понимать, что задачей компилятора ни в коем случае не является поиск ошибок в ваших программах, он сообщает о них лишь в том случае, когда не может правильно интерпретировать ваш текст. Успешно осуществив компиляцию, запустите свою программу. Не следует думать, что эта программа не содержит ошибок! Все логические ошибки, допущенные вами, остались в программе, и на этапе отладки вы должны найти их и исправить. (Не верьте тем программистам, которые утверждают, что они могут сразу написать правильную программу, - таких людей не бывает). Не существует никаких общих рецептов для отладки - класс программиста, главным образом, как раз и проявляется в том, как он отлаживает программы. Но один полезный совет можно дать: аккуратно и подробно выводите при отладке все вычисляемые вашей программой величины.

После того, как вы решите, что ваша программа работает правильно (обычно это не соответствует действительности), начинайте тестирование - выполняйте программу с различными наборами входных данных, причем они обязательно должны содержать *все особые случаи*. Когда вы убедитесь, что ваша программа иногда работает правильно, а иногда - нет, возвращайтесь к алгоритму, пересматривайте его и заново повторяйте все этапы. Успешно завершив тестирование, вы можете надеяться, что ваша программа верна.

Следует четко разграничивать два понятия - верная программа и хорошая программа. Всякая хорошая программа верна, но далеко не всякая верная программа хороша - она может использовать неэффективный (или неэффективно запрограммированный) алгоритм, занимать много лишней памяти, быть неряшливо оформленной и т.д. Старайтесь писать не только верные, но и *хорошие* программы!

2. Введение в язык Паскаль. Общая структура программы. Идентификаторы, комментарии, пробелы. Раздел описаний и раздел операторов

Запишем для начала программу на языке Паскаль :

```
{ эта программа просто выводит сообщение }
  BEGIN WRITELN('Привет !!!'); END.
```

Это правильная программа, и если вам удастся ее откомпилировать и запустить, она выведет на экран сообщение: "Привет !!!". Эту программу мы могли бы записать и так:

```
CONST Message='Привет !!!';
    BEGIN
      WRITELN(Message);
          END.
```

и так :

```
VAR Message:STRING[10];
BEGIN Message:='Привет !!!'; WRITELN(Message); END.
```

и еще множеством различных способов, но в каждой программе обязательно будет слово **BEGIN,** и в конце программы всегда будет стоять **END.** - признак конца программы. Перед **BEGIN** может что-то быть (как правило, это так), или может не быть ничего. То, что находится перед **BEGIN,** называется *разделом описаний*, то, что находится между **BEGIN** и **END.** , называется *разделом операторов*. Слова **BEGIN**, **END**, а также **CONST**, **VAR**, **STRING**, **WRITELN** являются *ключевыми словами* языка Паскаль, а слово **Message** - это идентификатор пользователя, т.е. *имя*, данное нами некоторому объекту - константе, переменной, или чему-то еще. Все ключевые слова и идентификаторы пользователя есть последовательности букв и цифр, начинающиеся с буквы. Буквами языка являются все латинские буквы и символ подчеркивания. Компилятор не различает большие и малые латинские буквы, поэтому вы можете записывать идентификаторы как захотите: **Begin**, **BEGIN**, **begin** и т.д. Вы можете выбирать любые идентификаторы пользователя, лишь бы они не совпадали с ключевыми словами; так, в нашем примере вместо **Message** вы можете написать **Q** или **__t123**, или **Y56_ert** и т.д. Однако все эти идентификаторы не несут в себе никакого смысла, затрудняют чтение и отладку программы и делают ее неряшливой; идентификатор **Message** имеет то достоинство, что из него уже ясно его назначение - содержать некоторое сообщение. Старайтесь всегда использовать в программе осмысленные идентификаторы! Язык Паскаль допускает идентификаторы длиной до 63 символов (точнее, компилятор различает первые 63 символа имени), поэтому не экономьте на именах переменных и функций, пусть лучше имена будут длинными, но понятными. Кроме ключевых слов и идентификаторов всякая программа содержит также пробелы и (в идеале) комментарии. Комментарии записываются в фигурных скобках и могут стоять в любом месте программы, пробелы являются разделителями, там, где допустим один пробел, можно поставить любое количество пробелов. Комментарии и пробелы следует использовать для аккуратного оформления текста программы. Хорошая программа обязательно должна быть документирована, т.е. содержать комментарии, поясняющие, как она работает.

3. Арифметические типы данных. Числовые константы и переменные. Оператор присваивания. Выражения

В языке Паскаль определены следующие арифметические типы данных: целочисленные типы - **Byte**, **ShortInt**, **Word**, **Integer** и **LongInt**; вещественные типы - **Single**, **Real**, **Double** и **Extended**; и не совсем вещественный тип **Comp**. Характеристики этих типов приведены в таблице 1 (запись **1.5e-45** означает 1.5, умноженное на 10 в степени -45, это общепринятое в языках программирования обозначение для вещественных чисел - *константа с плавающей точкой*).

Таблица 1

Арифметические типы данных

Название типа	Диапазон допустимых значений	Количество верных цифр	Размер в байтах
Byte	0...255	-	1
ShortInt	-128...127	-	1
Word	0..65535	-	2
Integer	-32768...32767	-	2
LongInt	-2147483648...2147483647	-	4
Single	±1.5e-45... ±3.4e+38	7-8	4
Real	±2.9e-39... ±1.7e+38	11-12	6
Double	±5.0e-324... ±1.7e+308	15-16	8
Extended	±3.4e-4932... ±1.1e+4932	19-20	10
Comp	-9.2e18...9.2e18	8	8

Типы **Byte** и **Word** используются для целых величин без знака, типы **ShortInt**, **Integer** и **LongInt** - для целых со знаком, типы **Single**, **Real**, **Double** и **Extended** - для вещественных величин. Тип **Comp** может содержать только *целые* числа от $-2^{63}+1$ до $+2^{63}-1$, но эти числа хранятся в *вещественном* формате, поэтому тип **Comp** считается вещественным. С данными типа Comp можно обращаться так же, как с данными других вещественных типов, но дробная часть числа при этом автоматически отбрасывается.

Целые числовые константы записываются в языке Паскаль в десятичном виде или в 16-ричном виде, 16-ричная константа начинается с символа **$** и содержит 16-ричные цифры : **0-9**,**A-F**. Например, число 255 можно записать как **$FF**. Числовые константы по умолчанию имеют тип **Integer** или **LongInt**. Вещественные константы записываются либо с фиксированной точкой, например, **-1.234**, либо с плавающей точкой, например, **-1.234E-5** или **555e12**.

В программе, как правило, приходится использовать переменные арифметических типов. Каждая такая переменная (и переменная любого другого типа) в языке Паскаль должна быть обязательно описана, т.е. должен быть явно указан ее тип. Описание переменных в общем случае имеет вид:

VAR *имя* **,** ... **,** *имя* **:** *тип* **;** *имя* **,** ... **,** *имя* **:** *тип* **;** ...

Здесь *имя* - имена переменных (идентификаторы), *тип* - типы переменных, **VAR** - ключевое слово, означающее, что после него следуют описания переменных. Переменные одного типа можно описать совместно, разделив их имена запятыми, а можно описывать и каждую переменную отдельно. Точка с запятой означает окончание описания переменных данного типа. Слово **VAR** может повторяться в программе сколько угодно раз. Выбор типа для той или иной переменной определяется назначением этой переменной. Пусть, например, переменная **i** служит счетчиком (индексом) элементов некоторой последовательности, причем известно, что количество элементов не может превосходить **100**. Мы можем описать переменную **i** любым целочисленным типом, но правильный выбор - **Byte** или **ShortInt**, любой другой тип будет избыточным. Всегда следует выбирать типы переменных осознанно; если вы не понимаете, какой тип должна иметь ваша переменная, вероятнее всего, эта переменная в программе не нужна. Для вещественных переменных чаще всего используется тип **Real**, являющийся основным вещественным типом в Паскале, поэтому мы везде будем писать **Real** для вещественных переменных, хотя это может быть и другой вещественный тип.

Пусть в программе нам необходимы переменные **b1,b2,b3,b4** типа **Byte**, переменные **i,j,k** типа **Integer** и переменные **r1,r2** типа **Real**. Их можно описать, например, так:

```
VAR b1,b2,b3,b4  : Byte;
      i,j,k      : Integer;
      r1,r2      : Real;
```

или так :

```
VAR b1          : Byte;
      i,j,k     : Integer;
VAR r1          : Real;
VAR b2,b3,b4 : Byte;
      r2        : Real;
```

Эти описания эквивалентны.

Всякая переменная обладает четырьмя атрибутами: *именем*, *типом*, *адресом* и *значением*. Имя переменной есть идентификатор, т.е. последовательность символов; тип переменной определяет ее свойства, диапазон допустимых значений и размер памяти, необходимый для размещения этой переменной; адрес

переменной указывает на место в памяти, где размещается ее значение; переменная всегда имеет некоторое значение, даже если вы ничего не сделали, чтобы определить это значение. В последнем случае говорят, что переменная не определена; это значит, что ее значение не известно нам заранее (ни в коем случае не следует думать, что неопределенные переменные имеют нулевые значения - это не так).

Каким же образом определить значение переменной ? Для этого используется *оператор присваивания*:

*имя***:=** *выражение***;**

Здесь мы встречаемся с двумя новыми понятиями - оператор и выражение. *Оператор* - это минимальная осмысленная конструкция в языке Паскаль, вся программа - это последовательность операторов. Оператор всегда заканчивается символом "**;**", кроме одного единственного оператора **END.** Допускаются пустые операторы "**;**" , не выполняющие никаких действий. *Выражение* - это конструкция, состоящая из одного или нескольких *операндов* и, возможно, знаков операций, и имеющая некоторое *значение*. Операндами могут быть константы, переменные и другие выражения, т.е. вы можете строить сколь угодно сложные выражения. Мы не знаем пока никаких знаков операций, но предположим, что знак + означает операцию сложения (это так и есть). Запишем несколько выражений:

1 (константа есть частный случай выражения);
b1 (переменная есть частный случай выражения);
25+1E3
b1+4.25+r2

Теперь мы можем присвоить переменной ее значение:

i:=-11; j:=22+i; k:=i+j+177;

Наряду с переменными в Паскале есть и другие именованные объекты - это *константы* (отличайте их от числовых констант, которые не имеют имени, а лишь значение). Константы бывают двух видов - нетипизированные и типизированные. Нетипизированные константы описываются, так же, как и переменные в разделе описаний, в виде :

CONST *имя=значение***;** *имя=значение***;** ...

Здесь *имя* - идентификатор, *значение* - вообще говоря, некоторое выражение, которое может включать и именованные константы, описанные выше, но только не переменные. Запишем несколько примеров:

```
CONST C=-155;
        D=C+100;
         E=1E2+C+D;
CONST F=D+1;
CONST G=C+F;
```

Нетипизированные константы, описанные в разделе описаний, вы можете затем использовать в разделе операторов в выражениях, но изменить их значения невозможно. Не совсем удачное название "нетипизированные" означает не отсутствие у констант типа - любая константа имеет совершенно определенный тип, который определяется ее значением, - а лишь то обстоятельство, что при описании таких констант тип не указывается *явно*. В нашем примере константы **C**,**D**,**F** и **G** имеют тип **Integer**, а константа **E** - тип **Real**. Второй класс именованных констант - *типизированные* константы, описание которых имеет вид:

CONST *имя***:***тип=значение***;** *имя***:***тип=значение***;** ...

Эти константы вы можете использовать так же, как и нетипизированные, но можете и изменять их значения (например, с помощью оператора присваивания) подобно переменным. Типизированные константы можно, с небольшими оговорками, рассматривать как переменные, которым присвоено начальное значение. Приведем пример :

```
CONST t:Word   = $FFFF;    b:Byte    = 11;    r:Real    = 1.23E-16;    z:Integer = 0;
BEGIN  t:=t-1; END.
```

4. Операторы ввода-вывода

Простейший оператор ввода в Паскале - оператор **READ**, он записывается в виде:

READ(*имя***,***имя***,...);**

где *имя* - имена переменных или типизированных констант. Вводимые значения задаются в виде допустимых в Паскале констант и разделяются любым количеством пробелов. Для окончания ввода следует нажать клавишу **Enter**. Оператор ввода можно записать и как **READLN**, при вводе числовых переменных они эквивалентны. Кроме того, оператор **READLN** без списка в скобках можно использовать для организации задержки в работе программы - он будет ожидать нажатия клавиши **Enter**.

Простейший оператор вывода записывается в виде:

WRITE(*выражение***,***выражение***,...);**

или

WRITELN(*выражение***,***выражение***,...);**

Вывести можно любое выражение, если необходимо вывести текст, он заключается в апострофы. Оператор **WRITELN** отличается от оператора **WRITE** тем, что *после* вывода происходит переход на новую строку. Можно использовать оператор **WRITELN** без списка вывода для пропуска строки. Запишем пример программы, осуществляющей ввод и вывод :

```
VAR  i : Integer;
       w : Word;
       r : Real;
BEGIN
            WRITELN;
            {------------ ввод ------------}
            WRITE('Введите целое число ');
            READ(i);
            WRITELN;
            WRITE('Введите натуральное число ');
            READ(w);
            WRITELN;
            WRITE('Введите вещественное число ');
            READ(r);
            WRITELN;
            {------------ вывод ------------}
            WRITELN('Вы ввели : ',i,' ',w,' ',r,' их сумма=',i+w+r);
            WRITELN('Нажмите Enter для выхода');
            READLN;
END.
```

Впервые записав осмысленную программу, остановимся и обсудим ее внешний вид. Даже на таком тривиальном примере мы можем понять некоторые основные правила оформления программы.

1. Организация диалога с пользователем. Прежде чем записать оператор **READ,** вы обязаны записать хотя бы один **WRITE**, который выведет на экран *приглашение* "*Введите ...*", причем из этого приглашения пользователь должен понять, какие именно данные ему следует ввести. Так, в нашем примере операторы WRITE('Введите i '); READ(i); были бы неуместны, так как пользователю неизвестно, что такое **i**, и он мог бы ввести, например, вещественное число, что привело бы к аварийному завершению программы.

2. Оформление текста программы. Хорошо оформленная программа легко читается и быстрее отлаживается, следует стремиться к "прозрачности" текста, но не к некоторой, вполне субъективной, "красоте". Так, скажем, операторы, выполняющиеся последовательно, следует и записывать строго друг под другом, но не "елочкой" или какой-либо другой фигурой. Средства, используемые для оформления текста, крайне просты и доступны всякому - это пробелы, пустые строки и комментарии.

При выводе чисел можно их *форматировать*, т.е. управлять формой их представления. Для этого в списке вывода после выводимого выражения можно указывать *модификаторы* : “:*L*:*d*“ - для вещественных значений и “:*L*” для вещественных и целых. *L* и *d* - целочисленные выражения, первое из них определяет, сколько всего позиций отводится для выводимого числа на экране, а второе - сколько выводится цифр после десятичной точки. Если при выводе вещественного числа задан модификатор “:*L*:*d*“, то оно выводится с фиксированной точкой, если же задан модификатор “:*L*” или он отсутствует - то с плавающей точкой. Пусть значение переменной **X** равно **123.45678**, тогда оператор

```
WRITE(X);        выведет " 1.2345678000E+02"
WRITE(X:8:2);    выведет "  123.46"
WRITE(X:10:5);   выведет " 1.235E+02"
WRITE(X:10);     выведет " 1.235E+02"
WRITE(X:8);      выведет " 1.2E+02"
WRITE(X:1);      выведет " 1.2E+02"
```

По умолчанию вещественные числа всегда разделяются при выводе пробелами, но если вы выводите подряд несколько целых чисел, не форматируя их и не выводя между ними пробелов, они будут выводиться подряд и сольются в одно число.

5. Арифметические операции. Стандартные математические функции

Для арифметических данных, т.е. для числовых констант, переменных и числовых функций определены шесть арифметических операций:

+	сложение
-	вычитание
*	умножение
/	вещественное деление
DIV	целая часть от деления
MOD	остаток от деления

Первые четыре операции определены для любых операндов - как целых, так и вещественных, причем результат операции "/" всегда вещественное число, даже если оба операнда целые. Операции **DIV** и **MOD** определены только для целых операндов. Кроме того, выделяют *унарную* операцию "**-**", которая применяется не к двум, а к одному операнду, например: **-x**.

Вообще говоря, язык Паскаль запрещает использовать в одном выражении разнотипные операнды, однако для арифметических данных сделано исключение. Перед выполнением арифметической операции один или оба операнда автоматически приводятся к одному типу, а затем уже подставляются в выражение. Значение любого выражения всегда имеет определенный тип - такой же, как у операндов после приведения их к одному типу. Правила преобразования целочисленных типов приведены в таблице 2.

Таблица 2

Правила преобразования типов

Операнды	Byte	ShortInt	Word	Integer	LongInt
Byte	Integer	Integer	Word	Integer	LongInt
ShortInt	Integer	Integer	LongInt	Integer	LongInt
Word	Word	LongInt	Word	LongInt	LongInt
Integer	Integer	Integer	LongInt	Integer	LongInt
LongInt	LongInt	LongInt	LongInt	LongInt	LongInt

Если один операнд выражения имеет целочисленный тип, а второй - вещественный, то первый автоматически приводится к вещественному типу и значение выражения будет вещественным. Целые значения можно присваивать вещественной переменной, но вещественные значения присвоить целой переменной нельзя! Присваивая значение целочисленной переменной и константе, вы должны следить, чтобы это значение не выходило за пределы диапазона допустимых значений переменной. В языке Паскаль есть возможность явно преобразовать целочисленное значение к любому из целочисленных типов, для этого используются стандартные функции с именами **Byte**, **ShortInt**, **Word**, **Integer** и **LongInt**. Например, преобразуем переменную типа **Word** к типу **Integer** :

```
VAR x : Word;
BEGIN
        x:=300;
        WRITELN(x,' ',Integer(x));
        x:=65535;
        WRITELN(x,' ',Integer(x));
END.
```

Программа выведет:

```
300 300
65535 -1
```

В первом случае преобразование происходит корректно, а во втором - с изменением значения.

Арифметическое выражение может содержать любое количество операндов и, соответственно, любое количество операций, которые выполняются в последовательности, определенной их приоритетом; приоритет операций *, /, **DIV**, **MOD** выше, чем операций + и -. Операции одного приоритета выполняются слева направо. Чтобы изменить порядок выполнения операций, вы можете использовать в выражении *круглые скобки*. Вычислим, например, частное от деления **X** на сумму **A**,**B** и **C** :

```
X/(A+B+C);
```

Набор встроенных математических функций в языке Паскаль невелик, он включает :

1. **Abs**(*x*) - абсолютная величина числа.

2. **Int**(*x*) - целая часть вещественного числа.
3. **Frac**(*x*) - дробная часть вещественного числа.
4. **Trunc**(*x*) - целая часть вещественного числа, преобразованная к типу **LongInt.**
5. **Round**(*x*) - округленное до целого вещественное число, преобразованное к типу **LongInt.**
6. **Sqr**(*x*) - квадрат числа.
7. **Sqrt**(*x*) - квадратный корень.
8. **Exp**(*x*) - экспонента.
9. **Ln**(*x*) - натуральный логарифм.
10. **Pi** - число *пи*.
11. **Sin**(*x*) - синус.
12. **Cos**(*x*) - косинус.
13. **Arctan**(*x*) - арктангенс.

Все остальные математические функции можно получить, пользуясь этим основным набором; например: десятичный логарифм - **Ln**(x)/**Ln**(10), тангенс - **Sin**(x)/**Cos**(x) и т.д. Аргументы функций могут быть любыми арифметическими выражениями и задаются в круглых скобках после имени функции, аргументы функций **Sin** и **Cos** выражаются в радианах. Вычислим квадрат синуса 70 градусов: **Sqr(Sin(Pi/180*70))**

Кроме перечисленных выше математических функций Паскаль предоставляет еще несколько полезных числовых функций и процедур разного назначения:

14. **High** (*целый тип*) - возвращает наибольшее возможное значение данного типа.
15. **Low** (*целый тип*) - возвращает наименьшее возможное значение данного типа.
16. **SizeOf** (*тип*)

SizeOf (*переменная*) - возвращает размер в байтах заданного типа или заданной переменной. Функция **SizeOf** применима к любому типу, в том числе и к структурированным типам - массивам, записям и некоторым другим, речь о которых пойдет ниже.

17. **Random**(*Range*:**Word**) - возвращает целое случайное число в диапазоне от 0 до *Range*-1.
18. **Random** - возвращает вещественное случайное число в из отрезка [0,1].
19. **Randomize** - *процедура*, инициализирующая генератор случайных чисел, используя текущее системное время

Выведем несколько случайных чисел в диапазоне от 0 до 99:

```
BEGIN
        Randomize;
        WRITELN(Random(100));
        WRITELN(Random(100));
        WRITELN(Random(100));
END.
```

При первом запуске программы она вывела числа **13**, **38**, **48**, при втором запуске - **63**, **99**, **6**, при третьем запуске - **23**, **87**, **92**. Это действие процедуры Randomize - поскольку при каждом запуске системное время, которое отсчитывает операционная система DOS, было различным, мы каждый раз получали различные последовательности случайных чисел. Теперь исключим из программы оператор **Randomize;** и запустим ее несколько раз - каждый раз мы будем получать тройку чисел **0**, **3**, **86**.

Обратите внимание, что процедура используется в *операторе вызова*, а функция используется в *выражении*. Запись **Random(100);** неверна, поскольку **Random** - это функция, но также неверна и запись **WRITELN(Randomize);**. Можно считать, что различие между процедурой и функцией состоит в том, что процедура выполняет некоторую последовательность действий, а функция вычисляет некоторое значение. Заметим, что **READ** и **WRITE** - это тоже процедуры.

Для работы с внутренним двоичным представлением двухбайтовых целых чисел (типа **Word** или **Integer**) существуют функции:

20. **Lo**(*x*) - возвращает младший байт аргумента.
21. **Hi**(*x*) - возвращает старший байт аргумента.
22. **Swap**(*x*) - меняет местами младший и старший байты.

Сделаем отступление ***о двоичной системе счисления***. Все данные в памяти компьютера хранятся закодированными в двоичной системе. Любая переменная занимает целое число *байтов*, а каждый байт есть последовательность из 8 двоичных цифр - *битов*. Например, значение переменной типа **Byte**, равное **11**, хранится как последовательность битов **0000 1011**, а если переменная имеет тип **Word**, то ее значение кодируется как **0000 0000 0000 1101**. **1024** байта (или 2 в 10-й степени) имеют свое название - *1К байт*, иногда эту величину также называют *килобайт*; **1024 К** байт называют *мегабайт*. Пусть переменная **t** типа **Word** имеет значение **40000**, или **1001 1100 0100 0000** в двоичной системе, тогда функция **Lo(t)** возвратит **64** (=

0100 0000), функция **Hi(t)** возвратит **156** (= **1001 1100**) и функция **Swap(t)** возвратит **16540** (= **0100 0000 1001 1100**).

Для целочисленных *переменных* определены процедуры:

23. **Inc**(*x*)
    Inc(*x*,*d*)
24. **Dec**(*x*)
    Dec(*x*,*d*).

Здесь *x* - имя переменной, *d* - любое целочисленное выражение. Процедура **Inc** увеличивает значение переменной на *d*, а процедура **Dec** - уменьшает на *d*; второй аргумент этих процедур можно не задавать, тогда он будет принят равным 1. Например, вместо операторов a:=a+3; b:=b-1; c:=c+a+b; мы могли бы написать Inc(a,3); Dec(b); Inc(c,a+b); , и такой способ записи был бы предпочтительней.

6. Символьный тип данных

Для хранения символьной информации в Паскале предусмотрен специальный тип данных **Char**. Допустимы переменные, нетипизированные и типизированные константы такого типа. Данные типа **Char** занимают 1 байт памяти. Неименованные символьные константы записываются в программе либо в виде '*символ*', либо в виде #*номер*. Все имеющиеся символы пронумерованы от **0** до **255**, символы с 0-го по 31-й - невидимые, как правило, они не отображаются на экране, 32-й символ - это пробел. Приведем также номера некоторых других символов (хотя помнить эти номера нет никакой необходимости):

'0'...**'9'** - 48...57,
'A'...**'Z'** - 65...90,
'a'...**'z'** - 97...122,
'А'...**'Я'** - 128...159,
'а'...**'п'** - 160...175,
'р'...**'я'** - 224...239.

Некоторые из невидимых символов могут оказаться вам полезны: символ **#7** - "звуковой сигнал", при выводе пищит; символ **#10** - "конец строки", при выводе он перемещает текущую позицию вывода на одну строку вниз; символ **#13** - "возврат каретки" - перемещает текущую позицию вывода в начало текущей строки. Запомните, что клавиша **Enter** генерирует два символа - **#10** и **#13**, это может вам впоследствии пригодиться.

Символьные данные можно вводить и выводить процедурами **READ** и **WRITE** при вводе и выводе символьные значения изображаются *без апострофов*. Для символьных величин определены функции:

25. **Ord**(*c*) - возвращает номер символа.
26. **Pred**(*c*) - возвращает символ с номером, меньшим на 1.
27. **Succ**(*c*) - возвращает символ с номером, большим на 1.

Эти функция, однако, определены не только для символов, но для любого *порядкового типа данных*. Порядковым типом называется такой тип, все допустимые значения которого можно пронумеровать от 0 до некоторого N (в математике к этому понятию близко понятие *счетного множества*). Из известных нам типов порядковыми являются все целочисленные типы: **Byte**, **ShortInt**, **Word**, **Integer**, **LongInt** - и не являются порядковыми все вещественные типы. Значение функции **Ord** от числового аргумента равно самому этому аргументу, **Pred**(*x*) дает значение *x*-1, а **Succ**(*x*) - значение *x*+1. Функция

28. **Chr**(*n*).

в некотором смысле обратна функции **Ord** : для заданного числового аргумента *n* она возвращает символ с соответствующим номером. Для символьных переменных (так же, как и для любых переменных *порядкового* типа) определены процедуры **Inc** и **Dec**. Еще одна специфически символьная функция:

29. **UpCase**(*c*).

Она преобразует значение аргумента, если это маленькая латинская буква, в соответствующую заглавную букву. К сожалению, функция не работает для русских букв.

Напишем простую программу, обрабатывающую символьные величины.

```
VAR c : Char;   n : Byte;
CONST Blank  =' ';  Space:Char =Blank;
BEGIN WRITE('введите какой-нибудь символ ');  READ(c);
        WRITELN('вы ввели символ',Space,c,Space,'его номер=',Ord(c));
        WRITELN('соседние с ним символы :',Space,Pred(c),Space,
                  'и',Space,Succ(c));
```

```
        WRITELN('UpCase(',c,')=',UpCase(c));  WRITELN;
        Space:='';  WRITE('теперь введите число от 33 до 255 ');  READ(n);
        WRITELN('символ с номером ',n,' - это ',Space,Chr(n),Space);
        WRITELN;
END.
```

7. Логический тип данных. Операции сравнения. Логические операции. Битовые операции

Логические, или булевские, данные предназначены для хранения логических значений "истина" или "ложь". Логические переменные и константы имеют тип **Boolean** и занимают в памяти 1 байт. Существует всего две логические константы - **TRUE** и **FALSE**. Тип **Boolean** - это порядковый тип, поэтому для него определены функции **Ord**, **Pred**, **Succ** и процедуры **Inc** и **Dec** (впрочем, довольно редко применяемые), причем **Ord(FALSE)** =0, **Ord(TRUE)** =1. Прежде чем перейти к логическим операциям, рассмотрим операции сравнения, которых в Паскале существует шесть :

= равно;
<> не равно;
< меньше;
<= меньше или равно;
> больше;
>= больше или равно.

Операции сравнения определены для любых однотипных операндов (числовых, символьных, логических); для числовых данных, так же, как и в случае арифметических операций, сделано исключение - вы можете сравнивать два числовых выражения любых типов, но сравнивать число и символ, число и логическую величину, символ и логическую величину нельзя! Результат операции сравнения есть **TRUE** или **FALSE**, в зависимости от того, выполнено или не выполнено условие. Числа сравниваются между собой естественным образом, символы - в соответствии с их номерами, а для логических величин справедливо неравенство **FALSE**<**TRUE**. Логических, или булевских, операций в Паскале четыре :

NOT - логическое отрицание;
AND - логическое "и";
OR - логическое "или";
XOR - логическое исключающее "или".

Правила выполнения этих операций таковы :

NOT - унарная (т.е. применимая к одному операнду) операция :

NOT FALSE = **TRUE** , **NOT TRUE** = **FALSE** .

Правила выполнения бинарных операций **AND**, **OR** и **XOR** приведены в таблице 3.

Таблица 3

Правила выполнения бинарных операций

Операнд		Результат операции		
a	b	a AND b	a OR b	a XOR b
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	FALSE

Приоритет операции **NOT** (как и всякой унарной операции) наивысший, следующий приоритет у операции **AND**, и наинизший приоритет - у операций **OR** и **XOR**. Выражения могут содержать не только разные логические операции, но и операции сравнения и арифметические, поэтому отметим, что приоритет логических и арифметических операций выше, чем операций сравнения. Существует функция, определенная для целочисленных аргументов и имеющая логическое значение, - это функция

30. **Odd**(*x*).

Она возвращает **TRUE**, если значение *x* нечетное, и **FALSE**, если оно четное. Логические значения можно выводить процедурой **WRITE**, но вводить логические переменные процедурой **READ** нельзя. Теперь попробуем записать программу, использующую логические данные.

```
VAR a,b,c,d : INTEGER;
BEGIN WRITELN('Введите 4 целых числа, a,b,c и d, среди ',
                    'которых должно быть 2 и только 2 одинаковых!');
        WRITE('a='); READ(a); WRITELN;
        WRITE('b='); READ(a); WRITELN;
        WRITE('c='); READ(a); WRITELN;
        WRITE('d='); READ(a); WRITELN;
        WRITELN('Вашу понятливость можно оценить как ',
                    (a=b)AND(a<>c)AND(a<>d)AND(c<>d)OR
                    (a=c)AND(a<>b)AND(a<>d)AND(b<>d)OR
                    (a=d)AND(a<>b)AND(a<>c)AND(b<>c)OR
                    (b=c)AND(b<>a)AND(b<>d)AND(a<>d)OR
                    (b=d)AND(b<>a)AND(b<>c)AND(a<>c)OR
                    (c=d)AND(c<>a)AND(c<>b)AND(a<>b));
        READLN;
 END.
```

Программа выведет **TRUE**, если введенные данные удовлетворили условию, и **FALSE** - в противном случае.

Рассмотрим теперь *битовые* операции: **AND**, **OR**, **XOR**, **ShL** и **ShR**, которые определены для целочисленных операндов (операции **AND**,**OR** и **XOR** совпадают по написанию с логическими операциями, но последние определены только для логических операндов). Операции **AND**, **OR** и **XOR** выполняются над каждой парой соответствующих битов операндов по тем же правилам, что и логические операции, если нулевой бит считать ложным, а единичный - истинным. Приведем простой пример:

```
VAR a,b : BYTE;
BEGIN  a:=100;
          b:=200;
          WRITELN(a AND b,' ',a OR b,' ',a XOR b);
END.
```

Программа выведет числа **64**, **236**, **172**. Каким образом они получены? Двоичное представление числа **100** равно **0110 0100**, двоичное представление числа **200** равно **1100 1000**. Выполним над этими числами, например, операцию **XOR** :

```
0110 0100
1100 1000
XOR
1010 1100.
```

Получили двоичное число **1010 1100** = 128+32+8+4 = **172**.

Операции **ShL** и **ShR** называются операциями соответственно левого сдвига и правого сдвига. Они сдвигают биты первого операнда на количество разрядов, равное значению второго операнда, освободившиеся разряды заполняются нулевыми битами. Например:

```
10 ShL 3 = 80
40 ShR 3 = 5.
```

Число **10** кодируется как **0000 1010**; сдвинем биты влево на 3 разряда, получим **0101 0000** = 64+16 = **80**. Таким образом, сдвигая биты влево на n разрядов, мы умножаем число на 2 в степени n, а сдвигая вправо - делим на 2 в степени n. Этим свойством операций сдвига пользуются, когда в программе приходится часто умножать или делить целые числа на степень двойки, т.к. операции сдвига выполняются намного быстрее, чем арифметические операции.

8. Условный оператор. Блок. Оператор выбора

Условный оператор в Паскале записывается в виде:

IF *логическое выражение* **THEN** *оператор*/*блок* [**ELSE** *оператор*/*блок*] *логическое выражение* - это любое выражение, значение которого имеет тип **Boolean**, *блок* - это последовательность операторов, заключенная в *логические скобки* : **BEGIN** *операторы* **END**; . Перед **ELSE** никогда не ставится "**;**" ! Перед **END** в большинстве случаев можно не ставить "**;**". Если значение логического выражения **TRUE**, то выполняется оператор или блок, стоящий после **THEN**, в противном случае - оператор или блок, стоящий после **ELSE**.

Конструкция **ELSE** необязательна, условный оператор можно использовать и в усеченном виде, тогда при значении логического выражения **FALSE** не выполняется никаких действий. Операторы, входящие в условный оператор, сами могут быть условными, т.е. допускается любая вложенность условных операторов. Запишем теперь предыдущую задачу о четырех числах, используя оператор **IF** :

```
VAR a,b,c,d : Integer;
BEGIN WRITELN('Введите 4 целых числа, a,b,c и d, среди ',
                    'которых должно быть 2 и только 2 одинаковых!');
      WRITE('a='); READ(a); WRITELN; WRITE('b='); READ(a); WRITELN;
      WRITE('c='); READ(a); WRITELN; WRITE('d='); READ(a); WRITELN;
      IF(a=b)AND(a<>c)AND(a<>d)AND(c<>d) OR(a=c)AND(a<>b)AND(a<>d)AND(b<>d)OR
        (a=d)AND(a<>b)AND(a<>c)AND(b<>c)OR(b=c)AND(b<>a)AND(b<>d)AND(a<>d)OR
        (b=d)AND(b<>a)AND(b<>c)AND(a<>c)OR(c=d)AND(c<>a)AND(c<>b)AND(a<>b)
             THEN WRITELN('Вы довольно понятливы')
             ELSE   WRITELN('Вы ошиблись !!!');
           READLN;
       END.
```

Можно решить эту задачу и другим способом :

```
VAR a,b,c,d   : Integer;
CONST num : Byte = 0;
BEGIN WRITELN('Введите 4 целых числа, a,b,c и d, среди ',
                    'которых должно быть 2 и только 2 одинаковых!');
      WRITE('a='); READ(a); WRITELN;  WRITE('b='); READ(a); WRITELN;
      WRITE('c='); READ(a); WRITELN;  WRITE('d='); READ(a); WRITELN;
      IF a=b THEN Inc(num); IF a=c THEN Inc(num); IF a=d THEN Inc(num);
      IF b=c THEN Inc(num); IF b=d THEN Inc(num); IF c=d THEN Inc(num);
      IF num=1 THEN WRITELN('Вы довольно понятливы')
               ELSE WRITELN('Вы ошиблись !!!');
      READLN;
END.
```

Теперь попробуем записать условный оператор, реализующий более сложную логическую структуру. Пусть даны три числа **d**, **m** и **y**, содержащие число, месяц и год для некоторой даты; необходимо выяснить, правильна ли эта дата.

```
VAR d,m : Byte;  y : Word; Valid : Boolean;
BEGIN WRITE('Введите дату ');  READ(d,m,y);
      IF (m=1)OR(m=3)OR(m=5)OR(m=7)
              OR(m=8)OR(m=10)OR(m=12) THEN
         IF (d>=1)AND(d<=31) THEN Valid:=TRUE
                             ELSE Valid:=FALSE
      ELSE
      IF (m=4)OR(m=6)OR(m=9)OR(m=11) THEN
         IF (d>=1)AND(d<=30) THEN Valid:=TRUE
                             ELSE Valid:=FALSE
      ELSE
      IF m=2 THEN
        IF (d>=1)AND(d<=28) THEN Valid:=TRUE
      ELSE
       IF d=29 THEN
         IF (y MOD 4=0)AND(y MOD 100>0)
           OR(y MOD 400=0) THEN Valid:=TRUE
                           ELSE Valid:=FALSE
      ELSE Valid:=FALSE
      ELSE Valid:=FALSE;
      IF Valid THEN WRITELN('Дата верна')
               ELSE WRITELN('Дата не верна');
END.
```

Оператор выбора во многих случаях удобнее, чем условный оператор, он записывается в виде :

CASE *выражение* **OF**

список значений **:** *оператор/блок*
.....
список значений **:** *оператор/блок*
[**ELSE** *оператор/блок*]
END;

Здесь *выражение* - это любое выражение порядкового типа, *список значений* - это список разделенных запятыми *константных выражений* или *диапазонов*, диапазон - это конструкция вида *константное выражение* **..** *константное выражение*. Константным будем называть любое выражение, в которое входят только неименованные и нетипизированные константы (т.е. в константное выражение не могут входить имена переменных, функции и типизированные константы). На самом деле константные выражения - это выражения, которые могут быть вычислены *до* выполнения программы, а отсюда уже и вытекают ограничения на их вид. Выражение, стоящее после **CASE**, и все константные выражения должны быть одного типа либо все иметь целочисленные типы.

Оператор выбора выполняется следующим образом: вычисляется выражение, стоящее после **CASE**, затем просматриваются все списки значений, и если значение выражения попало в список значений, выполняется соответствующий оператор или блок, и выполнение оператора **CASE** заканчивается; если значение выражения не содержится ни в одном из списков, то выполняется оператор или блок, стоящий после **ELSE**. Конструкция **ELSE** может отсутствовать, в этом случае оператор CASE может не выполнить никаких действий. В качестве примера использования оператора выбора решим предыдущую задачу о правильной дате.

```
VAR d,m : Byte;   y  : Word;   Valid : Boolean;
BEGIN WRITE('Введите дату '); READ(d,m,y);
        CASE m OF
              1,3,5,7,8,10,12 :
                CASE d OF 1..31 : Valid:=TRUE
                          ELSE Valid:=FALSE
                END;
             4,6,9,11 :
                CASE d OF 1..30 : Valid:=TRUE
                          ELSE Valid:=FALSE
                END;
              2 : CASE d OF 1..28 : Valid:=TRUE;
                            29 : Valid:=(y MOD 4=0)AND(y MOD 100>0);
                           ELSE  Valid:=FALSE
                  END;
            ELSE Valid:=FALSE;
        END;
        IF Valid THEN WRITELN('Дата верна')
                 ELSE WRITELN('Дата не верна');
END.
```

Вы можете видеть, что задачи такого типа решаются оператором **CASE** гораздо проще, чем оператором **IF**. Решим еще одну задачу: определить, какого рода символ введен - цифра, латинская буква, русская буква или ни то, ни другое и ни третье.

```
VAR c : Char;
BEGIN WRITE('Введите символ ');  READ(c);
         CASE c OF
            '0'..'9'                  : WRITELN('Вы ввели цифру');
            ' a'..'z','A'..'Z'        : WRITELN('Вы ввели латинскую букву');
            'а'..'п','р'..'я','А'..'Я'  : WRITELN('Вы ввели русскую букву');
             ELSE                       WRITELN('Вы ввели неизвестно что !');
         END;
END.
```

9. Операторы цикла

Для реализации циклических алгоритмов, т.е. алгоритмов, содержащих многократно повторяющиеся одинаковые операции, применяются специальные *операторы цикла*. В Паскале есть три вида циклов: **FOR**, **WHILE** и **REPEAT**. Оператор цикла **FOR** записывается в виде:

FOR *переменная:=начальное значение* **TO** *конечное значение* **DO**

оператор/блок

или

FOR *переменная:=начальное значение* **DOWNTO** *конечное значение* **DO**

оператор/блок.

Здесь *переменная* - любая переменная порядкового типа, называемая в таком контексте *переменной цикла*, *начальное значение* и *конечное значение* - выражения того же типа (исключение, как всегда делается для разнотипных целочисленных переменных). Цикл **FOR** выполняется следующим образом: переменной цикла присваивается начальное значение, после чего выполняется *тело цикла* (оператор или блок, стоящий после **DO**). Два этих действия вместе составляют один *шаг цикла*. Затем переменной цикла присваивается следующее (в цикле **FOR ... TO**) или предыдущее (в цикле **FOR ... DOWNTO**) значение (вспомним функции **Succ** и **Pred**) и выполняется следующий шаг цикла. Так происходит до тех пор, пока значение переменной цикла не станет больше (**FOR...TO**) или меньше (**FOR...DOWNTO**) конечного значения. Цикл **FOR** может не выполниться ни разу, если начальное значение больше конечного в цикле **FOR...TO** или меньше конечного в цикле **FOR...DOWNTO**. Запишем два примера использования цикла **FOR** : вычислим сумму квадратов натуральных чисел от 1 до N.

```
VAR i : Word;
CONST s : Real = 0; N = 22;
BEGIN FOR i:=1 TO N DO s:=s+SQR(i); WRITELN('сумма=',s); END.
```

и выведем на экран символы с номерами от 32 до 255

```
VAR c : Char;
BEGIN FOR c:=' ' TO #255 DO WRITE(c); WRITELN; END.
```

Второй тип цикла - цикл **WHILE** - записывается в виде:

WHILE *логическое выражение* **DO** *оператор/блок*

Здесь *логическое выражение* - любое выражение типа **Boolean**. Цикл выполняется следующим образом : вычисляется логическое выражение и, если оно истинно, выполняется тело цикла, в противном случае цикл заканчивается. Очевидно, что цикл **WHILE** может как не выполниться ни разу, так и выполняться бесконечное количество раз (в последнем случае говорят, что *программа зациклилась*). Запишем две предыдущие задачи, используя цикл **WHILE** :

```
CONST i : Word = 1; s : Real   = 0; N = 22;
BEGIN WHILE  i<=N DO BEGIN  s:=s+SQR(i);  INC(i); END;
        WRITELN('сумма=',s);
END.

VAR c : Char;
BEGIN c:=Pred(' ');
        WHILE c<#255 DO BEGIN c:=Succ(c);  WRITE(c);  END;
        WRITELN;
END.
```

В качестве упражнения, подумайте, почему программа

```
VAR c : Char;
BEGIN c:=' ';
        WHILE c<=#255 DO BEGIN WRITE(c);  c:=Succ(c);  END;
        WRITELN;
END.
```

оказывается зацикленной.

Третий тип цикла - **REPEAT -** записывается в виде:

REPEAT *операторы* **UNTIL** *логическое выражение*;

Если тело цикла **REPEAT** содержит больше одного оператора, нет необходимости использовать блок, поскольку сами ключевые слова **REPEAT** и **UNTIL** являются в данном случае логическими скобками. Перед **UNTIL** можно не ставить "**;**". Цикл **REPEAT** выполняется так : сначала выполняется тело цикла, затем вычисляется логическое выражение, и если оно истинно, цикл заканчивается. Таким образом, цикл **REPEAT** всегда выполняется хотя бы один раз и так же, как и **WHILE**, подвержен зацикливанию. Запишем наши примеры циклом **REPEAT** :

```
CONST  i : Word = 1; Real   = 0;  N = 22;
BEGIN REPEAT s:=s+SQR(i); INC(i)  UNTIL i>N;
        WRITELN('сумма=',s);
END.

VAR c : Char;
BEGIN c:=Pred(' ');
        REPEAT c:=Succ(c); WRITE(c)  UNTIL c=#255;
        WRITELN;
END.
```

Из приведенных примеров очевидно, что любой циклический алгоритм можно записать любым видом цикла, все они взаимозаменяемы и выбираются программистом в соответствии с его вкусами, однако можно порекомендовать в тех случаях, когда количество шагов цикла известно заранее, использовать цикл **FOR**.

В последней версии языка Паскаль появились процедуры **BREAK** и **CONTINUE**, аналогичные операторам **break** и **continue** языка **C**. Процедура **BREAK** приводит к немедленному окончанию цикла, в котором она вызвана. Вызов процедуры **CONTINUE** приводит к немедленному переходу к следующему шагу цикла. Запишем наши примеры, используя **BREAK** :

```
CONST  i : Word  = 1;  s : Real = 0;  N = 22;
BEGIN WHILE TRUE DO BEGIN
                s:=s+SQR(i); INC(i);  IF i>N THEN BREAK;  END;
        WRITELN('сумма=',s);
END.

VAR c : Char;
BEGIN c:=Pred(' ');
        REPEAT  c:=Succ(c);  WRITE(c); IF c=#255 THEN BREAK UNTIL FALSE;
        WRITELN;
END.
```

Чтобы привести осмысленный пример использования процедуры **CONTINUE**, изменим условие второй задачи следующим образом: вывести на экран все символы с 32-го по 255-й, не являющиеся русскими буквами.

```
VAR c : Char;
BEGIN FOR c:=' ' TO #255 DO BEGIN
            IF (c>='A')AND(c<='Я')OR(c>='a')AND(c<='п')OR
               (c>='р')AND(c<='я')  THEN CONTINUE;
               WRITE(c);
         END;
        WRITELN;
END.
```

Впрочем, последнюю задачу, очевидно, можно решить проще:

```
VAR c : Char;
BEGIN FOR c:=' ' TO #255 DO BEGIN
                IF NOT((c>='A')AND(c<='Я')OR(c>='a')AND(c<='п')OR
                        (c>='р')AND(c<='я')) THEN WRITE(c);
          WRITELN;
 END.
```

10. Метки. Оператор GOTO. Процедура Halt

Операторы в Паскале могут быть помечены. *Метки* - это идентификаторы, или целые числа от 0 до 9999, они могут записываться перед любым выполняемым оператором и отделяются от него двоеточием. Оператор может иметь любое количество меток. Все метки, использованные в программе, должны быть описаны в разделе описаний с ключевым словом **LABEL**. В одном операторе **LABEL** можно описать несколько меток, тогда они разделяются запятыми. Оператор безусловного перехода

GOTO *метка***;**

передает управление оператору с соответствующей меткой, при этом все операторы, расположенные между оператором **GOTO** и оператором, которому передается управление, не выполняются. С помощью оператора **GOTO** нельзя передать управление : внутрь цикла, внутрь условного оператора и внутрь оператора выбора.

Общепризнано, что оператор **GOTO** является вредным оператором, он усложняет алгоритмы, затрудняет чтение программы и является источником ошибок. Постарайтесь не применять этот оператор в своих программах.

Одним из случаев, когда программисту может показаться полезным оператор **GOTO**, является необходимость прекратить выполнение программы при возникновении той или иной ошибки. Пусть, например, программа вычисляет некоторую функцию от квадратного корня из заданного числа:

```
VAR  x : Real;
BEGIN WRITE('Введите число ');  READ(x);
        x:=SQRT(x);
        {вычисление функции от x}
END.
```

Если введено отрицательное число, то в третьем операторе программы произойдет аварийное прерывание. Стремясь избежать этого, мы могли бы записать программу в виде:

```
VAR  x : Real;
LABEL Finish;
BEGIN WRITE('Введите число ');  READ(x);
        IF x<0 THEN GOTO Finish;
        x:=SQRT(x);
       {вычисление функции от x}
Finish:END.
```

Однако можно не использовать **GOTO** :

```
VAR x : Real;
BEGIN WRITE('Введите число '); READ(x);
        IF x<0 THEN WRITELN('ошибка !')
        ELSE BEGIN
                x:=SQRT(x);
                {вычисление функции от x}
        END;
END.
```

Как видите, программа даже стала лучше, т.к. теперь она сообщает о неправильном вводе. Но она все-таки имеет один недостаток - условный оператор усложнил структуру программы. В этом и в других подобных случаях очень удобно пользоваться стандартной процедурой Паскаля **HALT**, которая останавливает выполнение программы, будучи вызвана в любом ее месте:

```
VAR x : Real;
BEGIN WRITE('Введите число '); READ(x);
        IF x<0 THEN BEGIN WRITELN('ошибка !'); HALT; END;
        x:=SQRT(x);
        {вычисление функции от x}
END.
```

Наша программа стала почти идеальной. Доведем ее до совершенства :

```
VAR x : Real;
BEGIN REPEAT
                WRITE('Введите неотрицательное число ');
                READ(x);
                WRITELN;
        UNTIL x>=0;
        x:=SQRT(x);
        {вычисление функции от x}
END.
```

11. Интервальные типы данных. Оператор TYPE. Массивы

Интервальный тип - это некоторый подтип порядкового типа данных (вспомним, что порядковые типы - это **ShortInt**, **Byte**, **Integer**, **Word**, **LongInt**, **Char** и **Boolean**). Пусть, например, некоторая переменная в

программе может принимать значения от -1 до 99. Мы могли бы описать ее как **LongInt** или **Integer** (глупо!), могли бы описать ее как **ShortInt**, что достаточно разумно. Но можно создать для нее и специальный тип данных, объединяющий только числа от -1 до 99 :

VAR x : -1..99;

Вместо имени одного из стандартных типов мы использовали в описании переменной построенный нами собственный *интервальный* тип. Таким образом описанная переменная x может принимать только значения -1,0,1,...,99 , в остальном она ничем не отличается от других целых переменных. Ее можно вводить, выводить, использовать в качестве переменной цикла, подставлять в выражения и т.п. Любой интервальный тип есть подтип некоторого стандартного базового типа, в нашем случае - типа **ShortInt**. Но если бы мы стали использовать интервальный тип **-1..200** , то он бы уже был подтипом типа **Integer**, а **0..200** - подтипом типа **Byte**. Компилятор Паскаля самостоятельно анализирует интервальные типы и подбирает для них минимальный подходящий базовый тип. Это нужно знать, чтобы определять размер и способ кодировки ваших переменных. Вы можете выполнить оператор

WRITE('переменная x:-1..99 занимает ',SizeOf(x),' байт');

и убедиться, что ее размер действительно равен 1.

В качестве базового типа можно использовать не только арифметические типы, но и типы **Char** и **Boolean** (правда, в последнем случае это довольно бессмысленно). Опишем, например, переменную, значением которой могут быть только маленькие латинские буквы :

VAR Letter : 'a'..'z';

или переменную, в которой могут храниться русские буквы:

VAR RusLetter : 'A'..'я';

В общем случае интервальный тип описывается как

константное выражение 1 **..** *константное выражение* 2,

где оба выражения имеют один порядковый тип и второе из них не меньше первого. Созданным вами типам вы можете давать имена, для этого используется оператор **TYPE** :

TYPE *имя типа=описание типа***;**

Операторы **TYPE** так же, как и все другие операторы описания, записываются в разделе описаний. В программе может быть сколько угодно операторов **TYPE**, и их можно чередовать с другими операторами описания, но любые идентификаторы, использованные в *описании типа*, должны быть описаны раньше. После того, как некоторый тип получил *имя*, вы в дальнейшем можете пользоваться этим именем вместо полного описания типа :

```
CONST Tmin=-5;
      Tmax=15;
TYPE  T_Range_Type=Tmin..Tmax;
VAR   t:T_Range_Type;
TYPE  T_Range_SubType=Tmin+3..Tmax-5;
VAR   t1:T_Range_SubType;
```

Заметим, что хороший программист *всегда* дает имена собственным типам, причем старается, чтобы эти имена были осмысленными.

Теперь, зная об интервальных типах, мы можем говорить о массивах. Массив во всех языках программирования - это множество *индексированных* (пронумерованных) однотипных элементов. В Паскале описание *одномерного* массива имеет вид:

ARRAY [*тип индекса*] **OF** *тип элемента*

Здесь *тип индекса* - **ShortInt**, **Byte**, **Char**, **Boolean** или интервальный тип; *тип элемента* - любой тип, в том числе и массив. Вы заметили, что не все *порядковые* типы можно использовать как тип индекса, это не значит, что, например, тип **Word** чем-то хуже типа **Byte**. Такое ограничение обусловлено тем, что в Паскале никакой объект не может иметь размер больше (64К - 2) байта, или 65534 байта. Это ограничение действует и для интервальных типов, так вы можете описать массив VAR a : ARRAY[1..65534] OF BYTE;
но не массив VAR a : ARRAY[1..65535] OF BYTE;
и не массив VAR a : ARRAY[1..33000] OF WORD;

Больше никаких ограничений на тип индекса не накладывается. Тип элементов массива может быть любым - целочисленным, вещественным, символьным, логическим, интервальным. Элементы массива могут быть *массивами*, тогда вы получите массив размерностью больше чем 1. Опишем несколько массивов:

VAR a : ARRAY[Char] OF 1..5;

- массив из 256 элементов, каждый из которых есть целое число от 1 до 5, индексы элементов изменяются от #0 до #255;

```
CONST Max = 99;
```

```
        Min = 10;
    TYPE  Nums = Min..Max;
    TYPE  ArrayType = ARRAY[-10..0] OF Nums;
    VAR   a : ArrayType;
```
- массив из 11 элементов с индексами от -10 до 0, каждый элемент - целое положительное число из двух цифр;

```
    TYPE IndexType = 'a'..'z';
    VAR  a : ARRAY[IndexType] OF BOOLEAN;
```
- массив из 26 элементов с индексами от 'a' до 'z', каждый элемент - логическая переменная.

В программе вы можете использовать как массивы целиком, так и отдельные элементы массивов. Элемент одномерного массива записывается в виде:

имя массива **[** *индексное выражение* **]**

Индексное выражение - это любое выражение соответствующего типа. Если элемент массива - не массив, то с ним можно выполнять *любые операции*, разрешенные для *простых переменных* соответствующего типа. Целому массиву можно лишь *присваивать* массив того же типа. Заметим, что если массивы описаны в программе таким образом:

```
VAR a       : ARRAY[1..3] OF REAL;
    b,c,d : ARRAY[1..3] OF REAL;
TYPE Massiv=ARRAY[1..3] OF REAL;
VAR e,f  : Massiv;
    g    : Massiv;
    h,i : Massiv;
```

то массивы **b**,**c**,**d** - однотипные и массивы **e**,**f**,**g**,**h**,**i** тоже однотипные, но массивы **a** и **b** (**a** и **c**,**a** и **d**) имеют *разный тип*; и массивы **b** (**c**,**d**,**a**) и **e** (**f**,**g**,**h**,**i**) тоже имеют *разный тип*! Компилятор считает, что две переменные имеют один и тот же тип, только если они описаны в *одном операторе через запятую*, либо *имена* их типов одинаковы! Запомните это очень важное правило.

Запишем пример программы, использующей (пока одномерные) массивы:

```
{ программа вводит массив из N целых чисел, где N не превосходит 20, и выводит его в порядке неубывания
}
    CONST Nmax=20;
    TYPE  IndexType=1..Nmax;
          Massiv=ARRAY[IndexType] OF Integer;
    VAR a  : Massiv;  i,j,N : IndexType;  t  : Integer;
    BEGIN WRITELN;
          REPEAT  WRITE('Введите длину массива от 1 до ',Nmax,' ');
                  READ(N); WRITELN;
          UNTIL (N>=1)AND(N<=Nmax);
    { Вводим массив поэлементно }
          WRITELN('Введите элементы массива');
          FOR i:=1 TO N DO READ(a[i]);
    { Сортируем элементы массива по неубыванию. Используем очень простой,  но
      неэффективный алгоритм сортировки - сравниваем каждый элемент с  каждым
        и, если первый больше второго, меняем их местами }
          FOR i:=1 TO N-1 DO FOR j:=i+1 TO N DO
              IF a[i]>a[j] THEN BEGIN t:=a[i]; a[i]:=a[j]; a[j]:=t;  END;
    { Выводим отсортированный массив поэлементно }
          WRITELN('Результат сортировки :');
          FOR i:=1 TO N DO WRITE(a[i]:8);
    END.
```

Обратите внимание на алгоритм перестановки двух элементов! Запись **a[i]:=a[j]; a[j]:=a[i];** , очевидно, привела бы к неверному результату. Использованный нами алгоритм сортировки вполне надежен, но не очень хорош, так как выполняет много лишних операций. Не составляет труда усовершенствовать его - для каждого i от 1 до N-1 найдем наименьший из элементов a_i, a_{i+1}, ... , a_N и поместим его на i-е место; такой

алгоритм выполняет столько же сравнений, сколько и первоначальный, но требует существенно меньшего количества перестановок.

```
FOR i:=1 TO N-1 DO BEGIN
      a_max:=a[i];  n_max:=i;
      FOR j:=i+1 TO N DO
            IF a[j]<a_max THEN BEGIN a_max:=a[j]; n_max:=j;  END;
      IF n_max<>i THEN BEGIN a[n_max]:=a[i]; a[i]:=a_max; END;
END;
```

Как видите, запись алгоритма несколько длиннее, и потребовалось две новых переменных **a_max** - типа **Integer** и **n_max** - типа **IndexType**. Это действие универсального закона сохранения - из двух *верных* алгоритмов лучший, как правило, сложнее.

Теперь перейдем к рассмотрению многомерных массивов. *Размерностью*, или количеством измерений массива, называется количество индексов у элемента массива, но не количество элементов в массиве. Мы уже знаем, что элемент массива может быть массивом, поэтому двумерный массив можно описать, например, так :

```
VAR a : ARRAY[1..10] OF ARRAY[1..20] OF Real;
```

Переменную **a** можно рассматривать как одномерный массив одномерных массивов и использовать в программе запись вида **a[i]** ; но можно рассматривать и как двумерный массив вещественных чисел. Элемент этого массива записывается в программе в виде **a[** *индексное выражение* **,** *индексное выражение* **]** и является переменной типа **Real**, например, **a[i+1,3]**. Впрочем, можно записать и так: **a[i+1][3]**, обе эти записи эквивалентны. Описание многомерных массивов также можно записывать компактно: вместо

ARRAY[*t1* **] OF ARRAY[** *t2* **] OF ARRAY** ... **OF** *t* **;**

можно записать

ARRAY[t1 **,** t2 **,** ... **] OF** *t* **;**

Впрочем, бывают случаи, когда первое описание предпочтительней. Теперь, умея работать с многомерными массивами, напишем программу, перемножающую две квадратные вещественные матрицы:

```
CONST Nmax=20; {максимальный размер матрицы}
TYPE  IndexType=1..Nmax;
      Matrix   =ARRAY[IndexType,IndexType] OF Real;
VAR   a,b,c : Matrix;  n,i,j,k : IndexType;
BEGIN WRITE('введите размер матриц '); READ(n);
        IF (n<1)OR(n>Nmax) THEN BEGIN
           WRITELN('неверный размер!'); Halt; END;
        WRITELN('введите матрицу A построчно ');
        FOR i:=1 TO n DO FOR j:=1 TO n DO READ(a[i,j]);
        WRITELN('введите матрицу B построчно ');
        FOR i:=1 TO n DO FOR j:=1 TO n DO READ(b[i,j]);
        FOR i:=1 TO n DO FOR j:=1 TO n DO BEGIN
              c[i,j]:=0;  FOR k:=1 TO n DO c[i,j]:=c[i,j]+a[i,k]*b[k,j]; END;
        WRITELN('матрица A*B :');
        FOR i:=1 TO n DO FOR j:=1 TO n DO WRITE(c[i,j]);
        WRITELN;
END.
```

Наша программа сработала правильно, но полученную матрицу вывела плохо - все элементы подряд без деления на строки. Исправим алгоритм вывода:

```
FOR i:=1 TO n DO BEGIN
      FOR j:=1 TO n DO WRITE(c[i,j]:8);
      WRITELN;
END;
```

Теперь матрица выводится аккуратно.

В Паскале допускаются типизированные константы - массивы, список значений элементов массива задается в круглых скобках и разделяется запятыми:

```
CONST a : ARRAY[1..5] OF Byte=(1,2,3,4,5);
      c : ARRAY[0..3] OF Char=('a','b','c','d');
      b : ARRAY[-1..1] OF Boolean=(FALSE,TRUE,FALSE);
```

Символьные массивы можно инициализировать и более простым способом:

```
CONST c : ARRAY[0..3] OF Char='abcd';
```

Если инициализируется многомерный массив, то, поскольку каждый его элемент есть массив, нужно использовать вложенную скобочную структуру:

CONST A : ARRAY[1..3,1..2] OF Real = ((0,1),(2,4),(3,5));

Каким именно образом сгруппировать значения элементов, легко понять, вспомнив, что массив **ARRAY[1..3,1..2] OF Real** есть на самом деле компактная запись описания **ARRAY[1..3] OF ARRAY[1..2] OF Real**.

Итак, мы узнали, что кроме величин известных нам арифметических, символьного, логического типа и интервальных типов, каждая из которых имеет *одно* значение, существуют массивы - совокупности многих значений. Первые величины называются *скалярными*, а массивы и ряд других типов, пока нам не известных, *структурированными* величинами.

12. Ошибки при выполнении программы. Опции компилятора

Умея пользоваться массивами, условными операторами и операторами цикла, вы можете писать довольно серьезные программы. При выполнении этих программ неизбежно будут возникать критические ошибки, приводящие к аварийному завершению программы. Такие ошибки по английски называются Run-time errors - ошибки времени выполнения. Рассмотрим пока только наиболее часто встречающиеся арифметические ошибки:

Division by zero - код ошибки 200;
Arithmetic overflow - код ошибки 215;
Range check error - код ошибки 201;
Floating point overflow - код ошибки 205;
Invalid floating point operation - код ошибки 207.

Ошибка **Division by zero** - деление на ноль - возникает при выполнении операций **DIV**, **MOD** и /, когда делитель равен нулю.

Ошибка **Arithmetic overflow** - целочисленное переполнение - возникает при выполнении арифметической операции над целыми числами, когда результат операции выходит за границы соответствующего типа. Такая ошибка произойдет, например, при выполнении программы

VAR a,b : Word; c : Integer; BEGIN a:=100; b:=200; c:=a-b; END.

Ошибка произошла, когда вычислилось значение выражения **a-b**, равное -100. Мы знаем, что при выполнении операции над операндами типа **Word** результат будет иметь тип **Word**, а -100 не является допустимым значением этого типа. То обстоятельство, что это значение мы собирались присвоить переменной типа **Integer**, не имеет значения, т.к. ошибка произошла *до* присваивания. Интересно, что, если описать **a** и **b** как **Byte**, то ошибки не будет (см. таблицу 2 в главе 5).

Ошибка **Range check error** - ошибка проверки диапазона - происходит в двух случаях. Во-первых, при попытке присвоить целочисленной переменной недопустимое значение, и, во-вторых, при использовании недопустимого индексного выражения для элемента любого массива. Проиллюстрируем оба эти случая на простых примерах.

VAR a,b,c : Word; BEGIN a:=$FFFF; b:=1; c:=a+b; END.

Мы попытались присвоить переменной типа **Word** значение 65536, которое не является допустимым для этого типа.

VAR x : ARRAY[2..8] OF Real; i : Byte;
BEGIN FOR i:=8 DOWNTO 1 DO x[i]:=Sqrt(i); END.

Ошибка произошла при обращении к первому элементу массива, который не существует. Фактически этот второй случай полностью аналогичен первому - мы попытались "присвоить" индексу массива, тип которого-2..8, значение 1.

Ошибка **Floating point overflow** - вещественное переполнение - возникает при выполнении операции над вещественными числами, когда результат операции слишком велик, или при попытке присвоить вещественной переменной слишком большое значение. Когда речь идет о вещественных числах, термин "слишком большое" следует понимать как большое по абсолютной величине - знак числа не имеет значения. Приведем пример программы, содержащей такую ошибку.

VAR r : Real; BEGIN r:=-1E20; r:=Sqr(r); END.

При возведении в квадрат величины **r** мы получим слишком большое для типа **Real** число 1E40.

Ошибка **Invalid floating point operation** возникает в трех случаях:

1) при вычислении корня из отрицательного числа;
2) при вычислении логарифма неположительного числа;

3) при вычислении функций Trunc и Round от слишком большого (по абсолютной величине) вещественного числа. Эта ошибка довольно очевидна, и мы не станем ее иллюстрировать.

Как же должен поступать программист, когда при выполнении его программы возникают ошибки? Прежде всего нужно локализовать ошибку, то есть найти оператор, в котором она произошла. В этом вам может помочь среда Turbo Pascal, если в ней правильно установлены *опции компилятора*. Опции компилятора позволяют изменять режим компиляции и задаются в подменю **Compiler** меню **Options** среды Turbo Pascal. Пока нас будут интересовать лишь пять опций: **Range checking**, **Stack cheking**, **I/O checking**, **Overflow checking**, **Debug information**. Если они включены, то настройка среды благоприятна для отладки вашей программы. Если они выключены, то их обязательно следует включить, а еще лучше задать их непосредственно в тексте своей программы. Опции записываются в программе в виде:

{$ *буква* + / - **}**

Каждой опции соответствует своя буква (эти буквы выделены в подменю **Compiler** цветом), символ "+" означает включить, а символ "-" - выключить. В программе можно задать одну опцию, например, **{$R+}** или несколько опций - **{$R+,I-,S+}** . Некоторые опции можно записывать только в самом начале программы, другие могут размещаться в любом ее месте.

Опция **Range checking** (R) отвечает за контроль ошибок **Range check error**, **Overflow checking** (C) - за контроль ошибок **Ariphmetic overflow**, **I/O cheking** (I) - за контроль ошибок ввода-вывода. Смысл опции **Stack cheking** (S) будет объяснен несколько позже, а опция **Debug information** (D) включает в код программы отладочную информацию, что позволяет среде Turbo Pascal при аварийном завершении программы показать курсором оператор, в котором произошла ошибка. Позаботьтесь, чтобы при отладке программы перед первым ее оператором была строка **{$R+,C+,I+,S+,D+}** - это поможет вам найти и устранить все ошибки. Некоторые неопытные программисты выключают эти опции, тогда программа не прерывается при некоторых ошибках, а продолжает выполняться, на этом основании делается вывод, что программа верна. Это самообман - программа выполняется, но выполняется *неправильно* и никак не сообщает об ошибках.

13. Процедуры и функции. Сфера действия описаний

В языке Паскаль (как вы уже поняли из предыдущего материала) существуют понятия *процедуры* и *функции*. Процедуры и функции можно определить как замкнутые программные единицы, реализующие некоторый алгоритм. Фактически процедура или функция - это *почти программа*, почти - потому что она не может выполняться самостоятельно, а всегда *вызывается* какой-то другой процедурой или функцией. Программы, которые мы до сих пор писали, тоже были *процедурами*, правда, несколько особенными - *главными процедурами*. Программа может содержать любое количество процедур и функций, но она всегда содержит одну и только одну главную процедуру, с которой начинается выполнение программы.

Структура процедуры или функции очень похожа на структуру главной процедуры, она также содержит раздел описаний и раздел операторов; раздел операторов начинается с **BEGIN** и заканчивается **END;** (но не **END.** - как у главной процедуры). Единственным новым оператором для вас будет *оператор заголовка*, с которого начинается всякая процедура и функция. Все процедуры и функции записываются в разделе описаний какой-либо другой процедуры или функции, в том числе и главной процедуры. Оператор заголовка *процедуры* имеет вид:

PROCEDURE *имя* **(** *список параметров* **) ;**

Здесь *имя* - имя процедуры (любой идентификатор), список параметров может отсутствовать, но если он есть, записывается в круглых скобках после имени процедуры и имеет вид :

[**VAR**] *имя* **,** ... *имя* **:** *тип* **;**

.....

[**VAR**] *имя* **,** ... *имя* **:** *тип*

Здесь *имя* - *имена параметров*, каждый параметр может использоваться внутри процедуры как обычная переменная соответствующего типа. Тип - *имя* типа, но не описание пользовательского типа; скажем, описание параметра в виде **x:1..5** неверно, но, если выше описан соответствующий тип: **TYPE MyType=1..5**, то параметр можно описать в виде **x:MyType**. Ключевое слово **VAR** перед описанием параметров означает в данном случае, что все параметры до "**;**" или до "**)**" - *параметры-переменные*; если же **VAR** отсутствует, то параметры являются *параметрами-значениями*. Смысл этих понятий мы рассмотрим несколько позже.

Процедуры вызываются в других процедурах и функциях с помощью уже известного вам *оператора вызова*:

имя **(** *список аргументов* **);**

Список аргументов задается в том и только в том случае, когда в заголовке процедуры задан список параметров. Аргументы в списке разделяются запятыми и могут быть любыми выражениями, если соответствующий параметр есть параметр-значение, или только именами переменных, если соответствующий параметр есть параметр-переменная. Количество аргументов всегда должно совпадать с количеством параметров, и тип аргумента должен быть таким же, как тип параметра. При вызове процедуры значение соответствующих аргументов передается параметрам, и таким образом процедура получает информацию из вызывающей процедуры или функции. Запишем программу, использующую процедуру, которая будет аккуратно выводить значение переменной :

```
PROCEDURE OutVar(x:Real; Name:Char);
BEGIN WRITELN('Переменная ',Name,' равна ',x); END;
VAR a,b,c,d : Real;
BEGIN WRITE('Введите переменные a,b,c,d ');  READ(a,b,c,d);
        OutVar(a,'a');  OutVar(b,'b');  OutVar(c,'c');  OutVar(d,'d');
END.
```

Наша процедура OutVar получает из главной процедуры вещественное число **x** и символ **Name**, но ничего не передает обратно. Теперь попробуем написать процедуру, которая по заданным значениям *x* и *y* вычисляет cos(x)+cos(y) и cos(x)-cos(y) :

```
PROCEDURE T(x,y:Real; Cplus,Cminus:Real);
BEGIN Cplus:=cos(x)+cos(y); Cminus:=cos(x)-cos(y);  END;
VAR p,m:Real;
BEGIN T(1.235,0.645,p,m);  WRITELN(p:7:3,m:7:3);  END.
```

Запустим эту программу и - вместо правильного результата 1.129,-0.470 - получим в лучшем случае нули. Дело в том, что через параметры-значения (а **Cplus** и **Cminus** описаны в нашей процедуре как параметры-значения!) невозможно передать информацию *из* процедуры, но лишь *в* процедуру. Чтобы правильно решить нашу задачу, следует **Cplus** и **Cminus** описать в заголовке как параметры-переменные:

```
PROCEDURE T(x,y:Real; VAR Cplus,Cminus:Real);
BEGIN Cplus:=cos(x)+cos(y); Cminus:=cos(x)-cos(y); END;
```

Таким образом, *входные* параметры процедуры могут быть и параметрами -значениями и параметрами-переменными, а *выходные* параметры - только параметрами-переменными. Для того, чтобы глубже понять это правило, выясним, что же происходит с параметрами и аргументами при вызове процедуры. В момент вызова для каждого параметра-значения в специальной области памяти, называемой *стеком* (за контроль переполнения стека отвечает описанная выше опция компилятора **Stack cheking**), создается его копия - переменная соответствующего типа, которой присваивается значение аргумента. В дальнейшем процедура работает с этой копией, и при выходе из процедуры копия уничтожается. Таким образом, никакие изменения параметра-значения не могут быть известны за пределами процедуры. По-другому обрабатываются параметры-переменные: в процедуру передается не значение аргумента, а его *адрес*, и она работает с аргументом (теперь понятно, почему аргумент, соответствующий параметру-переменной, должен быть только *именем переменной*: он должен иметь *адрес*). Так что все изменения параметра на самом деле происходят с аргументом и известны в вызывающей процедуре.

Функция, в отличие от процедуры, всегда *вычисляет* некоторое значение *скалярного* типа, которое внутри функции должно быть присвоено имени функции. Заголовок функции имеет вид:

FUNCTION *имя* **(** *список параметров* **) :** *тип* **;**

В остальном функции аналогичны процедурам. Обращение к функции осуществляется с помощью *указателя функции*:

имя **(** *список параметров* **)**

Указатель функции может использоваться как и любое другое выражение того же типа, но это *не оператор*, в отличие от оператора вызова. Запишем пример функции:

```
FUNCTION Na3(x:LongInt):Boolean;
{ функция проверяет, делится ли x на 3 }
BEGIN Na3:=x MOD 3=0; END;
VAR L:LongInt;
BEGIN WRITE('Введите целое число '); READ(L);
        WRITE('Число ',L);
        IF NOT Na3(L) THEN WRITE(' не');
        WRITELN(' делится на 3 !');
END.
```

В любой процедуре и функции можно использовать *черезвычайно полезную* стандартную процедуру **Exit** без параметров для немедленного выхода в вызывающую процедуру.

Все процедуры и функции Паскаля являются *рекурсивными*, то есть могут вызывать сами себя, никаких усилий со стороны программиста для этого не требуется. В качестве примера запишем функцию, вычисляющую n!

```
FUNCTION Factorial(n:Byte):Real;
BEGIN IF n<=1 THEN Factorial:=1
                  ELSE Factorial:=n*Factorial(n-1);
END;
```

Но это, конечно, очень плохая функция, гораздо лучше записать этот алгоритм так:

```
FUNCTION Factorial(n:Byte):Real;
VAR i:Byte;  f:Real;
BEGIN f:=1; FOR i:=2 TO n DO f:=f*i;
      Factorial:=f;
END;
```

Итак, мы знаем, что программа может содержать много процедур и функций, и в каждой из них могут быть описаны типы, константы и переменные. Но не все из них могут быть использованы в любом месте программы, каждое описание имеет строго определенную *сферу действия*. Пусть процедура **A** находится внутри процедуры **B** - условимся называть процедуру **A** *внутренней* по отношению к **B**, а процедуру **B** - *объемлющей* по отношению к **A**. Если же ни процедура **A** не находится внутри **B**, ни **B** не находится внутри **A**, то эти процедуры - *внешние* по отношению друг к другу. Сфера действия описания любого объекта включает ту процедуру, где он описан (начиная с места описания) и все внутренние процедуры, если там данный идентификатор не описан. В принципе, это дает возможность передавать информацию в процедуры и функции, минуя параметры, то есть пользоваться во внутренней процедуре переменными, описанными в объемлющей процедуре, но такой стиль программирования считается ненадежным. Старайтесь, если это возможно, *все* переменные, используемые в процедуре, описывать в этой процедуре.

Для чего нужны процедуры и функции, когда и как их следует применять? Многие начинающие программисты избегают процедур и функций, утверждая, что "без них проще". На самом деле обойтись без функций и процедур легко только в самых тривиальных программах. Сколько-нибудь сложная программа, записанная "одним куском", требует при отладке от программиста огромных усилий, которые зачастую все равно пропадают даром. Обязательно используйте в своих программах процедуры и функции! Хорошая программа должна содержать главным образом обращения к процедурам и функциям. Конечно, не существует никаких жестких правил, определяющих, когда использовать функции, а когда нет, но автор этой книжки может предложить несколько нестрогих, но полезных рецептов:

- выделяйте в процедуру (функцию) небольшой логически завершенный фрагмент алгоритма;
- не смешивайте в одной процедуре (функции) ввод-вывод данных и вычислительные алгоритмы;
- называйте свои процедуры (функции) *мнемоническими именами*;
- если алгоритм, который вы решили выделить в процедуру (функцию), все еще слишком сложен, оформите фрагмент этого алгоритма в другой процедуре (функции);
- если алгоритм должен вычислить одно скалярное значение, пусть это будет функция, а не процедура;
- если в вашей программе встречаются многократно вложенные циклы или "многоэтажные" условные операторы, это верный признак, что вам нужны процедуры (функции);
- если текст вашей программы не умещается на одном экране - подумайте о процедурах;
- используйте в процедурах и функциях процедуру Exit.

14. Множества

Понятие *множества* в Паскале очень близко к математическому определению: множество - это совокупность однотипных *неиндексированных* объектов. Множества описываются в виде:

SET OF *тип* ,

где *тип* - базовый для этого множества тип, т.е. тип *элементов* множества. Базовый тип должен быть порядковым типом мощностью не более 256 (т.е. допускающий не более 256 различных значений), причем порядковые номера (вспомним функцию **ORD**) наименьшего и наибольшего значений должны лежать на отрезке [0,255]. Таким образом, базовым типом для множества могут быть: типы **Char**, **Boolean**, **Byte** и все производные от **Byte** интервальные типы. Размер объекта типа “множество” можно определить по формуле: *раз-*

мер = (*мощность*-1) **DIV** 8 + 1, т.е. множества - довольно компактные объекты, самое большое множество имеет размер 32 байта. Неименованные константы типа множество записываются в виде:

[*подмножество* **,** *подмножество* **,** ... **]** ,

где *подмножество* - это либо отдельное значение, либо *диапазон.* Диапазон записывается как *начальное значение* **..** *конечное значение*. Любое из значений может быть как константой, так и выражением соответствующего типа. Запишем, например, константу-множество, содержащую числа 0, 1, 2, 3, 4, 8, 12, 13, 14, 15, 16, 22:

[0,1,2,3,4,6,12,13,14,15,16,22]

или

[0..4,6,12..16,22]

или

[0..2,3..4,6..6,12,13..16,22]

или

[22,13..15,1..6,0,12,16]

Все эти константы полностью эквивалентны, порядок записи элементов совершенно произволен. Допускаются *пустые* множества, они записываются так: **[ ]**. Опишем несколько переменных и типизированных констант:

```
TYPE MySet = SET OF 0..111;
VAR a : SET OF Char;
      b : MySet;
CONST c : MySet = [];
        d : SET OF Char = ['А'..'Я'];
        e : SET OF Boolean = [FALSE];
```

К множествам применимы следующие операции:

- множеству можно *присвоить* множество того же типа;
- операция *объединение* +
- операция *дополнение* -
- операция *пересечение* *
- операция *эквивалентность* =
- операция *не эквивалентность* <>
- операция *включение* <= и >=

Последние три операции дают значения логического типа - **TRUE** или **FALSE**. Пусть множество **A=[1..20]** , **B=[2..9,15..20]** , **C=[3..22]** , тогда **A+B=[1..20]** , **A+C=[1..22]**, **A-C=[1,2]**, **C-A=[21,22]**, **A*B=[1..20]**, **A*C=[3..20]**, **B*C=[3..9,15..20]** , **A=B =FALSE** , **A<>C =FALSE** , **B<=A =TRUE** , **A>=C =FALSE.**

Существует еще одна операция, связанная с множествами, - операция **IN**, она применяется к скалярной величине и множеству:

выражение **IN** *множество*

Здесь *выражение* - любое выражение базового для данного множества типа, результат операции - **TRUE**, если такой элемент есть в множестве, и **FALSE** - в противном случае.

Для множеств определены две стандартные процедуры:

Include (*множество* , *выражение*)

Exclude (*множество* , *выражение*)

Процедура **Include** добавляет элемент, равный значению *выражения* в *множество*, а процедура **Exclude** удаляет такой элемент из *множества.*

Теперь, когда мы знаем все возможности множеств, запишем программу, находящую все простые числа на отрезке [1,255] :

```
  {программа использует алгоритм "решето Эратосфена"}
TYPE  NumSet = SET OF 1..255;
CONST N : NumSet=[2..255]; { исключили 1 как не являющуюся простым числом }
VAR MaxDivider,d : Byte;    k  : Word;
BEGIN  MaxDivider:=Round(SQRT(255));
          d:=2;
          WHILE d<=MaxDivider DO BEGIN
                    k:=2*d;
                    WHILE k<=255 DO BEGIN  Exclude(N,k);  INC(k,d);  END;
                   INC(d);
```

```
            END;
            WRITELN('Простые числа :');
            FOR k:=1 TO 255 DO IF k IN N THEN WRITE(k:4);
END.
```

Решим еще одну задачу : ввести массив символов и подсчитать, сколько в нем русских и латинских букв.

```
TYPE  Letters = SET OF Char;
CONST Lat =  ['a'..'z','A'..'Z'];  Rus = ['а'..'п','р'..'я','А'..'Я'];
VAR   c : Char;
CONST RSum : Word=0;   LSum : Word=0;
BEGIN WRITE('Введите массив символов, затем нажмите Enter ');
          REPEAT READ(c);
                    IF c IN Lat THEN INC(LSum) ELSE IF c IN Rus THEN INC(RSum);
          UNTIL c=#10;
           WRITELN('Латинских букв ',LSum,' русских букв ',RSum);
END.
```

Обратите внимание, что в этой задаче нет необходимости заранее знать, сколько символов содержится в массиве (более того, в программе никакого массива и нет!), достаточно лишь помнить, что клавиша **Enter** генерирует символ **#10**.

15. Тип STRING

Тип **STRING** - это строковый тип в Паскале. Строкой называется последовательность символов. Строковыми константами вы уже неоднократно пользовались - это последовательность любых символов, заключенная в апострофы; допускаются и *пустые* строки, они записываются так: **''**. Строковые переменные и типизированные константы описываются в виде

STRING

или

STRING [*максимальная длина* **]**

Если максимальная длина не задана, то по умолчанию она берется равной 255. Максимальная длина при описании строковых данных задается целочисленным константным выражением и никогда не может превышать 255. Это ограничение обусловлено самой структурой типа **STRING** : фактически строка - это массив **ARRAY [Byte] OF Char**, причем в 0-м символе закодирована текущая длина строки. Строковые переменные могут иметь любую длину от 0 до максимальной. В программе строки можно использовать и как единый структурированный объект (чуть позже мы познакомимся с разнообразными возможностями обработки строк), и как массив символов, т.е. обращаться к элементам строк следует так же, как к элементам массивов. Для строк определены следующие операции :

- строке можно присвоить строку;
- строки можно вводить процедурой **READLN**;
- строки можно выводить процедурой **WRITE**[**LN**];
- для строк определена операция *сцепления* +, при этом вторая строка дописывается справа к первой и длина результата становится равной сумме длин операндов (если она не превосходит 255).

Запишем программу, выполняющую простейшие операции со строками:

```
TYPE ShortString = STRING[80];
VAR s1,s2 : ShortString;    s3  : STRING;
BEGIN WRITE('Введите 1-ю строку '); READLN(s1);
        WRITE('Введите 2-ю строку '); READLN(s2);
        WRITELN('Вы ввели ',s1,' и ',s2); WRITELN('s1+s2=',s1+s2);
        s3:=s1+s1+s1; WRITELN('s1,повторенная 3 раза ',s3);
END.
```

Обратите внимание, что при вводе строк всегда используется **READLN**, но не **READ**. Процедура **READ** в отличие от **READLN** считывает лишь символы до символа конца строки (клавиша **Enter**), который остается в буфере клавиатуры. Таким образом, пользуясь процедурой **READ** можно ввести только одну строку; все строки, вводимые вслед за первой, станут *пустыми*. Например, программа

```
VAR s1,s2 : STRING;
BEGIN WRITE('Введите 1-ю строку '); READ(s1);
        WRITE('Введите 2-ю строку '); READ(s2);
        WRITELN('Вы ввели "',s1,'" и "',s2,'"');
END.
```

при входном потоке abcdef **Enter** 123456 **Enter** выведет : Вы ввели "abcdef" и "". Запишем теперь программу, которая вводит некоторую строку, заменяет в ней все цифры на пробелы и дописывает в конец строки символы "???":

```
VAR s    : STRING;  L,i : Byte;
BEGIN WRITE('Введите строку '); READLN(s);
        L:=ORD(s[0]);
        FOR i:=1 TO L DO IF s[i] IN ['0'..'9'] THEN s[i]:=' ';
        FOR i:=L+1 TO L+3 DO s[i]:='?';
        WRITELN('Вот что получилось : ',s);
END.
```

Наша программа заменила цифры, но никаких "?" не добавила. Дело в том, что, обращаясь к элементам строки, невозможно изменить *текущую длину* строки. Второй цикл нашей программы сработал правильно, записав символы "?" в соответствующие элементы строки, но длина строки осталась прежней, и процедура **WRITELN** вывела только символы с 1-го по L-й. Чтобы решить задачу корректно, мы могли бы добавить в программу один оператор INC(s[0],3); но, конечно, лучше всего просто записать: s:=s+'???'; .

Для обработки строк в Паскале существует несколько стандартных функций и процедур :

1. **FUNCTION Length**(*S*: **String**): **Integer**; - возвращает длину строки.

2. **FUNCTION Concat**(*S1*[,*S2*,...,*Sn*]: **String**): **String**; - возвращает строку, полученную сцеплением аргументов, может использоваться вместо операции "+".

3. **FUNCTION Pos**(*Substr*: **String**; *S*: **String**): **Byte**; - возвращает номер первого слева символа строки *S*, начиная с которого строка *Substr* входит в *S*, если *Substr* не входит в *S*, то значение функции равно 0.

4. **FUNCTION Copy**(*S*: **String**; *Index*: **Integer**; *Count*: **Integer**): **String**; - возвращает подстроку строки *S*, которая начинается с символа с номером *Index* и имеет длину *Count*.

5. **PROCEDURE Delete**(**VAR** *S*: **String**; *Index*: **Integer**; *Count*:**Integer**); - удаляет из строки *S* подстроку, начинающуюся с символа с номером *Index* и имеющую длину *Count*.

6. **PROCEDURE Insert**(*Substr*: **String**; **VAR** *S*: **String**; *Index*: **Integer**); - вставляет в строку *S* подстроку *Substr* начиная с символа с номером *Index*.

Из вышеизложенного понятно, что процедуры и функции могут иметь параметры типа **STRING** (что неудивительно), но также допустимы *функции типа* **STRING**, хотя это и не скалярный тип. Еще две стандартные процедуры предназначены для перевода строки в число и числа в строку:

7. **PROCEDURE Val**(*S*: **STRING**;**VAR** *V*; **VAR** *Code*: **Integer**); - преобразует строку *S* в число *V* (если это возможно); *V* - любая переменная арифметического типа, переменная *Code* возвращает 0, если преобразование прошло успешно, или номер первого неправильного символа строки.

8. **PROCEDURE Str**(*X* [**:** *Width* [**:** *Decimals*]];**VAR** *S*:**STRING**); - преобразует произвольное арифметическое выражение *X* в строку *S*, параметры *Width* и *Decimals* позволяют форматировать строку и имеют такой же смысл, как и в процедуре **WRITE**[**LN**] .

Теперь, зная процедуру **Val**, вы можете организовать надежный ввод числовых данных в любой своей программе. Предположим, что программа должна вводить вещественное значение **F**. Мы можем записать это так :

```
VAR F : Real;  ...  BEGIN WRITE('Введите F '); READ(F);
```

Если пользователь правильно введет число, то все будет в порядке, но если он ошибочно нажмет не ту клавишу (например, запятую вместо точки и т.п.), то произойдет аварийное прерывание, программа завершится, и на экране появится сообщение "**Run-time error** ...". Программы, таким образом реагирующие на неверный ввод, - *плохие*! Хорошая программа обязана обрабатывать нажатие практически любых клавиш в любых комбинациях. Мы вполне способны написать такую программу :

```
VAR F : Real;   S : STRING;  Code : Integer; ...
BEGIN  REPEAT
                        WRITE('Введите F '); READLN(S);
                        Val(S,F,Code);  IF Code=0 THEN Break;
                        WRITELN('Ошибка ввода!');
          UNTIL FALSE;
```

Решим часто встречающуюся задачу *о распаковке текста*: дана строка, содержащая текст на русском языке (или на любом другом языке, в том числе и искусственном - вы увидите, что это не принципиально); нужно выделить слова, содержащиеся в этом тексте. Хотя эта задача и элементарна, ее решение не столь тривиально и требует предварительной разработки алгоритма. Сначала уясним, что такое текст. *Текстом* будем называть последовательность *слов*, разделенных любым количеством "*пробелов*". Слова - это последовательности *букв языка* (в нашем случае - русских букв), "*пробелы*" - любые символы, не являющиеся буквами. Итак, наш текст в общем случае имеет вид : ****X*X...*X* ,*** где ***X*** - слово, ******* - "пробел". Можно предложить следующий алгоритм распаковки:

1) удалим завершающие пробелы, после чего текст примет *регулярный* вид ****X*X...*X***;

2) удалим лидирующие пробелы;

3) выделим первое слово и удалим его из текста.

После выполнения пунктов 2 и 3 мы получили одно слово и текст стал короче на одно слово, сохранив при этом свою структуру. Очевидно, что пункты 2 и 3 следует выполнять до тех пор, пока текст *не пуст*. Запишем программу, реализующую этот алгоритм.

```
VAR s : STRING;  i : Byte;
CONST Letters : SET OF Char = ['a'..'п','р'..'я','A'..'Я']; {это алфавит}
BEGIN WRITE('Введите текст '); READLN(s);
        { удалим завершающие пробелы, здесь есть 1 ОШИБКА! }
        WHILE NOT(s[Length(s)] IN Letters) DO Delete(s,Length(s),1);
        WRITELN('Слова текста :');
        { организуем цикл ПО СЛОВАМ }
        WHILE s<>'' DO BEGIN
                { удалим лидирующие пробелы }
```

```
                WHILE NOT(s[1] IN Letters) DO Delete(s,1,1);
                { найдем границу первого слова, здесь есть 1 ОШИБКА! }
                 i:=1; WHILE s[i] IN Letters DO INC(i);
                { i - номер первого пробела }
                Dec(i);
                { выведем слово }
                WRITELN(Copy(s,1,i));
                { удалим слово из текста }
                Delete(s,1,i);
        END;
END.
```

На первый взгляд наша программа работает правильно (мы ввели фразу на русском языке и получили все слова из нее), но тестирование программы обязательно должно включать все *предельные*, или *особенные*, случаи. Введем, например, строку, не содержащую никаких слов, и программа *зациклится*! Это результат ошибки в первом цикле: если в тексте нет букв, все символы из него будут удалены, длина строки станет равной нулю, и в дальнейшем станет проверяться символ с номером 0, который равен **#0** и, естественно, не является буквой. Еще одна ошибка подобного рода *может* произойти при выделении последнего слова: мы увеличиваем индекс **i**, пока **i**-й символ - буква, и в конце концов дойдем до конца строки. Но переменная **s** всегда содержит 255 символов, символы с номерами **Length(s)+1**, **Length(s)+2** и т.д. существуют, и нет никаких гарантий, что они не являются русскими буквами. В этом случае мы можем получить последнее слово с "хвостом". Исправим нашу программу:

```
VAR s : STRING;  i : Byte;
CONST Letters : SET OF Char = ['a'..'п','р'..'я','A'..'Я']; {это алфавит}
BEGIN WRITE('Введите текст '); READLN(s);
        { удалим завершающие пробелы }
        WHILE NOT(s[Length(s)] IN Letters)AND(s<>'') DO Delete(s,Length(s),1);
        IF s='' THEN BEGIN  WRITELN('текст пуст');  Halt;  END;
        WRITELN('Слова текста :');
        { организуем цикл ПО СЛОВАМ }
        WHILE s<>'' DO BEGIN
                { удалим лидирующие пробелы }
                WHILE NOT(s[1] IN Letters) DO Delete(s,1,1);
                { найдем границу первого слова }
                 i:=1;  WHILE (s[i] IN Letters)AND(i<=Length(s)) DO INC(i);
                { i - номер первого пробела }
                Dec(i);
                { выведем слово }
                WRITELN(Copy(s,1,i));
                { удалим слово из текста }
                Delete(s,1,i);
        END;
END.
```

Теперь запишем то же самое, используя функции и процедуры :

```
VAR s : STRING;  i : Byte;
CONST Letters : SET OF Char = ['a'..'п','р'..'я','A'..'Я']; {это алфавит}

PROCEDURE DelTail(VAR s:STRING);
BEGIN WHILE NOT(s[Length(s)] IN Letters)AND(s<>'') DO Delete(s,Length(s),1); END;

PROCEDURE DelLead(VAR s:STRING);
BEGIN WHILE NOT(s[1] IN Letters) DO Delete(s,1,1); END;

FUNCTION MakeWord(s:STRING; VAR Bound:Byte):STRING;
BEGIN Bound:=1;
        WHILE (s[Bound] IN Letters)AND(Bound<=Length(s)) DO INC(Bound);
        Dec(Bound); MakeWord:=Copy(s,1,i); END;
```

```
BEGIN WRITE('Введите текст '); READLN(s);
        { удалим завершающие пробелы }
        DelTail(s);
        IF s='' THEN BEGIN WRITELN('текст пуст');  Halt; END;
        WRITELN('Слова текста :');
        { организуем цикл ПО СЛОВАМ }
        WHILE s<>'' DO BEGIN
                { удалим лидирующие пробелы }  DelLead(s);
                { выведем слово } WRITELN(MakeWord(s,i));
                { удалим слово из текста }  Delete(s,1,i);
        END;
END.
```

16. Графические средства языка Паскаль

Монитор персонального компьютера может работать в двух режимах: текстовом и графическом. Все, что вы делали до сих пор, вы делали в текстовом режиме. Текстовый экран содержит **2000** *знакомест* - **25** строк по **80** позиций, в каждом знакоместе может быть выведен один символ. Графический экран состоит из маленьких точек - *пикселов*, каждый из которых закрашен в какой-либо цвет. Для работы в графическом режиме существует обширная библиотека процедур и функций, находящихся в *модуле* **Graph**. Структуру модуля и правила создания пользовательских модулей мы рассмотрим несколько позже. Чтобы использовать стандартные модули, вам достаточно знать лишь один оператор :

USES *модуль* **,** ... **;**

Этот оператор должен быть *первым* оператором в программе, в нем перечисляются все модули, используемые данной программой; в частности, чтобы работать с графикой, вам достаточно записать **USES Graph;**. Теперь рассмотрим графические средства, предоставляемые этим модулем. Здесь описаны только наиболее употребительные и наиболее полезные, по мнению автора, средства. Тот, кто хочет изучить *все* возможности модуля **Graph**, может сделать это, пользуясь справочной службой среды Turbo Pascal (или Borland Pascal).

1. **PROCEDURE InitGraph**(**VAR** *GraphDriver*,*GraphMode*: **Integer**; *PathToDriver*: **STRING**); - эта процедура инициализирует графический режим, т.е. переключает монитор из текстового режима в графический. Любые графические процедуры и функции могут быть выполнены только в графическом режиме. Перед вызовом **InitGraph** необходимо первому аргументу присвоить значение **DETECT** (константа, описанная в модуле **Graph**). *PathToDriver* - это строка, содержащая путь к файлу - графическому драйверу, для мониторов от **EGA** до **SVGA** это файл **EGAVGA.BGI**. Графические драйверы всегда содержатся в директории, где находится сам Turbo Pascal (обычно в поддиректории **BGI**). Вы можете либо отыскать на диске этот файл и в программе правильно задать путь к нему, например, **'D:\TP\BGI'**, либо иметь этот файл в вашей рабочей директории, тогда путь задается пустой строкой. Если ни то, ни другое не сделано, графический режим не будет инициализирован.

2. **PROCEDURE CloseGraph**; - закрывает графический режим.

3. **FUNCTION GetMaxX** : **Integer**; .

4. **FUNCTION GetMaxY** : **Integer**; - возвращают соответственно номер самого правого и самого нижнего пиксела экрана. Пикселы нумеруются от **0** до **GetMaxX** слева направо и от **0** до **GetMaxY** сверху вниз. Разрешение графического экрана зависит от типа монитора и от выбранного графического режима. Например, для монитора **VGA** максимальное разрешение **640 × 480**, т.е. **GetMaxX** вернет **639**, а **GetMaxY - 479**.

5. **PROCEDURE SetBkColor**(*Color*: **Word**); - устанавливает фоновый цвет, после ее выполнения весь экран будет закрашен в цвет *Color*. Цветовая палитра также зависит от типа монитора и выбранного графического режима, но стандартная палитра для цветного монитора включает 16 цветов:

0 - **Black**	1 - **Blue**	2 - **Green**	3 - **Cyan**
4 - **Red**	5 - **Magenta**	6 - **Brown**	7 - **LightGray**
8 - **DarkGray**	9 - **LightBlue**	10 - **LightGreen**	11 - **LightCyan**
12 - **LightRed**	13 - **LightMagenta**	14 - **Yellow**	15 - **White**

Приведенные названия - это имена констант, описанных в модуле **Graph**; вы можете использовать их или номера цветов.

6. **PROCEDURE SetViewPort**(*x1*,*y1*,*x2*,*y2*: **Integer**; *Clip*: **Boolean**); - устанавливает графическое окно. *x1*,*y1*,*x2*,*y2* - координаты соответственно левого верхнего и правого нижнего углов окна. После выполнения

этой процедуры пикселы будут отсчитываться от левого верхнего угла окна. Логический параметр *Clip* определяет, следует ли усекать изображения на границах окна. Выполнять эту процедуру вовсе не обязательно, по умолчанию графическое окно занимает весь экран.

7. **PROCEDURE ClearDevice**; - закрашивает экран фоновым цветом.

8. **PROCEDURE PutPixel**(*X,Y*: **Integer**; *Color*: **Word**); - закрашивает пиксел с координатами *X,Y* цветом *Color*.

9. **FUNCTION GetPixel**(*X,Y*: **Integer**): **Word**; - возвращает цвет пиксела с координатами *X,Y*.

10. **PROCEDURE SetColor**(*Color* : **Word**); - устанавливает цвет линий, все выводимые на экран линии будут иметь цвет *Color* до выполнения следующей процедуры **SetColor**.

11. **PROCEDURE SetLineStyle**(*LineStyle*, *Pattern*, *Thickness*: **Word**); - устанавливает *стиль линий*, действует для всех выводимых линий до выполнения **SetLineStyle** с другими аргументами. Параметр *LineStyle* может принимать следующие значения:

0 - **SolidLn** - сплошная линия;
1 - **DottedLn** - пунктирная линия;
2 - **CenterLn** - штрих-пунктирная линия;
3 - **DashedLn** - штриховая линия;
4 - **UserBitLn** - линия, задаваемая программистом.

Если стиль линии - **4**, то форма линии определяется вторым параметром процедуры - *Pattern*. Толщина линии может принимать всего два значения:

1 - **NormWidth** - тонкая линия;
3 - **ThickWidth** - жирная линия.

12. **PROCEDURE Line**(*x 1,y1,x2,y2*: **Integer**); - рисует отрезок прямой от точки с координатами *x1,y1* до точки *x2,y2*.

13. **PROCEDURE MoveTo**(*x,y*: **Integer**); - перемещает графический курсор в точку *x,y*. Графический курсор не виден на экране, но ряд процедур использует текущее положение графического курсора.

14. **PROCEDURE LineTo**(*x,y*: **Integer**); - рисует отрезок от текущей точки (текущего положения графического курсора) до точки *x,y*.

15. **PROCEDURE MoveRel**(*Dx,Dy*: **Integer**);- перемещает графический курсор на *Dx* по горизонтали и на *Dy* по вертикали.

16. **PROCEDURE LineRel**(*Dx,Dy*: **Integer**); - рисует отрезок от текущей точки до точки со смещением *Dx,Dy*.

17. **FUNCTION GetX**: **Integer**; и

18. **FUNCTION GetY**: **Integer**; - возвращают текущие координаты графического курсора.

19. **PROCEDURE Rectangle**(*x1,y1,x2,y2*: **Integer**); - рисует прямоугольник, *x1,y1* - координаты левого верхнего угла, *x2,y2* - координаты правого нижнего угла.

20. **PROCEDURE Circle**(*X,Y*: **Integer**; *R*: **Word**); - рисует окружность радиуса *R* с центром в точке *X,Y*.

21. **PROCEDURE Ellipse**(*X,Y*: **Integer**; *f1,f2,Rx,Ry*: **Word**); - рисует дугу эллипса с полуосями *Rx,Ry* и центром в точке *X,Y* от угла *f1* до угла *f2* (углы задаются в градусах).

22. **PROCEDURE Arc**(*X,Y*: **Integer**; *f1,f2,R*: **Word**); - рисует дугу окружности радиуса *R* с центром в точке *X,Y* от угла *f1* до угла *f2*.

23. **PROCEDURE SetFillStyle**(*Pattern,Color*: **Word**); - устанавливает способ закраски. Параметр *Pattern* может принимать следующие значения: 0 - **EmptyFill** - не закрашивать, 1 - **SolidFill** - сплошная закраска, 2 - **LineFill**, 3 - **LtSlashFill**,
4 - **SlashFill**, 5 - **BkSlashFill**, 6 - **LtBkSlashFill**,7 - **HatchFill**, 8 - **XHatchFill**,
9 - **InterleaveFill**, 10 - **WideDotFill**, 11 - **CloseDotFill**.

24. **PROCEDURE Bar**(*x1,y1,x2,y2*: **Integer**); - рисует закрашенный прямоугольник, используя способ закраски, установленный процедурой **SetFillStyle**.

25. **PROCEDURE FillEllipse**(*X,Y*: **Integer**; *Rx,Ry*: **Word**); - рисует закрашенный эллипс.

26. **PROCEDURE Sector**(*X,Y*: **Integer**; *f1,f2,Rx,Ry*: **Word**); - рисует закрашенный эллиптический сектор.

27. **PROCEDURE PieSlice**(*X,Y*: **Integer**; *f1,f2,R*: **Word**); - рисует закрашенный круговой сектор.

28. **PROCEDURE FloodFill**(*X,Y*: **Integer**; *Border*: **Word**); - закрашивает замкнутую область, ограниченную линией цвета *Border*, *X,Y* - координаты любой внутренней точки области. Используется способ закраски "заливка жидкостью", поэтому, если ограничивающая линия имеет разрывы, "жидкость" выльется и закрасит все области экрана, которые сможет. Автор рекомендует самостоятельно провести эксперимент с этой процедурой.

29. **PROCEDURE SetTextStyle**(*Font*,*Direction*,*Size*: **Word**); - устанавливает способ вывода текста. *Font* - номер графического шрифта, принимающий значения 0 - **DefaultFont** , 1 - **TriplexFont** , 2 - **SmallFont** , 3 - **SansSerifFont** ,
4 - **GothicFont**. Нулевой шрифт - стандартный и поддерживается всегда. Если вы используете *штриховые шрифты* с 1-го по 4-й, то должны иметь в вашей рабочей директории шрифтовые файлы **TRIP.CHR** , **LITT.CHR** , **SANS.CHR** , **GOTH.CHR** (те из них, которые вам нужны). Параметр *Direction* определяет направление вывода текста (слева направо или сверху вниз) и принимает значения 0 - **HorizDir** ,1 - **VertDir**. Параметр *Size* определяет размер символов и изменяется от **1** до **10**.

30. **PROCEDURE OutText**(*S*: **STRING**); - выводит текст на графический экран, используя текущие координаты графического курсора (процедура **WRITE**[**LN**] в графическом режиме не работает).

31. **PROCEDURE OutTextXY**(*X*,*Y*: **Integer**; *S*: **STRING**); - выводит текст на графический экран, используя координаты *X*,*Y*.

32. **PROCEDURE SetTextJustify**(*Horiz*, *Vert*: **Word**); - устанавливает способ позиционирования текста. Параметр *Horiz* может принимать значения:

0 - **LeftText** - по левому краю,
1 - **CenterText** - по середине текста,
2 - **RightText** - по правому краю.

Параметр *Vert* может принимать значения:

0 - **BottomText** - по нижнему краю,
1 - **CenterText** - по середине текста,
2 - **TopText** - по верхнему краю.

Не пренебрегайте этой процедурой, если хотите аккуратно вывести подписи к вашему рисунку.

33. **FUNCTION TextWidth**(*S*: **STRING**): **Word**; - возвращает длину текста в пикселах.

34. **FUNCTION TextHeight**(*S*: **STRING**): **Word**; - возвращает высоту текста в пикселах.

35. **PROCEDURE SetVisualPage**(*Page* : **Word**); - устанавливает видимую графическую страницу (если в данном графическом режиме есть несколько видеостраниц). *Page* - номер страницы, равный 0,1 и т.д.

36. **PROCEDURE SetActivePage**(*Page* : **Word**); - устанавливает текущую графическую страницу, куда будет направлен весь вывод. Две последние процедуры могут быть использованы для создания мультипликации.

37. **PROCEDURE SetGraphMode**(*Mode*: **Integer**); - устанавливает *графическую моду*. Большинство графических *драйверов* допускает несколько мод. Какой графический драйвер задействован в данном компьютере, можно узнать по значению параметра *GrDriver* после выполнения процедуры **InitGraph**. Присваивая этой переменной значение **DETECT**, мы не задаем никакого драйвера, а лишь указываем, что процедура сама должна определить этот драйвер. В Паскале определены следующие константы драйверов: **DETECT**=0, **CGA**=1, **MCGA**=2, **EGA**=3, **EGA64**=4, **EGAMONO**=5, **IBM8514**=6, **HERCMONO**=7, **ATT400**=8, **VGA**=9, **PC3270**=10. При успешном выполнении процедура **InitGraph** возвратит одно из этих значений через параметр *GrDriver*. Параметру *GrMode* присваивается значение установленной графической моды (от 0 до 4), причем устанавливается *старшая мода*. У драйвера **VGA** есть три моды, различающиеся разрешением экрана и количеством видеостраниц:

0 - 640 × 200, 4 страницы,
1 - 640 × 350, 2 страницы,
2 - 640 × 480, 1 страница.

Именованные константы для графических мод также описаны в модуле **Graph**; так, для перечисленных выше мод это: **VGALo**, **VGAMed**, **VGAHi**.

38. **FUNCTION GetGraphMode** : **Integer**; - возвращает установленную графическую моду.

39. **PROCEDURE RestoreCrtMode**; - устанавливает текстовый режим монитора. Эта процедура совместно с **SetGraphMode** может использоваться для отладки графических программ. Предположим, что мы написали, но пока еще не отладили графическую программу. Мы хотим вывести какую-либо информацию, вычисляемую программой, на экран, но использовать для вывода процедуру **OutText** довольно затруднительно. Организуем нашу программу следующим образом :

```
... InitGraph ...  { здесь мы хотим вывести информацию }  RestoreCrtMode;  WRITELN(...
    { вернемся в графику }  SetGraphMode(GetGraphMode);  ...
```

40. **FUNCTION GraphResult** :**Integer**; - возвращает код завершения последней графической операции; если этот код равен **grOK** (=**0**), то операция выполнена успешно, в противном случае произошла ошибка.

Чтобы продемонстрировать некоторые из графических возможностей языка Паскаль, напишем программу, рисующую график функции $\cos^2 x$ на отрезке $[0,6\pi]$.

```
USES Graph;
```

```
CONST ScreenColor = DarkGray;  {цвет экрана}
      LineColor   = Yellow;      {цвет кривой}
      TextColor   = White;       {цвет подписей}
      AxisColor   = LightCyan;  {цвет координатных осей}
CONST n           = 200; {количество отрезков в графике}
      LeftBlank   = 100; {отступ слева}
      RightBlank  = 100; {отступ справа}
      TopBlank    = 100; {отступ сверху}
      BottomBlank = 60 ;  {отступ снизу}
      TicSize     = 5;    {размер делений на осях}
      PowerSize   = 3;    {размер цифры 2 (показатель степени)}
      TicsNumY    = 10;   {количество делений на оси Y}
      TicsNumX    = 6;    {количество делений на оси Y}
CONST x1=6*Pi;
FUNCTION f(x:REAL):Real; BEGIN f:=Sqr(Cos(x)); END;
VAR GrDriver,GrMode,Lx,Ly,Px,Py : Integer;
    i                           : Word;
    s                           : STRING;
    Mx,My,x                     : Real;
BEGIN {инициализируем графический режим}
      GrDriver:=DETECT;  InitGraph(GrDriver,GrMode,'');
      {закрасим экран фоновым цветом}
      SetFillStyle(1,ScreenColor);  Bar(0,0,GetMaxX,GetMaxY);
      {вычислим длины осей и положение начала координат}
      Lx:=GetMaxX+1-LeftBlank-RightBlank;
      Ly:=GetMaxY+1-TopBlank-BottomBlank;
      Px:=LeftBlank;  Py:=GetMaxY-BottomBlank;
      {нарисуем оси}
      SetColor(AxisColor);
      MoveTo(LeftBlank,TopBlank-1); LineRel(0,Ly);  LineRel(Lx,0);
      {оцифруем ось X}
      SetTextJustify(CenterText,TopText);
      FOR i:=1 TO TicsNumX DO BEGIN
            MoveTo(LeftBlank+i*Lx DIV TicsNumX,Py); LineRel(0,TicSize);
            IF i=1 THEN s:='' ELSE Str(i,s);  MoveRel(0,2); OutText(s+'Pi');
      END;
      {оцифруем ось Y}
      SetTextJustify(RightText,CenterText);
      FOR i:=1 TO TicsNumY DO BEGIN
            MoveTo(Px,Py-i*Ly DIV TicsNumY); LineRel(-TicSize,0);
            Str(i/10:3:1,s); MoveRel(-2,0); OutText(s);
      END;
      {выведем пояснительный текст}
      SetTextJustify(CenterText,CenterText);  SetColor(TextColor);
      MoveTo(LeftBlank+Lx DIV 2,TopBlank DIV 2);
      OutText('график функции Cos(x)');
      {выведем показатель степени}
       MoveRel(TextWidth('график функции Cos(x)') DIV 2-TextWidth('(x)'),TextHeight('s'));
      SetTextStyle(SmallFont,0,PowerSize); OutText('2');
      {вычислим масштабы по X и Y}
      Mx:=Lx/x1;  My:=Ly;
      {нарисуем график}
      SetLineStyle(0,0,3);  SetColor(LineColor);  MoveTo(Px,Py-Ly);
      FOR i:=1 TO n DO BEGIN
            x:=i*x1/n;  LineTo(Px+Round(Mx*x),Py-Round(My*f(x)));
     END;
     {сделаем задержку}
     READLN;
```

```
        {перейдем в текстовый режим}
        CloseGraph;
END.
```

Это неплохая программа, но автор хотел бы предостеречь от ее бездумного использования как эталона во всех случаях. Она использует априорную информацию о виде изображаемой функции: минимальное значение функции = 0, максимальное значение функции = 1, начало координат находится в точке пересечения осей и т.п. В ваших программах все может быть не так.

17. Кое-что о вещественных вычислениях

В отличие от целочисленных выражений, которые всегда вычисляются *точно*, вещественные выражения дают *приближенный* результат, и вещественные переменные содержат приближенные значения. Например, выполним программу

```
VAR x : Real; BEGIN x:=1/3;  WRITELN(x*3-1); END.
```

Мы получим не 0 (точное значение выводимого выражения), а, например, **4.547E-13**, а может быть, какое-нибудь другое маленькое число. Это обусловлено тем, что переменные типа **Real** хранят *конечное* число десятичных цифр (11-12 цифр), кроме того, эти цифры хранятся в *двоичном коде*, поэтому мы и не получили **1E-12** или **1E-13**. Таким образом, **x/a*a** далеко не всегда равно **x**. И наоборот, **a+x** может быть равно **a**, даже если **x** не равно нулю. Найдем такое положительное число, которое удовлетворяет уравнению **x+1=1** :

```
CONST x:Real=1; BEGIN WHILE x+1<>1 DO x:=x/2; WRITELN(x); END.
```

Мы получим, например, **5.42E-20** (результат зависит от типа компьютера).

Решим реальную задачу, в которой используются приближенные вычисления : вычислить сумму ряда $\sum x^i/i!$, $i=0,1,\ldots,\infty$. Несмотря на то, что необходимо просуммировать *бесконечное* число слагаемых, эта задача легко решается за конечное время, так как общий член ряда быстро убывает и начиная с некоторого **n** прибавление очередного слагаемого уже не будет изменять сумму. Сначала напишем плохую программу:

```
FUNCTION Factorial(n:Word):Real;
VAR i:Word; f:Real;
BEGIN f:=1; FOR i:=1 TO n DO f:=f*i; Factorial:=f; END;

FUNCTION Power(x:Real; n:Woed):Real;
VAR i:Word; f:Real;
BEGIN IF n=0 THEN Power:=1
        ELSE BEGIN f:=x;  FOR i:=2 TO n DO f:=f*x; Power:=f;  END;
END;

VAR   x,S1,S2 : Real;
CONST  i : Word = 0;
BEGIN WRITE('Введите x '); READ(x);
         S2:=0;
         REPEAT S1:=S2; S2:=S1+Power(x,i)/Factorial(i); INC(i); UNTIL S1=S2;
        WRITELN('Сумма ряда = ',S1);
END.
```

Запустим эту программу, задав x=**1**; мы получим верный результат **2.71828**... (его легко проверить, поскольку сумма нашего ряда равна **exp(x)**). А теперь введем x=**10**, ожидая получить exp(10)=**22026.4**... Однако ничего подобного не случилось! Turbo Pascal выбросил нас обратно в среду, выдал сообщение "**Error 205: Floating point overflow**" и показал курсором строку

```
FOR i:=1 TO n DO f:=f*i;
```

Это аварийное прерывание - *переполнение*, оно происходит всякий раз, когда вещественная величина превышает максимально допустимое значение **1.7E38**. Следовательно, для некоторого **i** мы уже не можем вычислить **i!** .

Означает ли это, что невозможно вычислить exp(10)? Вовсе нет; конечно, мы не можем задать нашей программе очень большой x, но значение **10** вполне приемлемо, дело здесь в качестве нашей программы. Действительно, посмотрим, как работает программа : для $i=0$ вычисляется x^0 и $0!$, затем для $i=1$ заново вычисляется x^1 и $1!$ и т.д. до получения результата; но $x^{i+1}=x*x^i$ и $(i+1)!=(i+1)*i!$, так что, зная предыдущие значения, достаточно выполнить всего одну операцию, чтобы получить последующие. Более того, нам вовсе не нужен факториал сам по себе, а только общий член ряда (в котором этот факториал находится *в знаменателе*). Нетрудно записать рекуррентную формулу для общего члена ряда : $a_0=1;\ a_i=x/i*a_{i-1}$.Кроме того, что избавимся от переполнения, пользуясь этой формулой, мы во много раз увеличим скорость нашей программы (не всегда функции бывают полезны).

```
VAR   x,S1,S2,a : Real;  CONST  i : Word = 0;
BEGIN WRITE('Введите x ');  READ(x);
        a:=1;  S2:=0;
        REPEAT S1:=S2;  S2:=S1+a;  INC(i);  a:=a*x/i; UNTIL S1=S2;
        WRITELN('Сумма ряда = ',S1);
END.
```

Программа сработала для **x=10** и **x=20** и **x=50**, но для **x=100** снова произошло переполнение. Но здесь уже ничего сделать нельзя, **exp(100)**>10^{43} и никак не может быть представлена вещественным значением типа **Real**. Решим еще одну, более содержательную, задачу: найти корень уравнения **f(x)=0** методом *бисекции*. Метод бисекции заключается в следующем: пусть уравнение **f(x)=0** имеет единственный корень на отрезке **[a,b]** , это значит, что график функции один раз пересекает ось **X** на этом отрезке. Определим знак функции в точке *a* и в точке **x=(a+b)/2**. Если эти знаки одинаковы, то, очевидно, корень лежит на отрезке **[x,b]** , в противном случае - на отрезке **[a,x]** . Таким образом, за один шаг метода мы ровно вдвое уменьшили наш отрезок; будем повторять эти операции до тех пор, пока отрезок не станет очень маленьким, и в качестве корня возьмем середину этого маленького отрезка. Попробуем реализовать этот метод:

```
CONST a : Real=0;  b : Real=10;
FUNCTION f(x:Real):Real; BEGIN f:=exp(x)-2; END;
CONST epsilon=1E-10; {"очень маленький" отрезок}
VAR x,Fa,Fx : Real;
BEGIN Fa:=f(a); {нет необходимости вычислять f(a) в цикле, т.к.ЗНАК
                                функции на левом конце отрезка не изменится! }
        WHILE b-a>epsilon DO BEGIN
                x:=(a+b)/2;  Fx:=f(x);
                IF Fx=0 THEN BEGIN  WRITELN(x); Halt; END;
                IF Fa*Fx<0 THEN b:=x  ELSE a:=x;
        END;
        WRITELN(x);
END.
```

Программа вычислила **0.693147**. Легко проверить, что это правильный результат (корень уравнения равен ***ln2***). Казалось бы, мы написали хорошую программу, но давайте теперь вычислим корень уравнения ***ln(x)-50=0***, ***a=1***, ***b=1e30*** -программа *зациклится* ! Выведем внутри цикла значения **a** и **b** : эти числа почти одинаковы (отличие в 12-й цифре) и не меняются, но поскольку их порядок 10^{21}, **b-a** существенно превосходит наш "очень маленький" отрезок. Есть два способа, которыми мы можем исправить программу. Первый заключается в правильном подборе "очень маленького" отрезка, но надо понимать, что вам придется подбирать этот отрезок *для каждого* нового уравнения, то есть фактически для каждого уравнения писать свою программу. Очевидно, что это бесперспективный путь. Выведем в нашей зацикленной программе не только **a** и **b**, но и **x**, может быть, это поможет нам придумать второй способ: значение **x**, оказывается, в точности равно **b**.

Мы могли бы прийти к выводу, что рано или поздно **x** станет равным или **a** или **b**, рассуждая чисто теоретически. Действительно, на каждом шаге цикла мы уменьшаем отрезок в два раза; если бы мы работали на вещественной оси, то величины **a** и **b** стремились бы друг к другу бесконечно, но, поскольку множество вещественных чисел в компьютере *дискретно* (из-за конечного числа цифр), настанет момент, когда между **a** и **b** больше *не будет ни одного числа*. После этого выражение **(a+b)/2** будет давать либо **a**, либо **b**. Воспользуемся этим обстоятельством и напишем *хорошую* программу:

```
CONST a : Real = 1;  b: Real = 1E30;
FUNCTION f(x:Real):Real; BEGIN f:=Ln(x)-50; END;
VAR x,Fa,Fx : Real;
BEGIN Fa:=f(a);  x:=(a+b)/2;
        WHILE (x<>a)AND(x<>b) DO BEGIN
                Fx:=f(x);
                IF Fx=0 THEN BEGIN  WRITELN(x); Halt;  END;
                IF Fa*Fx<0 THEN b:=x  ELSE a:=x;
                x:=(a+b)/2;
        END;
        WRITELN(x);
END.
```

Программа дала верный результат **5.184705...E21**.

Решим еще одну задачу: вычислить значения функции

f(x)=ln(1+ln(1+exp(exp(x))))

на отрезке **[0,1000]** с шагом **5**.

```
CONST x0 = 0;   x1 = 1000;   h  = 5;
FUNCTION f(x:Real):Real; BEGIN f:=ln(1+ln(1+exp(exp(x)))); END;
VAR i : Byte;  x : Real;
```

```
BEGIN FOR i:=0 TO Round((x1-x0)/h) DO BEGIN
                x:=x0+i*h;  WRITELN('x=',x:4:0,' f(x)=',f(x));  END;
END.
```

При **x=10** произошло переполнение. Означает ли это, что задача неразрешима? Нет, мы просто написали плохую программу, скопировав математическую формулу в оператор Паскаля. Посмотрим, в каком месте происходит переполнение - очевидно, при вычислении **exp(exp(x))** , других возможностей просто не существует. Это значит, что полученное значение **exp(exp(x))** превосходит **1E38**. Посмотрим на аргумент внутреннего логарифма: прибавление единицы к очень большому числу никак не изменит это число, следовательно, этой единицей можно пренебречь.Таким образом, для **x≥5** наша формула *упрощается*:

f(x)=ln(1+ln(1+exp(exp(x))))=ln(1+ln(exp(exp(x))))=ln(1+exp(x))

Исправим программу:

```
FUNCTION f(x:Real):Real;
BEGIN IF x<5 THEN f:=ln(1+ln(1+exp(exp(x))))  ELSE f:=ln(1+exp(x));
END;
```

Снова произошло переполнение, но теперь при **x=90**. Очевидно, что причина та же - не удалось вычислить **exp(x)**. Еще раз упростим формулу для **x≥90** : **f(x)=ln(1+exp(x))=ln(exp(x))=x -** и запишем, наконец, верную программу:

```
FUNCTION f(x:Real):Real;
BEGIN  IF x<5 THEN f:=ln(1+ln(1+exp(exp(x))))  ELSE
          IF x<90 THEN f:=ln(1+exp(x))  ELSE f:=x;
END;
```

Сделаем некоторые выводы из вышесказанного. Компьютерные вычисления несколько отличаются от абстрактных математических вычислений. В математике вещественная ось *непрерывна* (между двумя любыми вещественными числами находится бесконечное множество чисел) - компьютерное множество вещественных чисел *дискретно*. Математика оперирует с *бесконечно большими* и *бесконечно малыми* величинами - компьютерные вещественные числа *ограничены* сверху и снизу. Математические вычисления *точны* - компьютерные вычисления *приближенны*. Вы должны учитывать это, когда программируете какую-либо вычислительную задачу.

18. Записи

Записи - это объекты, объединяющие данные разных типов. Записи состоят из *полей*. Описание записи имеет вид:

RECORD *описания полей* **END;**

где *описания полей* - обычные описания переменных. Имена полей могут совпадать с именами других переменных, описанных в программе, и с именами полей других записей. Поля могут иметь любой тип, в том числе могут быть записями. Опишем несколько записей:

```
VAR c : RECORD
            Name  : STRING[40];
            Phone : STRING[20];
        END;
TYPE CoordType = RECORD
                     x,y : Integer;
                 END;
VAR a : ARRAY[1..3] OF CoordType;
TYPE PointType = RECORD
                     c     : CoordType;
                 Color : Word;
                 END;
VAR T : PointType;
```

Чтобы обратиться в программе к полю записи, нужно использовать *составное имя*: *имя записи.имя поля*, например: **c.Name[25]** , **a[2].x** , **T.c.y**. Поэтому имена полей и не обязаны быть *уникальными*. Поля записей можно использовать точно так же, как и простые переменные того же типа. Однотипные записи можно присваивать друг другу. Иногда полные имена полей бывают довольно громоздкими и усложняют текст программы, этого можно избежать, используя оператор **WITH**, который записывается в виде:

WITH *имя записи* **DO** *оператор*/*блок*

Внутри оператора **WITH** можно использовать *простые* имена полей данной записи. Однако будьте осторожны: если имена полей не уникальны, любые имена, совпадающие с именем какого-нибудь поля записи, внутри оператора **WITH** всегда интерпретируются как имена полей, хотя вы, возможно, хотели использовать здесь имя простой переменной. Типизированные константы - записи - допускаются и инициализируются в виде:

CONST *имя*:*тип*=(*имя поля*:*значение*; *имя поля*:*значение*; ...);

Опишем, например, константу типа **PoinType** :

```
CONST p : PointType = (c:(x:100; y:20); Color:15);
```

Выведем эту константу:

```
WRITELN(p.c.x,p.c.y,p.Color);
```

или

```
WITH p DO WRITELN(c.x,c.y,Color);
```

или

```
WITH p.c DO WRITELN(x,y,p.Color);
```

Теперь решим более содержательную задачу : даны **n** точек на плоскости, каждая из которых окрашена в свой цвет, упорядочить точки по цвету, а точки одного цвета - по неубыванию радиус-вектора. Будем считать, что **n** не превосходит **100**.

```
TYPE CoordType = RECORD  x,y : REAL; END;
        PointType = RECORD  XY:CoordType; Color:Word; R:Real; END;
VAR p  : ARRAY[1..100] OF PointType;  Min : PointType;  n,i,j,Num : Byte;
BEGIN WRITE('Введите количество точек '); READ(n);
        FOR i:=1 TO n DO BEGIN
              WRITE('Введите ',i:2,'-ю точку ');
              WITH p[i] DO READ(XY.x,XY.y,Color);
        END;
        FOR i:=1 TO n DO WITH p[i] DO R:=Sqrt(Sqr(XY.x)+Sqr(XY.y));
        FOR i:=1 TO n-1 DO BEGIN
            Min:=p[i];  Num:=i;
            FOR j:=i+1 TO n DO WITH p[j] DO
                IF (Color<Min.Color) OR (Color=Min.Color) AND (R<Min.R) THEN BEGIN
                   Min:=p[j]; Num:=j;  END;
            IF Num<>i THEN BEGIN p[Num]:=p[i]; p[i]:=Min; END;
    END;
    WRITELN('Упорядоченное множество точек :');
    FOR i:=1 TO n DO WITH p[i] DO  WRITELN(XY.x:10:1,XY.y:10:1,Color:5);
END.
```

В Паскале записи могут содержать так называемые *вариантные поля*, в которых могут храниться одновременно (в одном и том же месте памяти) данные разных типов. Описание вариантного поля имеет вид:

CASE *тип* **OF**
константа 1 **: (***описание поля***);**
константа 2 **: (***описание поля***);**
.....

Здесь *тип* - любой порядковый тип (фактически компилятор Turbo Pascal'я никак не использует этот тип, поэтому можно всегда писать, например, **Byte**); *константа 1*, *константа 2* и т.д. - это любые константы порядкового типа (их значение также не используется компилятором). Запись может иметь только одно вариантное поле, и это поле должно быть последним. Память для размещения вариантного поля отводится по самому большому варианту. Приведем пример использования такой записи:

```
VAR z : RECORD x,y : Integer;
                    CASE Byte OF
                          0       : (L : LongInt);
                          TRUE : (W : ARRAY[0..1] OF WORD);
                          'D'     : (B0: Byte; B : ARRAY[0..3] OF Byte);
                          -22     : (S : STRING);
          END;
BEGIN  z.S:='12345'; WITH z DO WRITELN(B0,B[0]:3,B[1]:3,B[2]:3,B[3]:3,'  ',S);
          z.L:=1000;   WITH z DO WRITELN(L,' ',W[0],' ',W[1]);
END.
```

Программа вывела : **5 49 50 51 52 12345**
1000 1000 0

Действительно, в первом байте вариантного поля первоначально хранилась длина строки - 5, а в следующих четырех - символы **'1'**,**'2'**,**'3'**,**'4'**, т.е. байты **49-52**. После выполнения второго оператора присваивания в младшем слове оказалось число **1000**, а в старшем - **0**. Фактически вариантное поле занимает **256** байт памяти, но при обращении к нему по именам **L** и **W** нам доступны первые **4** байта, а при обращении по имени **B** - первые **5** байт.

Таким образом, вариантное поле - это некоторый участок памяти, к которому можно обращаться *с разными именами*, каждое имя однозначно связано *с определенным типом*. Значение, хранящееся в этом участке памяти, конечно, в каждый момент времени совершенно определенное и не зависит от способа обращения, но *интерпретация* этой последовательности байтов зависит от используемого типа. Сам способ описания вариантного поля, который в нашем примере имеет вполне “дикий” вид (это сделано намеренно), остался в языке Паскаль от его ранних диалектов, в которых все конструкции имели совершенно определенный смысл. Сейчас компилятору они не нужны, но тем не менее их необходимо записывать, чтобы не нарушать синтаксис языка.

19. Тип "перечисление"

Тип "перечисление" записывается в виде:
(*идентификатор*$_1$**,...,***идентификатор*$_N$**)**

Идентификаторы, использованные при описании типа, автоматически становятся *константами* этого типа. Можно использовать в программе переменные и именованные константы типа "перечисление". К ним применимы функции **ORD**, **PRED** и **SUCC**. Переменной можно присвоить значение ее типа; такие переменные могут быть переменными цикла; тип “перечисление” может быть базовым типом множества. Но переменные типа “перечисление” нельзя вводить и выводить, они не могут быть преобразованы ни в какой другой тип. Попробуем использовать тип "перечисление":

```
TYPE Months = (Jan,Feb,Mar,Apr,Mai,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
   {идентификаторы Jan...Dec стали КОНСТАНТАМИ типа Months}
VAR M : Months;
CONST MM : Months = Mar;
TYPE M_Set = SET OF Months;
CONST Sem1 : M_Set = [Sep..Dec];  Sem2 : M_Set = [Feb..Mai];   Sess   : M_Set = [Jan,Jun];
CONST Year : BYTE=0;
BEGIN {определим, что за месяц MM}
          IF MM IN Sem1 THEN WRITELN('1-й семестр') ELSE
          IF MM IN Sem2 THEN WRITELN('2-й семестр') ELSE
          IF MM IN Sess   THEN WRITELN('сессия')         ELSE
                                WRITELN('каникулы');
   {посчитаем продолжительность учебного года}
   FOR M:=Jan TO Dec DO IF M IN Sem1+Sem2+Sess THEN INC(Year);
   WRITELN('Учебный год длится ',Year,' месяцев');
END.
```

20. Модуль CRT. Общие принципы организации интерфейса

CRT - еще один стандартный модуль Turbo Pascal'я, в котором содержатся разнообразные средства ввода-вывода. Процедуры и функции **CRT** помогут вам организовать хороший интерфейс в ваших программах. Кроме процедур и функций любой модуль может также содержать описание типов, константы и переменные, доступные в пользовательской программе (если имя модуля указано в операторе **USES**). В модуле **CRT** определена полезная переменная

VAR TextAttr : BYTE ,

в ней содержится текущий цвет фона и цвет символов, используемые при выводе на экран процедурами **WRITE** и **WRITELN**. Изменив эту переменную, вы задаете новый *цветовой атрибут*. Цветовой атрибут строится следующим образом : в четырех младших битах хранится цвет символов (от **0** до **15**), в следующих трех битах - цвет фона (от **0** до **7**), и старший бит отвечает за мерцание. Пусть, например, значение переменной **TextAttr** равно **237**, в двоичной записи - это **1110 1101** (если записывать биты от старшего к младшему).

Четыре младших бита (**1101**) дают цвет символов **13**, или **LightMagenta** - светло малиновый; следующие 3 бита (**110**) дают цвет фона **6**, или **Brown** - коричневый, старший бит - единичный. Таким образом, будут выводиться мерцающие светло-малиновые символы на коричневом фоне. Из сказанного ясно, что цветовые константы **Black** ... **White** определены в **CRT** точно так же, как и в модуле **Graph**. Кроме того, определена константа мерцания **Blink = 128**. Теперь построим нужный цветовой атрибут сами : мы хотим вывести желтые мерцающие символы на светло-сером фоне. Переменной **TextAttr** необходимо присвоить значение **14** (желтые символы) +**7** (серый фон) * **16** + **128** (мерцание), итого: 1**4+112+128=254**. Столь сложных вычислений легко избежать, если пользоваться 16-ричными числами; наш атрибут в 16-ричном виде записывается как **$7E+Blink**.Теперь рассмотрим некоторые функции и процедуры модуля **CRT** :

1. **FUNCTION KeyPressed : Boolean** - возвращает **TRUE**, если буфер клавиатуры не пуст (все нажатия клавиш во время работы программы накапливаются в специальном участке памяти - буфере клавиатуры, откуда затем поступают в программу). Функция не очищает буфер клавиатуры!

2. **FUNCTION ReadKey : Char** - считывает символ из буфера клавиатуры, если буфер пуст, то ожидает нажатия клавиши. Эту функцию удобно использовать для организации пауз в программе.

3. **PROCEDURE Delay(***MS***: Word)** - приостанавливает выполнение программы на *MS* миллисекунд.

4. **PROCEDURE Sound(***Hz***: Word)** - генерирует звуковой сигнал с частотой *Hz* герц.

5. **PROCEDURE NoSound** - выключает звуковой сигнал.

6. **PROCEDURE Window(***X1***,***Y1***,***X2***,***Y2***:Byte)** - определяет на экране текстовое окно, заданное координатами верхнего левого и нижнего правого углов. Текстовое окно - это прямоугольная область на экране, куда направляется весь вывод. Процедура не выполняет никаких *видимых* действий!

7. **PROCEDURE TextBackground(***Color***: Byte)** - задает цвет фона для всего последующего вывода.

8. **PROCEDURE TextColor(***Color***: Byte)** - задает цвет символов для всего последующего вывода. Процедуры **TextBackground** и **TextColor** вместе обеспечивают те же возможности, что и переменная **TextAttr**.

9. **PROCEDURE ClrScr** - очищает текущее окно, используя текущий фоновый цвет.

10. **PROCEDURE GotoXY(***X***,***Y***:Byte)** - перемещает курсор в позицию *X* строки *Y* текущего окна. Координаты отсчитываются от левого верхнего угла окна.

11. **FUNCTION WhereX : Byte** и

12. **FUNCTION WhereY : Byte** - возвращают текущие относительные координаты курсора (позицию и строку).

13. **PROCEDURE DelLine** - удаляет строку окна, в которой находится курсор, все нижние строки автоматически смещаются вверх.

14. **PROCEDURE InsLine** - вставляет пустую строку перед строкой, в которой находится курсор, все нижние строки автоматически смещаются вниз, и последняя строка окна теряется.

Теперь попробуем написать программу с приличным интерфейсом. Пусть программа загадает число, а пользователь должен его отгадать.

```
USES Crt;
VAR X,F : Byte;   S : STRING;   Code : Integer;  Yes  : Boolean;
BEGIN Window(1,1,80,25);  TextBackground(7);  ClrScr; {очистим экран}
        RANDOMIZE;  F:=RANDOM(10); {загадаем число}
        Window(26,11,54,13); TextAttr:=$20;  ClrScr; {распределим окно ввода}
        REPEAT
          GotoXY(3,2);  WRITE('Введите число от 0 до 9 '); READLN(S); VAL(S,X,Code);
          IF (Code=0)AND(X=F) THEN BEGIN  {правильное число}
             TextBackground(4);  ClrScr; TextColor(15);  GotoXY(8,2); WRITE('Вы угадали !!!');
             REPEAT UNTIL KeyPressed; HALT; END;
          {неправильное число} TextAttr:=$4F+Blink;  ClrScr;  Sound(500); GotoXY(11,2);
           WRITE('ошибка !'); Delay(500); NoSound;
           {запрос на повторение} TextAttr:=$20;  ClrScr; GotoXY(6,2);
           WRITE('Сдаетесь ? [Y/N]');  IF ReadKey IN ['Y','y'] THEN Halt;
        UNTIL FALSE;
END.
```

Вообще говоря, построение интерфейса не требует каких-либо специальных знаний. Главным образом следует руководствоваться здравым смыслом. Но некоторые простейшие правила можно привести:

- на экране не должно быть "мусора";
- пользователь в любой момент работы должен понимать, что от него требуется;
- неверный ввод данных не должен приводить к аварийному завершению программы;
- результаты работы должны выводиться в понятном и аккуратном виде.

Модуль CRT предоставляет некоторые средства, позволяющие строить довольно хороший интерфейс. Но намного более мощные и удобные средства содержатся в модуле **TpCrt** из библиотеки **Turbo Professional** для языка Паскаль. Эта библиотека не является частью системы программирования Turbo Pascal и из-за ограниченного объема данной книги здесь не описана. Однако автор рекомендует познакомиться хотя бы с некоторыми из ее возможностей, которые наверняка вам понравятся.

21. Модули. Создание и использование модулей

Наряду со стандартными модулями Turbo Pascal'я вы можете создавать и использовать в программах свои собственные модули. Модуль имеет следующую структуру:

UNIT *имя модуля*;
INTERFACE
интефейсная часть
IMPLEMENTATION
внутренняя часть
BEGIN
исполняемая часть
END.

Имя модуля - идентификатор длиной не более 8 символов, причем файл, содержащий модуль, должен иметь *такое же имя*. После ключевого слова **INTERFACE** располагаются описания типов, констант, переменных, функций и процедур, которые будут доступны пользователю этого модуля. Ключевое слово **IMPLEMENTATION** открывает внутреннюю часть модуля. В ней должны быть записаны *все* процедуры и функции, заголовки которых есть в интерфейсной части. Кроме того, во внутренней части можно описывать и другие переменные, константы, типы, функции и процедуры, но они уже будут недоступны другим программам. Исполняемая часть модуля может содержать обычные операторы, которые выполняются *до начала выполнения программы*, использующей этот модуль. Но как правило исполняемая часть пустая (а можно вообще не записывать слово **BEGIN**). Любой модуль может использовать другие модули как в интерфейсной, так и во внутренней частях, но следует избегать *перекрестных ссылок*, когда модуль ***А*** использует модуль ***Б***, а ***Б*** использует ***А***. Запишем небольшой модуль : пусть нам предстоит решать какие-нибудь задачи линейной алгебры, и мы хотим некоторые часто используемые алгоритмы объединить в модуль.

```
UNIT Algebra;
INTERFACE
      CONST Nmax=10;
          {полагаем,что наибольший размер матриц и векторов - 8}
      TYPE Vector = ARRAY[1..Nmax] OF Real; {тип для векторов}
      TYPE Matrix = ARRAY[1..Nmax] OF Vector; {тип для матриц}
      FUNCTION Scalar(n:Byte; a,b:Vector):Real; {скалярное произведение векторов длиной n}
      PROCEDURE SumMatrix(n,m:Byte; A,B:Matrix; VAR C:Matrix);{сумма матриц размера nxm}
      PROCEDURE MultMatrix(nA,mA,nB,mB:Byte; A,B:Matrix;  VAR C:Matrix);
                                                {произведение матриц размера nA x mA и nB x mB}
IMPLEMENTATION
      USES Crt;
      VAR i,j,k : Byte;
         {это ВНУТРЕННЯЯ процедура}
      PROCEDURE SumVector(n:Byte; a,b:Vector; VAR c:Vector);
      BEGIN  FOR i:=1 TO n DO c[i]:=a[i]+b[i];  END;
        {это ВНУТРЕННЯЯ процедура}
      PROCEDURE Peeek; BEGIN Sound(400); Delay(100); NoSound; END;
      FUNCTION Scalar;
      VAR S : Real;
      BEGIN S:=0; FOR i:=1 TO n DO S:=S+a[i]*b[i]; Scalar:=S; END;
      PROCEDURE SumMatrix;
      BEGIN FOR i:=1 TO n DO SumVector(m,A[i],B[i],C[i]); END;
      PROCEDURE MultMatrix;
      BEGIN IF mA<>nB THEN BEGIN Peeek; Exit; END;
                FOR i:=1 TO nA DO FOR j:=1 TO mB DO BEGIN
                      C[i,j]:=0; FOR k:=1 TO mA DO C[i,j]:=C[i,j]+A[i,k]*B[k,j];
                END;
       END;
BEGIN  {пустая исполняемая часть} END.
```

Если откомпилировать этот модуль, то будет создан файл с именем **ALGEBRA.TPU**, который впоследствии и будут использовать наши программы (расширение **TPU** означает **T**urbo **P**ascal **U**nit). Например:

```
 USES Algebra;
 VAR W,Q,Z : Matrix;
```

BEGIN MultMatrix(3,4,4,6,W,Q,Z); END.

Можно компилировать одновременно и программу, и все используемые в ней модули командой **F9** ("**Make**" в подменю "**Compile**"). При этом будут созданы и все **TPU**-файлы и **EXE**-файл. Не забудьте, что имя файла, в котором мы храним текст модуля **Algebra**, должно быть обязательно "**ALGEBRA.PAS**" !

22. Файлы

Паскаль-программа может работать с *внешними файлами*: читать из них информацию, записывать информацию в файл, корректировать файлы, создавать новые файлы, переименовывать и уничтожать существующие файлы. Различают три типа файлов: *текстовые*, *типизированные* и *бинарные*. Это различие влияет лишь на способы обращения к файлу, один и тот же файл на диске программа может рассматривать и как текстовый, и как типизированный, и как бинарный. Рассмотрим сначала текстовые файлы.

Для работы с текстовым файлом в программе следует описать *файловую переменную* типа **TEXT** :

VAR *f* **: TEXT;**

Прежде чем выполнять какие либо операции с файлом, необходимо *инициализировать* файл, т.е. установить связь файловой переменной с файлом на диске процедурой

1. **Assign(VAR *f*:TEXT**; *Name*:**String)** ,

где *Name* - правильно построенное имя файла, существующего или вновь создаваемого. После этого выполняется *открытие файла* одной из трех процедур:

2. **Reset(VAR *f*:TEXT)** - открывает файл для чтения.
3. **Rewrite(VAR *f*:TEXT)** - открывает файл для записи.
4. **Append(VAR *f*:TEXT)** - открывает файл для записи в конец файла.

Процедуры **Reset** и **Append** выполняются только для существующих файлов, процедура **Rewrite** - для любых файлов, но если файл существует, он будет уничтожен и создан заново. Чтение из файла и запись в файл выполняются процедурами **READ[LN]** и **WRITE[LN]**, но перед списком ввода или вывода задается файловая переменная:

5. **Read[Ln](VAR *f*:TEXT**; *список ввода*).
6. **Write[Ln](VAR *f*:TEXT**;*список вывода*).

Списки ввода и вывода строятся точно так же, как и в случае ввода с клавиатуры и вывода на экран. Особенностью текстовых файлов является то, что они состоят из *строк*, каждая из которых заканчивается *символом конца строки*. Процедура **WriteLn** записывает в файл этот символ, а процедура **Write** - нет. Вы можете сами управлять длинами записываемых строк, в нужный момент вызывая процедуру **WriteLn**. При вводе следует помнить, что если символ конца строки не считан процедурой **ReadLn**, то следующая строка недоступна. Как правило, текстовый файл используется для хранения строк или символов, но можно держать там и числа. Для текстовых файлов определены четыре логические функции:

7. **Function EOLN(VAR *f*:TEXT):Boolean** - возвращает **TRUE**, если при чтении достигнут конец строки.
8. **Function EOF(VAR *f*:TEXT):Boolean** - возвращает **TRUE**, если при чтении достигнут конец файла.
9. **Function SeekEOLN(VAR *f*:TEXT):Boolean** - возвращает **TRUE**, если в строке больше нет ничего, кроме пробелов.
10. **Function SeekEOF(VAR *f*:TEXT):Boolean** - возвращает **TRUE**, если в файле нет больше ничего, кроме пробелов.Функция **EOLN** пригодится вам, если вы читаете из текстового файла *символы*; функция **EOF** - если вы читаете *символы* или *строки*, а функции **SeekEOLN** и **SeekEOF** необходимы при вводе *чисел* из текстового файла. Функции **EOLN** и **SeekEOLN** также могут быть полезны при обычном вводе с клавиатуры. Приведем пример : пусть необходимо ввести некоторый массив натуральных чисел и некоторый массив символов. Известно, что в обоих массивах не более 100 элементов. Запишем программу, которой не нужно заранее знать, сколько элементов будет введено, это удобнее, чем сначала запрашивать количество элементов в массиве.

```
CONST Nmax=100;
VAR x : ARRAY[1..Nmax] OF Word;
        c : ARRAY[1..Nmax] OF Char;
        i : Byte;
CONST nX : Byte = 0;
            nC : Byte = 0;
BEGIN
     WRITELN('Введите числа');
     {вводим все числа, заканчивая клавишей Enter}
```

```
    WHILE NOT SeekEOLN DO BEGIN INC(nX); Read(x[nX]); END;
    {считываем конец строки, иначе не введутся символы}   READLN;
    WRITELN('Введите символы');  {вводим символы, заканчивая клавишей Enter}
    WHILE NOT EOLN DO BEGIN INC(nC); Read(c[nC]); END;
    WRITELN('Введено ',nX,' чисел :');
    FOR i:=1 TO nX DO WRITE(x[i]:8);  WRITELN;
    WRITELN('Введено ',nC,' символов :');
    FOR i:=1 TO nC DO WRITE(c[i]);  WRITELN;
END.
```

Вернемся к работе с файлами. Текстовый файл, открытый для чтения, можно *усекать* процедурой

11. **Procedure Truncate**(**VAR** *f*:**TEXT**).

Эта процедура уничтожает весь непрочтенный остаток файла. Файл закрывается процедурой.

12. **Procedure Close**(**VAR** *f*:**TEXT**),

после чего файловую переменную можно использовать для других целей. Любой файл можно удалить с диска процедурой.

13. **Procedure Erase**(**VAR** *f*).

Удаляемый файл должен быть инициализирован, но не открыт, или открыт, но затем закрыт. Запишем программу, которая читает текстовый файл и выводит его на экран:

```
VAR f : TEXT;  s : STRING;
CONST Name='test.pas';
BEGIN Assign(f,Name); Reset(f);
        WHILE NOT EOF(f) DO BEGIN  READLN(f,s); WRITELN(s);  END;
END.
```

Теперь выполним то же самое, не используя строку:

```
VAR f : TEXT;  c : Char;
CONST Name='test.pas';
BEGIN Assign(f,Name);  Reset(f);
        WHILE NOT EOF(f) DO BEGIN
            WHILE NOT EOLN(f) DO BEGIN  READ(f,c); WRITE(c); END;
            READLN(f); WRITELN;
        END;
END.
```

Если в этой программе опустить **READLN(f)** , то она *зациклится*. Прочтем из текстового файла числа (конечно, в таком файле должны быть записаны *только* числовые константы и пробелы):

```
VAR f : TEXT;  x : Real;
CONST Name='num.txt';
BEGIN Assign(f,Name); Reset(f);
        WHILE NOT SeekEOF(f) DO BEGIN  READ(f,x); WRITE(x:10); END;
END.
```

Числа в текстовом файле могут быть записаны как в одной, так и в нескольких строках и разделяться любым количеством пробелов.

Второй тип файлов - *типизированные* файлы. Файловая переменная описывается в этом случае как

FILE OF *тип* ,

где *тип* - любой тип, кроме файлового. Считается, что типизированный файл содержит некоторое количество *записей* одного и того же типа.Читать и записывать в такой файл можно только данные этого типа. В отличие от текстовых файлов типизированные допускают *прямой доступ*, т.е. вы можете записывать в любое место файла и читать из любого места файла. Процедура **Assign** применяется для типизированных файлов точно так же, как и для текстовых. Типизированный файл можно открыть процедурами **Reset** и **Rewrite** (процедура **Append** неприменима!), в обоих случаях файл доступен и для чтения, и для записи. Процедуру **Reset** следует применять для существующих файлов, а **Rewrite** - для новых файлов. Процедуры **Truncate** и **Close** работают точно так же, как и для текстовых файлов. Чтение и запись в типизированный файл осуществляется процедурами **READ** и **WRITE** (**READLN** и **WRITELN** не имеют смысла). Но в списках ввода и вывода можно записывать только переменные соответствующего типа. Функция **EOF** применима для типизированных файлов, а **EOLN** - нет. Прямой доступ к файлу осуществляется с помощью процедур **FileSize**, **FilePos** и **Seek**.

14. **PROCEDURE FileSize**(**VAR** *f*): **Longint** - возвращает текущий размер файла в записях, т.е. размер файла в байтах можно получить, умножив эту величину на размер одной записи.

15. **PROCEDURE FilePos**(**VAR** f): **Longint** - возвращает текущее значение *файлового указателя*. Файловый указатель хранит текущий адрес в файле, начиная с которого будет выполняться очередная операция ввода или вывода. При каждой операции ввода-вывода файловый указатель автоматически смещается на количество введенных или выведенных записей.

16. **PROCEDURE Seek**(**VAR** *f*; *n*:**Longint**); - устанавливает новое значение файлового указателя. Значение файлового указателя равно номеру последней обработанной записи, поэтому номер *текущей* записи будет равен *n*+1. Таким образом, чтобы установить указатель на первую запись, необходимо выполнить **Seek(f,0)**, а на последнюю - **Seek(f,FileSize(f)-1)**. Запишем программу, которая работает с типизированным файлом:

```
VAR  f  : FILE OF Real;  i,n  : Byte;  r : Real;
BEGIN Assign(f,'TMP');  Rewrite(f);
          Randomize;  n:=Random(100)+1; { количество записей в файле }
          FOR i:=1 TO n DO BEGIN r:=Sqrt(i); WRITE(f,r); END;
          WRITELN('Размер файла=',FileSize(f));
           i:=Random(n);  Seek(f,i);  READ(f,r);
          WRITELN('Запись номер ',FilePos(f),' содержит ',r);  Close(f);
END.
```

Третий тип файлов в Паскале - *бинарные файлы*. Бинарные файлы рассматриваются как последовательности *байтов* и могут содержать *любые данные*. Файловая переменная в этом случае должна иметь тип **FILE**. Бинарный файл открывается процедурами:

17. **PROCEDURE Reset**(**VAR** *f* : **FILE**; *RecSize*: **Word**);.

18. **PROCEDURE Rewrite**(**VAR** *f* : **FILE**; *RecSize*: **Word**);.

Второй параметр *RecSize* - это длина записи в байтах. Ввод-вывод в бинарный файл можно осуществлять порциями, кратными длине записи. Если при открытии файла задать длину записи в 1 байт, то такой файл сможет содержать любые данные. Для бинарных файлов также определены процедуры **Close**, **FileSize**, **FilePos** и **Seek**..Но чтение и запись осуществляются специальными процедурами:

19. **PROCEDURE BlockRead**(**VAR** *f*:**FILE**; **VAR** *Buf*; *Count*:**Word**
[;**VAR** *Res*:**Word**]);.

20. **PROCEDURE BlockWrite**(**VAR** *f*:**FILE**; **VAR** *Buf*; *Count*:**Word**
[;**VAR** *Res*:**Word**]);.

Здесь *Buf* - любая переменная, *Count* - количество вводимых или выводимых записей, *Res* - выходной параметр, возвращающий количество введенных или выведенных записей. Последний параметр необязателен, но в некоторых случаях его использование очень полезно. Запишем предыдущую программу, используя бинарный файл:

```
VAR f  : FILE;  i,n : Byte;  r : Real;
BEGIN Assign(f,'TMP'); Rewrite(f,SizeOf(REAL));
          Randomize;  n:=Random(100)+1; { количество записей в файле }
          FOR i:=1 TO n DO BEGIN r:=Sqrt(i); BlockWrite(f,r,1); END;
          WRITELN('Размер файла=',FileSize(f));
          i:=Random(n);  Seek(f,i);  BlockRead(f,r,1);
          WRITELN('Запись номер ',FilePos(f),' содержит ',r);  Close(f);
END.
```

В этой задаче мы не получили никаких преимуществ от использования бинарного файла, т.к. файл содержал однотипные данные - числа типа **Real**, но бинарные файлы могут содержать и данные разных типов вперемешку, тогда их использование очень эффективно. Решим еще одну задачу - быстро переписать содержимое одного файла (любого) в другой файл. Мы не будем интересоваться - что записано в файле, а лишь постараемся минимизировать количество обращений к файлам.

```
VAR Source,Target : FILE;
      Buf  : ARRAY[1..64000] OF Byte; {большой массив - буфер}
      Result       : Word;
CONST SourceName='TEST.PAS';
          TargetName='COPY.BIN';
BEGIN Assign(Source,SourceName);  Reset(Source,1);
          Assign(Target,TargetName);  Rewrite(Target,1);
          REPEAT
            {читаем из исходного файла как можно больше информации}
                        BlockRead(Source,Buf,SizeOf(Buf),Result);
```

```
        {записываем в новый файл столько, сколько удалось прочесть}
                BlockWrite(Target,Buf,Result);
        {прекращаем чтение-запись, когда прочитано меньше байт, чем  умещается в буфере, это
                значит, что в исходном файле больше ничего нет}
    UNTIL Result<SizeOf(Buf);
    WRITELN('OK...,см. файл "',TargetName,'"');
    Close(Source);   Close(Target);
END.
```

23. Модуль DOS и другие средства

Модуль **DOS** объединяет средства, позволяющие выполнять некоторые функции операционной системы. Мы изучим лишь часть из них.

1. **PROCEDURE GetDate(VAR** *Year*, *Month*, *Day*, *DayOfWeek*: **Word**) - возвращает текущую дату : год, номер месяца, число и номер дня недели. 0 соответствует воскресенью, 6 - субботе.

2. **PROCEDURE GetTime(VAR** *Hour*, *Minute*, *Second*, *Sec100*: **Word**) - возвращает текущее время : часы, минуты, секунды и сотые доли секунды.

3. **PROCEDURE FindFirst**(*Mask*: **String**; *Attr*: **Byte**; **VAR** *F*: **SearchRec**) - ищет в текущей или указанной директории первый файл, соответствующий заданной маске и атрибуту. Возвращает информацию о файле в переменной *F*. Маска может включать *путь* (если путь не задан, то поиск происходит в текущей директории) и должна содержать либо *имя файла*, либо *шаблон* (с использованием символа *). Параметр *Attr* может принимать одно из следующих значений:

ReadOnly =**$01** - файл только для чтения;
Hidden =**$02** - скрытый файл;
SysFile =**$04** - системный файл;
VolumeID =**$08** - заголовок тома;
Directory =**$10** - директория;
Archive =**$20** - архивный файл;
AnyFile =**$3F** - любой файл

либо быть равным какой-нибудь комбинации этих констант. Тип **SearchRec** определен в модуле **DOS** таким образом:

```
    TYPE SearchRec = RECORD
            Fill : array[1..21] of Byte; Attr : Byte; Time : Longint;
            Size : Longint; Name : STRING[12];  END;
```

Здесь **Attr** - атрибут файла, **Time** - время создания файла в упакованном виде, **Size** - размер файла в байтах, **Name** - имя файла, **Fill** - системное поле. Для распаковки времени создания файла служит процедура

4. **PROCEDURE UnpackTime**(*Time*: **Longint**; **VAR** *DT*: **DateTime**)

```
    TYPE DateTime = RECORD
                    Year,Month,Day,Hour,Min,Sec:Word; END;.
```

5. **PROCEDURE FindNext(VAR** *F*: **SearchRec**) - ищет следующий файл с атрибутами, заданными последним вызовом **FindFirst**. Процедуры **FindFirst** и **FindNext** возвращают через переменную.

6. **VAR DosError** : **Integer**

свой код завершения. Если значение этой переменной равно 0, процедура выполнилась успешно, в противном случае файл не был найден. Запишем программу, которая будет выводить на экран список файлов, имеющихся в корневой директории диска **C** и в текущей директории:

```
USES Crt,Dos;
PROCEDURE Find(Path:STRING; Attr:Byte);
VAR F  : SearchRec;  DT : DateTime;  c : CHAR;
BEGIN FindFirst(Path+'*.*',Attr,F);
    WHILE DosError=0 DO BEGIN
            UnpackTime(F.Time,DT);
            WITH F DO WRITELN(Name:12, Attr:4, DT.Year:5, DT.Month:3, DT.Day:3,
                                DT.Hour:3, DT.Min:3, DT.Sec:3,Size:7);
             FindNext(F);
    END;
    c:=ReadKey;
```

```
END;
BEGIN ClrScr;
        WRITELN('------------------ диск C ----------------------'); Find('C:\',Archive+Hidden);
        WRITELN('--------- текущая директория ------------'); Find('',Archive);
END.
```

Для поиска определенного файла на диске можно использовать функцию

7. **FUNCTION FSearch**(*Name*: **PathStr**; *DirList*: **STRING**): **PathStr .**

Здесь *Name* - имя файла возможно с добавлением пути (тип **PathStr** в модуле **DOS** определен как **STRING[79]**); *DirList* - список директорий, разделенных символами "**;**". Функция возвращает полное имя файла, если он найден, или пустую строку. Определить, существует ли на диске файл, можно и не используя средства модуля **DOS**. Открытие несуществующего файла для чтения приводит к ошибке, код которой возвращает стандартная функция

8. **FUNCTION IOResult**: **Integer .**

Если значение функции не ноль, то последняя операция ввода-вывода (к которым относится и открытие файла) не выполнилась. Чтобы предотвратить аварийное завершение программы, следует использовать *опцию компилятора* **{$I}** - контроль ввода-вывода:

```
VAR Name : STRING;   f : FILE;
BEGIN
        WRITE('Введите имя файла  '); READLN(Name);
        Assign(f,Name);
{$I-}
        Reset(f,1);
        IF IOResult<>0 THEN WRITELN('Файл не найден')
                        ELSE WRITELN('Файл найден !!!');
{$I+}
END.
```

Запись **{$I-}** означает “отключить контроль ввода-вывода”, а запись **{$I+}** - “включить контроль ввода-вывода”. В таком тривиальном примере включать опцию **I**, конечно, необязательно, но в любой реальной программе вы обязательно должны использовать эту опцию компилятора парами **{$I-}** и **{$I+}**. Еще две функции модуля **DOS** предназначены для получения информации о логических дисках:

9. **FUNCTION DiskSize**(*Disk*: **Byte**): **Longint** - возвращает размер логического диска в байтах, параметр *Disk* задает номер логического устройства: 0 - текущий диск, 1 - диск **A**, 2 - диск **B**, 3 - диск **C** и т.д. Если задан неверный номер устройства, функция возвращает значение **-1**, этим обстоятельством можно воспользоваться, чтобы определить наличие какого-либо диска в данном компьютере.

10. **FUNCTION DiskFree**(Disk: Byte): Longint - возвращает свободное пространство на диске в байтах.

Еще четыре процедуры, которыми заканчивается этот раздел, не входят в модуль **DOS**, но по своему назначению тесно связаны с этим модулем:

11. **PROCEDURE GetDir**(*Disk* : **Byte**; **VAR** *S*: **STRING**) - возвращает имя текущей директории на заданном диске.

12. **PROCEDURE ChDir**(*DirName* : **STRING**) - изменяет текущую директорию; для того, чтобы выйти во внешнюю директорию, нужно задать параметр **'..'**.

13. **PROCEDURE MkDir**(*DirName* : **STRING**) - создает в текущей директории поддиректорию.

14. **PROCEDURE RmDir**(*DirName* : **STRING**) - уничтожает пустую директорию.

24. Указатели и динамическая память

Указателями называются переменные и константы, значениями которых являются *адреса*. Различаются два вида указателей - *обобщенные* и *типизированные*. Обобщенные указатели имеют тип **POINTER** и могут содержать адреса любых объектов. Типизированные указатели имеют тип ^*базовый тип* и содержат только адреса объектов базового типа. Базовый тип может быть любым, кроме файлового. Существует одна константа-указатель **NIL**, равная некоторому *несуществующему* адресу. Указателям можно присваивать адреса переменных, для этого служит операция "*адрес*": **@***имя переменной*. Существует и обратная операция - "*значение*": *указатель*^ - результат этой операции есть значение, записанное по адресу, который содержит указатель. Операция "значение" неприменима к обобщенным указателям.Указателям можно присваивать адреса переменных соответствующего типа и другие указатели того же типа (или обобщенные указатели).

Обобщенному указателю можно присвоить любой указатель. Никакие арифметические операции к адресам не применимы. Запишем программу, выполняющую простейшие операции с указателями :

```
VAR  b  : Byte;   w  : Word;  pB : ^Byte;  pW : ^Word;
BEGIN pB:=@b;  pW:=@w;  READ(pB^,pW^);
        WRITELN(pB^,' ',pW^,' ',pW^ DIV pB^);
END.
```

Как видите, конструкция *указатель*^ может применяться точно так же, как и имя переменной соответствующего типа. Для работы с адресами существует четыре стандартных функции:

1. **FUNCTION Addr**(*X*): **Pointer** - возвращает адрес переменной *X*, по-существу, аналогична операции "адрес".

2. **FUNCTION Seg**(*X*): **Word** - возвращает сегментную часть адреса переменной *X*.

3. **FUNCTION Ofs**(*X*): **Word** - возвращает смещение адреса переменной *X*. (Полный адрес занимает 4 байта, значение первых двух называют *сегментом* а последних - *смещением*).

4. **FUNCTION Ptr**(*Seg*,*Ofs*: **Word**): **Pointer** - возвращает адрес, имеющий сегмент *Seg* и смещение *Ofs*.

Запишем пример использования этих функций:

```
CONST  L : LongInt = 123456789;
VAR    p : ARRAY [0..3] OF ^Byte;   i  : Byte;
BEGIN  FOR i:=0 TO 3 DO p[i]:=Ptr(Seg(L),Ofs(L)+i);
         WRITELN(p[0]^:4,p[1]^:4,p[2]^:4,p[3]^:4);
END.
```

Программа выведет байты **L** в последовательности от младшего к старшему:
21 205 91 7.

Приведенные примеры не очень содержательны, так как указатели главным образом используются для работы с *динамической памятью*. Динамическая память (или *хип*) - это область памяти, которую программа может использовать для размещения динамических переменных. В отличие от обычных (или статических) переменных, память под которые отводится компилятором до *начала выполнения программы* и освобождается после ее завершения, динамическая память распределяется и освобождается в процессе выполнения программы. Необходимость использования динамической памяти обусловлена, в частности, ограниченностью *сегмента данных*: все статические переменные, описанные в программе, не могут занимать более **64К** памяти. Динамическая память, как правило, имеет гораздо больший объем. Перечислим стандартные функции и процедуры для работы с хипом:

5. **FUNCTION MemAvail** : **LongInt** - возвращает размер свободной динамической памяти в байтах.

6. **FUNCTION MaxAvail** : **LongInt** - возвращает размер наибольшего свободного участка динамической памяти в байтах.

7. **PROCEDURE New**(**VAR** *P*:*указатель*) - отводит участок динамической памяти и присваивает указателю *P* адрес этого участка. Размер участка определяется базовым типом указателя.

8. **PROCEDURE Dispose**(**VAR** *P*:*указатель*) - освобождает участок динамической памяти, адрес которого хранится в указателе, после выполнения процедуры значение указателя не определено.

9. **PROCEDDURE Mark**(**VAR** *P*: **Pointer**) - записывает состояние динамической памяти в указатель *P*.

10. **PROCEDURE Release**(**VAR** *P*: **Pointer**) - возвращает динамическую память к состоянию, записанному в указателе **P**. Не следует использовать для освобождения памяти совместно **Dispose** и **Release**.

11. **PROCEDURE GetMem**(**VAR** *P*:**Pointer**; *Size*: **Word**) - распределяет участок динамической памяти размером *Size* байт и записывает его адрес в указатель *P*.

12. **PROCEDURE FreeMem**(**VAR** *P*:**Pointer**; *Size*: **Word**) - освобождает память, распределенную процедурой **GetMem**.

Приведем еще две процедуры, имеющие отношение к указателям:

13. **PROCEDURE Move**(**VAR** *Source*,*Dest*; *Count*: **Word**) - копирует *Count* байт из переменной *Source* в переменную *Dest*, причем можно использовать и имена переменных, и указатели с операцией "значение".

14. **PROCEDURE FillChar**(**VAR** *X*; *Count*:**Word**; *Value*) - заполняет *Count* байт переменной *X* значением *Value*. *Value* может быть либо типа **Byte**, либо типа **Char**.

Приведем пример использования последних двух процедур безотносительно к динамической памяти - пусть в программе описаны массивы:

```
A : ARRAY[1..100] OF Integer; B : ARRAY[1..1000] OF Integer;
```

требуется скопировать массив **A** в последние 100 элементов **B**, а остальные элементы **B** занулить:

```
FillChar(B,SizeOf(B),0);
Move(A,Ptr(Seg(B),Ofs(B)+900*SizeOf(Integer))^,SizeOf(A));
```

Теперь запишем программу, использующую массив, размещенный в динамической памяти:

```
CONST Nmax=10000;
TYPE Massiv = ARRAY[1..Nmax] OF Word;
VAR  p : ^Massiv;   i : Word;
BEGIN IF MaxAvail<SizeOf(Massiv) THEN BEGIN
            WRITELN('Не хватает памяти'); Halt; END;
        New(p);  Randomize;
        FOR i:=1 TO Nmax DO p^[i]:=Random(Nmax);
        { здесь могут быть различные действия с массивом }
        FOR i:=1 TO Nmax DO WRITE(p^[i]:5); DISPOSE(p);
END.
```

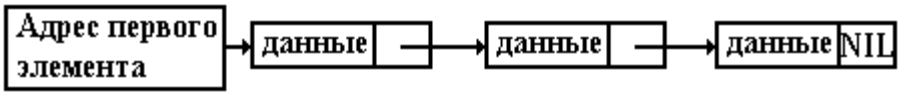
С динамическим массивом можно обращаться точно так же, как и с обычным, только *вместо имени* массива используется конструкция "**p^**".

Приведем пример использования процедур **GetMem** и **FreeMem**. Пусть в программе используется несколько (например, 5) больших массивов, причем в каждый момент времени в работе находится только один из них.

```
CONST N=6000;
VAR Massiv : ARRAY[1..N] OF Real;
VAR p : ARRAY[1..5] OF Pointer;   i : Byte;
BEGIN FOR i:=1 TO 5 DO BEGIN
                {<инициализация i-го массива>}
                IF MaxAvail<SizeOf(Massiv) THEN BEGIN
                   WRITELN('Не хватит памяти для ',i,'-го массива');
                   Halt; END;
                GetMem(p[i],SizeOf(Massiv));
                 Move(Massiv,p[i]^,SizeOf(Massiv));
    END;
    Move(p[2]^,Massiv,SizeOf(Massiv));
    {  работаем со вторым массивом .....  }
    {теперь "положим массив на место"}
    Move(Massiv,p[2]^,SizeOf(Massiv));
    { и так далее .... освободим память }
    FOR i:=1 TO 5 DO FreeMem(p[i],SizeOf(Massiv));
END.
```

25. Динамические структуры: списки, деревья

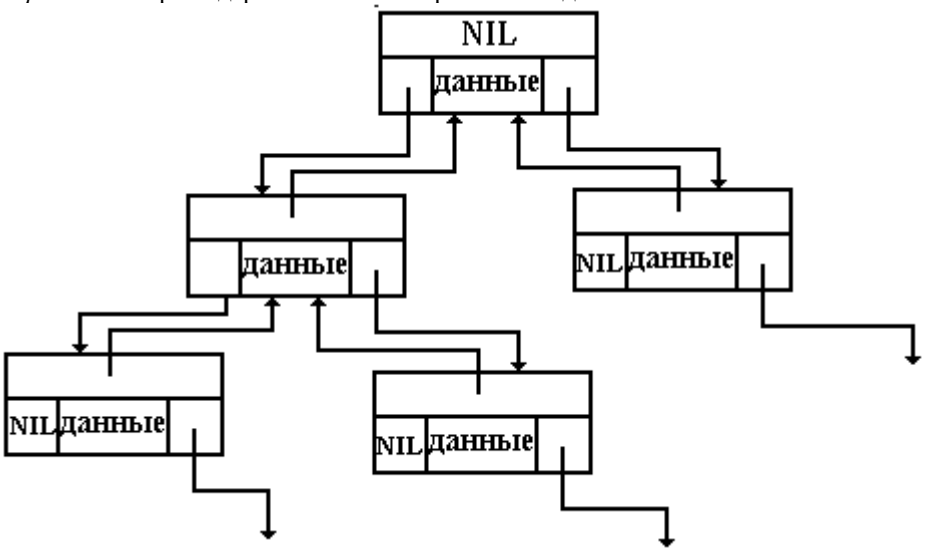
Примеры программ, приведенные в предыдущей главе, все еще были плохими. Для того, чтобы использовать динамические массивы таким образом, мы должны заранее знать *размеры* этих массивов. В реальных же задачах зачастую объем обрабатываемых данных заранее не известен. В этом случае удобно использовать динамические структуры данных, простейшей из которых является список. Список называется динамической структурой не только потому, что он размещается в динамической памяти, но главным образом потому, что его размер может меняться в процессе выполнения программы. Список не является объектом языка Паскаль и вообще никак не связан с конкретным языком программирования. Сама идея списка заключается в следующем : список есть совокупность однотипных *элементов*. Элементы размещаются в памяти произвольным образом, но в каждом из элементов хранится *адрес* следующего элемента. Таким образом, для обработки списка достаточно знать один единственный адрес - адрес *первого* элемента списка.Чтобы обозначить *последний* элемент, в нем в качестве адреса следующего элемента хранят константу **NIL**. Изобразим список:



Каждый элемент такого списка можно рассматривать как совокупность двух полей - поля данных, в котором содержатся собственно хранимая в списке информация (любого типа), и адресного поля, в котором записан адрес следующего элемента. Такой список называют *односвязным*, поскольку между парой элемен-

тов есть только одна связь. В некоторых случаях удобнее использовать двусвязный список, в каждом элементе которого хранятся адреса предыдущего и последующего элементов :

Деревом называют совокупность элементов, каждый из которых связан с одним элементом - предком и с несколькими элементами - потомками. Простейшее дерево - когда число потомков не превышает двух - называется *бинарным*. Бинарное дерево можно изобразить в виде:



Каждый элемент бинарного дерева содержит поле данных и три адресных поля. Рассмотрим теперь реализацию динамических структур в Паскаль-программе. Поскольку элементы будут размещаться в динамической памяти, *переменные* такого типа в программе не нужны, а нужен лишь *тип* элемента. В качестве типа элемента можно, очевидно, использовать только *запись*. Опишем сначала элемент односвязного списка:

```
TYPE  Adres   = ^Element;
      Element = RECORD Body : mun;  Next : Adres;  END;
```

В этом случае язык Паскаль допускает использование при описании типа **Adres** еще не описанного идентификатора **Element**. Но оба типа должны быть описаны в одном операторе **TYPE** ! Следующая запись неверна:

```
TYPE  Adres   = ^Element;
TYPE  Element = RECORD Body : mun;  Next : Adres; END;
```

Можно записать тип элемента списка и другим способом, используя нетипизированный указатель:

```
TYPE  Element = RECORD Body : mun; Next : POINTER; END;
```

Теперь мы легко сможем описать элементы двусвязного списка и бинарного дерева:

```
TYPE  Adres2   = ^Element2;
      Element2 = RECORD Body : mun; Previous,Next : Adres2; END;
TYPE  Adres_Tree   = ^Element_Tree;
      Element_Tree = RECORD Body: mun; Up,Left,Right: Adres_Tree; END;
```

Обработка динамических структур не составляет большого труда, единственное нетривиальное действие - это *создание* такой структуры. Рассмотрим алгоритм создания односвязного списка. Пусть список содержит вещественные числа, которые вводятся с клавиатуры:

```
TYPE  Adres   = ^Element;
      Element = RECORD Body : Real; Next : Adres; END;
VAR First,p : Adres;  Num : Real;
BEGIN First:=NIL;
   WHILE NOT SeekEOLN DO BEGIN
         READ(Num); NEW(p);  p^.Body:=Num;  p^.Next:=First; First:=p;
   END;
   { теперь выведем полученный список }
   p:=First;
   WHILE p<>NIL DO BEGIN  WRITELN(p^.Body); p:=p^.Next; END;
```

```
END.
```

Наша программа сформировала список, но числа записаны там *в обратном порядке*. В некоторых задачах это допустимо, в других, где порядок элементов важен, следует использовать, например, такой алгоритм:

```
TYPE  Adres   = ^Element;
        Element = RECORD Body : Real; Next : Adres; END;
VAR First,p,p0  : Adres;  Num : Real;
BEGIN First:=NIL;
        WHILE NOT SeekEOLN DO BEGIN
                READ(Num); NEW(p); p^.Body:=Num;
                IF First=NIL THEN First:=p ELSE p0^.Next:=p;
                p0:=p;
        END;
        p^.Next:=NIL;
        { теперь выведем полученный список }
        p:=First;
        WHILE p<>NIL DO BEGIN  WRITELN(p^.Body); p:=p^.Next; END;
END.
```

Выполним несколько простейших операций с нашим списком. Найдем сумму чисел, содержащихся в списке:

```
S:=0;  p:=First; WHILE p<>NIL DO BEGIN S:=S+p^.Body; p:=p^.Next; END;
```

Упорядочим список по неубыванию чисел:

```
p1:=First;
WHILE p1^.Next<>NIL DO BEGIN
    p2:=p1^.Next;
    WHILE p2<>NIL DO BEGIN
        IF p1^.Body>p2^.Body THEN BEGIN
            Num:=p1^.Body; p1^.Body:=p2^.Body;  p2^.Body:=Num;
        END;
        p2:=p2^.Next;
    END;
    p1:=p1^.Next;
END;
```

Найдем наименьший элемент списка и удалим его:

```
p:=First; Min:=1E38;
WHILE p<>NIL DO BEGIN
    IF p^.Body<Min THEN BEGIN Min:=p^.Body; MinAdr:=p; END;
    p:=p^.Next;
END;
p:=First; p0:=NIL;
WHILE p<>MinAdr DO BEGIN p0:=p; p:=p^.Next; END;
IF p=First THEN First:=p^.Next ELSE p0^.Next:=p^.Next;
DISPOSE(p);
```

Найдем наименьший элемент списка и вставим после него число 0:

```
p:=First; Min:=1E38;
WHILE p<>NIL DO BEGIN
    IF p^.Body<Min THEN BEGIN Min:=p^.Body; MinAdr:=p; END;
    p:=p^.Next;
END;
NEW(p); p^.Body:=0; p^.Next:=MinAdr^.Next; MinAdr^.Next:=p;
```

Уничтожим весь список:

```
p:=First;
WHILE p<>NIL DO BEGIN p0:=p^.Next; DISPOSE(p); p:=p0; END;
First:=NIL;
```

Операции с двусвязными списками, деревьями и другими сколь угодно сложными динамическими структурами выполняются аналогичным образом. Не отчаивайтесь, если вам поначалу будет трудно работать с динамическими структурами - большинство программистов прошло через это. Возьмите лист бумаги и

иллюстрируйте каждый оператор вашей программы, это поможет вам отчетливо понять структуру алгоритма.

26. Использование командной строки

Turbo Pascal позволяет передавать информацию в программу при ее запуске через командную строку. Для этого служат две стандартные функции - **ParamCount** и **ParamStr**.

FUNCTION ParamCount: **Word** - возвращает номер последнего заданного при запуске программы параметра. Параметры разделяются в командной строке пробелами.

FUNCTION ParamStr(*n*:**Word**): **String** - возвращает *n*-й параметр или пустую строку, если *n*>**ParamCount**. Параметры нумеруются, начиная с 0, причем 0-й параметр - это всегда имя выполняемой программы. Пусть программа была запущена из DOS командой **test.exe 1 abc** , тогда функция **ParamCount** вернет **2**, **ParamStr(0)='test.exe'**, **ParamStr(1)='1'**, **ParamStr(2)='abc'**, **ParamStr(3)=''**. При отладке программ, использующих командную строку, удобно пользоваться опцией **Parameters** подменю **Run**. Там вы можете задать все необходимые программе параметры (имя программы задавать не нужно) и отлаживать программу, не выходя в DOS. Напишем программу, которая будет складывать или вычитать два целых числа:

```
VAR a,b  : LongInt;   Code : Integer;   Plus : Boolean;
    BEGIN
            IF ParamCount<>3 THEN BEGIN
                WRITELN('test.exe <число> <+/-> <число>'); Halt; END;
            VAL(ParamStr(1),a,Code);
            IF Code<>0 THEN BEGIN
                WRITELN('1-е число задано неверно'); Halt; END;
            IF ParamStr(2)='+' THEN Plus:=TRUE  ELSE
            IF ParamStr(2)='-' THEN Plus:=FALSE
            ELSE BEGIN WRITELN('знак задан неверно'); Halt;  END;
            VAL(ParamStr(3),b,Code);
            IF Code<>0 THEN BEGIN
                 WRITELN('2-е число задано неверно'); Halt; END;
            IF Plus THEN WRITELN(a,'+',b,'=',a+b)
                       ELSE WRITELN(a,'-',b,'=',a-b);
    END.
```

27. Обработка программных прерываний

Программное прерывание - это ситуация, возникающая при выполнении программы, когда нормальное выполнение невозможно. Например: деление на ноль, переполнение, **Range Check Error**, обращение по неверному адресу, попытка открыть для чтения несуществующий файл и т.п. Обычная реакция программы на такие события - вывод сообщения об ошибке и аварийное завершение. Однако, пользуясь стандартными средствами Turbo Pascal'я, вы можете предусмотреть в своих программах и любую другую реакцию на прерывания. Для этого используются переменные **ExitProc**, **ExitCode**, **ErrorAdr** и процедура **RunError**.

Переменная - указатель **ExitProc** может содержать адрес процедуры, которая станет обрабатывать программные прерывания. Эта процедура не может иметь параметров и должна быть откомпилирована с опцией **{$F+}**. Если такая процедура предусмотрена в программе, ее адрес (с помощью операции **@**) нужно присвоить переменной **ExitProc**. После этого данная процедура будет вызываться всякий раз, когда произойдет программное прерывание. Следует учитывать, что нормальное завершение программы и остановка программы процедурой **Halt** также являются прерываниями, и, следовательно, процедура будет выполняться при любом завершении программы. Запишем пока тривиальную программу:

```
{$F+}
PROCEDURE MyExitProc;
BEGIN WRITELN('произошло прерывание !!!'); END;
{$F-}
 VAR x : REAL;
 BEGIN ExitProc:=@MyExitProc; WRITE('Введите число '); READ(x);
          WRITELN('это число в степени 33.3 равно ',Exp(33.3*Ln(x)));
 END.
```

Запустим программу и введем число **10**, программа выведет на экран результат, а затем сообщение "**произошло прерывание!!!**". Теперь введем число **20** - программа выведет сообщение "**это число в степени 33.3 равно произошло прерывание !!!**", затем сообщение об ошибке "**Runtime error 205 at 0000:00F9**", т.е. завершится все равно аварийно. Используем теперь переменные **ExitCode** и **ErrorAddr**, которые возвращают:

- *при нормальном завершении* **ExitCode** = 0 , **ErrorAddr** = **NIL**;
- *при выполнении процедуры* ***Halt*** **ExitCode** = аргументу **Halt** , **ErrorAddr** = **NIL**;
- *при аварийном завершении* **ExitCode** = коду ошибки, **ErrorAddr** = адресу прерывания.

Будем выводить сообщение только при аварийном завершении, т.е. когда **ErrorAddr** не равен **NIL**, и менять в этом случае значение **ErrorAddr**, чтобы предотвратить появление стандартного сообщения об ошибке:

```
PROCEDURE MyExitProc;
BEGIN IF ErrorAddr<>NIL THEN BEGIN
          WRITELN('произошло прерывание !!!'); ErrorAddr:=NIL; END;
END;
```

С такой процедурой наша программа будет выполняться обычным образом для "хороших" чисел, а для "плохих" станет сообщать: "*произошло прерывание*". Никаких других сообщений об ошибках не будет. Теперь немного усовершенствуем нашу процедуру, чтобы она сообщала, какая именно ошибка произошла:

```
PROCEDURE MyExitProc;
BEGIN IF ErrorAddr<>NIL THEN BEGIN
          WRITE('Произошло прерывание');
          CASE ExitCode OF
                106 : WRITELN(' (ошибка ввода)');
                207 : WRITELN(' (отрицательное число нельзя',
                              ' возводить в вещественную степень)');
                205 : WRITELN(' (слишком большое число)');
              ELSE  WRITELN(' (неизвестная ошибка)');
          END;
          ErrorAddr:=NIL;
      END;
 END;
```

Введем "**abc**" - программа сообщит "*ошибка ввода*"; введем "**-2**" - программа сообщит "*отрицательное число нельзя возводить в вещественную степень*"; введем "**100**" - программа сообщит "*слишком большое число*". Процедура

PROCEDURE RunError(*Errorcode*:**Byte**)

используется главным образом при отладке программ, она генерирует прерывание с заданным кодом. Например, для полной проверки нашей процедуры вставим в текст программы оператор **RunError(200);** - программа сообщит "*неизвестная ошибка*".

28. Параметры процедурных типов

Язык Паскаль допускает наличие у процедур и функций параметров, которые сами являются процедурами или функциями. Для того, чтобы записать такие параметры в заголовке процедуры или функции, они должны иметь некоторый *именованный* тип. Описание процедурного типа имеет вид:

TYPE *имя типа* = **PROCEDURE(***описание параметров***);**

Например,

```
TYPE MyProcType=PROCEDURE(VAR a:Word; b,c:Real; VAR tt:Char);
```

Таким образом, описание процедурного типа представляет собой заголовок процедуры без имени процедуры. *Имена параметров*, использованные в описании типа, могут быть произвольными (в нашем случае **a**,**b,c**,**tt** - совершенно случайные идентификаторы). Функциональный тип описывается следующим образом:

TYPE *имя типа* = **FUNCTION(***описание параметров***):***тип функции***;**

Например,

```
TYPE MyFuncType=FUNCTION(a:Real; b,c:LongInt):Boolean;
```

Не требуется никаких специальных действий, чтобы объявить какую-либо процедуру или функцию как имеющую соответствующий тип. Так, все функции типа **Boolean**, имеющие три параметра, первый из которых имеет тип **Real**, а два остальных - тип **LongInt**, автоматически будут иметь тип **MyFuncType**. Но

все процедуры и функции, использованные в программе как аргументы, должны быть откомпилированы с опцией **{$F+}** ! Попробуем записать программу, которая будет выводить таблицу значений некоторой функции:

```
TYPE FuncType=FUNCTION(Arg:Real):Real;
{$F+}
FUNCTION F_Cos(x:Real):Real; BEGIN F_Cos:=Cos(x); END;
FUNCTION F_Sin(x:Real):Real; BEGIN F_Sin:=Sin(x); END;
FUNCTION F_Tg(x:Real):Real;  BEGIN F_Tg:=Sin(x)/Cos(x); END;
{$F-}
PROCEDURE PrintFunc(Xmin,Xmax:Real; n:Word; F:FuncType; Header:STRING);
VAR h,x:Real;  i:Word;
BEGIN WRITELN; WRITELN(Header);  h:=(Xmax-Xmin)/n;
          FOR i:=0 TO n DO BEGIN
                x:=i*h+Xmin; WRITELN(x:10:5,F(x):10:5); END;
          WRITELN('Нажмите ENTER'); READLN;
END;
BEGIN  PrintFunc(0,Pi,20,F_Cos,'Значения функции cos');
           PrintFunc(0,Pi/2,20,F_Sin,'Значения функции sin');
           PrintFunc(0,Pi/4,10,F_Tg,'Значения функции tg');
END.
```

29. Описатель absolute. Нетипизированные параметры. Открытые массивы

В языке Паскаль определен описатель **absolute**, который можно использовать при описании переменных :

VAR *имя* **:** *тип* **absolute** *переменная* **;**

Здесь *имя* - имя описываемой переменной, *тип* - тип этой переменной, *переменная* - имя некоторой *ранее* описанной переменной, например:

```
VAR a:LongInt;   VAR b:Byte absolute a;
```

Смысл такого описания состоит в том, что переменная **b** будет размещена в памяти по тому же адресу, что и переменная **a**, и, изменяя переменную **a**, мы тем самым будем изменять **b**, и наоборот (вспомните о вариантных полях записей). Попробуем проделать какие-нибудь операции с нашими двумя переменными:

```
a:=257; WRITELN('a=',a,' b=',b);  b:=100; WRITELN('a=',a,' b=',b);
```

Программа выведет **a=257 b=1** и **a=356 b=100**. Фактически в этом примере мы можем работать либо с четырехбайтовой величиной **a**, либо с ее младшим байтом **b**.

Запишем еще один пример:

```
CONST s:STRING='abcdefgh'; VAR L:Byte absolute s;
BEGIN WRITELN('Длина s=',L); Dec(L,3);  WRITELN('Урезанная s : ',s);  END.
```

Программа выведет: **Длина s=8 Урезанная s : abcde** .Так можно работать с длиной строки, не используя функцию **Length** - переменная **L**, которая совмещена по памяти с 0-м символом строки, содержит текущую длину строки.

Теперь рассмотрим понятие "нетипизированные параметры". Язык Паскаль допускает использование *нетипизированных* параметров в процедурах и функциях, т.е. параметров, для которых не указывается тип (нетипизированный параметр обязательно должен быть параметром-переменной). Это означает, что в качестве аргумента такому параметру можно сопоставить переменную *любого типа*. Непосредственное использование таких параметров внутри процедур и функций невозможно, так как почти любая операция в языке Паскаль может осуществляться только над данными определенного типа. Но мы можем совместить по памяти с таким параметром какую-нибудь переменную (используя описатель **absolute**) и работать с этой переменной. Запишем в качестве примера функцию, вычисляющую наибольший элемент массива вещественных чисел любой длины:

```
FUNCTION Max(n:LongInt{длина массива}; VAR a):Real;
VAR Ar : ARRAY[1..10000] OF Real absolute a;  m : Real;  i : LongInt;
BEGIN m:=Ar[1];  FOR i:=2 TO n DO IF Ar[i]>m THEN m:=Ar[i];  Max:=m;
END;
```

Нетипизированные параметры используют, например, стандартные процедуры **BlockRead** и **BlockWrite**. В 7-й версии Turbo Pascal'я появилась возможность использования в качестве параметров процедур и функций *открытых массивов*. Открытый массив описывается в списке параметров в виде:

имя **: ARRAY OF** *тип элемента* ,

то есть не указывается тип индекса. Аргумент, сопоставляемый такому параметру, может быть любым массивом с элементами соответствующего типа. При этом нет необходимости передавать в процедуру или функцию границы индекса массива - для их определения служат стандартные функции:

FUNCTION Low(*x*) ,

FUNCTION High(*x*) .

Мы знаем, что аргументами этих функций могут быть имена порядковых типов, тогда функции возвратят наименьшее и наибольшее значение этого типа. Но функции **Low** и **High** могут также получать *имя обычного массива* или *имя открытого массива*. Для имени обычного массива функции возвращают наименьшее и наибольшее значения его индекса, например, если массив описан как **VAR r : ARRAY[-8..12] OF Real**, то **Low(r)** вернет **-8**, а **High(r)** вернет **12**. Для имени любого открытого массива (что наиболее важно для нас) функция **Low** всегда возвращает **0**, а функция **High** возвращает количество элементов массива без единицы. Те, кто знакомы с языком **C**, скажут, что именно так определяются индексы любого массива в **C**. Совершенно верно - открытые массивы введены в язык Паскаль, чтобы сблизить его с языком **C**. Решим нашу задачу о наибольшем элементе, используя открытый массив:

```
FUNCTION Max(VAR a:ARRAY OF Real):Real;
VAR i  : LongInt;  m  : Real;
BEGIN m:=a[0];  FOR i:=1 TO High(a) DO IF a[i]>m THEN m:=a[i];  Max:=m;
END;
```

30. Вызов внешних программ

В Паскаль-программе можно вызвать *внешнюю* программу, которая не обязательно должна быть написана на языке Паскаль. Для этого используется процедура **Exec** из модуля **DOS**:

PROCEDURE Exec(*Name*,*CmdLine*:**STRING**)

Процедура вызывает программу, которая содержится в файле *Name* (можно задавать полное имя). Этой программе передается командная строка *CmdLine*, таким образом можно передать информацию вызываемой программе. Если после вызова внешней программы основная программа будет продолжать работу, то необходимо вызвать процедуру

PROCEDURE SwapVectors

непосредственно *до* и непосредственно *после* процедуры **Exec**. **SwapVectors** сохраняет состояние программы в системной области, а затем восстанавливает это состояние.Переменная

VAR DosError: **Integer**

возвращает код завершения внешней программы; при нормальном завершении значение переменной равно 0. Запишем несложный пример использования процедуры **Exec**. Пусть существует внешняя программа, которая “пищит” и окрашивает экран в заданный цвет:

```
{ ТЕКСТ ВНЕШНЕЙ ПРОГРАММЫ }
USES Crt;
VAR Color : Byte;  Code  : Integer;
BEGIN IF ParamCount<>1 THEN Color:=4
        ELSE BEGIN Val(ParamStr(1),Color,Code);  IF Code<>0 THEN Color:=4;  END;
        WRITE(#7,#7,#7); Window(1,1,80,25); TextBackground(Color);   ClrScr;
END.
```

Откомпилируем эту программу, записав результат в файл **EXT_PRG.EXE**. Теперь запишем программу, которая вызовет **EXT_PRG.EXE** :

```
USES DOS;
BEGIN  SwapVectors; Exec('EXT_PRG.EXE','1'); SwapVectors;
            IF DosError=0 THEN WRITELN('OK')
            ELSE WRITELN('Ошибка номер ',DosError);
END.
```

Вполне возможно, что, запустив эту программу, мы получим сообщение "**ошибка номер 8**", этот код завершения означает "*не хватает памяти*". Дело в том, что процедура **Exec** пытается использовать память,

которую, возможно уже захватила основная программа. В этом случае следует уменьшить размер отводимой нашей главной программе памяти опцией компилятора **{$M}**. Синтаксис этой опции таков: **{$M** *размер стека***,** *минимальный размер хипа***,** *максимальный размер хипа***}**. Добавим в нашу основную программу строку **{$M 1024,0,0}** - хип в этой программе вообще не нужен, а размер стека в любом случае нельзя задать меньше, чем **1К**. Теперь наша программа отработает успешно.

31. Некоторые вычислительные алгоритмы

В этом разделе приводятся некоторые алгоритмы приближенных вычислений: метод простой итерации и метод Ньютона решения алгебраических уравнений; метод Гаусса и итерационные методы решения линейных систем; метод наименьших квадратов для аппроксимации таблично заданной функции; квадратурные формулы трапеций и Симпсона для приближенного вычисления определенных интегралов; метод Эйлера и метод Рунге-Кутта численного решения задачи Коши. Мы опустим обоснование этих численных методов, а приведем лишь общие схемы алгоритмов и примеры программ.

Приближенное решение алгебраических уравнений

В главе 17 уже был изложен один из алгоритмов численного решения алгебраических уравнений - метод бисекции. Этот алгоритм превосходно работает для любых уравнений, если заранее известен отрезок, на котором лежит единственный корень. В случае отсутствия такой априорной информации применяют другие численные методы, например, метод простой итерации или метод Ньютона.

Метод простой итерации определен для уравнения вида $x=\Phi(x)$, любое уравнение $F(x)=0$, очевидно, можно преобразовать к такому виду (однако не всякое такое преобразование будет подходящим для метода простой итерации). Алгоритм метода простой итерации крайне легок и заключается в следующем: выбирается некоторое начальное приближение X^0, затем вычисляются X^1,X^2 и так далее по формуле

$$X^{j+1} = \Phi (X^j) ; j=0,1,...$$

до тех пор, пока не выполнится условие $| X^{j+1} - X^j | < \varepsilon$, где ε - некоторое наперед заданное маленькое число, которое называется погрешностью, или точностью. Во многих случаях удается найти корень с абсолютной компьютерной точностью, т.е. для некоторого конечного j выполняется условие $X^{j+1} = X^j$; при этом не нужно задавать никакой точности, однако автор не гарантирует, что это справедливо для всех уравнений. Метод простой итерации сходится, т.е. за конечное число итераций дает значение корня с заданной точностью, при удачном выборе вида функции $\Phi(x)$ и нулевого приближения. Запишем программу, реализующую метод простой итерации:

```
TYPE FuncType = FUNCTION (x:Real):Real;
FUNCTION Fi(x:Real):Real; FAR;
{  Решаем уравнение  x^5  - x^2  - ln(2+x^2 )=0,   для метода простой итерации
                                                    преобразуем его к виду
                                                    x = (x^2+ln(2+x^2))^0.2 }
BEGIN Fi:=Exp(0.2*Ln(Sqr(x)+Ln(2+Sqr(x)))); END;
FUNCTION SimpleIteration(x0:Real; f:FuncType):Real;
{ Ищем корень с абсолютной точностью }
VAR x1 : Real;
BEGIN  x1:=x0;
          REPEAT x0:=x1;  x1:=f(x0); UNTIL x1=x0;
          SimpleIteration:=x1;
END;
CONST X0 = 1.0;
VAR Root : Real;
BEGIN Root:=SimpleIteration(X0,Fi);
          WRITELN('Корень уравнения x=Ф(x) равен ',Root);
          WRITELN('В этой точке x-Ф(x) равно ',Root-Fi(Root));
END.
```

Метод Ньютона применяется для уравнения $F(x)=0$ и записывается в виде:

$$X^{j+1} = X^j - \frac{F(X^j)}{F'(X^j)}; j = 1,2,...$$

Так же, как в методе простой итерации, задается начальное приближение X^0 и выполняются итерации либо до достижения абсолютной точности, либо до достижения наперед заданной погрешности. Метод Ньютона требует вычисления производной функции, но зато в случае отсутствия у уравнения кратных корней он гарантирует сходимость к одному из корней при любом начальном приближении.

```
TYPE FuncType = FUNCTION (x:Real):Real;
FUNCTION F(x:Real):Real; FAR;
{  Решаем уравнение  x^5 - x^2  - ln(2+x^2 )=0  }
VAR x2 : Real;
BEGIN x2:=Sqr(x); F:=x*Sqr(x2)-x2-Ln(2+x2); END;
FUNCTION dF(x:Real):Real; FAR;{ Производная функции F }
VAR x2 : Real;
BEGIN x2:=Sqr(x); dF:=5*Sqr(x2)-2*x*(1+1/(2+x2)); END;
FUNCTION Newton(x0:Real; f,df:FuncType):Real;
{ Ищем корень с абсолютной точностью }
VAR x1 : Real;
```

```
BEGIN x1:=x0;
        REPEAT x0:=x1; x1:=x0-f(x0)/df(x0); UNTIL x1=x0;
        Newton:=x1; END;
CONST X0 = 1.0;
VAR Root : Real;
BEGIN Root:=Newton(X0,F,dF);
    WRITELN('Корень уравнения F(x)=0 равен ',Root);
    WRITELN('В этой точке F(x) равна ',F(Root));
END.
```

Решение систем линейных алгебраических уравнений

Следующая группа вычислительных алгоритмов применяется для решения систем линейных алгебраических уравнений **A·X=B**. Для не слишком больших матриц применяют *точный* метод исключения Гаусса с выбором главного элемента. Система решается в два этапа: *прямой ход* приводит матрицу **A** к треугольному виду:

$$a_{ij} = a_{ij} - a_{kj} \cdot d_{ik}; b_i = b_i - b_k \cdot d_{ik}; d_{ik} = \frac{a_{ik}}{a_{kk}};$$

$$i = k+1, k+2, \ldots, n; j = k, k+1, \ldots, n; k = 1, \ldots, n.$$

В начале каждого шага по ***k*** стро́ки расширенной матрицы системы переставляются таким образом, чтобы выполнилось условие

$$|a_{kk}| = \max_{i = k\mathrm{K}\ n} \{|a_{ik}|\}.$$

Обратный ход дает искомый вектор **X** :

$$x_n = \frac{b_n}{a_{nn}}; x_i = \frac{1}{a_{ii}}\left(b_i - \sum_{j=i+1}^{n} a_{ij} \cdot x_j\right); i = n-1, n-2, \ldots, 1.$$

Несколько модифицированный метод Гаусса можно с успехом применять для вычисления определителя и обратной матрицы. Запишем программу, реализующую этот метод:

```
CONST Nmax=40;
{Максимальный размер матрицы, вы можете выбрать другое число}
TYPE VectorType = ARRAY[1..Nmax] OF Real;
     MatrixType = ARRAY[1..Nmax] OF VectorType;
FUNCTION Gauss( n : Byte {размер системы};  A : MatrixType {матрица системы};
      B : VectorType  {правые части};  VAR X : VectorType {вектор неизвестных}) : Boolean;
{ Функция будет возвращать TRUE, если систему удалось решить,  и FALSE, если матрица системы вырождена }
VAR i,j,k,iMax  : Byte;  tmp,Max,d : Real;  v  : VectorType;
BEGIN FOR k:=1 TO n-1 DO BEGIN { прямой ход }
      { ищем главный элемент }
        Max:=Abs(A[k,k]); iMax:=k;
        FOR i:=k+1 TO n DO
            IF Abs(A[i,k])>Max THEN BEGIN Max:=Abs(A[i,k]); iMax:=i; END;
        IF Max=0 THEN BEGIN { матрица вырождена }  Gauss:=FALSE; Exit; END;
        IF iMax<>k THEN BEGIN { переставляем строки }
          Tmp:=B[k]; B[k]:=B[iMax]; B[iMax]:=Tmp; v:=A[k]; A[k]:=A[iMax];  A[iMax]:=v; END;
        FOR i:=k+1 TO n DO BEGIN {вычитаем из i-ой строки k-ю }
            d:=A[i,k]/A[k,k];
            FOR j:=k TO n DO A[i,j]:=A[i,j]-d*A[k,j];
            B[i]:=B[i]-d*B[k]; END;
     END;
     { сейчас матрица системы - треугольная }
     IF A[n,n]=0 THEN BEGIN { матрица вырождена } Gauss:=FALSE; Exit; END;
     { обратный ход }
```

```
        X[n]:=B[n]/A[n,n];
        FOR i:=n-1 DOWNTO 1 DO BEGIN
              tmp:=B[i];
              FOR j:=i+1 TO n DO tmp:=tmp-A[i,j]*X[j];
              X[i]:=tmp/A[i,i];  END;
        Gauss:=TRUE;
END;
VAR n,i,j : Byte;  a  : MatrixType;   b,x   : VectorType;
BEGIN WRITE('Введите размер системы '); READ(n);
        WRITELN('Введите расширенную матрицу системы');
        FOR i:=1 TO n DO BEGIN
              FOR j:=1 TO n DO READ(a[i,j]); READ(b[i]); END;
        IF NOT Gauss(n,a,b,x) THEN BEGIN
           WRITELN('Матрица системы вырождена'); Halt; END;
        WRITELN('Решение системы и невязки :');
        FOR i:=1 TO n DO BEGIN
              FOR j:=1 TO n DO b[i]:=b[i]-a[i,j]*x[j];
              WRITELN(x[i]:12,' ',b[i]:12);  END;
END.
```

Хотя метод Гаусса и называют точным методом, невязки или погрешности, которые мы вывели в программе, не будут равны нулю. Но эти погрешности обусловлены не неточностью самого метода, а исключительно погрешностью вычисления арифметических операций.

Для очень больших систем, когда метод Гаусса становится неэффективным, применяют *итерационные методы*, например, метод простой итерации или метод Зейделя. Вычисления по *методу простой итерации* начинаются с произвольного вектора $\mathbf{X^0 = \{x_1^0, x_2^0, ..., x_n^0\}}$. Итерационный процесс осуществляется по формуле:

$$x_i^{j+1} = x_i^j + \frac{1}{a_{ii}} \cdot \left(b_i - \sum_{k=1}^{n} a_{ik} \cdot x_k^j \right); i = 1,2,\dots n; j = 0,1,\dots,$$

т.е. *все* неизвестные на следующей итерации вычисляются *только* через неизвестные на предыдущей итерации.

В *методе Зейделя* используется итерационная формула

$$x_i^{j+1} = x_i^j + \frac{1}{a_{ii}} \cdot \left(b_i - \sum_{k=1}^{i-1} a_{ik} \cdot x_k^{j+1} - \sum_{k=i}^{n} a_{ik} \cdot x_k^j \right); i = 1,2,\dots n; j = 0,1,\dots,$$

в которой при вычислении очередного неизвестного используются *последние* найденные значения остальных неизвестных. Вычисления заканчиваются, когда *невязки* системы становятся достаточно малыми. Итерационные методы сходятся не для всякой матрицы. Достаточным условием сходимости является *положительная определенность* матриц. Запишем программу, использующую метод простой итерации и метод Зейделя:

```
CONST Nmax=6;
TYPE VectorType = ARRAY[1..Nmax] OF Real;
       MatrixType = ARRAY[1..Nmax] OF VectorType;
PROCEDURE SimpleIteration(n:Byte; A:MatrixType;B:VectorType;Epsilon:Real; VAR X:VectorType);
VAR i,k : Byte;  X0 : VectorType;  s,Max,Tmp : Real;
BEGIN X0:=X;
        REPEAT
           FOR i:=1 TO n DO BEGIN s:=B[i];
                FOR k:=1 TO n DO s:=s-A[i,k]*X0[k];  X[i]:=X0[i]+s/A[i,i]; END;
         { Вычисляем максимальную невязку }
           Max:=0;
           FOR i:=1 TO n DO BEGIN Tmp:=Abs(X[i]-X0[i]);
                IF Max<Tmp THEN Max:=Tmp; END;
           X0:=X;
        UNTIL Max<Epsilon;
END;
PROCEDURE Zeidel (n:Byte; A:MatrixType; B:VectorType; Epsilon:Real; VAR X:VectorType);
```

```
VAR i,k : Byte;  s,Max,Tmp : Real;
BEGIN REPEAT Max:=0;
                    FOR i:=1 TO n DO BEGIN s:=B[i];
                        FOR k:=1 TO n DO s:=s-A[i,k]*X[k];
                        Tmp:=X[i];  X[i]:=X[i]+s/A[i,i];
                        Tmp:=Abs(X[i]-Tmp);
                        IF Max<Tmp THEN Max:=Tmp;
                    END;
         UNTIL Max<Epsilon;
END;
CONST a : MatrixType = ( (110,  3,  4, -5,  16,  20), (  1,  85, -6,  12,  11, -16), ( 24, -3,  54, -2,  0,  11),
                         ( -8,  9,  0,  75, -9,  12),  (  0,  2,  3, -5,  98,  34),  ( 16,  9, -4, -11,  0,  5));
          b  : VectorType = (100,200,300,400,500,600);
          x0 : VectorType = (0,0,0,0,0,0);
CONST Eps = 1E-8;
VAR n,i,j : Byte;  x : VectorType;  d : Real;
BEGIN
      n:=Nmax;  x:=x0;
      SimpleIteration(n,a,b,Eps,x);
      WRITELN('Решение системы методом простой итерации и невязки:');
      FOR i:=1 TO n DO BEGIN d:=b[i];
            FOR j:=1 TO n DO d:=d-a[i,j]*x[j];
            WRITELN(x[i]:12,' ',d:12); END;
       x:=x0;
       Zeidel(n,a,b,Eps,x);
       WRITELN('Решение системы методом Зейделя и невязки :');
       FOR i:=1 TO n DO BEGIN
             d:=b[i]; FOR j:=1 TO n DO d:=d-a[i,j]*x[j];
            WRITELN(x[i]:12,' ',d:12); END;
END.
```

Аппроксимация таблично заданной функции методом наименьших квадратов

Этот алгоритм часто используется в прикладных задачах, когда необходимо представить некоторый массив экспериментальных или наблюдательных данных какой-нибудь функциональной зависимостью. При этом вид аппроксимирующей функции задается заранее, а ее числовые коэффициенты подбираются таким образом, чтобы функция наилучшим образом аппроксимировала имеющиеся данные. Мы изучим способ аппроксимации алгебраическим полиномом. Пусть заданы массивы значений аргумента ***{x_j} j=1,...,m*** и функции ***{y_j} j=1,...,m*** и требуется аппроксимировать данную функцию алгебраическим полиномом степени ***n*** :

$$P(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \ldots a_n \cdot x^n .$$

Коэффициенты полинома ищутся из условия минимума функционала

$$\Delta = \sum_{j=1}^{m} \left(P(x_j) - y_j \right)^2 .$$

Каждая из величин $P(x_j)-y_j$ называется *невязкой* в точке x_j, а величина Δ, очевидно, есть сумма квадратов невязок. Таким образом, нашей задачей является выбор таких коэффициентов $a_0, a_1, \ldots, a_n$, при которых сумма квадратов невязок достигает наименьшего значения - отсюда и название метода. Очевидно, что функционал достигает минимума при выполнении условий:

$$\frac{\partial \Delta}{\partial a_0} \propto \sum_{j=1}^{m} \left(P(x_j) - y_j \right) = 0;$$

$$\frac{\partial \Delta}{\partial a_1} \propto \sum_{j=1}^{m} x_j \left(P(x_j) - y_j \right) = 0; \ldots; \frac{\partial \Delta}{\partial a_n} \propto \sum_{j=1}^{m} x_j^n \left(P(x_j) - y_j \right) = 0 .$$

Введем для удобства записи обозначения

$$X_k = \sum_{j=1}^{m} x_j^k \; ; \; Y_k = \sum_{j=1}^{m} x_j^k \cdot y_j$$

и перепишем систему уравнений в виде:

$$X_0 \cdot a_0 + X_1 \cdot a_1 + X_2 \cdot a_2 + \ldots + X_n \cdot a_n = Y_0$$
$$X_1 \cdot a_0 + X_2 \cdot a_1 + X_3 \cdot a_2 + \ldots + X_{n+1} \cdot a_n = Y_1$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$X_n \cdot a_0 + X_{n+1} \cdot a_1 + X_{n+2} \cdot a_2 + \ldots + X_{2n} \cdot a_n = Y_n .$$

Это система линейных алгебраических уравнений относительно коэффициентов a_j; ее решение можно найти одним из известных методов.

```
{ Аппроксимация методом наименьших квадратов }
CONST m=10; n=2;
TYPE VectorType = ARRAY[1..n+1] OF Real;
     MatrixType = ARRAY[1..n+1] OF VectorType;
     DataType   = ARRAY[1..m]   OF Real;
     XType      = ARRAY[0..2*n] OF Real;
     YType      = ARRAY[0..n]   OF Real;
{ Используем функцию Gauss из предыдущего пункта }
CONST x : DataType = (1,2, 3, 4, 5, 6, 7, 8, 9,  10);
      y : DataType = (4,9,16,25,36,49,64,81,100,121);
VAR i,j,k  : Byte;  XX : Xtype;  YY : Ytype;  A : MatrixType;  B,Coeff  : VectorType;  p : Real;
BEGIN  { Вычислим величины X и Y }
       FOR k:=0 TO 2*n DO XX[k]:=0;
       FOR i:=1 TO m DO BEGIN p:=1;  k:=0;
           WHILE k<=2*n DO BEGIN  XX[k]:=XX[k]+p; p:=p*x[i]; Inc(k); END;
       END;
       FOR k:=0 TO n DO YY[k]:=0;
       FOR i:=1 TO m DO BEGIN p:=y[i]; k:=0;
           WHILE k<=n DO BEGIN  YY[k]:=YY[k]+p; p:=p*x[i]; Inc(k); END;
       END;
   { сформируем матрицу системы }
     FOR i:=1 TO n+1 DO FOR j:=1 TO n+1 DO A[i,j]:=XX[i+j-2];
     FOR i:=1 TO n+1 DO B[i]:=YY[i-1];
   { вычислим коэффициенты полинома }
     IF Gauss(n+1,A,B,Coeff) THEN;
   { выведем результаты }
        WRITELN('Коэффициенты полинома :');
        FOR k:=1 TO n+1 DO WRITELN('a[',k-1,']=',Coeff[k]);
END.
```

Численное интегрирование

Существует большое количество методов приближенного вычисления определенных интегралов. Алгебраическая формула, дающая приближенное значение определенного интеграла, называется *квадратурной формулой*. Одной из простейших квадратурных формул является *формула трапеций*:

$$\int_a^b f(x)dx \approx h \cdot \left[\frac{1}{2} f(a) + f(a+h) + f(a+2h) + \ldots + \frac{1}{2} f(b) \right],$$

где $h = (b-a)/n$; n - некоторое целое положительное число, называемое числом разбиений отрезка $[a,b]$.

Квадратурная формула Симпсона записывается в виде:

$$\int_a^b f(x)dx \approx \frac{h}{3} \cdot \left[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + \ldots + 4f(b-h) + f(b) \right].$$

Число разбиений ***n*** для формулы Симпсона должно быть *четным*. Относительная погрешность вычисленного значения оценивается по формуле:

$$\delta = \frac{1}{r} \cdot \left| \frac{J_{2n} - J_n}{J_{2n}} \right|.$$

Здесь J_n и J_{2n} - значения, вычисленные для числа разбиений ***n*** и ***2n*** соответственно, ***r***=3 для квадратурной формулы трапеций и ***r***=12 для квадратурной формулы Гаусса. Чтобы начать вычисления, берут ***n***=1 или ***n***=2 и удваивают его до тех пор, пока относительная погрешность не станет меньше некоторого наперед заданного маленького числа.

Непосредственное использование квадратурных формул крайне неэффективно, поскольку приходится многократно повторять вычисление подынтегральной функции в тех точках, где она уже посчитана. Наиболее эффективный алгоритм вычислений по формуле трапеций можно записать в виде:

$$\int_a^b f\{x\}dx = J_\infty(b-a);\ J_1 = \frac{f_0 + f_1}{2};\ J_n = \frac{1}{2}J_{\frac{n}{2}} + \frac{1}{n}\sum_{i=1}^{n/2} f_{\frac{2i-1}{n}};\quad f_k = f(a + k(b-a)).$$

Для квадратурной формулы Симпсона наиболее эффективный алгоритм можно записать в виде:

$$\int_a^b f(x)dx = J_\infty(b-a);\ J_n = \frac{\Phi_n + S_n}{3n};\ \Phi_1 = f_0 + f_1;\ \Phi_n = \Phi_{\frac{n}{2}} + S_n;\ S_n = 2\sum_{j=1}^{n/2} f_{\frac{2i-1}{n}}.$$

Квадратурная формула Симпсона точнее, чем формула трапеций, и в большинстве случаев вычисление интеграла по формуле Симпсона происходит намного быстрее.

```
TYPE FuncType = FUNCTION (x:Real):Real;
FUNCTION Trapezia(a,b:Real; f:FuncType; Epsilon:Real):Real;
        { Квадратурная формула трапеций }
VAR i,n : LongInt;  J1,J2,d,h,s : Real;
BEGIN  J1:=(f(a)+f(b))/2;  n:=1;
        REPEAT n:=n ShL 1; h:=(b-a)/n; s:=0;
                FOR i:=1 TO n DIV 2 DO s:=s+f(a+(2*i-1)*h);
                J2:=J1/2+s/n;  d:=Abs(J1/J2-1)/3; J1:=J2;
        UNTIL d<Epsilon;
        Trapezia:=J2*(b-a);
END;
FUNCTION Simpson(a,b:Real; f:FuncType; Epsilon:Real):Real;
        { Квадратурная формула Симпсона }
VAR i,n : LongInt; J2,Jn,Q2,Qn,d,h,s : Real;
BEGIN  Q2:=f(a)+2*f((a+b)/2)+f(b); J2:=(Q2+2*f((a+b)/2))/6; n:=2;
        REPEAT n:=n ShL 1; h:=(b-a)/n; s:=0;
                FOR i:=1 TO n DIV 2 DO s:=s+f(a+(2*i-1)*h);
                s:=s*2; Qn:=Q2+s; Jn:=(Qn+s)/3/n;
                d:=Abs(J2/Jn-1)/12; J2:=Jn; Q2:=Qn;
        UNTIL d<Epsilon;
        Simpson:=Jn*(b-a);
END;
FUNCTION F(x:Real):Real; FAR; BEGIN F:=Sqr(Sqr(x)); END;
BEGIN WRITELN(Trapezia(1,2,F,1E-6));
        WRITELN(Simpson(1,2,F,1E-6));
END.
```

Численное решение задачи Коши

Задача Коши для численного решения ставится следующим образом: найти решение дифференциального уравнения ***y'=f(x,y)*** с краевым условием $y(x^0)=y^0$ в точке ***x=X*** (или на некотором конечном множестве точек). Будем строить свой вычислительный алгоритм так: разобьем отрезок ***[x°,X]*** на ***n*** равных частей точками $x_0, x_1, x_2, \ldots, x_n$, при этом, очевидно, $x_0=x^0$ и $x_n=X$. Обозначим $h=(X-x^0)/n$. Зная x_0 и $y(x_0)$ и пользуясь одним из известных численных методов, найдем $y(x_1)$; зная x_1 и $y(x_1)$, найдем $y(x_2)$ и так далее, пока не будет найден $y(x_n)$, равный (с некоторой погрешностью) искомой величине ***Y***. Обозначим это найденное значение Y_n - для ***n*** разбиений отрезка. Теперь увеличим число разбиений вдвое так же, как при численном интегри-

ровании, и найдем Y_{2n}. Оценив относительную погрешность $\delta = \left|\frac{Y_n - Y_{2n}}{Y_{2n}}\right|$, либо закончим вычисления, либо еще раз удвоим ***n***. Существует много численных методов, позволяющих по известным значениям ***x*** и ***y(x)*** найти ***y(x+h)***. Самый простой из них называется *методом Эйлера*:

$$y(x+h) = y(x) + f\left(x + h/2, y + k_1/2\right)\cdot h\,; \quad k_1 = f(x,y)\cdot h\,.$$

Более сложен (но и намного более точен) *метод Рунге-Кутта* четвертого порядка:

$$y(x+h) = y(x) + \frac{k_1 + 2k_2 + 2k_3 + k_4}{6}; \quad k_1 = f(x,y)\cdot h;$$

$$k_2 = f\left(x + h/2, y + k_1/2\right)\cdot h; \quad k_3 = f\left(x + h/2, y + k_2/2\right)\cdot h; \quad k_4 = f(x+h, y+k_3)\cdot h\,.$$

Запишем программу, реализующую эти два метода; обратите внимание на два последних оператора программы: примерно за одно и то же время метод Эйлера достигает лишь погрешности 1E-3, в то время как метод Рунге-Кутты дает 1E-5.

```
TYPE Func2Type = FUNCTION (x,y:Real):Real;
FUNCTION Euler(x0,y0:Real; XX:Real; f:Func2Type; Epsilon:Real):Real;
VAR n,i : LongInt;  h,x,y,Y1,Y2,k1,d : Real;
BEGIN  n:=1;  h:=XX-x0;  Y1:=y0+f(x0+h/2,y0+f(x0,y0)*h/2)*h;
          REPEAT   n:=n ShL 1;   h:=(XX-x0)/n;  y:=y0;
                     FOR i:=1 TO n DO BEGIN
                           x:=x0+(i-1)*h; k1:=f(x,y)*h; y:=y+f(x+h/2,y+k1/2)*h;  END;
                     Y2:=y;  d:=Abs(Y1/Y2-1);
          UNTIL d<Epsilon;
          Euler:=Y2;
END;

FUNCTION Runge_Kutta(x0,y0:Real; XX:Real; f:Func2Type; Epsilon:Real):Real;
VAR n,i  : LongInt;  h,x,y,Y1,Y2,k1,k2,k3,k4,d : Real;
BEGIN  n:=1;  h:=XX-x0;
          k1:=f(x0,y0)*h;  k2:=f(x0+h/2,y0+k1/2)*h; k3:=f(x0+h/2,y0+k2/2)*h;  k4:=f(XX,y0+k3)*h;
          Y1:=y0+(k1+2*(k2+k3)+k4)/6;
          REPEAT   n:=n ShL 1;  h:=(XX-x0)/n;  y:=y0;
                     FOR i:=1 TO n DO BEGIN  x:=x0+(i-1)*h;  k1:=f(x,y)*h; k2:=f(x+h/2,y+k1/2)*h;
                           k3:=f(x+h/2,y+k2/2)*h; k4:=f(x+h,y+k3)*h; y:=y+(k1+2*(k2+k3)+k4)/6;
                     END;
                    Y2:=y;  d:=Abs(Y1/Y2-1);
         UNTIL d<Epsilon;
          Runge_Kutta:=Y2;
END;

FUNCTION F(x,y:Real):Real; FAR; BEGIN F:=y/2/x+1/Sqrt(x); END;

BEGIN WRITELN;
          WRITELN(Euler(2,0.9802581,2.5,F,1E-3));
          WRITELN(Runge_Kutta(2,0.9802581,2.5,F,1E-5));
END.
```

32. Объекты

Объектом в языке Паскаль называется совокупность данных и процедур и функций, обрабатывающих эти даннные. Программирование с использованием объектов называется *объектно-ориентированным* программированием. Объектно-ориентированное программирование, как правило, используется для создания

очень сложных диалоговых программ. В простых задачах - таких, как в наших примерах, - использование объектов может показаться (на самом деле так оно и есть) искусственным приемом.

Данные в объекте называются *полями*, а процедуры и функции - *методами*. Объектный тип описывается в виде:

TYPE *имя типа*=**OBJECT** *описание полей описание методов* **END;**

Поля объектов описываются так же, как поля записей, а описание метода - это заголовок процедуры или функции. Сами методы располагаются в программе после описания объектного типа. В заголовке процедуры или функции указывается ее составное имя: *имя типа.имя метода*. Все поля объекта непосредственно доступны в каждом из его методов, их не нужно передавать через список параметров. В программе можно описать любое количество переменных объектного типа. Принято называть *объектом* - объектный тип, а переменную такого типа - *экземпляром* объекта. Структура объекта похожа на структуру записи, и для обращения к полям и методам объектов также используется либо составное имя:

имя экземпляра объекта **.** *имя поля/метода* ,

либо оператор **WITH** *имя экземпляра объекта*, внутри которого можно записывать простые имена полей и методов. Экземпляры одного и того же объекта можно присваивать друг другу.

ПРИМЕР 1. "Простой объект":

```
Type ObjT1=OBJECT x:Real; n:Word; Function Power:Real; END;

Function ObjT1.Power:Real;
VAR i : Word;  p : Real;
BEGIN p:=1; FOR i:=1 TO n DO p:=p*x;  Power:=p; END;

VAR O1_1,O1_2 : ObjT1;
BEGIN WITH O1_1 DO BEGIN  x:=2;  n:=4; END;
        WITH O1_2 DO BEGIN  x:=3;  n:=3; END;
        WRITELN(O1_2.Power-O1_1.Power:4:1);
END.
```

Программа выведет: **11.0**

Наиболее важным свойством объектов является механизм *наследования*. Объект может быть объявлен *потомком* ранее описанного объекта и унаследовать от объекта-родителя все его поля и методы. Объект потомок может также иметь собственные поля и методы, которых не было у объекта-родителя. Объект потомок описывается так:

OBJECT(*имя родителя***)**
описание новых полей
описание новых методов **END;**

Экземпляру объекта-родителя можно присваивать экземпляр объекта-потомка, но не наоборот.

ПРИМЕР 2. "Наследование":

```
Type ObjT1=...;
     ObjT2=OBJECT(ObjT1) y:Real; Function RealPower:Real; END;
     ObjT3=OBJECT(ObjT2) Function Power10:Real; END;

Function ObjT1.Power...;
Function ObjT2.RealPower:Real; BEGIN RealPower:=Exp(y*Ln(x)); END;
Function ObjT3.Power10:Real; BEGIN Power10:=Exp(y*Ln(10)); END;

VAR O1 : Objt1; O2 : ObjT2;  O3 : ObjT3;
BEGIN WITH O3 DO BEGIN x:=2; y:=1/3; n:=5;  END;  O2:=O3;  O1:=O3;
        WRITELN(O3.Power10-O2.RealPower+O1.Power:7:4);
END.
```

Программа выведет: **32.8945** = $10^{1/3}-2^{1/3}+2^5$

Объект-потомок может заменять методы объекта-родителя на собственные методы с теми же именами. Такое свойство объектов называется *полиморфизмом*.

ПРИМЕР 3. "Полифорфизм":

```
Type ObjT1=...;
     ObjT2=...;
     ObjT3=...;
     ObjT4=OBJECT(ObjT3) Function Power(t:INTEGER):REAL; END;
```

```
Function ObjT1.Power...
Function ObjT2.RealPower...
Function ObjT3.Power10...
Function ObjT4.Power(t:Integer):Real;
VAR i : Word;  p  : Real;  Minus : Boolean;
BEGIN Minus:=t<0;  p:=1;
          FOR i:=1 TO ABS(t) DO p:=p*x;
          IF Minus THEN Power:=1/p  ELSE Power:=p;
END;

VAR O4 : ObjT4;
BEGIN O4.x:=2; WITH O4 DO WRITELN(Power(2)-Power(-2):5:2); END.
```

Программа выведет : **3.75** = 2^2-2^{-2}

Некоторые из методов объекта могут быть *виртуальными*. При описании таких методов в объекте после заголовка процедуры или функции записывается конструкция **VIRTUAL;** Объект, содержащий хотя бы один виртуальный метод, должен иметь и специальный метод - *конструктор*. Конструктор полностью тождественен процедуре, но слово **Procedure** в нем заменяется на слово **Constructor**. Виртуальные методы присоединяются к объекту не на этапе компиляции, а только при вызове конструктора (при этом содержимое конструктора не имеет никакого значения, он может быть и пустым). Конструкторы не могут быть виртуальными. Невиртуальные методы называются *статическими*. Объекты-потомки могут заменять родительские виртуальные методы только виртуальными, а родительские статические методы - только статическими. Если объект-потомок заменяет родительский виртуальный метод своим, то у нового метода должен быть точно такой же список параметров, как и у родительского. На статические методы это правило не распространяется.

ПРИМЕР 4a. "Статические методы":

```
Type TA = OBJECT
             Procedure Out;  Function Message:String;  END;
     TB = OBJECT(TA)
             Function Message:String;  END;

Procedure TA.Out; BEGIN WRITELN(Message); END;
Function TA.Message:STRING; BEGIN Message:='TA'; END;
Function TB.Message:STRING; BEGIN Message:='TB'; END;

VAR A:TA; B:TB;
BEGIN A.Out; B.Out; WRITELN(B.Message); END.
```

Программа выведет : **TA TA TB**.

В Примере 4a метод **Out** родительского объекта **TA** собирается полностью; в частности, к нему подключается метод **Message** объекта **TA**. Замена метода **Message** в объекте **TB** уже никак не может повлиять на унаследованный этим объектом статический метод **Out**.

ПРИМЕР 4b. "Виртуальные методы":

```
Type TA = OBJECT
             Procedure Out;  Function Message:String; Virtual; Constructor Init;  END;
     TB = OBJECT(TA)
             Function Message:String; Virtual;  Constructor Init; END;

Procedure TA.Out; BEGIN WRITELN(Message); END;
Function TA.Message:STRING; BEGIN Message:='TA'; END;
Function TB.Message:STRING; BEGIN Message:='TB'; END;
Constructor TA.Init; BEGIN END;
Constructor TB.Init; BEGIN END;

VAR A : TA;  B : TB;
BEGIN  A.Init;  A.Out;  B.Init;  B.Out;  END.
```

Программа выведет : **TA TB**.

В примере 4b метод **Message** является виртуальным, это значит, что компилятор не подключает этот метод к процедуре **Out** до выполнения конструктора. Какая именно функция будет использоваться в качестве метода **Message**, после компиляции еще не известно. Выполнение оператора **A.Init** приводит к подстановке

вместо неопределенного виртуального метода **Message** конкретной функции **TA.Message**. Если виртуальных методов в объекте несколько, эта операция выполняется для каждого из них при вызове конструктора. Аналогично при вызове **B.Init** виртуальный метод **Message** в экземпляре объекта **B** будет заменен на функцию **TB.Message** всюду, где этот метод используется, в том числе и внутри метода **Out**. Использование в объектах виртуальных методов предполагает, что потомки данного объекта будут изменять эти методы. Если изменение метода не ожидается, то он объявляется статическим.

Экземпляры объектов можно размещать в динамической памяти. Для этого используется либо *процедура* **New** :

New(*указатель на объект*[**,***конструктор*]**);** ,

либо *функция* **New** :

*указатель на объект***:=New(***тип объекта*[**,***конструктор*]**);**

Конструктор обязательно указывается для объектов, имеющих виртуальные методы, и задается своим простым именем. В динамически размещаемых объектах можно использовать специальный метод - деструктор. Деструктор - это процедура, в которой ключевое слово **Procedure** заменяется словом **Destructor**. Динамически размещенный объект уничтожается процедурой

Dispose(<*указатель на объект*>[**,**<*деструктор*>]**);**

В деструкторе можно предусмотреть все необходимые действия по очистке памяти.

ПРИМЕР 5. "Динамические объекты":

```
Type MType=ARRAY[1..100] OF Word;
     MPtrType=^MType;
Type ObjType = OBJECT
                    n:Byte;
                    p:MPtrType;
                    Procedure Fill; Virtual;
                    Procedure Out; Virtual;
                    Constructor Make;
                    Destructor Crush;
                  END;
Type PObjType = ^ObjType;

Procedure ObjType.Fill; VAR i:Byte;
BEGIN FOR i:=1 TO n DO p^[i]:=Random(100); END;
Procedure ObjType.Out; VAR i:Byte;
BEGIN FOR i:=1 TO n DO Write(p^[i]:4); WriteLn; END;
Constructor ObjType.Make; BEGIN New(p); END;
Destructor ObjType.Crush; BEGIN Dispose(p); END;

VAR X,Y:PObjType;
BEGIN X:=New(PObjType,Make);
        With X^ DO BEGIN  n:=40;  Fill;   Out;   END;
        New(Y);
        With Y^ DO BEGIN  Make;  n:=20;  Fill;  Out;  Crush; END;
        Dispose(X,Crush);
        Dispose(Y);
END.
```